Network Optimisation Problems

Definitions & terminology

A network or graph $G = (V, E)$ consists of a set V of vertices or nodes, and a set $E \subseteq V \times V$ of ordered pairs of nodes, called arcs or edges

If the order doesn't matter, i.e., if (u, v) & (v, u) refer to the same edge, then the edge is called undirected

Otherwise, the edge (u, v) is said to be directed or oriented from u to v; u is called the tail vertex & v the head.

Sometimes, the term arc is used only for directed edges

Some more terminology

A path between u and v is a subgraph of G with nodes $\omega_1, \omega_2, \dots, \omega_k$ and arcs $e_1, e_2, \dots, e_{k-1}$ such that $u = \omega_1$, $v = \omega_k$, and the endpoints of arc e_i are ω_i and ω_{i+1}.

Remarks : 1) A subgraph of G is a graph $G' = (V', E')$ such that $V' \subseteq V$, $E' \subseteq E$.

2) Note that e_i could be (ω_i, ω_{i+1}) or (ω_{i+1}, ω_i). The directions of edges along a path need not line up.

3) The nodes $\omega_1, \dots, \omega_k$ need not be distinct.

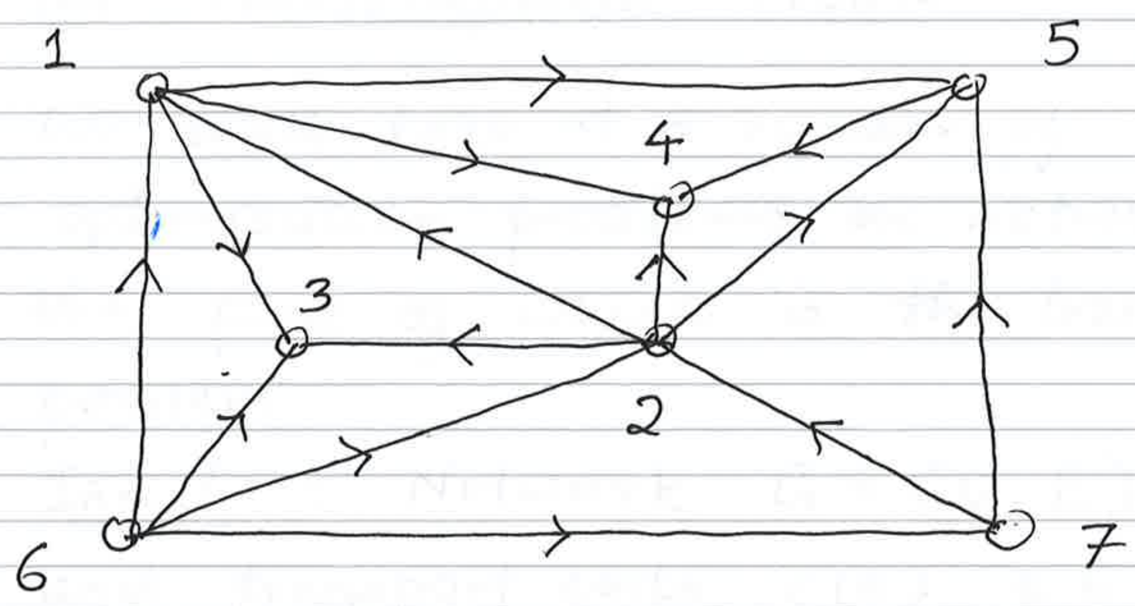
A cycle is a path $\omega_1, \dots, \omega_k$ such that $\omega_1 = \omega_k$, and containing at least 2 distinct arcs.

A network is called connected if there is a path between every pair of vertices u, v. It is called acyclic if it contains no cycles.

A connected, acyclic network is called a tree.

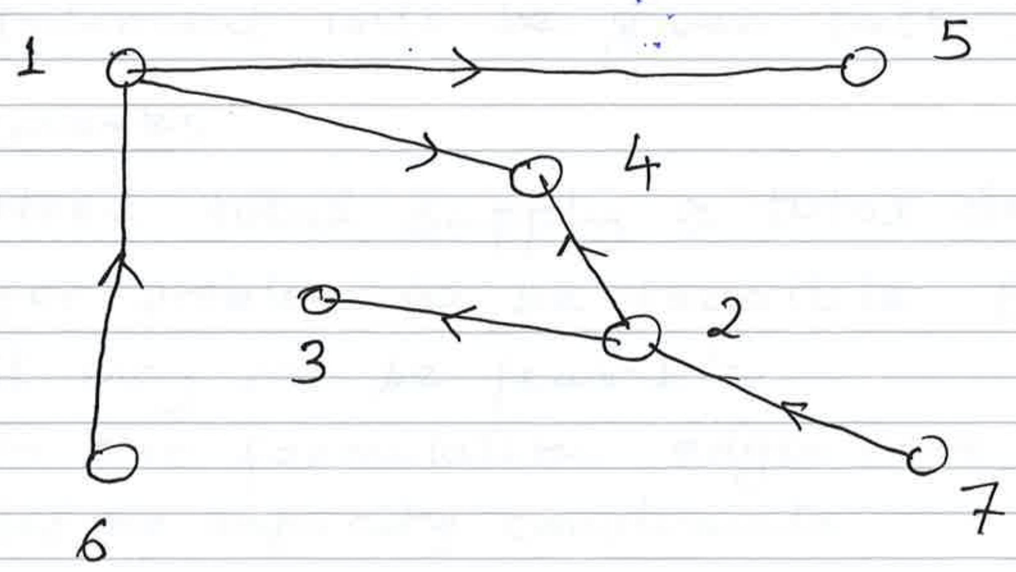
If $T \subseteq G$ is a tree, $T = (V_T, E_T)$ and $V_T = V$, then T is called a spanning tree of G.

Example (from Chvatal, "Linear Programming")



The vertices {1, 4, 5, 2, 3, 1, 4} together with the edges {(1, 4), (5, 4), (2, 5), (2, 3), (1, 3), (1, 4)} form a path. Excluding the last vertex & edge, they form a cycle.

The following is a spanning tree:



The Transshipment Problem

We shall look at a number of optimisation problems on networks, the first of which is the transshipment problem.

Input : Network $G = (V, E)$, and transport costs $c(e)$, $e \in E$, along each edge, as well as supplies & demands $b(v)$, $v \in V$, at the vertices

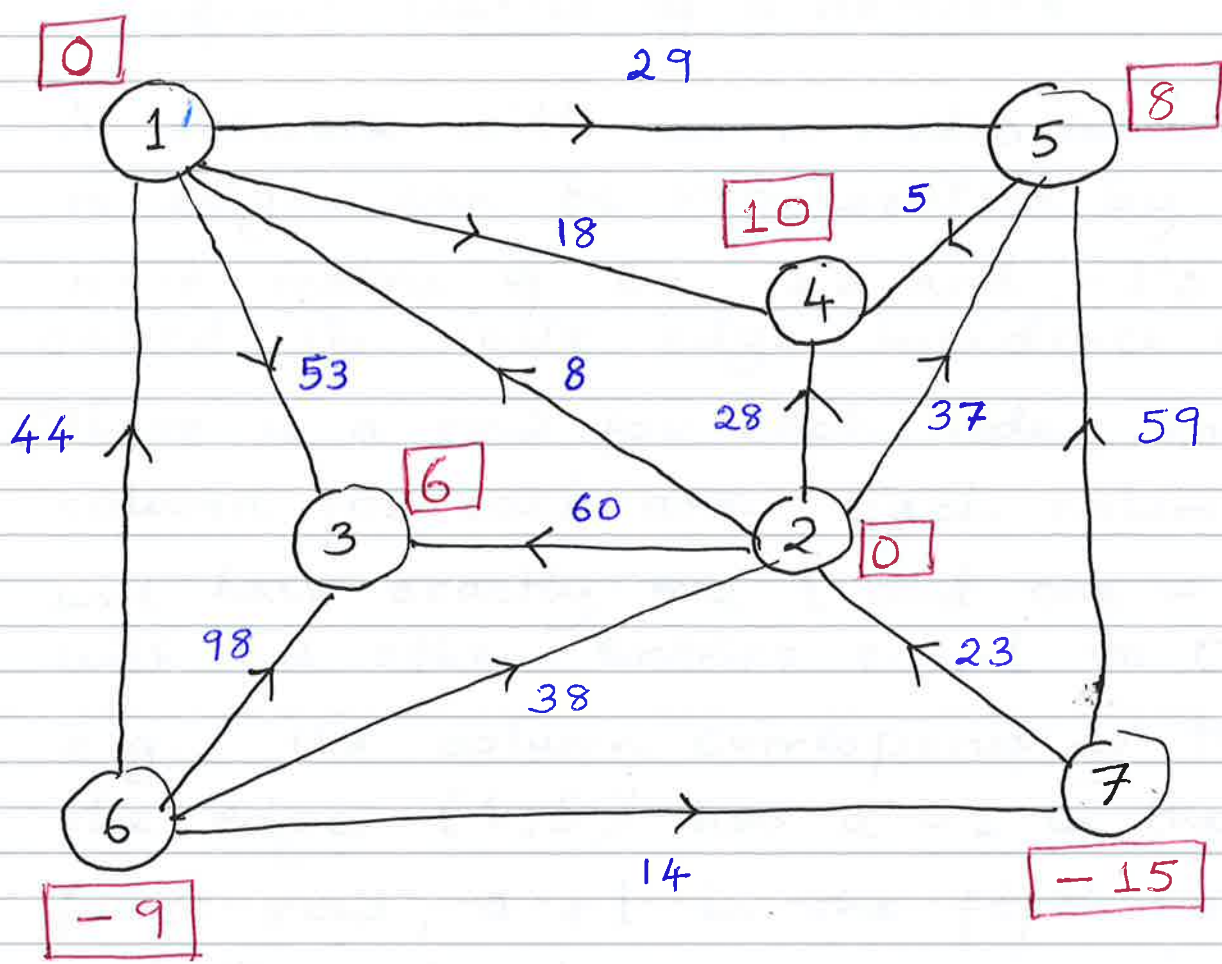
Problem : Meet the vertex demands by shipping from the vertices with supply, at minimum total cost.

A more formal & precise problem statement will be given later.

Remarks

1. Need total supply $\geq$ total demand, for problem to be feasible. Even then, it may not be feasible.
2. In our formulation, edges have costs but no capacity constraints.
3. There is a single, undifferentiated commodity to be transported.

Example (from Chvatal, "Linear Programming", W. H. Freeman & Co)



Vertex labels are inside the circles denoting vertices.

Demands are shown in red boxes next to the vertex; a negative demand denotes a supply.

Costs are shown in blue next to edges, and are unit costs for shipping 1 unit of the commodity along the edge.

We will assume the total demand is zero.

Representation as LP

Incidence matrix of a network

A network with n nodes and m edges can be represented by an $n \times m$ matrix of 0's, 1's and −1's, called its node-edge incidence matrix. There is a row for each node, and a column for each arc. Each column will have exactly one 1 and one −1, with all other entries equal to 0.

e.g., the column corresponding to the edge (1, 5) has a −1 in the first row, a +1 in the fifth row, and 0's elsewhere.

A schedule of shipments $\underline{x}$ is a vector indexed by the edges $e \in E$ such that x_e denotes the amount shipped along edge e, and these satisfy flow conservation: the total amount (net) of shipment entering a node is equal to the demand at that node.

LP formulation

Notation :

Incidence matrix $A \in \mathbb{R}^{n \times m}$

Demand vector $\underline{b} \in$ ~~$\mathbb{R}^{m \times 1}$~~ $\mathbb{R}^{n \times 1}$

Cost vector $\underline{c} \in \mathbb{R}^{m \times 1}$

(vectors are column vectors unless otherwise specified)

Assumption : $\underline{b}^T \mathbb{1} = \sum_{i=1}^{n} b_i = 0$

$\mathbb{1}$ is the all-1s column vector.

Transshipment problem :

$$\min_{\underline{x} \in \mathbb{R}^m} \underline{c}^T \underline{x}$$

subject to $A \underline{x} = \underline{b}$, $\underline{x} \geq \underline{0}$

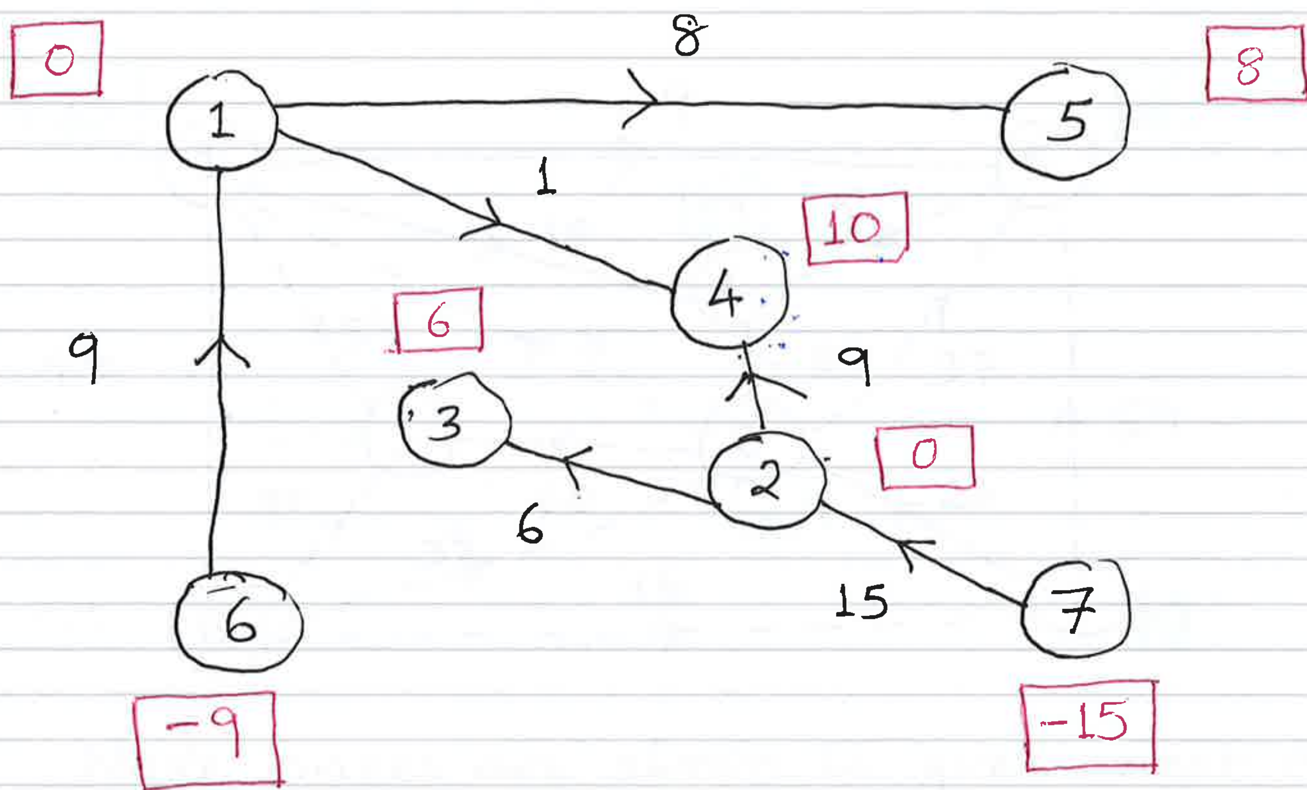
We will now see how to solve this LP using the simplex method, which takes a particularly simple form on this problem

The Network Simplex Method

A <u>feasible</u> <u>tree</u> <u>solution</u> for the transshipment problem is a spanning tree T of the network $G = (V, E)$, and a schedule of shipments $\underline{x}$ satisfying the demands, such that $x_e = 0$ if $e \notin T$

It is the analogue of basic feasible solutions in the simplex method.

A feasible tree solution for our transshipment problem is shown below:

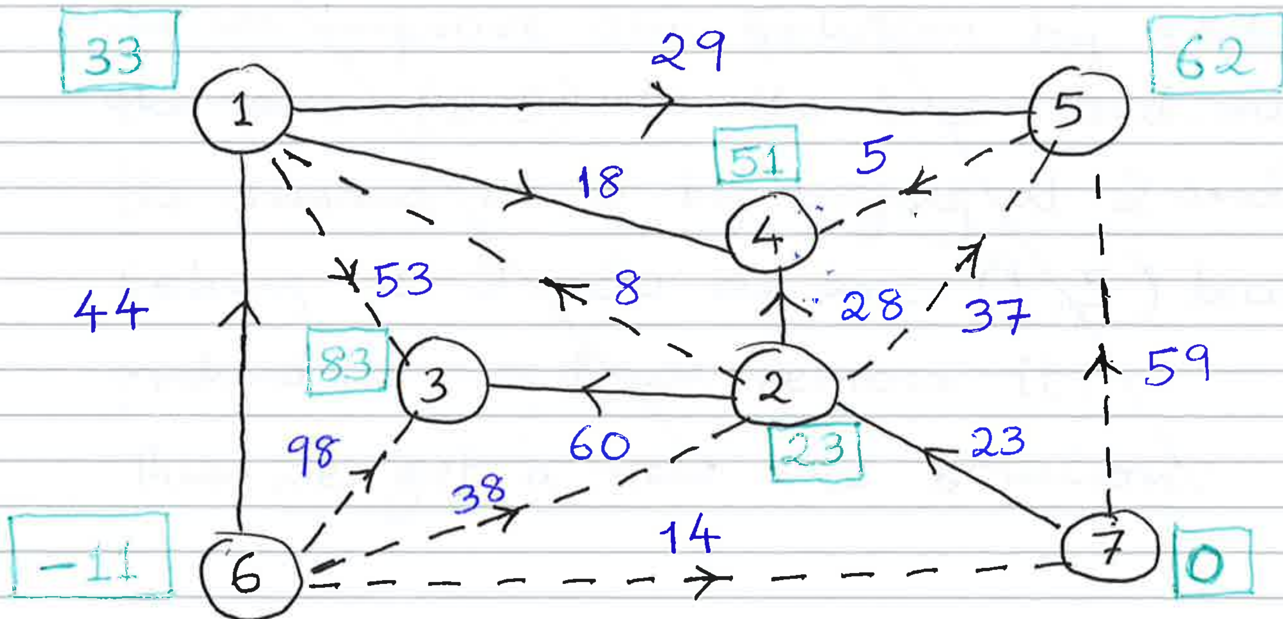


Flows are shown next to edges; edge costs are not shown.

Network Simplex (contd.)

The network simplex method starts from a feasible tree solution (we will see how to initialise it later) and seeks to improve it, based upon an intuitive economic interpretation.

Given a feasible tree solution, we can assign unit prices to the commodity at different nodes, relative to a reference price at some specific node. This is shown for our tree solution above:



Node prices are shown in green next to the nodes; demands are not shown.
Edge costs are shown next to edges; edges not in the tree are shown dashed.

Network simplex (contd.)

Given a feasible tree solution, it implies a set of 'fair prices' for the commodity at different nodes, if that shipping schedule was used.

Do these prices create a profit opportunity for a competitor?

Denote the price vector $y \in \mathbb{R}^n$.

Are there nodes i and j such that

$$y_i + c_{ij} < y_j \ ?$$

Network simplex (contd.)

In our example, $y_7 = 0$, $c_{75} = 59$, $y_5 = 62$, so a competitor could make a profit of £3/unit by buying at 7 & selling at 5. Alternatively, we can improve the solution, by adding arc (7,5) to the tree. This creates the cycle $\{7, 5, 1, 4, 2, 7\}$, so it is no longer a tree. To get a tree, we need to remove one of these arcs. But which one?

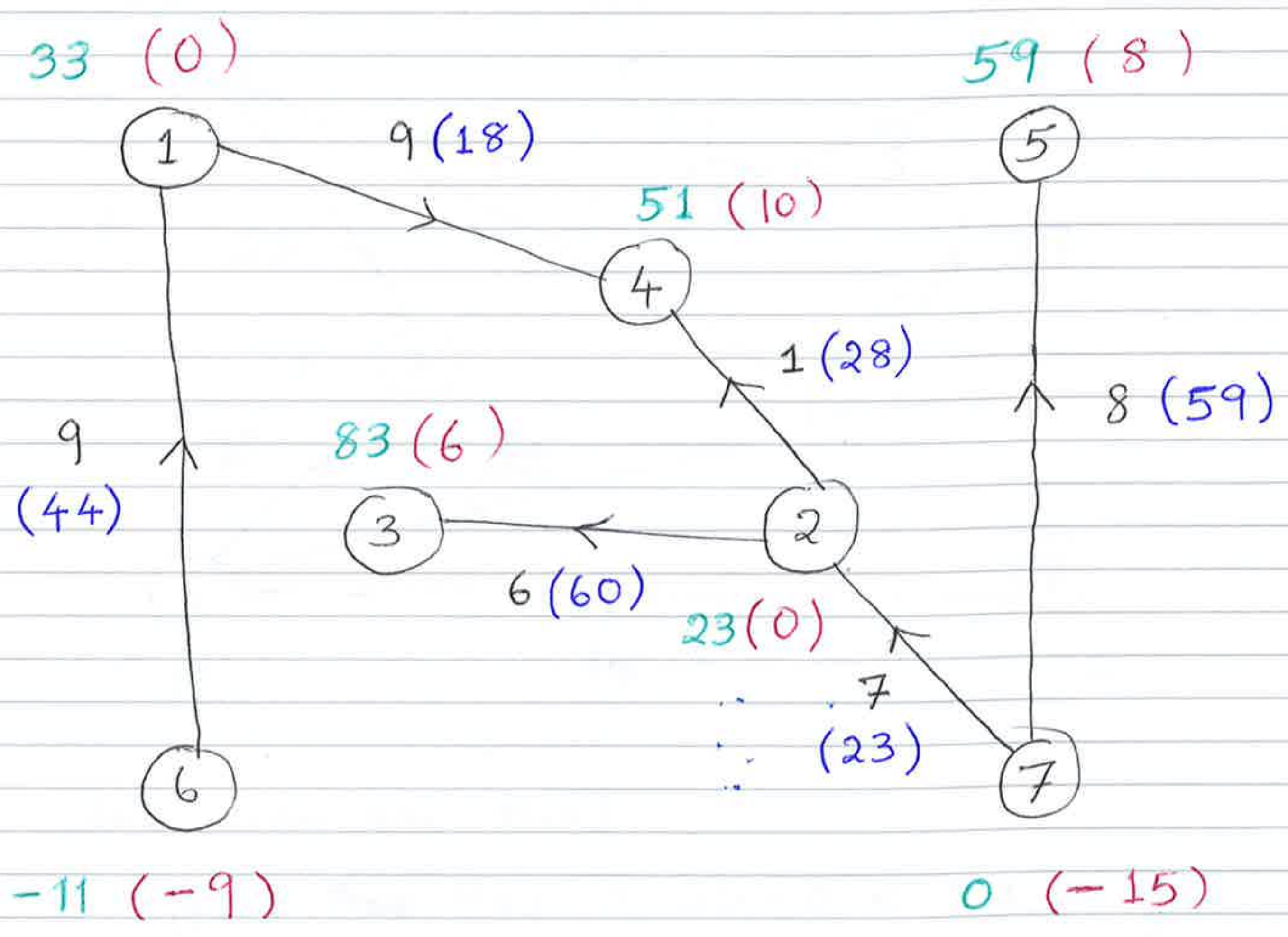
Consider pushing t units of flow around this cycle. This changes the flows in our solution to

$x_{75} = t$, $x_{15} = 8-t$, $x_{14} = 1+t$,
$x_{24} = 9-t$, $x_{72} = 15-t$.

The largest value of t that would yield a feasible flow is $t = 8$. So we set $t = 8$. The flow on arc (1,5) is now zero, so this arc drops out of the new tree solution

Network simplex (contd.)

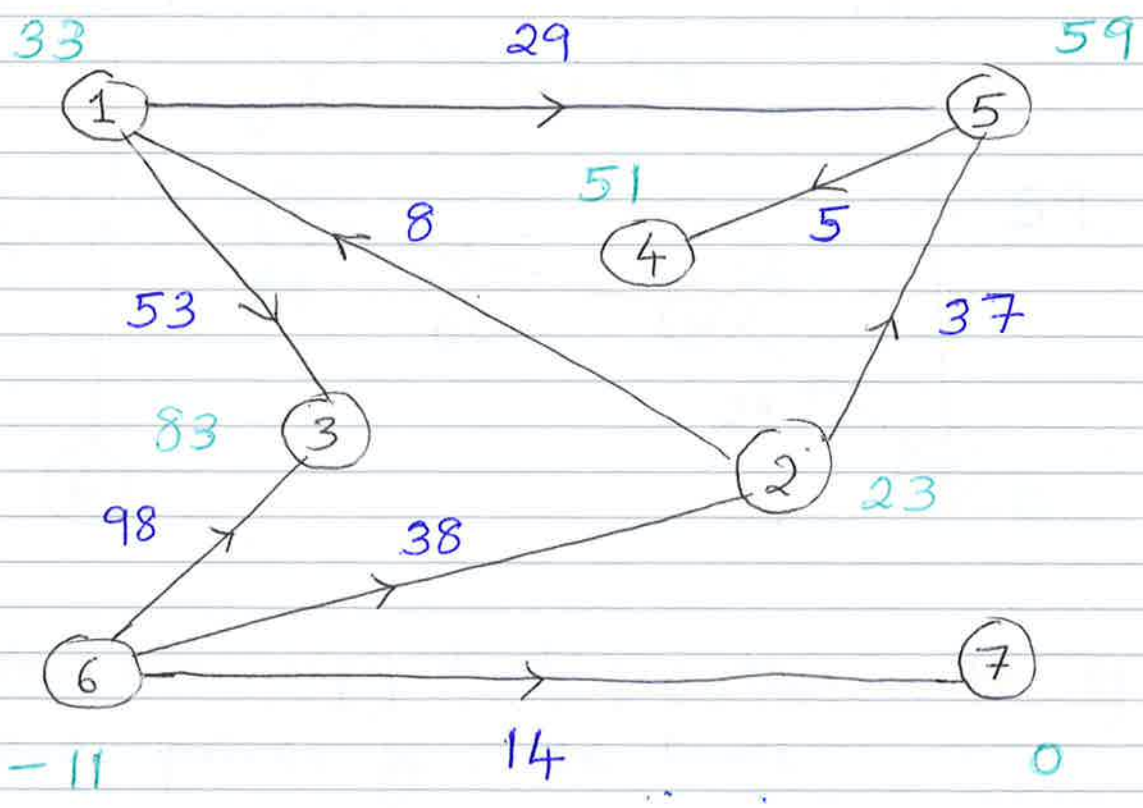
The new tree solution along with arc flows (and costs in brackets), and revised node prices is shown below (node demands in brackets).



We are now ready to repeat the process of trying to find an improvement.

Network simplex (contd.)

We now show the node prices, and the residual arcs (arcs not in the current tree) with their costs.



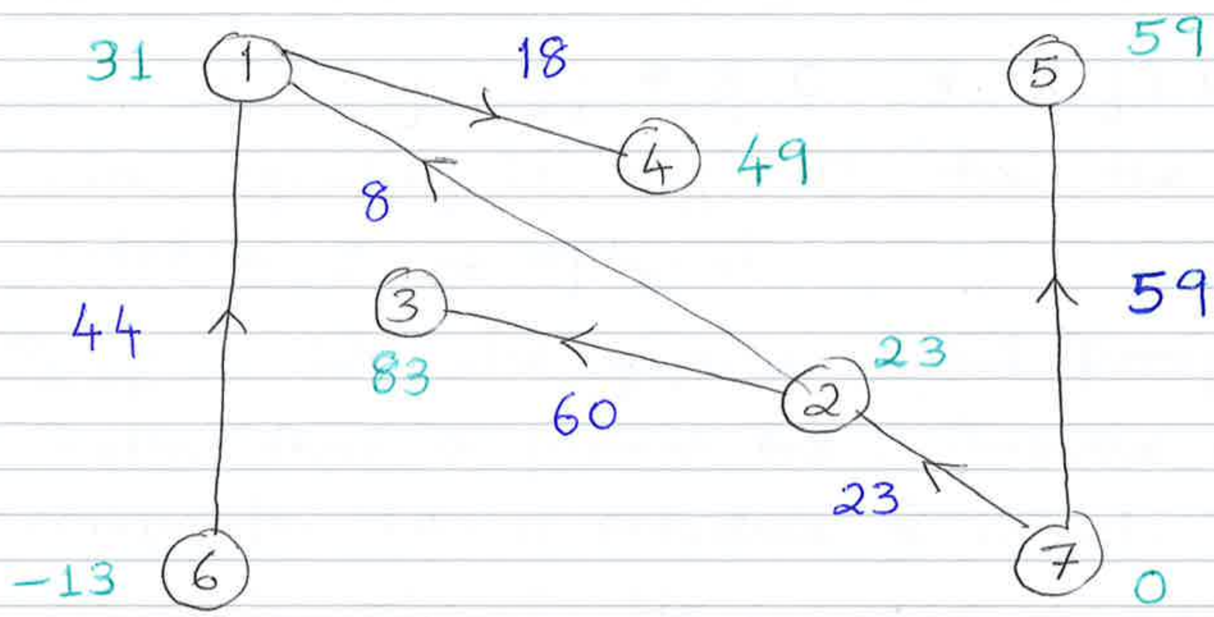
We can see that

$$y_2 + c_{21} = 23 + 8 < 33 = y_1$$

So we bring arc (2,1) into the previous tree solution, creating the cycle $\{2, 1, 4, 2\}$. Pushing 1 unit of flow along this cycle makes the flow on arc (2,4) zero.

Network simplex (contd.)

The new tree, with edge costs & node prices, are shown below:



You can verify that for this vector of node prices y, we have

$$y_i + c_{ij} \geq y_j \quad \forall (i,j) \in E$$

Claim : The resulting flow x is the optimal solution to the transshipment problem

Remark : In this example, the solution is unique, and uniquely determined by the corresponding tree. Even if the solution isn't unique, the total cost of any solution should be the same.

Proof of optimality :

We claimed above that the flow obtained by our algorithm is optimal. Why ?

We will give a verbal justification rather than a formal proof. Suppose a competitor has a shipping schedule $\tilde{\underline{x}}$, which he claims is optimal. He incurs a cost $\underline{c}^T \tilde{\underline{x}}$ to meet the demands using this schedule.

We offer to trade with him at the prices $\underline{y}$ calculated above. We claim that, if he accepts our offer, then he can meet the demands at the same or lower total cost.

To see this, observe that

$$c_{ij} \geq y_j - y_i \quad \forall (i,j) \in E$$

$$\therefore \quad \underline{c}^T \tilde{\underline{x}} = \sum_{(i,j)\in E} c_{ij} \tilde{x}_{ij}$$

$$\geq \sum_{(i,j)\in E} (y_j - y_i) \tilde{x}_{ij}$$

Thus, he is no worse off trading with us. But we are not trading at a loss, we break even on our shipping costs.

This verbal argument is in fact the Duality Theorem.

Initialisation

Starting from a feasible tree solution, we have seen how to iteratively improve it to solve the transhipment problem. But how do we find a feasible tree solution in the first place?

We do so by solving an auxiliary transshipment problem!

Introduce an artificial extra node w, artificial extra arcs from every source node u (a node with negative demand) to w, and artificial arcs from w to every sink node v (node with positive demand). Assign cost 1 to each artificial arc & cost 0 to every original arc.

The auxiliary problem has an obvious initial feasible tree solution. Can you see it? If an optimal solution of the auxiliary problem has zero cost, then it uses no artificial arcs & yields a feasible tree solution of the original problem. If not, then the original problem is infeasible.

Integrality Theorem

In our example, we used integer values for the costs & demands in order to make calculations simpler. And we ended up with a solution where the flows on all edges were integer-valued. Was this just a coincidence?

Theorem: Consider a transshipment problem

$$\min \ \underline{c}^T \underline{x} \quad \text{subj. to } A\underline{x} = \underline{b}, \ \underline{x} \geq 0$$

such that all components of the vector $\underline{b}$ are integers. If the problem has at least one feasible solution $\underline{x}$, then it has an integer-valued feasible solution. If it has an optimal solution $\underline{x}^*$, then it has an integer-valued optimal solution.

Remark: Note that only $\underline{b}$ has to be integer-valued; the costs $\underline{c}$ can be arbitrary.

The Shortest Path Problem

Consider a graph $G = (V, E)$ as before. Also as before, we will assume that the edges are directed. This involves no loss of generality, as each undirected edge can be represented by a pair of oppositely directed edges, (i,j) and (j,i). There is a length associated with each edge, which we will assume to be non-negative.

A directed path between u and v is a subgraph of G with nodes

$$u = w_1, w_2, \ldots, w_k = v$$

and edges (w_i, w_{i+1}), $i = 1, \ldots, k-1$. Note the difference with the definition of a path.

We let d_{ij} denote the length of edge (i,j). The total length of a directed path $\{w_1, w_2, \ldots, w_k\}$ (the edges are implied in the notation) is defined as

$$d_{w_1, w_2} + d_{w_2, w_3} + \ldots + d_{w_{k-1}, w_k}.$$

Shortest Path Problem (contd.)

The distance from u to v is defined as the minimum length of any path connecting u to v. A path achieving the minimum is called a shortest path from u to v. Denote the distance D_{uv}.

Remarks: On directed graphs, D_{uv} need not be equal to D_{vu} (distances are not symmetric). However, they do satisfy the triangle inequality:

$$D_{uv} \leq D_{uw} + D_{wv} \quad \forall \ w \in V.$$

On undirected graphs, i.e., if $d_{ij} = d_{ji}$ $\forall \ i,j \in V$, then $D_{uv} = D_{vu}$.

There are 3 versions of the shortest path problem.

1. Given $s, t \in V$, compute D_{st} and find a shortest s-t path
2. Given $s \in V$, compute D_{st} $\forall \ t \in V$
3. Compute D_{st} $\forall \ s,t \in V$.

We will study algorithms for (2).

Dijkstra's algorithm

The algorithm starts with a set consisting only of $\{s\}$, to which it knows a shortest path from s. It iteratively grows this set until it contains all nodes. The pseudocode is below:

Step 0: $P = \{s\}$, $D_s = 0$
$D_v = d_{sv} \quad \forall \; v \neq s$. $\left(d_{sv} := \infty \text{ if } (s,v) \notin E \right)$

Step 1: Find $u \notin P$ such that
$$D_u = \min_{v \notin P} D_v$$
Set $P = P \cup \{u\}$. If $P = V$, stop.

Step 2: Set $D_v = \min \{ D_v, D_u + d_{uv} \}$
Return to Step 1.

Q: 1) Does this algorithm work?
2) How many computations does it involve?

Correctness of Dijkstra's algorithm

Claim : At the beginning of each iteration of Step 1, D_v is the minimum distance from s to v along paths using only vertices in P (except for the final vertex P).

Proof : This is clearly true after Step 0. So we only need to check that it remains true at each iteration of Step 2. Suppose it was true before Step 1 was executed. At Step 1, another node u was added to P, whose distance from s using only nodes in P was D_u.

Now, the shortest path from s to v using only nodes in P either remains unchanged, or changes to include u. In the latter case, u must be the last node within P of a shortest s-v path, otherwise it would have entered P earlier. Thus, the minimisation in Step 2 exactly accounts for these two possibilities.

Computational Complexity

We denote $|V|$ by n and $|E|$ by m.
Note that Dijkstra's algorithm starts with $P = \{s\}$ and ends with $P = V$.
Thus, there are n (or $n-1$) iterations.
Each iteration involves finding the minimum in a set of size at most n (which involves n comparisons), and updating distances for at most n nodes (involving n additions & n minima).
So there are at most $3n$ computations per iteration.
So, the computational complexity is at most $3n^2 = O(n^2)$.

A more careful analysis shows that when node v enters P, we need only update distances for the neighbours of v, and not all nodes.
Using clever data structures for maintaining the distance estimates D_v, the best known implementation of Dijkstra's algorithm has a complexity of
$O(m + n \log n)$.

Bellman-Ford algorithm

Dijkstra's algorithm obtains, on its kth iteration, the set of k nodes closest to $\{s\}$. In successive, it grows to include all nodes. Exactly one node is added in each ~~step~~ iteration, so it always needs $n-1$ iterations.

The Bellman-Ford algorithm computes, at its k-th iteration, the length of all shortest paths from s of at most k hops. The pseudocode is below:

Initialisation : $D_s^0 = 0$, $D_v^0 = \infty \quad \forall v \neq s$

Iteration : $D_v^{k+1} = \min_u \{ D_u^k + d_{uv} \}$

Termination : Stop if $D^{k+1} = D^k$.

It is guaranteed that $D^n = D^{n-1}$ as any path without repeated vertices has at most $n-1$ edges (hops). But the algorithm could terminate sooner.

Dijkstra's algorithm requires all edge lengths to be non-negative, but Bellman-Ford doesn't. If $D^n \neq D^{n-1}$, then the graph contains a negative cost cycle, and the shortest path problem is unbounded.

Correctness of Bellman-Ford algorithm

To check correctness, we just need to prove the claim that D^k is the vector of lengths of shortest paths with at most k edges from s. We do this by induction on k.

- The base case $k=0$ is obvious
- Suppose the result is true for k. The shortest path from s to v with at most $k+1$ hops either has k or fewer hops, or has k hops up to u, and an extra hop from u to v.

Complexity

At each iteration, we need to update a vector of length $n = |V|$.
The update at a single vertex involves an addition & minimisation over all its neighbours. So the update of the vector needs $O(m)$ computations.
There are at most n iterations, so the complexity is $O(mn)$

Remark: In practice, it could be a lot less.