## Efficient graph data structure (C++)

Vertices are in a node array, arcs are in an arc array. The arcs are sorted so that arcs out of the same vertex are consecutive in the array. Each vertex points to the first arc on its arc list. There is an additional sentinel vertex (last on the list) and an additional sentinel arc (last on the list). The sentinel vertex points to the sentinel arc.

A common operation of flow algorithms is a scan of arcs out of a vertex. Since the arcs are adjacent, we get good locality.

For each arc has a head, capacity (or residual capacity), and a reverse arc pointer. When we increase flow on an arc, we decrease its residual capacity and increase the residual capacity of the reverse arc.

One can use vertex/arc pointers or array indices to access vertices or arcs. We’ll have to figure out what is more time-efficient. For space efficiently, indices are better because is graphs are not too large, one can use 32 bit integers, and except for very large graphs, vertex indices may be 32 bits (but arcs require 64 bits). I will use “pointer” below, but it may be implemented as an index.

## Efficient tree representation for network simplex

Each vertex has a parent, a and a pointer for a doubly linked list. Vertices in the tree are linked in the DFS traversal order. Each vertex also maintains the of the sub-tree rooted at it. During a pivot (described later), the tree changes and the information is updated in time proportional to the sum of the sub-tree size and the size of the sub-tree root to the tree root path.  
Both trees of the basis S and T are represented by this next/prev doubly linked list and parent pointers. There is also a field to know to which tree a vertex belongs.

To visit all vertices in a sub-tree rooted at v, we start at v and follow the next pointers for size(v) steps.

## Basic Network Simplex for Max-Flow

Network simplex is an augmenting path algorithm. Each step of the algorithm (pivot) may augment flow on a path and may rearrange the basis (defined below). A pivot will do one, the other, or both. Pivots that do not augment are called degenerate.

The algorithm maintains a flow (either explicitly or using residual capacities) and two trees, an out-tree S rooted at the source s and an in-tree T rooted at the sink t. Every vertex is either in S or in T. The basis B is the union of S and T. The version of the algorithm we consider maintains the invariant that every arc in S is residual (S arcs are directed away from s).

A pivot arc is a residual arc from S to T. If there is no pivot arc, (S,T) is a cut and the current flow is maximum. Given a pivot arc (v,w), a pivot on (v,w) works as follows. Let P be the path obtained by concatenating the s-v path in S, (v,w), and the w-t path in T. If P is a residual path, we augment flow, saturating at least one arc on P. Otherwise (degenerate pivot) P already contains a saturated arc. Let (x,y) be the first saturated arc on P.

The arc (v,w) is an entering arc and (x,y) is a leaving arc. We update B by replacing (x,y) by (v,w). Note that if (x,y) and (v,w) are the same, B does not change; in this case, (v,w) got saturated, so we did augment flow.

If (x,y) is in S, call Q the subtree of S rooted at y. Q will move from S to T during the pivot. To do that, we need to change its root to be v in the doubly linked list representation. Call R the tree that will represent Q rooted at v.   
We need to: transform Q into R, reverse parent pointers in the y to v path, and make w the parent of v. We will do this following the old parents from v to y.   
Before describing how to transform Q into R, we need to define two operations over the doubly linked list representation:  
To delete from a tree Q a subtree rooted at u: start at u, follow next pointers for the SIZE steps to find the last vertex, remove the sublist corresponding to the subtree (hence the need of a doubly linked list). Then go from u up to the root of Q updating subtree sizes.   
To add to a tree R a subtree as a child of u: insert the subtree list in the tree list immediately after u, go up updating subtree sizes.  
Now we can describe in more detail how to change the root for the pivot in the case (x,y) in S:

Delete from S the subtree rooted at y (Q)  
Call R the subtree of Q rooted at v and delete it from Q.  
Set z=v  
while z!=y   
set p=(old) parent of z  
Delete from Q the subtree rooted at p and add it to R as a child of z  
make z the parent of p  
set z=p and repeat  
Finally R has the same elements as Q but is rooted at v, we can connect R to T by adding R to T as a child of w. The case when (x,y) is in T is symmetric.

Note that augmentation can saturate several tree arcs, so we have a choice of leaving arcs. A choice rule we use (which is known to prevent cycling) is to choose the arc closest to s. Note that this maintains the invariant that S tree arcs directed away from s are residual (if this invariant holds for the initial tree).

One way to initialize the tree is to build a BFS tree S out of s in the residual graph with t deleted. We can delete vertices other than t which are not in S (they can be reached only via t) and make T = {t}.

We also need to maintain pivot arcs, the residual arcs from S to T. We compute the set P of pivot arcs during initialization and update it during a pivot. The update works as follows. Suppose a subtree X of S moves into T (the other case is symmetric). For every vertex x in X, we examine edges {x,y}. If (x,y) is residual and y is in T, we delete (x,y) from P. If (y,x) is residual and y is in S, we add (y,x) to P.

## Goldfarb-Hao Algorithm

One still needs to specify a rule for selecting an entering arc (v,w). GH prove that the following rule leads to a polynomial time algorithm. (This is not obvious. The proof is in the paper.)

**Pivot arc selection**: Among entering arcs, select the one closest to s in the residual graph.

GH finds the distances from s with respect to the following length function: l(v,w) = 1 if (v,w) or (w,v) is in S union T, and l(v,w) = 1 if (v,w) is a residual non-tree arc. Note that the tree arcs have length 1 even if they are non-residual; such arcs are called pseudo-residual. Other arcs have infinite length.

The intuition for pseudo-residual arcs is as follows. If the problem is non-degenerate (in LP sense), then all tree arcs are residual in both directions, so there are no pseudo-residual arcs. In the non-degenerate case also only one arc is saturated by an augmentation, by the way, so the leaving arc is unique.

GH show that with this pivot rule, distances are monotone.

GH have to be a little careful to have a better running time; they make sure that the total number of shortest path computations is O(n), so the corresponding cost is O(nm). The next algorithm achieves this in a simpler way.

## Goldberg-Grigoriadis-Tarjan Algorithm

GGH algorithm implements GH selection but uses relabel operation to maintain distances. Unlike the standard push-relabel algorithm that maintains estimates on distances to t, the network simplex algorithm maintains estimates on distances from s, so the relabel operation is modified accordingly.

Each vertex v maintains a *distance label* d(v) . We define d(s) = 0 and maintain the invariant that for every arc (v,w), d(w) <= d(v) + l(v,w) (*dual feasibility*). One can show that d(v) is a lower bound on the distance from s to v w.r.t. the length function l.

Every vertex also maintains a *current arc pointer* cur(v) to an arc (u,v). We say that a vertex v is *current* if its current arc (u,v) if d(u) + l(u,v) = d(v). One can show that if all vertices except s are current, then d gives correct distances.

GGH maintains the distances incrementally. Initially S contains all vertices except t, so d(v) = 0 for all v except d(t) = 1. (Remember that we deleted unreachable vertices.) In GGT, distance labels of all vertices reachable from s are exact (equal to the true distances) before a pivot, and all these vertices are current.

A *relabel* operation finds the minimum of d(u)+l(u,v) for all arcs into v, sets d(v) to the minimum, and sets cur(v) to the first arc on the arc list for which d(v) = d(u) + l(u,v). The *make\_cur(v)* operation advances cur(v) until d(v) = d(u) + l(u,v) or until the end of the list is reached. In the latter case, the operation fails and we put v on the to\_relabel list. Relabel extracts vertices from this list and processes them.

After an augmentation, we first apply make\_cur to all vertices make non-current by the augmentation. Then we process the to\_relabel list. During this processing, if relabel(v) makes u non-current, we apply make\_cur(u) and if it fails, we add u to the relabel list.

Since d is monotone, each call to relabel(v) increases d(v). The work of examining the arc list of v by make\_cur and relabel can be charged to the increase of d(v) and costs degree(v) per an increase. If d(v) increases to n or above, we know that there is no finite length path from s to v, so we can delete v (it is on t side of a minimum cut).

We also maintain a set of non-current vertices. While the list is non-empty, we take a vertex v from the list and call make\_cur(v). This either makes v current or deletes v. In addition, we modify the relabel operation to make sure that vertices made non-current by the increase of d(v) are added to the non-current list. After relabel(v) increases d(v), we check all arcs (v,u). If u was current before the relabeling and cur(u) = (v,u), we make u non-current.

All vertices are current before a pivot. If the pivot augmentation saturates an arc (x,y), y may become non-current (if cur(y) = (x,y)). In this case we add y to the non-current list. After all saturated arcs are examined in this fashion, all non-current vertices are on the list. We repeatedly remove vertices from the list and apply make\_cur operation to them. This may add neighbor vertices to the list, but eventually the list becomes empty. At this point we make a new pivot.

Because of the monotonicity of d, the total time for make\_cur and relabel is O(nm).

In this algorithm, we maintain a set of pivot arcs as a priority queue, with the distance of the tail as the key. We choose the minimum arc as a pivot arc at each iteration.

## New Lazy Simplex Algorithm

The new algorithm is based the following fact: if all vertices with distance label <= k are current, then the distance labels of these vertices are equal to the correct distance. We make two modification to the GGT algorithm. First, we maintain to\_relabel list as a priority queue with d as the key. Second, we define M to be the minimum distance label of an arc on the pivot arc queue. We stop relabeling when either to\_relabel queue is empty (as in GGT) or when the minimum key of a vertex on the queue is greater than M.

A possible heuristic to borrow from the push-relabel framework is global update: once we did enough work to justify a breadth-first search, we do the search and update distance labels and cur arcs.

## Corrections and Additions

Instead of pivot arc set, it is better to maintain a set of vertices v such that there is at least one pivot arc (v,w). Note that v must me in S. If we select a vertex v for pivot, we scan the arc list to find an outgoing pivot arc. This way, if we increase d(v), we just need to increase key of v when we maintain the set as a priority queue. Otherwise we need to find all arcs (v,w) on the queue.

Related change: if a vertex v moves from S to T and v on the pivot list, we delete v from the pivot list. If v moves from T to S, we scan v to see if it needs to be added to the pivot list.

We apply make\_cur as soon as we discover that a vertex became non-current. If make\_cur fails, we add the vertex to to\_relabel. This way we do not need to maintain an extra set of vertices to apply make\_cur to.

The priority queues should be maintained as an array [0,…,n] of buckets.