Algorithm Tutorials

Minimum Cost Flow, Part 1: Key Concepts

By Zealint
TopCoder Member

This article covers the so-called "min-cost flow" problem, which has many applications for both TopCoder competitors and professional programmers. The article is targeted to readers who are not familiar with the subject, with a focus more on providing a general understanding of the ideas involved rather than heavy theory or technical details; for a more in-depth look at this topic, check out the references at the end of this article, in particular [1]. In addition, readers of this article should be familiar with the basic concepts of graph theory -- including shortest paths [4], paths with negative cost arcs, negative cycles [1] -- and maximum flow theory's basic algorithms [3].

The article is divided into three parts. In Part 1, we'll look at the problem itself. The next part will describe three basic algorithms, and Part 3 some applications to the problem will be covered in Part 3.

Statement of the Problem

What is the minimum cost flow problem? Let's begin with some important terminology.

Let $G = (V, E)$ be a directed network defined by a set V of vertexes (nodes) and set E of edges (arcs). For each edge $(i, j) \in E$ we associate a **capacity** u_{ij} that denotes the maximum amount that can flow on the edge. Each edge $(i, j) \in E$ also has an associated **cost** c_{ij} that denotes the cost per unit flow on that edge.

We associate with each vertex $i \in V$ a number b_i. This value represents supply/demand of the vertex. If $b_i > 0$, node i is a **supply node**; if $b_i < 0$, node i is a **demand node** (its demand is equal to $-b_i$). We call vertex i a **transshipment** if b_i is zero.

For simplification, let's call G a **transportation network** and write $G = (V, E, u, c, b)$ in case we want to show all the network parameters explicitly.

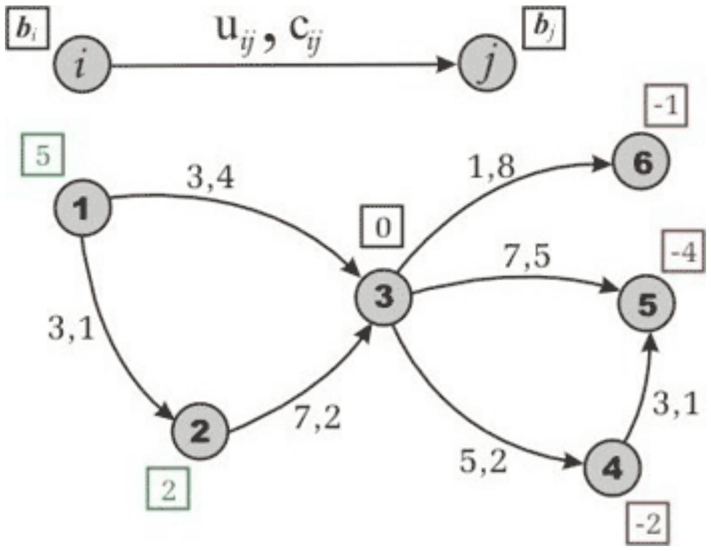

Figure 1. An example of the transportation network. In this we have 2 supply vertexes (with supply values 5 and 2), 3 demand vertexes (with demand values 1, 4 and 2), and 1 transshipment node. Each edge has two numbers, capacity and cost, divided by comma.

Representing the flow on arc $(i,j) \in E$ by x_{ij}, we can obtain the optimization model for the minimum cost flow problem:

$$\text{Minimize } z(x) = \sum_{(i,j) \in E} c_{ij} x_{ij}$$

subject to

$$\sum_{\{j:(i,j) \in E\}} x_{ij} - \sum_{\{j:(j,i) \in E\}} x_{ji} = b_i \qquad \text{for all } i \in V,$$

$$0 \le x_{ij} \le u_{ij} \qquad \text{for all } (i,j) \in E.$$

The first constraint states that the total outflow of a node minus the total inflow of the node must be equal to mass balance (supply/demand value) of this node. This is known as the **mass balance constraints**. Next, the **flow bound constraints** model physical capacities or restrictions imposed on the flow's range. As you can see, this optimization model describes a typical relationship between warehouses and shops, for example, in a case where we have only one kind of product. We need to satisfy the demand of each shop by transferring goods from the subset of warehouses, while minimizing the expenses on transportation.

This problem could be solved using simplex-method, but in this article we concentrate on some other ideas related to network flow theory. Before we move on to the three basic algorithms used to solve the minimum cost flow problem, let's review the necessary theoretical base.

Finding a solution

When does the minimum cost flow problem have a feasible (though not necessarily optimal) solution? How do we determine whether it is possible to translate the goods or not?

If $\delta \overset{def}{=} \sum_{i \in V} b_i \ne 0$ then the problem has no solution, because either the supply or the demand dominates in the network and the mass balance constraints come into play.

We can easily avoid this situation, however, if we add a special node r with the supply/demand value $b_r = -\delta$. Now we have two options: If $\delta > 0$ (supply dominates) then for each node $i \in V$ with $b_i > 0$ we add an arc (i,r) with infinite capacity and zero cost; otherwise (demand dominates), for each node $i \in V$ with $b_i < 0$, we add an arc (r,i) with the same properties. Now we have a new network with $\sum_{i \in V \cup \{r\}} b_i = 0$ -- and it is easy to prove that this network has the same optimal value as the objective function.

Consider the vertex r as a rubbish or scrap dump. If the shops demand is less than what the warehouse supplies, then we have to eject the useless goods as rubbish. Otherwise, we take the missing goods from the dump. This would be considered shady in real life, of course, but for our purposes it is very convenient. Keep in mind that, in this case, we cannot say what exactly the "solution" of the corrected (with scrap) problem is. And it is up to the reader how to classify the emergency uses of the "dump." For example, we can suggest that goods remain in the warehouses or some of the shop's demands remain unsatisfied.

Even if we have $\delta = 0$ we are not sure that the edge's capacities allow us to transfer enough flow from supply vertexes to demand ones. To determine if the network has a feasible flow, we want to find any transfer way what will satisfy all the problem's constraints. Of course, this feasible solution is not necessarily optimal, but if it is absent we cannot solve the problem.

Let us introduce a source node s and a sink node t. For each node $i \in V$ with $b_i > 0$, we add a source arc (s,i) to G with capacity b_i and cost 0. For each node $i \in V$ with $b_i < 0$, we add a sink arc (i,t) to G with capacity $-b_i$ and cost 0.

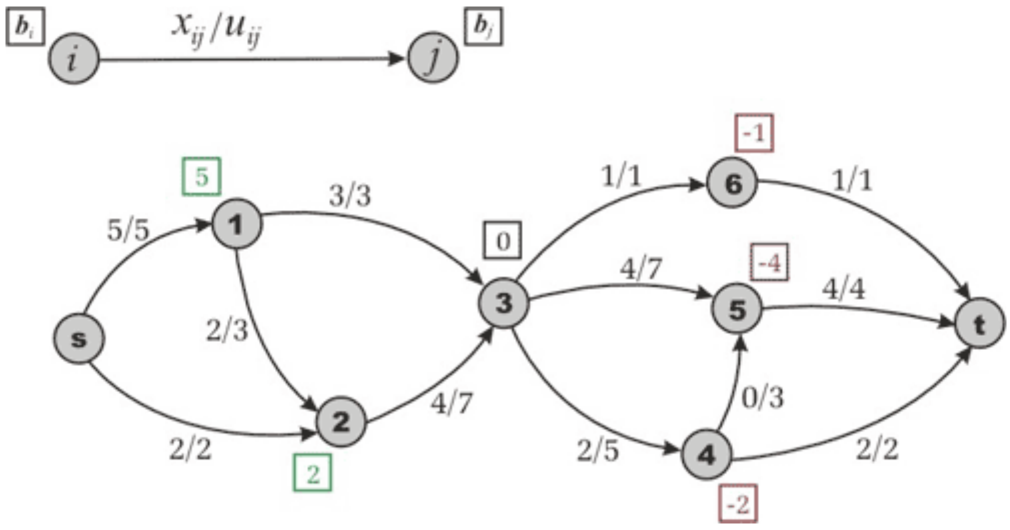

Figure 2. Maximum flow in the transformed network. For simplicity we are ignoring the costs.

The new network is called a **transformed network**. Next, we solve a maximum flow problem from s to t (ignoring costs, see fig.2). If the maximum flow saturates all the source and sink arcs, then the problem has a feasible solution; otherwise, it is infeasible. As for why this approach works, we'll leave its proof to the reader.

Having found a maximum flow, we can now remove source, sink, and all adjacent arcs and obtain a feasible flow in G. How do we detect whether the flow is optimal or not? Does this flow minimize costs of the objective function z? We usually verify "optimality conditions" for the answer to these questions, but let us put them on hold for a moment and discuss some assumptions.

Now, suppose that we have a network that has a feasible solution. Does it have an optimal solution? If our network contains the negative cost cycle of infinite capacity then the objective function will be unbounded. However, in some tasks, we are able to assign finite capacity to each uncapacitated edge escaping such a situation.

So, from the theoretical point of view, for any minimum cost flow problem we have to check some conditions: The supply/demand balance, the existence of a feasible solution, and the last situation with uncapacitated negative cycles. These are necessary conditions for resolving the problem. But from the practical point of view, we can check the conditions while the solution is being found.

Assumptions

In understanding the basics of network flow theory it helps to make some assumptions, although sometimes they can lead to a loss of generality. Of course, we could solve the problems without these assumptions, but the solutions would rapidly become too complex. Fortunately, these assumptions are not as restrictive as they might seem.

Assumption 1. *All data (u_{ij}, c_{ij}, b_i) are integral.*

As we have to deal with a computer, which works with rational numbers, this assumption is not restrictive in practice. We can convert rational numbers to integers by multiplying by a suitable large number.

Assumption 2. *The network is directed.*

If the network were undirected we would transform it into a directed one. Unfortunately, this transformation requires the edge's cost to be nonnegative. Let's validate this assumption.

To transform an undirected case to a directed one, we replace each undirected edge connecting vertexes i and j by two directed edges (i,j) and (j,i), both with the capacity and cost of the replaced arc. To establish the correctness of this transformation, first we note that for undirected arc $(i,j)\in E$ we have constraint $x_{ij}+x_{ji}\le u_{ij}$ and the term $c_{ij}x_{ij}+c_{ij}x_{ji}$ in the objective function. Given that $c_{ij}\ge 0$ we see that in some optimal solution either x_{ij} or x_{ji} will be zero. We call such a solution non-overlapping. Now it is easy to make sure (and we leave it to the reader) that every non-overlapping flow in the original network has an associated flow in the transformed network with the same cost, and vise versa.

Assumption 3. *All costs associated with edges are nonnegative.*
This assumption imposes a loss of generality. We will show below that if a network with negative costs had no negative cycle it would be possible to transform it into one with nonnegative costs. However, one of the algorithms (namely cycle-canceling algorithm) which we are going to discuss is able to work without this assumption.

For each vertex $i\in V$ let's denote by π_i a number associated with the vertex and call it the **potential** of node i. Next define the **reduced cost** c_{ij}^{π} of an edge $(i,j)\in E$ as

$$c_{ij}^{\pi}=c_{ij}+\pi_i-\pi_j$$

How does our objective value change? Let's denote reduced value by $z(x,\pi)$. Evidently, if $\pi=0$, then

$$z(x,0)=\sum_{(i,j)\in E}c_{ij}x_{ij}=z(x)$$

For other values of π we obtain following result:

$$z(x,\pi)=\sum_{(i,j)\in E}c_{ij}^{\pi}x_{ij}=\sum_{(i,j)\in E}(c_{ij}+\pi_i-\pi_j)x_{ij}=$$
$$z(x)+\sum_{(i,j)\in E}\pi_i x_{ij}-\sum_{(i,j)\in E}\pi_j x_{ij}=$$
$$=z(x)+\sum_{i\in V}\pi_i\sum_{\{j:(i,j)\in E\}}x_{ij}-\sum_{j\in V}\pi_j\sum_{\{i:(i,j)\in E\}}x_{ij}=$$
$$=z(x)+\sum_{i\in V}\pi_i\left(\sum_{\{j:(i,j)\in E\}}x_{ij}-\sum_{\{j:(j,i)\in E\}}x_{ji}\right)=z(x)+\sum_{i\in V}\pi_i b_i$$

For a fixed π, the difference $z(x,\pi)-z(x)$ is constant. Therefore, a flow that minimizes $z(x,\pi)$ also minimizes $z(x)$ and vice versa. We have proved:

Theorem 1. *For any node potential π the minimum cost flow problems with edge costs c_{ij} or c_{ij}^{π} have the same optimal solutions. Moreover,*

$$z(x,\pi)=z(x)+\sum_{i\in V}\pi_i b_i$$

The following result contains very useful properties of reduced costs.

***Theorem 2**. Let G be a transportation network. Suppose P is a directed path from $a \in V$ to $b \in V$. Then for any node potential π*

$$\sum_{(i,j)\in P} c_{ij}^{\pi} = \sum_{(i,j)\in P} c_{ij} + \pi_a - \pi_b$$

Suppose W is a directed cycle. Then for any node potential π

$$\sum_{(i,j)\in W} c_{ij}^{\pi} = \sum_{(i,j)\in W} c_{ij}$$

This theorem implies the following reasoning. Let's introduce a vertex s and for each node $i \in V$, we add an arc (s,i) to G with some positive capacity and zero cost. Suppose that for each $i \in V$ number π_i denotes length of the shortest path from s to i with respect to cost function c. (Reminder: there is no negative length cycle). If so, one can justify (or read it in [2]) that for each $(i,j) \in E$ the shortest path optimality condition is satisfied:

$$\pi_j \le \pi_i + c_{ij}$$

Since, $c_{ij}^{\pi} = c_{ij} + \pi_i - \pi_j$ and $0 \le \pi_i - \pi_j + c_{ij}$ yields $c_{ij}^{\pi} \ge 0$. Moreover, applying theorem 2, we can note that if G contains negative cycle, it will be negative for any node potential π in reduced network. So, if the transportation network has no negative cycle, we will be able to reduce costs and make them positive by finding the shortest paths from the introduced vertex s, otherwise, our assumption doesn't work. If the reader asks how to find the shortest path in graph with negative costs, I'll refer you back to the basics of the graph theory. One can use Bellman-Ford (label-correcting) algorithm to achieve this goal [1, 2].

Remember this reduced cost technique, since it appears in many applications and other algorithms (for example, Johnson's algorithm for all pair shortest path in sparse networks uses it [2]).

***Assumption 4**. The supply/demand at the vertexes satisfy the condition $\sum_{i\in V} b_i = 0$ and the minimum cost flow problem has a feasible solution.*

This assumption is a consequence of the "Finding a Solution" section of this article. If the network doesn't satisfy the first part of this assumption, we can either say that the problem has no solution or make corresponding transformation according to the steps outlined in that section. If the second part of the assumption isn't met then the solution doesn't exist.

By making these assumptions we do transform our original transportation network. However, many problems are often given in such a way which satisfies all the assumptions.

Now that all the preparations are behind us, we can start to discuss the algorithms in Part 2.

References

[1] Ravindra K. Ahuja, Thomas L. Magnanti, and James B. Orlin. *Network Flows: Theory, Algorithms, and Applications*.
[2] Thomas H. Cormen, Charles E. Leiserson, Ronald L. Rivest. *Introduction to Algorithms*.
[3] **_efer_**. Algorithm Tutorial: Maximum Flow.
[4] **gladius**. Algorithm Tutorial: Introduction to graphs and their data structures: Section 3.

Algorithm Tutorials

Minimum Cost Flow, Part 2: Algorithms

Archive
Normal view
Discuss this article
Write for TopCoder

By Zealint
TopCoder Member

In Part 1, we looked at the basics of minimum cost flow. In this section, we'll look at three algorithms that can be applied to minimum cost flow problems.

Working with Residual Networks

Let's consider the concept of residual networks from the perspective of min-cost flow theory. You should be familiar with this concept thanks to maximum flow theory, so we'll just extend it to minimum cost flow theory.

We start with the following intuitive idea. Let G be a network and x be a feasible solution of the minimum cost flow problem. Suppose that an edge (i,j) in E carries x_{ij} units of flow. We define the residual capacity of the edge (i,j) as $r_{ij} = u_{ij} - x_{ij}$. This means that we can send an additional r_{ij} units of flow from vertex i to vertex j. We can also cancel the existing flow x_{ij} on the arc if we send up x_{ij} units of flow from j to i over the arc (i,j). Now note that sending a unit of flow from i to j along the arc (i,j) increases the objective function by c_{ij}, while sending a unit of flow from j to i on the same arc decreases the flow cost by c_{ij}.

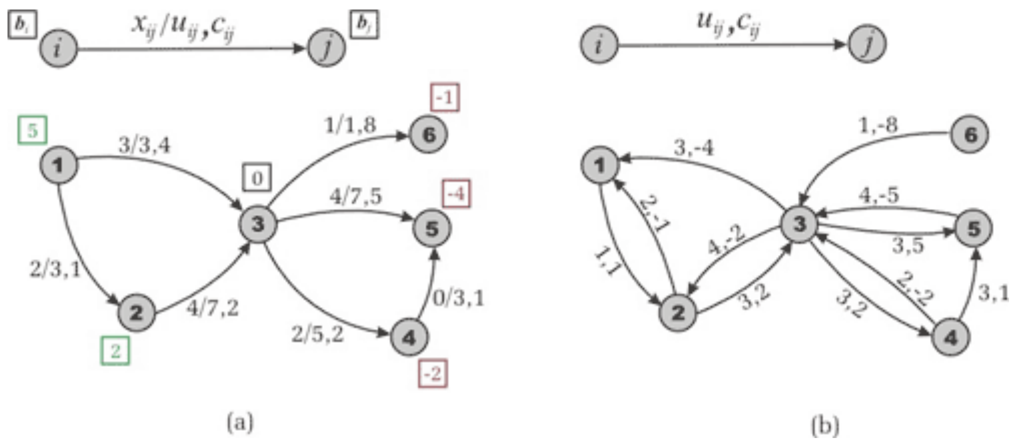

Figure 1. The transportation network from Part 1. (a) A feasible solution. (b) The residual network with respect to the found feasible solution.

Based on these ideas we define the residual network with respect to the given flow x as follows. Suppose we have a transportation network $G = (V,E)$. A feasible solution x engenders a new (residual) transportation network, which we are used to defining by $G_x = (V,E_x)$, where E_x is a set of residual edges corresponding to the feasible solution x.

What is E_x? We replace each arc (i,j) in E by two arcs (i,j), (j,i): the arc (i,j) has cost c_{ij} and (residual) capacity $r_{ij} = u_{ij} - x_{ij}$, and the arc (j,i) has cost $-c_{ij}$ and (residual) capacity $r_{ji}=x_{ij}$. Then we construct the set E_x from the new edges with a positive residual capacity. Look at Figure 1 to make sure that you understand the construction of the residual network.

You can notice immediately that such a definition of the residual network has some technical difficulties. Let's sum them up:

- If G contains both the edges (i,j) and (j,i) (remember assumption 2) the residual network may contain four edges between i and j (two parallel arcs from i to j and two contrary). To avoid this situation we have two options. First, transform the original network to one in which the network contains either edge (i,j) or edge (j,i), but not both, by splitting the vertexes i and j. Second, represent our network by the adjacency list, which is handling parallel arcs. We could even use two adjacency matrixes if it were more convenient.
- Let's imagine now that we have a lot of parallel edges from i to j with different costs. Unfortunately, we can't merge them by summarizing their capacities, as we could do while we were finding the maximum flow. So, we need to keep each of the parallel edges in our data structure separate.

The proof of the fact that there is a one-to-one correspondence between the original and residual networks is out the scope of this article, but you could prove all the necessary theorems as it was done within the maximum flow theory, or by reading [1].

Cycle-canceling Algorithm

This section describes the negative cycle optimality conditions and, as a consequence, cycle-canceling algorithm. We are starting with this important theorem:

***Theorem 1 (Solution Existence)**. Let G be a transportation network. Suppose that G contains no uncapacitated negative cost cycle and there exists a feasible solution of the minimum cost flow problem. Then the optimal solution exists.*

Proof. One can see that the minimum cost flow problem is a special case of the linear programming problem. The latter is well known to have an optimal solution if it has a feasible solution and its objective function is bounded. Evidently, if G doesn't contain an uncapacitated negative cycle then the objective function of the minimum cost flow problem is bounded from below -- therefore, the assertion of the theorem follows forthwith.

We will use the following theorem without proof, because we don't want our article to be overloaded with difficult theory, but you can read the proof in [1].

***Theorem 2 (Negative Cycle Optimality Conditions)**. Let x^* be a feasible solution of a minimum cost flow problem. Then x^* is an optimal solution if and only if the residual network G_{x^*} contains no negative cost (directed) cycle.*

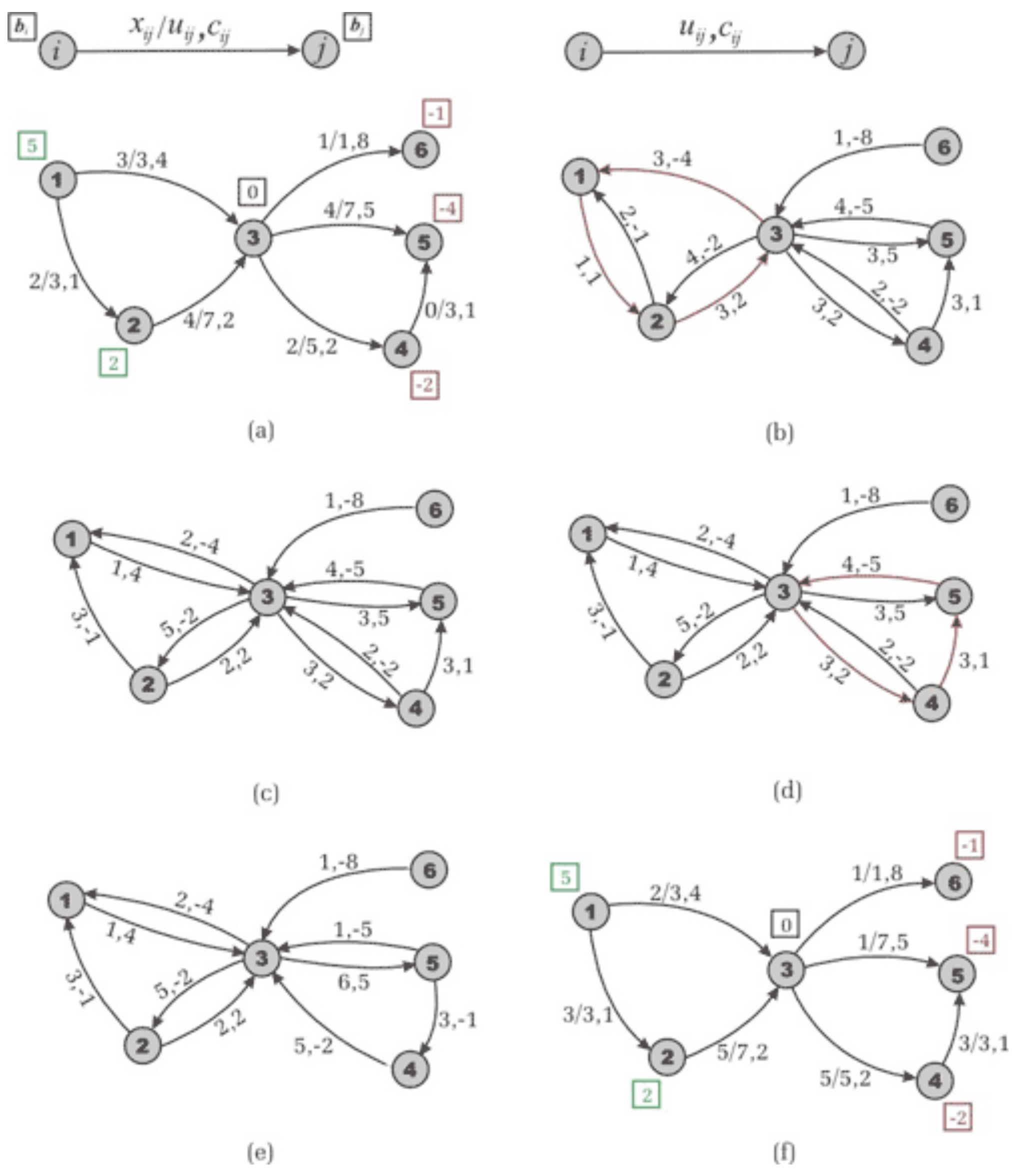

Figure 2. Cycle-Canceling Algorithm, example of the network from Figure 1. (a) We have a feasible solution of cost 54. (b) A negative cycle 1-2-3-1 is detected in the residual network. Its cost is -1 and capacity is 1. (c) The residual network after augmentation along the cycle. (d) Another negative cost cycle 3-4-5-3 is detected. It has cost -2 and capacity 3. (e) The residual network after augmentation. It doesn't contain negative cycles. (f) Optimal flow cost value is equal to 47.

This theorem gives the cycle-canceling algorithm for solving the minimum cost flow problem. First, we use any maximum flow algorithm [3] to establish a feasible flow in the network (remember assumption 4). Then the algorithm attempts to improve the objective function by finding negative cost cycles in the residual network and augmenting the flow on these cycles. Let us specify a program in pseudo code like it is done in [1].

```
Cycle-Canceling

1    Establish a feasible flow $x$ in the network
2    while ( $G_x$ contains a negative cycle ) do
3        identify a negative cycle $W$
4        $\delta \leftarrow \min\{r_{ij} : (i,j) \in W\}$
5        augment $\delta$ units of flow along the cycle $W$
```

```
6        update $G_x$
```

How many iterations does the algorithm perform? First, note that due to assumption 1 all the data is integral. After line 1 of the program we have an integral feasible solution x. It implies the integrality of G_x. In each iteration of the cycle in line 2 the algorithm finds the minimum residual capacity in the found negative cycle. In the first iteration δ will be an integer. Therefore, the modified residual capacities will be integers, too. And in all subsequent iterations the residual capacities will be integers again. This reasoning implies:

Theorem 3 (Integrality Property)*. If all edge capacities and supplies/demands on vertexes are integers, then the minimum cost flow problem always has an integer solution.*

The cycle-canceling algorithm works in cases when the minimum cost flow problem has an optimal solution and all the data is integral and we don't need any other assumptions.

Now let us denote the maximum capacity of an arc by U and its maximum absolute value of cost by C. Suppose that m denotes the number of edges in G and n denotes the number of vertexes. For a minimum cost flow problem, the absolute value of the objective function is bounded by mCU. Any cycle canceling decreases the objective function by a strictly positive amount. Since we are assuming that all data is integral, the algorithm terminates within $O(mCU)$ iterations. One can use $O(nm)$ algorithm for identifying a negative cycle (for instance, Bellman-Ford's algorithm or label correcting algorithm [1]), and obtain complexity $O(nm^2CU)$ of the algorithm.

Successive Shortest Path Algorithm

The previous algorithm solves the maximum flow problem as a subtask. The successive shortest path algorithm searches for the maximum flow and optimizes the objective function simultaneously. It solves the so-called max-flow-min-cost problem by using the following idea.

Suppose we have a transportation network G and we have to find an optimal flow across it. As it is described in the "Finding a Solution" section we transform the network by adding two vertexes s and t (source and sink) and some edges as follows. For each node i in V with $b_i > 0$, we add a source arc (s,i) with capacity b_i and cost 0. For each node i in V with $b_i < 0$, we add a sink arc (i,t) with capacity $-b_i$ and cost 0.

Then, instead of searching for the maximum flow as usual, we send flow from s to t along the shortest path (with respect to arc costs). Next we update the residual network, find another shortest path and augment the flow again, etc. The algorithm terminates when the residual network contains no path from s to t (the flow is maximal). Since the flow is maximal, it corresponds to a feasible solution of the original minimum cost flow problem. Moreover, this solution will be optimal (and we are going to explain why).

The successive shortest path algorithm can be used when G contains no negative cost cycles. Otherwise, we cannot say exactly what "the shortest path" means. Now let us justify the successive shortest path approach. When the current flow has zero value, the transportation network G doesn't contain a negative cost cycle (by hypothesis). Suppose that after some augmenting steps we have flow x and G_x still contains no negative cycles. If x is maximal then it is optimal, according to theorem 2. Otherwise, let us denote the next successfully found shortest path in G_x by P.

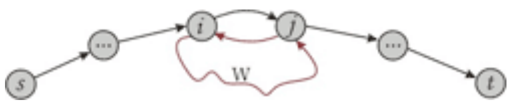

Figure 3. How could a negative cycle appear in a residual network?

Suppose that after augmenting the current flow x along path P a negative cost cycle W turned up in the residual network. Before augmenting there were no negative cycles. This means that there was an edge (i,j) in P (or subpath $(i,\dots,j)$ in P) the reversal of which (j,i) closed cycle W after the augmentation. Evidently, we could choose another path from s to t, which goes from s to i then from i to j along edges of W then from j to t. Moreover, the cost of this path is less than the cost of P. We have a contradiction to the supposition that P is the shortest.

What do we have? After the last step we have a feasible solution and the residual network contains no negative cycle. The latter is the criterion of optimality.

A simple analysis shows that the algorithm performs at most $O(nB)$ augmentations, where B is assigned to an upper bound on the largest supply of any node. Really, each augmentation strictly decreases the residual capacity of a source arc (which is equal to the supply of the corresponding node). Thanks to the integrality property it decreases by at least one unit. By using an $O(nm)$ algorithm for finding a shortest path (there may be negative edges), we achieve an $O(n^2mB)$ complexity of the successive shortest path algorithm.

```
Successive Shortest Path
1    Transform network $G$ by adding source and sink
2    Initial flow $x$ is zero
3    **while** ( $G_x$ contains a path from $s$ to $t$ ) **do**
4        Find any shortest path $P$ from $s$ to $t$
5        Augment current flow $x$ along $P$
6        update $G_x$
```

Let us reveal the meaning of node potentials from assumption 3. As it is said within assumption 3, we are able to make all edge costs nonnegative by using, for instance, Bellman-Ford's algorithm. Since working with residual costs doesn't change shortest paths (by theorem 2, part 1) we can work with the transformed network and use Dijkstra's algorithm to find the successive shortest path more efficiently. However, we need to keep the edge costs nonnegative on each iteration -- for this purpose, we update node potentials and reduce costs right after the shortest path has been found. The reduce cost function could be written in the following manner:

```
Reduce Cost ( $\pi$ )
1    **For each** $(i,j)$ in $E_x$ **do**
2            $c_{ij} \leftarrow c_{ij} + \pi_i - \pi_j$
3            $c_{rev(i,j)} \leftarrow 0$
```

Having found the successive shortest path we need to update node potentials. For each i in V the potential π_i is equal to the length of the shortest paths from s to t. After having reduced the cost of each arc, we will see that along the shortest path from s to i arcs will have zero cost while the arcs which lie out of any shortest path to any vertex will have a positive cost. That is why we assign zero cost to any reversal arc $(c_{rev(i,j)})$ in the Reduce Cost Procedure in line 3. The augmentation (along the found path) adds reversal arc (j,i) and due to the fact that (reduced) cost $c_{ij} = 0$ we make $(c_{rev(i,j)}) = 0$ beforehand.

Why have we denoted cost of reversal arc by $(c_{rev(i,j)})$ instead of c_{ji}? Because the network may contain both arcs (i,j) and (j,i) (remember assumption 2 and "Working with Residual Networks" section). For other arcs (which lie out of the augmenting path) this forcible assignment does nothing, because its reversal arcs will not appear in the residual network. Now we propose a pseudo-code program:

```
Successive Shortest Path with potentials
```

```
1   Transform network $G$ by adding source and sink
2   Initial flow $x$ is zero
3   Use Bellman-Ford's algorithm to establish potentials $\pi$
4   Reduce Cost ( $\pi$ )
5   while ( $G_x$ contains a path from $s$ to $t$ ) do
6       Find any shortest path $P$ from $s$ to $t$
7       Reduce Cost ( $\pi$ )
8       Augment current flow $x$ along $P$
9       update $G_x$
```

Before starting the cycle in line 5 we calculate node potentials π and obtain all costs to be nonnegative. We use the same massif of costs c when reducing. In line 6 we use Dijkstra's algorithm to establish a shortest path with respect to the reduced costs. Then we reduce costs and augment flow along the path. After the augmentation all costs will remain nonnegative and in the next iteration Dijkstra's algorithm will work correctly.

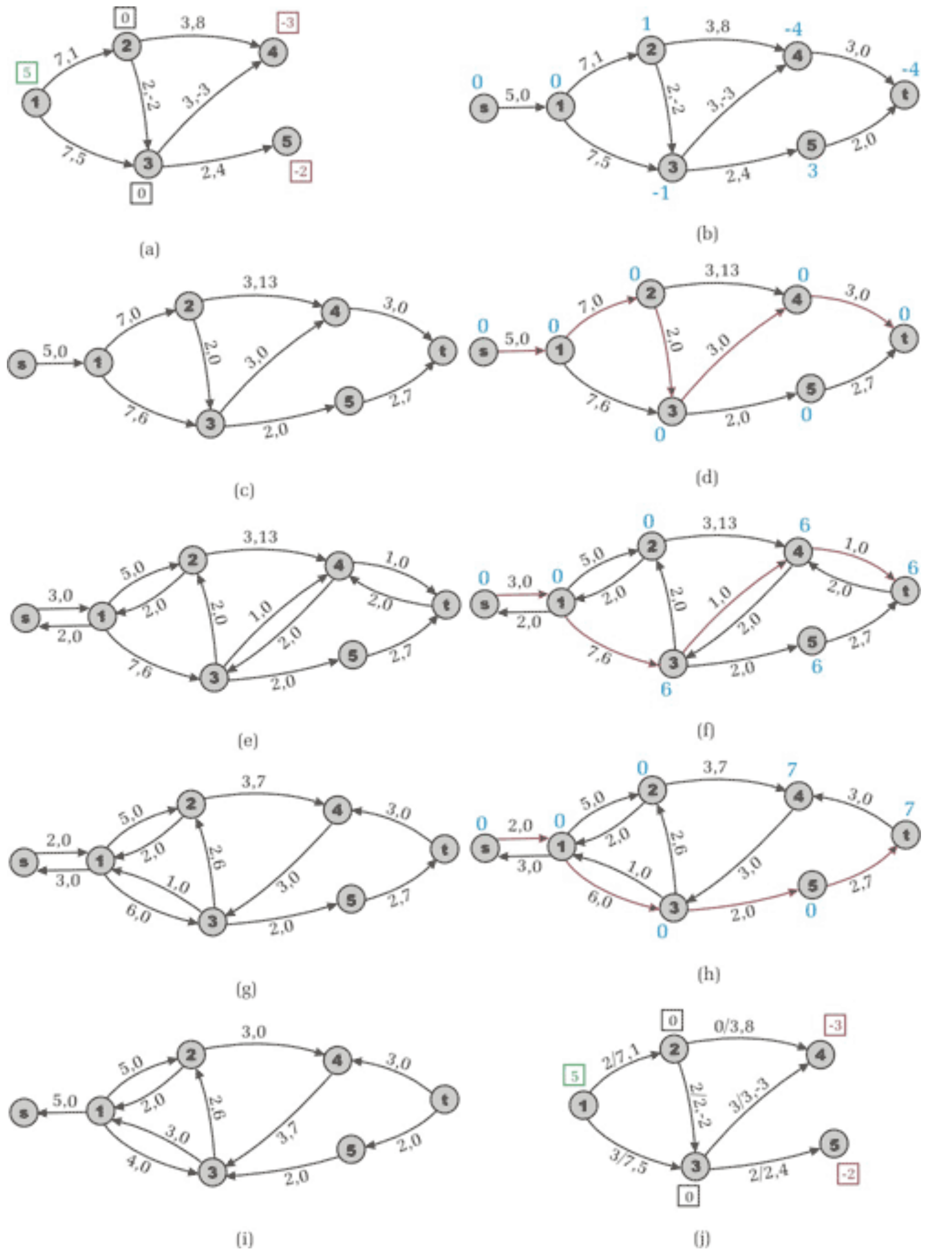

Figure 4. The Successive shortest Path Algorithm. (a) Initial task. (b) Node potentials are calculated after line 3 of the program. (c) Reduced costs after line 4. (d) The first augmenting path s-1-2-3-4-t of capacity 2 is found and new node potentials are calculated. (e) The residual network with reduced costs. (f) The second augmenting path s-1-3-4-t of capacity 1 is found. (g) The residual network with reduced costs. (h) The third shortest augmenting path s-1-3-5-t and new node potentials are found. (i) The residual network contains no augmenting paths. (j) The reconstructed transportation network. Optimal flow has cost 12.

We use Bellman-Ford's algorithm only once to avoid negative costs on edges. It takes $O(nm)$ time. Then $O(nB)$ times we use Dijkstra algorithm, which takes either $O(n^2)$ (simple realization) or $O(m\log n)$ (heap realization for sparse network, [4]) time. Summing up, we receive $O(n^3B)$ estimate working time for simple realization and $O(nmB\log n)$ if using heap. One could even use Fibonacci Heaps to obtain $O(n\log n+m)$ complexity of Dijkstra's shortest path algorithm; however I wouldn't recommend doing so because this case works badly in practice.

Primal-Dual Algorithm

The primal-dual algorithm for the minimum cost flow problem is similar to the successive shortest path algorithm in the sense that it also uses node potentials and shortest path algorithm to calculate them. Instead of augmenting the flow along one shortest path, however, this algorithm increases flow along all the shortest paths at once. For this purpose in each step it uses any maximum flow algorithm to find the maximum flow through the so called **admissible network**, which contains only those arcs in G_x with a zero reduced cost. We represent the admissible residual network with respect to flow x as G_x°. Let's explain the idea by using a pseudo-code program.

```
Primal-Dual
1    Transform network $G$ by adding source and sink
2    Initial flow $x$ is zero
3    Use Bellman-Ford's algorithm to establish potentials $\pi$
4    Reduce Cost ( $\pi$ )
5    while ( $G_x$ contains a path from $s$ to $t$ ) do
6        Calculate node potential $\pi$ using Dijkstra's algorithm
7        Reduce Cost ( $\pi$ )
8        Establish a maximum flow $y$ from $s$ to $t$ in $G_x^\circ$
9        $x \leftarrow x + y$
10       update $G_x$
```

For a better illustration look at Figure 5.

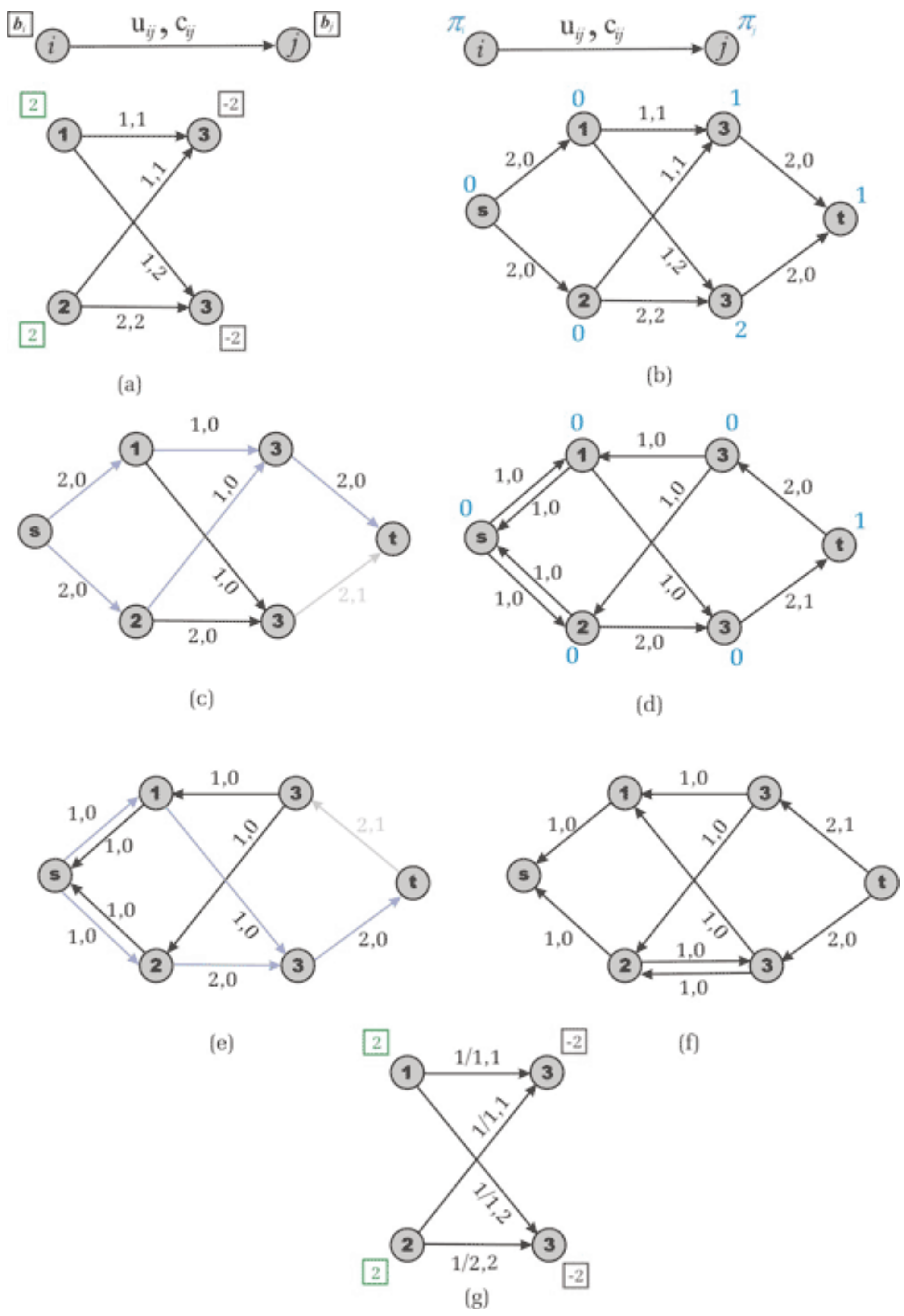

Figure 5. Primal-Dual algorithm. (a) Example network. (b) Node potentials are calculated. (c) The maximum flow in the admissible network. (d) Residual network and new node potentials. (e) The maximum flow in the admissible network. (f) Residual network with no augmenting paths. (g) The optimal solution.

As mentioned above, the primal-dual algorithm sends flow along all shortest paths at once; therefore, proof of correctness is similar to the successive shortest path one.

First, the primal-dual algorithm guarantees that the number of iterations doesn't exceed $O(nB)$ as well as the successive shortest path algorithm. Moreover, since we established a maximum flow in G_x°, the residual network

G_x contains no directed path from vertex s to vertex t consisting entirely of arcs of zero costs. Consequently, the distance between s and t increases by at least one unit. These observations give a bound of $min\{nB,nC\}$ on the number of iterations which the primal-dual algorithm performs. Keep in mind, though, that the algorithm incurs the additional expense of solving a maximum flow problem at every iteration. However, in practice both the successive shortest path and the primal-dual algorithm work fast enough within the constraint of 50 vertexes and reasonable supply/demand values and costs.

In the next section, we'll discuss some applications of the minimum cost flow problem.

References

[1] Ravindra K. Ahuja, Thomas L. Magnanti, and James B. Orlin. *Network Flows: Theory, Algorithms, and Applications*.
[2] Thomas H. Cormen, Charles E. Leiserson, Ronald L. Rivest. *Introduction to Algorithms*.
[3] **_efer_**. Algorithm Tutorial: Maximum Flow.
[4] **gladius**. Algorithm Tutorial: Introduction to graphs and their data structures: Section 3.

Algorithm Tutorials

Minimum Cost Flow, Part 3: Applications

Archive
Normal view
Discuss this article
Write for TopCoder

By **Zealint**
TopCoder Member

The last part of the article introduces some well known applications of the minimum cost flow problem. Some of the applications are described according to [1].

The Assignment Problem

There are a number of *agents* and a number of *tasks*. Any agent can be assigned to perform any task, incurring some cost that may vary depending on the agent-task assignment. We have to get all tasks performed by assigning exactly one agent to each task in such a way that the total cost of the assignment is minimal with respect to all such assignments.

In other words, consider we have a square matrix with n rows and n columns. Each cell of the matrix contains a number. Let's denote by c_{ij} the number which lays on the intersection of i-th row and j-th column of the matrix. The task is to choose a subset of the numbers from the matrix in such a way that each row and each column has exactly one number chosen and sum of the chosen numbers is as minimal as possible. For example, assume we had a matrix like this:

$$\begin{pmatrix} 1 & \overset{*}{3} & 2 \\ 2 & 6 & \overset{*}{4} \\ \overset{*}{3} & 7 & 6 \end{pmatrix}$$

In this case, we would chose numbers 3, 4, and 3 with sum 10. In other words, we have to find an integral solution of the following linear programming problem:

$$\text{Minimize } z(x) = \sum_{i=1}^{n}\sum_{j=1}^{n} c_{ij}x_{ij}$$

subject to

$$\sum_{j=1}^{n} x_{ij} = 1 \quad \text{for all } i = 1,\ldots,n,$$

$$\sum_{i=1}^{n} x_{ij} = 1 \quad \text{for all } j = 1,\ldots,n,$$

$$0 \le x_{ij} \le 1, \; x_{ij} \text{ is integral} \quad \text{for all } i,j = 1,\ldots,n.$$

If binary variable $x_{ij} = 1$ we will choose the number from cell (i,j) of the given matrix. Constraints guarantee that each row and each column of the matrix will have only one number chosen. Evidently, the problem has a feasible solution (one can choose all diagonal numbers). To find the optimal solution of the problem we construct the bipartite transportation network as it is drawn in Figure 1. Each edge (i,j') of the graph has unit capacity and cost c_{ij}. All supplies and demands are equal to 1 and -1 respectively. Implicitly, minimum cost flow solution corresponds to the optimal assignment and vise versa. Thanks to *left-to-right* directed edges the network contains no negative cycles and one is able to solve it with complexity of $O(n^3)$. Why? *Hint: use the successive shortest path algorithm.*

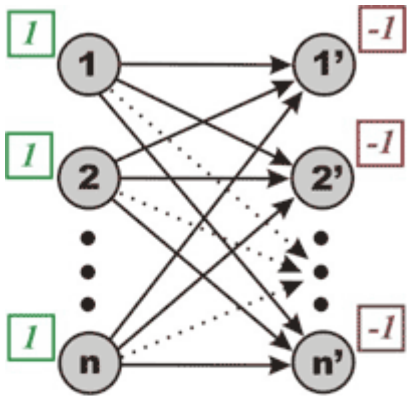

Figure 1. Full weighted bipartite network for the assignment problem. Each edge has capacity 1 and cost according to the number in the given matrix.

The assignment problem can also be represented as weight matching in a weighted bipartite graph. The problem allows some extensions:

- Suppose that there is a different number of supply and demand nodes. The objective might be to find a maximum matching with a minimum weight.
- Suppose that we have to choose not one but k numbers in each row and each column. We could easily solve this task if we considered supplies and demands to be equal to k and $-k$ (instead of 1 and -1) respectively.

However, we should point out that, due to the specialty of the assignment problem, there are more effective algorithms to solve it. For instance, the Hungarian algorithm has complexity of $O(n^3)$, but it works much more quickly in practice.

Discrete Location Problems

Suppose we have n building sites and we have to build n new facilities on these sites. The new facilities interact with m existing facilities. The objective is to assign each new facility i to the available building site j in such a way that minimizes the total transportation cost between the new and existing facilities. One example is the location of hospitals, fire stations etc. in the city; in this case we can treat population concentrations as the existing facilities.

Let's denote by d_{kj} the distance between existing facility k and site j; and the total transportation cost per unit distance between the new facility i and the existing one k by w_{ik}. Let's denote the assignment by binary variable x_{ij}. Given an assignment x we can get a corresponding transportation cost between the new facility i and the existing facility k:

$$w_{ik} \sum_{j=1}^{n} d_{kj} x_{ij}$$

Thus the total transportation cost is given by

$$z(x) = \sum_{i=1}^{n}\sum_{k=1}^{m} w_{ik} \sum_{j=1}^{n} d_{kj} x_{ij} = \sum_{i=1}^{n}\sum_{j=1}^{n}\left(\sum_{k=1}^{m} w_{ik} d_{kj}\right) x_{ij}$$

Note, that $c_{ij} = \sum_{k=1}^{m} w_{ik} d_{kj}$ is the cost of locating the new facility i at site j. Appending necessary conditions, we obtain another instance of the assignment problem.

The Transportation Problem

A minimum cost flow problem is well known to be a *transportation problem in the statement of network*. But there is a special case of transportation problem which is called *the transportation problem in statement of matrix*. We can obtain the optimization model for this case as follows.

$$\text{Minimize } z(x) = \sum_{i=1}^{m}\sum_{j=1}^{n} c_{ij} x_{ij}$$

subject to

$$\sum_{j=1}^{n} x_{ij} = b_i \quad \text{for all } i = 1,\dots,m,$$
$$\sum_{i=1}^{m} x_{ij} = d_j \quad \text{for all } j = 1,\dots,n,$$
$$0 \le x_{ij} \le u_{ij}, \quad \text{for all } i = 1,\dots,m \text{ and } j = 1,\dots,n$$

For example, suppose that we have a set of m warehouses and a set of n shops. Each warehouse i has nonnegative supply value b_i while each shop j has nonnegative demand value d_j. We are able to transfer goods from a warehouse i directly to a shop j by the cost c_{ij} per unit of flow.

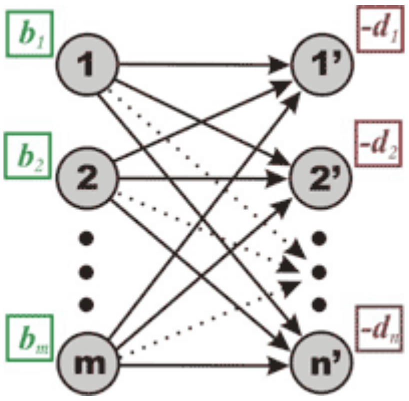

Figure 2. Formulating the transportation problem as a minimum cost flow problem. Each edge connecting a vertex i and a vertex j' has capacity u_{ij} and cost c_{ij}.

There is an upper bound to the amount of flow between each warehouse i and each shop j denoted by u_{ij}. Minimizing total transportation cost is the object. Representing the flow from a warehouse i to a shop j by x_{ij} we obtain the model above. Evidently, the assignment problem is a special case of the transportation problem in the statement of matrix, which in turn is a special case of the minimum cost flow problem.

Optimal Loading of a Hopping Airplane

We took this application from [1]. A small commuter airline uses a plane with the capacity to carry at most p passengers on a "hopping flight." The hopping flight visits the cities 1, 2, …, n, in a fixed sequence. The plane can pick up passengers at any node and drop them off at any other node.

Let b_{ij} denote the number of passengers available at node i who want to go to node j, and let f_{ij} denote the fare per passenger from node i to node j.

The airline would like to determine the number of passengers that the plane should carry between the various origins and destinations in order to maximize the total fare per trip while never exceeding the plane capacity.

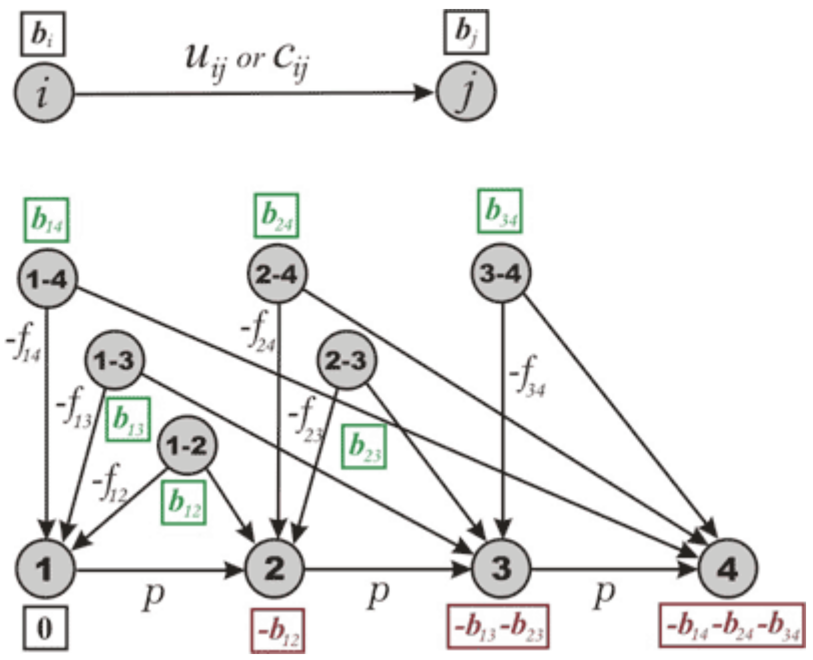

Figure 3. Formulating the hopping plane flight problem as a minimum cost flow problem.

Figure 3 shows a minimum cost flow formulation of this hopping plane flight problem. The network contains data for only those arcs with nonzero costs and with finite capacities: Any arc without an associated cost has a zero cost; any arc without an associated capacity has an infinite capacity.

Consider, for example, node 1. Three types of passengers are available at node 1, those whose destination is node 2, node 3, or node 4. We represent these three types of passengers by the nodes 1-2, 1-3, and 1-4 with supplies b_{12}, b_{13}, and b_{14}. A passenger available at any such node, say 1-3, either boards the plane at its origin node by flowing though the arc $(1\text{-}3,1)$ and thus incurring a cost of $-f13$ units, or never boards the plane which we represent by the flow through the arc $(1\text{-}3,3)$.

We invite the reader to establish one-to-one correspondence between feasible passenger routings and feasible flows in the minimum cost flow formulation of the problem.

Dynamic Lot Sizing

Here's another application that was first outlined in [1]. In the dynamic lot-size problem, we wish to meet prescribed demand d_j for each of K periods $j = 1, 2, \ldots, K$ by either producing an amount a_j in period j and/or by drawing upon the inventory I_{j-1} carried from the previous period. Figure 4 shows the network for modeling this problem.

The network has $K+1$ vertexes: The j-th vertex, for $j = 1, 2, \ldots, K$, represents the j-th planning period; node 0 represents the "source" of all production. The flow on the "production arc" $(0,j)$ prescribes the production level a_j

in period j, and the flow on "inventory carrying arc" $(j,j+1)$ prescribes the inventory level I_j to be carried from period j to period $j+1$.

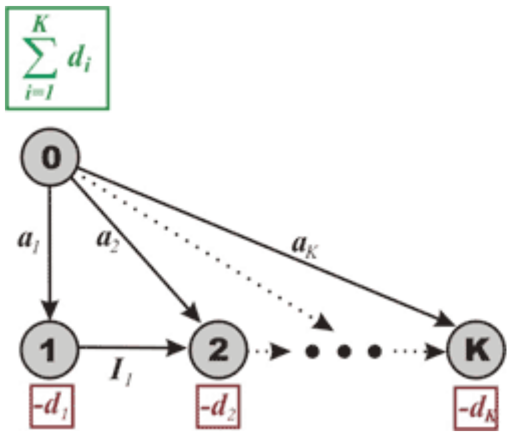

Figure 4. Network flow model of the dynamic lot-size problem.

The mass balance equation for each period j models the basic accounting equation: Incoming inventory plus production in that period must equal the period's demand plus the final inventory at the end of the period. The mass balance equation for vertex 0 indicates that during the planning periods 1, 2, …, K, we must produce all of the demand (we are assuming zero beginning and zero final inventory over the planning horizon).

If we impose capacities on the production and inventory in each period and suppose that the costs are linear, the problem becomes a minimum cost flow model.

References

[1] Ravindra K. Ahuja, Thomas L. Magnanti, and James B. Orlin. *Network Flows: Theory, Algorithms, and Applications*.
[2] Thomas H. Cormen, Charles E. Leiserson, Ronald L. Rivest. *Introduction to Algorithms*.
[3] **_efer_**. Algorithm Tutorial: Maximum Flow.
[4] **gladius**. Algorithm Tutorial: Introduction to graphs and their data structures: Section 3.