Maths 3468, Hilary 2012: further algorithms and numerical methods

Colm Ó Dúnlaing

April 23, 2012

Contents

1 Graphs and digraphs

(1.1) : Directed graphs, edges, and undirected graphs. A *directed graph* or *digraph* for short is a system comprising a finite set V of *nodes* and a set E of *directed edges*; each directed edge is an ordered pair (u, v) of distinct nodes. (Thus: 'self-loops' (u, u) do not occur in these structures.)

'Edge' means directed edge when we speak about digraphs. The *inverse* of an edge (u, v) is the edge (v, u).

An undirected graph, or *graph* for short, can be viewed either as a digraph in which the inverse of every (directed) edge is also an edge, or a structure like a graph except that the edges are *unordered* pairs $\{u, v\}$ of distinct nodes.

Remark. In a graph or digraph, for every pair $\{u, v\}$ of nodes, either they are distinct and joined by an edge, or they are not; if we allowed more than one edge joining the same two nodes, it would be called a multigraph; the single-edge version used to be called a *simple* (di-) graph.

Remark. I like to use the word 'node' when a graph is thought of combinatorially, as an abstract relation; but if it is thought of geometrically, as in a drawing of a graph, I prefer to call them *vertices*.

(1.2) Incident edges and degree. In a digraph the edge (u, v) is *incident* to u and v: it is an in-edge for v and an out-edge for u. It is an edge *(out) from* u *into* or *to* v. One calls the node u its *from-node* and v its *to-node*. The *indegree* of a node is the number of in-edges it has; its *outdegree* is the number of out-edges it has.

In an undirected graph, the edge $\{u, v\}$ is considered incident both to u and to v; the *degree* of a node is the number of edges incident to it. (The sum of node degrees in a graph is twice the number of edges).

(1.3) Node count n and edge count m. Given a digraph or graph G, we shall always use n and m to denote the number of nodes and edges, respectively, which G possesses. The 'size' of the graph is taken to be $m + n$. Many of our algorithms will run in time $O(m + n)$, which we consider to be linear time.

(1.4) Representations. There are two common ways of realising digraphs as data-structures. The simplest is by *adjacency matrix* A for the edges. Suppose that the nodes are indexed from 1 to n; then A is an $n \times n$ binary-valued matrix matrix. If (i, j) is an edge then $A(i, j) = 1$, otherwise $A(i, j) = 0$. A digraph can have up to $n(n-1)$ edges, but often it has far fewer than this number, and the *adjacency list* realisation is preferred.

(1.5) Forward adjacency lists. Usually this representation means that the out-edges incident to every node are stored in a list, called an out-list or *forward adjacency list*. Alternatively, this list may contain not the edges (u, v) but their to-nodes v.

(1.6) Backward adjacency lists. Sometimes the in-lists are stored with the nodes (backward adjacency lists), or both in- and out-lists are stored.

(1.7) Paths and cycles. Let G be a digraph. A *path* in G is a sequence

$$u_0 e_1 u_1 e_2 \dots u_k$$

of nodes and edges in G where $e_j = (u_{j-1}, u_j)$. Obviously, a path is defined by the sequence of nodes $u_0 \dots u_k$ of nodes. The definition given is useful in more general graphs (where the same pair of nodes can be joined by several edges).

The *length* of the path is k (which can be zero), and the path is *from* u_0 *to* u_k. Paths of length 0 are called *trivial*. If no node occurs more than once in the sequence, the path is called *simple*.

A *cycle* is a path whose first and last nodes are the same. A cycle is *simple* if, apart from the first node occurring twice, no node occurs more that once in the sequence.

(1.8) If $G = (V, E)$ and $G' = (V', E')$ are directed or undirected graphs, then G' is a *subgraph* of G if $V' \subseteq V$ and $E' \subseteq E$.

1.1 Topological sort and acyclic graphs.

A digraph is *acyclic* if it has no nontrivial cycles. A *topological order* on the nodes of a digraph G is a sequencing $u_1, \dots, u_n$ of its nodes with the property that if (u_i, u_j) is an edge, then $i < j$.

(1.9) Lemma *If G' is a subgraph of an acyclic digraph G then G' is acyclic (trivial).* ∎

(1.10) Lemma *If a digraph can be topologically sorted then it is acyclic.*

Proof. Equivalently, if a nontrivial cycle exists then no sequencing of the nodes can be a topological order. For suppose $u_1, \dots u_n$ is any sequencing of the nodes in a digraph which possesses a nontrivial cycle. Let u_i be the node of lowest rank (topological order) in the cycle. The cycle can be presented as a path in which u_i occurs first and last. Let u_j be the second-last node: then (u_j, u_i) is an edge and $j > i$, so the sequence is not a topological order. ∎

Conversely we present an algorithm to topologically sort an acyclic digraph. The algorithm is based on a characterisation of acyclic digraphs (Lemma 1.13).

(1.11) Definition *If G is a digraph and S a subset of nodes of G, then $G \backslash S$ is the digraph obtained by deleting the nodes in S and all edges with one or both ends in S.*

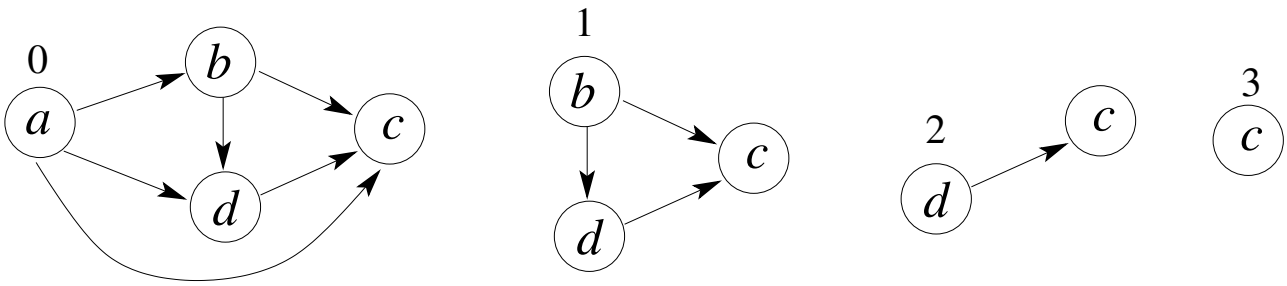

Figure 1: topological sort

Formally, if $G = (V, E)$, then $G\backslash S = (V\backslash S, E')$, where

$$E' \quad = \quad \{(u, v) \in E \colon u, v \notin S\}.$$

(1.12) Definition *A* source *in a digraph is a node with indegree zero.*

(1.13) Lemma *A digraph G is acyclic iff either* (i) *it is empty, or* (ii) *it has a source u and the digraph $G\backslash\{u\}$ is acyclic.*

Proof. Obviously an empty digraph is acyclic. Suppose G is nonempty.

If G is acyclic with n nodes, then every path in G has length $< n$ (a longer path would contain a nontrivial cycle). So G contains a longest path, beginning at u, say. Then u must be a source: otherwise the path could be extended.

Again. $G\backslash\{u\}$ must be acyclic, since it is a subgraph of an acyclic digraph. Hence condition (ii) is necessary.

Conversely, suppose that condition (ii) holds. Then G must be acyclic: if it contains a nontrivial cycle, then that cycle must include u, since otherwise it would be a cycle in $G\backslash\{u\}$. This is impossible since u is a source. ∎

(1.14) Corollary *A directed graph G is acyclic iff it admits a topological order.*

Proof. The 'if' part is Lemma 1.10.

Only if. By induction on n (no. of nodes). If $n = 1$ then there is only one node sequence, trivially a topological order. Assume true for $n - 1$. Suppose $|G| = n > 1$. Since G is acyclic and nonempty, it has a source w, and $G\backslash w$ is acyclic. By induction it admits a topological order: write it as $\{u_2, \ldots, u_n\}$. Let $w = u_1$, so we have a sequence

$$u_1, u_2, \ldots, u_n$$

By construction, there exists no directed edge (u_j, u_i) with $j > i \geq 2$. Nor can there exist a directed edge (u_j, u_1), since $u_1 = w$ is a source. Hence we have a topological order on G. ∎

This theorem is the basis of a simple and efficient algorithm: see Figure 1. Suppose, as is usually the case, that G is represented by forward adjacency lists (1.5). In place of backward adjacency lists it is enough to maintain an indegree function, giving for each node v the number of edges into v in the remaining digraph (maybe after other nodes have been deleted). The sources in the deleted graph

are those of indegree zero.

Remark. Code is given in (pseudo-) Eiffel, not C. The main difference is that arrays start at 1 (actually, the range can be chosen arbitrarily), so 0 means 'not there.' In C, -1 means 'not there.'

```
from
  initialise the indegree function, and a set Q
  containing all sources

  j := 0
until
  Q is empty
loop
  select a node u in Q
  and remove from Q

  for all edges (u,v) out from u,
  decrement indegree (v), adding
  v to $Q$ if its indegree is zero.

  set topological_rank ( u ) := j
  j := j + 1
end
  if  j < n then
    output "Graph contains a cycle."
```

(1.15) With the adjacency list representation of a digraph the above skeleton code can be implemented in time $O(m+n)$ — linear time.

(1.16) Theorem *If the graph is acyclic, the above algorithm generates a topological order. Otherwise the algorithm terminates with $j < n$. (As usual, n is the number of nodes in the digraph.)*

Proof. Whenever the algorithm processes a node u, for all edges (w, u) of the (original) digraph, w was processed before u. Hence if $j = n$ at the end, we have a topological order of the digraph.

If $j < n$ at the end then, as in Lemma 1.13, the graph is not acyclic. ∎

(1.17) Depth-first search (DFS) of a digraph or graph means processing the nodes in a depth-first order, so-called. It is easiest to describe as a recursive procedure. Below is exhibited a version which assigns to each node a 'parent' link to the unique node from which it was (first) visited; this link is part of a tree structure; the routine also installs preorder and postorder ranks for the nodes as they are visited. These ranks correspond to preorder and postorder in the tree constructed.

The variables `pre_count, post_count` are 'global variables,' initially -1. The 'pre_rank' field is initially -1 in all nodes. 'Parent' is initially null.

```
void dfs ( NODE * u )
{
    u -> pre_rank = pre_count;
    ++ pre_count;

    for all out-edges (u,v) -- with from-node u
      if ( v -> pre_rank < 0 )
      {
        v -> parent = u;
        dfs ( v );
      }
    u -> post_rank = post_count;
    ++ post_count;
}

void full_dfs ()
{
    for all nodes u
      if ( u -> pre_rank < 0 )
        dfs ( u );
}
```

(1.18) 3 states of a node. During a full DFS of a graph, all the nodes are in one of three *states: unseen, active, finished.*

- u is unseen when dfs(u) has not begun.
- u is active when dfs(u) has begun but not finished. Several nodes can be active simultaneously, since dfs is used recursively.
- u is finished when dfs(u) has terminated.

(1.19) Lemma *The states of a node u can be characterised as follows:*

- *u is unseen iff* u . pre_rank $= 0$
- *u is active iff* u . pre_rank $\neq 0$ *and* u . post_rank $= 0$
- *u is finished iff* u . post_rank $\neq 0$ *(Proof omitted.)*

(1.20) Full dfs is linear time. Full DFS creates a 'depth-first spanning forest' of the digraph (implicit in the 'parent' links).

We analyse the runtime as follows. For each call to dfs(u) let k be the out-degree of u. Charge $1 + k$ units for the work it does, excluding recursive calls to dfs. Adding these charges for all nodes gives the overall cost of the full dfs, which is therefore $O(m + n)$ — linear time.

(1.21) Crucial DFS reachability properties. We say a node v is *reachable* from a node u in a digraph G if there exists a path from u to v in G.

The following lemma gives a property of the preorder ranking and the DFS forest structure, a property very useful for justifying algorithms based on DFS. No proof of the lemma is attempted. (But note that after completion of full dfs, a node u was visited before v if and only if its preorder rank precedes that of v.)

(1.22) Lemma (DFS reachability property). *If V is the set of all nodes in G, $u \in V$, and W is the set of nodes which have been visited prior to invocation of dfs(u), then upon completion of dfs(u) the descendants of u in the dfs forest are precisely those nodes reachable from u in $G \backslash W$.*

Therefore if v is a descendant of u then v is reachable from u in G.[1] ∎

No proof of this lemma is supplied; but we may take it as the basic property of DFS. In terms of the ranks assigned to the nodes it can be made more explicit:

(1.23) Lemma *Following full DFS, a node v is descendant of u in the forest iff either of the following equivalent conditions hold:*

(i) pre_rank(u) $\leq$ pre_rank(v)*, and* post_rank(u) $\leq$ post_rank(v);

(ii) pre_rank(u) $\leq$ pre_rank(v) $\leq R$,
where R is the maximal preorder rank of all descendants of u.

Sketch proof. u is an ancestor of v iff $u = v$ or dfs(u) begins before dfs(v) and ends after dfs(v) ends. Equivalently, u precedes v in preorder and follows v in postorder, since preorder and postorder ranks are computed at beginning and end of dfs(u)... ∎

An immediate application of DFS is for topological sort of an *acyclic* digraph.

(1.24) Lemma *After full dfs of an acyclic digraph G, postorder rank gives a* reverse *topological order.*

Proof. Suppose that DFS has been carried out in a digraph, but reverse postorder ranking is not a topological order. Then there exists an edge (u, v) of the digraph, where u precedes v in postorder.

If u follows v in preorder, then u is a descendant of v in the forest (Lemma 1.23), so u is reachable from v and G is not acyclic.

If u precedes v in preorder, then v should be a descendant of u by Lemma 1.22, so u should follow v in postorder, a contradiction. **Q.E.D.**

1.2 Strong connectivity

A development of the above result is concerned with what might be considered the opposite of acyclicity: strong connectedness.

[1] But not vice-versa.

(1.25) Definition *In a digraph G, nodes u and v are* strongly connected *if both v is reachable from u and u is reachable from v.*

Equivalently, there exists a (directed) cycle containing both u and v.

The graph G is strongly connected if every two nodes are strongly connected.

A strongly connected component, or SCC *is a subgraph whose nodes are an equivalence class K under this relation and whose edges are $\{(u, v) \in E : \ u, v \in K\}$.*

(One says that it is the subgraph spanned by K*.)*

The next lemma is a kind of tongue-twister.

(1.26) Lemma *SCCs are strongly connected.*

Proof. Let K be an SCC and suppose u, v are nodes in K. There exists a directed cycle C containing u and v. But all nodes in C are strongly connected. Therefore $C \subseteq K$. Thus K is strongly connected. ∎

(1.27) Definition *The* root *of a SCC K, following DFS, is the earliest node visited: thus all nodes in K are descendants of the root.*

For any node u, write scc_root (u) *for the root of the SCC containing u.*

(1.28) Lemma *Suppose a digraph G is subjected to DFS. Let K be a strongly connected component, and suppose u is its root. Then all other nodes in K are descendants of u.*

Proof. Let S be the set of nodes visited earlier than u (preceding u in preorder). Since K is strongly connected, every node in K is reachable from u by a path within K. By Lemma 1.22, every node reachable from u in $G \backslash S$ is a descendant of u. Since K is a subgraph of $G \backslash S$ the result follows. ∎

(1.29) Lemma *Suppose DFS is applied to a digraph G. Let (u, v) be an edge of G. Then either u is an ancestor or descendant of v, or v precedes u both in preorder and postorder.*

Proof. Suppose v does not precede u in preorder, so u was visited before v. Then v would be a descendant of u by Lemma 1.22.

So if u is not an ancestor of v, v must precede u in preorder, so either by Lemma 1.23) v is an ancestor of u, or v precedes u in postorder also. **Q.E.D.**

(1.30) Following the above lemma, we can classify the edges after dfs in four different ways, as shown in the table in Figure 2. An edge from node to descendant is a *tree edge* if from parent to child, otherwise a *forward edge.* An edge from node to ancestor is a *back edge*. An edge (u, v) where u is visited before v but is not an ancestor of v is a *cross edge*.

$u < v$ (preorder)?	$u < v$ (postorder)?	
yes	yes	impossible
yes	no	forward or tree
no	yes	back
no	no	cross

Figure 2: classifying edges (u, v) after DFS of digraph

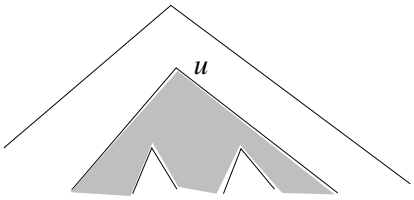

Figure 3: prefix subtree (shaded).

1.3 SCCs and DFS

(1.31) Lemma *Let $u, v \in K$ where K is a SCC of G: then there exists a directed cycle* entirely in K *which contains u and v.*

Indeed, any path from u to v is entirely within K.

Proof. Certainly there exist paths from u to v and v to u, which combine to form a directed cycle C in G containing u and v. But being a cycle, all nodes in C are strongly connected, so $C \subseteq K$.

If P' is another path from u to v, it too can be completed to a cycle and every node in P' is in K. ∎

(1.32) Definition *Suppose G is given a full DFS. If K is a scc, then its* first node*, first(K) is the earliest node in K visited during the dfs.*

(1.33) Lemma *Every node in K is a descendant (in the DFS tree) of its first node.*

Proof. Let u be the first node in K and let S be the nodes in G visited before u. Every node in K is reachable from u by a path in K, hence a path in $G \backslash S$. By Lemma 1.22, every node in K is descended from u. ∎

(1.34) Definition *Let T be any tree.*

(i) *For any subset W of the nodes of T, the* forest spannned by W *is the forest F whose nodes are W and whose parents function is inherited from T. In other words, if x has parent y in T, and $x \in W$, then either* (a) $y \notin W$ *and x is a root in F, or* (b) $y \in W$ *and y is the parent of x in F.*

(ii) *A* prefix *of T is a tree S whose nodes W are nodes of T, where W is closed with respect to ancestorhood, i.e., if $x \in W$ then every ancestor of x (in T) is in W, and S is spanned by W.*

A prefix subtree *of a tree T (Figure 3) is a nonempty prefix of some subtree T_u of T (with root u).*

(1.35) Lemma *The nodes in any SCC of G span a prefix subtree of the DFS tree.*

Proof. Let K be the SCC and let u be its first node. All nodes in K are descendants of u. If w is a node in K, let P be the ancestors of w between u and w (inclusive). By Lemma 1.31, P is entirely in K, as required. ∎

1.4 Procedure to compute strong components

This procedure will be scc(u), an enhanced DFS procedure.[2]

(1.36) Definition *A* pushdown stack *is a LIFO (last in, first out) queue wit two operations:* **push** *an item onto the stack, the item becoming the 'topmost' stack element, and* **pop** *the item currently on top of the stack (and therefore pushed later than the other nodes on the stack).*

Here is an approximate version of the algorithm (Figure 4). There is a stack `stack` which is initially empty.

(1.37) Lemma *Granted that the condition* `u == scc_root (u)` *can be tested in some way, the above procedure outputs the SCCs correctly.*

Sketch proof. By induction on postorder rank (which is not actually computed). Suppose that u is an SCC root of an SCC K. Recall that the nodes in K span a prefix subtree (of the DFS forest) rooted at u, so they all get pushed on the stack during `scc (u)`.

Suppose that v is a descendant of u which is root of another SCC. All its descendants are pushed on the stack during scc (v); but they are all cleared at the final part of that scc. On the other hand, no descendant of v is in K, because otherwise $v \in K$ (Lemma ——), so no node from K is cleared off the stack.

Therefore at the end of dfs (u), u and those nodes above it on the stack are precisely the nodes in K and are output correctly at the end of dfs (u). ∎

All that is needed is some way of testing whether u is an SCC root (at the end of dfs(u)).

Two extra fields, on_stack and back_ref, are associated with each node u.

Recall the terms 'unseen, active, finished.' It is intended that when v is finished, `v->on_stack` means that the earliest node in the same SCC as v is active (and still on the stack).

The pointer `back_ref` generally points to an earlier node in the same SCC. We could try and arrange that at the end of dfs(u),

$$\texttt{u->back\_ref} = \text{scc\_root}\,(u),$$

but this is not generally possible. A path from u to scc_root (u) in G could be a mixture of tree edges (forward edges are redundant), cross edges, and back edges. But the back edges are to ancestors which are still active, so it is uncertain whether they are SCC roots.

So we must settle for: when scc(u) ends,

$$\texttt{u->back\_ref} = u \quad \Longleftrightarrow \quad u \text{ is root of its SCC}$$

The full version, version 2

- Pushes nodes on the stack in preorder.

[2] Aho, Hopcroft, and Ullman, 'The design and analysis of computer algorithms,' or AHU DACA for short; they cite Tarjan, 'Depth first search and linear graph algorithms,' *SIAM J. Computing* **1:2** 1972, 146–160. There is another linear-time algorithm due to Sharir which is much easier to explain, but less efficient since it uses both forward and backward adjacency lists.

```
void scc ( NODE * u )
{
  u->pre_rank = pre_count ++;
  push u on stack;
  for all edges (u,v)
    if ( v -> pre_rank < 0 )
    {
      v->parent = u;
      scc ( v );
    }

/*
 * post_ref is not needed, but would
 * be assigned at this point
 */

  if ( u == scc_root (u), i.e., earliest in SCC )
  {
    printf "new SCC"
    repeatedly
      v = top node on stack, popped from stack
      printf v
    until
      v == u
  }
}

main()
{
  initialise stack -- empty, pre_count, etc

  for each node u
    if ( u->pre_rank < 0 )
      scc (u);
}
```

Figure 4: scc code, version 1.

- `u->on_stack` means what it says.
- The crux is: u remains on the stack until dfs(scc_root (u)) finishes.
- In dfs(u), `u->back_ref` is initialised to u, and updated as described in the Figure 5.

(1.38) Lemma *The following invariants hold after* scc(u) *ends:*

(1) `u->back_ref` $= w$ *where* $w = u$ *or there is a descendant* v *of* u *such that* (v, w) *is a back edge or a cross edge and* w *was on the stack when* scc(u) *ended.*

(2) *The nodes on the stack are those nodes* v *for which* scc_root (v) *is active (scc unfinished).*

(3) *For every SCC* K*, let* S *be the nodes in* K *which are on the stack. Then the nodes in* S *(if any) are in a contiguous interval on the stack.*

(4)
$$\texttt{u->back\_ref} = u \iff u = \text{scc\_root}\,(u).$$

Proof (sketch). (1) is fairly evident from an inspection of the code.
For brevity we shall skip the details of (2,3). The rest (i.e., (4)) is proved by induction.

Basis. If u is the first node in postorder, then it is a leaf in the DFSF, with only back edges and no cross edges (cross edges end at nodes with lower postorder rank).

(4) If there is a back edge (u, v), then v is in the same SCC and $u \neq$ scc_root (u). Otherwise u forms an SCC on its own, $u =$ scc_root (u), and it is removed from the stack, preserving (2) and (3).

Induction. Let $w =$ `u->back_ref` (when dfs(u) ends).

(4) Suppose $w \neq u$. Let (v, w) be the back edge or cross edge mentioned in (1). By (2), scc_root (w) is active when dfs(v) finishes, so it is an ancestor of v. It is ancestor or descendant of u, and it precedes u in preorder (since w does), so it is a proper ancestor of u; and there is a path from u to this node, so scc_root $(u) =$ scc_root $(w) \neq u$.

Conversely, suppose $u \neq$ scc_root (u). There is a path from u to scc_root (u) in G, and all nodes on the path are in the same SCC. Without loss of generality, there are no forward edges on the path (they can be replaced by branches of the DFSF).

There is an edge (v, w) on the path where v is a descendant of u but w is not. It must be a back edge or a cross edge. Since w is in the same SCC as v, it is on the stack during dfs(v), and it is a candidate for `u->back_ref`.

If it is a back edge, then, since w is not a descendant of u, it is a proper ancestor and precedes u in preorder.

If it is a cross edge, then w precedes v both in preorder and postorder, and v precedes u in postorder. If w follows u in preorder, then w is a descendant of u, which is false: therefore w precedes u in preorder, it is a candidate for `u->back_ref`, and $u \neq$ `u->back_ref` when dfs(u) ends. ∎

```
void scc ( NODE * u )
{
  u -> pre_rank = pre_count; ++ pre_count;
  u -> back_ref = u;
  push ( stack, u ); u -> on_stack = 1;

  for all out-edges (u,v) (i.e., from node u)
  {
    if (  v->pre_rank < 0 )
    {
      v -> parent = u;
      scc ( v )
      if ( v -> on_stack && v->pre_rank
                < u->back_ref->pre_rank )
      { u->back_ref = v; }
    }
    else if ( v->on_stack && v->pre_rank < u->back_ref->pre_rank )
    { u->back_ref = v; }
    if ( u -> back_ref == u )
    {
      printf ("Strong component...\n")
      finished = 0;
      while ( ! finished )
      {
        v := pop ( stack );
        v . set_on_stack ( false )
        printf ( "v in suitable form\n" );
        finished = ( u == v );
      }
    }
  }   /* end for-loop */
}
```

Figure 5: scc code, version 2

2 Floyd-Warshall and Dijkstra algorithms

(2.1) Shortest weighted path (lightest path). Suppose that a digraph G carries *edge weights,* real numbers associated to the edges of the digraph. The weighted length of a path in G is defined as the sum of the weights of the edges on the path (zero for paths of length zero). The weighted distance from u to v is the minimum value of weighted path-lengths of all paths from u to v in G. This value could be $-\infty$.

The *shortest path problem* is to calculate for each pair u, v of nodes the weighted distance from u to v in G. If one only wants to compute the minimal weight of all paths from a fixed node u it is called the *single source* shortest-path problem. It is better to speak of a *lightest path problem* since 'shortest' suggests fewest edges.

Floyd-Warshall algorithm. This solves the shortest (weighted) path problem for those graphs which, while they may have negative edges, do not have negative cycles; this ensures that all weighted distances are finite.

(2.2) Lemma (i) *If there exist negative cycles, then there exist negative simple cycles.*

(ii). *If there are no negative cycles, then the weighted distance from u to v is infinite or achieved along a simple path.*

Sketch proof. (i) Given a negative cycle which is not simple, decompose it into a union of two shorter cycles. One of these must be negative. Repeat the process until a negative simple cycle is obtained.

(ii) Let u and v be two nodes, and suppose that a path from u to v exists. Assuming there are no negative cycles then a lightest path exists between them; choose a lightest path between them containing as few nodes as possible. Then the path must be simple, since otherwise a cycle could be removed from it without increasing the weighted path-length. ∎

The Floyd-Warshall algorithm is the only graph algorithm we shall see which uses (something like) an adjacency matrix. A matrix W is initialised as follows:[3]

- If $i = j$, then `W.item(i,j) = 0`.
- Otherwise, if (i, j) is an edge, then `W.item(i,j)` is the weight of that edge,
- otherwise `w.item(i,j)` $= \infty$.

The nodes are indexed from 1 to n.

(2.3) Runtime and correctness of Floyd-Warshall algorithm. The runtime of this algorithm is obviously proportional to n^3.

Purely to analyse the algorithm's correctness, sets $P(i, j, k)$ are defined recursively. Each set $P(i, j, k)$ is a set of paths from i to j in the graph. We use the following notation: if ρ and σ are paths in the graph, and σ begins where ρ ends, then $\rho\sigma$ is the 'concatednated path.' Explicitly, if $\rho = u_0, \ldots u_k$ and $\sigma = v_0, \ldots v_\ell$ where $u_k = v_0$, then

$$\rho\sigma = u_0, \ldots u_k, v_1, \ldots v_\ell.$$

[3] There is a correction here, as the diagonal entries are initialised to 0, not ∞.

```
from
  k := 1
until
  k > n
loop
  for all indices i, j
    x := w.item (i, k) + W.item (k, j)
    if x < w.item (i, j) then
      w.put (x, i, j)
    end
  end
  k := k + 1
end
```

Eiffel code 2.1: Floyd-Warshall algorithm.

The sets $P(i, j, k)$ are defined as follows. $P(i, j, 0) = \{(i, j)\}$ if the latter is an edge, otherwise $\emptyset$.

When $k < n$, $P(i, j, k + 1)$ is defined as

$$P(i, j, k) \cup \{\pi\rho\colon \pi \in P(i, k + 1, k), \rho \in P(k + 1, j, k)\}.$$

Intuitively, $P(i, j, k)$ consists of all paths from i to j whose intermediate nodes are all $\leq k$, and such that if an intermediate node k' occurs more than once, then between occurrences there is another node of higher index (in particular, k is visited at most once). The following lemma is easy to prove by induction.

(2.4) Lemma (i) $W(i, j, k)$ *is the minimum weighted length of all paths in* $P(i, j, k)$. (ii) $P(i, j, n)$ *includes all simple paths from* i *to* j. ∎

This basically justifies the algorithm. If at the end of processing the matrix W has a negative diagonal entry, then it has a negative cycle and the problem remains unsolved. Otherwise, all paths in $P(i, i, n)$ are nonnegative, so there are no negative cycles; and $W(i, j)$ gives the minimum weighted distance over a set of paths including all the simple paths, so it gives the correct answer.

(2.5) Dijkstra's algorithm with single source s. This algorithm calculates the minimum weighted distance $d(v)$, say, of every node v from a fixed 'source' node s.

The algorithm requires all edge-weights to be nonnegative, but it is more efficient than the Floyd-Warshall algorithm, and is suitable for the adjacency list realisation.

In Dijkstra's algorithm the set of nodes is partitioned into two sets P and T, 'permanent' and 'tentative.' Initially P contains only the source node s. For each node v an estimate, possible too high, of its weighted distance from u is stored with v and called $w(v)$. Initially $w(v) = 0$ if $v = s$, ∞ otherwise.

```
from
  T := V; P := empty set; initialise w(...)
until
  T is empty
loop
  choose some u in T with
  w(u) minimal, and move u
  from T into P ------------------ (A)

  for all u's out-edges (u,v)
    if v is in T
      x := w(u) + w(u,v)
      if x < w(v) then
        w(v) := x
      end
    end
  end ---------------------------- (B)
end
```

Eiffel code 2.2: Dijkstra's algorithm

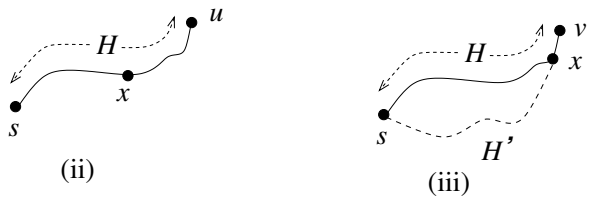

Figure 6: Dijkstra where (ii) or (iii) fails first.

(2.6) Lemma Correctness of Dijkstra's algorithm. *The following invariant condition is maintained:* (i) *For all nodes* v, $w(v) \geq d(v)$. (ii) *For permanent nodes* v, $w(v) = d(v)$. (iii) *At point* (B) *in each iteration, for all nodes* v, $w(v)$ *is the minimum weighted length of all those paths from* s *to* v *all of whose nodes, except maybe the last, are permanent.*

Proof. (i): It is easy to show by induction that either $w(v) = \infty$ or $w(v)$ is the weighted distance along *some* path from s to v, hence $w(v) \geq d(v)$.

Initially, (ii) is true vacuously since $P = \emptyset$. Also, (iii) is true vacuously.

Suppose that (ii) or (iii) is violated sometime during the algorithm.

Suppose that (ii) is violated first. This can only happen when a node u is made permanent but $w(u) > d(u)$. This would happen at the point marked (A) in the code.

Let H be a lightest path from s to u. Because (iii) holds for all nodes including u, this path contains at least one 'tentative' node; let x be the *first.* Let $H[s : x]$ and $H[x : u]$ be the subpaths from s to x and from x to u, $w(H)$ the weight of H.

Then

$$d(u) = w(H) = w(H[s : x]) + w(H[x : u]).$$

Since $w(H[x : u]) \geq 0$, $w(H[s : x]) \leq d(u) < w(u)$. But all nodes in $H[s : x]$ except x are permanent, so by (iii), $w(H[s : x]) = w(x)$. Therefore $w(x) < w(u)$, and x is tentative, so x should have been chosen before u, which is false: therefore (ii) cannot fail before (iii). See Figure 6 (ii).

Suppose that (iii) is violated for the first time. We only consider when this happens at point (B) of the code. There exists a node v such that $w(v)$ is too high at point (B).

So there is a path H from s to v with all nodes except maybe v in P, and $w(H) < w(v)$. Without loss of generality, H is simple, visiting each node at most once.

Let x be the second-last node in H. If $x \neq u$ then x was made permanent before u. After x was made permanent there was a path H' from s to x, containing only permanent nodes, such that $w(H') = w(x) = d(x)$. At that time u is tentative, so H' avoids u. Extend the path H' by the edge (x, v), and we have a path H'' of weight $d(x) + w((x, v)) \leq w(H)$. All nodes in H'' except v were made permanent when or before x was. Therefore $w(v) \leq w(H'')$ (after x was made permanent). This contradicts the inequalities $w(H'') \leq w(H) < w(v)$.

Hence u is the second-last node in H. Since H is simple, all nodes before u on H were made permanent before u, so the weighted length of H is at most

$$w(u) + w((u, v)).$$

So $w(v)$ should have been revised to this value, a contradiction. Q.E.D.

(2.7) Runtime of Dijkstra's algorithm. Runtime. $O(n^2)$ with simple implementations. These can be improved ($O(m + n \log n)$).

3 Network Flows

A *Network* N is a directed graph with a distinguished *source node* s and *target node* t, and for each edge (u, v) a nonnegative *capacity* $c(u, v) \in [0, \infty)$.

There can be edges entering s and edges leaving t, so s is not a source in the sense of acyclic digraphs.

Let E be the set of directed edges of N. A *Flow* on N is a map $f\colon E \to \mathbb{R}$ satisfying the following:

For any node v, the *net flow out of* v is defined as

$$\sum_{(v,w)\in E} f(v, w) - \sum_{(u,v)\in E} f(u, v),$$

- The flow must be *conserved* at every node v except s or t: that is, the net flow out of v is zero except if v is source or target.
- The flow must not exceed capacity: for every edge (u, v), $0 \leq f(u, v) \leq c(u, v)$.

(3.1) Lemma *Let v be the net flow out of s. Then the net flow out of t is $-v$.*

Proof. Add together the net flows out of all nodes v. For every edge (u, v), $f(u, v)$ is added to the net flow out of u and subtracted from the net flow out of v. Therefore the sum of all net flows is zero. Since the net flow is zero except at s and t, the net flows out of s and t are complementary. ∎

(3.2) Definition *The* Volume *of f is the net flow out of s.*

The maximum (volume) flow problem is: given a network N, to construct a flow f whose volume is maximal.

Given a flow f, one wants to determine whether its volume is maximal, and if not, to construct a flow f' whose volume is greater than that of f. This can be done using *Augmenting paths*.

(3.3) Definition *If e is an edge of N, suppose $e = (u, v)$. Then e^+ is (u, v), called a* forward edge*, and e^- is (v, u), called a* backward edge*. Given a flow f, the (residual) capacity of e^+ is defined as $c(u, v) - f(u, v)$, and the (residual) capacity of e^- is defined as $f(v, u)$.*

An augmenting path *is a simple path consisting of signed (forward or backward) edges, all of positive residual capacity. The* capacity *of an augmenting path is the minimum capacity of the edges along the path.*

Note that an augmenting path consists of a sequence of *signed* edges; the signs are explicit, and the underlying edges need not be edges of N. Of course, the underlying edges must define a simple path. Given a flow f and an augmenting path Π from s to t, let ε be its capacity. Define a mapping f' on E:

For any edge (u, v) of N, if it occurs as a forward edge in Π (i.e., $(u, v)^+$ is an edge on Π), then $f'(u, v) = f(u, v) + \varepsilon$; if it occurs as a backward edge on Π, i.e., $(u, v)^-$ is an edge on Π, then $f'(u, v) = f(u, v) - \varepsilon$; otherwise $f'(u, v) = f(u, v)$. Since Π is simple, the edge cannot occur both as forward and backward edges.

(3.4) Lemma *With f, Π, ε, and f' as above, f' is a flow and its volume exceeds that of f by ε.*

Proof. Let v be a node different from s and t. If v is not on Π then the flow values of all edges incident to v are the same for f and f', so the net flow out of v remains zero. If v is on Π then there are exactly two (signed) edges ${e_1}^{\pm}$ and ${e_2}^{\pm}$ on Π incident to v. If both are backward edges then the flow is reduced by ε both, so the net flow out of v is unchanged. Similarly for the other three combinations of signs.

Thus conservation holds for f'.

Also, the capacity constraint holds. Suppose (u, v) is an edge. If $f'(u, v) = f(u, v)$, then the capacity constraint still holds. If $f'(u, v) = f(u, v) + \varepsilon$, then $0 < f'(u, v) \leq c(u, v)$ by definition of ε. If $f'(u, v) = f(u, v) - \varepsilon$, then $0 \leq f'(u, v) < c(u, v)$ by definition of ε.

Therefore f' is a flow. The source s is incident to exactly one edge on Π; if it is a forward edge then the flow out of s is increased by ε; if it is a backward edge then the flow into s is decreased by ε. In either case the net flow out of s is increased by ε. ∎

Construction of augmenting path. One can search for augmenting paths from s by building a tree of augmenting paths from s and continuing until t is reached or the tree cannot be extended. In the description, a tree of augmenting paths is being built from s. S is the set of tree nodes and F, the 'fringe,' is the set of nodes (always leaves of the current tree) from which the tree can be extended.

The algorithm is as follows (f is a flow fixed on N).

```
from
  S := {s}; F := {s}; s.set_parent (Void)
until
  t in S or F empty
loop
  Choose some node v in F and delete it from F

  for all edges (v,w) incident to v loop
    if w not in S and f(v,w) < c(v,w) then
      add w to S and to F
      w.set_parent (v)
      w.mark ( "forward" )
    end
  end

  for all edges (u,v) incident to v loop
    if u not in S and f(u,v) > 0 then
      add u to S and to F
      u.set_parent (v)
      u.mark ( "backward" )
    end
  end
end
```

Obviously if t gets added to S then there exists an augmenting path (and it can be recovered from the tree links). Otherwise, let S be the set of nodes labelled by the above procedure, and T its complement.

(3.5) Definition *A* Cut *is a partition S, T of the nodes of the network with s in S and t in T. The* capacity *of the cut is the sum*

$$\sum \{c(u,v) \colon u \in S, v \in T, (u,v) \in E\}.$$

(3.6) Lemma *For any cut S, T, the flow volume is bounded by the capacity of the cut. If the above algorithm fails to construct an augmenting path, then the cut S, T in which S is the set of nodes visited has capacity equal to the flow volume, and hence the flow is maximal.*

Proof. For any cut S, T, consider

$$\sum_{v \in S, w \in T} f(v,w) - \sum_{v \in S, u \in T} f(u,v),$$

which is called the *net flow across the cut.* If one adds together the net flows out of all nodes v in S, the total is the flow volume, because only s has nonzero net flow in S. Let (u,v) be an edge with at least one end in S. If the other end is also in S, then the flow $f(u,v)$ makes a net contribution of zero to this total. If $u \in S$ and $v \in T$ then $f(u,v)$ is added to the total; if $u \in T$ and $v \in S$ then $f(u,v)$ is subtracted from the total. Hence the flow volume is the net flow across the cut.

Hence the flow volume is bounded by

$$\sum_{v \in S, w \in T} f(v,w) \leq \sum_{v \in S, w \in T} c(v,w),$$

so the flow volume is bounded by the cut capacity.

If S, T form the cut defined after an unsucessful attempt to form an augmenting path, then for any $u \in S$, $v \in T$, if (u,v) is an edge then $f(u,v) = c(u,v)$, since u but not v is visited; if (v,u) is an edge then the flow is zero on it, since u but not v is visited. Hence for this cut the flow across the cut equals its capacity. ∎

(3.7) Corollary (theorem of Ford and Fulkerson). *The maximum flow volume equals the minimum cut capacity. (Proof omitted.)* ∎

We now have an outline procedure for constructing a maximum volume flow: begin with the zero flow, repeatedly search for an augmenting path, and, if found, augment, until a maximum flow is reached.

There is no bound on the possible number of iterations; if the edge capacities are irrational this procedure could continue indefinitely. Efficiencies are gained if the tree of augmenting paths is built up breadth-first. This is achieved by maintaining F as a FIFO queue. Following this strategy, we are assured that the augmenting path chosen is always as short as possible.

(3.8) Definition *For any node u and all relevant k, let $\sigma^{(k)}(u)$ be the length of the shortest augmenting path from s to u after k (shortest-path) augmentations have been executed; and let $\tau^{(k)}(u)$ be the length of the shortest augmenting path from u to t after k augmentations. The default (when no such path exists) is ∞.*

(3.9) Lemma *For any u, the quantities $\sigma^{(k)}(u)$ and $\tau^{(k)}(u)$ are nondecreasing with k.*

Proof. The argument is the same for σ and τ: we consider only σ. Suppose in contradiction that $\sigma^{(k+1)}(v) < \sigma^{(k)}(v)$ for some v. Choose a shortest augmenting path Π from s to v after the $k+1$st augmentation; its length is $\sigma^{(k+1)}(v)$. Clearly $v \neq s$: let u be the second-last node on Π. We can assume that v is as close as possible to s on Π, so

$\sigma^{(k+1)}(v) < \sigma^{(k)}(v)$; $\sigma^{(k+1)}(u) \geq \sigma^{(k)}(u)$: note that $\sigma^{(k+1)}(v) = 1 + \sigma^{(k+1)}(u)$.

Therefore $1 + \sigma^{(k)}(u) \leq 1 + \sigma^{(k+1)}(u) < \sigma^{(k)}(v)$. This implies that the edge (u, v) is available neither as a forward nor a backward edge after the kth augmentation, so the flow on (u, v) and (v, u) was unchanged after the $k+1$st augmentation; therefore the edge (u, v) is not available for augmenting after the $k+1$st augmentation, a contradiction, since (u, v) is on Π. ∎

(3.10) Corollary (Edmonds and Karp). *If the 'shortest augmenting path' strategy is used, then the algorithm runs in time $O(m^2 n)$, where n is the number of nodes and m the number of edges of N (assuming $m \geq n$).*

Proof. We shall see that the running time is $O(mn(m+n)$. The assumption that $m \geq n$ — which is true for all reasonable networks — allows this to be simplified to $O(m^2 n)$. The cost of constructing an augmenting path is $O(m+n)$, so it is enough to show that there are $O(mn)$ augmenting phases.

An edge $e^{\pm}$ on an augmenting path is *critical* if its (residual) capacity equals the capacity of the path. Let $e = (u, v)$ be an edge. Claim that it can be critical at most n times. Suppose that it is critical in the $k+1$st augmenting path, and that it is later critical in the $k'+1$st. Suppose that the edge first recurs after the $k+1$st augmentation (as a forward or backward edge) in the $\ell+1$st augmentation.

If Π is the $i+1$st augmenting path, and w is any node on Π, then the length of Π is

$$\sigma^{(i)}(w) + \tau^{(i)}(w),$$

By the above lemma, it is enough to show that $\sigma^{(\ell)}(w) \geq 2 + \sigma^{(k)}(w)$, where w is a node common to the $k+1$st and $\ell+1$st augmenting paths; because then the length of the $\ell+1$st augmenting path exceeds that of the $k+1$st by at least 2; no augmenting path has length greater than $n-1$; so e can occur as a critical edge at most n times, and therefore there are at most mn augmentations.

Either e^+ or e^- occurs in the $k+1$st augmenting path. If e^+, i.e., (u, v), occurs in the $k+1$st path, then

$$\sigma^{(k)}(v) = 1 + \sigma^{(k)}(u)$$

and after the $k+1$st augmentation, the edge (u, v) is saturated by the flow. The flow on (u, v) is then unchanged up to the ℓth augmentation, so it remains saturated. Therefore its flow can only be reduced in the $\ell+1$st augmentation, e^- is on that augmenting path, and therefore

$$\sigma^{(\ell)}(u) = 1 + \sigma^{(\ell)}(v).$$

Therefore

$$\sigma^{(\ell)}(u) = 1 + \sigma^{(\ell)}(v) \geq 1 + \sigma^{(k)}(v) = 2 + \sigma^{(k)}(u).$$

If e^- occurs in the $k+1$st augmenting path, then by similar arguments

$$\sigma^{(\ell)}(v) \geq 2 + \sigma^{(k)}(v).$$

In any case, the quantity

$$\sigma^{(\cdot)}(u) + \sigma^{(\cdot)}(v)$$

increases by at least 2, and never exceeds $2n$, so there each edge is critical at most n times as claimed. ∎

(3.11) *Layered networks.* The algorithm can be improved by using the idea of *Layered network.* This is used to augment more effectively than a single augmenting path would allow. The $i+1$st augmenting network L_{i+1} is built after i (generalised) augmentations have been performed, beginning with the zero flow. After i augmentations, let f be the flow. The augmenting network L_{i+1} is defined as follows:

(i) Apply a breadth-first search to locate a shortest augmenting path from s to t. If this search is unsuccessful, the flow is maximal and L_{i+1} is undefined. Otherwise, let the nodes visited be assigned *levels* indicating their distance from s in this breadth-first search. Thus the level of a node v is $\sigma^{(i)}(v)$ (default: ∞.) Unvisited nodes are not in L_{i+1}.

(ii) Add forward edges between adjacent layers of L_{i+1} as follows: if u and v are at levels j and $j+1$, say, in the network, and either (a) (u,v) is an unsaturated edge, (b) (v,u) has nonzero flow, or (c) both[4] then add the edge (u,v) to the layered network. In case (a), its capacity is $c(u,v)-f(u,v)$. In case (b), its capacity is $f(v,u)$. In case (c), its capacity is $c(u,v)+f(v,u)-f(u,v)$. The edge should be labelled to indicate which case applies.

To augment the flow, first construct a maximal flow ϕ on L_{i+1}. This is easier to construct for the restricted kind of network L_{i+1} (we shall describe an $O(n^2)$ method below). For every edge (u,v) of L_{i+1}, in case (a) add $\phi(u,v)$ to the flow on (u,v); in case (b) subtract $\phi(u,v)$ from $f(v,u)$; and in case (c), if $\phi(u,v) \leq f(v,u)$ then subtract it from $f(v,u)$, otherwise reduce $f(v,u)$ to zero and add the balance to $f(u,v)$.

In L_{i+1} every path from s to t is an augmenting path (and all such augmenting paths have the same length). The flow ϕ can be characterised as follows: every path from s to t in the layered network contains at least one critical edge.

(3.12) Lemma *If u and v are both in L_i and L_{i+1}, and (u,v) is an edge of L_{i+1}, then $\sigma^{(i-1)}(v) \leq 1+\sigma^{(i-1)}(u)$.*

Proof. Otherwise u and v are not in adjacent layers of L_i (and v is further from s than u). Then (u,v) cannot be an edge of L_i, so the flow on (u,v) and (v,u) was unchanged at the i-th augmentation; and it saturated (u,v) and was zero on (v,u). Hence it cannot be an edge of L_{i+1}. ∎

(3.13) Lemma *If L_i and L_{i+1} are both defined, then $\sigma^i(t) \geq 1+\sigma^{(i-1)}(t)$.*

Proof. Let $\Pi = v_0, v_1 \ldots v_k$ be a path from s to t in L_{i+1}. We want to show that $k > \sigma^{(i-1)}(t)$.

Assume the contrary.

First we deduce that all nodes v_j are nodes in L_i. For otherwise, choose j so that v_{j+1} is not, but all earlier nodes on Π are. Notice by the previous lemma (using induction) that v_j is at level j or less in L_i. There are two possible reasons:

(a) $v_{j+1} \neq t$ but t is in the next level to v_j in L_i. Since $j+1 \neq k$, $\sigma^{(i-1)}(t) \leq j+1 < k$, contrary to assumption.

(b) The edge (v_j, v_{j+1}) (if it exists in N) is saturated (after the $i-1$st augmentation), and (v_{j+1}, v_j) has zero flow (if it exists in N). Then the flow on both these edges is unchanged after the ith augmentation, so (v_j, v_{j+1}) is not an edge of L_{i+1}, a contradiction.

Therefore under our assumption, all nodes v_j are in L_i. It follows from the above lemma that for any j, (i) $\sigma^{(i-1)}(t) \leq \sigma^{(i-1)}(v_j) + k - j$, and (ii) $\sigma^{(i-1)}(v_j) \leq j$. Both these facts can be proved with an inductive argument based on the previous lemma.

[4] Actually, (c) does not occur if the flow has been constructed according to the algorithm.

From (i), if any v_j is closer to s in L_i than in L_{i+1}, then t is closer to s in L_i than in L_{i+1}, which we assumed not to be the case. Combining this with (ii), we deduce that v_j is at level j in L_i for all j.

Therefore, for each j, (v_j, v_{j+1}) is an edge of L_i; for if any of these edges is not admissible, the flow remains the same on it and its inverse, and it is not admissible as an edge in L_{i+1}.

So Π is an augmenting path from s to t in L_i. By the maximality of the augmentation, at least one edge on Π should have been saturated by the ith augmentation. So that edge should not have been in L_{i+1}. This contradiction finishes the proof. ∎

(3.14) Construction of ϕ (sketch).[5] Let L be a layered (augmenting) network, ϕ a flow on L. For any node v in L (except s and t), let its *potential* $\rho_\phi(v)$ be the minimum of the following two quantities: total residual capacity of (a) all edges coming into v, or (b) all going out of v.

Delete from L all saturated edges and all nodes with zero potential (and their incident edges), except s and t. Let v be a node with minimum potential d. It is possible to 'push' d extra units of flow from v to the next layer, and 'pull' d units of flow from the previous layer. One can repeat this process until d extra units of flow are brought forward from s to t.

During this process edges can become saturated, so they are deleted from L, and nodes (including v) acquire zero potential, so they can be deleted from L.

One repeats this process until all nodes except s and t have been eliminated. With careful implementation, the cost of selecting a node of minimum potential is $O(n)$, and the cost of bringing d more units of flow from s to t is proportional to $n + e$ where e is the number of edges deleted. The overall cost is then $O(n^2 + m)$ or $O(n^2)$.

4 LU factorisation

Consider Gaussian Elimination without pivoting on the matrix

$$\begin{matrix} 1 & 2 & 3 \\ 4 & 5 & 6 \\ 7 & 8 & 10 \end{matrix}$$

```
1 2  3 = R1          1   2   3
4 5  6 - 4 R1  ->    0  -3  -6
7 8 10 - 7 R1        0  -6 -11
This can be 'inverted':
1 2  3        1 0 0           1  2   3
4 5  6  =     4 1 0  times     0 -3  -6
7 8 10        7 0 1           0 -6 -11
Operating on the second column:
1  2   3                 1  2  3
0 -3  -6   = R2     ->   0 -3 -6
0 -6 -11   - 2*R2       0  0  1   ... Upper triangular.
This can be inverted
1  2   3       1 0 0           1  2  3
```

[5] Malhotra, Kumar, Maheshwari: An $O(|V|^3)$ algorithm for finding maximum flows in networks. *IPL* **7:6** (1978) 277–278.

```
0 -3  -6   =  0 1 0  times  0 -3 -6
0 -6 -11      0 2 1         0  0  1
so
1 2  3       1 0 0         1 0 0         1  2  3
4 5  6  =    4 1 0  times   0 1 0  times  0 -3 -6
7 8 10       7 0 1         0 2 1         0  0  1
or
1 2  3     1 0 0         1  2  3
4 5  6  =  4 1 0  times  0 -3 -6
7 8 10     7 2 1         0  0  1
--- a lower triangular matrix L (with 1s on the diagonal)
times an upper triangular matrix U.
This last is called an LU factorisation.
```

The factorisation can be calculated quite easily, when there is no pivoting (swapping). Start with

$$A = \begin{bmatrix} 1 & 2 & 3 \\ 4 & 5 & 6 \\ 7 & 8 & 10 \end{bmatrix} = LU = \begin{bmatrix} 1 & 0 & 0 \\ ? & 1 & 0 \\ ? & ? & 1 \end{bmatrix} \begin{bmatrix} ? & ? & ? \\ 0 & ? & ? \\ 0 & 0 & ? \end{bmatrix}$$

The first row of U equals that of A.

$$\begin{bmatrix} 1 & 2 & 3 \\ 4 & 5 & 6 \\ 7 & 8 & 10 \end{bmatrix} = \begin{bmatrix} 1 & 0 & 0 \\ ? & 1 & 0 \\ ? & ? & 1 \end{bmatrix} \begin{bmatrix} 1 & 2 & 3 \\ 0 & ? & ? \\ 0 & 0 & ? \end{bmatrix}$$

The first column of L can be completed.

$$\begin{bmatrix} 1 & 2 & 3 \\ 4 & 5 & 6 \\ 7 & 8 & 10 \end{bmatrix} = \begin{bmatrix} 1 & 0 & 0 \\ 4 & 1 & 0 \\ 7 & ? & 1 \end{bmatrix} \begin{bmatrix} 1 & 2 & 3 \\ 0 & ? & ? \\ 0 & 0 & ? \end{bmatrix}$$

From the second row of L the second row of U can be computed.

$$\begin{bmatrix} 1 & 2 & 3 \\ 4 & 5 & 6 \\ 7 & 8 & 10 \end{bmatrix} = \begin{bmatrix} 1 & 0 & 0 \\ 4 & 1 & 0 \\ 7 & ? & 1 \end{bmatrix} \begin{bmatrix} 1 & 2 & 3 \\ 0 & -3 & -6 \\ 0 & 0 & ? \end{bmatrix}$$

From the second column of U the third row, second column of L can be computed.

$$\begin{bmatrix} 1 & 2 & 3 \\ 4 & 5 & 6 \\ 7 & 8 & 10 \end{bmatrix} = \begin{bmatrix} 1 & 0 & 0 \\ 4 & 1 & 0 \\ 7 & 2 & 1 \end{bmatrix} \begin{bmatrix} 1 & 2 & 3 \\ 0 & -3 & -6 \\ 0 & 0 & ? \end{bmatrix}$$

From the third row of L the third row of U can be computed.

$$\begin{bmatrix} 1 & 2 & 3 \\ 4 & 5 & 6 \\ 7 & 8 & 10 \end{bmatrix} = \begin{bmatrix} 1 & 0 & 0 \\ 4 & 1 & 0 \\ 7 & 2 & 1 \end{bmatrix} \begin{bmatrix} 1 & 2 & 3 \\ 0 & -3 & -6 \\ 0 & 0 & 1 \end{bmatrix}.$$

Now the LU decomposition can be used to solve any system of equations $Ax = b$: $LUx = b$; solve $Ly = b$ by substitution, then solve $UX = y$ by back-substitution. For example,

$$\begin{bmatrix} 1 & 2 & 3 \\ 4 & 5 & 6 \\ 7 & 8 & 10 \end{bmatrix} \begin{bmatrix} x \\ y \\ z \end{bmatrix} = \begin{bmatrix} 2 \\ 5 \\ 9 \end{bmatrix}$$

First solve

$$\begin{bmatrix} 1 & 0 & 0 \\ 4 & 1 & 0 \\ 7 & 2 & 1 \end{bmatrix} \begin{bmatrix} X \\ Y \\ Z \end{bmatrix} = \begin{bmatrix} 2 \\ 5 \\ 9 \end{bmatrix}$$

$X = 2$; $4X + Y = 5$ so $Y = -3$; $7X + 2Y + Z = 9$ so $Z = 1$. Then solve

$$\begin{bmatrix} 1 & 2 & 3 \\ 0 & -3 & -6 \\ 0 & 0 & 1 \end{bmatrix} \begin{bmatrix} x \\ y \\ z \end{bmatrix} = \begin{bmatrix} 2 \\ -3 \\ 1 \end{bmatrix}.$$

$z = 1$; $-3y - 6z = -3$, so $-3y = 3$ and $y = -1$; $x + 2y + 3z = 2$, so $x - 2 + 3 = 2$ and $x = 1$.

It is relatively easy to invert L and U separately, so one can calculate A^{-1} as $U^{-1}L^{-1}$ (note the order in which they are multiplied).

The general procedure is as follows, if pivoting is not used. Suppose A is an $n \times n$ matrix. Let $A = a_{ik}$, $L = \ell_{ij}$, and $U = u_{jk}$. L and U have restricted format:

$$\begin{aligned} \ell_{ij} &= 0 \quad \text{if } i < j, \\ \ell_{ij} &= 1 \quad \text{if } i = j, \quad \text{and} \\ u_{jk} &= 0 \quad \text{if } j > k. \end{aligned}$$

It is quite easy to compute the factorisation without pivoting. In this case, the first row of U is copied from A. One can then calculate the first column of L, second row of U, second column of L, and so on.

One can prove this by induction. The inductive hypothesis is: suppose that the first $i - 1$ rows of U, and the first $i - 1$ columns of L, have been calculated.

Since $\ell ii = 1$, the i-th row of ℓ is fully known. If we multiply U by this row, we get the i-th row of A:

$$a_{ik} = \sum_j \ell_{ij} u_{jk} = u_{ik} + \sum_{j<i} \ell_{ij} u_{jk}.$$

The only unknown values here are u_{ik}, and they can be calculated from the equations.

When the first i rows of U are known, the i-th column of L can be calculated.

$$\begin{aligned} a_{ri} &= \sum_{j \le i} \ell_{rj} u_{ji} \\ \ell_{ri} u_{ii} &= a_{ri} - \sum_{j<i} \ell_{rj} u_{ji} \end{aligned}$$

Pivoting can also be allowed for. What it means is that a *row-permuted* version of A can be factorised as LU. Partial pivoting amounts to swapping rows of A, and *parts of* rows of L. Take the same example again.

$$A = \begin{bmatrix} 1 & 2 & 3 \\ 4 & 5 & 6 \\ 7 & 8 & 10 \end{bmatrix} = LU = \begin{bmatrix} 1 & 0 & 0 \\ ? & 1 & 0 \\ ? & ? & 1 \end{bmatrix} \begin{bmatrix} ? & ? & ? \\ 0 & ? & ? \\ 0 & 0 & ? \end{bmatrix}$$

Swap rows 1 and 3 of L, to bring up the pivot element. Then the first row of A matches that of U.

$$\begin{bmatrix} 7 & 8 & 10 \\ 4 & 5 & 6 \\ 1 & 2 & 3 \end{bmatrix} = \begin{bmatrix} 1 & 0 & 0 \\ ? & 1 & 0 \\ ? & ? & 1 \end{bmatrix} \begin{bmatrix} 7 & 8 & 10 \\ 0 & ? & ? \\ 0 & 0 & ? \end{bmatrix}$$

The first column of L can be completed.

$$\begin{bmatrix} 7 & 8 & 10 \\ 4 & 5 & 6 \\ 1 & 2 & 3 \end{bmatrix} = \begin{bmatrix} 1 & 0 & 0 \\ 4/7 & 1 & 0 \\ 1/7 & ? & 1 \end{bmatrix} \begin{bmatrix} 7 & 8 & 10 \\ 0 & ? & ? \\ 0 & 0 & ? \end{bmatrix}$$

Now we consider pivoting on the second row. That is, perhaps the second and third rows should be swapped to increase the absolute value of u_{22}.

If we swap, we must swap within the first column of L. Without swapping,

$$(4/7)8 + u_{22} = 5, \quad u_{22} = 3/7$$

and with swapping

$$(1/7)8 + u_{22} = 2, \quad u_{22} = 6/7$$

which is larger (in absolute value): we swap.

$$\begin{bmatrix} 7 & 8 & 10 \\ 1 & 2 & 3 \\ 4 & 5 & 6 \end{bmatrix} = \begin{bmatrix} 1 & 0 & 0 \\ 1/7 & 1 & 0 \\ 4/7 & ? & 1 \end{bmatrix} \begin{bmatrix} 7 & 8 & 10 \\ 0 & 6/7 & ? \\ 0 & 0 & ? \end{bmatrix}$$

Then $10/7 + u_{23} = 3$ or $u_{23} = 11/7$.

$$\begin{bmatrix} 7 & 8 & 10 \\ 1 & 2 & 3 \\ 4 & 5 & 6 \end{bmatrix} = \begin{bmatrix} 1 & 0 & 0 \\ 1/7 & 1 & 0 \\ 4/7 & ? & 1 \end{bmatrix} \begin{bmatrix} 7 & 8 & 10 \\ 0 & 6/7 & 11/7 \\ 0 & 0 & ? \end{bmatrix}$$

There is no question of pivoting again. $(4/7)(8) + (6/7)\ell_{32} = 5$, so $\ell_{32} = 1/2$.

$$\begin{bmatrix} 7 & 8 & 10 \\ 1 & 2 & 3 \\ 4 & 5 & 6 \end{bmatrix} = \begin{bmatrix} 1 & 0 & 0 \\ 1/7 & 1 & 0 \\ 4/7 & 1/2 & 1 \end{bmatrix} \begin{bmatrix} 7 & 8 & 10 \\ 0 & 6/7 & 11/7 \\ 0 & 0 & ? \end{bmatrix}$$

Then $(4/7)10 + (1/2)(11/7) + u_{33} = 6$, $u_{33} = -1/2$.

$$\begin{bmatrix} 7 & 8 & 10 \\ 1 & 2 & 3 \\ 4 & 5 & 6 \end{bmatrix} = \begin{bmatrix} 1 & 0 & 0 \\ 1/7 & 1 & 0 \\ 4/7 & 1/2 & 1 \end{bmatrix} \begin{bmatrix} 7 & 8 & 10 \\ 0 & 6/7 & 11/7 \\ 0 & 0 & -1/2 \end{bmatrix}$$

5 More about LU factorisation

Gauss-Jordan elimination without pivoting is unreliable. For example, if ϵ is small enough, GJE applied without pivoting to

$$\begin{bmatrix} \epsilon & 1 \\ 1 & 1 \end{bmatrix} \begin{bmatrix} x \\ y \end{bmatrix} = \begin{bmatrix} a \\ b \end{bmatrix}$$

will lose accuracy. However, our analysis will only mention pivoting at the start. Partial pivoting means, of course, that rows are swapped so that the diagonal entry in the i-th row has maximum absolute value.

(5.1) Lemma *Let A be a matrix subject to Gausssian or Gauss-Jordan elimination. The reduction may involve swapping rows. Then it is possible to swap the rows of A at the start so that afterwards swapping is not applied.*

In other words, elimination with pivoting on A is equivalent to elimination without swapping on a row-permuted version of A.

Sketch proof. We consider two successive EROs e_i and e_{i+1} where e_i is not a swap operation but e_{i+1} swaps rows p and q. It is enough (using induction) to show that these two operations can be replaced by two EROs in which the swap comes first. See the case-analysis. If e_i is a scale operation, it scales the k-th row by c. If a subtract operation, it subtracts c times row r from row s. Since swapping p and q is the same as swapping q and p, all cases are covered below.

Scale, p, q, k different, then swap.	Swap, then scale as before.
Swap, p, q, r, s different, then swap.	Swap, then subtract as before
Scale row p, then swap.	Swap, then scale row q.
Subtract $c\times$ row p from row q, then swap.	Swap, then subtract $c\times$ row q from row p.
Subtract $c\times$ row p from row $s \neq q$, then swap.	Swap, then subtract $c\times$ row q from row s.
Subtract $c\times$ row $r \neq q$ from row p, then swap.	Swap, then subtract $c\times$ row r from row q.

This concludes the sketch proof. ∎

The LU-decomposition can be calculated, without pivoting (therefore inaccurate and sometimes unsuccessful), as follows. The first row of U equals that of A, and the first row of L is fixed in advance. We need only calculate

$$\ell_{ji}$$

for $j > i$ and

$$u_{ij}$$

for $j \geq i$.

Since $LU = A$, generally

$$\sum_j \ell_{ij} u_{jk} = a_{ik}.$$

The rest of U and L can be calculated as follows. For $i = 1$ to m, supposing the first $i - 1$ rows of U and columns of L have already been calculated, the i-th row of U and column of L can be calculated as follows.

For $k = 1, 2, \ldots, m$,

$$\sum_{j=1}^{m} \ell_{ij} u_{jk} = a_{ik}.$$

But $\ell_{ii} = 1$ and $\ell_{ij} = 0$ if $j > i$, so for $k = i, i+1, \ldots m$,

$$u_{ik} + \sum_{j=1}^{i-1} \ell_{ij} u_{jk} = a_{ik}.$$

This uses the first $i-1$ columns of L and the first $i-1$ rows of U, which are already computed.

$$u_{ik} = a_{ik} - \sum_{j=1}^{i-1} \ell_{ij} u_{jk} \qquad k = i, i+1, \ldots, m. \tag{5.1}$$

Next the i-th column of L can be computed.

$$\sum_{j} \ell_{kj} u_{ji} = a_{ki}.$$

Since $u_{ji} = 0$ for $j > i$,

$$\sum_{j=1}^{i} \ell_{kj} u_{ji} = a_{ki}$$

$$\ell_{ki} u_{ii} = a_{ki} - \sum_{j=1}^{i-1} \ell_{kj} u_{ji},$$

$$\ell_{ki} = \frac{a_{ki} - \sum_{j=1}^{i-1} \ell_{kj} u_{ji}}{u_{ii}}, \qquad k = i+1, \ldots, m. \tag{5.2}$$

Example.

$$\begin{bmatrix} 1 & 2 & 3 \\ 4 & 5 & 6 \\ 7 & 8 & 8 \end{bmatrix} = \begin{bmatrix} 1 & 0 & 0 \\ 4 & 1 & 0 \\ 7 & * & 1 \end{bmatrix} \begin{bmatrix} 1 & 2 & 3 \\ 0 & * & * \\ 0 & 0 & * \end{bmatrix}$$

$$\begin{bmatrix} 1 & 2 & 3 \\ 4 & 5 & 6 \\ 7 & 8 & 8 \end{bmatrix} = \begin{bmatrix} 1 & 0 & 0 \\ 4 & 1 & 0 \\ 7 & * & 1 \end{bmatrix} \begin{bmatrix} 1 & 2 & 3 \\ 0 & -3 & -6 \\ 0 & 0 & * \end{bmatrix}$$

$$\begin{bmatrix} 1 & 2 & 3 \\ 4 & 5 & 6 \\ 7 & 8 & 8 \end{bmatrix} = \begin{bmatrix} 1 & 0 & 0 \\ 4 & 1 & 0 \\ 7 & 2 & 1 \end{bmatrix} \begin{bmatrix} 1 & 2 & 3 \\ 0 & -3 & -6 \\ 0 & 0 & * \end{bmatrix}$$

$$\begin{bmatrix} 1 & 2 & 3 \\ 4 & 5 & 6 \\ 7 & 8 & 8 \end{bmatrix} = \begin{bmatrix} 1 & 0 & 0 \\ 4 & 1 & 0 \\ 7 & 2 & 1 \end{bmatrix} \begin{bmatrix} 1 & 2 & 3 \\ 0 & -3 & -6 \\ 0 & 0 & -1 \end{bmatrix}$$

6 Error analysis

Errors in calculation are of two types (three if one includes data measurement errors).

- Truncation errors, which are inevitable when convergent processes (with an irrational result), like Newton-Raphson, are terminated after finitely many steps.
- Rounding errors, inevitable with floating-point calculation, whether single- or double-precision.

Errors are generally given **relatively,** as a proportion of the correct answer — or rather, as a proportion of some related quantity. For example, if $\tilde{s}$ is a floating-point evaluation of a sum

$$s = \sum_1^n x_i$$

then the error might be expressed as

$$|s - \tilde{s}| \leq \delta \sum |x_i|$$

(It is known[6] that when $n = 3$ s can be zero without being able to guarantee $\tilde{s} = 0$, so there is no guaranteed error bound relative to s itself.)

(6.1) Definition *Single-precision floating-point numbers have a sign bit, 8 exponent bits, and 23 mantissa bits. The 'machine accuracy'* ϵ_{mach} *for single precision is defined as the smallest* $\varepsilon > 0$ *such that*

$$1 + \varepsilon$$

is a normalised floating-point value. For single precision,

$$\epsilon_{mach} = 2^{-23}.$$

Single precision numbers are not used except where storage is scarce, such as on a satellite or a graphics chip.

Double-precision numbers have 11 exponent bits and 52 mantissa. For double precision,

$$\epsilon_{mach} = 2^{-52}.$$

Here we consider rounding errors. Remember the general principle of IEEE floating-point operations:

> **IEEE guaranteed accuracy.**
> Given single- or double-precision floating-point numbers in normalised range, in the basic arithmetic operations $+, -, \times, /, \sqrt{}$, (and some others), the relative error is bounded by ϵ_{mach}, 2^{-23} or 2^{-52} respectively.

Let us call this the IEEE tolerance.

All our rounding-error estimates use the next definition and Lemma 6.4.

[6] Demmel et al., 2003.

(6.2) Definition *Given* $n\epsilon_{mach} < 1$,

$$\gamma_n = \frac{n\epsilon_{mach}}{1 - n\epsilon_{mach}}$$

(This definition includes $\gamma_0 = 0$*.)*

(6.3) Lemma *If* $p < q$ *(and* $q\epsilon_{mach} < 1$*) then* $\gamma_p < \gamma_q$*.*

Proof. $p/(1 - p\epsilon_{\text{mach}}) < p/(1 - q\epsilon_{\text{mach}}) < q/(1 - q\epsilon_{\text{mach}})$. **Q.E.D.**

(6.4) Lemma *Suppose* n *and* δ_i *,*$1 \leq i \leq n$*, are given, where* $n\epsilon_{mach} < 1$*, and* $|\delta_i| \leq \epsilon_{mach}$ *for* $1 \leq i \leq n$*. Let* $\prod_i (1 + \delta_i)^{\pm 1} = 1 + \theta_n$*. Then*

$$|\theta_n| \leq \gamma_n$$

Partial proof. We give a partial proof because the full proof, while not difficult, is long. Let us consider just two special cases: where all the exponents are positive and the δ_i are nonnegative, and where all the exponents are positive and the δ_i are nonpositive.

In the first case,

$$1 \leq \prod(1 + \delta_i) \leq (1 + \epsilon_{\text{mach}})^n =$$
$$1 + n\epsilon_{\text{mach}} + \frac{n(n-1)}{2!}\epsilon_{\text{mach}}^2 \ldots \leq$$
$$1 + n\epsilon_{\text{mach}} + (n\epsilon_{\text{mach}})^2 + \ldots = 1 + \frac{n\epsilon_{\text{mach}}}{1 - n\epsilon_{\text{mach}}} = 1 + \gamma_n,$$

as required. In the second case,

$$1 \geq \prod(1 + \delta_i) \geq (1 - \epsilon_{\text{mach}})^n =$$
$$1 - n\epsilon_{\text{mach}} + \frac{n(n-1)}{2!}\epsilon_{\text{mach}}^2 \ldots \geq$$
$$1 - n\epsilon_{\text{mach}} - (n\epsilon_{\text{mach}})^2 - \ldots = 1 - \frac{n\epsilon_{\text{mach}}}{1 - n\epsilon_{\text{mach}}} = 1 - \gamma_n,$$

as required. ∎

Here are some applications.

(6.5) Lemma *Unless* n *is absurdly large, the rounding error in evaluating*

$$x_1 + \ldots + x_n$$

is bounded by $\gamma_{n-1} \sum |x_i|$*.*

Proof. Evaluating gives

$$\sum x_i =$$
$$((((x_1 + x_2)(1 + \delta_1) + x_3)(1 + \delta_2) + \ldots + x_{n-1})(1 + \delta_{n-2}) + x_n)(1 + \delta_{n-1}) =$$
$$x_1(1 + \theta_{n-1}) + x_2(1 + \theta_{n-1}) + x_3(1 + \theta_{n-2}) + \ldots + x_n(1 + \theta_1) =$$
$$(\sum x_i) + x_1\theta_{n-1} + x_2\theta_{n-1} + x_3\theta_{n-2} + \ldots + x_n(1 + \theta_1)$$

Here θ_r is a product of r terms of the form $(1+\delta_i)$ where $|\delta_i| \le \epsilon_{\text{mach}}$. By the above two lemmas, $|\theta_r| \le \gamma_{n-1}$ for $1 \le r \le n$, and the overall error is bounded in absolute value by

$$(|x_1| + \ldots + |x_n|)\gamma_{n-1}. \quad \blacksquare$$

Unfortunately, since the signs of the terms can vary, there is no bound connecting the error to $\sum x_i$: the best we can hope for is relate the error to $\sum |x_i|$.

Similarly, the error in evaluating a scalar product

$$\sum_1^n x_i y_i = ((x_1y_1(1+\delta_1) + x_2y_2(1+\delta_2))(1+\delta_3) + x_3y_3(1+\delta_4))(1+\delta_5)\ldots$$

is bounded by

$$\gamma_n \sum |x_i||y_i|.$$

(6.6) Definition *If $x = [x_1, \ldots, x_n]$ is a row or column vector, then $|x| = [|x_1|, \ldots, |x_n|]$. That is, the components of x are $|x_i|$. Similarly, if A is a matrix $[a_{ij}]$, then $|A|$ is the matrix $[|a_{ij}|$.*

With this notation, the error in evaluating the scalar product $x^T y$ is bounded by $\gamma_n |x|^T |y|$.

Backward error analysis. Experts in the field say that it is often easier to relate the approximate solution to a problem to the exact solution to a nearby problem.

For example, suppose $Ux = b$ were solved with approximate solution $\tilde{x}$. We may write Δx for $x - \tilde{x}$. The *forward error* would be Δx. On the other hand, if we produce a related matrix ΔU such that $(U + \Delta U)\tilde{x} = b$ *exactly*, analysis of ΔU would be called backward error analysis.

(6.7) Theorem (forward and back substitution, backward error analysis). *If U is upper triangular, then*

$$(U + \Delta U)\tilde{x} = b \qquad \textit{where}$$
$$|\Delta U| \le \gamma_n |U|. \quad \text{(Definition 6.6)}$$

and similarly for forward substitution, i.e., solving $Lx = b$ where L is lower triangular.

(6.8) Use of backward error analysis. We have

$$Ux = b; \quad (U + \Delta U)\tilde{x} = b$$

so

$$U\Delta x = (\Delta U)\tilde{x}$$

where $\Delta x = x - \tilde{x}$. Also $|\Delta U| \le \gamma_n |U|$.

A result, roughly as difficult as Theorem 6.7, concerns LU factorisation.

(6.9) Proposition

$$\textit{If } A = LU, \qquad A + \Delta A = \tilde{L}\tilde{U}$$

where, in the notation of Definition 6.6,

$$|\Delta A| \le \gamma_n |\tilde{L}||\tilde{U}|. \quad \blacksquare$$

Example: LU-factorisation of the matrix

$$\begin{bmatrix} \epsilon & 1 \\ 1 & 1 \end{bmatrix}$$

(without pivoting) produces

$$\begin{bmatrix} 1 & 0 \\ (1/\epsilon) & 1 \end{bmatrix} \begin{bmatrix} \epsilon & 1 \\ 0 & 1-(1/\epsilon) \end{bmatrix}$$

$$|\tilde{L}||\tilde{U}| = \begin{bmatrix} \epsilon & 1 \\ 1 & (2/\epsilon)-1 \end{bmatrix}.$$

7 Stability of back substititution and LU factorisation

This is taken from *Accuracy and stability of numerical algorithms,* by Nicholas Higham.

(7.1) Theorem *Let U be an $n \times n$ upper-triangular matrix, b a column vector. It is required to solve*

$$Ux = b$$

solved using floating point, with solution $\tilde{x}$. Then we can perturb U so

$$(U + \Delta U)\tilde{x} = b$$
$$|\Delta U| \leq \gamma_n |U|$$

(7.2) Lemma *The rounding error in evaluating*

$$y = \frac{c - \sum_{i=1}^{k-1} a_i b_i}{b_k}$$

(in the given order) produces a computed $\tilde{y}$, where

$$b_k \tilde{y}(1+\theta_k) = c - \sum_{i=1}^{k-1} a_i b_i (1+\theta_i)$$

where $|\theta_i| \leq \gamma_i$.

Proof. Let

$$y_0 = c$$
$$y_1 = (y_0 - a_1 b_1(1+\epsilon_1))(1+\delta_1)$$
$$y_2 = (y_1 - a_2 b_2(1+\epsilon_2))(1+\delta_2)$$
$$\cdots$$
$$y_{k-1} = (y_{k-2} - a_{k-1} b_{k-1}(1+\epsilon_{k-1}))(1+\delta_{k-1})$$
$$\tilde{y} = \frac{y_{k-1}}{b_k}(1+\delta_k)$$

Whence

$$\frac{\tilde{y} b_k}{(1+\delta_1)\cdots(1+\delta_{k-1})(1+\delta_k)} = c - \sum_{i=1}^{k-1} \frac{a_i b_i (1+\epsilon_i)}{(1+\delta_1)\cdots(1+\delta_{i-1})},$$

which furnishes the θ_i: Lemma 6.4 does the rest (granted $k\epsilon_{\text{mach}} < 1$). ∎

7.1 Backward analysis of $Ux = b$.

If the given upper-triangular system is solved in the normal way, $\tilde{x}_n = b_n/u_{nn}$ (rounded) etcetera, then

$$(U + \Delta U)\tilde{x} = b$$

where $|\Delta U| \leq \gamma_n |U|$.

Proof.

$$x_n = b_n/u_{nn}$$

$$x_i = \frac{b_i - \sum_{j=i+1}^{n} u_{ij}x_j}{u_{ii}}$$

$$\tilde{x}_i u_{ii}(1 + \theta_{i,n+1-i}) = b_i - \sum_{j=i+1}^{n} u_{ij}\tilde{x}_j(1 + \theta_{i,n+1-j})$$

$$\sum_{j \geq i} u_{ij}(1 + \theta_{i,n+1-j})\tilde{x}_j = b_i$$

where $|\theta_{i,n+1-j}| \leq \gamma_{n+1-j}$ (Lemma 7.2). Also

$$\Delta u_{ij} = \theta_{i,n+1-j} u_{ij}. \quad \blacksquare$$

7.2 Backward analysis of $A = LU$

To show that $\tilde{L}\tilde{U} = A + \Delta A$ where $|\Delta A| \leq \gamma_n |\tilde{L}||\tilde{U}|$, we reconsider how $\tilde{L}$ and $\tilde{U}$ are calculated.

For $k = 1, \ldots, n$,

- We derive Δa_{kr}, where $k \leq r$ (i.e., entries on or below the diagonal in ΔA), by calculating the kth row of U: for $r = k, \ldots, n$,

$$u_{kr} = a_{kr} - \sum_{j<k} \ell_{kj} u_{jr}.$$

- Applying Lemma 7.2, we derive Δa_{kr}, $k \leq r$.

$$\tilde{u}_{kr}(1 + \theta_{k,r,k}) = a_{kr} - \sum_{j<k} \tilde{\ell}_{kj}\tilde{u}_{jr}(1 + \theta_{k,r,j})$$

where $|\theta_{k,r,j}| \leq \gamma_{k-1}$ (not γ_k: the final division allowed by Lemma 7.2 is by 1).

$$\sum_{j \leq k} \tilde{\ell}_{kj}\tilde{u}_{jr}(1 + \theta_{k,r,j}) = a_{kr}$$

$$\Delta a_{kr} = -\sum_{j \leq k} \theta_{k,r,j}\tilde{\ell}_{kj}\tilde{u}_{jr}$$

$$|\Delta a_{kr}| \leq \sum_{j \leq k} \gamma_{k-1}|\tilde{\ell}_{kj}||\tilde{u}_{jr}|$$

- Then we get Δa_{ik}, $i > k$, i.e., entries below the diagonal in ΔA, by calculating and the kth column of L: for $i = k+1, \ldots, n$,

$$\ell_{ik} = \frac{a_{ik} - \sum_{j<k} \ell_{ij} u_{jk}}{u_{kk}}.$$

$$\tilde{\ell}_{ik}\tilde{u}_{kk}(1+\phi_{k,i,k}) = a_{ik} - \sum_{j<k} \tilde{\ell}_{ij}\tilde{u}_{j,k}(1+\phi_{i,j,k})$$

where $|\phi_{i,j,k}| \leq \gamma_k$. After some algebra, we get

$$|\Delta a_{ik}| \leq \gamma_k \sum_{j \leq k} |\ell_{ij}||u_{jk}|. \quad \blacksquare$$

8 Strassen's method of multiplying matrices

This is taken from AHU DACA.

Strassen's matrix multiplication algorithm is based on the observation that all 4 entries in the product matrix

$$\begin{bmatrix} c_{11} & c_{12} \\ c_{21} & c_{22} \end{bmatrix} = \begin{bmatrix} a_{11} & a_{12} \\ a_{21} & a_{22} \end{bmatrix} \begin{bmatrix} b_{11} & b_{12} \\ b_{21} & b_{22} \end{bmatrix}$$

are linear combinations of the following bilinear expressions

$$\begin{aligned} m_1 &= (a_{12} - a_{22})(b_{21} + b_{22}) \\ m_2 &= (a_{11} + a_{22})(b_{11} + b_{22}) \\ m_3 &= (a_{11} - a_{21})(b_{11} + b_{12}) \\ m_4 &= (a_{11} + a_{12})b_{22} \\ m_5 &= a_{11}(b_{12} - b_{22}) \\ m_6 &= a_{22}(b_{21} - b_{11}) \\ m_7 &= (a_{21} + a_{22})b_{11} \end{aligned}$$

Namely

$$\begin{aligned} c_{11} &= m_1 + m_2 - m_4 + m_6 \\ c_{12} &= m_4 + m_5 \\ c_{21} &= m_6 + m_7 \\ c_{22} &= m_2 - m_3 + m_5 - m_7. \end{aligned}$$

A table will show at a glance the correctness of these formulas.

	m_1	m_2	m_3	m_4	m_5	m_6	m_7
$a_{11}b_{11}$		+	+				
$a_{11}b_{12}$			+		+		
$a_{11}b_{21}$							
$a_{11}b_{22}$		+		+	−		
$a_{12}b_{11}$							
$a_{12}b_{12}$							
$a_{12}b_{21}$	+						
$a_{12}b_{22}$	+			+			
$a_{21}b_{11}$			−				+
$a_{21}b_{12}$			−				
$a_{21}b_{21}$							
$a_{21}b_{22}$							
$a_{22}b_{11}$		+				−	+
$a_{22}b_{12}$							
$a_{22}b_{21}$	−					+	
$a_{22}b_{22}$	−	+					

There are 7 multiplications and 18 additions.

Furthermore, commutativity is not assumed, so the matrix entries can be themselves matrices.

Let $n = 2^k$ and $T(n)$ be the cost of multiplying two $n \times n$ matrices using this method recursively. Note that the cost of adding two $n \times n$ matrices is $O(n^2)$, which explains the last term in the next equation.

$$T(2^{k+1}) = 7T(2^k) + 18(2^{2k})$$

Let $s(k) = T(2^k)$.

$$s(k+1) - 7s(k) = 18(4^k)$$

Let $u_k = s(k)/7^k$

$$7^{k+1}(u_{k+1} - u_k) = 18(4^k)$$

$$u_k - u_0 = \frac{18}{7} \sum_{r=0}^{k-1} \left(\frac{4}{7}\right)^r .$$

$$u_k < u_0 + 6$$

$$u_k \quad \text{is} \quad O(1)$$

$$s_k \quad \text{is} \quad O(7^k)$$

$$T(n) \quad \text{is} \quad O(n^{\log_2 7}).$$

(This bound has been improved; I'm not sure what it stands at nowadays.)

9 Miller's analysis

This comes from 'Computational complexity and numerical stability,' by Webb Miller, *SIAM J. Computing* **4:2** (June 1975), 97–107. Also, 'Remarks on the complexity of roundoff analysis,' *Computing* **12**, (1974), 149–161, by the same author.

9.1 Error propagation in straight-line programs

This is from the 1974 paper.

A straight-line program is a sequence of instructions of the form

$$Z \leftarrow X \# Y$$

where Z is a variable, and either X or Y is (so we can have constant coefficients), and the operation is either $+$, $-$, $/$, or $\times$. If X (or Y) is a variable which has not been on the left-hand-side of any earlier instruction, then it is an *input variable*. Evaluation is in floating-point arithmetic.

Let δ_j be the relative error associated with the j-th instruction. From the IEEE guarantee, for all j,

$$|\delta_j| \leq \epsilon_{\text{mach}}.$$

Initial values to the input variables should be specified by some vector $\vec{d}$. The exact value of Z is a rational function of $\vec{d}$ and the computed value is a rational function depending also on the relative errors

$$\tilde{Z} = R_Z(\vec{d}, \vec{\delta}).$$

(9.1) Lemma *Rounding error in Z*

$$|Z(\vec{d}) - \tilde{Z}(\vec{d})| = |Z(\vec{d}) - R_Z(\vec{d}, \vec{\delta})| \approx \left| \sum_j \delta_j \frac{\partial R}{\partial \delta_j}(\vec{d}, \vec{0}) \right| \approx$$

$$\epsilon_{mach} \sum_j \left| \frac{\partial R}{\partial \delta_j}(\vec{d}, \vec{0}) \right|$$

(Proof: mean-value-theorem.) ∎

(9.2) Lemma *The effect of perturbing $\vec{d}$ to $\vec{d}'$ (with ϵ_{mach} tolerance: $\Delta d_i = d_i(1 + \pi_i)$) is*

$$|Z(\vec{d}) - Z(\vec{d} + \Delta\vec{d})| \approx \left| \sum_i d_i \pi_i \frac{\partial Z}{\partial d_i}(\vec{d}) \right| \approx \epsilon_{mach} \sum_i \left| d_i \frac{\partial Z}{\partial d_i}(\vec{d}) \right|. \quad \blacksquare$$

Backward error analysis means that it is possible to perturb the data so that

$$\tilde{Z}(\vec{d}) = Z(\vec{d} + \Delta\vec{d})$$

This means that, given $\vec{\delta}$,

$$\sum_i d_i \left. \frac{\partial Z}{\partial d_i} \right|_{\vec{d}} \pi_i = \sum_j \left. \frac{\partial R_Z}{\partial \delta_j} \right|_{\vec{d},\vec{0}} \delta_j$$

has a solution $\vec{\pi}$, which can be expressed in a rough and ready way:

$$\sum_j \left| \frac{\partial R_Z}{\partial \delta_j} \right| \quad \text{is} \quad O\left(\sum_i \left| \left(\frac{\partial Z}{\partial d_i} \right) d_i \right| \right). \tag{9.3}$$

9.2 Polynomial programs

A *polynomial program* is a straight-line program without divisions.

A 'defined variable' is one which occurs exactly once as a LHS.

Division is not allowed. (Hence 'polynomial.' Obviously every variable can be expressed as a polynomial in the initial variables and constants, with integer coefficients if the constants are given the same status as initial variables.)

We can write

$$V(\vec{d})$$

for V as a polynomial in the initial data, with no constant term and with integer coefficients.

Suppose V is a defined variable. If we omit the statement

$$V \leftarrow \dots$$

from the program, then the other variables are polynomials in the initial variable *and* V.

(9.4) Definition

$$\Delta Z_V(\vec{d}) = \left(\frac{\partial Z}{\partial V}|_{(\vec{d},V(\vec{d}))} \right) V(\vec{d})$$

This polynomial measures the sensitivity of $Z(\vec{d}$ to errors in the production of $V(d)$. For a relative error $(1+\delta)$ produces (to first order, according to Taylor's Series) of $\delta \Delta Z_V(\vec{d})$.

Put another way, if δ is the rounding error incurred in computing V, then

$$\Delta Z_V = \frac{\partial Z}{\partial \delta}$$

Example. $\vec{d} = (a_1, a_2, b_1, b_2)$. Assuming commutative multiplication.

$$\begin{gathered} T \leftarrow a_1 \times a_2 \\ U \leftarrow b_1 \times b_2 \\ V \leftarrow a_1 + b_2 \\ W \leftarrow a_2 + b_1 \\ X \leftarrow V \times W \quad (a_1a_2 + a_1b_1 + b_2a_2 + b_2b_1) \\ Y = X - T \quad (a_1b_1 + b_2a_2 + b_2b_1) \\ Z = Y - U \quad (a_1b_1 + b_2a_2) \end{gathered}$$

$$\begin{gathered} Z(\vec{d}) = a_1b_1 + a_2b_2 \\ Z(\vec{d}, W) = (a_1 + b_2)W - a_1a_2 - b_1b_2 \\ \frac{\partial Z}{\partial W)}(\vec{d}, W) = a_1 + b_2 \\ \Delta Z_W(\vec{d}) = (a_1 + b_2)(a_2 + b_1) \end{gathered}$$

(9.5) Definition (tracing back). *Let Z be a* defined *variable of a polynomial program. If Z is defined by an addition or subtraction,*

- *mark the instruction defining Z.*
- *Recursively, if an instruction*

$$V \leftarrow X \# Y, \quad \text{where } \# \text{ is not multiplication,}$$

is marked, and X is defined by $\pm$, mark that instruction; likewise Y.

Delete the unmarked instructions.

$I(Z)$ is defined as the initial (i.e., undefined) variables in the reduced program. The initial (undefined) variables of the reduced programs are variables which were initial or defined by multiplication. On the other hand, if Z is defined by a multiplication, then $I(Z) = \{Z\}$.

Z is an integer linear combination of variables in $I(Z)$:

$$Z(\vec{d}) = \sum_{U \in I(Z)} i_Z(U) \cdot U(\vec{d})$$

(9.6) Theorem *If $V \in I(Z)$ is defined by a multiplication and $i_Z(V) \neq 0$, then*

$$\frac{\partial Z}{\partial V}|_{\vec{d}, V(\vec{d})} \not\equiv 0$$

Proof.

$$\frac{\partial Z}{\partial V} = \sum_{U \in I(Z)} i_Z(U) \frac{\partial U}{\partial V}$$

Consider the terms in this sum. If $U = V$, the term is $i_Z(V)$, a nonzero integer. If U is an initial variable different from V then $\partial U / \partial V = 0$. If U is defined by multiplication, $U \leftarrow X \times Y$, then

$$\frac{\partial U}{\partial V} = X \frac{\partial Y}{\partial V} + Y \frac{\partial X}{\partial V}$$

and it contains no constant term. Hence $\partial Z / \partial V$ contains a nonzero constant term $i_Z(V)$ and is nonzero. ∎

9.3 Bilinear forms

$$B(\vec{a}, \vec{b}) = \sum_{i=1}^{m} \sum_{j=1}^{n} \sigma_{ij} a_i b_j$$

is a bilinear form in the data $\vec{a}, \vec{b}$.

(9.7) Definition

$$|(\vec{a}, \vec{b})|_B = \max\{|a_i||b_j| : \sigma_{ij} \neq 0\}$$

A program (polynomial program) to evaluate B is

- *Brent stable* if for each defined variable V

$$\Delta B_V(\vec{a},\vec{b}) \text{ is } O(|\vec{a}||\vec{b}|)$$

- *Restricted Brent stability* is

$$\Delta B_V(\vec{a},\vec{b}) \text{ is } O(|\vec{a},\vec{b}|_B)$$

- Weak stability:

$$\Delta B_V(\vec{a},\vec{b}) \quad \text{is} \quad O\left(|\vec{a}|\sum_{i=1}^{m}\left|\frac{\partial B}{\partial a_i}(\vec{a},\vec{b})\right| + |\vec{b}|\sum_{j=1}^{n}\left|\frac{\partial B}{\partial b_j}(\vec{a},\vec{b})\right|\right)$$

- Strong stability:

$$\Delta B_V(\vec{a},\vec{b}) \quad \text{is} \quad O\left(\sum_{i=1}^{m}\left|a_i\,\frac{\partial B}{\partial a_i}(\vec{a},\vec{b})\right| + \sum_{j=1}^{n}\left|b_j\frac{\partial B}{\partial b_j}(\vec{a},\vec{b})\right|\right)$$

Fact. Strong stability is equivalent or close to backward stability. Meaning: a program is backward stable if rounding errors can be *'blamed'* on bad data. It matches equation 9.3

(9.8) Proposition *The above classification is from weakest to strongest.* ∎

(9.9) Definition B *is a* permutation bilinear form *if for each* j, σ_{ij} *is nonzero for at most one i, and vice-versa.*

(9.10) Lemma *For permutation bilinear forms, restricted Brent stability is equivalent to strong stability.*

Proof. It is enough to prove

$$|(\vec{a},\vec{b})|_B \quad \text{is} \quad O\left(\sum_i\left|a_i\frac{\partial B}{\partial a_i}\right| + \sum_j\left|b_j\frac{\partial B}{\partial b_j}\right|\right)$$

Suppose

$$|(\vec{a},\vec{b})|_B = |a_i||b_k|$$

Let

$$M = \max\{\left|\frac{1}{\sigma_{rs}}\right| : \ \sigma_{rs} \neq 0\}.$$

$$\begin{aligned} |a_i||b_k| &\leq M|a_i\sigma_{ik}b_k| \\ &= M|a_i\sum_j \sigma_{ij}b_j| \end{aligned}$$

because B is a permutation bilinear form.

$$= M \left| a_i \frac{\partial B}{\partial a_i} \right|$$
$$\leq M \sum_i \left| a_i \frac{\partial B}{\partial a_i} \right|$$

and the last expression is

$$O\left(\sum_i \left| a_i \frac{\partial B}{\partial a_i} \right| + \sum_j \left| b_j \frac{\partial B}{\partial b_j} \right| \right). \quad \blacksquare$$

(9.11) Theorem *Suppose V is a variable in a program for B and $i_B(V) \neq 0$ and V is defined by a multiplication. If B has Brent stability then the definition of V must be equivalent to*

$$V \leftarrow L_1(\vec{a}) * L_2(\vec{b})$$
$$\textit{or} \quad V \leftarrow L_1(\vec{b}) * L_2(\vec{a})$$

where L_1 and L_2 are linear (with coefficients in $\mathbb{Z}$).[7]

Proof. First claim that $\Delta B_V(\vec{a}, \vec{b})$ is bilinear in which each term is of the form $\alpha a_i b_j$.

Suppose otherwise: we shall deduce that

$$\Delta B_V(\vec{a}, \vec{b}) \quad \text{is not} \quad O(|\vec{a}||\vec{b}|).$$

By Theorem 9.6, $\Delta B_V \neq 0$.

If ΔB_V has a term involving none of the b_j

$$\alpha a_1^{p_1} \cdots$$

($\alpha \neq 0$). Allow $\vec{a}$ to vary while leaving $\vec{b} = \vec{0}$, we get a nonzero function which can not, of course, be $O(|\vec{a}||\vec{0}|)$.

Similarly, every term in ΔB_V involves some of the a_i.

The only case left is where ΔB_V has a term of degree ≥ 3. Fix $\vec{a}, \vec{b}$ so that $\Delta B_V(\vec{a}, \vec{b}) \neq 0$. Write $\Delta_V(\vec{a}, \vec{b})$ as a sum of homogeneous polynomials of decreasing homogeneous degree:

$$\Delta_V(\vec{a}, \vec{b}) = \sum h_i(\vec{a}, \vec{b}).$$

Note that $\deg(h_1) \geq 3$. Fix $\vec{a}, \vec{b}$ so that $h_1(\vec{a}, \vec{b}) \neq 0$. Let

$$p(t) = \Delta_V(t\vec{a}, t\vec{b}) = \sum h_i(t\vec{a}, t\vec{b})$$

This is a polynomial in t of degree $\deg(h_1)$ which is at least 3. On the other hand

$$|t\vec{a}||t\vec{b}|$$

[7] The order of multiplication is unimportant. We are concerned with commutative floating-point operations, not matrix operations.

has degree 2 in t and can't dominate $|p(t)|$.

This proves the claim:

$$\frac{\partial B}{\partial V} V$$

(as a polynomial in $\vec{a}, \vec{b}$) is of the form

$$\sum \rho_{ij} a_i b_j$$

where $\rho_{ij} \in \mathbb{Z}$.

Every variable in the program is a polynomial in $\vec{a}, \vec{b}$ whose constant term is zero (this is easily shown by induction). V is defined by multiplication, so $V = X \times Y$ for some other variables X, Y, and $\partial B/\partial V$ is another integer polynomial, $p(\vec{a}, \vec{b})$, but it can have nonzero constant term. Also $\partial B/\partial V \neq 0$, so none of these factors is zero.

Thus

$$p(\vec{a}, \vec{b}) X(\vec{a}, \vec{b}) Y(\vec{a}, \vec{b}) = \sum \rho_{ij} a_i b_j \neq 0.$$

This is possible only if p is constant and X and Y are linear, where X involves only the a_i and Y the b_j, or vice-versa. Thus $V = L_1(\vec{a}) L_2(\vec{b})$ as claimed. ∎

(9.12) Theorem *Let V be defined by a multiplication with $i_B(V) \neq 0$, and assume the program is Brent stable, so we can write*

$$V(\vec{a}, \vec{b}) = \sum_{ij} \eta_{ij} a_i b_j$$

Suppose B has restricted Brent stability. For any i, j such that $\eta_{ij} \neq 0$, $\sigma_{ij} \neq 0$.

Proof. The condition of restricted Brent stability is

$$\Delta B_V(\vec{a}, \vec{b}) \quad \text{is} \quad O(|(\vec{a}, \vec{b})|_B)$$

so if $a_i b_j$ appears in ΔB_V then it appears in B (since if $\sigma_{ij} = 0$ then the left-hand side must be zero). ∎

(9.13) Theorem *Suppose that B is a* permutation *bilinear form and the program has restricted Brent stability. Then if $\sigma_{ij} \neq 0$, there is a multiplication in the program of the form $(\alpha a_i) * (\beta b_j)$ (or $(\alpha b_j) * (\beta a_j)$).*

Proof. From the above theorem, $a_i b_j$ occurs in some V where V is defined by multiplication, with $i_B(V) \neq 0$. Suppose

$$V = \left(\sum \alpha_r a_r \right) \left(\sum \beta_s b_s \right).$$

By the above theorem, all terms $a_r b_s$, where $\alpha_r \neq 0$ and $\beta_s \neq 0$, occur in B. Since B is a permutation bilinear form, there is only one nonzero α_r and one nonzero β_s:

$$V = \alpha_r \beta_s a_r b_s. \quad ∎$$

9.4 Matrix multiplication

A polynomial program to evaluate a system of s bilinear forms B_ℓ has *simultaneous Brent stability* if each B_ℓ has Brent stability. Similarly for restricted Brent stability, etcetera.

(9.14) Lemma *The n^2 entries in a matrix product are permutation bilinear forms in the matrix entries.* ∎

(9.15) Corollary *Every polynomial program for matrix multiplication which has simultaneous restricted Brent stability has at least n^3 multiplications.*

Proof. It must calculate n^3 products $\alpha a_{ij} b_{jk}$. ∎

(9.16) Corollary *Strassen's method is restricted Brent unstable.* ∎

10 Linear programming

The phrase arose in connection with production planning in the USAF in 1947. 'Programming' means planning. A linear programming problem is as follows:

> Given a closed convex nonnegative polyhedron P in $\mathbb{R}^d$, and a linear function $f : \mathbb{R}^d \to \mathbb{R}$, to return a point maximising (or perhaps minimising) $f|P$, or to determine that no such point exists.

To keep things simple, we assume that P is a (nonempty) polytope, in which case the point necessarily exists.

A point q in a convex set S is *extremal* if for any points $p, r \in S$, if $q = (p+r)/2$ then $p = q = r$. For polytopes, one should think of a *corner* or *vertex*.

(10.1) Lemma *Assuming P is nonempty and bounded, the linear programming problem must have an extremal solution.*

That is, the maximum or minimum is always achieved at a vertex.

Proof. Let m be the maximum value of $f|P$, and let $\Pi = f^{-1}(m)$. The set $\Pi \cap P$ is a nonempty polytope. Suppose $q \in \Pi = (p+r)/2$ where $p, r \in P$. Then both p and r are in Π, since if $f(p) < m$ then $f(r) > m$ and vice-versa. Therefore every extreme point of $\Pi \cap P$ is an extreme point of P; and (by induction on affine dimension) $\Pi \cap P$ contains an extreme point. **Q.E.D.**

A linear programming problem will be formulated as follows:

- Maximise $C^T X$ subject to
- $X \geq O \in \mathbb{R}^d$ and
- $AX \leq B$, $\qquad (*)$

where A is an $m \times d$ matrix, $C \in \mathbb{R}^d$, and $B \in \mathbb{R}^m$.

The function $f(X) = C^T X$ is called the *objective function* or **cost** function. The polytope P defined by the constraints on X is called the **Feasible region** and points in P are called *feasible solution.* An extreme point of P is called a **basic feasible solution** (it will become clear why 'basic' is mentioned).

The first step is to lift the problem into an isomorphic polytope in $\mathbb{R}^{d+m}$. Each of the rows of the matrix inequality (*) is converted to an inequality by adding what is called a **slack variable**:

$$\sum_{j=0}^{j=d} a_{ij}x_j + x_{d+i} = b_i.$$

There is an affine map from $\mathbb{R}^d$ to $\mathbb{R}^{d+m}$:

$$\phi\colon X \mapsto \begin{bmatrix} X \\ B - AX \end{bmatrix}.$$

Given a point in $\mathbb{R}^{d+m}$, let X be its first d coordinates and U its last m. Then $[X^T \;\; U^T]^T$ is the image of a feasible point, i.e., it belongs to ϕP, if and only if $U \geq O$ and $AX + U = B$.

We now revise the formulation of the linear programming problem, lifting it to $d+m$ dimensions. Let $n = d + m$.

- Maximise $C^T X$ subject to
- $AX + U = B$ and
- $X \geq O, U \geq O$.

The map

$$\mathbb{R}^n \to \mathbb{R}^m; \quad \begin{bmatrix} X \\ U \end{bmatrix} \mapsto AX + U$$

has rank m (i.e., is surjective). The associated $m \times n$ matrix is $[A \;\; I]$. In general, given an $m \times n$ matrix M of rank m, suppose that m linearly independent columns have been selected (these columns, and the corresponding variables, are called the *basis*). Then

- All other columns of M can be expressed uniquely in terms of the basis columns;
- The set of solutions to $MY = B$ ($Y \in \mathbb{R}^n$) can be parametrised by the $n - m$ variables *not* appearing in the basis.

In the context of the simplex method, an extreme point of the feasible polyhedron P (or rather $\phi P \subseteq \mathbb{R}^n$) is at the intersection of d or more bounding faces. It is defined by choosing the d nonbasic variables, setting them to zero, and solving for the other variables (assuming, that is, that the corresponding columns are linearly independent) to satisfy the equation $MY = B$. It is easiest to update M and B so that the basis columns form a permutation of I.

Of course, an arbitrary choice of nonbasic variables could yield a point not in P, because some of the basic variables could become negative.

The simplex method looks for an optimal extreme point on P, beginning with some extreme point (a 'basic feasible solution.') Given an extreme point— d of the variables, nonbasic, set to zero; intersection of d hyperplanes in $\mathbb{R}^n$ with the affine space $MY = B$. The set of solutions to $MY = B$ can be parametrised by the nonbasic variables; likewise the cost function. One maintains the cost function so that it is indeed parameterised by the nonbasic variables, so that its coefficients are zero for the basic variables. If all the cost coefficients are nonpositive then the given solution is optimal. Otherwise, choose a nonbasic variable x_c whose cost coefficient is maximal. One is going to locate an adjacent point on the boundary of P by allowing this variable to become positive. The remaining nonbasic variables remain nonzero. Therefore, one is seeking to increase the cost value in a space represented by the intersection of $d - 1$ hyperplanes — a line I; it lies on an edge of the polyhedron P.

Now, assuming M and B are maintained so that the basis columns form a permutation of the identity matrix I, each row of the system of equations can be written in the form

$$x_k + \sum_{j \in N} a'_{ij} x_j = b'_i,$$

where x_k is the basic variable in the i-th row and N is the subsequence of nonbasic indices. Interpreting the i-th row as describing a bounding hyperplane H_i of P, setting x_k to zero and adjusting x_c (if $a'_{ic} \neq 0$) amounts to finding where I intersects H_i. Whichever row yields the smallest nonnegative value of x_c defines the basic variable x_k which is exchanged with x_c.

Adjust M and B and the cost coefficients, so that the c-th column becomes one of the columns of I, and x_c gets zero cost coefficient (and x_k's changes); and continue.

Example. Maximise $5x_1 + 4x_2 + 3x_3$ subject to

$$\begin{bmatrix} 2 & 3 & 1 \\ 4 & 1 & 2 \\ 3 & 4 & 2 \end{bmatrix} \leq \begin{bmatrix} 5 \\ 11 \\ 8 \end{bmatrix}$$

and $x_1, x_2, x_3 \geq 0$. For this example O is a feasible (extreme) point, and we start there. Introducing slack variables x_4, x_5, x_6, the system is summarised by the array

$$\begin{array}{cccccccc} 2 & 3 & 1 & 1 & 0 & 0 & 5 \\ 4 & 1 & 2 & 0 & 1 & 0 & 11 \\ 3 & 4 & 2 & 0 & 0 & 1 & 8 \\ 5 & 4 & 3 & 0 & 0 & 0 & 0 \end{array}$$

(the bottom right-hand corner will hold the negative of the constant part of the cost function). The image of O in $\mathbb{R}^6$ is $[0, 0, 0, 5, 11, 8]^T$. Since x_1 has maximal cost coefficient, it is to be increased. The three rows put a limit on x_1 of $2.5, 2.75, 2.67$ respectively on x_1, so the first row defines the new bounding hyperplane forming a corner with the edge $x_2 = x_3 = 0$ of the polyhedron. To convert the array as required, the first column should become $[100]^T$. Divide the first row by 2, and use it to clear the other three entries in the first column (including the cost coefficient), yielding

$$\begin{array}{cccccccc} 1 & 1.5 & .5 & .5 & 0 & 0 & 2.5 \\ 0 & -5 & 0 & -2 & 1 & 0 & 1 \\ 0 & -.5 & .5 & -1.5 & 0 & 1 & .5 \\ 0 & -3.5 & .5 & -2.5 & 0 & 0 & -12.5 \end{array}$$

The nonbasic variables are x_2, x_3, x_4, and the new extreme point is $[2.5, 0, 0, 0, 1, .5]^T$. From the cost coefficients, x_3 should be increased. The constraints imposed put upper limits of 5, none, and 1 respectively, so the third row defines the new corner and x_6 is traded for x_3 in the basis. The new basic variables are x_1, x_3, x_5. Revising the array, we get

$$\begin{array}{rrrrrrr} 1 & 2 & 0 & 2 & 0 & -1 & 2 \\ 0 & -5 & 0 & -2 & 1 & 0 & 1 \\ 0 & -1 & 1 & -3 & 0 & 2 & 1 \\ 0 & -3 & 0 & -1 & 0 & -1 & -13 \end{array}$$

The corresponding point is $[2, 0, 1, 0, 1, 0]^T$, projecting down to $[2, 0, 1]^T$ in P. The value of the cost function is 13 at this point. All the cost coefficients are now nonpositive, so this point is optimal.

Certain problems with the simplex method.

Initial feasible point. To locate an initial (extreme) feasible point, when O is not feasible (i.e., some coordinate of B is negative), the following trick can be used: Minimise x_0 subject to $x_0, X \geq 0$, $AX \leq B + x_0$ (meaning of right hand side: $B + [x_0, x_0, \ldots x_0]^T$). Choosing the smallest x_0 making $B + x_0$ nonnegative in all its coordinates, and $X = O$, we have an initial point for this problem; if the optimal solution yields $x_0 = 0$ then it is a starting point for the original problem; otherwise there is no solution to the original problem, or at least no basic solution (P is either empty or has too few faces to have any corners— in this case it is unbounded.)

Degeneracy. This can be problematical: it means that some extreme points are at the intersection of more than d faces of P. We ignore this problem. In other words, we assume that P is a simple polytope.

Efficiency. Klee and Minty gave a linear programming problem in $\mathbb{R}^d$ in which the feasible region P is a (slightly perturbed) d-dimensional cube ($2d$ inequalities), and the cost function so chosen so that the simplex algorithm visits all 2^d corners of the polyhedron. Hence the runtime can be exponential in the number of inequalities.

Duality. The duality theorem will be important for Karmakar's algorithm. The idea is that to every linear programming problem there is a 'dual problem.'

(10.2) Definition *The* **Dual problem** *to*

Maximise $C^T X$ subject to $AX \leq B$ (and $X \geq O$)

is

Minimise $Y^T B$ subject to $Y^T A \geq C^T$ and $Y \geq O$.

A point Y in the feasible region for the dual problem is called dual feasible. *The first problem is sometimes called the* primal problem.

(10.3) Theorem (Duality Theorem). *Given a primal problem in $\mathbb{R}^d$ and its dual in $\mathbb{R}^m$,* (i) *if $X \in \mathbb{R}^d$ is feasible and $Y \in \mathbb{R}^m$ is dual feasible, then $C^T X \leq Y^T B$, and* (ii) *these values are equal for an optimal pair X, Y.*

Proof. (i) $C^T X \leq Y^T AX \leq Y^T B$.

(ii) Apply the simplex method to the primal problem, by introducing slack variables and so forth. Let $U \in \mathbb{R}^m$ represent the slack variables. Write the ultimate cost function (on the image points in $\mathbb{R}^{d+m}$) as

$$C_0^T X + C_1^T U + Z_0.$$

Since an optimum has been located, all the cost coefficients are negative or zero.

Let X_0 be the optimal (extremal) solution found, and let U_0 be the corresponding slack values.

$$C^T X_0 = C_0^T X_0 + C_1^T U_0 + Z_0.$$

The cost coefficients are zero for variables in the basis, and variables not in the basis are zero, so $C^T X_0 = Z_0$.

Let $Y^0 = -C_1$. First write the relation between the cost coefficients:

$$C^T X \equiv Z_0 + [C_0^T C_1^T](\phi X),$$

i.e.,

$$C^T X \equiv Z_0 + C_0^T X + C_1^T (B - AX).$$

Hence we have the following identity (valid for all $X \in \mathbb{R}^d$):

$$(C^T - C_0^T - Y_0^T A)X \equiv Z_0 - Y_0^T B.$$

The left-hand side is linear and the right-hand side is constant, hence both sides are identically zero. From the RHS, $Y_0^T B = Z_0$. From the LHS, $Y_0^T A - C^T = -C_0^T$. Since C_0 is nonpositive, Y_0 is dual feasible. Since X_0 is the given extremal solution, $C^T X_0 = Z_0$. Hence $Y_0^T B = C^T X_0$. **Q.E.D.**

We could solve the primal problem, therefore, by solving the following 'primal-dual' problem:

Minimise $Y^T B - C^T X$ subject to $X, Y \geq O$, $AX \leq B$, and $Y^T A \geq C^T$.

The advantage of this (which will be important for Karmakar's algorithm) is that the minimum, if it exists, is 0.

11 Karmakar's algorithm

This is a linear programming algorithm which, unlike the simplex method, searches for an optimum from the interior of the feasible region; roughly speaking, it moves in the direction of optimum by following the path of maximum gradient.

The method uses projective transformations rather than linear transformations. Recall the standard model of the projective plane: the Euclidean plane is identified with the plane $z = 1$ in 3 dimensions; a point $(a, b, 1)$ on this plane is identified with the line $\{(ta, tb, t) \colon t \neq 0 \in \mathbb{R}\}$ through the origin (omitting the origin). The Euclidean plane is extended to include 'points at infinity' which correspond to lines in the plane $z = 0$. In this geometric model, any two lines intersect in a unique point whether or not they are parallel. Parallel lines intersect in a point at infinity. A map from projective n-space to projective m-space is called *projective* if it can be represented by a linear map from $\mathbb{R}^{n+1}$ to $\mathbb{R}^{m+1}$.

The STANDARD n-SIMPLEX Δ_n is the set

$$\{(x_1, \ldots, x_{n+1}) \in \mathbb{R}^{n+1} \colon x_j \geq 0, \sum x_j = 1\}.$$

There is a projective map from the projective plane to projective 3-space which takes the (closure of the) positive quadrant bijectively onto the standard 2-simplex: $(a, b, 1) \mapsto (ca, cb, c)$ where $c =$

$1/(1+a+b)$. (More precisely, the projective point $\{(ta, tb, t) : t \neq 0 \in \mathbb{R}\}$ maps to the intersection of this line with the 2-simplex. Points at infinity map to points along the bottom side of the 2-simplex.)

The CENTRE a_0 of Δ_n is the point $(1/(n+1), \ldots 1/(n+1))$. The *interior* of Δ_n is the set of all points in it with strictly positive coordinates. Given a point $a = (a_1, \ldots a_{n+1})$ in the interior of Δ_n, there is a projective map taking Δ_n bijectively onto itself and taking a to a_0:

$$x = (x_1, \ldots x_{n+1}) \mapsto x' = (x'_1, \ldots x'_{n+1}),$$

where

$$x'_j = \frac{x_j/a_j}{\sum(x_i/a_i)}.$$

If we consider the map

$$x' \mapsto x; x_j = \frac{x'_j a_j}{\sum x'_i a_i}$$

then the composite maps takes x to $x/\sum x_j$. The restriction of the map to the hyperplane $\sum x_j = 1$ is bijective; it carries Δ_n bijectively onto itself and a to a_0.

Karmakar's algorithm begins with a sequence of transformations, ultimately yielding an optimisation problem of the form

minimise $c^T x$

subject to $x \in \Delta_n$ and $Ax = O$,

where the minimum (if it exists) is 0, and a_0 is a feasible point.

We start with the usual linear programming problem:

Maximise $c^T x$ subject to $x \geq 0$ and $Ax \leq b$.

The corresponding primal-dual problem is

Minimise $b^T y - c^T x$ subject to $Ax \leq b$, $A^T y \geq c$, and $x, y \geq O$.

This problem has the property that the minimum, if it exists, furnishes a minimum to the primal problem, and also the minimum is zero. Hence we can relabel the variables and matrices involved and assume the usual formulation, except that now the minimum, if it exists, is 0.

Next add slack variables in the usual way, so that the problem has the form

Minimise $c^T x$ subject to $x \geq O$ and $Ax = b$.

The minimum is actually zero. Suppose that n is the number of variables in this problem.

Next take any point $a \in \mathbb{R}^n$ where a has strictly positive coordinates, and consider the problem

Minimise λ subject to $Ax - \lambda(Aa - b) = b$, $c^T x - \lambda c^T a = 0$, and $x, \lambda \geq 0$.

If the previous problem had an optimal solution x achieving the zero minimum cost function, then by setting $\lambda = 0$ one gets an optimal solution to the last problem; conversely, an optimal solution of zero to the last problem furnishes an optimal solution to the one before. The last problem has the advantage that a feasible initial point is available: namely, $x = a, \lambda = 1$. So the formulation can be expressed as follows:

Minimise $c^T x$ subject to $Ax = b$ and $x \geq 0$. If the minimum exists it is zero; and a feasible point a with strictly positive coordinates is known.

Then map projective n-space to projective $n+1$-space by a projective map which takes the positive orthant $x \geq O$ (or its closure in projective n-space) bijectively onto Δ_n, and a to the centre a_0 of Δ_n; if $(a_1, \ldots a_n)$ are the coordinates of a, then $x \mapsto x'$ where

$$x'_i = \frac{x_i/a_i}{1 + \sum(x_j/a_j)} (1 \leq i \leq n),$$

and $x'_{n+1} = \frac{1}{1+\sum(x_j/a_j)}$. Note that $\sum x'_j = 1$, and this maps the positive orthant to Δ_n. Also, $a \mapsto a_0$. The map is easily inverted:

$$x_i = \frac{a_i x'_i}{x'_{n+1}}.$$

Write a_{ij} for the components of A, b_i for those of b, c_j for those of c. The image of the feasible region consists of all points x' in Δ_n such that

$$\sum a_{ij} a_j x'_j - b_i x'_{n+1} = 0.$$

Also, the cost function can be transformed:

$$c^T x = \sum c_j a_j x'_j / x'_{n+1}.$$

Granted that the minimum cost (if it is attainable) is zero, the factor x'_{n+1} can be ignored, and we can express the problem as follows:

Minimise $c^T x$ subject to $x \in \Delta_n$, $Ax = O$. The minimum cost (if attainable) is 0, and a_0 is a feasible point.

Karmakar's algorithm proceeds iteratively. A single improvement step consist of the following. The radius of the maximum inscribed sphere (centred at a_0) in Δ_n is $\frac{1}{\sqrt{n(n+1)}}$; Let $r = 1/\sqrt{n(n+1)}$. (In principle the algorithm could choose a sphere of any radius less than the maximum possible. The choice of $1/4$ maximum radius avoids some rounding problems.)

Let a be a given interior (all coordinates positive) feasible point. Suppose $a = (a_1, \ldots, a_{n+1})^T$. Let D_a be the diagonal matrix diag$(a_1, \ldots, a_{n+1})$. Let $E = (1, \ldots, 1)^T$ and let a_0 be the centre of Δ_n as before: $a_0 = E/(n+1)$.

Note that the transformation

$$X \mapsto \frac{D_a X}{E^T D_a X}$$

is a projective transformation which we have already seen; it takes a_0 to a and maps Δ_n bijectively onto itself.

The step in Karmakar's algorithm is to replace a by a new feasible interior point a' as follows: minimise

$$C^T D_a Z$$

subject to

$$AD_a Z = O, \quad Z \in \Delta_n, \text{and } |D_a Z - a_0| \leq \alpha r;$$

then let

$$a' = \frac{D_a Z}{E^T D_a Z}.$$

Calculation of the point Z is not hard. Take the transformed cost vector $D_a C$, and project orthogonally it onto the space

$$\{U : AD_a U = O,\ \ E^T U = O\}.$$

This calculation is $O(n^3)$, involving some matrix calculations to produce a projection matrix. Let $\hat{C}$ be the normalisation of the resulting vector. Then $\hat{C}$ is a unit vector parallel to the space $AD_a Z = O, E^T Z = 1$, directed towards maximal increase of the cost within that space. Hence the point desired is

$$Z = a_0 - \alpha r \hat{C}.$$

These revisions continue until the cost value is sufficiently small, at which time an optimal solution is calculated by locating a nearby extremal point on the boundary of the feasible region.

Rationale of Karmakar's algorithm. We invoke without proof two propositions.

(11.1) Proposition *There is a constant $K > 0$ such that if L is the length of the input (total bit complexity) and X_1, X_2 are basic feasible solutions such that $C^T X_1 \neq C^T X_2$, then $|C^T X_1 - C^T X_2| \geq 2^{-KL}$.* ∎

(11.2) Proposition *If a is a feasible solution then one can construct (fairly efficiently) a basic feasible solution with no greater cost.* ∎

Next we define a 'potential function.' For any point X with coordinates x_j, let us write $\prod X$ for $\prod_{j=1}^{n+1} x_j$, and the potential

$$f(X) = \frac{(C^T X)^{n+1}}{\prod X}.$$

(11.3) Theorem *There is a constant $k < 1$ such that if the minimum is zero, and a' is derived from a by a single revision, then*

$$f(a') \leq k f(a).$$

With the aid of this theorem we can prove that Karmakar's algorithm is efficient.

(11.4) Corollary *Karmakar's algorithm terminates after $O(nL)$ revisions, where each revision involves $O(n^3)$ arithmetic operations. Thus the runtime (counting an arithmetic operation as unit time) is $O(Ln^4)$.*

PROOF. The algorithm starts at $a_0 = E/n + 1$. After t revisions it has reached a point a such that the potentials are:

$$f(a) \leq k^t f(a_0).$$

i.e.,

$$\frac{(C^T a)^{n+1}}{\prod a} \leq k^t \frac{(C^T a_0)^{n+1}}{\prod a_0}.$$

Now, $\prod a \leq \prod a_0$ since the arithmetic mean bounds the geometric mean. Therefore

$$C^T a \leq k^{t/(n+1)} C^T a_0.$$

(If the potential has not reached this bound then the desired minimum is not zero and the search can be abandoned.)

We want to drive this down to 2^{-KL}. Note that $C^T a_0$ is the average of the cost coefficients, hence may be bounded by 2^L (L, recall, is the input length). So we want to choose t so that

$$k^{t/(n+1)} 2^L \leq 2^{-KL}$$

So $t = (n+1)(K+1)L/\log_2(1/k)$ will do the trick. This is $O(nL)$.

After this many iterations, $C^T a \leq 2^{-KL}$. Round a to a basic feasible solution b whose cost is no greater. By the first of the above propositions, b must be optimal, since the existence of any lower-cost basic feasible solution would contradict the proposition. ∎

Thorem 11.3 remains to be proved.

(11.5) Lemma *Following the notation used to describe a step in Karmakar's algorithm, if Z is the minimising point, and the true mininum is zero (or ≤ 0) then*

$$C^T D_a Z \leq \frac{1}{n+1}(1 - \frac{\alpha}{n}) C^T a.$$

PROOF Invoking the projective bijection T_a on Δ_n taking a_0 to a, choose a point $u \in \Delta_n$ such that $C^T T_a u \leq 0$ and $A T_a u = O$. (So just choose u so $T_a u$ is optimal.)

The radius R of the sphere circumscribing Δ_n is $\sqrt{n/(n+1)}$. Therefore, since $|u - a_0| \leq R$, $\alpha|u - a_0|/n \leq \alpha r$. Now, $A D_a u = 0$ so $D_a u$ is in the space on which Z was minimised. So

$$C^T D_a Z \leq C^T D_a (a_0 + (\alpha/n)(u - a_0)) \leq C^T D_a a_0 (1 - \alpha/n) = C^T a (1 - \alpha/n)/(n+1). \quad ∎$$

(11.6) Lemma *The minimum of $\prod X$ subject to $|X - a_0| = \alpha r$ and $E^T X = 1$ is $(n+1)^{-(n+1)}(1 - \alpha)(1 + \alpha/(n+1))^n$.*

PROOF. We first consider the 3-dimensional case, with the variables and constraints slightly altered: minimise $(x+c)(y+c)(z+c)$ subject to $x + y + z = A$ and $x^2 + y^2 + z^2 = B$. First we eliminate the possibility that $x = y = z$, since otherwise $x = y = z = A/3$. This point is the point closest to the origin in the plane $x + y + z = A$, so if it is on the sphere $x^2 + y^2 + z^2 = B$ then the plane is tangent to the sphere and the constraints yield a single point.

Next we apply Lagrange multipliers to the minimisation problem:

$$\partial/\partial x : (y+c)(z+c) + \lambda + 2\mu x = 0$$

similarly, taking derivatives with respect to y and z:

$$(x+c)(z+c) + \lambda + 2\mu y = 0$$

$$(x+c)(y+c) + \lambda + 2\mu z = 0.$$

Assuming $x \neq y$, the first two can be solved for λ and μ:

$$(y-x)(z+c) + 2\mu(x-y) = 0,$$

whence $\mu = (z+c)/2$, and therefore

$$(x+c)(z+c) + \lambda + y(z+c) = 0,$$

so $\lambda = -(x+y+c)(z+c)$. Substituting these in the third equation, we get

$$z(z+c) + (x+c)(y+c) - (x+y+c)(z+c) = 0,$$

$$z^2 - (x+y)z + xy = 0.$$

Therefore $z = x$ or $z = y$. Without loss of generality assume $z = y$. Our problem therefore becomes that of minimising

$$(x+c)(y+c)^2$$

subject to $x + 2y = A$ and $x^2 + 2y^2 = B$. (There are only two points satisfying these constraints.)

Write $x = A/3 - 2v$, $y = A/3 + v$. This ensures the first constraint is automatically satisfied. Substituting into the quadratic constraint, we get

$$(A/3 - 2v)^2 + 2(A/3 + v)^2 = B,$$

which simplifies to

$$6v^2 = B - A^2/3.$$

Thus there are two oppositely signed solutions $v = \pm w$ where $w = \sqrt{(3B - A^2)/18}$. Write u for $A/3 + c$. Then the solutions are

$$(u - 2w)(u + w)^2 \text{ and } (u + 2w)(u - w)^2.$$

If we subtract the first from the second and simplify, we get $4w^3$ which is positive. Hence the first is the minimum, and we conclude:

> The three-dimensional problem is minimised when (if necessary permuting the variables) $x < y = z$. Moreover, x, y, and z are unique.

This leads us directly to the solution of the general problem. Suppose that at a minimum the variables are, without loss of generality, in sorted order: $x_1 \leq x_2 \leq x_3 \ldots x_{n+1}$. If $x_1 = x_2$, then there would be a triple $x_i = x_{i+1} < x_{i+2}$. We could fix the other variables and alter these three in accordance with the three-dimensional case, reducing the product. Hence $x_1 < x_2$. If $x_2 < x_{n+1}$ then one could vary x_1, x_2, x_{n+1} and fix the other variables, in accordance with the three-dimensional case, reducing the product.

Hence $x_1 < x_2 = x_3 = \ldots x_{n+1}$. We calculate x_1 and x_2:

$$x_1 + nx_2 = 1; \quad x_2 = (1 - x_1)/n.$$

Substitute this into $\sum(x_j - 1/(n+1))^2 = \alpha^2 r^2$:

$$(x_1 - \frac{1}{n+1})^2 + n(\frac{1}{n(n+1)} - \frac{x_1}{n})^2 = \alpha^2 r^2$$

so $(n+1)/n(x_1 - 1/(n+1))^2 = \alpha^2 r^2 = \alpha^2/(n(n+1))$, bearing in mind what r is. Therefore $x_1 = (1 \pm \alpha)/(n+1)$. Then $x_2 = (1 - x_1)/n = 1/n - 1/(n(n+1)) \mp \alpha/(n(n+1))$, which equals $(1 \mp \alpha/n)/(n+1)$. Since $x_1 < x_2$, the first contains $-\alpha$ and the second $+\alpha$, and we finally get the estimate:

$$\prod X \leq (n+1)^{-(n+1)}(1-\alpha)(1+\frac{\alpha}{n})^n. \quad \blacksquare$$

12 Revision syllabus 2012

Modules 3467 and 3468 will be examined in the same style. General remarks in the 3467 syllabus apply here. Anything on the quizzes is fair game.

TOPICS:

Definitions concerning directed graphs.

'Source deletion' topological sort algorithm.

Depth-first search with crucial reachability property (Lemma 1.22, not proved). DFS method of topological sort.

Strongly connected components (sccs). The SCC algorithm including stack, edge classification, and 'back_ref': characterise what this evaluates to.

Floyd-Warshall and Dijkstra algorithms.

Network flows, augmenting paths, Ford and Fulkerson theorem, breadth-first construction with runtime $O(m^2 n)$ (Edmonds and Karp). Layered networks will not be asked.

Numerical linear algebra: LU-factorisation of a matrix (no pivoting).

Error analysis (IEEE standard) for back substitution and LU factorisation. Example of inaccuracy within the IEEE standard, and how it is consistent with the error analysis.

Strassen's matrix multiplication, runtime analysis, without requiring the exact formulae exploited in Strassen's method.

Miller's analysis: polynomial programs, $\Delta Z_V(\vec{d})$, bilinear forms, Brent stability, restricted Brent stability, and strong stability. Permutation bilinear form. Theorems 9.11, 9.12, 9.13, inaccuracy of Strassen's method.

Linear programming: simplex method, applied; know the Klee-Minty result. Duality theorem.