Network optimization: An overview

Mathias Johanson
Alkit Communications

1 Introduction

Various kinds of network optimization problems appear in many fields of work, including telecommunication systems, commodity transportation, railroad and highway traffic planning, electrical power distribution, and much more. The fundamental question in network optimization is how to efficiently transport some entity (data packets, electrical power, vehicles, etc.) from one point to another in a network, given a number of limiting constraints, such as the capacity of the communication links of the network. General optimization problems can be approached in many different ways, e.g. using linear programming, operations research theory, discrete simulation, and using algorithmic approaches from the computer science field. This overview introduces some of the fundamental concepts, algorithms and applications of network optimization theory, using a computer science perspective. The focus is on computer networks, but the theories and algorithms discussed are applicable also in other domains.

2 Network optimization problems

Many of the most important network optimization problems can be related to the following general problems:

- *Shortest path problems*
  How can we get from one point to another in a network using the shortest (or cheapest) path?
- *Maximum flow problems*
  How can we achieve as high flows as possible between two points in a network, given some link capacity restrictions?
- *Minimum cost flow problems*
  Given a cost per unit flow on each link in a network, how can we assign flows to the links in the most cost effective way?

A large number of related problems can be derived from the above-mentioned general problems, including assignment problems, transportation problems, circulation problems, convex cost flow problems, multi-commodity flow problems, minimum spanning tree problems, and matching problems.

It can be shown that virtually all network flow problems can be transformed into one another. Hence, a solution to one of the problems is a solution to all others, using a suitable transform. Specifically, the shortest path problem and the maximum flow problem can easily be stated as special cases of the minimum cost flow problem (MCFP). Therefore, the MCFP is considered the most fundamental network flow problem, and the design of algorithms for network flow problems are mainly targeting the MCFP.

2.1 Formulation of the minimum cost flow problem

Let (N, A) be a directed graph defined by a set N of n nodes and a set A of m directed arcs. Each arc (i, j) in A has an associated cost c_{ij} denoting the cost per unit flow on that arc. The upper and lower bounds on the flow that can be supported by each link (arc) is denoted by l_{ij} and u_{ij} respectively. The supply/demand of node i in the network is denoted by $b(i)$. The MCFP can then be stated mathematically as follows:

$$\text{Minimize} \sum_{(i,j)\in A} c_{ij}x_{ij} \quad \text{subject to} \sum_{\{j:(i,j)\in A\}} x_{ij} - \sum_{\{j:(i,j)\in A\}} x_{ji} = b(i), \quad \forall i \in N,$$

$$\text{where } l_{ij} \le x_{ij} \le u_{ij}, \forall (i,j) \in A, \text{ and } \sum_{i=1}^{n} b(i) = 0. \qquad (1)$$

The variable x_{ij} represents the flow on arc $(i, j)\in A$.

3 Some basic concepts and terminology

Strong connectivity: A network is strongly connected if for every pair of nodes i and j the network contains a directed path from node i to node j.

Topological ordering: A labeling of the nodes of a directed graph (with monotonically increasing integers) is a topological ordering of nodes if every arc joins a lower-labeled node to a higher-labeled node. A network that contains a directed cycle has no topological ordering.

Cut: A cut is a set of arcs of a graph that partitions the graph into two parts.

Tree: A tree is a connected graph that contains no cycle.

Forest: A forest is a graph that contains no cycle (i.e. it is a set of trees).

Residual network: The "remaining flow network" representing the feasible incremental flows that remain at an intermediate step of an iterative algorithm.

4 Algorithm development and analysis

The goal when designing network optimization algorithms is to arrive at an algorithm that solves the optimization problem in an efficient way, i.e. in polynomial time. Several approaches exists:

- *Geometric improvement techniques*
  An algorithm runs in polynomial time if at every iteration it makes an improvement in the objective function value that is proportional to the difference between the objective function values of the current (intermediary) solution and the optimal solution. That is, if the algorithm makes a significant contribution to the solution for each iteration, it will be efficient.
- *Dynamic programming*
  The dynamic programming strategy decomposes the problem into stages and uses a recursive relationship to go from one stage to another.
- *Scaling*
  The scaling approach solves a sequence of simpler approximate versions of a given problem, determined by scaling the problem data (for instance using bit scaling by increasing the precision one bit) in such a way that the approximations gradually approach the solution of the final problem.

A common trait of most algorithm design techniques for network optimization is an iterative *divide-and-conquer* approach that somehow seeks to gradually approach the final solution by solving smaller intermediary subtasks.

The efficiency of network optimization algorithms is usually measured using *big O, big Ω, and big Θ notation*:

- An algorithm is said to execute in $O(f(n))$ time if for some numbers c and n_0 the time taken by the algorithm is at most $cf(n)$ for all $n \geq n_0$.

- An algorithm is said to execute in $\Omega(f(n))$ time if for some numbers k and n_0 the time taken by the algorithm on some problem instance is at least $kf(n)$ for all $n \geq n_0$.

- An algorithm is said to be $\Theta(f(n))$ if the algorithm is both $O(f(n))$ and $\Omega(f(n))$.

Thus, O is an upper bound on algorithm complexity, Ω is a lower bound, and Θ is both an upper and a lower bound.

5 Shortest path problems

Shortest path problems arise frequently in many applications. In computer networks, routing algorithms rely heavily on shortest path computations. There are a number of variations of the shortest path problem, the most common (in computer networking at least) being the question of how to find the shortest path from one node to all other nodes in a network (the single-source shortest path problem with nonnegative arc lengths).

The shortest path problem can be formulated as a minimum cost flow problem by modifying $b(i)$ in equation (1) so that $b(s) = n - 1$, for the source node s, and $b(i) = -1$ for all other nodes in the network.

Shortest path algorithms can be classified into two classes: *label-setting algorithms* and *label-correcting algorithms*. Both progress iteratively assigning labels to nodes. The label of a node represents the upper bound on the shortest path from the source to the node. Label-setting algorithms designate one label as permanent (i.e. optimal) for each iteration of the algorithm, whereas label-correcting algorithms assign temporal labels that become permanent at the last step of the algorithm.

Label-setting algorithms are only applicable for acyclic networks with arbitrary arc length and for networks with nonnegative arc lengths. Label-correcting algorithms work for all types of networks. Since computer networks generally have nonnegative link costs, label-setting algorithms are typically preferred, since the complexity of label-correcting algorithms is higher. (Label-correcting algorithms are NP-complete.)

5.1 Dijkstra's algorithm

Dijkstra's algorithm is a label-setting algorithm that finds the shortest path from a source node to all other nodes in a network with nonnegative arc lengths. It does so by maintaining two lists: the list of *permanently* labeled nodes (the permanent list) and the list of *temporarily* labeled nodes (the temporary list). The labels represent the distance to the source node. For each iteration of the algorithm, one node is transferred from the temporary list to the permanent list. Initially all nodes are in the temporary list, with distance labels set to infinity for all nodes but the source which has a distance label of zero. Consequently, the permanent list is initially empty. For each iteration the node with the smallest distance label is transferred from the temporary list to the permanent list. Hence, the first node to be transferred is the source node having a zero distance label. Then, for each iteration, each node j adjacent to the node i being transferred to the permanent list, is investigated to see if its distance label $d(j)$ is greater than $d(i) + c_{ij}$, where c_{ij} is the cost of arc (i, j). If so, $d(j)$ is set to $d(i) + c_i$ and the shortest path from the source to node j is recorded as the path through node i. The algorithm iterates until all nodes have been transferred to the permanent list. Upon termination, the tree formed by the arcs recorded when updating the distance labels of the nodes will be a shortest path tree.

The heart of the algorithm is the observation that for each iteration of the algorithm, one node can always be transferred to the permanent list, that is, its optimal shortest path from the source can be found. This observation relies on the fact that since we always choose the node with the lowest distance label in the temporary list, there cannot be another node in the temporary list through which a shorter path from the source can be constructed compared to some path constructed only through nodes in the permanent list. If that was the case, that node should already have been transferred to the permanent list.

5.1.1 Complexity of Dijkstra's algorithm

The complexity of the most straightforward implementation of Dijkstra's algorithm is $O(n^2)$, since each node selection operation requires the temporary list to be scanned, and the node selection operation is performed n times. The distance update operation is performed m times, with each update operation requiring constant time. Each node can have at most $n - 1$ adjacent nodes, so $m < n^2$ and thus $O(m) < O(n^2)$.

The performance of the algorithm can easily be improved by maintaining a sorted temporary list. For instance, using a binary heap data structure the complexity can be reduced to $O(m \log n)$.

6 Maximum flow

The maximum flow problem is concerned with finding the maximum flow between a source node and a sink node, without exceeding the arc capacities of the network. The shortest path problem and the maximum flow problem are complementary and capture different aspects of the minimum cost flow problem: The shortest path problem involves arc costs but not arc capacities, whereas the maximum flow problem involves arc capacities but not arc costs. The maximum flow problem can be stated formally as follows:

Let (N, A) be a network with nonnegative arc capacities u_{ij} associated with each arc $(i, j) \in A$. Let s be a source node and t a sink node. Then the maximum flow problem is:

Maximize v subject to

$$\sum_{\{j:(i,j)\in A\}} x_{ij} - \sum_{\{j:(j,i)\in A\}} x_{ji} = \begin{cases} v \text{ for } i = s \\ 0 \text{ for all } i \in N - \{s \text{ and } t\} \\ -v \text{ for } i = t \end{cases}$$

$0 \le x_{ij} \le u_{ij}$ for each $(i, j) \in A$.

6.1 The augmenting path algorithm

A directed path from the source to the sink in the residual network is called an *augmenting path*. The residual capacity of an augmenting path is the minimum residual capacity of the arcs in the path. Whenever a network contains an augmenting path it is possible to send additional flow from the source to the sink. The *augmenting path algorithm* exploits this observation by identifying augmenting paths and augmenting flows on these paths until there is no more augmenting paths in the network.

The augmenting path algorithm operates on the *residual network*. The residual network is a virtual network formed by replacing the arcs of the original network with arcs representing the residual capacity, that is the arc (i, j) with capacity u_{ij} carrying a flow $x_{ij} \leq u_{ij}$ is replaced by an arc (i, j) with residual capacity $u_{ij} - x_{ij}$ and an arc (j, i) with capacity x_{ij}. The arc (j, i) in the residual network represents the amount of flow that can be reduced between i and j by reducing the flow x_{ij} on arc (i, j) in the original network.

6.1.1 The labeling algorithm

The most straightforward implementation of the augmenting path algorithm is the *labeling algorithm*. The labeling algorithm fans out from the source node to its adjacent nodes (in the residual network), labeling nodes as they are reached and recording the path they are reached through. For each iteration the algorithm has partitioned the nodes of the network into two groups: the labeled nodes and the unlabeled nodes. The labeled nodes are those that the algorithm has reach so far, and hence there exists a directed path from the source to each labeled node in the residual network. The algorithm iteratively selects a labeled node, fans out to its adjacent nodes labeling more nodes and so on. Eventually the sink node will be labeled. Then a maximum flow is augmented on the augmenting path, whereupon all the labels are erased and the algorithm repeats the procedure on the new residual network. The algorithm terminates when it has scanned all labeled nodes and the sink is still unlabeled. This implies that there is no longer a path from the source to the sink in the residual network.

It can be showed that the flow found by the labeling algorithm is a maximum flow. (The flow equals the capacity of the cut between the labeled nodes and the unlabeled nodes. Since the value of any flow is less than or equal to the capacity of any cut in the network, the flow must be a maximum flow, and the cut is a minimum cut.)

The labeling algorithm performs one augmentation per iteration. Each augmentation involves at most m arcs, so the complexity of the augmentation process is $O(m)$. The number of iterations performed depends on the arc capacities and the amount of flow that is augmented for each iteration. If the arc capacities are bounded above by U, the capacity of the cut $(s, N - \{s\})$ is at most nU. Therefore, the maximum flow value is bounded by nU. The algorithm increases the value of the augmenting flow with at least one unit for each iteration, so the algorithm will terminate within nU iterations. Consequently, the complexity of the labeling algorithm is $O(nmU)$.

6.2 The max-flow min-cut theorem

One of the most fundamental theorems of network flow theory is the *max-flow min-cut theorem*, which establishes the following intuitive property:

> *The maximum value of the flow from a source node s to a sink node t in a capacitated network equals the minimum capacity among all s-t cuts.*

A corollary of the max-flow min-cut theorem is the *augmenting path theorem*, stating the following:

> *A flow x is a maximum flow if and only if the residual network G(x) contains no augmenting path.*

The max-flow min-cut theorem tells us that by solving a maximum flow problem we also solve a complementary minimum cut problem (and vice versa). Finding the maximum flow between two nodes in the network is conceptually equivalent to finding the cut, among all cuts in the network separating the source and sink nodes, that has the least capacity. In fact, the max-flow min-cut theorem is a special case of the well-known *strong duality theorem* of linear programming.

6.3 Polynomial maximum flow algorithms

A problem with the labeling algorithm is that the complexity, $O(nmU)$, get prohibitively high if the arc capacities (U) are high. The example depicted in Figure 1 illustrates the problem. In the network of Figure 1, a is the source node and d is the destination node. The algorithm first augments a flow of one unit along the path a-c-b-d. Next, the algorithm augments a unit flow along the path a-b-c-d. Then the algorithm can choose to augment along the a-c-b-d path again, and then alternating between the two paths 100 times, augmenting a unit flow for each iteration. Clearly, a worst case scenario like this is a major drawback of the algorithm. (Replace 100 with an arbitrarily large capacity to make the example more dramatic.)

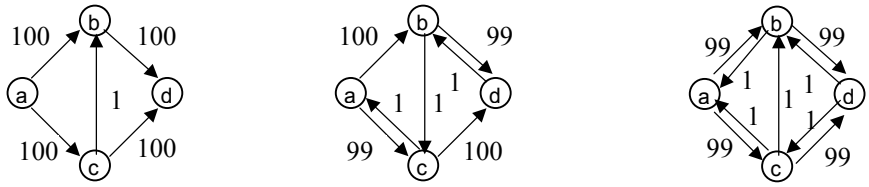

Figure 1 Example network illustrating shortcomings of the labeling algorithm

To avoid the performance problems that can arise with the labeling algorithm, more clever augmenting path algorithms have been devised. The *capacity scaling algorithm* solves the maximum flow problem in $O(m \log U)$ time, by requiring the flow that is augmented for each iteration to be "sufficiently large," in the style of the geometric improvement techniques, discussed in section 4.

The *shortest augmenting path algorithm* further reduces the complexity by always augmenting flow along a shortest path from the source to the sink in the residual network. The complexity of the shortest augmenting path algorithm is $O(n^2m)$.

7 Minimum cost flow algorithms

The minimum cost flow problem, which we stated formally in section 2.1, is a generalization of the shortest path and maximum flow problems. Many of the algorithms developed for the minimum cost flow problem combine techniques from both shortest path and maximum flow algorithms. Since the MCFP considers both arc capacities and arc costs it is harder to solve than the shortest path and maximum flow problems.

7.1 The cycle-canceling algorithm

One of the simplest algorithms for solving the MCFP is the *cycle-canceling algorithm*, which relies on the observation that a feasible solution to the MCFP is an optimal solution if and only if the residual network contains no negative cost (directed) cycle. This is true since if there is a negative cost cycle, the feasible flow can be augmented along the cycle, improving the objective function value and hence the feasible flow is not optimal. (The converse can also easily be shown to be true.)

The cycle-canceling algorithm establishes a feasible flow in the network by solving a maximum flow problem. Then it iteratively identifies negative cost cycles in the residual network, augmenting flow on these cycles until no more negative cost cycles exist. When the algorithm terminates, the feasible flow will be the minimum cost flow.

The complexity of the basic cycle-canceling algorithm is $O(nm^2CU)$, where C and U are upper bounds on the arc costs and capacities respectively. (Identifying negative cost cycles can be done in $O(nm)$ time, and mCU is the worst case number of iterations, since it is an upper bound on the initial flow cost and the objective function is changed by a negative amount for each iteration.)

By augmenting flow in the negative cost cycle with maximum improvement – or, alternatively, to augment flow along the negative cost cycle with minimum mean cost – the complexity of the cycle canceling algorithm can be improved to polynomial time.

7.2 The successive shortest path algorithm

Contrary to the cycle-canceling algorithm which maintains a feasible flow for each iteration and attempts to achieve optimality, the successive shortest path algorithm maintains an optimal flow for each step and strives for feasibility. The flow satisfies the nonnegativity and capacity constraints, but violates the mass balance constraints of the nodes (i.e. that the sum of the supply and demand for each node is zero). For each iteration the algorithm selects a node with excess supply and a node with unfulfilled demand and augments a flow between these two nodes along a "shortest path" in the residual network. The shortest path is computed using Dijkstra's algorithm with respect to the *reduced costs* of the arcs of the path. The reduced cost c_{ij} of path (i, j) with respect to a set of *node potentials* π is defined as $c_{ij}^{\pi} = c_{ij} - \pi(i) + \pi(j)$. The node potential $\pi(i)$ of node i is a real number used in much the same way as the distance labels that are used in the shortest path label-setting and label-correcting algorithms. The successive shortest path algorithm terminates when the flow satisfies all the mass balance constraints, and consequently the flow is a feasible (and optimal) solution.

7.3 The primal-dual algorithm and the out-of-kilter algorithm

The *primal-dual* algorithm is similar to the successive shortest path algorithm in the sense that it maintains a flow satisfying the reduced cost optimality condition for each iteration, and turns this flow into a feasible flow by

augmenting flows along shortest paths in the residual network. Unlike the successive shortest path algorithm the flow augmentations are performed along all shortest paths solving a maximum flow problem.

The *out-of-kilter* algorithm maintains a solution that satisfies only the mass balance constraints for each iteration, and modifies flows and potentials in a way that moves the solution closer to the optimal flow. Thus, the intermediary solutions might violate both the optimality criteria and the flow bound restrictions. The out-of-kilter algorithm is similar to the successive shortest path and the primal-dual algorithms in that every iteration solves a shortest path problem, augmenting flows on shortest paths.

The successive shortest path, the primal-dual and the out-of-kilter algorithms are all pseudopolynomial (polynomial in n and m and also polynomially dependent on the arc cost/capacity bounds).

7.4 Polynomial minimum cost flow algorithms

A problem with the successive shortest path algorithm is that the flow being augmented between two nodes can be small, thereby requiring a large number of iterations in the worst case. In the same way that the capacity scaling algorithm improves the labeling algorithm for the maximum flow problem by requiring each flow augmentation to be sufficiently large, the *capacity scaling algorithm for the minimum cost flow problem* enhances the successive shortest path algorithm. By requiring the flow augmentations to be sufficiently large the number of iterations is reduced, improving the running time of the algorithm.

For each iteration of the capacity scaling algorithm, the augmentation between an excess and a deficit node is required to be of the magnitude Δ. If no node has an excess of at least Δ, or no node has a deficit of at least $-\Delta$, Δ is reduced by a factor two. Initially Δ is set to $2^{\lfloor \log U \rfloor}$, where U is an upper bound on arc capacity. When the algorithm terminates $\Delta = 1$, and the optimal flow has been found. This geometric improvement technique reduces the complexity of the successive shortest path algorithm from $O(nU \cdot S(n, m, nC))$ to $O(m \log U \cdot S(n, m, nC))$, where $S(n, m, nC)$ is the time required to solve a shortest path problem with n nodes, m arcs, and arc costs bounded by C.

A number of strictly polynomial algorithms (i.e. only dependent on n and m) for the minimum cost flow problem have been discovered, including the *minimum mean cycle-canceling algorithm* and the *repeated capacity scaling algorithm*. The most efficient known algorithm for the minimum cost flow problem is the *enhanced capacity scaling algorithm* which has a complexity of $O((m \log n)(m + n \log n))$.

8 Network optimization and linear programming

Linear programming is a powerful approach to any optimization problem with a linear objective function and linear constraints. In essence, linear programming solves an optimization problem by solving a system of linear equations. The most pervasive and powerful method for solving linear programming problems is the *simplex method*.

Since the minimum cost flow problem can be stated as a linear programming problem, the simplex method can be used to solve shortest path, maximum flow, and minimum cost flow problems. However, because of the special structure of network flow problems, the general simplex method without modifications to exploit the underlying network structure of the problem is not a competitive approach, compared to the algorithms discussed previously. The matrix representing the linear equations derived from the mass balance equations and the network structure is simply too sparse to make a brute force general linear programming solution efficient. Fortunately, however, by modifying the simplex algorithm to take advantage of the networks structure a very efficient algorithm, called the *network simplex algorithm,* can be designed. In practical applications, the network simplex algorithm is one of the most commonly used tools to solve network flow optimization problems.

Many of the theoretic results of network flow theory have their counterparts in linear programming theory. For instance, the max-flow min-cut theorem is a special case of the strong duality theorem, which states that any linear minimization problem can be formulated as an equivalent maximization problem and vice versa.

The minimum cost flow problem can be stated as a linear program as follows:

$$\text{Minimize } cx$$

$$\text{subject to}$$

$$Nx = b,$$

$$0 \le x \le u.$$

Here, N is an m x n matrix called the *node-arc incidence matrix* and the linear equation system $Nx = b$ is the mass balance equations, representing the inflow and outflow of each node in the network.

9 Minimum spanning trees

A spanning tree T of an undirected graph G is a connected acyclic subgraph of G that spans all the nodes of G. If G has n nodes, every spanning tree of G has $n - 1$ arcs. If each arc of G has an associated cost (or length) the minimum spanning tree of G is the spanning tree that has the smallest total cost of its constituent arcs.

Minimum spanning tree problems are very similar to the shortest path problem. The main difference is that for minimum spanning tree problems, the arcs are undirected.

9.1 Kruskal's algorithm

A simple polynomial time algorithm for the minimum spanning tree problem is *Kruskal's algorithm*, which builds a minimum spanning tree by adding arcs to the tree one at a time. First, all arcs are sorted in non-decreasing order of their costs. A list that is initially empty defines the set of arcs that will eventually constitute the spanning tree. For each iteration, the arc with the lowest cost is added to the spanning tree list providing it doesn't create a cycle with the arcs already in the list. If the arc considered does create a cycle it is discarded and not considered for inclusion in the spanning tree list anymore. When $n - 1$ arcs have been added to the spanning tree list, the algorithm terminates.

The correctness of the algorithm relies on the key observation that every arc in a minimum spanning tree is a minimum cost arc across the cut that is defined by removing it from the tree. (If this was not the case, we could substitute an arc violating this property by another arc from the cut with a lower cost, creating a spanning tree with lower total cost.) As the algorithm progresses, the spanning tree list will be a forest of minimum spanning trees of subgraphs of the full graph. Each time an arc is added to the spanning tree list, it will either connect two trees into one, or form a new tree in the forest. Since the arcs are chosen in order of their associated costs, an arc connecting two trees will be the smallest cost arc in the cut between the two trees it connects, and thus as trees are merged, they will be the minimum spanning tree of the graph made up from the nodes they span. Eventually, the minimum spanning tree of the full graph will be found.

The complexity of Kruskal's algorithm is clearly polynomial, since the sorting of the arcs can be carried out in $O(m \log n)$ time and the cycle detection operation of each iteration of the algorithm, even with a simplistic implementation, can be carried out in $O(n)$ time, giving an total complexity of $O(nm)$. (A simple way to detect whether the addition of arc (i, j) will create a cycle is to scan each tree of the forest in the spanning tree list to see if both node i and node j belong to the same tree. This requires $O(n)$ time.) More efficient implementations of Kruskal's algorithm have also been developed.

9.2 Prim's algorithm

Prim's algorithm is another simple, yet efficient algorithm for finding minimum spanning trees. It builds a spanning tree by fanning out from a node, adding arcs one by one. For each step, the nodes of the graph belong either to the subset of nodes S spanned by the nodes that have been reached by the spanning tree so far, or to the subset $\overline{S}$ of nodes that have not yet been reached. The arc from the cut $[S, \overline{S}]$ with the lowest cost is added to the spanning tree for each step (implying that the node that is reached by this arc will be added to S). When $n - 1$ arcs have been added to the spanning tree (or, equivalently, when the set $\overline{S}$ is empty), the algorithm terminates.

Since every arc in a minimum spanning tree is a minimum cost arc across the cut that is defined by removing it from the tree, the correctness of the algorithm follows from the fact that at each iteration we add the lowest cost arc from the cut $[S, \overline{S}]$.

For each of the $n - 1$ iterations of the algorithm we scan at most m arcs (in a naïve implementation) to find the minimum cost arc of the cut $[S, \overline{S}]$. Thus, the complexity of the algorithm is clearly bounded by $O(mn)$. This worst case complexity can be improved by using a cleverly designed data structure for the arcs, so that the arc selection process is quicker. For instance, a Fibonacci heap implementation achieves a complexity of $O(m + n \log n)$.

10 Summary

Network optimization is concerned with how to efficiently transport some entity from one point to another in a network, given a number of limiting constraints, such as the capacities and costs of the communication links of the network. We have seen that the minimum cost flow problem is a generalization of the shortest path problem and the maximum flow problem, that considers both link capacities and link costs. Since the objective function and the capacity and cost constraints are usually (but not always) represented by linear relationships, network optimization can be seen as a special case of general discrete linear optimization, and can be solved by linear programming. Due to the strong duality theorem of linear programming, any minimization problem can be converted into an equivalent maximization problem. In terms of network optimization this result leads to the

max-flow min cut theorem, which states that finding the maximum flow between two nodes in a network is conceptually equivalent to finding the cut, among all cuts in the network separating the source and sink nodes, that has the least capacity.

Efficient (polynomial time) algorithms have been developed for the shortest path problem and the maximum flow problem as well as for the more general minimum cost flow problem. Since the shortest path and maximum flow problems are simplified specializations of the minimum cost flow problems, shortest path and maximum flow algorithms are typically simpler and more efficient than minimum cost flow problems.

We have explored the minimum spanning tree problem, and presented two efficient and simple algorithms with polynomial complexity.

Applications of network optimization theory are vast, and with the ever increasing proliferation of computer networks and other communication systems, it will continually be of tremendous importance.

References

R. K. Ahuja, T. L. Magnati, J. B Orlin, "Network flows: Theory, algorithms, and applications," Prentice Hall, 1993.