Lecture Notes for IEOR 266:
Graph Algorithms and Network Flows

Professor Dorit S. Hochbaum

Contents

These notes are based on "scribe" notes taken by students attending Professor Hochbaum's course IEOR 266. The current version has been updated and edited by Professor Hochbaum in 2024.

1 Introduction

We will begin the study of network flow problems with a review of the formulation of linear programming (LP) problems. Let the number of decision variables, x_j's, be N, and the number of constraints be M. LP problems take the following generic form:

$$
\begin{aligned}
\min \quad & \sum_{j=1}^{N} c_j x_j \\
\text{s.t.} \quad & \sum_{j=1}^{N} a_{i,j} x_j \leq b_i \quad \forall i \in \{1, \ldots, M\} \\
& x_j \geq 0 \qquad \forall j \in \{1, \ldots, N\}
\end{aligned}
\tag{1}
$$

Integer linear programming (ILP) has the following additional constraint:

$$
x_j \text{ integer} \quad \forall j \in \{1, \ldots, N\}
\tag{2}
$$

It may appear that ILP problems are simpler than LP problems, since the solution space in ILP is countably finite while the solution space in LP is infinite; one obvious solution to ILP is *enumeration*, i.e. systematically try out all feasible solutions and find one with the minimum cost. As we will see in the following ILP example, enumeration is not an acceptable algorithm as, even for moderate-size problems, its running time would be extremely large.

1.1 Assignment problem

In words, the assignment problem is the following: given n tasks to be completed individually by n people, what is the minimum cost of assigning all n tasks to n people, one task per person, where $c_{i,j}$ is the cost of assigning task j to person i? Let the decision variables be defined as:

$$
x_{i,j} = \begin{cases} 1 & \text{if person } i \text{ takes on task } j \\ 0 & \text{otherwise} \end{cases}
\tag{3}
$$

The problem can now be formulated as:

$$
\begin{aligned}
\min \quad & \sum_{j=1}^{n} \sum_{i=1}^{n} c_{i,j} x_{i,j} \\
\text{s.t.} \quad & \sum_{j=1}^{n} x_{i,j} = 1 \qquad \forall i = 1, \ldots, n \\
& \sum_{i=1}^{n} x_{i,j} = 1 \qquad \forall j = 1, \ldots, n \\
& x_{i,j} \geq 0 \qquad x_{i,j} \text{ integer}
\end{aligned}
\tag{4}
$$

The first set of constraints ensures that exactly one person is assigned to each task; the second set of constraints ensures that each person is assigned exactly one task. Notice that the upper bound constraints on the x_{ij} are unnecessary. Also notice that the set of constraints is not independent (the sum of the first set of constraints equals the sum of the second set of constraints), meaning that one of the $2n$ constraint can be eliminated.

While at first it may seem that the integrality condition limits the number of possible solutions and could thereby make the integer problem easier than the continuous problem, the opposite is actually true. Linear programming optimization problems have the property that there exists an optimal solution at a so-called *extreme point* (a basic solution); the optimal solution in an integer program, however, is not guaranteed to satisfy any such property and the number of possible integer valued solutions to consider becomes prohibitively large, in general.

Consider a simple algorithm for solving the Assignment Problem: It enumerates all possible assignments and takes the cheapest one. Notice that there are $n!$ possible assignments. If we

consider an instance of the problem in which there are 70 people and 70 tasks, that means that there are

$$
\begin{aligned}
70! \;&=\; 11978571669969891796072783721689098 \\
&\quad 7364589381425464258575553628646628 \\
&\quad 0095827898453196800000000000000000 \\
&\approx\; 2^{332}
\end{aligned}
$$

different assignments to consider. Our simple algorithm, while correct, is not at all practical! The existence of an algorithm does not mean that there exits a useful algorithm.

While linear programming belongs to the class of problems P for which "good" algorithms exist (an algorithm is said to be good if its running time is bounded by a polynomial in the size of the input), integer programming belongs to the class of NP-hard problems for which it is considered highly unlikely that a "good" algorithm exists. Fortunately, as we will see later in this course, the Assignment Problem belongs to a special class of integer programming problems known as the *Minimum Cost Network Flow Problem*, for which efficient polynomial algorithms *do* exist.

The reason for the tractability of the assignment problem is found in the form of the constraint matrix. The constraint matrix is *totally unimodular* (TUM). Observe the form of the constraint matrix A:

$$
A = \begin{bmatrix}
0 & 1 & \cdots \\
1 & 0 & \cdots \\
0 & \vdots & \cdots \\
\vdots & \vdots & \\
1 & \vdots & \\
0 & 1 &
\end{bmatrix}
\tag{5}
$$

We first notice that each column contains exactly two 1's. Notice also that the rows of matrix A can be partitioned into two sets, say A_1 and A_2, such the two 1's of each column are in different sets. It turns out these two conditions are sufficient for the matrix A to be TUM. Note that simply having 0's and 1's are not sufficient for the matrix to be TUM. Consider the constraint matrix for the *vertex cover* problem:

$$
A = \begin{bmatrix}
0 & \cdots & 1 & 0 & \cdots & 1 \\
0 & 1 & 0 & \cdots & 1 & 0 & \cdots \\
\vdots & & & & &
\end{bmatrix}
\tag{6}
$$

Although this matrix also contains 0's and 1's only, it is not TUM. In fact, vertex cover is an NP-complete problem. We will revisit the vertex cover problem in the future. We now turn our attention to the formulation of the generic minimum cost network flow problem.

1.2   Basic graph definitions

- A graph or undirected graph G is an ordered pair $G := (V, E)$. Where V is a set whose elements are called vertices or nodes, and E is a set of unordered pairs of vertices of the form $[i, j]$, called edges.

- A directed graph or digraph G is an ordered pair $G := (V, A)$. Where V is a set whose elements are called vertices or nodes, and A is a set of ordered pairs of vertices of the form (i, j), called arcs. In an arc (i, j) node i is called the *tail* of the arc and node j the *head* of the arc. We sometimes abuse of the notation and refer to a digraph also as a graph.

- A path (directed path) is an ordered list of vertices $(v_1, \ldots, v_k)$, so that $(v_i, v_{i+1}) \in E$ $((v_i, v_{i+1}) \in A)$ for all $i = 1 \ldots, k$. The length of a path is $|(v_1, \ldots, v_k)| = k$.

- A cycle (directed cycle) is an ordered list of vertices $v_0, \ldots, v_k$, so that $(v_i, v_{i+1}) \in E$ $((v_i, v_{i+1}) \in A)$ for all $i = 1, 2, \ldots, n$ and $v_0 = v_k$. The length of a cycle is $|(v_0, \ldots, v_k)| = k$.

- A simple path (simple cycle) is a path (cycle) where all vertices $v_1, \ldots, v_k$ are distinct.

- An (undirected) graph is said to be *connected* if, for every pair of nodes, there is an (undirected) path starting at one node and ending at the other node.

- A directed graph is said to be *strongly connected* if, for every (ordered) pair of nodes (i, j), there is a directed path in the graph starting in i and ending in j.

- The *degree* of a vertex is the number of edges incident to the vertex. In the homework assignement you will prove that $\sum_{v \in V} degree(v) = 2|E|$.

- In a directed graph the *indegree* of a node is the number of incoming arcs taht have that node as a head. The outdegree of a node is the number of outgoing arcs from a node, that have that node as a tail.

  Be sure you can prove,

  $$\sum_{v \in V} indeg(v) = |A|,$$

  $$\sum_{v \in V} outdeg(v) = |A|.$$

- A *tree* can be characterized as a connected graph with no cycles. The relevant property for this problem is that a tree with n nodes has $n - 1$ edges.

  Definition: An undirected graph $G = (V, T)$ is a tree if the following three properties are satisfied:

  Property 1: $|T| = |V| - 1$.
  Property 2: G is connected.
  Property 3: G is acyclic.

  (Actually, it is possible to show, that any two of the properties imply the third).

- A graph is *bipartite* if the vertices in the graph can be partitioned into two sets in such a way that no edge joins two vertices in the same set.

- A *matching* in a graph is set of graph edges such that no two edges in the set are incident to the same vertex.
  The *bipartite (nonbipartite) matching problem*, is stated as follows: Given a bipartite (nonbipartite) graph $G = (V, E)$, find a maximum cardinality matching.

2   Totally unimodular matrices

An $m \times n$ matrix is said to be *unimodular* if for every submatrix of size $m \times m$ the value of its determinant is $1, 0$, or -1. A matrix, A, is said to be *totally unimodular* if the determinant of every square submatrix is $1, 0$ or -1.

Consider the general linear programming problem with integer coefficients in the constraints, right-hand sides and objective function:

(LP)

$$\begin{aligned} \max \quad & cx \\ \text{subject to} \quad & Ax \leq b \end{aligned}$$

Recall that a basic solution of this problem represents an extreme point on the polytope defined by the constraints (i.e. cannot be expressed as a convex combination of any two other feasible solutions). Also, if there is an optimal solution for LP then there is an optimal, basic solution. If in addition A is totally unimodular we have the following result.

Lemma 2.1. *If the constraint matrix A in LP is totally unimodular and the vector b is integer, then there exists an integer optimal solution. In fact every basic or extreme point solution is integral.*

Proof. If A is $m \times n$, $m < n$, a basic solution is defined by a nonsingular, $m \times m$ square submatrix of A, $B = (b_{ij})$, called the basis, and a corresponding subset of the decision variables, x_B, called the basic variables, such that

$$x_B = B^{-1}b.$$

By Cramer's rule, we know

$$B^{-1} = \frac{C^T}{det(B)},$$

where $C = (c_{ij})$ is an $m \times m$ matrix, called the cofactor matrix of B. Each c_{ij} is the determinant of a submatrix formed by by deleting row i and column j from B; this determinant is then multiplied by an appropriate sign coefficient (1 or -1) depending on whether $i + j$ is even or odd. That is,

$$C_{ij} = (-1)^{i+j} \cdot det \begin{pmatrix} b_{1,1} & \dots & b_{1,j-1} & X & b_{1,j+1} & \dots & b_{1,m} \\ \vdots & \vdots & \vdots & \vdots & \vdots & \vdots & \vdots \\ b_{i-1,1} & \dots & b_{i-1,j-1} & X & b_{i-1,j+1} & \dots & b_{i-1,m} \\ X & X & X & X & X & X & X \\ b_{i+1,1} & \dots & b_{i+1,j-1} & X & b_{i+1,j+1} & \dots & b_{i+1,m} \\ \vdots & \vdots & \vdots & \vdots & \vdots & \vdots & \vdots \\ b_{m,1} & \dots & b_{m,j-1} & X & b_{m,j+1} & \dots & b_{m,m} \end{pmatrix}.$$

Thus, since the determinant is a sum of terms each of which is a product of entries of the submatrix, then all entries in C are integer. In addition, A is totally unimodular and B is a basis (so $|det(B)| = 1$). Therefore, all of the elements of B^{-1} are integers. So we conclude that any basic solution $x_B = B^{-1}b$ is integral. $\square$

Although we will require all coefficients to be integers, the lemma also holds if only the right hand sides vector b is integer.

This result has ramifications for combinatorial optimization problems. Integer programming problems with totally unimodular constraint matrices can be solved simply by solving the linear

programming relaxation. This can be done in polynomial time via the Ellipsoid method, or more practically, albeit not in polynomial time, by the simplex method. The optimal basic solution is guaranteed to be integral.

The set of totally unimodular integer problems includes network flow problems. The rows (or columns) of a network flow constraint matrix contain exactly one 1 and exactly one -1, with all the remaining entries 0. A square matrix A is said to have the *network property* if every column of A has at most one entry of value 1, at most one entry of value -1, and all other entries 0.

Lemma 2.2. *If the constraint matrix A has at most one 1 and at most one -1 in each column (row) than A is totally unimodular.*

Proof. The proof is by induction on k, the size of the submatrix of A. For $k = 1$ the determinant value is the value of the entry, which is in $\{-1, 0, 1\}$ as required.

Consider a $k \times k$ submatrix of A, B_k. If B_k has a column of all zeros, then $det(B_k) = 0$. If every column of B_k has exactly one 1 and one -1, them $det(B_k) = 0$, since the sum of all rows is zero. Therefore there exists a column of B_k with exactly one 1, or exactly one -1. The determinant is then $1 \times det(B_{k-1})$ in the first case, and $-1 \times det(B_{k-1})$ in the second case, which is -1, 0 or 1 by the inductive assumption. $\square$

3 Minimum cost network flow problem

We now formulate the Minimum Cost Network Flow Problem. In general,

Problem Instance: Given a network $G = (V, A)$, with a cost c_{ij}, upper bound u_{ij}, and lower bound l_{ij} associated with each directed arc $(i, j) \in A$, and a supply b_i at each node $i \in V$. Find the cheapest *flow* that satisfies: 1) the capacity constraints on the arcs, 2) the supplies/demands at the nodes, 3) that the flow is conserved through the network.

Linear Programming Formulation: The problem can be formulated as an LP with variables x_{ij} indicating the amount of flow on arc (i, j):

$$
\begin{aligned}
\min \quad & \sum_{j=1}^{n} \sum_{i=1}^{n} c_{ij} x_{ij} \\
\text{subject to} \quad & \sum_{k \in N} x_{ik} - \sum_{k \in N} x_{ki} = b_i \quad \forall i \in V \\
& l_{ij} \leq x_{ij} \leq u_{ij}, \qquad\qquad x_{ij} \ \forall (i, j) \in A
\end{aligned}
$$

The first set of constraints is called the *flow balance constraints*. The flow balance constraint for node i indicates that the outflow from node i minus the inflow to node i is equal to the supply of node i. Nodes with negative supply are sometimes referred to as *demand* nodes and nodes with 0 supply are sometimes referred to as *transshipment* nodes. The second set is formed by the *capacity* constraints. Notice that the constraints are dependent (left hand sides add to 0, so that the right hand sides must add to for consistency) and, as before, one of the constraints can be removed.

The distinctive feature of this matrix of the flow balance constraint matrix is that each column has precisely one 1 and one -1 in it. Can you figure out why?

$$A = \begin{bmatrix} 0 & 1 & \cdots \\ 0 & -1 & \cdots \\ 1 & 0 & \\ 0 & \vdots & \\ -1 & \vdots & \\ 0 & & \\ \vdots & & \end{bmatrix} \tag{7}$$

A matrix that has at most one 1 and at most one -1 per column (or row) is *totally unimodular*. Therefore MCNF problems with all supplies and upper/lower bounds that are integer valued can be solved using LP techniques as all extreme points of the linear programming relaxation of the Minimum Cost Network Flow Problem are integer valued. However, we will investigate more efficient techniques to solve MCNF problems, which found applications in many areas. In fact, the assignment problem is a special case of MCNF problem, where each person is represented by a supply node of supply 1, each job is represented by a demand node of demand 1, and there are an arc from each person j to each job i, labeled by $c_{i,j}$, that denotes the cost of person j performing job i.

Example of general MCNF problem - the chairs problem

We now look at an example of general network flow problem, as described in handout #2. We begin the formulation of the problem by defining the units of various quantities in the problem: we let a cost unit be 1 cent, and a flow unit be 1 chair. To construct the associated graph for the problem, we first consider the wood sources and the plants, denoted by nodes $WS1, WS2$ and $P1, \ldots, P4$ respectively, in Figure 1a. Because each wood source can supply wood to each plant, there is an arc from each source to each plant. To reflect the transportation cost of wood from wood source to plant, the arc will have cost, along with capacity lower bound (zero) and upper bound (infinity). Arc $(WS2, P4)$ in Figure 1a is labeled in this manner.

(a) wood source & plants   (b) plants & cities
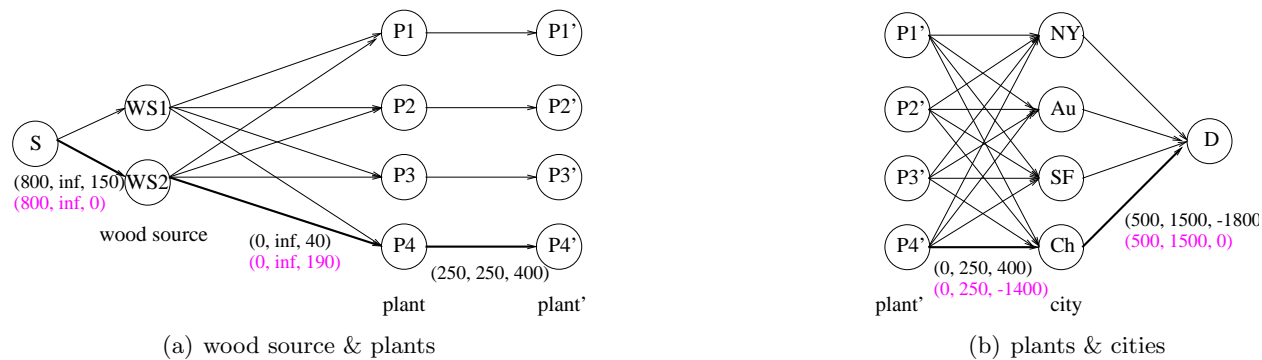
Figure 1: Chair Example of MCNF

It is not clear exactly how much wood each wood source should supply, although we know that contract specifies that each source must supply at least 800 chairs worth of wood (or 8 tons). We represent this specification by *node splitting*: we create an extra node S, with an arc connecting to each wood source with capacity lower bound of 800. Cost of the arc reflects the cost of wood at each wood source. Arc $(S, WS2)$ is labeled in this manner. Alternatively, the cost of wood can

be lumped together with the cost of transportation to each plant. This is shown as the grey color labels of the arcs in the figure.

Each plant has a production maximum, production minimum and production cost per unit. Again, using the node splitting technique, we create an additional node for each plant node, and labeled the arc that connect the original plant node and the new plant node accordingly. Arc $(P4, P4')$ is labeled in this manner in the figure.

Each plant can distribute chairs to each city, and so there is an arc connecting each plant to each city, as shown in Figure 1b. The capacity upper bound is the production upper bound of the plant; the cost per unit is the transportation cost from the plant to the city. Similar to the wood sources, we don't know exactly how many chairs each city will demand, although we know there is a demand upper and lower bound. We again represent this specification using the node splitting technique; we create an extra node D and specify the demand upper bound, lower bound and selling price of each unit of each city by labeling the arc that connects the city and node D.

(a) circulation arc

(b) drainage arc
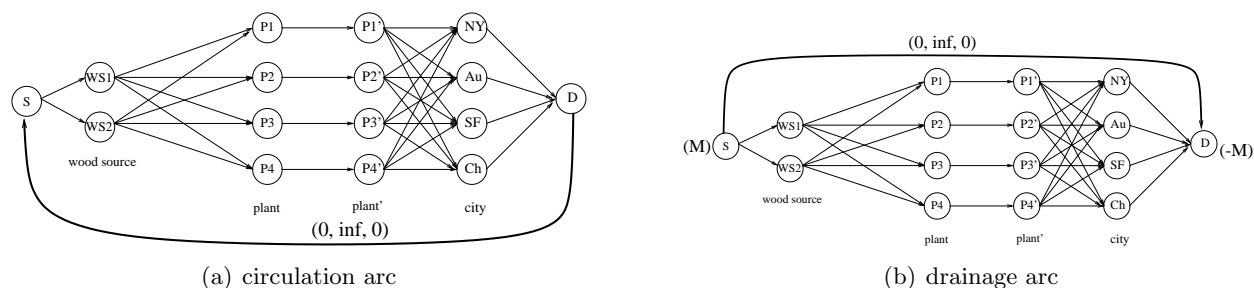
Figure 2: Complete Graph for Chair Example

Finally, we need to relate the supply of node S to the demand of node D. There are two ways to accomplish this. We observe that it is a closed system; therefore the number of units supplied at node S is exactly the number of units demanded at node D. So we construct an arc from node D to node S, with label $(0, \infty, 0)$, as shown in Figure 2a. The total number of chairs produced will be the number of flow units in this arc. Notice that all the nodes are now transshipment nodes — such formulations of problems are called *circulation problems*.

Alternatively, we can supply node S with a large supply of M units, much larger than what the system requires. We then create a *drainage arc* from S to D with label $(0, \infty, 0)$, as shown in Figure 2b, so the excess amount will flow directly from S to D. If the chair production operation is a money losing business, then all M units will flow directly from S to D.

3.1 Transportation problem

A problem similar to the minimum cost network flow, which is an important special case is called the *transportation problem*: the nodes are partitioned into suppliers and consumers only. There are no transhippment nodes. All edges go directly from suppliers to consumers and have infinite capacity and known per unit cost. The problem is to satisfy all demands at minimum cost. The assignment problem is a special case of the transportation problem, where all supply values and all demand values are 1.

The graph that appears in the transportation problem is a so-called **bipartite** graph, or a graph in which all vertices can be divided into two classes, so that no edge connects two vertices in the same class:
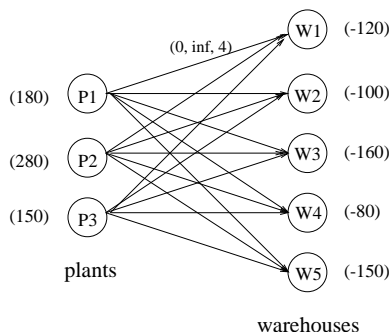
Figure 3: Example of a Transportation Problem

$$(V_1 \cup V_2; A) \qquad \forall (u, v) \in A$$
$$u \in V_1, \quad v \in V_2 \tag{8}$$

A property of bipartite graphs is that they are 2-colorable, that is, it is possible to assign each vertex a "color" out of a 2-element set so that no two vertices of the same color are connected.

An example transportation problem is shown in Figure 3. Let s_i be the supplies of nodes in V_1, and d_j be the demands of nodes in V_2. The transportation problem can be formulated as an IP problem as follows:

$$\begin{aligned}
\min \quad & \sum_{i \in V_1, j \in V_2} c_{i,j} x_{i,j} \\
\text{s.t.} \quad & \sum_{j \in V_2} x_{i,j} = s_i && \forall i \in V_1 \\
& -\sum_{i \in V_1} x_{i,j} = -d_j && \forall j \in V_2 \\
& x_{i,j} \geq 0 && x_{i,j} \text{ integer}
\end{aligned} \tag{9}$$

Note that assignment problem is a special case of the transportation problem. This follows from the observation that we can formulate the assignment problem as a transportation problem with $s_i = d_j = 1$ and $|V_1| = |V_2|$.

3.2 The maximum flow problem

Recall the IP formulation of the *Minimum Cost Network Flow* problem.

$$\begin{aligned}
& \text{Min} && cx \\
(\text{ MCNF}) \quad & \text{subject to} && Tx = b && \text{Flow balance constraints} \\
& && l \leq x \leq u, && \text{Capacity bounds}
\end{aligned}$$

Here $x_{i,j}$ represents the flow along the arc from i to j. We now look at some specialized variants.

The *Maximum Flow* problem is defined on a directed network with capacities u_{ij} on the arcs, and no costs. In addition two nodes are specified, a source node, s, and sink node, t. The objective is to find the maximum flow possible between the source and sink while satisfying the arc capacities. If $x_{i,j}$ represents the flow on arc (i, j), and A the set of arcs in the graph, the problem can be formulated as:

$$
\text{(MaxFlow)} \quad
\begin{array}{ll}
\text{Max} & V \\
\text{subject to} & \sum_{(s,i) \in A} x_{s,i} = V \qquad \text{Flow out of source} \\
& \sum_{(j,t) \in A} x_{j,t} = V \qquad \text{Flow in to sink} \\
& \sum_{(i,k) \in A} x_{i,k} - \sum_{(k,j) \in A} x_{k,j} = 0, \qquad \forall k \neq s, t \\
& l_{i,j} \leq x_{i,j} \leq u_{i,j}, \quad \forall (i,j) \in A.
\end{array}
$$

The *Maximum Flow* problem and its dual, known as the *Minimum Cut* problem have a number of applications and will be studied at length later in the course.

3.3 The shortest path problem

The *Shortest Path* problem is defined on a directed, weighted graph, where the weights may be thought of as distances. The objective is to find a path from a source node, s, to node a sink node, t, that minimizes the sum of weights along the path. To formulate as a network flow problem, let $x_{i,j}$ be 1 if the arc (i,j) is in the path and 0 otherwise. Place a supply of one unit at s and a demand of one unit at t. The formulation for the problem is:

$$
\text{(SP)} \quad
\begin{array}{ll}
\text{Min} & \sum_{((i,j) \in A} d_{i,j} x_{i,j} \\
\text{subject to} & \sum_{(i,k) \in A} x_{i,k} - \sum_{(j,i) \in A} x_{j,i} = 0 \qquad \forall i \neq s, t \\
& \sum_{(s,i) \in A} x_{s,i} = 1 \\
& \sum_{(j,t) \in A} x_{j,t} = 1 \\
& 0 \leq x_{i,j} \leq 1
\end{array}
$$

This problem is often solved using a variety of search and labeling algorithms depending on the structure of the graph.

3.4 The single source shortest paths

A related problem is the *single source shortest paths* problem (SSSP). The objective now is to find the shortest path from node s to all other nodes $t \in V \setminus s$. The formulation is very similar, but now a flow is sent from s, which has supply $n - 1$, to every other node, each with demand one.

$$
\text{(SSSP)} \quad
\begin{array}{ll}
\text{Min} & \sum_{(i,j) \in A} d_{i,j} x_{i,j} \\
\text{subject to} & \sum_{(i,k) \in A} x_{i,k} - \sum_{(j,i) \in A} x_{j,i} = -1 \qquad \forall i \neq s \\
& \sum_{(s,i) \in A} x_{s,i} = n - 1 \\
& 0 \leq x_{i,j} \leq n - 1, \quad \forall (i,j) \in A.
\end{array}
$$

But in this formulation, we are minimizing the sum of all the costs to each node. How can we be sure that this is a correct formulation for the single source shortest paths problem? In the next theorem we show that there exists an optimal solution $\mathbf{x}^*$ where the positive flow arcs form a *branching* rooted in s. A branching is a directed spanning tree with all arcs directed down from the root, and the indegree of each node other than the root is 1. A property of a branching is that there is a unique path in the branching from s to every node. This path necessarily carries to every node i a flow amount equal to the number of nodes in the subtree rooted in i. Now if the unique path in the branching from s to a node i is not shortest, then there is a shorter path to which we can transfer one unit of flow from s to i from the path in the branching to the shorter path and therefore generate a feasible flow for (SSSP) that is strictly lower in value, thus contradicting optimality.

Theorem 3.1 (Branching theorem). *Given a directed graph $G = (V, A)$, there exists an optimal solution x_{ij}^* to a single-source shortest paths problem on G with source node $s \in V$ such that*

$$indeg_{A^+}(i) = 1 \qquad \forall\ i \in V \setminus \{s\}$$

where $A^+ = \{(i, j) \in A \mid x_{ij}^* > 0\}$.

Proof. Let x_{ij}^* be an optimal solution to a single-source shortest path problem on G with source node s. Furthermore let x_{ij}^* be an optimal solution that minimizes the sum of indegrees with respect to the arcs in the solution set A^+

$$\sum_{i \in V \setminus \{s\}} indeg_{A^+}(i).$$

We will show that this optimal solution that minimizes the sum of indegrees cannot contain a node of indegree strictly greater than 1 by contradiction. This is equivalent to claiming that the sum of indegrees is $n - 1$, which is the minimum possible in a connected graph. Let the *level* of node $v \in V$ be equal to the length (in terms of number of arcs) of the shortest path in A^+ from s to v.

Let i_0 be a node with indegree greater than 1 such no nodes in any path from s to i_0 have indegree greater than 1. That is, i_0 is a lowest index level node of indegree > 1. Intuitively, i_0 is the "first" node we encounter with indegree greater than 1. Therefore there are at least two paths from s to i_0. At least one of the incoming flows to i_0 is from its "parent", a node in one level above that of i_0. Denote this parent node of i_0 by m and it has indegree of 1. Since each node interior to the path from s to i_0 via m is of indegree 1, and all nodes other than s have demand of 1 unit, the incoming flows to these nodes is strictly greater than the flow out of each of these nodes. That is, the flows on this path from s to i_0 are monotone decreasing.

Let the other incoming flow to i come from node n, and find a path from s to node n in A^+. The total cost of both paths, via m and via n, to i_0 must be the same, as otherwise we can move the bottleneck amount (the smallest flow along an arc on the path) from the more expensive path to the cheaper and get a strictly better solution, which contradicts optimality. We can now subtract the amount $x_{mi_0}^*$ on arcs of the path from s to i via m and still each arc will have non-negative flow, and arc (m, i_0) will have 0 flow. Now add the amount $x_{mi_0}^*$ to the flow on the path from s to node n and then to i_0. This transformation preserves the flow balance at all nodes on the paths and at node 0 and therefore this is a feasible solution. It does not create any new positive flow arc and therefore it does not change the indegree of any node except for i_0. Since the cost of both paths must be the same, this is an optimal solution with a strictly smaller indegree of node i_0, and therefore smaller sum of indegrees. This is illustrated in Figure 4 where p is the first node p common to both paths, possibly s. That is, p is the highest indexed level common ancestor.

Here is an alternative proof: Consider two paths from s to i_0, via m and via n. For either of these paths define the *bottleneck arc* as the arc with the smallest amount of flow, and the *bottleneck flow* as the amount of flow on the bottleneck. Let x_{ij}^* be an arc that carries the smaller bottleneck amount on either one of the two paths. We now subtract the amount x_{ij}^* from the path that contains the bottleneck arc (i, j) and add it to the other. Arc (i, j) then carries a zero flow, flow balance and non-negativity are preserved, and no new positive flow arc is created. Thus we found another optimal solution with strictly smaller sum of indegrees (or equivalently, smaller size of $|A^+|$) which is a contradiction,

$\square$
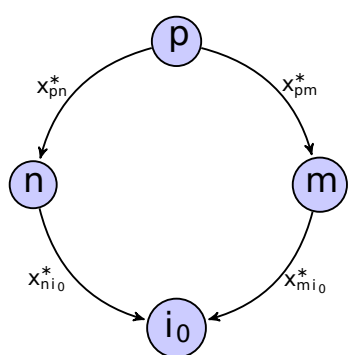
(a) The bottleneck flows are $x^*_{mi_0}$ and $x^*_{ni_0}$.
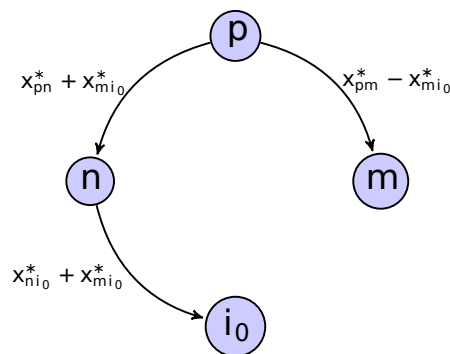
(b) The value of the objective function is unchanged by moving $x^*_{mi_0}$ to the other path, but $indeg_{A^+}(i_0)$ has reduced by 1.

Figure 4: A simplified pair of equal-cost paths from p to i_0.

The same theorem may also be proved following Linear Programming arguments by showing that a basic solution of the algorithm generated by a linear program will always be acyclic in the undirected sense.

3.5 The bipartite matching problem

A matching is defined as M, a subset of the edges E, such that each node is incident to at most one edge in M. Given a graph $G = (V, E)$, find the largest cardinality matching. Maximum cardinality matching is also referred to as the Edge Packing Problem. (We know that $M \leq \lfloor \frac{V}{2} \rfloor$.)

Maximum Weight Matching: We assign weights to the edges and look for maximum total weight instead of maximum cardinality. (Depending on the weights, a maximum weight matching may not be of maximum cardinality.)

Smallest Cost Perfect Matching: Given a graph, find a perfect matching such that the sum of weights on the edges in the matching is minimized. On bipartite graphs, finding the smallest cost perfect matching is the assignment problem.

Bipartite Matching, or *Bipartite Edge-packing* involves pairwise association on a bipartite graph. In a bipartite graph, $G = (V, E)$, the set of vertices V can be partitioned into V_1 and V_2 such that all edges in E are between V_1 and V_2. That is, no two vertices in V_1 have an edge between them, and likewise for V_2. A matching involves choosing a collection of arcs so that no vertex is adjacent to more than one edge. A vertex is said to be "matched" if it is adjacent to an edge. The formulation for the problem is

$$
\begin{array}{lll}
& \text{Max} & \sum_{(i,j) \in E} x_{i,j} \\
\text{(BM)} & \text{subject to} & \sum_{(i,j) \in E} x_{i,j} \leq 1 \qquad \forall i \in V \\
& & x_{i,j} \in \{0, 1\}, \quad \forall\, (i, j) \in E.
\end{array}
$$

Clearly the maximum possible objective value is $\min\{|V_1|, |V_2|\}$. If $|V_1| = |V_2|$ it may be possible to have a *perfect matching*, that is, it may be possible to match every node.

The *Bipartite Matching* problem can also be converted into a maximum flow problem. Add a source node, s, and a sink node t. Make all edge directed arcs from, say, V_1 to V_2 and add a directed arc of capacity 1 from s to each node in V_1 and from each node in V_2 to t. Now maximize the flow from s to t. The existence of a flow between two vertices corresponds to a matching. Due to the capacity constraints, no vertex in V_1 will have flow to more than one vertex in V_2, and no vertex in V_2 will have a flow from more than one vertex in V_1.

Alternatively, one more arc from t back to s with cost of -1 can be added and the objective can be changed to maximizing circulation. Figure 5 depicts the circulation formulation of bipartite matching problem.
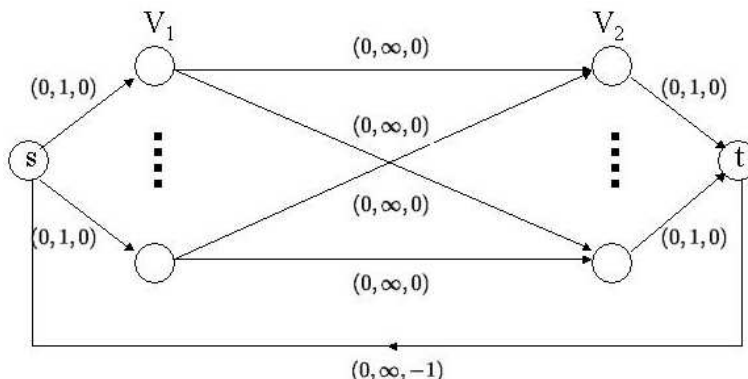
Figure 5: Max-flow with circulation arc formulation of Maximum Cardinality Bipartite Matching

The problem may also be formulated as an assignment problem. Figure 6 depicts this formulation. An assignment problem requires that $|V_1| = |V_2|$. In order to meet this requirement we add dummy nodes to the deficient component of B as well as dummy arcs so that every node dummy or original may have its demand of 1 met. We assign costs of 0 to all original arcs and a cost of 1 to all dummy arcs. We then seek the assignment of least cost which will minimize the number of dummy arcs and hence maximize the number of original arcs.
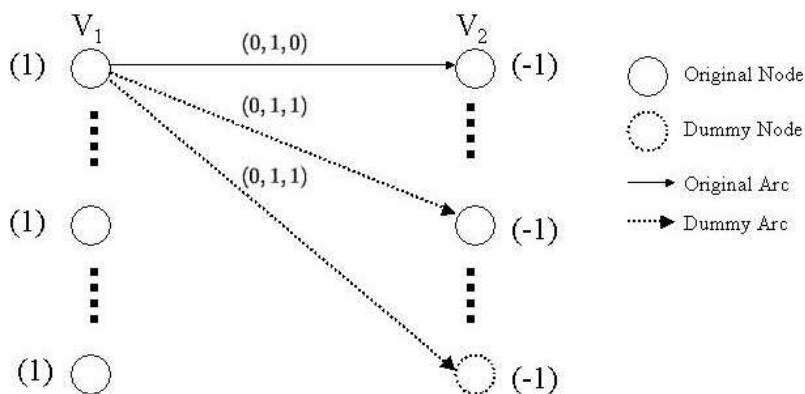
Figure 6: Assignment formulation of Maximum Cardinality Bipartite Matching

The *General Matching*, or *Nonbipartite Matching* problem involves maximizing the size of a collection of edges so that no two edges share an endpoint. This problem cannot be converted to a flow problem, but is still solvable in polynomial time.

Another related problem is the weighted matching problem in which each edge $(i, j) \in E$ has

weight $w_{i,j}$ assigned to it and the objective is to find a matching of maximum total weight. The mathematical programming formulation of this maximum weighted problem is as follows:

$$
\begin{array}{lll}
& \text{Max} & \sum_{(i,j)\in E} w_{i,j} x_{i,j} \\
\text{(WM)} & \text{subject to} & \sum_{(i,j)\in E} x_{i,j} \leq 1 \qquad \forall i \in V \\
& & x_{i,j} \in \{0,1\}, \quad \forall\, (i,j) \in E.
\end{array}
$$

3.6 Summary of classes of network flow problems
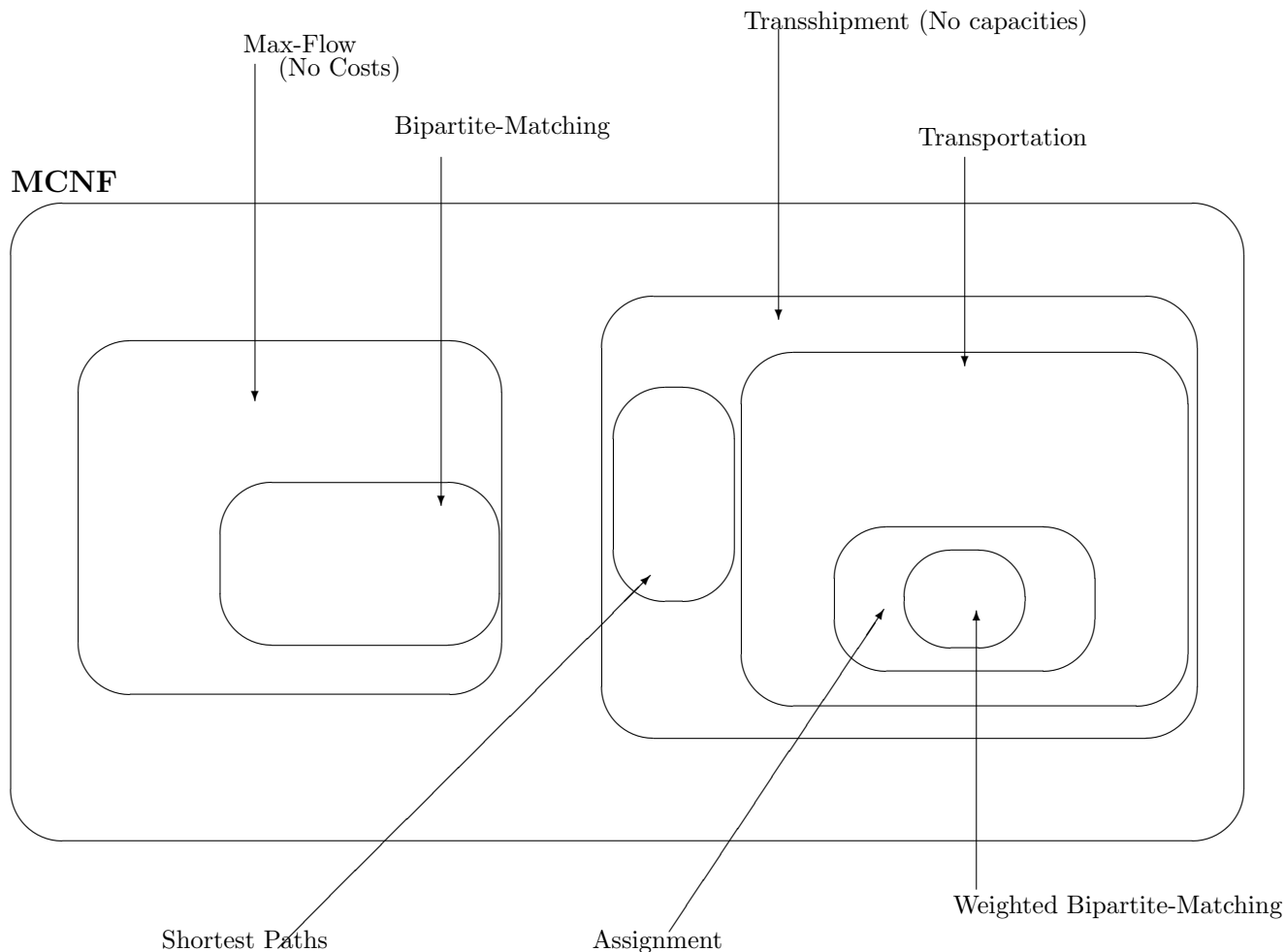
Figure 7: Classification of MCNF problems

There are a number of reasons for classifying special classes of MCNF as we have done in Figure 7. The more specialized the problem, the more efficient are the algorithms for solving it. So, it is important to identify the problem category when selecting a method for solving the problem of interest. In a similar vein, by considering problems with special structure, one may be able to *design* efficient special purpose algorithms to take advantage of the special structure of the problem.

Note that the sets actually overlap, as *Bipartite Matching* is a special case of both the *Assignment Problem* and *Max-Flow*.

3.7   Negative cost cycles in graphs

One of the issues that can arise in weighted graphs is the existence of negative cost cycles. In this case, solving the MCNF problem gives a solution which is unbounded, there is an infinite amount of flow on the subset of the arcs comprising the negative cost cycle.

This problem also occurs if we solve the shortest paths problem as MCNF with upper bounds on the flow on the arcs. To see this, say we modify the formulation with an upper bound on the flow of 1. Then, we can still have a MCNF solution that is not the shortest path, as shown in the Figure 8. In this problem the *shortest path* is $(s, 1, t)$ while the solution to *MCNF* is setting the variables $(x_{2,4}, x_{4,3}, x_{3,2}, x_{s,1}, x_{1,t}) = 1$. These solutions are clearly different, meaning that solving the MCNF problem is not equivalent to solving the shortest path problem for a graph with negative cost cycles.
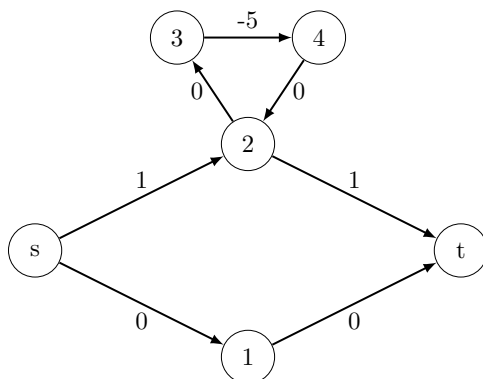
Figure 8: Example of problem with Negative Cost Cycle.
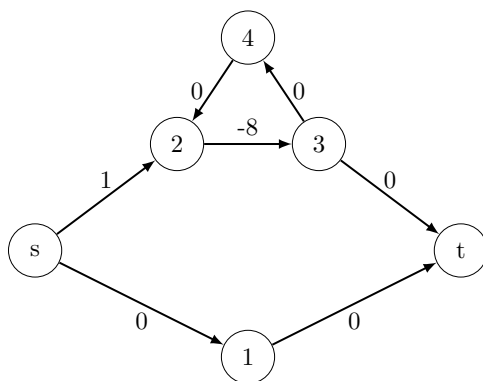
Figure 9: Example of a shortest path problem with negative cost cycle where removing the cycle does not give a shortest path.

Even if we can detect the existence of cycles in the graph on the positive flow arcs (which have flow of 1 unit, as this is the upper bound), and remove them, the remaining path is not necessarily shortest. An example of such a case is provided in Figure 9.

All shortest paths algorithms must therefore have the capability for detecting negative cost cycles in the network. We will see that later on in the course.

4 Other network problems

4.1 Eulerian tour

We first mention the classical Königsberg bridge problem. This problem was known at the time of Euler, who solved it in 1736 by posing it as a graph problem. The graph problem is, given an undirected connected graph, is there a path that traverses all edges exactly once and returns to the starting point. A graph where there exists such path is called an *Eulerian graph.* (You will have an opportunity to show in your homework assignment that a connected graph is Eulerian if and only if all node degrees are even.)

Arbitrarily select edges forming a trail until the trail closes at the starting vertex. This must happen since every vertex that is visited along this trail has an unvisited edge to leave it. If there are still unused edges in the graph, go to a vertex that has a positive, and hence even, degree. It is better to find such a node among those nodes already visited, but not mandatory. Start a new tour from that vertex until coming back to it. This subtour can then be inserted to the prior tour (if that was a vertex already visited) to create an expanded tour that traverses all the edges in the union of these tours. repeat this until there is no vertex of positive degree in the graph of unused edges.

For this algorithm we will use the *adjacency list* representation of a graph, the adjacency lists will be implemented using pointers. That is, for each node i in G, we will have a list of neighbors of i. In these lists each neighbor *points* to the next neighbor in the list and the last neighbor points to a special character to indicated that it is the last neighbor of i:

$$
\begin{aligned}
n(i) &= (i_1 \rightsquigarrow i_2 \rightsquigarrow \cdots \rightsquigarrow i_k \multimap) \\
d_i &= |\{n(i)\}|
\end{aligned}
$$

We will maintain the set of edges in the tour ET. We also have an ordered list $Tour$, and a *tour pointer* p^T that numbers the nodes in the tour in the order they were added. The mapping $T[i]$ identifies the ith node in the tour. Finally, we maintain a pointer fp that maintains the index of the first node in the tour that may have positive degree in the tour.

Pseudocode

> $u \leftarrow 1$ ("initial" node in the tour)
> $Tour \leftarrow (1,)$
> $p^T \leftarrow 2$ (index at which to insert in tour)
> $fp \leftarrow 1$ (index of first positive degree node in tour)
> $ET \leftarrow \emptyset$ (edges in tour)
> While $ET \neq E$ do
>> Select node $v \leftarrow n(u)$ such that $[u,v] \notin ET$
>> $d_u \leftarrow d_u - 1$
>> $d_v \leftarrow d_v - 1$
>> Insert u in $Tour$ at index p^T
>> $p^T \leftarrow p^T + 1$
>> $ET \leftarrow ET \cup \{[u,v]\}$
>> If $d_v > 0$ then
>>> $u \leftarrow v$
>> else - (find first node u in the tour with $d_u > 0$)
>>> while $d_{T[fp]} = 0$
>>>> $fp \leftarrow fp + 1$
>>> $u \leftarrow T[fp]$

$$p^T \leftarrow fp$$

        end if

    end while

done

The algorithm traverses each edge twice. Once to extend the tour and once to check for positive degree nodes. The pointer fp point to node $T[fp]$. This is the first node in the tour that may have a positive degree, since fp is increased only if the degree of fp points to a node with degree zero.

4.1.1 Eulerian walk

An Eulerian walk is a path that traverses all edges exactly once (without the restriction of having to end in the starting node). It can be shown that a graph has an Eulerian walk or Eulerian path if and only if there are exactly 0 or 2 nodes of odd degree.

4.2 Chinese postman problem

The Chinese Postman Problem is closely related to the Eulerian walk problem.

4.2.1 Undirected chinese postman problem

Optimization Version: Given a connected graph $G = (V, E)$ with weighted edges, traverse all edges at least once, so that the total distance traversed is minimized.

This can be solved by first, identifying the odd degree nodes of the graph and constructing a complete graph with these nodes and the shortest path between them as the edges. Then if we can find a minimum weight perfect matching of this new graph, we can decide which edges to traverse twice on the Chinese postman problem and the problem reduces to finding an Eulerian walk.

Decision Version: Given an edge-weighted graph $G = (V, E)$, is there a tour traversing each edge at least once of total weight $\leq M$?

For edge weights $= 1$, the decision problem with $M = |E| = m$ is precisely the Eulerian Tour problem. It will provide an answer to the question: *Is this graph Eulerian?*

4.2.2 Directed chinese postman problem

Given a strongly connected directed graph $G = (V, A)$ with weighted arcs, traverse all arcs at least once, so that the total distance traversed is minimized.

Note that the necessary and sufficient condition of the existence of a directed Eulerian tour is that the indegree must equal the outdegree on every node. If this condition is not satisfied then there is a non-empty subset of excess nodes, in which the indegree exceeds the outdegree and the excess is equal to the difference, and a non-empty subset of deficit nodes, in which the outdegree exceeds the indegree and the deficit is equal to this difference. Construct a transportation problem graph with the set of supply nodes equal to the set of excess nodes where the supplies are equal to the respective excesses, and a set of demand nodes equal to the set of deficit nodes with demands equal to the respective deficits. It is known that the total excesses must be equal to the total deficits (reader's challenge: prove this). Therefore the total supply is equal to the total demand. For every pair of excess node and deficit node find the respective shortest path in the graph between these two nodes and assign the cost of this shortest path as the cost of the respective arc between the two nodes. Now find the minimum cost solution to the transportation problem. For arc (i, j) in the transportation graph with flow f_{ij}, add the arcs on the shortest path from i to j f_{ij} times to

the graph. When done, the indegree of each node is equal to the outdegree, and this was done at total minimum cost.

4.2.3 Mixed chinese postman problem

Chinese Postman Problem with both directed and undirected edges. Surprisingly, this problem is intractable.

4.3 Hamiltonian tour problem

Given a graph $G = (V, E)$, is there a tour visiting each node exactly once?

This is a NP -complete decision problem and no polynomial algorithm is known to determine whether a graph is Hamiltonian.

4.4 Traveling salesman problem (TSP)

Optimization Version: Given a graph $G = (V, E)$ with weights on the edges, visit each node exactly once along a tour, such that we minimize the total traversed weight.

Decision version: Given a graph $G = (V, E)$ with weights on the edges, and a number M, is there a tour traversing each node exactly once of total weight $\leq M$?

4.5 Vertex packing problems (Independent Set)

Given a graph $G = (V, E)$, find a subset S of nodes such that no two nodes in the subset are neighbors, and such that S is of maximum cardinality. (We define neighbor nodes as two nodes with an edge between them.)

We can also assign weights to the vertices and consider the weighted version of the problem.

4.6 Maximum clique problem

A "Clique" is a collection of nodes S such that each node in the set is adjacent to every other node in the set. The problem is to find a clique of maximum cardinality.

Recall that given G, $\bar{G}$ is the graph composed of the same nodes and exactly those edges not in G. Every possible edge is in either G or $\bar{G}$. The maximum independent set problem on a graph G, then, is equivalent to the maximum clique problem in $\bar{G}$.

4.7 Vertex cover problem

Find the smallest collection of vertices S in an undirected graph G such that every edge in the graph is incident to some vertex in S. (Every edge in G has an endpoint in S).

Notation: If V is the set of all vertices in G, and $S \subset V$ is a subset of the vertices, then $V - S$ is the set of vertices in G not in S.

Lemma 4.1. *If S is an independent set, then $V \setminus S$ is a vertex cover.*

Proof. Suppose $V \setminus S$ is not a vertex cover. Then there exists an edge whose two endpoints are not in $V \setminus S$. Then both endpoints must be in S. But, then S cannot be an independent set! (Recall the definition of an independent set – no edge has both endpoints in the set.) So therefore, we proved the lemma by contradiction. □

The proof will work the other way, too:

Result: S is an independent set if and only if $V - S$ is a vertex cover.

This leads to the result that the sum of a vertex cover and its corresponding independent set is a fixed number (namely $|V|$).

Corollary 4.2. *The complement of a maximal independent set is a minimal vertex cover, and vice versa.*

The Vertex Cover problem is exactly the complement of the Independent Set problem.

A 2-Approximation Algorithm for the Vertex Cover Problem: We can use these results to construct an algorithm to find a vertex cover that is at most twice the size of an optimal vertex cover.

(1) Find M, a maximum cardinality matching in the graph.

(2) Let $S = \{u \mid u$ is an endpoint of some edge in $M\}$.

Claim 4.3. *S is a feasible vertex cover.*

Proof. Suppose S is not a vertex cover. Then there is an edge e which is not covered (neither of its endpoints is an endpoint of an edge in M). So if we add the edge to M ($M \cup e$), we will get a feasible matching of cardinality larger than M. But, this contradicts the maximum cardinality of M. □

Claim 4.4. *$|S|$ is at most twice the cardinality of the minimum VC*

Proof. Let OPT = optimal minimal vertex cover. Suppose $|M| > |OPT|$. Then there exists an edge $e \in M$ which doesn't have an endpoint in the vertex cover. Every vertex in the VC can cover at most one edge of the matching. (If it covered two edges in the matching, then this is not a valid matching, because a match does not allow two edges adjacent to the same node.) This is a contradiction. Therefore, $|M| \leq |OPT|$. Because $|S| = 2|M|$, the claim follows. □

4.8 Edge cover problem

This is analogous to the Vertex cover problem. Given $G = (V, E)$, find a subset $EC \subset E$ such that every node in V is an endpoint of some edge in EC, and EC is of minimum cardinality.

As is often the case, this *edge* problem is of polynomial complexity, even though its *vertex* analog is NP -hard.

4.9 b-factor (b-matching) problem

This is a generalization of the Matching Problem: given the graph $G = (V, E)$, find a subset of edges such that each node is incident to at most b edges. Therefore, the Edge Packing Problem corresponds to the 1-factor Problem.

4.10 Graph colorability problem

Given an undirected graph $G = (V, E)$, assign colors $c(i)$ to node $i \in V$, such that $\forall (i, j) \in E; c(i) \neq c(j)$.

Example – Map coloring: Given a geographical map, assign colors to countries such that two adjacent countries don't have the same color. A graph corresponding to the map is constructed as follows: each country corresponds to a node, and there is an edge connecting nodes of two neighboring countries. Such a graph is always planar (see definition below).

Definition – Chromatic Number: The smallest number of colors necessary for a coloring is called the *chromatic number* of the graph.

Definition – Planar Graph: Given a graph $G = (V, E)$, if there exists a drawing of the graph in the plane such that no edges cross except at vertices, then the graph is said to be *planar*.

Theorem 4.5 (4-color Theorem). *It is always possible to color a planar graph using 4 colors.*

Proof. A computer proof exists that uses a polynomial-time algorithm (Appel and Haken (1977)). An "elegant" proof is still unavailable. ☐

Observe that each set of nodes of the same color is an independent set. (It follows that the cardinality of the maximum clique of the graph provides a lower bound to the chromatic number.) A possible graph coloring algorithm consists in finding the maximum independent set and assigning the same color to each node contained. A second independent set can then be found on the subgraph corresponding to the remaining nodes, and so on. The algorithm, however, is both inefficient (finding a maximum independent set is *NP*-hard) and not optimal. It is not optimal for the same reason (and same type of example) that the *b-matching* problem is not solvable optimally by finding a sequence of maximum cardinality matchings.

Note that a 2-colorable graph is bipartite. It is interesting to note that the opposite is also true, i.e., every bipartite graph is 2-colorable. (Indeed, the concept of k-colorability and the property of being k-partite are equivalent.) We can show it building a 2-coloring of a bipartite graph through the following breadth-first-search based algorithm:

Initialization: Set $i = 1$, $L = \{i\}$, $c(i) = 1$.

General step: Repeat until L is empty:

- For each previously uncolored $j \in N(i)$, (a neighbor of i), assign color $c(j) = c(i) + 1 \ (mod \ 2)$ and add j at the end of the list L. If j is previously colored and $c(j) = c(i) + 1 (mod \ 2)$ proceed, else stop and declare *graph is not bipartite.*

- Remove i from L.

- Set $i =$ first node in L.

Termination: Graph is bipartite.

Note that if the algorithm finds any node in its neighborhood already labeled with the opposite color, then the graph is not bipartite, otherwise it is. While the above algorithm solves the 2-color problem in $O(|E|)$, the d-color problem for $d \geq 3$ is *NP*-hard even for planar graphs (see Garey and Johnson, page 87).

4.11 Minimum spanning tree problem

Definition: An undirected graph $G = (V, T)$ is a tree if the following three properties are satisfied:

Property 1: $|T| = |V| - 1$.

Property 2: G is connected.

Property 3: G is acyclic.

(Actually, it is possible to show that any two of the properties imply the third).

Definition: Given an undirected graph $G = (V, E)$ and a subset of the edges $T \subseteq E$ such that (V, T) is tree, then (V, T) is called a *spanning tree* of G.

Minimum spanning tree problem: Given an undirected graph $G = (V, E)$, and costs $c : E \to \mathcal{R}$, find $T \subseteq E$ such that (V, T) is a tree and $\sum_{(i,j) \in T} c(i, j)$ is minimum. Such a graph (V, T) is a *Minimum Spanning Tree* (MST) of G.

The problem of finding an MST for a graph is solvable by a "greedy" algorithm in $O(|E| log |E|)$. A lower bound on the worst case complexity can be easily proved to be $|E|$, because every arc must be looked up at least once. The open question is to see if there exists a linear algorithm to solve the problem (a linear time *randomized* algorithm was devised by Klein and Tarjan, but so far no deterministic algorithm has been provided; still, there are "almost" linear time deterministic algorithms known for solving the MST problem).

5 Complexity analysis

5.1 Measuring quality of an algorithm

Algorithm: One approach is to enumerate the solutions, and select the best one.

Recall that for the assignment problem with 70 people and 70 tasks there are $70! \approx 2^{332.4}$ solutions. The existence of an algorithm does not imply the existence of a <u>good</u> algorithm!

To measure the complexity of a particular algorithm, we count the number of operations that are performed as a function of the 'input size'. The idea is to consider each elementary operation (usually defined as a set of simple arithmetic operations such as $\{+, -, \times, /, <\}$) as having unit cost, and measure the number of operations (in terms of the size of the input) required to solve a problem. The goal is to measure the rate, ignoring constants, at which the running time grows as the size of the input grows; it is an asymptotic analysis.

Traditionally, complexity analysis has been concerned with counting the number of operations that must be performed in the worst case.

Definition 5.1 (Concrete Complexity of a problem). *The complexity of a problem is the complexity of the algorithm that has the lowest complexity among all algorithms that solve the problem.*

5.1.1 Examples

Set Membership - Unsorted list: We can determine if a particular item is in a list of n items by looking at each member of the list one by one. Thus the number of comparisons needed to find a member in an unsorted list of length n is n.

Problem: given a real number x, we want to know if $x \in S$.

Algorithm:
1. {initialize}: $i = 1$;
2. Compare x to s_i;
3. Stop if $x = s_i$;
4. else if $i \leftarrow i + 1 \leq n$ goto 2 else stop x is not in S.

Complexity is n comparisons in the worst case. This is also the **concrete complexity** of this problem. Why?

Set Membership - Sorted list: Given a list $\{a_1 \leq a_2 \leq \ldots \leq a_n\}$ We can determine if a particular item is in a list of n elements via **binary search**. The number of comparisons needed to find a member in a sorted list of length n is proportional to $\log_2 n$.

Problem: given a real number x, we want to know if $x \in S$.

Algorithm:
1. Set $first = a_1$, $last = a_n$;
2. Select $med = \lfloor \frac{first+last}{2} \rfloor$ and compare s_{med} to x;

3. If $s_{med} = x$ stop, output $x \in S$.

4. If $s_{med} < x$ then $first \leftarrow med + 1$ $\{S = (s_{med+1}, \ldots, s_{last})\}$, else $last \leftarrow med - 1$ $\{S = (s_{first}, \ldots, s_{med-1})\}$.

5. If $first < last$ go to 1, else compare x to $first$ and stop:
   If $x = first$ output $x \in S$, otherwise output $x \notin S$.

Complexity: after k^{th} iteration $\frac{n}{2^{k-1}}$ elements remain. We are done searching for k such that $\frac{n}{2^{k-1}} \leq 2$, which implies:

$$\log_2 n \leq k$$

Thus the total number of comparisons is at most $\log_2 n$.

Aside: This binary search algorithm can be used more generally to find the *zero* in a *monotone increasing* and *monotone nondecreasing* functions.

Matrix Multiplication: The straightforward method for multiplying two $n \times n$ matrices takes n^3 multiplications and $n^2(n-1)$ additions. Algorithms with better complexity (though not necessarily practical, see comments later in these notes) are known. Coppersmith and Winograd (1990) came up with an algorithm with complexity $Cn^{2.375477}$ where C is very large, but independent of n. Indeed, the constant term C is so large that in their paper Coppersmith and Winograd admit that their algorithm is impractical in practice.

Forest Harvesting: In this problem we have a forest divided into a number of cells, say a rectangular grid $m \times n$. For each cell we have the following information: H_i - benefit for the timber company to harvest, U_i - benefit for the timber company not to harvest, and B_{ij} - the border effect, which is the benefit received for harvesting exactly one of cells i or j. One way to solve is to look at every possible combination of harvesting and not harvesting and pick the best one. This algorithm requires (2^{mn}) operations. (We will see a much more efficient algorithm later.)

An algorithm is said to be *polynomial* if its running time is bounded by a polynomial in the size of the input. All but the forest harvesting algorithm mentioned above are polynomial algorithms.

An algorithm is said to be *strongly polynomial* if the running time is bounded by a polynomial in the size of the input *and* is independent of the numbers involved; for example, a max-flow algorithm whose running time depends upon the size of the arc capacities is *not* strongly polynomial, even though it may be polynomial (as in the scaling algorithm of Edmonds and Karp). The algorithms for the sorted and unsorted set membership have strongly polynomial running time. So does the greedy algorithm for solving the minimum spanning tree problem. This issue will be returned to later in the course.

Sorting: We want to sort a list of n items in nondecreasing order.

    Input: $S = \{s_1, s_2, \ldots, s_n\}$

    Output: $s_{i1} \leq s_{i2} \leq \cdots \leq s_{in}$

    Bubble Sort: $n' = n$

        While $n' \geq 2$

            $i = 1$

            while $i \leq n' - 1$

                If $s_i > s_{i+1}$ then $t = s_{i+1}$, $s_{i+1} = s_i$, $s_i = t$

                $i = i + 1$

            end while

            $n' \leftarrow n' - 1$

        end while

        Output $\{s_1, s_2, \ldots, s_n\}$

Basically, we iterate through each item in the list and compare it with its neighbor. If the number on the left is greater than the number on the right, we swap the two. We repeat this operation for each number in turn in the array. At the end of the first pass, the last number in the newly ordered list is in the correct location - that is, it is greater or equal than all the preceding elements. This last number can then be removed from the array to be sorted and placed permanently in the last position of the sorted array. At the end of the second pass, the last and the penultimate numbers are in the correct positions. And so forth. This "invariant" proves the correctness of the algorithm. These iterations are repeated for one number at a time and therefore at most n times. The size of the array to be sorted is n in the first iteration and the number of comparisons $n - 1$. There are $n - 2$ in the second iteration, and $n - i$ comparisons in the ith iteration.

The complexity complexity of this algorithm is therefore $\sum_{k=2}^{n}(k-1) = n(n-1)/2 = O(n^2)$.

Merge sort (a recursive procedure):

Sort(L_n): (this procedure returns L_n^{sort} sorted in ascending order)

begin

    If $n < 2$

        return L_n^{sort}

    else

        partition L_n into $L_{\lfloor n/2 \rfloor}, L_{\lceil n/2 \rceil}$ $(L_n = L_{\lfloor n/2 \rfloor} \cup L_{\lceil n/2 \rceil})$

        Call Sort($L_{\lfloor n/2 \rfloor}$) (to get $L_{\lfloor n/2 \rfloor}^{sort}$)

        Call Sort($L_{\lceil n/2 \rceil}$) (to get $L_{\lceil n/2 \rceil}^{sort}$)

        Call Merge($L_{\lfloor n/2 \rfloor}^{sort}, L_{\lceil n/2 \rceil}^{sort}$)

    end if

end

Merge(L_1,L_2): (this procedure returns L^{sort})

begin

    $L^{sort} \leftarrow empty - list$

    While (length(L_1)> 0 and length(L_2)> 0)

        if first(L_1) $\leq$ first(L_2)

            add first(L_1) to end of L^{sort}

            remove first(L_1) from L_1

        else

            add first(L_2) to end of L^{sort}

            remove first(L_2) from L_2

        end if

    end while

    if length(L_1) > 0

        append L_1 to the end of L^{sort}

    if length(L_2) > 0

        append L_2 to L^{sort}

    end if

    return L^{sort}

end

In words, the merge sort algorithm is as follows:

1. If the input sequence has fewer than two elements, return.
2. Partition the input sequence into two halves.
3. Sort the two subsequences using the same algorithm.
4. Merge the two sorted subsequences to form the output sequence.

Analysis:

Size of list	Comparisons
2	$n/2$
4	$3 \cdot n/4$
8	$7 \cdot n/8$
$\ldots$	$\ldots$
n	$n-1$

We have at most most $\log_2 n$ levels, with at most n comparisons, for the Merge$(,)$ procedure, in each. Therefore we perform at most $n \log_2 n$ comparisons.

An alternative way to find the number of comparisons is as follows. The work done in sorting a list of n elements is the work done to sort two lists of size $n/2$ each, and the work done to merge the two lists which is n.

$$f(n) = 2 \, f\left(\frac{n}{2}\right) + n$$

Using $f(1) = 1$, we can solve the above recurrence relation, and obtain $f(n) = n \log_2 n$. We conclude that the complexity complexity of this algorithm is: $n \log_2 n$.

5.2 Growth of functions

n	$\lceil \log n \rceil$	$n-1$	2^n	$n!$
1	0	0	2	1
3	2	2	8	6
5	3	4	32	120
10	4	9	1024	3628800
70	7	69	2^{70}	$\approx 2^{332}$

We are interested in the <u>asymptotic</u> behavior of the running time.

5.3 Definitions for asymptotic comparisons of functions

We define for functions f and g,

$$f, g \; : \; Z^+ \to [0, \infty) \;.$$

1. $f(n) \in O(g(n))$ if $\exists$ a constant $c > 0$ such that $f(n) \leq cg(n)$, for all n sufficiently large.
2. $f(n) \in o(g(n))$ if, for any constant $c > 0$, $f(n) < cg(n)$, for all n sufficiently large.
3. $f(n) \in \Omega(g(n))$ if $\exists$ a constant $c > 0$ such that $f(n) \geq cg(n)$, for all n sufficiently large.
4. $f(n) \in \omega(g(n))$ if, for any constant $c > 0$, $f(n) > cg(n)$, for all n sufficiently large.
5. $f(n) \in \Theta(g(n))$ if $f(n) \in O(g(n))$ and $f(n) \in \Omega(g(n))$.

Examples:
- Bubble sort has complexity $O(n^2)$
- Merge sort has complexity $O(n \log n)$
- Matrix multiplication has complexity $O(n^3)$

5.4 Properties of asymptotic notation

We mention a few properties that can be useful when analyzing the complexity of algorithms.

Proposition 5.2. $f(n) \in \Omega(g(n))$ *if and only if* $g(n) \in O(f(n))$.

The next property is often used in conjunction with *L'hôpital's Rule*.

Proposition 5.3. *Suppose that* $\lim_{n\to\infty} \frac{f(n)}{g(n)}$ *exists:*

$$\lim_{n\to\infty} \frac{f(n)}{g(n)} = c \ .$$

Then,

1. $c < \infty$ *implies that* $f(n) \in O(g(n))$.

2. $c > 0$ *implies that* $f(n) \in \Omega(g(n))$.

3. $c = 0$ *implies that* $f(n) \in o(g(n))$.

4. $0 < c < \infty$ *implies that* $f(n) \in \Theta(g(n))$.

5. $c = \infty$ *implies that* $f(n) \in \omega(g(n))$.

An algorithm is *good* or polynomial-time if the complexity is $O(\text{polynomial(length of input)})$. This polynomial must be of fixed degree, that is, its degree must be independent of the input length. So, for example, $O(n^{\log n})$ is not polynomial.

5.5 Caveats of complexity analysis

One should bear in mind a number of caveats concerning the use of complexity analysis.

1. **Ignores the size of the numbers.** The real model is a poor one when dealing with very large numbers, as each operation is given unit cost, regardless of the size of the numbers involved. But multiplying two huge numbers, for instance, may require more effort (and time) than multiplying two small numbers.

2. **Is worst case.** So complexity analysis does not say much about the average case. Traditionally, complexity analysis has been a pessimistic measure, concerned with worst-case behavior. The *simplex method* for linear programming is known to be exponential (in the worst case), while the *ellipsoid* algorithm is polynomial; but, for the ellipsoid method, the average case behavior and the worst case behavior are essentially the same, whereas the average case behavior of simplex is much better than it's worst case complexity, and in practice is preferred to the ellipsoid method.

    Similarly, Quicksort, which has $O(n^2)$ worst case complexity, is often chosen over other sorting algorithms with $O(n \log n)$ worst case complexity. This is because QuickSort has $O(n \log n)$ average case running time and, because the constants (that we ignore in the O notation) are smaller for QuickSort than for many other sorting algorithms, it is often preferred to algorithms with "better" worst-case complexity and "equivalent" average case complexity.

3. **Ignores constants.** We are concerned with the asymptotic behavior of an algorithm. But, because we ignore constants (as mentioned in QuickSort comments above), it may be that an algorithm with better complexity only begins to perform better for instances of inordinately large size.

   Indeed, this is the case for the $O(n^{2.375477})$ algorithm for matrix multiplication, that is "...*wildly impractical for any conceivable applications.*" [1]

4. **O(n^{100}) is polynomial.** An algorithm that is *polynomial* is considered to be "good". So an algorithm with $O(n^{100})$ complexity is considered good even though, for reasons already alluded to, it may be completely impractical.

Still, complexity analysis is in general a very useful tool in both determining the intrinsic "hardness" of a *problem* and measuring the quality of a particular *algorithm*.

5.6 A sketch of the ellipsoid method

We now outline the two key features of the ellipsoid method. A linear programming problem may be formulated in one of two ways - a primal and dual representation. The first key feature combines both formulations in a way that forces all feasible solutions to be optimal solutions.

Primal Problem:
$$\max\{c^T x : \ Ax \leq b, \ x \geq 0\} \tag{10}$$

Dual Problem:
$$\min\{b^T y : \ y^T A \geq c, y \geq 0\} \tag{11}$$

The optimal solution satisfies:
$$c^T x = b^T y \tag{12}$$

We create a new problem combining these two formulations:

$$
\begin{array}{rcl}
Ax & \leq & b \\
x & \geq & 0 \\
y^T A^T & \geq & c \\
y & \geq & 0 \\
b^T y & \leq & c^T x
\end{array}
$$

Note that by weak duality theorem $b^T y \geq c^T x$, so with the inequality above this gives the desired equality. Every feasible solution to this problem is an optimal solution. The ellipsoid algorithm works on this feasibility problem as follows:

1. If there is a feasible solution to the problem, then there is a basic feasible solution. Any basic solution is contained in a sphere (ellipsoid) of radius $n\Delta$, where Δ is the largest subdeterminant of the augmented constraint matrix.

2. Check if the center of the ellipsoid is feasible. If yes, we are done. If the number of iterations has exceeded the set limit, K, stop, output "infeasible". Otherwise, find a constraint that is violated at the center.

---

[1] See *Coppersmith D. and Winograd S.*, Matrix Multiplication via Arithmetic Progressions. *Journal of Symbolic Computation*, 1990 Mar, V9 N3:251-280.

3. This violated constraint specifies a hyperplane through the center and a half ellipsoid that contains all the feasible solutions.

4. Reduce the search space to the smallest ellipsoid that contains the feasible half ellipsoid.

5. Repeat the algorithm at the center of the new ellipsoid.

Obviously this algorithm is not complete. How do we know when to stop and decide that the feasible set is empty? It is proved that the volume of the ellipsoids scales down in each iteration by a factor of about $e^{-1/2(n+1)}$, where n is the number of linearly independent constraints. Also note that a basic feasible solution only lies on a grid of mesh length $(\Delta(\Delta - 1))^{-1}$. This is because when we write a component of a basic feasible solution as the quotient of two integers, the denominator cannot exceed Δ (by Cramer's Rule). Therefore the component-wise difference of two basic feasible solutions can be no less than $(\Delta(\Delta - 1))^{-1}$. Given this, it can be shown that the convex hull of basic feasible solutions has volume of at least $\frac{1}{2^{n+2}\Delta^{n+2}}$. While the volume of the original search space has volume $O(n^n \Delta^n)$.

The number of iterations K required until the volume goes below the minimum volume that contains all the feasible solutions satisfies,

$$(n\Delta)^n e^{-\frac{K}{2(n+1)}} \leq \frac{1}{2^{n+2}\Delta^{n+2}}$$

Therefore the algorithm terminates in at most $K = O(n^2 \log \Delta)$ iterations. Denoting $L = \log \Delta$ this number of iterations is $O(n^2 L)$. The amount of work per iteration is polynomial as well, and therefore the Ellipsoid is a polynomial time algorithm.

The ellipsoid method is the first polynomial time algorithm that was found for linear programming. However, it is not very efficient in practice.

6 Graph representations

There are different ways to store a graph in a computer, and it is important to understand how each representation may affect or improve the complexity analysis of the related algorithms. The differences between each representation are based on the way information is stored, so for instance, one representation may have a characteristic X very handy but characteristic Y may be very operation intensive to obtain, while an alternative representation may favor readiness of characteristic Y over characteristic X.

6.1 Node-arc adjacency matrix

This representation is relevant since it is the natural representation of the flow balance constraints in the LP formulation of Network Flow problems. This representation consists on a n by m matrix, so that each of its rows correspond to a node and each of its columns correspond to an arc. The $(i,j)^{th}$ entry of the Node-arc adjacency matrix is {1,-1}, if i^{th} node is the start-point or endpoint (respectively) of the jth edge and {0} otherwise.

As mentioned in an earlier lecture, the node-arc adjacency matrix is *totally unimodular*.

The density of a node-arc adjacency matrix is $\frac{2}{n}$

6.2 Node-node adjacency matrix

This representation consists on a n by n matrix, so that each of its rows and each of its columns correspond to a node and the row-column intersections correspond to a (possible) edge. The $(i,j)^{th}$

entry of the Node-Node Adjacency Matrix is 1, if there is an edge from the ith node to the jth node and 0 otherwise. In the case of undirected graphs it is easy to see that the Node-Node Adjacency Matrix is a symmetric matrix.

The network is described by the matrix N, whose (i, j)th entry is n_{ij}, where

$$n_{ij} = \begin{cases} 1 & \text{if (i, j)} \in \text{E} \\ 0, & \text{otherwise} \end{cases}$$

Figure 10 depicts the node-node adjacency matrices for directed and undirected graphs.

(a) node-node adjacency matrix for a directed graph

(b) node-node adjacency matrix for an undirected graph
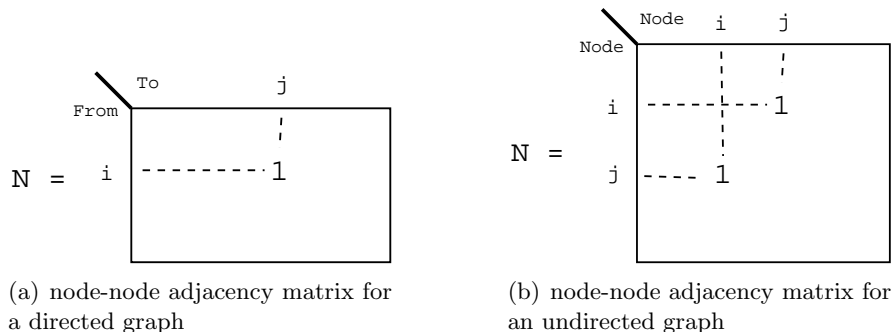
Figure 10: Node-adjacency lists

Note: For undirected graphs, the node-node adjacency matrix is symmetric. The density of this matrix is $\frac{2|E|}{|V|^2}$ for an undirected graph, and $\frac{|A|}{|V|^2}$, for a directed graph. If the graph is complete, then $|A| = |V| \cdot |V - 1|$ and the matrix N is dense.

Is this matrix a good representation for a breadth-first-search? (discussed in the next section). To determine whether a node is adjacent to any other nodes, n entries need to be checked for each row. For a connected graph with n nodes and m edges, $n - 1 \leq m \leq n(n-1)/2$, and there are $O(n^2)$ searches.

6.3 Node-adjacency list

Consider a *tree,* which is a very sparse graph with $|E| = |V| - 1$. If we represent an undirected tree using the matrices described above, we use a lot of memory. A more compact representation is to write each node and its associated adjacency list.

This representation consists on n linked lists, each one corresponding to a node (i) and storing all the nodes (j) for which the edge (i,j) is present in the graph. The importance of this representation is that it is the most space efficient and it normally leads to the most efficient algorithms as well.

Node	Adjacent Nodes
1	*Neighbors of node* 1
2	*Neighbors of node* 2
⋮	⋮
n	*Neighbors of node* n

Such a representation requires $2\,|E| + |V|$ memory spaces as opposed to $|E| \cdot |V|$ memory spaces for the matrix representation.

There are 2 entries for every edge or arc, this results in $O(m+n)$ size of the adjacency list representation. This is the most compact representation among the three presented here. For sparse graphs it is the preferable one.

6.4 Comparison of the graph representations

6.4.1 Storage efficiency comparison

Representation	Storage	Non-Zeros	Density	Notes
Node-Arc Matrix	$O(mn)$	$O(2m)$	$2/n$	Inefficient
Node-Node Matrix	$O(n^2)$	$O(m)$	m/n^2	Efficient for Dense Graphs
Node-Adjacency List	$O(m+n)$	0	1	Efficient for Sparse Graphs

6.4.2 Advantages and disadvantages comparison

Representation	Advantages	Disadvantages
Node-Arc Matrix	Nice Theoretical Properties	Not efficient in practice
Node-Node Matrix	Existence of edge in $O(1)$ time	List neighbors in $O(n)$ time
Node-Adjacency List	List neighbors in $O(deg(i))$ time	Existence of edge in $O(deg(i))$ time

7 Graph search algorithms

Given a graph $G = (V, A)$ and a source node $s \in V$, we want to check if all the nodes $\in V$ can be reached by s using only the arcs $\in A$. Essentially we want the list of all nodes reachable from s and possibly also the path from s to each node. As seen on class, this set of paths must form a tree, this follows from the correctness-proof of the LP formulation of the single source shortest path problem.

7.1 Generic search algorithm

A generic search algorithm starts with the special node, designates it as the root of a tree, and places it in a list L. The algorithm then selects at each iteration a node from the list, and find a neighbor (out-neighbor for a directed graph), if exists, and adds it to the list. The process continues until all nodes have been visited, or all edges (arcs) have been scanned. The selection mechanism from the list differentiates between search algorithms.

In the procedure below $N_E(v)$ is a neighbor of node v in the graph induced by the set of edges E. For a directed graph, the same algorithm applies with $N_A^+(v)$ referring to an out-neighbor of v with respect to the set of arcs A.

Pseudocode for a generic search algorithm
$L \leftarrow \{root\}$, $V \leftarrow V \setminus \{root\}$;
While $L \neq \emptyset$
    Select $v \in L$ (use selection rule);
    If there exists $N_E(v) = u$, $u \in V \setminus L$,
        $L \leftarrow L \cup \{u\}$; $V \leftarrow V \setminus \{u\}$; $E \leftarrow E \setminus \{[v, u]\}$;
    else $L \leftarrow L \setminus \{v\}$;
    end if
end while

Two basic methods that are used to define the selection mechanism are *Breadth-First-Search* (BFS) and *Depth-First-Search* (DFS).
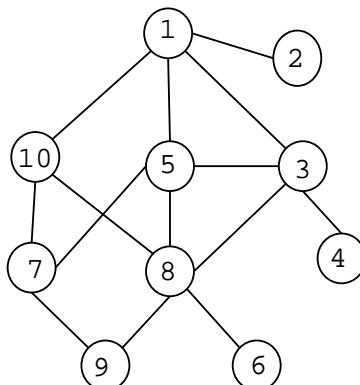
Figure 11: Graph to be searched

7.2 Breadth first search (BFS)

Breadth-first-search employs a *first-in-first-out* (FIFO) strategy and is usually implemented using a queue. To obtain the pseudocode of BFS we only need to modify the pseudocode for the generic search so that when we remove the edge (u, v) from L we remove it from the **start** of L.

In breadth-first-search, the graph is searched in an exhaustive manner; that is, all previously unscanned neighbors of a particular node are tagged and added to the end of the reachability list (L), and then the previously unscanned neighbors of the next node in the list are added, etc.

The original graph, rearranged in a breadth-first-search, shows the typically short and bushy shape of a BFS tree as shown in Figure 12.
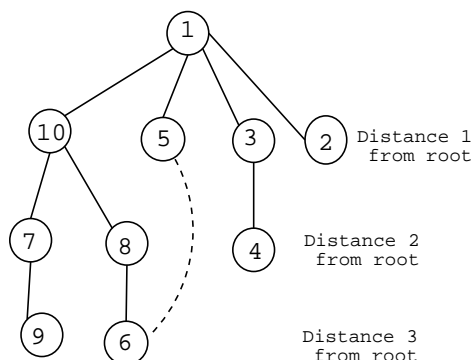
Figure 12: Solid edges represent the BFS tree of the example graph. The dotted edges cannot be present in a BFS tree.

Note that the BFS tree is not unique and depends on the order in which the neighbors are visited. Not all edges from the original graph are represented in the tree. However, we know that the original graph cannot contain any edges that would bridge two nonconsecutive levels in the tree. By level we mean a set of nodes such that the number of edges that connect each node to the root of the tree is the same.

Consider any graph. Form its BFS tree. Any edge belonging to the original graph and not in the BFS can either be from level i to level $i + 1$; or between two nodes at the same level. For

example, consider the dotted edge in Figure 12 from node 5 to node 6. If such an edge existed in the graph, then node 6 would be a child of node 5 and hence would appear at a distance of 2 from the root.

Note that an edge forms an even cycle if it is from level i to level $i+1$ in the graph and an odd cycle if it is between two nodes at the same level.

Theorem 7.1 (BFS Property 1). *The level of a node, $d(v)$, is the distance from s to v.*

Proof. First, it is clear that every element reachable from s will be explored and given a d.

By induction on $d(v)$, Q remains ordered by d, and contains only elements with d equal to k and $k+1$. Indeed, elements with $d = k$ are dequeued first and add elements with $d = k+1$ to Q. Only when all elements with $d = k$ have been used do elements with $d = k+2$ start being introduced. Let "stage k" be the period where elements with $d = k$ are dequeued.

Then, again by induction, at the beginning of stage k, all nodes at distance from s less than k have been enqueued, and only those, and they contain their correct value. Assume this for k. Let v be a node at distance $k+1$. It cannot have been enqueued yet at the beginning of stage k, by assumption, so at that point $d(v) = \infty$. But there exists at least one node u at distance k that leads to u. u is in Q, and $d(u) = k$ by assumption. The smallest such u in Q will give v its value $d(v) = d(u) + 1 = k+1$. Nodes at distance more than $k+1$ have no node at distance k leading to them and will not be enqueued. □

Corollary 7.2 (BFS Property 2). *For undirected graphs, if $\{i,j\} \in E$ then $|d(i) - d(j)| \leq 1$*
(i.e. neighbors must be in adjacent levels)
For directed graphs if $(i,j) \in A$ then $|d(i) - d(j)| \leq 1$
(i.e. Can't skip levels in forward arcs)

Claim 7.3 (BFS Complexity). *BFS runs in $O(m+n)$ time.*

7.3 Depth first search (DFS)

Depth-first-search employs a *last-in-first-out* (LIFO) strategy and is usually implemented using a stack. To obtain the pseudocode of BFS we only need to modify the pseudocode for the generic search so that when we remove the edge (u,v) from L we remove it from the **end** of L.

In depth-first-search, one searches for a previously unscanned neighbor of a given node. If such a node is found, it is added to the end of the list and is scanned out from; otherwise, the original node is removed from the list, and we scan from the node now at the end of the list.

The same graph, rearranged in a depth-first-search demonstrates the typically long and skinny shape of a DFS tree as shown in Figure 13.

Claim 7.4 (DFS Complexity). *DFS runs in $O(m+n)$ time.*

7.4 Applications of BFS and DFS

7.4.1 Checking if a graph is strongly connected

Definition 7.5. *A graph is **strongly connected** if we can reach all nodes $\in V$ starting from any other node $\in V$.*

Naive Algorithm
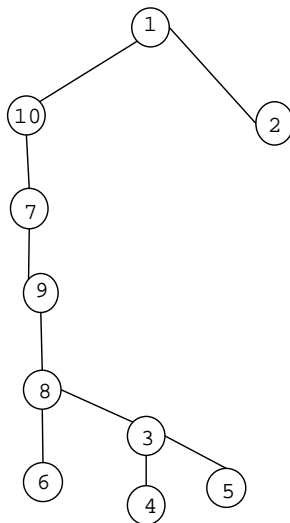
Figure 13: DFS tree of the example graph

```
00: SC(G)
01: For every node in V do
02:    R <-- DFS(G,v); //or BFS
03:    If R not equal to V then STOP AND OUTPUT "FALSE"
04: OUTPUT "TRUE"
```

Since DFS/BFS take $O(m+n)$ time and we have $O(n)$ executions, the complexity of the naive algorithm is $O(mn)$.

Claim 7.6. *Checking strong connectivity in any graph takes $O(m+n)$ time.*

Proof.

For Undirected Graphs

Only one run of DFS or BFS is needed; since, if from any arbitrary node v we can reach all other nodes $\in V$, then we can reach any node from any other node at least through a path containing node v.

For Directed Graphs

From an arbitrarily node v run one DFS or BFS, if it can reach all the other nodes $\in V$, then "flip/reverse" all the edges in G and run once more BFS/DFS starting again from v. It the second time again v can reach all the other nodes $\in V$, then this means that in the original graph every node can reach v. Finally, if every node can reach v, and v can reach every other node, then every node can reach any other node through a path containing v.

In any case we will call DFS/BFS once or twice respectively and this will take $O(m+n)$ time. $\square$

7.4.2 Checking if a graph is acyclic

Definition 7.7. *A graph is **acyclic** if it does not contain cycles.*

Claim 7.8. *Checking if a graph is acyclic takes $O(m+n)$ time.*

Proof. Run DFS from an arbitrarily node, if there is an arc to a node already visited, then the graph contains a cycle. This takes $O(m+n)$ time. $\square$

Although we can use also BFS to detect cycles, DFS tends to find a cycle 'faster'. This is because if the root is not part of the cycle BFS still needs to visit all of its neighbors; while DFS will move away from the root (and hopefully towards the cycle) faster. Also, if the cycle has length k then BFS needs at least to explore floor$(k/2)$ levels.

7.4.3 Checking if a graph is bipartite

Definition 7.9. *A graph is **bipartite** if V can be partitioned into two sets such that all edges/arcs in E/A have an endpoint on each of the two sets.*

Claim 7.10. *Checking if a graph is bipartite takes $O(m + n)$ time.*

Proof. We know that a graph is bipartite if and only if it does not contains a an odd cycle. So we can run BFS from an arbitrary node v, and if it contains an edge between nodes in the same level then the graph has an odd cycle, and thus it is not bipartite. $\square$

Similarly, we can determine if a graph has an *odd* cycle: An edge found from a node to a child (of a previously processed node) in the next level. There is another way of detecting an odd cycle; By assigning two alternating colors to nodes visited with any search algorithm: if there is an edge from a node of a given color then the neighbor gets the second color. If an edge is discovered between two nodes of same color, then there is an odd cycle.

8 Shortest paths

8.1 Introduction

Consider graph $G = (V, A)$, with cost c_{ij} on arc (i, j). There are several different, but related, shortest path problems:

- *Shortest $s - t$ path:* Find a shortest path from s to t (single pair).

- *Single source shortest paths:* Find shortest path from s to all nodes.

- *All pairs shortest paths:* Find the SP between every pair of nodes.

Additionally, we differentiate between shortest paths problems depending on the type of graph we receive as input.

We will begin by considering shortest paths (or longest paths) in *Directed Acyclic Graphs* (DAGs).

8.2 Properties of DAGs

A DAG is a Directed Acyclic graph, namely, there are no directed cycles in the graph.

Definition 8.1. *A Topological ordering is the assignments of numbering to the nodes $1, 2, \cdots, n$ such that all arcs go in the forward direction:*

$$(i, j) \in E \Rightarrow i < j$$

It should be noted that a DAG can have $O(n^2)$ arcs. We can construct such graph by the following construction: Given a set of nodes, number them arbitrarily and add all arcs (i, j) where $i < j$.

8.2.1 Topological sort and directed acyclic graphs

Given a directed graph (also called a *digraph*), $G = (N, A)$, we wish to *topologically sort* the nodes of G; that is, we wish to assign labels $1, 2, \ldots, |N|$ to the nodes so that $(i, j) \in A \Rightarrow i < j$.

A topological sort does not exist in general, and a trivial example to show this is a graph with two nodes, with directed edges from each node to the other. It does, however, always exist for acyclic graphs. To prove this, the following definition and lemma will be useful.

Definition 8.2. *A (graph) property, P, is* **hereditary** *if the following is true: given a graph G with property P, any subgraph of G also has property P.*

Some examples of hereditary properties:
- Bipartiteness.
- Having no directed cycles (i.e. acyclic, or a DAG)

Examples of non-hereditary properties:
- Connectivity.
- Being an Eulerian graph.

Lemma 8.3. *In DAG, G, there always exists a node with in-degree* 0*. (Similarly, a DAG always has a node with out-degree* 0*.)*

Proof. Assume by contradiction that G there is no node with in-degree 0. We arbitrarily pick a node i in G, and apply a Depth-First-Search by visiting the predecessor nodes of each such node encountered. Since every node has a predecessor, when the nodes are exhausted (which must happen since there are a finite number of nodes in the graph), we will encounter a predecessor which has already been visited. This shows that there is a cycle in G, which contradicts our assumption that G is acyclic. □

Theorem 8.4. *There exists a topological sort for a digraph G if and only if G is a DAG (i.e. G has no directed cycles).*

Proof.
[**Only if part**] Suppose there exists a topological sort for G. If there exists cycle $(i_1,...,i_k,i_1)$, then, by the definition of topological sort above, $i_1 < i_2 < ... < i_k < i_1$, which is impossible. Hence, if there exists a topological sort, then G has no directed cycles.
[**If part**] In a DAG, the nodes can be topologically sorted by using the following procedure. Assigning a node with in-degree 0 the lowest remaining label (by the previous lemma, there always exists such a node). Removing that node from the graph (recall that being a DAG is hereditary), and repeat until all nodes are labeled. The correctness of this algorithm is established by induction on the label number. The pseudocode for the algorithm to find a topological sort is given below. □

Topological sort (pseudocode):
$i \leftarrow 1$
While $V \neq \emptyset$
    Select $v \in V$ with in-degree$_V(v) = 0$
    label$(v) \leftarrow i$
    $i \leftarrow i + 1$
    $V \leftarrow V \setminus \{v\}; A \leftarrow A \setminus \{(v, w) \in A\}$
    update in-degrees of neighbors of v
end while

Note that the topological sort resulting from this algorithm is not unique. Further note that an alternative approach works also: Take a node with out-degree 0, assign it the highest remaining label, remove it from the graph, and repeat until no nodes remain.

An example of the topological order obtained for a DAG is depicted in Figure 14. The numbers in the boxes near the nodes represent a topological order.
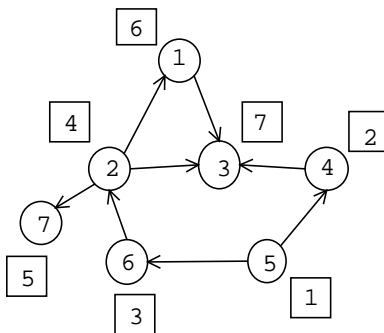
Figure 14: An example of topological ordering in a DAG. Topological orders are given in boxes.

Consider the complexity analysis of the algorithm. We can have a very loose analysis and say that each node is visited at most once and then its list of neighbors has to be updated (a search of $O(m)$ in the node-arc adjacency matrix, or $O(n)$ in the node-node adjacency matrix, or the number of neighbors in the adjacency list data structure). Then need to find among all nodes one with indegree equal to 0. This algorithm runs in $O(mn)$ time, or if we are more careful it can be done in $O(n^2)$. The second factor of n is due to the search for a indegree 0 node. To improve on this we maintain a "bucket" or list of nodes of indegree 0. Each time a node's indegree is updated, we check if it is 0 and if so add it to the bucket. Now to find a node of indegree 0 is done in $O(1)$, just lookup the bucket in any order. Updating the indegrees is done in $O(m)$ time total throughout the algorithm, as each arc is looked at once. Therefore, *Topological Sort* can be implemented to run in $O(m)$ time.

8.3   Properties of shortest paths

The Shortest Path Problem (henceforth SP problem) on a digraph $G = (V, A)$ involves finding the least cost path between two given nodes $s, t \in V$. Each arc $(i, j) \in A$ has a weight assigned to it, and the cost of a path is defined as the sum of the weights of all arcs in the path.

The property proved in the next proposition is a rephrasing of the *principle of optimality* (first stated by Richard Bellman in 1952), that serves to justify dynamic programming algorithms.

Proposition 8.5. *If the path $(s = i_1, i_2, \ldots, i_h = t)$ is a shortest path from node s to t, then the subpath $(s = i_1, i_2, \ldots, i_k)$ is a shortest path from source to node i_k, $\forall\ k = 2, 3, \ldots, h - 1$. Furthermore, for all $k_1 < k_2$ the subpath $(i_{k_1}, i_{k_1+1}, \ldots, i_{k_2})$ is a shortest path from node i_{k_1} to i_{k_2}.*

Proof. The properties given above can be proved by a contradiction-type argument. $\qquad\square$

Proposition 8.6. *Let the vector $\vec{d}$ represent the shortest path distances from the source node. Then*

1. *$d(j) \leq d(i) + c_{ij},\ \forall\, (i, j) \in A$.*

2. *A directed path from s to k is a shortest path if and only if $d(j) = d(i) + c_{ij},\ \forall\, (i, j) \in P$.*

The property given above is useful for backtracking and identifying the tree of shortest paths.

Given the distance vector $\vec{d}$, we call an arc *eligible* if $d(j) = d(i) + c_{ij}$. We may find a (shortest) path from the source s to any other node by performing a breadth first search of eligible arcs. The graph of eligible arcs looks like a tree, plus some additional arcs in the case that the shortest paths are not unique. But we can choose one shortest path for each node so that the resulting graph of all shortest paths from s forms a tree.

Aside: When solving the shortest path problem by linear programming (MCNF formulation), a basic solution is a tree. This is because a basic solution is an independent set of columns; and a cycle is a dependent set of columns (therefore "a basic solution cannot contain cycles").

Given an undirected graph with nonnegative weights. The following analogue algorithm solves the shortest paths problem:

The String Solution: Given a shortest path problem on an undirected graph with non-negative costs, imagine physically representing the vertices by small rings (labeled 1 to n), tied to other rings with string of length equaling the cost of each edge connecting the corresponding rings. That is, for each arc (i, j), connect ring i to ring j with string of length c_{ij}.

Algorithm: To find the shortest path from vertex s to all other vertices, hold ring s and let the other nodes fall under the force of gravity.

Then, the distance, $d(k)$, that ring k falls represents the length of the shortest path from s to k. The strings that are taut correspond to eligible arcs, so this arcs will form the tree of shortest paths.

8.4   Alternative formulation for SP from s to t

We have already seen how to model the single source shortest path problem as a Minimum Cost Network Flow problem. Here we will give an alternative mathematical programming formulation for the SP problem.

$$
\begin{aligned}
\max \quad & d(t) \\
d(j) \;\leq\; & d(i) + c_{ij} \; \forall (i,j) \in A \\
d(s) \;=\; & 0
\end{aligned}
$$

Where the last constraint is needed as an anchor, since otherwise we would have an infinite number of solutions. To see this, observe that we can rewrite the first constraint as $d(j) - d(i) \leq c_{ij}$; thus, given any feasible solution $\mathbf{d}$, we can obtain an infinite number of feasible solutions (all with same objective value) by adding a constant to all entries of $\mathbf{d}$.

This alternative mathematical programming formulation is nothing else but the dual of the minimum-cost-network-flow-like mathematical programming formulation of the shortest path problem.

8.5   Shortest paths on a directed acyclic graph (DAG)

We consider the single source shortest paths problem for a directed acyclic graph G.

We address the problem of finding the SP between nodes i and j, such that $i < j$, assuming the graph is topologically ordered to start with. It is beneficial to sort the graph in advance, since

we can trivially determine that there is no path between i and j if $i > j$; in other words, we know that the only way to each node j is by using nodes with label less than j. Therefore, without loss of generality, we can assume that the source is labeled 1, since any node with a smaller label than the source is unreachable from the source and can thereby be removed.

Given a topological sorting of the nodes, with node 1 the source, the shortest path distances can be found by using the recurrence,

$$d(j) \;=\; \min_{\substack{(i,j)\in A \\ 1\le i<j}} \{d(i) + c_{ij}\}\,.$$

The validity of the recurrence is easily established by induction on j. It also follows from property 8.6. The above recursive formula is referred to as a dynamic programming equation (or Bellman equation).

What is the complexity of this dynamic programming algorithm? It takes $O(m)$ operations to perform the topological sort, and $O(m)$ operations to calculate the distances via the given recurrence (we obviously compute the recurrence equation in increasing number of node labels). Therefore, the distances can be calculated in $O(m)$ time. Note that by keeping track of the i's that minimize the right hand side of the recurrence, we can easily backtrack to reconstruct the shortest paths.

It is important how we represent the **output** to the problem. If we want a list of all shortest paths, then just writing this output takes $O(n^2)$ (if we list for every node its shortest path). We can do better if we instead just output the shortest path tree (list for every node its predecessor). This way the output length is $O(m)$.

Longest Paths in DAGs:
If we want instead to find the longest path in a DAG, then one possible approach would be to multiply by -1 all the arc costs and use our shortest path algorithm. However, recall that solving the shortest path problem in graphs with negative cost cycles is "a very hard problem". Nevertheless, if the graph where we want to find the longest path is a DAG, then it does not have cycles. Therefore by multiplying by -1 the arc costs we cannot create negative cycles. In conclusion, our strategy will indeed work to find the longest path in a DAG. Alternatively we can find the longest path from node 1 to every other node in the graph, simply by replacing *min* with *max* in the recurrence equations give above.

8.6 Applications of the shortest/longest path problem on a DAG

Maximizing Rent

See the handout titled *Maximizing Rent: Example.*

The graph shown in Figure 15 has arcs going from the arrival day to the departure day with the bid written on the arcs. It is particularly important to have arcs with cost 0 between consecutive days (otherwise we would not allow the place to be empty for some days–which is NOT our objective, what we want is to maximize the rent obtained!). These arcs are shown as dotted arcs in the graph. To find the maximum rent plan one should find the longest path in this graph. The graph has a topological order and can be solved using the shortest path algorithm. (Note that the shortest and longest path problems are equivalent for acyclic networks, but dramatically different when there are cycles.)

Equipment Replacement Minimize the total cost of buying, selling, and operating the equipment over a planning horizon of T periods.
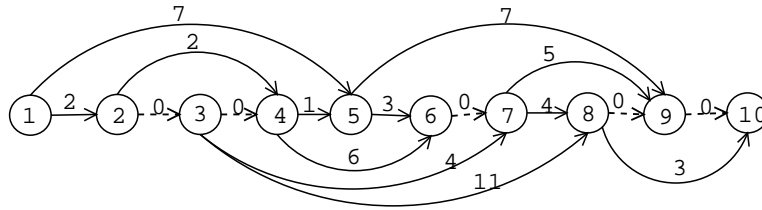
Figure 15: Maximizing rent example

See the handout titled *The Equipment Replacement Problem.*

Lot Sizing Problem Meet the prescribed demand d_j for each of the periods $j = 1, 2, ..., T$ by producing, or carrying inventory from previous production runs. The cost of production includes fixed set up cost for producing a positive amount at any period, and a variable per unit production cost at each period. This problem can be formulated as a shortest path problem in a DAG (AM&O page 749) if one observes that any production will be for the demand of an integer number of consecutive periods ahead. You are asked to prove this in your homework assignment.

Project Management - Critical Path Method (CPM)
See the handout titled *Project Formulation for Designer Gene's Relocation.*

This is a typical project management problem. The objective of the problem is to find the earliest time by which the project can be completed.

This problem can be formulated as a longest path problem. The reason we formulate the problem as a *longest* path problem, and not as a *shortest* path problem (even though we want to find the *earliest* finish time) is that *all* paths in the network have to be traversed in order for the project to be completed. So the longest one will be the "bottleneck" determining the completion time of the project.

We formulate the problem as a longest path problem as follows. We create a node for each task to be performed. We have an arc from node i to node j if activity i must precede activity j. We set the length of arc (i, j) equal to the duration of activity i. Finally we add two spacial nodes *start* and *finish* to represent the *start* and the *finish* activities (these activities with duration 0). Note that in this graph we cannot have cycles, thus we can solve the problem as a longest path problem in an acyclic network.

Now we give some notation that we will use when solving this type of problems. Each node is labeled as a triplet (node name, time in node, earliest start time). Earliest start time of a node is the earliest time that work can be started on that node. Hence, the objective can also be written as the earliest start time of the finish node.

Then we can solve the problem with our dynamic programming algorithm. That is, we traverse the graph in topological order and assign the label of each node according to our dynamic programming equation:

$$t_j = \max_{(i,j)}\{t_i + c_{ij}\},$$

where t_i is the earliest start time of activity i; and c_{ij} is the duration of activity i.

To recover the longest path, we start from node t_{finish}, and work backwards keeping track of the preceding activity i such that $t_j = t_i + c_{ij}$. Notice that in general more than one activity might satisfy such equation, and thus we may have several longest paths.

Alternatively, we can formulate the longest path as follows:

$$
\begin{array}{rcl}
\min & & t_{finish} \\
t_j & \geq & t_i + c_{ij} \ \forall (i,j) \in A \\
t_{start} & = & 0
\end{array}
$$

From the solution of the linear program above we can identify the longest path as follows. The constraints corresponding to arcs on the longest path will be satisfied with equality. Furthermore, for each unit increase in the length of these arcs our objective value will increase also one unit. Therefore, the dual variables of these constraints will be equal to one (while all other dual variables will be equal to zero–by complementary slackness).

The longest path (also known as critical path) is shown in Figure 16 with thick lines. This path is called critical because any delay in a task along this path delays the whole project path.
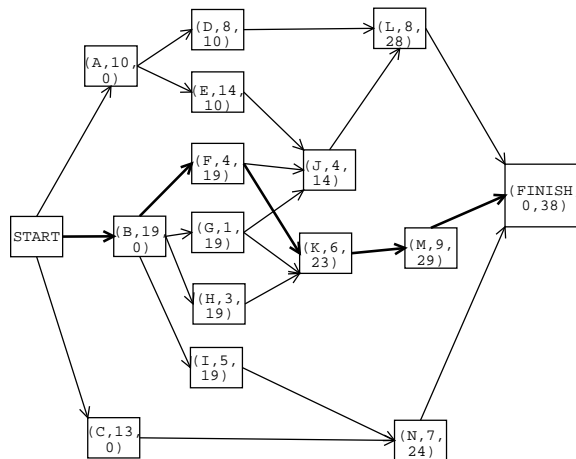
Figure 16: Project management example. Notation (activity, duration, earliest-start-time).

Binary knapsack problem

Problem: Given a set of items, each with a weight and value (cost), determine the subset of items with maximum total weight and total cost less than a given budget.

The binary knapsack problem's mathematical programming formulation is as follows.

$$
\begin{array}{rcl}
\max & \displaystyle\sum_{j=1}^{n} w_j x_j & \\
& \displaystyle\sum_{j=1}^{n} v_j x_j & \leq \ B \\
& x_j & \in \ \{0,1\}
\end{array}
$$

The binary knapsack problem is a hard problem. However it can be solved as a longest path problem on a DAG as follows.

Let $f_i(q)$ be the maximum weight achievable when considering the first i items and a budget of q, that is,

$$f_i(q) = \max \quad \sum_{j=1}^{i} w_j x_j$$

$$\sum_{j=1}^{i} v_j x_j \leq q$$

$$x_j \in \{0, 1\}$$

Note that we are interested in finding $f_n(B)$. Additionally, note that the $f_i(q)$'s are related with the following dynamic programming equation:

$$f_{i+1}(q) = \max\{f_i(q), f_i(q - v_{i+1}) + w_{i+1}\}.$$

The above equation can be interpreted as follows. By the principle of optimality, the maximum weight I can obtain, when considering the first $i + 1$ items will be achieved by either:

1. Not including the $i+1$ item in my selection. In which case my total weight will be equal to $f_i(q)$, i.e. equal to the maximum weight achievable with a budget of q when only considering only the first i items; or
2. Including the $i + 1$ item in my selection. In which case my total weight will be equal to $f_i(q - v_{i+1}) + w_{i+1}$, i.e. equal to the maximum weight achievable when considering the first i items with a budget of $q - v_{i+1}$ plus the weight of this $i + 1$ object.

We also have the the boundary conditions:

$$f_1(q) = \begin{cases} 0 & 0 \leq q < v_1 \\ w_1 & B \geq q \geq v_1 \end{cases}.$$

The graph associated with the knapsack problem with 4 items, $w = (6, 8, 4, 5)$, $v = (2, 3, 4, 4)$, and $B=12$ is given in Figure 17. (The figure was adapted from: Trick M., A dynamic programming approach for consistency and propagation of knapsack constraints.)

Note that in general a *knapsack graph* will have $O(nB)$ nodes and $O(nB)$ edges (each node has at most two incoming arcs), it follows that we can solve the binary knapsack problem in time $O(nB)$.

Remark: The above running time is **not** polynomial in the length of the input. This is true because the number B is given using only $\log B$ bits. Therefore the size of the input for the knapsack problem is $O(n \log B + n \log W)$ (where W is the maximum w_i). Finally, since, $B = 2^{\log B}$, then a running time of $O(nB)$ is really exponential in the size of the input.

8.7   Dijkstra's algorithm

Dijkstra's algorithm is a special purpose algorithm for finding shortest paths from a single source, s, to all other nodes in a graph with non-negative edge weights.

Dijkstra's algorithm assigns distance labels (from node s) to all other nodes in the graph. Node labels are either *temporary* or *permanent*. Initially all nodes have temporary labels. At any
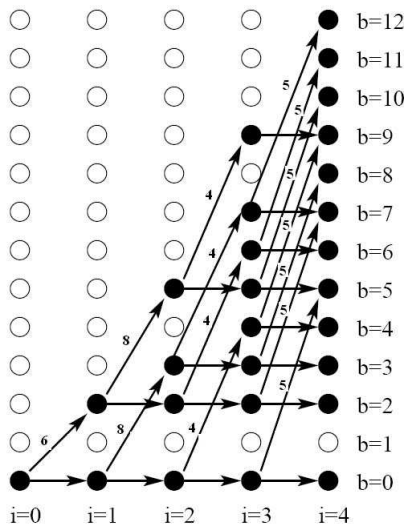
Figure 17: Graph formulation of a binary knapsack problem.

iteration, a node with the least distance label is marked as permanent, and the distances to its successors are updated. This is continued until no temporary nodes are left in the graph.

We give the pseudocode for Dijkstra's algorithm below. Where P is the set of nodes with permanent labels, and T of *temporarily labeled nodes*. Let $s = 1$.

> begin
> $N^+(i) := \{j|(i,j) \in A\}$;
> $P := \{1\}; T := V \setminus \{1\}$;
> $d(1) := 0$ and $pred(1) := 0$;
> $d(j) := c_{1j}$ and $pred(j) := 1$ for all $j \in N^+(1)$,
> and $d(j) := \infty$ otherwise;
> while $P \neq V$ do
> > begin
> > (Node selection, also called $FINDMIN$)
> > let $i \in T$ be a node for which $d(i) = min\{d(j) : j \in T\}$;
> > $P := P \cup \{i\}; T := T \setminus \{i\}$;
> > (Distance update)
> > for each $j \in N^+(i)$ do
> > > if $d(j) > d(i) + c_{ij}$ then
> > > $d(j) := d(i) + c_{ij}$ and $pred(j) := i$;
> > end
> end

We start from one invariant of Dijkstra's algorithm. Let i_k be the node that joins P in iteration k, and $d^k()$ the node labels at iteration k. The invariant states that the labels of the nodes that join P can only go up.

Claim 8.7 (Invariant 1 of Dijkstra's algorithm). *For every index k, $k = 1, \ldots, n - 1$, $d^k(i_k) \leq d^{k+1}(i_{k+1})$.*

Proof. At iteration k, $d^k(i_k) \leq d^k(i_{k+1})$, since i_k had the lowest label among the temporary nodes

at iteration k.
$$d^{k+1}(i_{k+1}) = \min\{d^k(i_{k+1}), d^k(i_k) + c_{i_k,i_{k+1}}\} \geq d^k(i_k). \qquad \square$$

A second invariant of Dijkstra's algorithm is that the labels of nodes $d^k(i)$ at iteration k are the shortest paths labels on paths that include only nodes of P^k as intermediate nodes. Here P^k is the set of permanent nodes at iteration k.

Claim 8.8 (Invariant 2 of Dijkstra's algorithm). *The labels of all nodes at iteration k, $d^k()$, are the shortest paths labels on paths that include only nodes of P^k as intermediate nodes.*

Proof. By induction on iteration number k. Obviously holds for $k = 1$ for $P^1 = \{s\}$. Assume for k and prove for $k + 1$.

By induction $d^k()$ are the shortest paths labels on paths that include only nodes of P^k as intermediate nodes. Consider iteration $k + 1$. The shortest paths that include only nodes of P^{k+1} as intermediate nodes, either include only nodes of P^k, or else they include also i_{k+1}. If the shortest path to node i contains only nodes of P^k, then $d^k(i) = d^{k+1}(i)$ is the correct length by the induction hypothesis. (Note that because of the previous claim, the label of i_{k+1} is the largest among nodes of P^{k+1}.) Otherwise, labels of i that change in iteration $k + 1$ must include also i_{k+1}, and i_{k+1} is the last node and direct predecessor of i. The length of such paths is $d^k(i_{k+1}) + c_{i_{k+1},i}$, which by induction on $d^k(i_{k+1})$ include only nodes of P^k, and now node i_{k+1} is included as well. $\qquad \square$

Theorem 8.9 (The correctness of Dijkstra's algorithm). *Once a node joins P its label is the shortest path label: $d^k(i_{k+1})$ is the shortest path label of i_{k+1}.*

Proof. Denote $i^* = i_{k+1}$. $P = P^k$, $T = V \setminus P$.
Suppose not, then the shortest path length $d^*(i^*) < d^k(i^*)$ and on any shortest path to i^* there must be nodes of T. Consider, one such shortest path to i^*. Let j_1 be the first node of T on this shortest path to i^*. Then the section of the path from s to j_1 has only nodes of P as intermediate nodes and is of length $d^k(j_1)$. Hence, $d^k(j_1) \geq d^k(i^*)$. Since the sub path from j_1 to i^* must be of non-negative length, this path must be at least as long as $d^k(i^*)$. Contradiction with $d^*(i^*) < d^k(i^*)$. $\qquad \square$

Complexity Analysis
There are two major operations in Dijkstra's algorithm :
Find minimum, which has $O(n^2)$ complexity; and
Update labels, which has $O(m)$ complexity.
Therefore, the complexity of Dijkstra's algorithm is $= O(n^2 + m)$.

There are several implementations that improve upon this running time.
One improvement uses *Binary Heaps*: whenever a label is updated, use binary search to insert that label properly in the sorted array. This requires $O(\log n)$. Therefore, the complexity of finding the minimum label in temporary set is $O(m \log n)$, and complexity of the algorithm is $O(m + m \log n) = O(m \log n)$.
Another improvement is *Radix Heap* which has complexity $O(m + n\sqrt{\log C})$ where $C = max_{(i,j)\in A} c_{ij}$. Note that this complexity is not strongly polynomial.
Currently, the best *strongly polynomial* implementation of Dijkstra's algorithm uses *Fibonacci Heaps* and has complexity $O(m + n \log n)$. Since Fibonacci heaps data structure is difficult to program, this implementation is not used in practice.

It is important to stress that Dijkstra's algorithm does not work correctly in the presence of negative edge weights. In this algorithm once a node is included in the permanent set it will not be checked later again, but because of the presence of negative edge that is not correct. The problem is that the distance label of a node in the permanent set might be reduced after its inclusion in the permanent set (when considering negative cost arcs).

See Figure 18 for an example such that Dijkstra's algorithm does not work in presence of negative edge weight. Using the algorithm, we get $d(3) = 3$. But the actual value of $d(3)$ is 2.
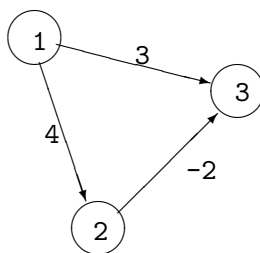
Figure 18: Network for which Dijkstra's algorithm fails

8.8 Bellman-Ford algorithm for single source shortest paths

This algorithm can be used in networks that contain negative weight edges. Such algorithms must be able to detect the existence of negative cost cycles which will have to be part of the correctness proof. The algorithm is simple, consisting of n "pulses", where a pulse is the update of labels along all m arcs.

Input: A (general) graph $G = (V, A)$, and a source node $s = 1$.
Output: Single source shortest paths from 1.
Pseudocode
$Initialize : d(1) := 0, d(i) := \infty, i = 2, \ldots, n$
Repeat n times
$\quad Pulse : \forall \ (i, j) \in A, d(j) := min\{d(j), d(i) + C_{ij}\}$

To track the paths one needs to record the argument of the predecessor of each node for which the label is updated. To prove the algorithm's correctness we will use the following lemma.

Lemma 8.10. *If there are no negative cost cycles in the network $G = (N, A)$, then there exists a (simple) shortest path from s to any node i, that uses at most $n - 1$ arcs.*

Proof. Suppose that G contains no negative cycles. Observe that at most $n - 1$ arcs are required to construct a path from s to any node i. To prove the observation, consider a path, P, from s to i which has more than $n - 1$ arcs and therefore traverses at least one node twice. Such path must contain a cycle a cycle.

$$P = s \to i_1 \to i_2 \to \ldots \to (i_j \to i_k \to \ldots \to i_j) \to i_\ell \to \ldots \to i.$$

Since G has no negative length cycles, the length of P is no less than the length of $\bar{P}$ where

$$\bar{P} = s \to i_1 \to i_2 \to \ldots \to i_j \to i_k \to \ldots \to i_\ell \to \ldots \to i.$$

Thus, we can remove all cycles from P and still retain a shortest path from s to i. Since the final path is simple, and thus acyclic, it must have no more than $n-1$ arcs. □

Theorem 8.11 (Invariant of Bellman-Ford algorithm). *After pulse k, all shortest paths from s of length k (in terms of number of arcs) or less have been identified.*

Proof. Notation : $d_k(j)$ is the label $d(j)$ at the end of pulse k.

The proof is by induction. For $k = 1$, obvious. Assume true for k, and prove for $k + 1$. By contradiction, suppose this is not the case. Then there is a node j for which the shortest path on $k + 1$ or fewer arcs is of cost $d^*(j)$ which is strictly less than $d_{k+1}(j)$,

$$d^*(j) < d_{k+1}(j).$$

Let this path of cost $d^*(j)$ be $(s, i_1, i_2, ..., i_\ell, j)$ which uses $k + 1$ or fewer arcs. By the principle of optimality (every sub-path is a shortest path) $(s, i_1, i_2, ..., i_\ell)$ is a shortest path from s to i_ℓ using k or fewer arcs.

By the inductive assumption the cost of the path $(s, i_1, i_2, ..., i_\ell)$ is $d_k(i_\ell)$, since it is a shortest path containing k or fewer arcs. Hence $d^*(j) = d_k(i_\ell) + c_{i_\ell,j}$.

But, after pulse $k + 1$, $d_{k+1}(j) \leq d_k(i_\ell) + c_{i_\ell,j} = d^*(j)$, which contradicts the assumption that $d^*(j) < d_{k+1}(j)$.

Thus, the $d()$ labels are correct after pulse $k + 1$. □

Note that the shortest paths identified in the theorem are not necessarily *simple*. They may use negative cost cycles.

Theorem 8.12. *If there exists a negative cost cycle, then there exists a node j such that $d_n(j) < d_{n-1}(j)$ where $d_k(i)$ is the label at iteration k.*

Proof. Suppose *not*, then for all nodes, $d_n(j) = d_{n-1}(j)$, since the labels cannot strictly decrease, and also labels never increase.

Let $(i_1 - i_2 - ... - i_k - i_1)$ be a negative cost cycle, and denote $i_{k+1} = i_1$. For all nodes in the cycle $d_n(i_j) = d_{n-1}(i_j)$.

From the optimality conditions, $d_n(i_{r+1}) \leq d_{n-1}(i_r) + C_{i_r,i_{r+1}}$ where $r = 1,...,k$, (for $r = k$ we get $d_n(i_1) \leq d_{n-1}(i_k) + C_{i_k,i_1}$).

We sum up these inequalities and get, $\sum_{j=1}^{k} d_n(i_j) \leq \sum_{j=1}^{k} d_{n-1}(i_j) + \sum_{j=1}^{k} C_{i_j,i_{j+1}}$.

Since $d_n(i_j) = d_{n-1}(i_j)$, the above inequality implies that $0 \leq \sum_{j=1}^{k} C_{i_j,i_{j+1}}$. But this contradicts the assumption that this cycle is of negative length, $\sum_{j=1}^{k} C_{i_{j-1},i_j} < 0$. That is, we get an inequality which says that 0 is less than a negative number which is a contradiction. □

Corollary 8.13. *The Bellman-Ford algorithm identifies a negative cost cycle if one exists.*

By running n pulses of the Bellman-Ford algorithm, we either detect a negative cost cycle (if one exists) or find a shortest path from s to all other nodes i.

Corollary 8.14. *The Bellman-Ford algorithm is correct.*

A consequence of Bellman-Ford algorithm is that finding a negative cost cycle in a graph, or proving that none exists, can be done in polynomial time. Interestingly, finding the *most* negative cost cycle is *NP*-hard (reduction from Hamiltonian Cycle problem), but finding *a* negative cost cycle can be done in polynomial time.

Remark: Minimizing mean cost cycle is also polynomially solvable.

Complexity Analysis
Complexity $= O(mn)$ since each pulse is $O(m)$, and we have n pulses.
Note that it suffices to apply the pulse operations only for arcs (i, j) where the label of i changed in the previous pulse. Thus, if no label changed from the previous iteration, then we are done, even if the iteration number is less than n.

8.9   Floyd-Warshall algorithm for all pairs shortest paths

The Floyd-Warshall algorithm is designed to solve all pairs shortest paths problems for graphs that may contain negative cost arcs. As all algorithms for shortest paths on general graphs, this algorithm will detect negative cost cycles, if exist.

The algorithm maintains a matrix (d_{ik}) such that at iteration j, d_{ik} is the shortest path from i to k using nodes $1, 2, ...j$ as intermediate nodes. After the algorithm terminates, assuming that no negative cost cycle is present, the shortest path from nodes i to k is d_{ik}.

Pseudocode

   Input: An $n \times n$ matrix $[c_{ij}]$
   Output: An $n \times n$ matrix $[d_{ij}]$ is the shortest distance from i to j under $[c_{ij}]$;
         An $n \times n$ matrix $[e_{ij}]$ is an index of an intermediate node in the path from i to j.
   begin
      for all $i \neq j$ **do** $d_{ij} := c_{ij}$;
      for $i = 1, \ldots, n$ **do** $d_{ii} := \infty$;
      for $j = 1, \ldots, n$ **do**
         for $i = 1, \ldots, n, i \neq j$, **do**
            for $k = 1, \ldots, n, k \neq j$, **do**
               $d_{ik} := \min \{d_{ik}, d_{ij} + d_{jk}\}$
               $e_{ik} := \begin{cases} j & \text{if } d_{ik} > d_{ij} + d_{jk} \\ e_{ik} & \text{otherwise} \end{cases}$
   end

The main operation in the algorithm is: *Pulse j : $d_{ik} = \min\{d_{ik}, d_{ij} + d_{jk}\}$. This operation is sometimes referred to as a *triangle operation* (see Figure 19).
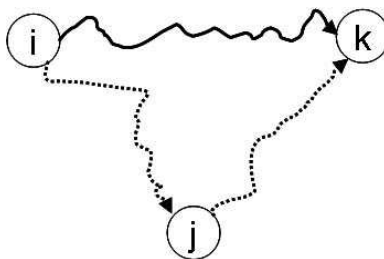
Figure 19: Triangle operation: Is the (dotted) path using node j as intermediate node shorter than the (solid) path without using node j?

Invariant property of Floyd-Warshall Algorithm:
After iteration k, d_{ij} is the shortest path distance from i to j involving a subset of nodes in $\{1, 2, \ldots, k\}$ as intermediate nodes.

Complexity Analysis

At each iteration we consider another node as intermediate node $\rightarrow O(n)$ iterations.
In each iteration we compare n^2 triangles (for n^2 pairs) $\rightarrow O(n^2)$.
Therefore complexity of the Floyd-Warshall algorithm $= O(n^3)$.

See the handout titled *All pairs shortest paths.*

At the initial stage, $d_{ij} = c_{ij}$ if there exist an arc between node i and j
$\qquad\qquad\qquad\quad d_{ij} = \infty$ otherwise
$\qquad\qquad\qquad\quad e_{ij} = 0$
First iteration : $\quad d_{24} \leftarrow \min(d_{24}, d_{21} + d_{14}) = 3$
$\qquad\qquad\qquad\quad e_{24} = 1$
$\qquad\qquad\qquad\quad$ Update distance label
Second iteration : $\quad d_{41} \leftarrow \min(d_{41}, d_{42} + d_{21}) = -2$
$\qquad\qquad\qquad\quad d_{43} \leftarrow \min(d_{43}, d_{42} + d_{23}) = -3$
$\qquad\qquad\qquad\quad d_{44} \leftarrow \min(d_{44}, d_{42} + d_{24}) = -1$
$\qquad\qquad\qquad\quad e_{41} = e_{43} = e_{44} = 2$
$\qquad\qquad\qquad\quad$ No other label changed

Note that we found a negative cost cycle since the diagonal element of matrix $D, d_{44} = -1 \leq 0$. Negative d_{ii} means there exists a negative cost cycle since it simply says that there exists a path of negative length from node i to itself.

8.10 D.B. Johnson's algorithm for all pairs shortest paths

This algorithm takes advantage of the idea that if all arc costs were nonnegative, we can find all pairs shortest paths by solving $O(n)$ single source shortest paths problems using Dijkstra's algorithm. The algorithm converts a network with negative-cost arcs to an equivalent network with nonnegative-cost arcs. After this transformation, we can use Dijkstra's algorithm to solve n SSSP (and get all-pairs shortest paths APSP).

  D.B. Johnson's algorithm $(G = (V, A)$ arc costs c_{ij} for $(i, j) \in A)$
  1. Add a node s to V and arcs from s to each node $j \in V$ with costs $c_{sj} = 0$.
  2. Use Bellman-Ford algorithm to find the shortest paths from node $s \in V$ to every other node. If a negative cost cycle is detected by the algorithm, then terminate at this step. Otherwise, let $d(i)$ be the shortest distance from s to node i.
  3. Convert the arc costs in the original graph to nonnegative costs by setting $c'_{ij} = c_{ij} + d(i) - d(j)$.
  4. Apply Dijkstra's algorithm n times to the graph G with arc costs c'.

  We comment that if there is a node in V so that all nodes are reachable from it, it can serve as node s. Also note that with the setting of the costs of the arcs from s to 0 it implies that all labels $d(i)$ are non-positive. It is however correct to set the costs of these arcs to any quantity other than 0.

The correctness of the algorithm follows from the claims below.

Claim 8.15. *In step 1, adding node s to the graph does not create or remove negative cost cycles.*

Proof. Since s only has outgoing arcs, then it cannot be part of any cycle; similarly the added arcs cannot be part of any cycle. Therefore we have not created any more cycles in G. Finally, if G contained a negative cycle, this cycle remains unchanged after adding s to G. $\qquad\square$

Claim 8.16. *The costs c' satisfy, $c'_{ij} \geq 0 \forall (i,j) \in A$.*

Proof. Since $d(i)$ are shortest path labels. They must satisfy $c_{ij} + d(i) \geq d(j)$ (shortest paths property). This implies that $c'_{ij} = c_{ij} + d(i) - d(j) \geq 0$ $\qquad\qquad\square$

Let $d^P_c(s,t)$ be the distance of path P from s to t with the costs c.

Claim 8.17. *For every pair of nodes s and t, and a path P, $d^P_c(s,t) = d^P_{c'}(s,t) - d(s) + d(t)$. That is, minimizing $d_c(s,t)$ is equivalent to minimizing $d_{c'}(s,t)$.*

Proof. Let the path from s to t, be $P = (s, i_1, i_2, .., i_k, t)$.
The distance along the path satisfies,

$$d^P_{c'}(s,t) = c_{si_1} + d(s) - d(i_1) + c_{i_1 i_2} + d(i_1) - d(i_2) \ldots + c_{si_k} + d(i_k) - d(t) = d^P_c(s,t) + d(s) - d(t).$$

Here all the $d(i)$ terms cancel out except for the first and last terms. Thus the length of all paths from s to t, when using costs c' instead of costs c, change by the constant amount $d(s) - d(t)$.
Therefore, the distances of path P when using c and c' are related as follows:

$$d^P_{c'}(s,t) = d^P_c(s,t) + d(s) - d(t).$$

Thus, $\min_P d^P_{c'}(s,t) = \min_P d^P_c(s,t) +$ constant. Hence minimizing $d_{c'}(s,t)$ is equivalent to minimizing $d_c(s,t)$. $\qquad\qquad\square$

Complexity Analysis
step 1: Add s and n arcs $\rightarrow O(n)$ step 2: Bellman-Ford $\rightarrow O(mn)$.
step 3: Updating c'_{ij} for m arcs $\rightarrow O(m)$.
step 4: Dijkstra's algorithm n times $\rightarrow O(n(m + n \log n))$.
Total complexity: $O(n(m + n \log n))$.

So far this complexity is the best strongly polynomial complexity established for the all-pairs shortest paths problem (on general graphs).

8.11   Matrix multiplication algorithm for all pairs shortest paths

The algorithm

Consider a matrix D with $d_{ij} = c_{ij}$ if there exists an arc from i to j, ∞ otherwise.
Let $d_{ii} = 0$ for all i. Then we define the "dot" operation as
$D^{(2)}_{ij} = (D \odot D^T)_{ij} = \min\{d_{i1} + d_{1j}, \ldots, d_{in} + d_{nj}\}$.

$D^{(2)}_{ij}$ represents the shortest path with 2 or fewer edges from i to j .
$D^{(3)}_{ij} = (D \odot D^{(2)T})_{ij} =$ shortest path with number of edges ≤ 3 from i to j
$\vdots$
$D^{(n)}_{ij} = (D \odot D^{(n-1)T})_{ij} =$ shortest path with number of edges $\leq n$ from i to j

Complexity Analysis
One matrix "multiplication" (dot operation) $\rightarrow O(n^3)$
n multiplications leads to the total complexity $O(n^4)$

By doubling, we can improve the complexity to $O(n^3 log_2 n)$. Doubling is the evaluation of $D_{ij}^{(n)}$ from its previous matrices that are powers of two.

"Doubling": Improving Matrix Multiplication
Improving complexity by Doubling (binary powering):
$D^{(4)} = D^{(2)} \odot D^{(2)T}$
$D^{(8)} = D^{(4)} \odot D^{(4)T}$
$\vdots$
$D^{(2^k)} = D^{(2^{k-1})} \odot D^{(2^{k-1})T}$
Stop once k is such that $2^k > n$ (i.e. need only calculate up to $k = \lceil log_2 n \rceil$).
Then, $D^{(n)}$ can be found by "multiplying" the appropriate matrices (based on the binary representation of n) that have been calculated via doubling. Or we can compute $D^{(2^k)}$, where $k = \lceil log_2 n \rceil$ and it will be equal to $D^{(n)}$ since $D^{(n)} = D^{(n)} \odot D^T$ (because the shortest path may go through at most n nodes).
$\Rightarrow$ Complexity of algorithm with *doubling* is $(n^3 log_2 n)$.

Back-tracking:
To find the paths, construct an $'e'$ matrix as for Floyd-Warshall's algorithm where $e_{ij}^{(k)} = p$ if

$$\min(d_{i1}^{(k-1)} + d_{1j}^{(k-1)}, \ldots, d_{in}^{(k-1)} + d_{nj}^{(k-1)}) = d_{ip}^{(k-1)} + d_{pj}^{(k-1)}.$$

p is then an intermediate node on the shortest path of length $\leq k$ from i to j.

Identifying negative cost cycles:
This is left as a question to class. Suppose there is a negative cost cycle. Then how is it detected if we only have powers of 2 computations of the matrix? The problem is to show that if there is a negative cost cycle then starting at some iteration $\leq n$ the distance labels of some nodes must go down at each iteration. What are these nodes?

Seidel's algorithm for Unit Cost Undirected graph:
(1) Seidel has used the matrix multiplication approach for the all pairs shortest paths problem in an undirected graph. Seidel's algorithm only works when the cost on all the arcs is 1. It uses actual matrix multiplication to find all pairs shortest paths. The running time of Seidel's algorithm is $O(M(n) \log n)$, where $M(n)$ is the time required to multiply two $n \times n$ matrices.
(2) Currently, the most efficient (in terms of asymptotic complexity) algorithm known for matrix multiplication is $O(n^{2.376})$. However, there is a huge constant which is hidden by the complexity notation, so the algorithm is not used in practice.

8.12 Why finding shortest paths in the presence of negative cost cycles is difficult

Suppose we were to apply a minimum cost flow formulation to solve the problem of finding a shortest path from node s to node t in a graph which contains negative cost cycles. We could give a capacity of 1 to the arcs in order to guarantee a bounded solution. This is not enough, however,
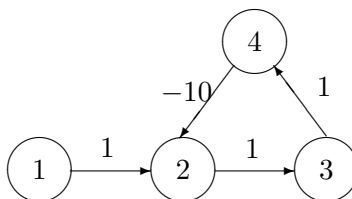
Figure 20: A shortest path problem with a negative cost cycle
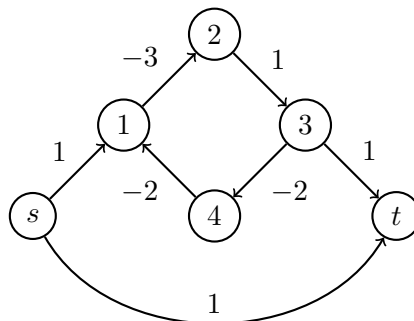
Figure 21: Another shortest path problem with a negative cost cycle

to give us a correct solution. The problem is really in ensuring that we get a *simple* path as a solution.

Consider the network in Figure 20. Observe first that the shortest *simple* path from node 1 to node 2 is the arc $(1, 2)$. If we formulate the problem of finding a shortest path from 1 to 2 as a MCNF problem, we get the following linear program:

$$
\begin{aligned}
\min \quad & x_{12} + x_{23} + x_{34} - 10x_{42} \\
\text{s.t.} \quad & x_{12} && = && 1 \\
& x_{23} - x_{12} - x_{42} && = && -1 \\
& x_{34} - x_{23} && = && 0 \\
& x_{42} - x_{34} && = && 0 \\
& 0 \le x_{ij} \le 1 && && \forall (i, j) \in A.
\end{aligned}
$$

Recall that the interpretation of this formulation is that $x_{ij} = 1$ in an optimal solution means that arc (i, j) is on a shortest path from node 1 to node 2 and the path traced out by the unit of flow from node 1 to node 2 is a shortest path. In this case, however, the optimal solution to this LP is $x_{12} = x_{23} = x_{34} = x_{42} = 1$ which has value -7. Unfortunately, this solution does not correspond to a simple path from node 1 to node 2.

Another example in Figure 21. Here the optimal solution to the respective min cost network flow is the cycle $(1, 2, 3, 4, 1)$ of cost -6 and the arc from s to t of cost 1 for a total of -5. The shortest path however is $(s, 1, 2, 3, t)$ of cost 0.

9 Maximum flow problem

9.1 Introduction

See the handout titled *Travelers Example: Seat Availability*.

The *Maximum Flow* problem is defined on a directed network $G = (V, A)$ with capacities u_{ij} on the arcs, and no costs. In addition two nodes are specified, a source node, s, and sink node, t. The objective is to find the maximum flow possible between the source and sink while satisfying the arc capacities. (We assume for now that all lower bounds are 0.)
Definitions:
- A *feasible flow* f is a flow that satisfied the flow-balance and capacity constraints.
- A *cut* is a partition of the nodes (S, T), such that $S \subseteq V$, $s \in S$, $T = V \setminus S$, $t \in T$.
- *Cut capacity:* $U(S, T) = \sum_{i \in S} \sum_{j \in T} u_{ij}$. (**Important:** note that the arcs in the cut are **only** those that go from S to T.)

Let $|f|$ be the total amount of flow out of the source (equivalently into the sink). It is easy to observe that any feasible flow satisfies

$$|f| \leq U(S, T) \tag{13}$$

for any (S, T) cut. This is true since all flow goes from s to t, and since $s \in S$ and $t \in T$ (by definition of a cut), then all the flow must go through the arcs in the (S, T) cut. Inequality 13 is a special case of the *weak duality theorem* of linear programming.

The following theorem, which we will establish algorithmically, can be viewed as a special case of the strong duality theorem in linear programming.

Theorem 9.1 (Max-Flow Min-Cut). *The value of a maximum flow is equal to the capacity of a cut with minimum capacity among all cuts. That is,*

$$\max |f| = \min U(S, T)$$

Next we introduce the terminology needed to prove the Max-flow/Min-cut duality and to give us an algorithm to find the max flow of any given graph.
Residual graph: The residual graph, $G_f = (V, A_f)$, with respect to a flow f, has the following arcs:
- forward arcs: $(i, j) \in A_f$ if $(i, j) \in A$ and $f_{ij} < u_{ij}$. The residual capacity is $u_{ij}^f = u_{ij} - f_{ij}$. The residual capacity on the forward arc tells us how much we can increase flow on the original arc $(i, j) \in A$.
- reverse arcs: $(j, i) \in A_f$ if $(i, j) \in A$ and $f_{ij} > 0$. The residual capacity is $u_{ji}^f = f_{ij}$. The residual capacity on the reverse arc tells us how much we can decrease flow on the original arc $(i, j) \in A$.

Intuitively, residual graph tells us how much **additional** flow can be sent through the original graph with respect to the given flow. This brings us to the notion of an augmenting path.

In the presence of lower bounds, $\ell_{ij} \leq f_{ij} \leq u_{ij}$, $(i, j) \in A$, the definition of forward arc remains, whereas for reverse arc, $(j, i) \in A_f$ if $(i, j) \in A$ and $f_{ij} > \ell_{ij}$. The residual capacity is $u_{ji}^f = f_{ij} - \ell_{ij}$.

9.2 Linear programming duality and max flow min cut

Given $G = (N, A)$ and capacities u_{ij} for all $(i, j) \in A$:

Consider the formulation of the maximum flow problem on a general graph. Let x_{ij} be a variable denoting the flow on arc (i, j).

$$
\begin{array}{ll}
\text{Max} & x_{ts} \\
\text{subject to} & \sum_i x_{ki} - \sum_j x_{jk} = 0 \quad k \in V \\
& 0 \le x_{ij} \le u_{ij} \quad \forall (i, j) \in A.
\end{array}
$$

For dual variables we let $\{z_{ij}\}$ be the nonnegative dual variables associated with the capacity upper bounds constraints, and $\{\lambda_i\}$ the variables associated with the flow balance constraints. (I also multiply the flow balance constraints by -1, so they represent $\text{Inflow}_i - \text{Outflow}_i = 0$.)

$$
\begin{array}{ll}
\text{Min} & \sum_{(i,j) \in A} u_{ij} z_{ij} \\
\text{subject to} & z_{ij} - \lambda_i + \lambda_j \ge 0 \quad \forall (i, j) \in A \\
& \lambda_s - \lambda_t \ge 1 \\
& z_{ij} \ge 0 \quad \forall (i, j) \in A.
\end{array}
$$

The dual problem has an infinite number of solutions: if (λ^*, z^*) is an optimal solution, then so is $(\lambda^* + C, z^*)$ for any constant C. To avoid that we set $\lambda_t = 0$ (or to any other arbitrary value). Observe now that with this assignment there is an optimal solution with $\lambda_s = 1$ and a partition of the nodes into two sets: $S = \{i \in V | \lambda_i = 1\}$ and $\bar{S} = \{i \in V | \lambda_i = 0\}$.

The complementary slackness condition state that the primal and dual optimal solutions x^*, λ^*, z^* satisfy,

$$
x_{ij}^* \cdot [z_{ij}^* - \lambda_i^* + \lambda_j^*] = 0
$$

$$
[u_{ij} - x_{ij}^*] \cdot z_{ij}^* = 0.
$$

In an optimal solution $z_{ij}^* - \lambda_i^* + \lambda_j^* = 0$ except for the arcs in $(\bar{S}, S)$, so the first set of complementary slackness conditions provides very little information on the primal variables $\{x_{ij}\}$. Namely, that the flows are 0 on the arcs of $(\bar{S}, S)$. As for the second set, $z_{ij}^* = 0$ on all arcs other than the arcs in the cut $(S, \bar{S})$. So we can conclude that the cut arcs are saturated, but derive no further information on the flow on other arcs.

The only method known to date for solving the minimum cut problem requires finding a maximum flow first, and then recovering the cut partition by finding the set of nodes reachable from the source in the residual graph (or reachable from the sink in the reverse residual graph). That set is the source set of the cut, and the recovery can be done in linear time in the number of arcs, $O(m)$.

On the other hand, if we are given a minimum cut, there is no efficient way of recovering the flow values on each arc other than essentially solving from scratch. The only information given by the minimum cut, is the value of the maximum flow and the fact that the arcs on the cut are saturated. Beyond that, the flows have to be calculated with the same complexity as would be required without the knowledge of the minimum cut.

This asymmetry implies that it may be easier to solve the minimum cut problem than to solve the maximum flow problem. Still, no minimum cut algorithm has ever been discovered, in the sense that every so-called s, t minimum cut algorithm known, computes first the maximum flow.

9.3 Applications

9.3.1 Hall's theorem

Hall discovered this theorem about a necessary and sufficient condition for the existence of a perfect matching independently of the max flow min cut theorem. However the max flow min cut theorem can be used as a quick alternative proof to Hall's theorem.

Theorem 9.2 (Hall). *Given a bipartite graph, $B = (V_1 \cup V_2, E)$ where $|V_1| = |V_2| = n$.*
There is a perfect matching in B iff $\forall X \subseteq V_1$, $|N(X)| \geq |X|$,
where $N(X)$ is the set of neighbors of X, $N(X) \subseteq V_2$.

Proof.
($\Rightarrow$) If there is a perfect matching, then for every $X \subseteq V_1$ the set of its neighbors $N(X)$ contains at least all the nodes matched with X. Thus, $|N(X)| \geq |X|$.

($\Leftarrow$) We construct a flow graph by adding: a source s, a sink t, unit-capacity arcs from s to all nodes in V_1, and unit-capacity arcs from all nodes in V_2 to t. Direct all arcs in the original graph G from V_1 to V_2, and set the capacity of these arcs to ∞ (See Figure 22).
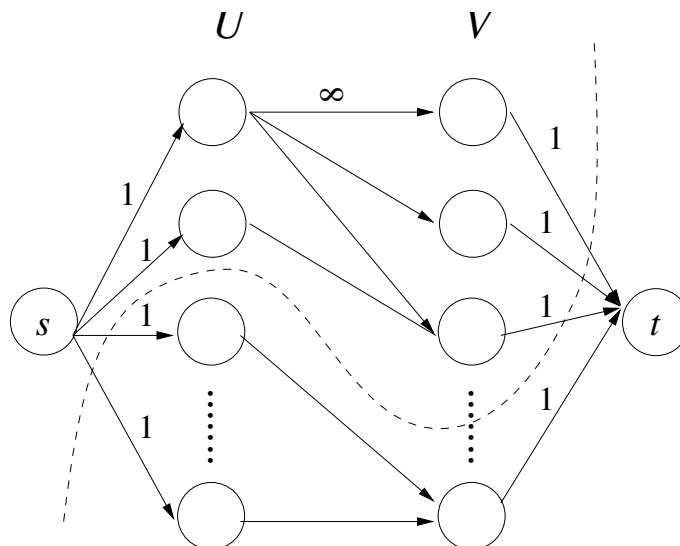
Figure 22: Graph for Hall's Theorem

Assume that $|N(X)| \geq |X|$, $\forall\ X \subseteq U$. We need to show that there exists a perfect matching. Consider a finite cut (S, T) in the flow graph defined. Let $X = V_1 \cap S$. We note that $N(X) \subseteq S \cap V_2$, else the cut is not finite.

Since $|N(X)| \geq |X|$, the capacity of the any (S, T) cut satisfies,

$$U(S, T) \geq |N(X)| + n - |X| \geq n.$$

Also, because of the construction of the network there is a cut, the one that separates s from the rest of the graph, or t from the rest of the graph, of capacity n, therefore the minimum cut capacity is n.

Now, for (S, T) a minimum cut,

$$n = U(S, T) = \text{ max flow value } = \text{ size of matching } \leq n.$$

It follows that all the inequalities must be satisfied as equalities, and the matching size is n. $\square$

Define, **Discrepancy** $= -\min_{X \subseteq V_1} \{|N(X)| - |X|\} = \max_X \{|X| - |N(X)|\}$, $X \subseteq V_1$
Since $U(S, T) = n + |N(X)| - |X|$, where $X = V_1 \cap S$,

$$\min U(S, T) = n + \min\{|N(X)| - |X|\} = n - D.$$

Hence the minimum cut in such a graph with discrepancy D is $n - D$.

Lemma 9.3. *For a bipartite graph with discrepancy D, there is a matching of size $n - D$*

Proof.
Since the minimum cut capacity is $n - D$ the maximum flow is of value $n - D$. Since every matching corresponds to a flow, the maximum matching corresponding to that flow is of size $n - D$. $\square$

9.3.2 The selection problem

Suppose you are going on a hiking trip. You need to decide what to take with you. Many individual items are useless unless you take something else with them. For example, taking a bottle of wine without an opener does not make too much sense. Also, if you plan to eat soup there, for example, you might want to take a set of few different items: canned soup, an opener, a spoon and a plate. The following is the formal definition of the problem.

Given a set of items $J = \{1, \ldots, n\}$, a cost for each item c_j, a collection of sets of items $S_i \subseteq J$ for $i = 1, \ldots, m$, and a benefit for each set b_i. We want to maximize the net benefit (= total benefit - total cost of items) selected.

We now give the mathematical programming formulation:

$$\max \quad \sum_{j=1}^{m} b_i x_i - \sum_{i=1}^{n} c_j y_j$$
$$x_i \leq y_j \quad \text{for } i = 1, \ldots, m, \, \forall j \in S_i$$
$$x_i \in \{0, 1\} \quad \text{for } i = 1, \ldots, m$$
$$y_j \in \{0, 1\} \quad \text{for } j = 1, \ldots, n.$$

Where,
$$x_i = \begin{cases} 1 & \text{if set } j \text{ is selected} \\ 0 & \text{otherwise} \end{cases}$$
$$y_j = \begin{cases} 1 & \text{if item } j \text{ is included} \\ 0 & \text{otherwise} \end{cases}$$

We immediately recognize that the constraint matrix is totally unimodular (each row has one 1 and one -1). Thus when solving the LP relaxation of the problem we will get a solution for the above IP.

Nevertheless, we can do better than solving the LP. In particular we now show how the Selection Problem can be solved by finding a minimum *s-t* cut in an appropriate network.

First we make the following observation. The optimal solution to the selection problem will consist of a collection of sets, $S_j, j \in J$, and the union of their elements ($\bigcup S_j, j \in J$). In particular we will never pick "extra items" since this only adds to our cost. Feasible solutions with this structure are called *selections*.

As per the above discussion, we can limit our search to selections, and we can formulate the selection problem as follows. Find the collection of sets S such that:

$$\max_S \sum_{i \in S} b_i - \sum_{j \in \bigcup_{i \in S} S_i} c_j \tag{14}$$

The following definition will be useful in our later discussion. Given a directed graph $G = (V, A)$, a *closed set* is a set $C \subseteq V$ such that $u \in C$ and $(u, v) \in A \implies v \in C$. That is, a closed set includes all of the successors of every node in C. Note that both $\emptyset$ and V are closed sets.

Now we are ready to show how we can formulate the selection problem as a minimum cut problem. We first create a bipartite graph with sets on one side, items on the other. We add arcs from each set to all of its items. (See figure 23.) Note that a selection in this graph is represented by a collection of set nodes and all of its successors. Indeed there is a one-to-one correspondence between selections and closed sets in this graph.
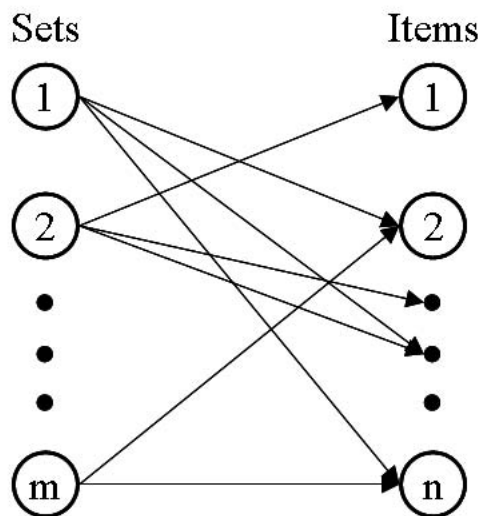
Figure 23: Representing selections as closed sets

Next we transform this graph into a maxflow-mincut graph. We set the capacity of all arcs to infinity. We add a source, a sink, arcs from s to each set i with capacity b_i, and arcs from from each set j to t with capacity c_j. (See figure 24.)

We make the following observations. There is a one-to-one correspondence between finite cuts and selections. Indeed the source set of any finite (S, T) cut is a selection. (If $S_j \in S$ then it must be true that all of its items are also in S—otherwise the cut could not be finite.) Now we are ready to state our main result.

Theorem 9.4. *The source set of a minimum cut (S,T) is an optimal selection.*
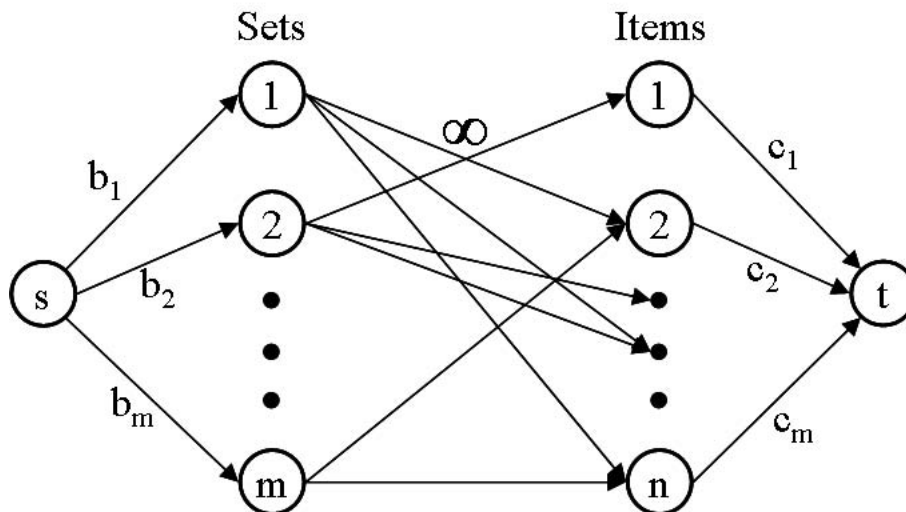
Figure 24: Formulation of selection problem as a min-cut problem

Proof.

$$
\begin{aligned}
\min_{S} U(S,T) &= \min \sum_{i \in T} b_i + \sum_{j \in S} c_j \\
&= \min \sum_{i=1}^{m} b_i - \sum_{i \in S} b_i + \sum_{j \in S} c_j \\
&= B + \min - \sum_{i \in S} b_i + \sum_{j \in S} c_j \\
&= B - \max \sum_{i \in S} b_i - \sum_{j \in S} c_j.
\end{aligned}
$$

Where $B = \sum_{i=1}^{m} b_i$. □

9.3.3 Vertex cover on bipartite graphs

The vertex cover problem is defined as follows: Given an undirected graph $G = (V, E)$ we want to find the set $S \subseteq V$ with minimum cardinality such that every edge in E is adjacent to (at least) one node in S. In the weighted version, each node $i \in V$ has an associated weight $w_i \geq 0$ and we want to minimize the total weight of the set S. Note that, if the graph has negative or zero weights, then we simply include all those nodes in S and remove their neighbors from V.

Let $x_i = \begin{cases} 1 & \text{if } i \in V \text{ is in the cover} \\ 0 & \text{otherwise} \end{cases}$.

The (weighted) vertex cover problem can be formulated as the integer program below.

$$
\min \sum_{i \in V} w_i x_i \tag{$\mathcal{VC}$.0}
$$

$$
\text{s.t. } x_i + x_j \geq 1 \quad \forall \, (i,j) \in E \tag{$\mathcal{VC}$.1}
$$

$$
x_i \in \{0,1\} \quad \forall \, i \in V
$$

The vertex cover problem on bipartite graphs $G = (V_1 \cup V_2, E)$ can be solved in polynomial time. This follows since constraint matrix of the vertex cover problem on bipartite graphs is totally

unimodular. To see this, observe that the constraint matrix can be separated in columns into two parts corresponding to V_1 and V_2 respectively, each row within which contains exactly one 1.

To see this, let
$$x_i = \begin{cases} 1 & \text{if } i \in V_1 \text{ is in the cover} \\ 0 & \text{otherwise} \end{cases},$$
$$y_i = \begin{cases} -1 & \text{if } i \in V_2 \text{ is in the cover} \\ 0 & \text{otherwise} \end{cases}.$$

The integer programming formulation for the vertex cover problem on bipartite graph can then be transformed to the following:

$$\min \sum_{j \in V_1} w_j x_j + \sum_{j \in V_2} (-w_j) y_j \qquad (\mathcal{VC}_B)$$
$$\text{s.t. } x_i - y_j \geq 1 \quad \forall \, (i, j) \in E$$
$$x_i \in \{0, 1\} \quad \forall \, i \in V_1$$
$$y_j \in \{-1, 0\} \quad \forall \, j \in V_2$$

The following s, t-Graph (Figure 25) can be constructed to solve the vertex cover problem on bipartite graph.

1. Given a bipartite graph $G = (V_1 \cup V_2, E)$, set the capacity of all edges in A equal to ∞.

2. Add a source s, a sink t, set A_s of arcs from s to all nodes $i \in V_1$ (with capacity $u_{s,i} = w_i$), and set A_t of arcs from all nodes $j \in V_2$ to t (with capacity $u_{j,t} = w_j$).
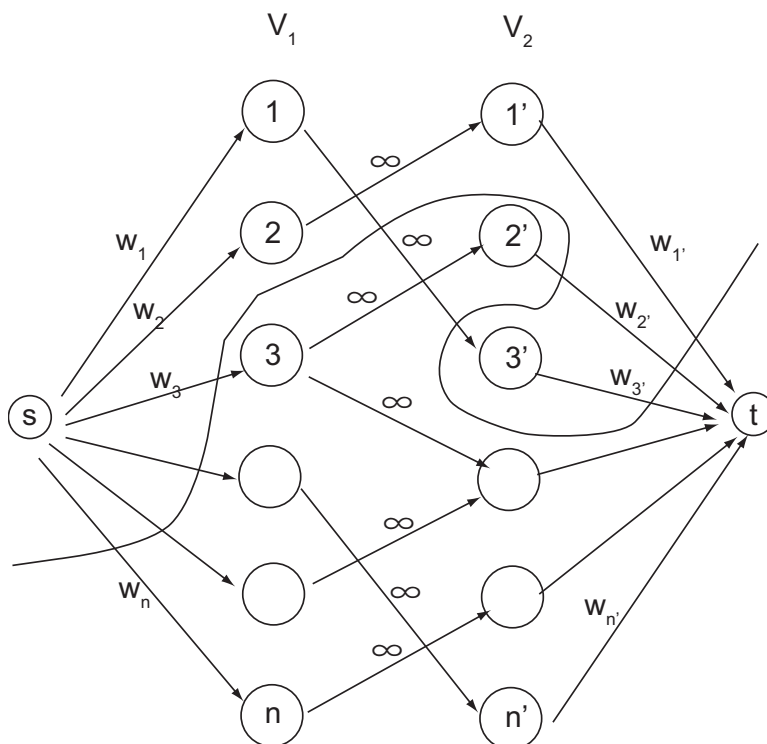
Figure 25: An illustration of the s, t-Graph for the vertex cover problem on bipartite graph and an example of a finite $s - t$ cut.

It is easy to see that if $(s \cup S, t \cup T)$ is a finite $s - t$ cut on $G_{s,t}$, then $(V_1 \cap T) \cup (V_2 \cap S)$ is a vertex cover. Furthermore, if $(s \cup S, t \cup T)$ is an optimal solution to the minimum $s - t$ cut problem

on $G_{s,t}$, then $(V_1 \cap T) \cup (V_2 \cap S)$ is a vertex cover with the minimum weights, where

$$C(s \cup S, t \cup T) = \sum_{i \in V_1 \cap T} w_i + \sum_{j \in V_2 \cap S} w_j = \sum_{j \in VC^*} w_j$$

9.3.4   Forest clearing

The Forest Clearing problem is a real, practical problem. We are given a forest which needs to be selectively cut. The forest is divided into squares where every square has different kind of trees. Therefore, the value (or benefit from cutting) is different for each such square. The goal is to find the most beneficial way of cutting the forest while preserving the regulatory constraints.

Version 1. One of the constraints in forest clearing is preservation of deer habitat. For example, deer like to eat in big open areas, but they like to have trees around the clearing so they can run for cover if a predator comes along. If they don't feel protected, they won't go into the clearing to eat. So, the constraint in this version of Forest Clearing is the following: no two adjacent squares can be cut. Adjacent squares are two squares which share the same side, i.e. adjacent either vertically or horizontally.

Solution. Make a *grid-graph* $G = (V, E)$. For every square of forest make a node v. Assign every node v, weight w, which is equal to the amount of benefit you get when cut the corresponding square. Connect two nodes if their corresponding squares are adjacent. The resulting graph is bipartite. We can color it in a chess-board fashion in two colors, such that no two adjacent nodes are of the same color. Now, the above Forest Clearing problem is equivalent to finding a max-weight independent set in the grid-graph G. In a previous section it was shown that Maximum Independent Set Problem is equivalent to Minimum Vertex Cover Problem. Therefore, we can solve our problem by solving weighted vertex cover problem on G by finding Min-Cut in the corresponding network. This problem is solved in polynomial time.

Version 2. Suppose, deer can see if the diagonal squares are cut. Then the new constraint is the following: no two adjacent vertically, horizontally or diagonally squares can be cut. We can build a grid-graph in a similar fashion as in *Version 1*. However, it will not be bipartite anymore, since it will have odd cycles. So, the above solution would not be applicable in this case. Moreover, the problem of finding Max Weight Independent Set on such a grid-graph with diagonals is proven to be *NP*-complete.

Another variation of Forest Clearing Problem is when the forest itself is not of a rectangular shape. Also, it might have "holes" in it, such as lakes. In this case the problem is also *NP*-complete.
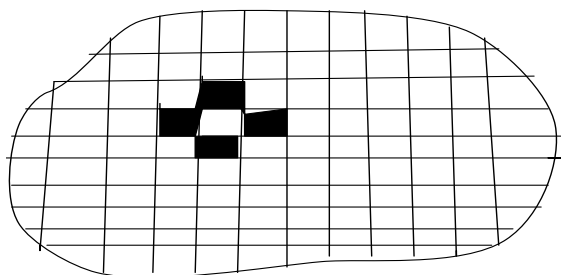
The checkered board pattern permitted to clear

Because there are no odd cycles in the above grid graph of version 1, it is bipartite and the aforementioned approach works. Nevertheless, notice that this method breaks down for graphs

where cells are neighbors also if they are adjacent diagonally. An example of such a graph is given below.
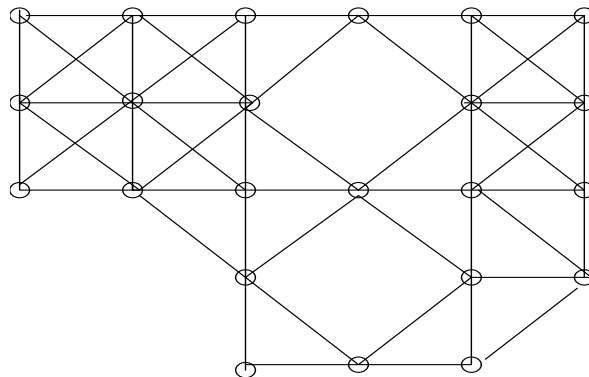
Such graphs are no longer bipartite. In this case, the problem has indeed been proven NP-complete in the context of producing memory chips (next application).

9.3.5 Producing memory chips (VLSI layout)

We are given a silicon wafer of 2G RAM VLSI chips arranged in a grid-like fashion. Some chips are damaged, those will be marked by a little dot on the top and cannot be used. The objective is to make as many 8G chips as possible out of good 2G chips. A valid 8G chip is the one where 4 little chips of 2G form a square, (i.e. they have to be adjacent to each other in a square fashion.) Create a grid-graph $G = (V, E)$ in the following manner: place a node in the middle of every possible 8G chip. (The adjacent 8G chips will overlap.) Similar to *Version 2* of Forest Clearing Problem put an edge between every two adjacent vertically, horizontally or diagonally nodes. Now, to solve the problem we need to find a maximum independent set on G. This problem is proven to be NP-complete.

9.4 Flow Decomposition

In many situations we would like, given a feasible flow, to know how to *recreate* this flow with a bunch of $s - t$ paths and cycles, each of which carries a positive amount of flow. This motivates the following definition.

A *primitive element* is either:
1. A simple path P from s to t with flow δ.
2. A cycle Γ with flow δ.

Theorem 9.5 (Flow Decomposition Theorem). *Any feasible flow f can be decomposed into no more than m primitive elements.*

Proof. The proof is by construction; that is, we will give an algorithm to find the primitive elements. Let $A^+ = \{(i, j) \in A : f_{ij} > 0\}$, and $G^+ = (V, A^+)$.

Pseudocode

While there is an arc in A^+ adjacent to s (i.e. while there is a positive flow from s to t), do:
Begin DFS in G^+ from s, and stop when we either reach t along a simple path P, or found a simple cycle Γ. Let ξ denote the set of arcs of the primitive element found.
Set $\delta = \min_{(i,j) \in \xi}\{f_{ij}\}$.

Record either (P, δ) or (Γ, δ) as a primitive element.
Update the flow in G by setting: $f_{ij} \leftarrow f_{ij} - \delta, \; \forall (i, j) \in \xi$.
Update A^+. Note that the number of arcs in A^+ is reduced by at least one arc.
If no arc in A^+ leaves s, then A^+ must consist of cycles. While $A^+ \neq \emptyset$ do:
Pick any arc $(i, j) \in A^+$ and start DFS from i.
When we come back to a visited node (which is guaranteed by flow conservation), we found another primitive element, a (simple) cycle Γ.
Set $\delta = \min_{(i,j) \in \Gamma} \{f_{ij}\}$.
Record (Γ, δ) as a primitive element.
Update the flow in G by setting: $f_{ij} \leftarrow f_{ij} - \delta, \; \forall (i, j) \in \xi$.
Update A^+. { Note that the number of arcs in A^+ goes down by at least 1.}

The above algorithm is correct since, every time we update the flow, the new flow must still be a feasible flow. Since, each time we find a primitive element we remove at least one arc from A^+, we can find at most m primitive elements. □

An interesting question is the complexity of the algorithm to generate the primitive elements and how it compares to maximum flow. We now show that the overall complexity is $O(mn)$. Assuming the we have for each node the adjacency list - the outgoing arcs. Follow a path from s in a DFS manner, visiting one arc at a time. If a node is repeated (visited twice) then we found a cycle. We trace it back and find its bottleneck capacity and record the primitive element. The bottleneck arcs are removed, and the adjacency lists affected are updated. This entire operation is done in $O(n)$ as we visit at most n nodes.

If a node is not repeated, then after at most n steps we reached the sink t. The path followed is then a primitive path with flow equal to its bottleneck capacity. We record it and update the flows and adjacency lists in $O(n)$ time.

Since there are $O(m)$ primitive elements, then all are found in $O(mn)$. Notice that this is (pretty much) faster than any known maximum flow algorithm. Flow decomposition can be done faster with dynamic trees, in complexity $O(m \log n)$.

9.5   Algorithms

First we introduce some terminology that will be useful in the presentation of the algorithms.
Residual graph: The residual graph, $G_f = (V, A_f)$, with respect to a flow f, has the following arcs:

- forward arcs: $(i, j) \in A_f$ if $(i, j) \in A$ and $f_{ij} < u_{ij}$. The residual capacity is $u^f_{ij} = u_{ij} - f_{ij}$. The residual capacity on the forward arc tells us how much we can increase flow on the original arc $(i, j) \in A$.
- reverse arcs: $(j, i) \in A_f$ if $(i, j) \in A$ and $f_{ij} > 0$. The residual capacity is $u^f_{ji} = f_{ij}$. The residual capacity on the reverse arc tells us how much we can decrease flow on the original arc $(i, j) \in A$.

Intuitively, residual graph tells us how much **additional** flow can be sent through the original graph with respect to the given flow. This brings us to the notion of an augmenting path.

In the presence of lower bounds, $\ell_{ij} \leq f_{ij} \leq u_{ij}, \; (i, j) \in A$, the definition of forward arc remains, whereas for reverse arc, $(j, i) \in A_f$ if $(i, j) \in A$ and $f_{ij} > \ell_{ij}$. The residual capacity is $u^f_{ji} = f_{ij} - \ell_{ij}$.

Augmenting path: An augmenting path is a path from s to t in the residual graph. The *capacity of an augmenting path* is the minimum residual capacity of the arcs on the path – the bottleneck

capacity.

If we can find an augmenting path with capacity δ in the residual graph, we can increase the flow in the original graph by adding δ units of flow on the arcs in the original graph which correspond to forward arcs in the augmenting path and subtracting δ units of flow on the arcs in the original graph which correspond to reverse arcs in the augmenting path.

Note that this operation does not violate the **capacity constraints** since δ is the smallest arc capacity in the augmenting path in the residual graph, which means that we can always add δ units of flow on the arcs in the original graph which correspond to forward arcs in the augmenting path and subtract δ units of flow on the arcs in the original graph which correspond to reverse arcs in the augmenting path without violating capacity constraints.

The **flow balance** constraints are not violated either, since for every node on the augmenting path in the original graph we either increment the flow by δ on one of the incoming arcs and increment the flow by δ on one of the outgoing arcs (this is the case when the incremental flow in the residual graph is along forward arcs) or we increment the flow by δ on one of the incoming arcs and decrease the flow by δ on some other incoming arc (this is the case when the incremental flow in the residual graph comes into the node through the forward arc and leaves the node through the reverse arc) or we decrease the flow on one of the incoming arcs and one of the outgoing arcs in the original graph (which corresponds to sending flow along the reverse arcs in the residual graph).

9.5.1   Ford-Fulkerson algorithm

The above discussion can be summarized in the following algorithm for finding the maximum flow on a give graph. This algorithm is is also known as the **augmenting path algorithm**. Below is a pseudocode-like description.

Pseudocode: (G=(V,A), $s, t \in V$, capacities u_{ij} for $(i,j) \in A$)
f: flow;
G_f: the residual graph with respect to the flow;
P_{st}: a path from s to t;
u_{ij}: capacity of arc from i to j.

Initialize $f = 0$
If $\exists P_{st} \in G_f$ do
    find $\delta = \min_{(i,j) \in P_{st}} u_{ij}$
    $f_{ij} = f_{ij} + \delta \quad \forall (i,j) \in P_{st}$
else stop f is max flow.

Detailed description:
1. Start with a feasible flow (usually $f_{ij} = 0 \; \forall (i,j) \in A$).
2. Construct the residual graph G_f with respect to the flow.
3. Search for augmenting path by doing breadth-first-search from s (we consider nodes to be adjacent if there is a positive capacity arc between them in the residual graph) and seeing whether the set of s-reachable nodes (call it S) contains t.
   If S contains t then there is an augmenting path (since we get from s to t by going through a series of adjacent nodes), and we can then increment the flow along the augmenting path by the value of the smallest arc capacity of all the arcs on the augmenting path.
   We then update the residual graph (by setting the capacities of the forward arcs on the augmenting path to the difference between the current capacities of the forward arcs on the

augmenting path and the value of the flow on the augmenting path and setting the capacities of the reverse arcs on the augmenting path to the sum of the current capacities and the value of the flow on the augmenting path) and go back to the beginning of step 3.

If S does not contain t then the flow is maximum.

To establish correctness of the augmenting path algorithm we prove the following theorem which is actually stronger than the the max-flow min-cut theorem.

Reminder: f_{ij} is the flow from i to j, $|f| = \sum_{(s,i) \in A} f_{si} = \sum_{(i,t) \in A} f_{it} = \sum_{u \in S, v \in T} f_{uv}$ for any cut (S, T).

Theorem 9.6 (Augmenting Path Theorem). *(generalization of the max-flow min-cut theorem) The following conditions are equivalent:*

1. *f is a maximum flow.*
2. *There is no augmenting path for f.*
3. *$|f| = U(S, T)$ for some cut (S, T).*

Proof.
$(1 \Rightarrow 2)$ If $\exists$ augmenting path p, then we can strictly increase the flow along p; this contradicts that the flow was maximum.

$(2 \Rightarrow 3)$ Let G_f be the residual graph w.r.t. f. Let S be a set of nodes reachable in G_f from s. Let $T = V \setminus S$. Since $s \in S$ and $t \in T$ then (S, T) is a cut. For $v \in S, w \in T$, we have the following implications:

$$(v, w) \notin G_f \quad \Rightarrow \quad f_{vw} = u_{vw} \text{ and } f_{wv} = 0$$
$$\Rightarrow \quad |f| = \sum_{v \in S, w \in T} f_{vw} - \sum_{w \in T, v \in S} f_{wv} = \sum_{v \in S, w \in T} u_{vw} = U(S, T)$$

$(3 \Rightarrow 1)$ Since $|f| \le U(S, T)$ for any (S, T) cut, then $|f| = U(S, T) \implies f$ is a maximum flow. $\square$

Note that the equivalence of conditions 1 and 3 gives the max-flow min-cut theorem.

Given the max flow (with all the flow values), a min cut can be found in by looking at the residual network. The set S, consists of s and the all nodes that s can reach in the final residual network and the set $\bar{S}$ consists of all the other nodes. Since s can't reach any of the nodes in $\bar{S}$, it follows that any arc going from a node in S to a node in $\bar{S}$ in the original network must be saturated which implies this is a minimum cut. This cut is known as the minimal source set minimum cut. Another way to find a minimum cut is to let $\bar{S}$ be t and all the nodes that can reach t in the final residual network. This a *maximal source set minimum cut* which is different from the minimal source set minimum cut when the minimum cut is not unique.

While finding a minimum cut given a maximum flow can be done in linear time, $O(m)$, we have yet to discover an efficient way of finding a maximum flow given a list of the edges in a minimum cut other than simply solving the maximum flow problem from scratch. Also, we have yet to discover a way of finding a minimum cut without first finding a maximum flow. Since the minimum cut problem is asking for less information than the maximum flow problem, it seems as if we should be able to solve the former problem more efficiently than the later one. More on this in Section 9.2.

In a previous lecture we already presented Ford-Fulkerson algorithm and proved its correctness. In this section we will analyze its complexity. For completeness purposes we give a sketch of the algorithm.

Ford-Fulkerson Algorithm
Step 0: $f = 0$
Step 1: Construct G_f (Residual graph with respect to flow f)
Step 2: Find an augmenting path from s to t in G_f
        Let path capacity be δ
        Augment f by δ along this path
        Go to step 1
If there does not exist any path
        Stop with f as a maximum flow.

Theorem 9.7. *If all capacities are integer and bounded by a finite number U, then the augmenting path algorithm finds a maximum flow in time $O(mnU)$, where $U = \max_{(v,w)\in A} u_{vw}$.*

Proof. Since the capacities are integer, the value of the flow goes up at each iteration by at least one unit.

Since the capacity of the cut $(s, N \setminus \{s\})$ is at most nU, the value of the maximum flow is at most nU.

From the two above observations it follows that there are at most $O(nU)$ iterations. Since each iteration takes $O(m)$ time–find a path and augment flow, then the total complexity is $O(nmU)$. $\square$

The above result also applies for rational capacities, as we can scale to convert the capacities to integer values.

Drawbacks of Ford-Fulkerson algorithm:
1. The running time is not polynomial. The complexity is exponential (pseudo-polynomial) since it depends on U.
2. In Ford-Fulkerson's algorithm any augmenting path can be selected. In the graph of Figure 26, the algorithm could take 4000 iterations for a problem with maximum capacity of 2000. This is because the augmenting path is not "chosen wisely" (see Indiana Jones and the Last Crusade). We will see that a wise choice of the augmenting path leads to polynomial time algorithms.
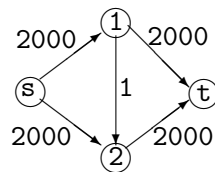
Figure 26: Graph leading to long running time for Ford-Fulkerson algorithm.

3. Finally, for irrational capacities, this algorithm may converge to the wrong value (see Papadimitriou and Steiglitz p.126-128)

Improvements for enhancing the performance of Ford-Fulkerson algorithm include:
1. Augment along maximum capacity augmenting paths.
2. Using the concept of capacity scaling to cope with the non-polynomial running time.
3. Augmenting along the shortest (in number of edges) path.
Each alternative leads to different types of max-flow algorithms discussed next.

9.5.2 Maximum capacity augmenting path algorithm

The maximum capacity augmenting path algorithm is Ford-Fulkerson algorithm where the selection of the augmenting path is the one of maximum capacity.
Pseudocode:
f: flow;
G_f: the residual graph with respect to the flow;
P_{st}: a path from s to t;
u_{ij}: capacity of arc from i to j.

Initialize $f = 0$
If t is reachable from s in G_f, do
    find maximum capacity augmenting path in G_f (see algorithm below) P_{st} of capacity $\delta = \min_{(i,j) \in P_{st}} u_{ij}^f$.
    { update} $f_{ij} = f_{ij} + \delta \quad \forall (i,j) \in P_{st}$
else stop f is max flow.

We now address the issue of how to find a maximum capacity path in a graph. This can be done with an adaptation of Dijkstra's algorithm. Let the label of a node be ℓ_j, the max capacity of a path from s that includes only permanent nodes (of P) as intermediate nodes. At each iteration we add to the set of permanent nodes a node with *largest* label reachable from the permanent set.

Maximum Capacity Path Algorithm
Initialize: $P \leftarrow \{s\}$, $T \leftarrow V \setminus \{s\}$, $\ell_s = \infty$, $\ell_j = 0$ for $j \in V \setminus \{s\}$, $i^* = s$.
While T is not empty do
        update nodes labels for all $j \in N^+(i^*) \cap T$,
$$\ell_j \leftarrow \max\{\ell_j, \min\{\ell_{i^*}, c_{i^*j}\}\}.$$
        select $i^* \in T$ of maximum label ℓ_{i^*};
        add i^* to the set of permanent nodes and remove from temporary
        $P \leftarrow P \cup \{i^*\}$, $T \leftarrow T \setminus \{i^*\}$.
        If $T \neq \emptyset$ go to update.
End Do

The complexity of finding the max capacity augmenting path using Dijkstra's as above with Fibonacci heaps is $O(m + n \log n)$.

The complexity of the max capacity augmenting paths algorithm

Each iteration of the max capacity augmenting path algorithm consists of finding a maximum capacity path in the residual graph. This is done with the Maximum Capacity Path Algorithm. We now address the required number of iterations.
    Suppose that the value of the maximum flow, f^*, is v^*. We are at an iteration with flow f of value v. Then the maximum flow value in G_f is $v^* - v$. From the flow decomposition theorem, there are $k \leq m$ paths from s to t with sum of flows equal to $v^* - v$. $\sum_{i=1,k} \delta_i = v^* - v$. Therefore, the max capacity s, t path has capacity at most the average path capacity:
$\Rightarrow \exists i | \delta_i \geq \frac{(v^*-v)}{k} \geq \frac{(v^*-v)}{m}$.
$\Rightarrow$ remaining flow (after augmenting along a maximum capacity path) is $\leq \frac{m-1}{m}(v^* - v)$.
The remaining flow after repeating this process for q iterations is $(\frac{m-1}{m})^q v^*$. The max flow is found when $(\frac{m-1}{m})^q v^* \leq 1$, since flows must be integers. Thus it suffices to have $q \geq \frac{log(v^*)}{\log(\frac{m}{m-1})}$.

Now $\log(\frac{m}{m-1}) \approx \frac{1}{m}$. Thus the overall of the max capacity augmenting path algorithm is $O(m(m + n \log n) \log(nU))$.

Why is the largest capacity augmenting path not necessarily a primitive path? A primitive path never travels backwards along an arc, yet an augmenting path may contain backward arcs. Thus, knowing the flow in advance is a significant advantage (not surprising).

9.5.3   Capacity scaling algorithm

We note that the Ford-Fulkerson algorithm is good when our capacities are small. Most algorithms with this type of properties can be transformed to polynomial time algorithms using a *scaling* strategy.

The idea in this algorithm is to construct a series of max-flow problems such that: the number of augmentations on each problem is small; the maximum flow on the previous problem to find a feasible flow to the next problem with a small amount of work; the last problem gives the optimal solution to our original max-flow problem.

In particular, at each iteration k, we find a maximum flow in the network P_k with capacities restricted to the k most significant digits (in the binary representation). (We need $p = \lfloor \log_2 U \rfloor + 1$ digits to represent the capacities.) Thus, at iteration k, we will consider the network where the capacities are given by the following equation:

$$u_{ij}^{(k)} = \left\lfloor \frac{u_{ij}}{2^{p-k}} \right\rfloor .$$

Note that the capacities of the kth iteration can be obtained as follows:

$$u_{ij}^{(k)} = 2u_{ij}^{(k-1)} + \text{next significant digit},$$

where the next significant digit can be 0 or 1. It follows, from the above equation, that doubling the max-flow from the previous iteration is a feasible flow for the current iteration.

Finally, note that the residual flow in the current iteration can't be more than m. This is true since in the previous iteration we had a max-flow and a corresponding min-cut. The residual capacity at each arc of the min-cut from the previous scaling iteration can grow at most by one unit as these arcs were saturated in that previous iteration. Therefore the max-flow in the residual graph of the current iteration can be at most m units.

The preceding discussion is summarized in the following algorithm.

Capacity scaling algorithm
$f_{ij} := 0$;
Apply the augmenting path algorithm to P_0 and resulting in maximum flow vector f_0.
For $k := 1$ to p Do
    Multiply all residual capacities and flows f_{k-1} of residual graph $G_{f_{k-1}}$ from previous iteration by 2;
    let this initial flow be $f = 2f_{k-1}$.
    Add 1 to all capacities of arcs that have 1 in the (k+1)st position
        (of the binary representation of their original capacity);
    Apply the augmenting path algorithm starting with the current flow f to generate a maximum flow f_k;
End Do

Theorem 9.8. *The capacity scaling algorithm runs in $O(m^2 \log_2 U)$ time.*

Proof. In iteration k consider the flows on the arcs on the minimum cut corresponding to the maximum flow f_{k-1}. Since the cut arcs were saturated and the capacities can increase by at most
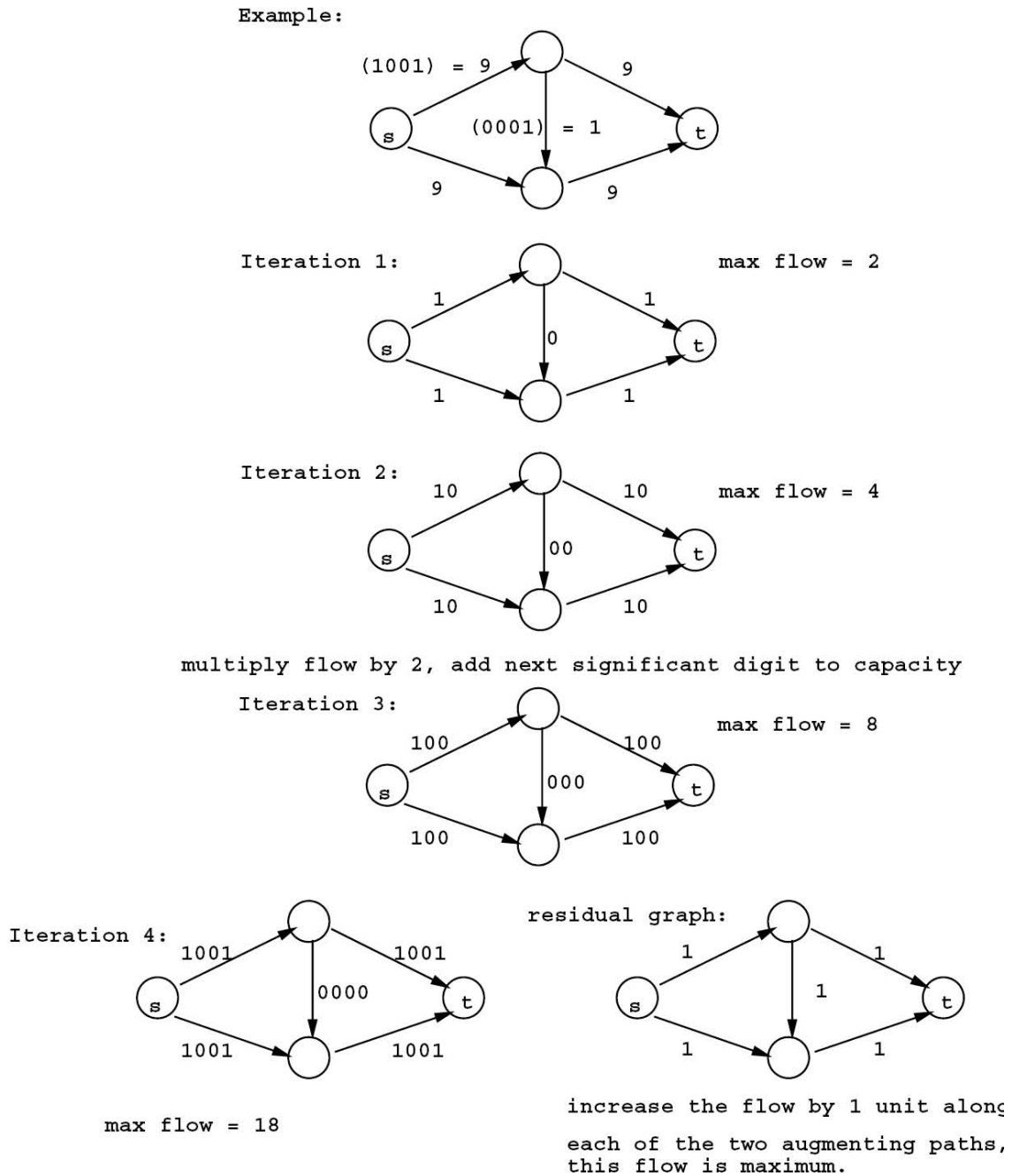
Figure 27: Example Illustrating the Capacity Scaling Algorithm

one each, then the amount of residual max flow in P_k is bounded by the number of arcs in the cut which is $\leq m$. So the number of augmentations at each iteration is $O(m)$.

The complexity of finding an augmenting path is $O(m)$.

The total number of iterations is $O(\log_2 U)$.

Thus the total complexity is $O(m^2 \log_2 U)$. □

Notice that this algorithm is polynomial, but still not strongly polynomial. Also it cannot handle real capacities.

We illustrate the capacity scaling algorithm in Figure 27.

9.5.4 Dinic's algorithm for maximum flow

Dinic's algorithm is also known as the Blocking Flow Algorithm, or shortest augmenting paths algorithm. In order to describe this algorithm we need the following definitions.

Layered Network: Given a flow, f, and the corresponding residual graph, $G_f = (V, A_f)$, a *layered network $AN(f)$* (AN stands for Auxiliary Network) is constructed as follows:

Using breadth-first-search in G_f, we construct a tree a rooted at s. Each node in V gets a distance label which is its distance in the BFS tree from s. Thus, the nodes that have an incoming arc from s get assigned a label of 1 and nodes at the next level of the tree get a label of 2, etc. Nodes with the same label are said to be at the same layer. Let the distance label of t be $k = d(t)$ then all nodes in layer $\geq k$ are removed from the layered network and all arcs are removed except those which go from a node at one layer to a node at the next consecutive layer.

Note that this layered network comprises all of the shortest paths from s to t in G_f.

Blocking flow: A flow is called blocking flow for network G' if every s, t path intersects with at least one saturated arc. Note that a blocking flow is not necessarily a maximum flow! For example, in the network in Figure 26 the flow of 1 unit from s to 1 to 2 to t is a blocking flow, but obviously not maximum (the maximum flow is of value 2000).

Stage k: We call stage k the augmentations along paths in the layered network where $k = d(t)$ (a network that includes k layers).

Now we are ready to give the basic implementation of Dinic's algorithm.

Dinic's algorithm

$f_{ij} = 0 \ \forall (i, j) \in A$

Use BFS to construct the first layered network $AN(f)$.

While $d(t) < \infty$ do:

Find a blocking flow f' in $AN(f)$.

Construct G_f and its respective $AN(f)$.

End While.

Find a blocking flow (naive implementation)

While DFS finds a path P from s to t do:

Augment the flow along P.

Remove saturated arcs in $AN(f)$.

Cross out nodes with zero indegree or outdegree equal to zero.

End While.

Theorem 9.9. *Dinic's algorithm solves correctly the max-flow problem.*

Proof. The algorithm finishes when s and t are disconnected in residual graph and so there is no augmenting path. Thus by the augmenting path theorem the flow found by Dinic's algorithm is a maximum flow. □

Complexity Analysis

We first argue that, after finding the blocking flow at the k-layered network, the shortest path in the (new) residual graph is of length $\geq k + 1$ (i.e. the level of t must strictly increase from one

stage to the next one). This is true since all paths of length k intersect at least one saturated arc. Therefore, all the paths from s to t in the next layered network must have at least $k+1$ arcs (all paths with length $\leq k$ are already blocked). Thus, Dinic's algorithm has at most n stages (one stage for each layered network.

To find the blocking flow in stage k we do the following work: 1) Each DFS and update takes $O(k)$ time (Note that by definition of $AN(f)$ we can't have cycles.), and 2) we find the blocking flow in at most m updates (since each time we remove at least one arc from the layered network).

We conclude that Dinic's algorithm complexity is $O(\sum_1^n mk) = O(mn^2)$.

Now we will show that we can improve this complexity by finding more efficiently the blocking flow. In order to show this, we need to define one more concept.

Throughput: The *throughput* of a node is the minimum of the sum of incoming arcs capacities and the outgoing arcs capacity (that is, $thru(v) \equiv \min \left\{ \sum_{(i,v) \in A^f} u_{i,v}^f, \sum_{(v,j) \in A^f} u_{v,j}^f \right\}$). The throughput of a node is the largest amount of flow that could possibly be routed through the node.

The idea upon which the improved procedure to find a blocking flow is based is the following. We find the node v with the minimum throughput $thru(v)$. We know that at this node we can "pull" $thru(v)$ units of flow from s, and "push" $thru(v)$ units of flow to t. After performing this pulling and pushing, we can remove at least one node from the layered network.

Find a blocking flow (improved)

    While there is a node $v \neq s, t$ with $thru(v) > 0$ in $AN(f)$.

        Let v be the node in $AN(f)$ for which $thru(v)$ is the smallest; Set $\delta \leftarrow thru(v)$.

        Push δ from v to t.

        Pull δ from s to v.

        Delete (from $AN(f)$) all saturated arcs, v, and all arcs incident to v;

    End While

Push δ from v to t

    $Q \leftarrow \{v\}$; $excess(v) \leftarrow \delta$

    While $Q \neq \emptyset$

        $i \leftarrow$ first node in Q (remove i from Q)

        For each $j \in A^+(i)$. (there is an arc (i,j)), and while $excess(i) > 0$

            $\Delta \leftarrow \min \left\{ u_{ij}^f, excess(i) \right\}$

            $excess(i) \leftarrow excess(i) - \Delta$

            $excess(j) \leftarrow excess(j) + \Delta$

            $thru(j) \leftarrow thru(j) - \Delta$

            $f_{ij} \leftarrow f_{ij} + \Delta$

            if $j \notin Q$ add j to end of Q

        End For

    End While

The "Pull" procedure has a similar implementation.

Theorem 9.10. *Dinic's algorithm solves the max-flow problem for a network $G = (V, A)$ in $O(n^3)$ arithmetic operations.*

Proof. As before, the algorithm has $O(n)$ stages.

At each stage we need to create the layered network with BFS. This takes $O(m)$ complexity.

The operations at each stage can be either a saturating push along an arc or an unsaturating push along an arc. So we can write number of operations as sum of saturating and unsaturating pull and push steps, $N = N_s + N_u$. We first note that once an arc is saturated, it is deleted from graph, therefore $N_s = O(m)$. However we may have many unsaturating steps for the same arc.

The key observation is that we have at most n executions of the push and pull procedures (along a path) at each stage. This is so because each such execution results in the deletion of the node v with the lowest throughput where this execution started. Furthermore, in each execution of the push and pull procedures, we have at most n unsaturated pushes – one for each node along the path. Thus, we have at each stage at most $O(n^2)$ unsaturated pushes and $N = N_s + N_u = O(m) + O(n^2) = O(n^2)$. Therefore, the complexity per stage is $O(m + n^2) = O(n^2)$.

We conclude that the total complexity is thus $O(n^3)$. $\qquad\square$

Dinic's algorithm for unit capacity networks

Definition 9.11. *A unit capacity network is an $s - t$ network, where all arc capacities are equal to 1.*

Some problems that can be solved by finding a maximum flow in unit capacity networks are the following.
- Maximum Bipartite matching,
- Maximum number of edge disjoint Paths.

Lemma 9.12. *In a unit capacity network with distance from s to t greater or equal to ℓ the maximum flow value $|\hat{f}|$ satisfies $\sqrt{|\hat{f}|} \leq 2\frac{|V|}{\ell}$.*

Proof. Construct a layered network with V_i the set of the nodes in Layer i (See Figure 28). And let $S_i = V_0 \cup V_1 \cup \cdots \cup V_i$
$1 \leq i \leq \ell - 1$.
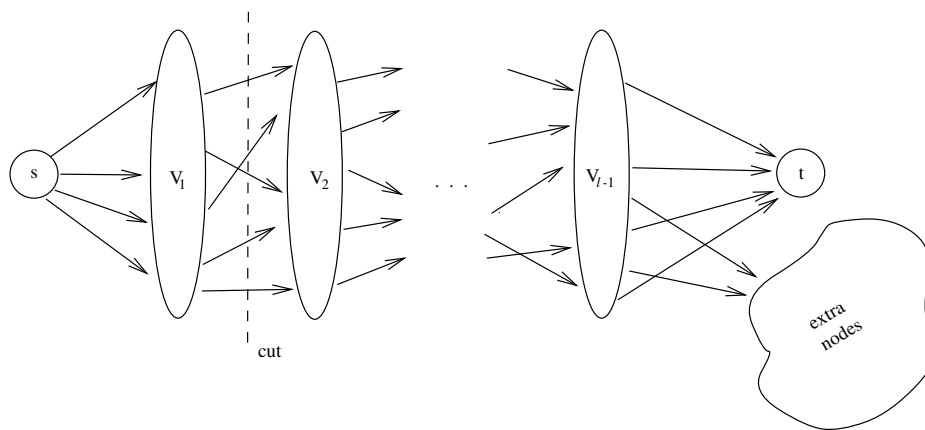
Figure 28: The layered network, distance from s to t is greater or equal to ℓ

Then $(S_i, \bar{S}_i)$ is a s-t cut. Using the fact that the maximum flow value is less than the capacity of any cut,

$$|\hat{f}| \le |V_i||V_{i+1}| \Rightarrow \text{ either } |V_i| \ge \sqrt{|\hat{f}|} \text{ or } |V_{i+1}| \ge \sqrt{\hat{f}}$$

The first inequality follows since the maximum possible capacity of a cut is equal to $|V_i||V_{i+1}|$. Since each arc has a capacity of one, and the maximum of number of arcs from V_i to V_{i+1} is equal to $|V_i||V_{i+1}|$ (all the combinations of the set of nodes V_i and V_{i+1}).

So at least $\frac{\ell}{2}$ of the layers have at least $\sqrt{|\hat{f}|}$ nodes each
(consider the pairs, $V_0 V_1, \ \ldots, \ V_2 V_3, \ \ldots$).

$$\frac{\ell}{2}\sqrt{|\hat{f}|} \le \sum_{i=1}^{\ell} |V_i| \le |V| \Rightarrow \sqrt{|\hat{f}|} \le \frac{2|V|}{\ell}$$

$\square$

Claim 9.13. *Dinic's Algorithm, applied to the unit capacity network requires at most $O(n^{2/3})$ stages.*

Proof. Notice that in this case, it is impossible to have non-saturated arc processing, because of the unit capacities. It follows that the total computation per stage is no more than $O(m)$. If Max flow $\le 2n^{2/3}$ then done. (Each stage increments the flow by at least one unit.)

If Max flow $> 2n^{2/3}$, run the algorithm for $2n^{2/3}$ stages. The distance labels from s to t will be increased by at least 1 for each stage. Let the Max flow in the residual network (after $2n^{2/3}$ stages) be g. The shortest path from s to t in this residual network satisfies $\ell \ge 2n^{2/3}$.

Apply lemma 9.12 to this residual network.

$$\sqrt{|g|} \le \frac{2|V|}{2n^{2/3}} = n^{1/3}, |V| = n \Rightarrow |g| \le n^{2/3}$$

It follows that no more than $n^{2/3}$ additional stages are required. Total number of stages $\le 3n^{2/3}$. $\square$

Complexity Analysis
Any arc processed is saturated (since this is a unit capacity network). Hence $O(m)$ work per stage. There are $O(n^{2/3})$ stages. This yields an overall complexity $O(mn^{2/3})$.
Remark: A. Goldberg and S. Rao [GR98] devised a new algorithm for maximum flow in general networks that uses ideas of the unit capacity as subroutine. The overall running time of their algorithm is $O(\min\{n^{2/3}, m^{1/2}\}m \log(n^2/m) \log U)$ for U the largest capacity in the network.

Dinic's Algorithm for Simple Networks
In a simple network each node has either exactly one incoming arc of capacity one or exactly one outgoing arc of capacity one (See Figure 29 b). Bipartite matching is an example of a problem that can be formulated as a maximum flow problem on a simple network.

For simple networks, Dinic's Algorithm works more efficiently than it does for unit capacity networks, since the throughput of each node is one.

Lemma 9.14. *In a simple network, with distance from s to t greater than or equal to ℓ, the max flow, $\hat{f}$ satisfies $|\hat{f}| \le \frac{|V|}{(\ell-1)}$*
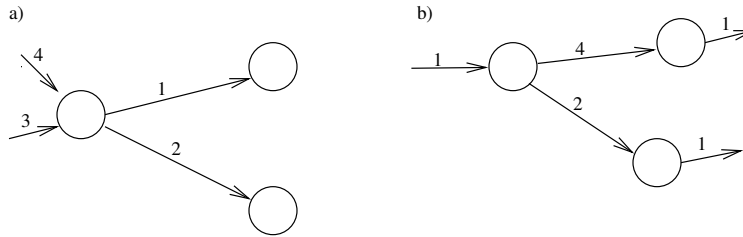
Figure 29: a) not a simple network b) a simple network

Proof. Consider layers $V_0, V_1 \ldots$ and the cut $(S_i, \bar{S}_i)$, where, $S_i = V_0 \cup \cdots \cup V_i$. Since for $i = 1 \ldots \ell - 1$ the throughput of each node for each layer V_i is 1,

$$|\hat{f}| \leq |V_i| \text{ i} = 1 \ldots \ell\text{-}1$$

$$\text{Hence, } |\hat{f}| \leq \min_{i=1 \ldots \ell-1} |V_i|$$

$$\sum_{i=1}^{\ell-1} |V_i| \leq |V| \rightarrow \min_{i=1 \ldots \ell-1} |V_i| \leq \frac{|V|}{(\ell-1)}$$

$\square$

Claim 9.15. *Applying Dinic's Algorithm to a simple network, the number of stages required is* $O(\sqrt{n})$.

Proof. If the max flow $\leq \sqrt{n}$ done. Otherwise, run Dinic's algorithm for $\sqrt{n}$ stages. $\ell \geq \sqrt{n} + 1$ in that residual network, since each stage increments the flow by at least one unit and increases the distance label from the source to the sink by at least one at each stage. In the residual network, the max flow g satisfies $|g| \leq \frac{|V|}{(\sqrt{n})} \leq \sqrt{n}$. So, we need at most $\sqrt{n}$ additional stages. Thus the total number of the stages is $O(\sqrt{n})$ $\square$

Complexity Analysis
Recall we are considering of simple networks. All push/pull operations are saturating a node since there is only one incoming arc or outgoing arc with capacity 1 each node v. The processing thus makes the throughput of node v be 0.

Note that even though we only have to check $O(n)$ nodes, we still have to update the layered network which requires $O(m)$ work. Hence the work per stage is $O(m)$.

Thus, the complexity of the algorithm is $O(m\sqrt{n})$

References

[Din1]   E. A. Dinic. Algorithm for Solution of a Problem of Maximum Flows in Networks with Power Estimation. *Soviet Math. Dokl.*, 11, 1277-1280, 1970.

[EK1]   J. Edmonds and R. M. Karp. Theoretical Improvements in Algorithmic Efficiency for Network Flow Problems. *J. of ACM*, 19, 248-264, 1972.

[FF1]   L. R. Ford, Jr. and D. R. Fulkerson. Maximal Flow through a Network. *Canad. J. Math.*, 8, 399-404, 1956.

[GR98]   A. V. Goldberg and S. Rao. Beyond the flow decomposition barrier. *Journal of the ACM* 45, 783–797, 1998.

[GT1]   A. V. Goldberg and R. E. Tarjan. A new approach to the maximum flow problem. *J. of ACM*, 35, 921-940, 1988.

10 Goldberg's Algorithm - the Push/Relabel Algorithm

10.1 Overview

All algorithms discuss till now were "primal" algorithms, that is, they were all based on augmenting feasible flows. The pus/relabel algorithm for the maximum flow problem is basd on using "preflows". This algorithm was originally developed by Goldberg and is known as the *Preflow Algorithm*. A generic algorithm and a particular variant (*lift-to-front*) will be presented.

Goldberg's algorithm can be contrasted with the algorithms we have learned so far in following ways:

- As opposed to working on feasible flows at each iteration of the algorithm, preflow algorithm works on preflows: the flows through the arcs need not satisfy the flow balance constraints.

- There is never an augmenting path in the residual network. The preflow is possibly infeasible, but super-optimal.

Definitions

Preflow: Preflow $\vec{f}$ in a network is defined as the one which

1. Satisfies the capacity constraints (i.e., $0 \le f_{ij} \le U_{ij}, \;\; \forall (i,j)$).
2. The incoming flow at any node except the source node must be greater than or equal to the outgoing flow, i.e.,
$$\Sigma_j f_{ji} - \Sigma_k f_{ik} \ge 0$$

Excess: Excess of a node is defined w.r.t. a flow and is given by

$$e_f(v) = \text{ incoming flow}(v) - \text{ outgoing flow}(v).$$

Active Node: A node $v \ne t$ is called *active* if $e_f(v) > 0$.

Distance Labels: Each node v is assigned a label. The labeling is such that it satisfies the following inequality
$$d(i) \le d(j) + 1 \;\; \forall (i,j) \in A_f.$$

Admissible Arc An arc (u,v) is called *admissible* if $(u,v) \in A_f$ and $d(u) > d(v)$. Notice that together with the validity of the distance labels this implies that $d(u) = d(v) + 1$.

Lemma 10.1. *The admissible arcs form a directed acyclic graph (DAG).*

Proof: For contradiction suppose the cycle is given as $(i_1, i_2, ..., i_k, i_1)$. From the definition of an admissible arc, $d(i_1) > d(i_2) > \ldots > d(i_k) > d(i_1)$, from which the contradiction follows. $\square$

Lemma 10.2. *The distance label of node i, $d(i)$, is a lower bound on the shortest path distance in A_f from i to t, if such path exists.*

Proof: Let the shortest path from i to t consist of k arcs: $(i, i_1, ..., i_{k-1}, t)$. From the validity of the distance labels,

$$d(i) \leq d(i_1) + 1 \leq d(i_2) + 2 \leq \ldots \leq d(t) + k = k.$$

$\square$

Corollary 10.3. *If the distance label of a node i, satisfies $d(i) \geq n$, then there is no path in A_f from i to t. That is, t is not reachable from i.*

10.2 The Generic Algorithm

```
begin
        /* Preprocessing steps */
        f = 0
        d(s) = n, d(i) = 0,  ∀i ≠ s
        ∀(s, j) ∈ E  f_sj = u_sj
        e_f(j) = u_sj
        while network contains an active node do
                begin
                        /* Push step */
                        select an active node (v)
                        while ∃ admissible arcs (v, w)
                          then
                            Let δ = min{e_f(v), U_f(v, w)}, where (v, w) is admissible
                            Send δ units of flow from v to w.
                          else
                        /* Relabel step (all (v, w) are inadmissible) */
                        set d(v) = min_{(v,w)∈A_f}{d(w) + 1}
                end
end
```

Now we prove the following lemmas about this algorithm which will establish its correctness.

Lemma 10.4. *Relabeling step maintains the validity of distance labels.*

Proof: Relabeling is applied to node i with $d(i) \leq d(j) \ \forall (i, j) \in A_f$. Suppose the new label is

$$d'(i) = \min_{(i,j) \in A_f} \{d(j) + 1\}$$

This implies

$$d'(i) \leq d(j) + 1 \quad \forall (i, j) \in A_f$$

$\square$

Lemma 10.5 (Super optimality). *For preflow $\vec{f}$ and distance labeling $\vec{d}$ there is no augmenting path from s to t in the residual graph G_f.*

Proof: Since $d(s) = n$ this follows directly from Corollary 10.3. $\square$

Lemma 10.6 (Optimality). *When there is no active node in the residual graph ($\forall v, \ e_v = 0$) preflow is a maximum flow.*

Proof: Since there is no node with positive excess, the preflow is a feasible flow. Also from the previous lemma there is no augmenting path, hence the flow is optimal. □

Lemma 10.7. *For any active node i, there exists a path in G_f from i to s.*

Proof: Let S_i be the nodes reachable in G_f from i. If $s \notin S_i$, then all nodes in S_i have non-negative excess. (Note that s is the only node in the graph that is allowed to have a negative excess - deficit.) In-flow into S_i must 0 (else can reach more nodes than S_i)

$$
\begin{aligned}
0 &= inflow(S_i) \\
&\geq inflow(S_i) - outflow(S_i) \\
&= \sum_{j \in S_i} [inflow(j) - outflow(j)] \\
&= \sum_{j \in S_i \setminus \{i\}} [inflow(j) - outflow(j)] + e_f(i) \\
&> 0.
\end{aligned}
$$

Since the node is active, hence it has strictly positive excess. Therefore, a contradiction. $\Rightarrow s \in S_i \Rightarrow$ there exists a path in G_f from i to s. □

Lemma 10.8. *When all active nodes satisfy $d(i) \geq n$, then the set of nodes $S = \{i|d(i) \geq n\}$ is a source set of a min-cut.*

Proof: Firstly, there is no residual (augmenting) path to t from any node in S (Corollary 10.3). Let $S^+ = \{i \mid d(i) \geq n, \ e(i) > 0\}$
From Lemma 10.7, each active node can send back excess to s. Hence, the preflow can be converted to (a feasible) flow and the saturated cut $(S, \bar{S})$ is a min-cut. Therefore the resulting feasible flow is maximum flow. (Although the conversion of preflow to a feasible flow can be accomplished by the algorithm, it is more efficient to send flow from excess nodes back to source by using the flow decomposition algorithm. That is, terminate the push-relabel algorithm when there is no active node of label $< n$.) □

Note that when we terminate the push-relabel algorithm when there is no active node of label $< n$, any node of label $\geq n$ must be active. This is since when it was last relabeled it was active, and could not have gotten rid of the access as of yet.

Notice also that the source set S is minimal source set of a min cut. The reason is that the same paths that are used to send the excess flows back to source, can then be used in the residual graph to reach those nodes of S.

Lemma 10.9. $\forall v, \ d(v) \leq 2n - 1$

Proof: Consider an active node v. From lemma 10.7, s is reachable from v in G_f. Consider a simple path, P, from v to s. $P = (i_0, i_1, ... i_k)$ where $i_0 = v, i_k = s$. Now,

$$
\begin{aligned}
d(s) &= n \text{ (fixed)} \\
d(i_0) &\leq d(i_1) + 1 \leq d(i_2) + 2 ..... \leq d(i_k) + k \\
d(i_0) &\leq d(i_k) + k = n + k \\
d(i_0) &\leq n + k
\end{aligned}
$$

Since any simple path has no more that n nodes, we have $k \leq n - 1$, which implies that $\Rightarrow d(i_0) \leq 2n - 1$. □

Lemma 10.10.   *1. $d(v)$ never decreases when relabeled*

    2. Immediately after relabel(v), v does not have incoming admissible arcs

Proof:

1. Before $relabel(v)$, $d(v) \leq d(w)$ $\forall (v,w) \in A_f$. After relabel the new label is $d(w)+1 \geq d(v)+1$ $\Rightarrow d(v)$ increased after relabeling.

2. Assume $(u,v) \in A_f$ and previously admissible $\Rightarrow d(u) = d(v)+1$. After relabel $\bar{d}(v)$ is the new label and $\bar{d}(v) \geq d(v) + 1 \geq d(u)$. Since v is now at least as high as u, (u,v) is inadmissible.

$\square$

Lemma 10.11 (Complexity of Relabel)**.**

1. *Total number of relabels = $O(n^2)$*

2. *Complexity of all relabels = $O(mn)$*

Proof:

1. Because of Lemma 10.9 each node can be relabeled at most 2n-1 times. Total number of nodes = n $\Rightarrow$ Total number of all relabels is less than $n(2n - 1) = O(n^2)$

2. The cost of relabel(v) is equal to the out-degree of v in G_f because for each v, we have to check all its adjacent nodes. Therefore, the cost of relabeling each node throughout the algorithm is $\leq n \times deg(v)$. The cost of relabeling all nodes is then at most $\sum_{v \in V} n \times deg(v) = 2mn$.

$\square$

Lemma 10.12 (Complexity of Push)**.** *Total number of saturating pushes is $O(mn)$*

Proof: After a saturating push on (u,v), we cannot push again to v, unless v pushes on (v,u). Therefore $d(v)$ has increased by at least 2 units and hence $d(u)$ must also have increased by at least 2 units between saturated pushes on (u,v) (See Figure 30.)
$\Rightarrow$ number of saturated pushes on $(u,v) \leq \frac{2n-1}{2} \leq n$
Therefore, for $O(m)$ arcs, total number of saturating pushes = $O(mn)$ $\square$

Lemma 10.13. *Total number of non saturating pushes = $O(n^2 m)$*

Proof: Define $\Phi = \sum_{v \ active} d(v)$
Initially $\Phi = 0$ because all initial labels are zero. A non-saturating push (v,w) reduces Φ by at least 1, because it deactivates node v and reduces Φ by $d(v)$. Note that $d(v) \geq 1$ always for a push(v,w) to occur. But d(w) may now become active and add to Φ. Therefore $\Phi' = \Phi + d(w) - d(v)$. However $d(v) > d(w) \Rightarrow \Phi$ is reduced by at least 1. In fact the decrease is exactly one if w was not previsouly active, since $d(v) \leq d(w) + 1$ for valid labeling; $d(v) \geq d(w) + 1$ for push(v,w) to occur. The total number of non saturating pushes is bounded by the total amount that Φ can increase during the algorithm. Φ can increase due to:

1. Saturating push :
   A saturating push can increase Φ by at most 2n-2 since it can activate at most one node and $d(v) \leq 2n - 1$.
   Total amount the Φ can increase per saturating push = $O(n)$
   Total number of saturating pushes = $O(nm)$
   Total amount that Φ can increase due to all saturating pushes = $O(n^2 m)$.
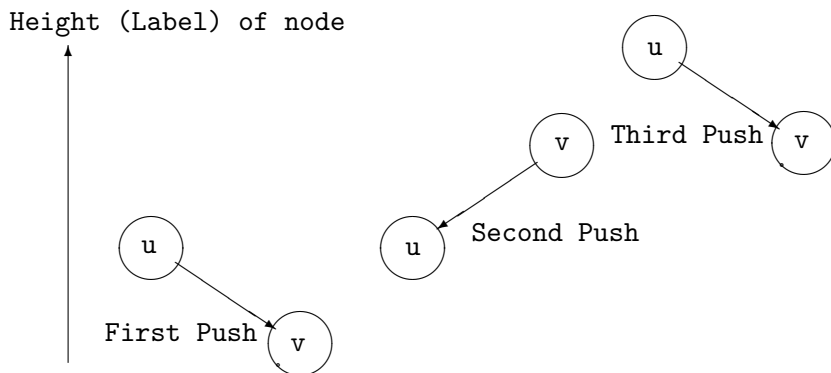
Figure 30: Sequence of saturating pushes on (u, v)

2. Relabeling:
   Each relabel increases Φ by the increase in the label.

   Maximum increase of label per node $= 2n - 1$
   Total increase in Φ by relabeling $= O(n^2)$. This term is dominated by $O(n^2 m)$.

Since total increase $= O(n^2 m)$, the number of non-saturating pushes is at most $O(n^2 m)$ □

Overall Complexity

Cost of relabels $= O(nm)$
Cost of saturating pushes $= O(nm)$
Cost of non-saturating pushes $= O(n^2 m)$
Therefore total complexity is $O(n^2 m)$.

10.3 Variants of Push/Relabel Algorithm

- FIFO: A node that becomes active is placed at the beginning of the active node list. Running time $= O(n^3)$

- Highest label preflow: picks an active node to process that has the highest distance label $O(\sqrt{m}n^2)$

- Dynamic tree implementation: Running time $= O(mn \frac{\log n^2}{m})$

10.4 Wave Implementation of Goldberg's Algorithm (Lift-to-front)

This algorithm has running time $O(n^3)$. The algorithm is listed below:

preprocess()
While (there exists an active node(i))
    Do
        if v is active then push/relabel v
        if v gets relabeled, place v in front of L
        else replace v by the node after v in L
    end

end

See the handout titled *Wave Algorithm.*
(from Cormen. Leiserson and Rivest).

Since the run time for relabel and saturating push are less than $O(n^3)$, we only have to show the run time of non-saturating push to be $O(n^3)$ to proof the overall running time to be $O(n^3)$.

Lemma 10.14. *The complexity of wave algorithm is $O(n^3)$.*

Proof. It suffices to show that there are $O(n^3)$ non-saturating pushes for the wave algorithm. Define a phase to be the period between two consecutive relabels.
There are $\leq n(2n-1)$ relabels. So there are $O(n^2)$ phases and we need to show that for each node there are ≤ 1 non-saturating push per phase. (Lemma 10.16.)

Claim 10.15. *The list L is in topological order for the admissible arcs of A_f.*

Proof. By induction on the iteration index. After preprocess, there are no admissible arcs and so any order is topological. Given a list in topological order, process node u. 'Push' can only eliminate admissible arcs (with a saturating push) and 'relabel' can create admissible arcs, but none into u (lemma 10.10), so moving u to the front of L maintains topological order.

Lemma 10.16. *At most one non-saturating push per node per phase.*

Proof. Let u be a node from which a non-saturating push is executed. Then u becomes inactive. Also all the nodes preceding u in L are inactive. In order for u to become active, one node preceding it has to become active. But every node can only make nodes lower than itself active. Hence the only way for u to become active is for some node to be relabeled.

It follows that the complexity of wave implementation of Goldberg's algorithm is $O(n^3)$.

11   The pseudoflow algorithm (Hochbaum's PseudoFlow HPF)

In this section we provide a description of the pseudoflow algorithm. Our description here is different from than in [Hoc08] although the algorithm itself is the same. This description uses terminology and concepts similar in spirit to push-relabel in order to help clarify the similarities and differences between the two algorithms.

The first step in the pseudoflow algorithm, called the *Min-cut Stage* finds a minimum cut in G_{st}. Source-adjacent and sink-adjacent arcs are saturated throughout this stage of the algorithm; consequently, the source and sink have no role to play in the Min-cut Stage.

The algorithm may start with any other pseudoflow that saturates arcs in $A_s \cup A_t$. Other than that, the only requirement of this pseudoflow is that the collection of *free arcs*, namely the arcs that satisfy $\ell_{ij} < f_{ij} < u_{ij}$, form an acyclic graph.

Each node in $v \in V$ is associated with at most one *current arc* $(u, v) \in A^f$; the corresponding *current node* of v is denoted by current$(v) = u$. The set of current arcs in the graph satisfies the following invariants at the beginning of every major iteration of the algorithm:

Property 1.   *(a) The graph does not contain a cycle of current arcs.*

(b) If $e(v) \neq 0$, then node v does not have a current arc.

Each node is associated with a *root* that is defined constructively as follows: starting with node v, generate the sequence of nodes $\{v, v_1, v_2, \ldots, v_r\}$ defined by the current arcs $(v_1, v), (v_2, v_1), \ldots, (v_r, v_{r-1})$ until v_r has no current arc. Such node v_r always exists, as otherwise a cycle will be formed, which would violate Property 1(a). Let the unique root of node v be denoted by root(v). Note that if a node v has no current arc, then root(v) = v.

The set of current arcs forms a *current forest*. Define a *component* of the forest to be the set of nodes that have the same root. It can be shown that each component is a directed tree, and the following properties hold for each component.

Property 2. *In each component of the current forest,*
(a) The root is the only node without a current arc.
(b) All current arcs are pointed away from the root.

While it is tempting to view the ability to maintain pseudoflows as an important difference between the two algorithms, it is trivial to modify the push-relabel algorithm (as shown in [CH09]) to handle pseudoflows.

The key differences between the pseudoflow and push-relabel algorithms are: (1) that the pseudoflow algorithm allows flow to be pushed along arcs (u, v) in which $\ell(u) = \ell(v)$ whereas this is not allowed in push-relabel; and (2) that flow is pushed along a path rather than along single arcs. Note that [GR98] proposed a maximum flow algorithm with complexity superior to that of push-relabel that relied on being able to send flow along arcs with $\ell(u) = \ell(v)$.

11.1   Initialization

The pseudoflow algorithm starts with a pseudoflow and an associated current forest, called in [Hoc08], a *normalized tree*. Each non-root in a branch (tree) of the forest has a parent node, and the unique arc from the parent node is called *current arc*. The root has no parent node and no current arc.

The generic initialization is the *simple* initialization: source-adjacent and sink-adjacent arcs are saturated while all other arcs have zero flow.

If a node v is both source-adjacent and sink-adjacent, then at least one of the arcs (s, v) or (v, t) can be pre-processed out of the graph by sending a flow of $\min\{c_{sv}, c_{vt}\}$ along the path $s \to v \to t$. This flow eliminates at least one of the arcs (s, v) and (v, t) in the residual graph. We henceforth assume w.l.o.g. that no node is both source-adjacent and sink-adjacent.

The simple initialization creates a set of source-adjacent nodes with excess, and a set of sink-adjacent nodes with deficit. All other arcs have zero flow, and the set of current arcs is selected to be empty. Thus, each node is a singleton component for which it serves as the root, even if it is *balanced* (with 0-deficit).

A second type of initialization is obtained by saturating all arcs in the graph. The process of saturating all arcs could create nodes with excesses or deficits. Again, the set of current arcs is empty, and each node is a singleton component for which it serves as the root. We refer to this as the *saturate-all initialization* scheme.

11.2   A labeling pseudoflow algorithm

In the labeling pseudoflow algorithm, each node $v \in V$ is associated with a distance label $\ell(v)$ with the following property.

Figure 31: (a) Components before merger (b) Before pushing flow along admissible path from r_i to r_j (c) New components generated when arc (u,v) leaves the current forest due to insufficient residual capacity.

(a)   (b)   (c)
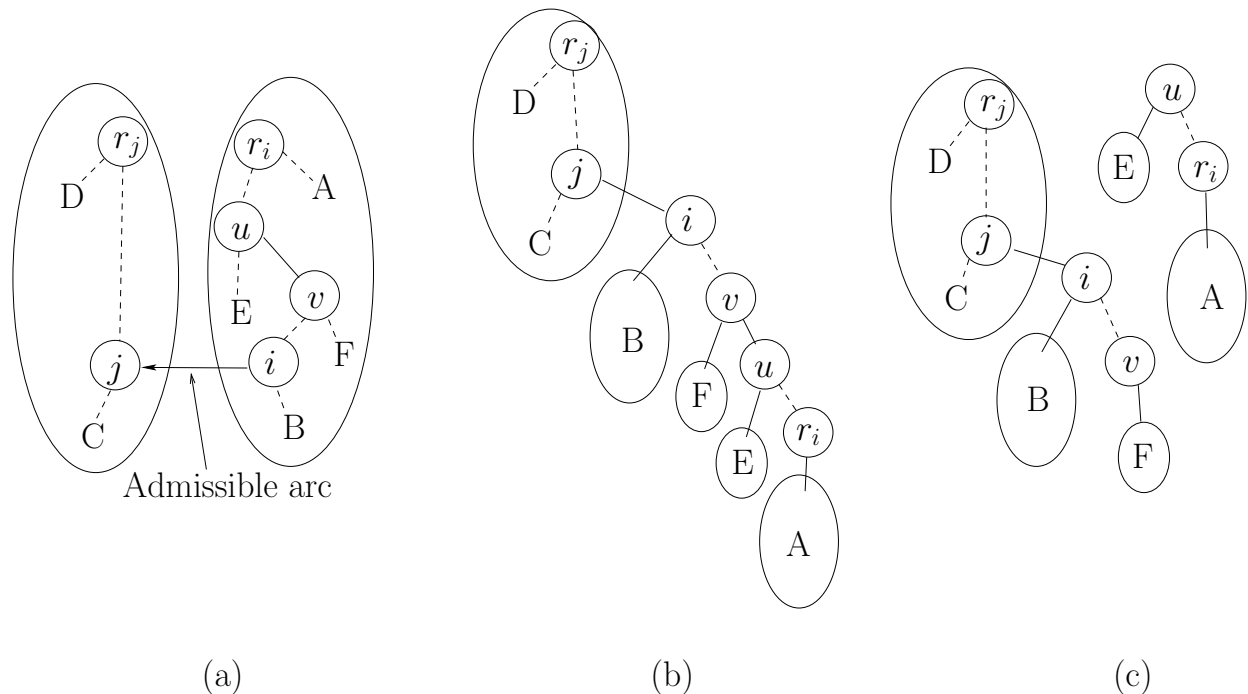
Property 3. *The node labels satisfy:*
 (a) For every arc $(u,v) \in A^f$, $\ell(u) \leq \ell(v) + 1$.
 (b) For every node $v \in V$ with strictly positive deficit, $\ell(v) = 0$.

Collectively, the above two properties imply that $\ell(v)$ is a lower bound on the distance (in terms of number of arcs) in the residual network of node v from a node with strict deficit. A residual arc (u,v) is said to be *admissible* if $\ell(u) = \ell(v) + 1$.

A node is said to be *active* if it has strictly positive excess. Given an admissible arc (u,v) with nodes u and v in different components, we define an *admissible path* to be the path from root(u) to root(v) along the set of current arcs from root(u) to u, the arc (u,v), and the set of current arcs (in the reverse direction) from v to root(v).

We say that a component of the current forest is a *label-n component* if for every node v of the component $\ell(v) = n$. We say that a component is a *good active component* if its root node is active and if it is not a label-n component.

An iteration of the pseudoflow algorithm consists of choosing a good active component and attempting to find an admissible arc from a *lowest labeled* node u in this component. (Choosing a lowest labeled node for processing ensures that an admissible arc is never between two nodes of the same component.) If an admissible arc (u,v) is found, a *merger* operation is performed. The merger operation consists of pushing the entire excess of root(u) towards root(v) along the admissible path and updating the excesses and the arcs in the current forest to preserve Property 1.

If no admissible arc is found, $\ell(u)$ is increased by 1 unit. A schematic description of the merger operation is shown in Figure 31. The pseudocode for the generic labeling pseudoflow algorithm is given in Figures 32 through 34.

```
/*
Min-cut stage of the generic labeling pseudoflow algorithm. All nodes in label-n compo-
nents form the nodes in the source set of the min-cut.
*/
procedure GenericPseudoflow (V_{st}, A_{st}, c):
    begin
        SimpleInit (A_s, A_t, c);
        while ∃ a good active component T do
            Find a lowest labeled node u ∈ T;
            if ∃ admissible arc (u, v) do
                Merger (root(u), · · · , u, v, · · · , root(v));
            else do
                ℓ(u) ← ℓ(u) + 1;
    end
```

Figure 32: Generic labeling pseudoflow algorithm.

```
/*
Saturates source- and sink-adjacent arcs.
*/
procedure SimpleInit(A_s, A_t, c):
    begin
        f, e ← 0;
        for each (s, i) ∈ A_s do
            e(i) ← e(i) + c_{si}; ℓ(i) = 1;
        for each (i, t) ∈ A_t do
            e(i) ← e(i) − c_{it}; ℓ(i) = 0;
        for each v ∈ V do
            ℓ(v) ← 0;
            current(v) ← ∅;
    end
```

Figure 33: Simple initialization in the generic labeling pseudoflow algorithm.

11.3   The monotone pseudoflow algorithm

In the generic labeling pseudoflow algorithm, finding the lowest labeled node within a component may take excessive time. The *monotone implementation* of the pseudoflow algorithm efficiently finds the lowest labeled node within a component by maintaining an additional property of *monotonicity* among labels in a component.

Property 4 ([Hoc08])**.** *For every current arc (u, v), $\ell(u) = \ell(v)$ or $\ell(u) = \ell(v) - 1$.*

This property implies that within each component, the root is the lowest labeled node and node labels are non-decreasing with their distance from the root. Given this property, all the lowest labeled nodes within a component form a sub-tree rooted at the root of the component. Thus, once a good active component is identified, all the lowest labeled nodes within the component are examined for admissible arcs by performing a depth-first-search in the sub-tree starting at the root.

```
/*
Pushes flow along an admissible path and preserves invariants.
*/
procedure Merger(v₁, ⋯ , vₖ):
    begin
        for each j = 1 to k − 1 do
            if e(vⱼ) > 0 do
                δ ← min{c(vⱼ, vⱼ₊₁), e(vⱼ)};
                e(vⱼ) ← e(vⱼ) − δ;
                e(vⱼ₊₁) ← e(vⱼ₊₁) + δ;
            if e(vⱼ) > 0 do
                current(vⱼ) ← ∅;
            else do
                current(vⱼ) ← vⱼ₊₁;
    end
```

Figure 34: Push operation in the generic labeling pseudoflow algorithm.

In the generic labeling algorithm, a node was relabeled if no admissible arc was found from the node. In the monotone implementation, a node u is relabeled only if no admissible arc is found *and* for all current arcs (u, v) in the component, $\ell(v) = \ell(u) + 1$. This feature, along with the merger process, inductively preserves the monotonicity property. The pseudocode for the Min-cut Stage of the monotone implementation of the pseudoflow algorithm is given in Figure 35.

```
/*
Min-cut stage of the monotone implementation of pseudoflow algorithm.  All nodes in
label-n components form the nodes in the source set of the min-cut.
*/
procedure MonotonePseudoflow (Vₛₜ, Aₛₜ, c):
    begin
        SimpleInit (Aₛ, Aₜ, c);
        while ∃ a good active component T with root r do
            u ← r;
            while u ≠ ∅ do
                if ∃ admissible arc (u, v) do
                    Merger (root(u), ⋯ , u, v, ⋯ , root(v));
                    u ← ∅;
                else do
                    if ∃w ∈ T : (current(w) = u) ∧ (ℓ(w) = ℓ(u)) do
                        u ← w;
                    else do
                        ℓ(u) ← ℓ(u) + 1;
                        u ← current(u);
    end
```

Figure 35: The monotone pseudoflow algorithm.

The monotone implementation simply delays relabeling of a node until a later point in the

algorithm, which does not affect correctness of the labeling pseudoflow algorithm.

11.4 Complexity summary

In the monotone pseudoflow implementation, the node labels in the admissible path are non-decreasing. To see that notice that for a merger along an admissible arc (u, v) the nodes along the path $\text{root}(u), \cdots, u$ all have equal label and the nodes along the path $v, \cdots, \text{root}(v)$ have non-decreasing labels (from Property 4). A merger along an admissible arc (u, v) either results in arc (u, v) becoming current, or in (u, v) leaving the residual network. In both cases, the only way (u, v) can become admissible again is for arc (v, u) to belong to an admissible path, which would require $\ell(v) \geq \ell(u)$, and then for node u to be relabeled at least once so that $\ell(u) = \ell(v) + 1$. Since $\ell(u)$ is bounded by n, (u, v) can lead to $O(n)$ mergers. Since there are $O(m)$ residual arcs, the number of mergers is $O(nm)$.

The work done per merger is $O(n)$ since an admissible path is of length $O(n)$. Thus, total work done in mergers including pushes, updating the excesses, and maintaining the arcs in the current forest, is $O(n^2 m)$.

Each arc (u, v) needs to be scanned at most once for each value of $\ell(u)$ to determine if it is admissible since node labels are non-decreasing and $\ell(u) \leq \ell(v) + 1$ by Property 3. Thus, if arc (u, v) were not admissible for some value of $\ell(u)$, it can become admissible only if $\ell(u)$ increases. The number of arc scans is thus $O(nm)$ since there are $O(m)$ residual arcs, and each arc is examined $O(n)$ times.

The work done in relabels is $O(n^2)$ since there are $O(n)$ nodes whose labels are bounded by n.

Finally, we need to bound the work done in the depth-first-search for examining nodes within a component. Each time a depth-first-search is executed, either a merger is found or at least one node is relabeled. Thus, the number of times a depth-first-search is executed is $O(nm + n^2)$ which is $O(nm)$. The work done for each depth-first-search is $O(n)$, thus total work done is $O(n^2 m)$.

Lemma 11.1. *The complexity of the monotone pseudoflow algorithm is $O(n^2 m)$.*

Regarding the complexity of the algorithm, [Hoc08] showed that an enhanced variant of the pseudoflow algorithm is of complexity $O(mn \log n)$. That variant uses dynamic trees data structure and is not implemented here. Recently, however, [HO12] showed that the "highest label" version of the pseudoflow algorithm has complexity $O(mn \log \frac{n^2}{m})$ and $O(n^3)$, with and without the use of dynamic trees, respectively.

References

[CH09] B.G. Chandran and D.S. Hochbaum. A computational study of the pseudoflow and push-relabel algorithms for the maximum flow problem. *Operations Research*, 57(2):358 – 376, 2009.

[Hoc08] D.S. Hochbaum. The pseudoflow algorithm: A new algorithm for the maximum-flow problem. *Operations Research*, 56(4):992–1009, 2008.

[HO12] D.S. Hochbaum and J.B. Orlin. Simplifications and speedups of the pseudoflow algorithm. *Networks*, 2012. (to appear).

12 The minimum spanning tree problem (MST)

Given a connected graph $G = (V, E)$, with weight w_e for all edge in E, find a subset of edges E_T that contains no cycles and so that the spanning tree $G_T = (V_T, E_T)$ of minimum total weight. (Spanning means that $V_T = V$.)

12.1 IP formulation

The decision variables for the IP formulation of MST are:

$$x_e = \begin{cases} 1 & \text{if edge } e \in E_T \\ 0 & \text{otherwise} \end{cases}$$

The constraints of the IP formulation need to enforce that the edges in E_T form a tree. Recall from early lectures that a tree satisfies the following three conditions: It has $n - 1$ edges, it is connected, it is acyclic. Also recall that if any two of these three conditions imply the third one. An IP formulation of MST is:

$$\min \quad \sum_{e \in E} w_e x_e \tag{15a}$$

$$\text{s.t.} \quad \sum_{e \in E} x_e = n - 1 \tag{15b}$$

$$\sum_{e \in (S,S)} x_e \leq |S| - 1 \quad \forall \, S \subseteq V \tag{15c}$$

$$x_e \in \{0, 1\} \quad \forall \, e \in E \tag{15d}$$

where (S, S) denotes the set of edges that have both endpoints in S. Inequality (15c) for S enforces the property that if a subset of edges in (S, S) is connected (induces a connected subgraph), then this subset is acyclic. If (S, S) is disconnected and contains a cycle, then there is another set $S' \subset S$ that violates the respective constraint for S'. Therefore the edges in E_T can't form cycles.

We have the following observations.

1. The constraint matrix of problem (15) does not have a network flow structure, and is not totally unimodular. However Jack Edmonds proved that the LP relaxation has integral extreme points.

2. The formulation contains an exponential number of constraints.

3. Even though the LP relaxation has integral extreme points, this does not imply that we can solve the MST problem in polynomial time. This is because of the exponential size of the formulation. Nevertheless, we can use the ellipsoid method with a *separation algorithm* to solve problem (15):

   Relax the set of constraints given in (15c), and solve the remaining problem. Given the solution to such relaxed problem, find one of the relaxed constraints that is violated (this process is called separation) and add it. Resolve the problem including the additional constraints, and repeat until no constraint is violated.

   It is still not clear that the above algorithm solves in polynomial time MST. However, in light of the equivalence between separation and optimization (one of the central theorems in optimization theory), and since we can separate the inequalities (15c) in polynomial time, it follows that we can solve problem (15) in polynomial time. This can be done using the Ellipsoid algorithm for linear programming.

4. An alternative IP formulation of MST is obtained by imposing the connectivity condition (instead of the acyclicity condition). This condition is enforce by the following set of inequalities, which guarantee that there is at least one edge across each cut:

$$\sum_{e \in (S, \bar{S})} x_e \geq 1 \quad \forall \, S \subset V.$$

Note that, like the set of constraints (15c), the number of these inequalities is exponential.

12.2 Properties of MST

12.2.1 Cut Optimality Condition

Theorem 12.1. *Let T^* be a spanning tree in $G = (V, E)$. Consider edge $[i, j]$ in T^*. Suppose removing $[i, j]$ from T^* disconnects the tree into S and $\bar{S}$. T^* is a MST iff for all such $[i, j]$, for all $[u, v], u \in S, v \in \bar{S}, w_{uv} \geq w_{ij}$.*

Proof. Suppose T^* is an MST and $w_{ij} > w_{kl}$ for some $k \in S, l \in \overline{S}$. Let $T' = T^* \bigcup T(\overline{S}) \bigcup [k, l]$. Then T' is a spanning tree of lower weight than T^*.

Given a spanning tree T^* that satisfies the cut optimality condition. Suppose there exists some MST $T' \neq T^*$, with $[k, l] \in T'$ but not in T^*.
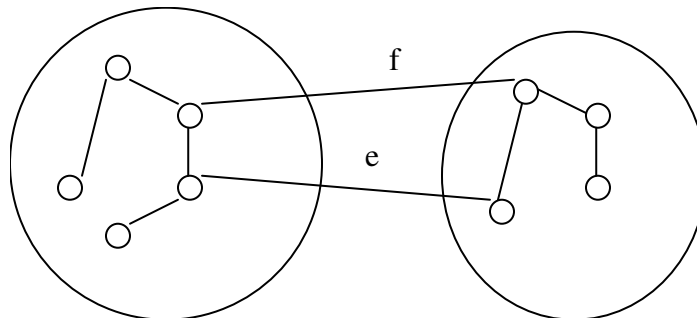
Figure 36: Cut Optimality Condition

We will construct T'' from T^* with one less edge different from T'. Add $[k, l]$ to T^*, creating a cycle. Let $[i, j]$ be the arc with the largest weight in the cycle. Removing this creates a new spanning tree. But, by the cut optimality condition, $w_{ij} \leq w_{kl}$, but by the optimality of T', $w_{ij} \geq w_{kl}$, so the new tree has the same weight as the old tree. We can repeat this process for all iterations, and find that T^* has the same cost as T'. □

12.2.2 Path Optimality Condition

Theorem 12.2. *T^* is a MST iff for all $[i, j]$ not in T^*, $c_{ij} \geq c_{kl}$ for all $[k, l]$ in the unique path from i to j in T^*.*

Proof. Let T^* be an MST. Suppose $\exists [i, j]$ not in T^* that has a strictly lower cost than an edge in the path from i to j in T^*. Then add $[i, j]$ to T^*, forming a unique cycle, and remove one such edge in the path. The new spanning tree is of strictly lower cost, which is a contradiction.
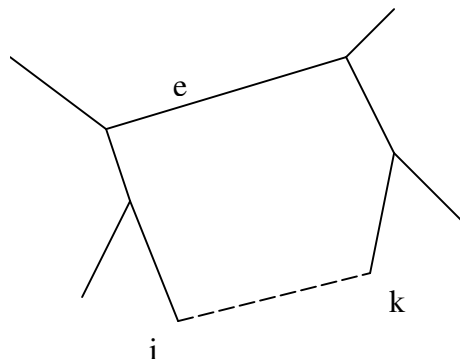
Figure 37: Path optimality condition

Let T^* be a spanning tree that satisfies the path optimality condition. Remove edge $[i, j] \in T^*$, disconnecting T^* into S and $\overline{S}$ such that $i \in S, j \in \overline{S}$.

For every edge $[k, l]$ not in T^*, $k \in S, l \in \overline{S}$, the path from k to l must have used $[i, j]$ in T^*. But $c_{ij} \leq c_{kl}$ by path optimality, which is the same statement as cut optimality, so the statements are equivalent. $\qquad \square$

12.3 Algorithms of MST

The correctness of the two algorithms for MST given here follows from the following theorem:

Theorem 12.3. *Let $F = \{(U_1, E_1), ...(U_k, E_k)\}$ be a spanning forest (a collection of disjoint trees) in G. Pick edge $[i, j]$ of minimum weight such that $i \in U_1$, $j \in U_1^c$. Then some MST containing $E_1 \bigcup E_2 \bigcup ...E_k$ will contain edge $[i, j]$.*

Proof. Suppose no MST containing F contains edge $[i, j]$. Suppose you are given T^* not containing $[i, j]$. Add $[i, j]$ to E_{T^*}, this will create a cycle, remove the other edge from U_1 to U_1^c that belongs to the created cycle. Since the weight of this removed edge must be greater or equal than that of edge $[i, j]$, this is a contradiction. $\qquad \square$

12.3.1 Prim's algorithm

Begin with a spanning forest (in the first iteration let each node being one of the "trees" in this forest). Find the shortest edge $[i, j]$ such that $i \in U_1$ and $j \in U_1^c$), add $[i, j]$ to T_1 and j to U_1. Repeat this process of "growing" U_1 until $U_1 = V$.
begin
$U = \{1\}; T = \emptyset$.
while $U \neq V$ **do**
    Pick $[i, j] \in E$ such that $i \in U, j \in U^c$ of minimum weight
    $T \leftarrow T \bigcup [i, j]; P \leftarrow P \bigcup j$
end

The while loop is executed $n - 1$ times, and finding an edge of minimum weight takes $O(m)$ work, so this naive implementation is an $O(nm)$ algorithm. However, note that similarity between this algorithm and Dijkstra's algorithm; in particular note that we can apply the same tricks to improve its running time.

We can make this a faster algorithm by maintaining, in a binary heap, a list of sorted edges.

Each edge is added to the heap when one of its end points enters P, while the other is outside P. It is then removed from the heap when both its end points are in P. Adding an element to a heap takes $O(\log m)$ work, and deleting an element also takes $O(\log m)$ work. Thus, total work done in maintaining the heap is $O(m \log m)$, which is $O(m \log n)$ since m is $O(n^2)$. Thus, total work done in this binary heap implementation is $O(m \log n)$.

With Fibonacci heaps, this can be improved to $O(m + n \log n)$.

For dense graphs, there is a faster, $O(n^2)$ algorithm. This algorithm maintains an array $Q[i]$ for the shortest distance from P to each node $i \in \bar{P}$, outside of P.

begin
   $P = \{1\}, Q[i] = w_{1i}$
   while $P \neq V$ **do**
      Pick $[i, j] \in E$ such that $i \in P, j \in P^c$ of minimum weight
      $T \leftarrow T \bigcup [i, j], P \leftarrow P \bigcup \{j\}$
      for each $[k, j] : k \notin P$ **do**
         $Q[k] \leftarrow \min\{Q[k], w_{kj}\}$
end

This is an $O(n^2)$ algorithm since the while loop is executed $O(n)$ times, and the for loop is executed $O(n)$ times within each while loop. For dense graphs $O(n^2)$ is $O(m)$ and therefore this is an optimal, linear time, algorithm for MST on dense graphs.
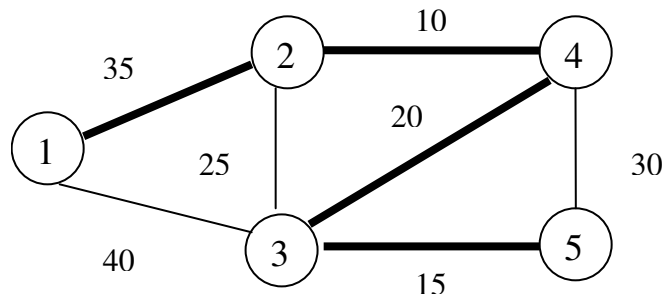
Figure 38: An example of applying Prim's Algorithm

12.3.2 Kruskal's algorithm

Kruskal's algorithm works with a forest and at each iteration it adds an edge that reduces the number of components by one. The algorithm takes as input the ordered list of edges: Sort the edges in non-decreasing order of weight: $w_{e_1} \leq c_{e_2} \leq ...c_{e_m}$.

begin
   $k \leftarrow 1, i \leftarrow 1, E_T \leftarrow \emptyset$
   while $i \leq n$ **do**
      let $e_k = [i, j]$
      if i and j are not connected in (V, E_T) **do**
         $E \leftarrow E \bigcup [i, j]$
         $T \leftarrow T \bigcup \{i, j\}$
      $k \leftarrow k + 1$
end
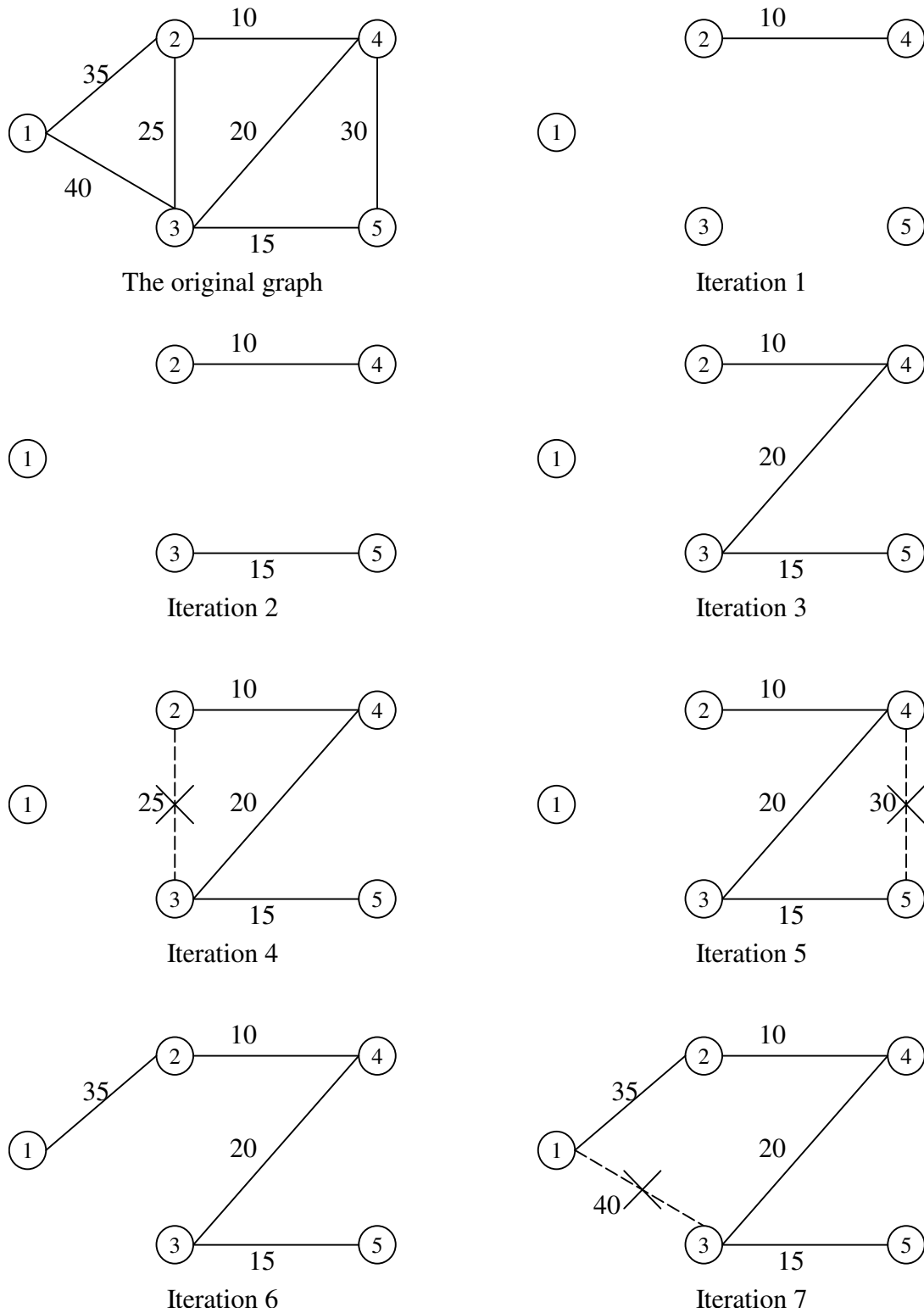
Figure 39: An example of Kruskal's algorithm

The algorithm is illustrated in Figure 39.

Complexity: The while loop is executed at most m times, and checking if i and j are in the same

component can be done with DFS on (V, E_T), this takes $O(n)$. Therefore this naive implementation has a running time of $O(nm)$.

We will now show that Kruskal's algorithm has a running time of $O(m + n \log n)$ (plus the time to sort the edges) without special data structures:

Each component is maintained as a linked list with a first element label, for each node in the component, as a representative of the component. The size of the component is also stored with the representative "first" of the component.

1. Checking for edge $[i, j]$ whether i and j belong to the same component is done by comparing their labels of component's first. This is done in $O(1)$.
2. If for edge $[i, j]$, i and j belong to the different components, then the edge is added and the two components are merged. In that case the *smaller* component's nodes labels are all updated to assume the label of the representative of the larger component. The size of the component that is the second label of "first" is modified by adding to it the size of the smaller component. The complexity is the size of the smaller component merged.

For each edge $[i, j]$ inspected, checking whether i and j belong to the same component or not takes $O(m)$ operations throughout the algorithm. To evaluate the complexity of merger, note that the "first" label of a node i is updated when i is in the smaller component. After the merger the size of the component that contains i at least doubles. Hence the label update operation on node i can take place at most $O(\log n)$ times. Therefore the total complexity of the labels updates throughout the algorithm is $O(m \log n)$.

Hence, the total complexity of Kruskal's algorithm is $O(m \log n + m + n \log n)$. The complexity of the initial sorting of the edges, $O(m \log n)$, dominates the run time of the algorithm.

12.4   Maximum spanning tree

Note that we never assumed nonnegativity of the edge weights. Therefore we can solve the maximum spanning tree problem with the above algorithms after multiplying the edge weights by -1.

13   Complexity classes and NP-completeness

In optimization problems, there are two interesting issues: one is *evaluation*, which is to find the optimal value of the objective function (evaluation problems); the other one is *search*, which is to find the optimal solution (optimization problems).

13.1   Search vs. Decision

**Decision Problem** - A problem to which there is a yes or no answer.

Example. SAT = {Does there exist an assignment of variables which satisfies the boolean function ϕ; where ϕ is a conjunction of a set of clauses, and each clause is a disjunction of some of the variables and/or their negations?}

Evaluation Problem - A problem to which the answer is the cost of the optimal solution.

Note that an evaluation problem can be solved by solving a auxiliary decision problems of the form "Is there a solution with value less than or equal to M?". Furthermore, using binary search, we only have to solve a polynomial number of auxiliary decision problems.

Optimization Problem, the search problem - A problem to which the answer is an optimal solution.

To illustrate, consider the Traveling Salesperson Problem (**TSP**). TSP is defined on an undirected graph, $G = (V, E)$, where each edge $(i, j) \in E$ has an associated distance c_{ij}.

TSP_OPT = { Find a tour (a cycle that visits each node exactly once) of total minimum distance. }

TSP_EVAL = { What is the total distance of the tour with total minimum distance in $G = (V, E)$? }

TSP_DEC = { Is there a tour in $G = (V, E)$ with total distance $\leq M$? }

Given an algorithm to solve **TSP_DEC**, we can solve **TSP_EVAL** as follows.

1. Find the upper bound and lower bound for the TSP optimal objective value. Let $C_{min} = \min_{(i,j) \in E} c_{ij}$, and $C_{max} = \max_{(i,j) \in E} c_{ij}$. Since a tour must contain exactly n edges, then an upper bound (lower bound) for the optimal objective value is $n \cdot C_{max}$, $(n \cdot C_{min})$.

2. Find the optimal objective value by binary search in the range $[n \cdot C_{min}, n \cdot C_{max}]$. This binary search is done by calling the algorithm to solve **TSP_DEC** $O(\log_2 n(C_{max} - C_{min}))$ times, which is a polynomial number of times.

However, note that knowing the optimal objective value **does not** imply that we know the solution corresponding to it.

An important problem for which the distinction between the decision problem and giving a solution to the decision problem – the *search problem* – is significant is primality. It was an open question (until Aug 2002) whether or not there exist polynomial-time algorithms for testing whether or not an integer is a prime number. [2] However, for the corresponding search problem, finding all factors of an integer, no similarly efficient algorithm is known.

Another example: A graph is called k-connected if one has to remove at least k vertices in order to make it disconnected. A theorem says that there exists a partition of the k-connected graph into k connected components of sizes $n_1, n_2 \ldots n_k$ such that $\sum_{i=1}^{k} n_k = n$ (the number of vertices), such that each component contains one of the vertices $v_1, v_2 \ldots v_k$. The optimization problem is finding a partition such that $\sum_i |n_i - \bar{n}|$ is minimal, for $\bar{n} = n/k$. No polynomial-time algorithm is known for $k \geq 4$. However, the corresponding recognition problem is trivial, as the answer is always *Yes* once the graph has been verified (in polynomial time) to be k-connected. So is the evaluation problem: the answer is always the sum of the averages rounded up with a correction term, or the sum of the averages rounded down, with a correction term for the residue.

For the rest of the discussion, we shall refer to the decision version of a problem, unless stated otherwise.

13.2 The class *NP*

A very important class of decision problems is called *NP*, which stands for nondeterministic polynomial-time. It is an abstract class, not specifically tied to optimization.

Definition 13.1. *A decision problem is said to be in NP if for all "Yes" instances of it there exists a polynomial-length "certificate" that can be used to verify in polynomial time that the answer is indeed Yes.*

Imagine that you are a *verifier* and that you want to be able to confirm that the answer to a given decision problem is indeed "yes". Problems in *NP* have poly-length certificates that, without

---

[2]There are reasonably fast superpolynomial-time algorithms, and also Miller-Rabin's algorithm, a randomized algorithm which runs in polynomial time and tells either "I don't know" or "Not prime" with a certain probability. In August 2002 (here is the official release) "Prof. Manindra Agarwal and two of his students, Nitin Saxena and Neeraj Kayal (both BTech from CSE/IITK who have just joined as Ph.D. students), have discovered a polynomial time deterministic algorithm to test if an input number is prime or not. Lots of people over (literally!) centuries have been looking for a polynomial time test for primality, and this result is a major breakthrough, likened by some to the P-time solution to Linear Programming announced in the 70s."
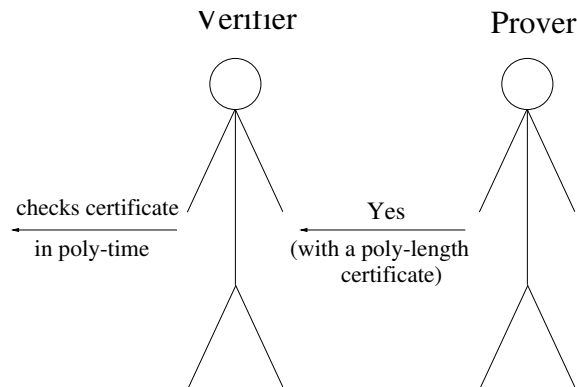
Figure 40: Recognizing Problems in *NP*

necessarily indicating how the answer was obtained, allow for this verification in polynomial time (see Figure 40).

To illustrate, consider again the decision version of TSP. That is, we want to know if there exists a tour with total distance $\leq M$. If the answer to our problem is "yes", the prover can provide us with such a tour and we can verify in polynomial time that 1) it is a valid tour and 2) its total distance is $\leq M$. However if the answer to our problem is "no", then (as far as we know) the only way that to verify this would be to check every possible tour (this is certainly not a poly-time computation).

13.2.1   Some Problems in *NP*

Hamiltonian Cycle = {Is there a Hamiltonian circuit in the graph $G = (V, A)$?}
    <u>Certificate</u> - the tour.
Clique = {Is there a clique of size $\leq k$ in the graph $G = (V, E)$?}
    <u>Certificate</u> - the clique.
Compositeness = {Is N composite (can N be written as $N = ab$, a and b are integers, s.t. $a, b > 1$)? }
    <u>Certificate</u> - the factors a and b. (*Note 1:* $\log_2 N$ is the length of the input, length of certificate $\leq 2 \log_2 N$; and we can check in poly-time (multiply). *Note 2:* Giving only one of the factors is also OK.)
Primality = {Is N prime?}
    <u>Certificate</u> - not so trivial. In fact, it was only in 1976 that Pratt showed that one exists.
TSP = {Is there a tour in $G = (V, A)$ the total weight of which is $\leq M$?}
    <u>Certificate</u> - the tour.
Dominating Set = {Is there is a dominating set $S \subseteq V$ in a graph $G = (V, A)$ such that $|S| \leq k$?}
    <u>Certificate</u> - the set. (A vertex is said to dominate itself and its neighbors. A dominating set dominates all vertices)
LP = {Is there a vector $\vec{x}$, $\vec{x} \geq 0$ and $A\vec{x} \leq b$ such that $C\vec{x} \leq k$?}
    <u>Certificate</u> - the feasible vector $\vec{x}$. (This vector may be arbitrarily large. However, every basic feasible solution $\vec{x}$ can be shown to be poly-length in terms of the input. Clearly, checkable in polynomial-time.)
IP = {Is there a vector $\vec{x}$, $\vec{x} \geq 0, \vec{x} \in Z$ and $A\vec{x} \leq b$ such that $C\vec{x} \leq k$?}
    <u>Certificate</u> - a feasible vector $\vec{x}$. (just like LP, it can be certified in polynomial time; it is also necessary to prove that the certificate is only polynomial in length. see AMO, pg 795.)

k-**center** = {Given a graph $G = (V, E)$ weighted by $w : V \times V \mapsto R^+$, is there a subset $S \subseteq V$, $|S| = k$, such that $\forall v \in V \setminus S$, $\exists s \in S$ such that $w(v, s) \le m$?}
　　Certificate - the subset.
Partition = {Given a list of integers $\{b_1, \dots, b_n\}$, of sum $\sum_{i=1}^{n} b_i = 2K$, is there a subset $S \subseteq \{1, \dots, n\}$ such that $\sum_{i \in S} b_i = K$?}
　　Certificate - the subset.

13.3   co-NP

Suppose the answer to the recognition problem is *No*. How would one certify this?

Definition 13.2. *A decision problem is said to be in co-NP if for all "No" instances of it there exists a polynomial-length "certificate" that can be used to verify in polynomial time that the answer is indeed No.*

13.3.1   Some Problems in *co-NP*

Primality = {Is N prime? }
　　"No" Certificate - the factors.
LP = {Is there a vector $\vec{x}$, $\vec{x} \ge 0$ and $A\vec{x} \le b$ such that $C\vec{x} \le k$? }
　　"No" Certificate - the feasible *dual* vector $\vec{y}$ such that $\vec{y} \le 0$, $\vec{y}^T A \le c$ and $\vec{y}^T b > k$ (from Weak Duality Thm.)

13.4   *NP* **and** *co-NP*

As a verifier, it is clearly nice to have certificates to confirm both "yes" *and* "no" answers; that is, for the problem to be in both *NP* and *co-NP* . From Sections 13.2.1 and 13.3.1, we know that the problems **Prime** and **LP** are in both *NP* and *co-NP* . However, for many problems in *NP* , such as **TSP**, there is no obvious polynomial "no" certificate.
　　Before we look at these 2 types of problems in detail, we define P , the class of polynomially solvable problems.

Definition 13.3. *Let P be the class of polynomial-time (as a function of the length of the input) solvable problems.*

　　It is easy to see that $P \subseteq NP \cap co\text{-}NP$ – just find an optimal solution. It is not known, however, whether, $P = NP \cap co\text{-}NP$; the problems that are in the intersection but for which no polynomial-time algorithm is known are usually very pathological. Primality $\in NP \cap co\text{-}NP$; this gave rise to the conjecture that primality testing can be carried out in polynomial time, as indeed it was recently verified to be.
　　Notice that the fact that a problem belongs to $NP \cap co\text{-}NP$ does not automatically suggest a polynomial time algorithm for the problem.
　　However, being in $NP \cap co\text{-}NP$ is usually considered as strong evidence that there *exists* a polynomial algorithm. Indeed, many problems were known to be in $NP \cap co\text{-}NP$ well before a polynomial algorithm was developed (ex. **LP**, **primality**); this membership is considered to be strong evidence that there was a polynomial algorithm out there, giving incentive for researchers to put their efforts into finding one. Both **primality** and **compositeness** lie in $\in NP \cap co\text{-}NP$; this gave rise to the conjecture that both can be solved in polynomial time.
　　On the flip-side, *NP* -Complete problems, which we turn to next, are considered to be immune to any guaranteed polynomial algorithm.
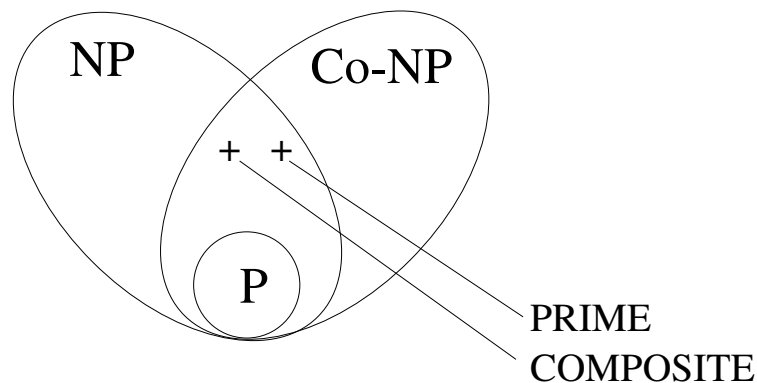
Figure 41: *NP* , *co-NP* , and *P*

13.5 *NP* -completeness and reductions

A major open question in contemporary computer science is whether $P = NP$. It is currently believed that $P \neq NP$.

As long as this conjecture remains unproven, instead of proving that a certain *NP* problem is not in P , we have to be satisfied with a slightly weaker statement: if the problem is in P then $P = NP$, that is, a problem is "at least as hard" as any other *NP* problem. Cook has proven this for a problem called *SAT*; for other problems, this can be proven using *reductions*.

In the rest of this section, we formally define *reducibility* and the complexity class *NP-complete*. Also, we provide some examples.

13.5.1 Reducibility

There are two definitions of reducibility, Karp and Turing; they are known to describe the same class of problems. The following definition uses Karp reducibility.

Definition 13.4. *A problem P_1 is said to reduce in polynomial time to problem P_2 (written as "$P_1 \propto P_2$") if there exists a polynomial-time algorithm A_1 for P_1 that makes calls to a subroutine solving P_2 and each call to a subroutine solving P_2 is counted as a single operation.*

We then say that P_2 is at least as hard as P_1. (Turing reductions allow for only *one* call to the subroutine.)

Important: In all of this course when we talk about reductions, we will always be referring to polynomial time reductions.

Theorem 13.5. *If $P_1 \propto P_2$ and $P_2 \in P$ then $P_1 \in P$*

Proof. Let the algorithm A_1 be the algorithm defined by the $P_1 \propto P_2$ reduction. Let A_1 run in time $O(p_1(|I_1|))$ (again, counting each of the calls to the algorithm for P_2 as one operation), where $p_1()$ is some polynomial and $|I_1|$ is the size of an instance of P_1.

Let algorithm A_2 be the poly-time algorithm for problem P_2, and assume this algorithm runs in $O(p_2(|I_2|))$ time.

The proof relies on the following two observations:

1. The algorithm A_1 can call at most $O(p_1(|I_1|))$ times the algorithm A_2. This is true since each call counts as one operation, and we know that A_1 performs $O(p_1(|I_1|))$ operations.

2. Each time the algorithm A_1 calls the algorithm A_2, it gives it an instance of P_2 of size at most $O(p_1(|I_1|))$. This is true since each bit of the created P_2 instance is either a bit of the instance $|I_1|$, or to create this bit we used at least one operation (and recall that A_1 performs $O(p_1(|I_1|)$ operations).

We conclude that the resulting algorithm for solving P_1 (and now counting all operations) performs at most $O(p_1(|I_1|) + p_1(|I_1|) * p_2((p_1(|I_1|))))$ operations. Since the multiplication and composition of polynomials is still a polynomial, this is a polynomial time algorithm. □

Corollary 13.6. *If $P_1 \propto P_2$ and $P_1 \notin P$ then $P_2 \notin P$*

13.5.2 NP-Completeness

A problem Q is said to be NP-hard if $B \propto Q \ \forall B \in NP$. That is, if all problems in NP are polynomially reducible to Q.

A decision problem Q is said to be NP-Complete if:
  1. $Q \in NP$, and
  2. Q is NP-Hard.

It follows from Theorem 13.5 that if *any NP*-complete problem were to have a polynomial algorithm, then all problems in NP would. Also it follows from Corollary 13.6 that is we prove that *any NP*-complete problem has no polynomial time algorithm, then this would prove that no NP-complete problem has a polynomial algorithm.

Note that when a decision problem is NP-Complete, it follows that its optimization problem is NP-Hard.

Conjecture: $P \neq NP$. So, we do not expect any polynomial algorithm to exist for an NP-complete problem.

Now we will show specific examples of NP-complete problems.

SAT = {Given a boolean function in conjunctive normal[3] form (CNF), does it have a satisfying assignment of variable? }

$$(X_1 \vee X_3 \vee \bar{X}_7) \wedge (X_{15} \vee \bar{X}_1 \vee \bar{X}_3) \wedge (\qquad) \ldots \wedge (\qquad)$$

Theorem 13.7 (Cook-Levin (1970))**.** *SAT is NP-Complete*

Proof.

SAT is in NP: Given the boolean formula and the assignment to the variables, we can substitute the values of the variables in the formula, and verify in linear time that the assignment indeed satisfies the boolean formula.

---

[3] In boolean logic, a formula is in conjunctive normal form if it is a conjunction of clauses (i.e. clauses are "linked by and"), and each clause is a disjunction of literals (i.e. literals are "linked by or"; a literal is a variable or the negation of a variable).

$B \propto SAT \ \forall B \in NP$ **:** The idea behind this part of the proof is the following: Let X be an instance of a problem $\mathcal{Q}$, where $\mathcal{Q} \in NP$. Then, there exists a SAT instance $F(X, \mathcal{Q})$, whose size is polynomially bounded in the size of X, such that $F(X, \mathcal{Q})$ is satisfiable if and only if X is a "yes" instance of $\mathcal{Q}$.

The proof, in very vague terms, goes as follows. Consider the process of verifying the correctness of a "yes" instance, and consider a Turing machine that formalizes this verification process. Now, one can construct a SAT instance (polynomial in the size of the input) mimicking the actions of the Turing machine, such that the SAT expression is satisfiable if and only if verification is successful.

$\square$

Karp[4] pointed out the practical importance of Cook's theorem, via the concept of reducibility.

To show that a problem $\mathcal{Q}$ is NP-complete, one need only demonstrate two things:

1. $\mathcal{Q} \in NP$, and
2. $\mathcal{Q}$ is NP-Hard. However, now to show this we only need to show that $\mathcal{Q}' \propto \mathcal{Q}$, for some NP-hard or NP-complete problem $\mathcal{Q}'$. (A lot easier than showing that $B \propto \mathcal{Q} \ \forall B \in NP$, right?)

Because all problems in NP are polynomially reducible to $\mathcal{Q}'$, if $\mathcal{Q}' \propto \mathcal{Q}$, it follows that $\mathcal{Q}$ is at least as hard as $\mathcal{Q}'$ and that all problems in NP are polynomially reducible to $\mathcal{Q}$.

Starting with SAT, Karp produced a series of problems that he showed to be NP-complete.[5]

To illustrate how we prove NP-Completeness using reductions, and that, apparently very different problems can be reduced to each other, we will prove that the Independent Set problem is NP-Complete.

Independent Set (IS) = {Given a graph $G = (V, E)$, does it have a subset of nodes $S \subseteq V$ of size $|S| \geq k$ such that: for every pair of nodes in S there is no edge in E between them?}

Theorem 13.8. *Independent Set is NP-complete.*

Proof.

IS is in NP: Given the graph $G = (V, E)$ and S, we can check in polynomial time that: 1) $|S| \geq k$, and 2) there is no edge between any two nodes in S.

IS is NP-Hard: We prove this by showing that $SAT \propto IS$.

i) Transform input for IS from input for SAT in polynomial time

Suppose you are given an instance of SAT, with k clauses. Form a graph that has a *component* corresponding to each clause. For a given clause, the corresponding component is a complete graph with one vertex for each variable in the clause. Now, connect

---

[4]Reference: R. M. Karp. *Reducibility Among Combinatorial Problems*, pages 85–103. Complexity of Computer Computations. Plenum Press, New York, 1972. R. E. Miller and J. W. Thatcher (eds.).

[5]For a good discussion on the theory of NP-completeness, as well as an extensive summary of many known NP-complete problems, see: M.R. Garey and D. S. Johnson. *Computers and Intractability: A Guide to the Theory of NP-completeness.* Freeman, San Francisco, 1979.

two nodes in different components if and only if they correspond to a variable and its negation. This is clearly a polynomial time reduction.

ii) Transform output for SAT from output for IS in polynomial time

If the graph has an independent set of size $\geq k$, then our the SAT formula is satisfiable.

iii) Prove the correctness of the above reduction

To prove the correctness we need to show that the resulting graph has an independent set of size at least k **if and only if** the SAT instance is satisfiable.

($\rightarrow$) [G has an independent set of size at least $k \rightarrow$ the SAT instance is satisfiable]

We create a satisfiable assignment by making true the literals corresponding to the nodes in the independent set; and we make false all other literals. That this assignment satisfies the SAT formula follows from the following observations:

1. The assignment is valid since the independent set may not contain a node corresponding to a variable and a node corresponding to the negation of this variable. (These nodes have an edge between them.)

2. Since 1) the independent set may not contain more than one node of each of the *components* of G, and 2) the value of the independent set is $\geq k$ (it is actually equal to k); then it follows that the independent set must contain exactly one node from each component. Thus every clause is satisfied.

($\leftarrow$) [the SAT instance is satisfiable $\rightarrow$ G has an independent set of size at least k]

We create an independent set of size equal to k by including in S one node from each clause. From each clause we include in S exactly one of the nodes that corresponds to a literal that makes this clause true. That we created an independent set of size k follows from the following observations:

1. The constructed set S is of size k since we took exactly one node "from each clause", and the SAT formula has k clauses.

2. The set S is an independent set since 1) we have exactly one node from each component (thus we can't have intra-component edges between any two nodes in S); and 2) the satisfying assignment for SAT is a valid assignment, so it cannot be that both a variable and its negation are true, (thus we can't have inter-component edges between any two nodes in S).

$\square$

An example of the reduction is illustrated in Figure 42 for the 3-SAT expression $(x_1 \vee x_2 \vee x_3) \wedge (\bar{x}_1 \vee \bar{x}_2 \vee x_4) \wedge (x_1 \vee \bar{x}_3 \vee x_4) \wedge (\bar{x}_2 \vee x_3 \vee \bar{x}_4)$.

<u>*0-1 Knapsack* is *NP*-complete.</u>

Proof. First of all, we know that it is in NP, as a list of items acts as the certificate and we can verify it in polynomial time.

Now, we show that *Partition* $\propto$ *0-1 Knapsack*. Consider an instance of *Partition*. Construct an instance of *0-1 Knapsack* with $w_i = c_i = b_i$ for $i = 1, 2, \ldots, n$, and $W = K = \frac{1}{2} \sum_{i=1}^{n} b_i$. Then, a solution exists to the *Partition* instance if and only if one exists to the constructed *0-1 Knapsack* instance.

Exercise: Try to find a reduction in the opposite direction. $\square$

<u>*k-center* is *NP*-complete.</u>
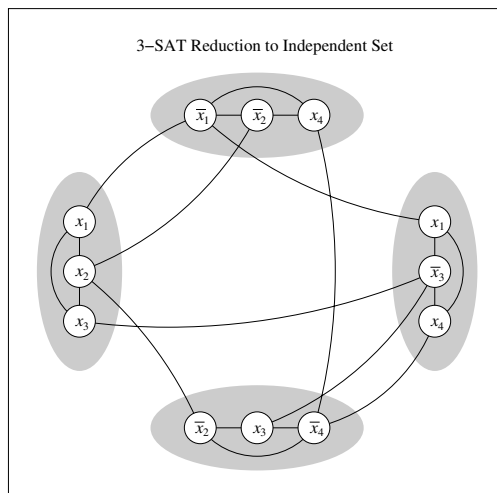
Figure 42: Example of 3-SAT reduction to Independent set

Proof. The *k-center* problem is in *NP* from Section 13.2. The *dominating set* problem is known to be *NP-complete*; therefore, so is *k-center* (from Section 13.5.1). □

About the Knapsack problem:

The Knapsack problem is:

Problem: Given a set of items, each with a weight and value (cost), determine the subset of items with maximum total weight and total cost less than a given budget.

The binary knapsack problem's mathematical programming formulation is as follows.

$$\max \quad \sum_{j=1}^{n} w_j x_j$$
$$\sum_{j=1}^{n} v_j x_j \leq B$$
$$x_j \in \{0,1\}$$

The problem can be solved by a longest path procedure on a graph – DAG– where for each $j \in \{1, \ldots, n\}$ and $b \in \{0, 1, \ldots, B\}$ we have a node. This algorithm can be viewed alternatively also as a dynamic programming with $f_j(b)$ the value of $\max \sum_{i=1}^{j} u_i x_i$ such that $\sum_{i=1}^{j} v_i x_i \leq b$. This is computed by the recursion:

$$f_j(b) = \max\{f_{j-1}(b); f_{j-1}(b - v_j) + u_j\}.$$

There are obvious boundary conditions,

$$f_1(b) = \begin{cases} 0 & \text{if } b \leq v_1 - 1 \\ u_1 & \text{if } b \geq v_1 \end{cases}$$

The complexity of this algorithm is $O(nB)$. So if the input is represented in *unary* then the input length is a polynomial function of n and b and this run time is considered polynomial. The knapsack problem, as well as any other *NP*-complete problem that has a poly time algorithm for unary input is called *weakly NP-complete*.

14 Approximation algorithms

Approximation algorithms have developed in response to the impossibility of solving a great variety of important optimization problems. Too frequently, when attempting to get a solution for a problem, one is confronted with the fact that the problem is *NP*-hard.

If the optimal solution is unattainable then it is reasonable to sacrifice optimality and settle for a 'good' feasible solution that can be computed efficiently. Of course we would like to sacrifice as little optimality as possible, while gaining as much as possible in efficiency.

Approaches to deal with *NP*-hard problems:

Integer Programming Integer programming tools are forms of implicit enumeration algorithms that combine efficient derivations of lower and upper bounds with a hopeful search for an optimal solution. The amount of time required to solve even a typical moderate-size optimization problem is exorbitant. Instead, the user interrupts the enumeration process either when the current solution is deemed satisfactory, or when the running time has exceeded reasonable limit. The point is that integer programming algorithms provide no *guarantee*. It is impossible to tell if 5 additional minutes of running time would get you a significantly better solution, or if 5 more days of running time would yield no improvement at all.

Heuristics 'Heuristics' is the nomenclature for rules-of-thumb which in itself is a euphemism for simple and invariably polynomial algorithms. Heuristics work quickly and efficiently. The quality of the solution they deliver is another matter altogether: heuristics provide no way to know how close (or far) is the solution found with respect to the optimal solution.

Metaheuristics Metaheuristics are heuristic methods that can be applied to a very general class of computational problems. Some examples are: simulated annealing, genetic algorithms, tabu search, ant colony algorithms, etc. As hueristics, metaheuristics provide no way to know how close (or far) is the solution found with respect to the optimal solution.

Approximation algorithms An approximation algorithm is necessarily polynomial, and is evaluated by the worst case possible relative error over all possible instances of the problem.

Definition 14.1. *A polynomial algorithm, $\mathcal{A}$, is said to be a δ-approximation algorithm for a minimization problem P if for every problem instance I of P with an optimal solution value $OPT(I)$, it delivers a solution of value $\mathcal{A}(I)$ satisfying,*

$$\mathcal{A}(I) \leq \delta \, OPT(I).$$

Naturally, $\delta > 1$ and the closer it is to 1, the better. Similarly, for maximization problems a δ-approximation algorithm delivers for every instance I a solution that is at least δ times the optimum.

An alternative, and equivalent, definition of approximation algorithms is the following. For a minimization problem, a δ-*approximation algorithm* is a polynomial algorithm, such that for every instance I, the algorithm delivers a value $Alg(I)$ with $Opt(I) \leq Alg(I) \leq \delta \cdot Opt(I)$, where $Opt(I)$ is the optimal solution value. i.e.

$$\text{Sup}_I \left\{ \frac{Alg(I)}{Opt(I)} \right\} \leq \delta$$

An algorithm that is a δ-approximation, guarantees relative error $\leq \delta - 1$:

$$\frac{Alg(I) - Opt(I)}{Opt(I)} \leq \delta - 1 \ .$$

For a maximization problem, $Opt(I) \geq Alg(I) \geq \delta \cdot Opt(I)$, $0 \leq \delta \leq 1$.

14.1   Traveling salesperson problem (TSP)

The Travelling Salesperson Problem (TSP for short) is to visit a given collection of cities (or nodes in a graph) once each and return to the home base while traversing the shortest possible tour. The problem is a well known and studied intractable problem. Here we consider the TSP with triangle inequality, or Δ-inequality satisfied (weights on edges satisfying $\forall_{ijk}$, $w_{ij} + w_{jk} \geq w_{ik}$). Even with triangle inequality the problem is still NP-hard. However, this problem has a relatively easy 2-approximation algorithm based on the generation of a Minimum Spanning Tree (MST). A minimum spanning tree can be found in a graph polynomially and very efficiently (we will cover this problem in a future lecture). Recall that a graph is Eulerian (or has an Eulerian tour) if and only if every node in the graph has even degree. The Eulerian tour in an Eulerian graph can be constructed quickly - in linear time.

MST-Algorithm:

Step 1: Find a minimum spanning tree (MST) in the graph.

Step 2: "Double" the edges of the spanning tree (to get a graph in which each node has even degree). Construct an Eulerian tour.

Step 3: Construct a valid tour that visits each node exactly once using "short-cuts".

Since the Δ-inequality is satisfied, the shortcut tour is no longer than the Eulerian tour, the length of which is no more than two times the length of the minimum spanning tree (denoted by $|MST|$). Also $|MST| \leq |TSP|$ for $|TSP|$ denoting the length of the optimal tour, because a tour is a spanning tree with one additional edge:

$$\Rightarrow |MST| \leq |TSP| \leq 2|MST| \ .$$

Now,
$$|\text{MST-Algorithm}(I)| \leq 2|MST| \leq 2|TSP| = 2OPT(I)$$

$\Rightarrow$ MST-Algorithm is a 2-approximation algorithm.

A more subtle scheme for approximating TSP with Δ-inequality is due to Christofides (1978). That algorithm uses the fact that some nodes in the spanning tree are of even degree and doubling the edges adjacent to these nodes is superfluous and wasteful. Instead the algorithm due to Christofides works by adding only the bare minimum necessary additional edges to produce an Eulerian graph, thus delivering a better approximation, 3/2-approximation.

Christofides 1.5-approximation Algorithm:

Step 1: Find a minimum spanning tree (MST) in the graph. Let the set of odd degree vertices in MST be V^{odd}.

Step 2: Find the minimum weight perfect matching induced on the set of nodes V^{odd}, M^*. Add the edges of M^* to the edges of MST to create a graph in which each node has even degree. Construct an Eulerian tour.

Step 3: Construct a valid tour that visits each node exactly once using "short-cuts".

We know that the weight of the edges in MST form a lower bound on the optimum. We now show that the weight of the edges in M^* forms a lower bound to $\frac{1}{2}|TSP|$. To see that observe that the optimal tour can be shortcut and restricted to the set of nodes V^{odd} without an increase in cost. Now V^{odd} contains an even number of vertices. An even length tour on V^{odd} can be partitioned to two perfect matchings M_1 and M_2 by taking the alternate set of edges in one, and the remaining alternate set in the other. We thus have

$$|M_1|, \ |M_2| \geq |M^*|.$$

Thus, $|TSP| \geq |M_1| + |M_2| \geq 2|M^*|$ and $|M^*| \leq \frac{1}{2}|TSP|$.

Therefore the algorithm delivered by Christofides' algorithm delivers a tour of length $|MST| + |M^*| \leq \frac{3}{2}|TSP|$.

15 Integer programs with two variables per inequality (IP2)

15.1 A practical example: open-pit mining

Open-pit mining is a surface mining operation in which blocks of earth are extracted from the surface to retrieve the ore contained in them. During the mining process, the surface of the land is being continuously excavated, and a deeper and deeper pit is formed until the operation terminates. The final contour of this pit mine is determined before mining operation begins. (Figure 43) To

Figure 43: Open pit mining
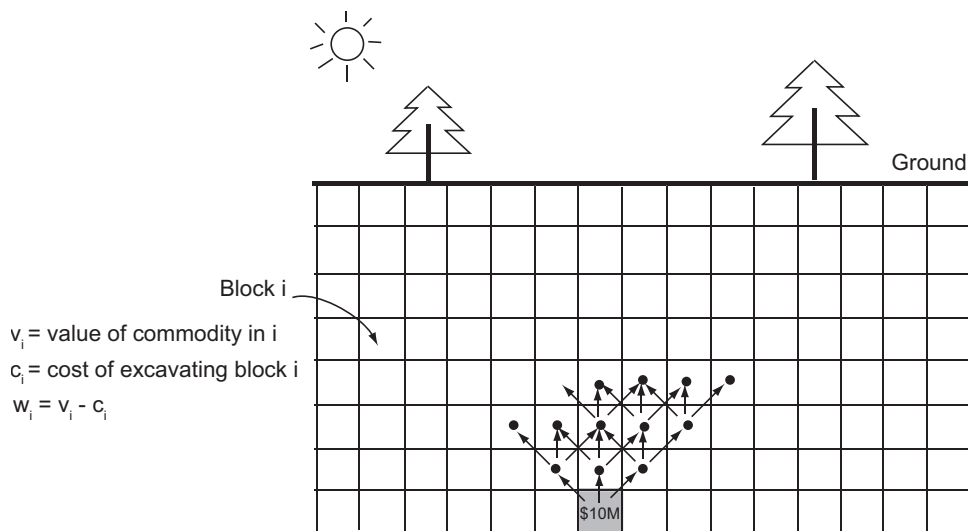
design the optimal pit – one that maximizes profit, the entire area is divided into blocks, and the value of the ore in each block is estimated by using geological information obtained from drill cores. Each block has a weight associated with it, representing the value of the ore in it, minus the cost involved in removing the block. While trying to maximize the total weight of the blocks

to be extracted, there are also contour constraints that have to be observed. These constraints specify the slope requirements of the pit and precedence constraints that prevent blocks from being mined before others on top of them. Subject to these constraints, the objective is to mine the most profitable set of blocks.

The problem can be represented on a directed graph $G = (V, A)$. Each block i corresponds to a node with weight w_i representing the net value of the individual block. The net value w_i is computed as the assessed value v_i of the ore in that block, from which the cost c_i of excavating that block alone is deducted. There is a directed arc from node i to node j if block i cannot be excavated before block j which is on a layer right above block i. This precedence relationship is determined by the engineering slope requirements. Suppose block i cannot be excavated before block j, and block j cannot be excavated before block k. By transitivity this implies that block i cannot be excavated before block k. We choose in this presentation not to include the arc from i to k in the graph and the existence of a directed path from i to k implies the precedence relation. Including only arcs between immediate predecessors reduces the total number of arcs in the graph. Thus to decide which blocks to excavate in order to maximize profit is equivalent to finding a maximum weighted set of nodes in the graph such that all successors of all nodes are included in the set.

Notice that the problem is trivial if $w_i \leq 0, \forall i \in V$ (in which no block would be excavated) or if $w_i \geq 0, \forall i \in V$ (in which all the blocks would be excavated).

Let $x_i = \begin{cases} 1 & \text{if block } i \text{ is selected} \\ 0 & \text{otherwise} \end{cases}$.

Then the open pit mining problem can be formulated as follows:

$$
\begin{aligned}
\max \quad & \sum_{i \in V} w_i x_i \\
\text{s.t.} \quad & x_i \leq x_j \quad \forall (i, j) \in A \\
& 0 \leq x_i \leq 1 \quad \forall i \in V
\end{aligned}
$$

Notice that each inequality constraint contains exactly one 1 and one -1 in the coefficient matrix. The constraint matrix is totally unimodular. Therefore, we do not need the integrality constraints.

The following website `http://riot.ieor.berkeley.edu/riot/Applications/OPM/OPMInteractive.html` offers an interface for defining, solving, and visualizing the open pit mining problem.

The open-pit mining problem is a special case of the *maximum closure problem*. The next subsection discusses the maximum closure problem in detail.

15.2   The maximum closure problem

We begin by defining a closed set on a graph.

Definition 15.1. *Given a directed graph $G = (V, A)$, a subset of the nodes $D \subseteq V$ is* closed, *if for every node in D, its successors are also in D.*
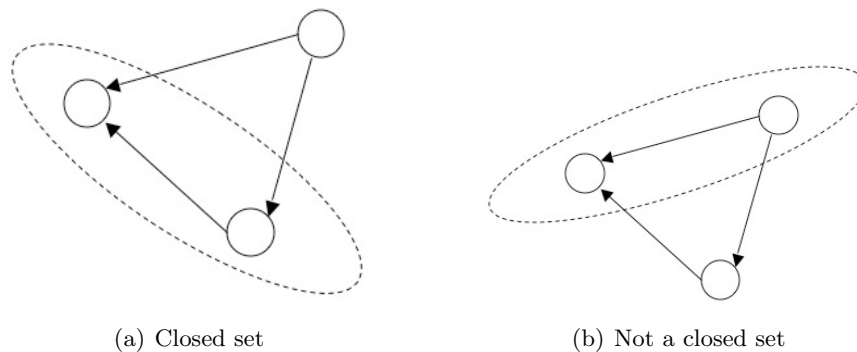
(a) Closed set   (b) Not a closed set

Figure 44: Closed set v.s. unclosed set

Consider a directed graph $G = (V, A)$ where every node $i \in V$ has a corresponding weight w_i. The *maximum closure problem* is to find a closed set $V' \subseteq V$ with maximum total weight. That is, the maximum closure problem is:

Problem Name: *Maximum closure*

Instance: *Given a directed graph $G = (V, A)$, and node weights (positive or negative) w_i for all $i \in V$.*

Optimization Problem: *find a closed subset of nodes $V' \subseteq V$ such that $\sum_{i \in V'} w_i$ is maximum.*

We can formulate the maximum closure problem as an integer linear program (ILP) as follows.

$$
\begin{aligned}
\max \quad & \sum_{i \in V} w_i x_i \\
\text{s.t.} \quad & x_i \leq x_j \quad \forall (i, j) \in A \\
& x_i \in \{0, 1\} \quad \forall i \in V
\end{aligned}
$$

where x_i is a binary variable that takes the value 1 if node i is in the maximum closure, and 0 otherwise. The first set of constraints imposes the requirement that for every node i included in the set, its successor is also in the set. Observe that since every row has exactly one 1 and one -1, the constraint matrix is totally unimodular (TUM). Therefore, its linear relaxation formulation results in integer solutions. Specifically, this structure also indicates that the problem is the dual of a flow problem.

Johnson [Joh68] seems to be the first researcher who demonstrated the connection between the maximum closure problem and the selection problem (i.e., maximum closure on bipartite graphs), and showed that the selection problem is solvable by max flow algorithm. Picard [Pic76], demonstrated that a minimum cut algorithm on a related graph, solves the maximum closure problem.

Let $V^+ \equiv \{j \in V | w_i > 0\}$, and $V^- \equiv \{i \in V | w_i \leq 0\}$. We construct an s, t-graph G_{st} as follows. Given the graph $G = (V, A)$ we set the capacity of all arcs in A equal to ∞. We add a source s, a sink t, set A_s of arcs from s to all nodes $i \in V^+$ (with capacity $u_{s,i} = w_i$), and set A_t of arcs from all nodes $j \in V^-$ to t (with capacity $u_{j,t} = |w_j| = -w_j$). The graph $G_{st} = \{V \cup \{s, t\}, A \cup A_s \cup A_t\}$ is a *closure graph* (a closure graph is a graph with a source, a sink, and with all finite capacity arcs adjacent only to either the source or the sink.) This construction is illustrated in Figure 45.
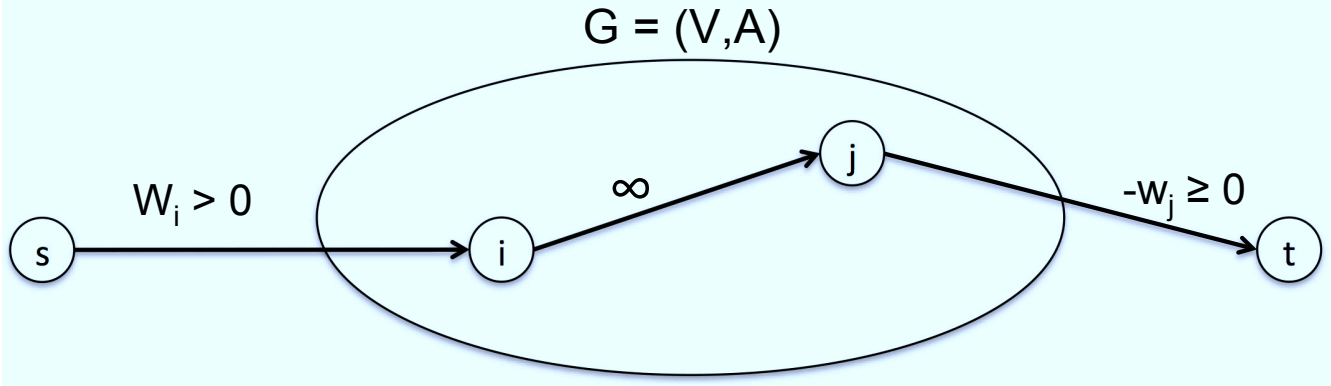
Figure 45: Visual representation of G_{st}.

Claim 15.2. *If $(s \cup S, t \cup T)$ is a finite $s-t$ cut on $G_{s,t}$, then S is a closed set on G.*

Proof. Assume by contradiction that S is not closed. This means that there must be an arc $(i,j) \in A$ such that $i \in S$ and $j \in T$. This arc must be on the cut (S,T), and by construction $u_{i,j} = \infty$, which is a contradiction on the cut being finite. $\square$

Theorem 15.3. *If $(s \cup S, t \cup T)$ is an optimal solution to the minimum $s-t$ cut problem on $G_{s,t}$, then S is a maximum closed set on G.*

Proof.

$$
\begin{aligned}
C\left(s \cup S, t \cup T\right) &= \sum_{(s,i) \in A_{st},\ i \in T} u_{s,i} + \sum_{(j,t) \in A_{st},\ j \in S} u_{j,t} \\
&= \sum_{i \in T \cap V^+} w_i + \sum_{j \in S \cap V^-} -w_j \\
&= \sum_{i \in V^+} w_i - \sum_{i \in S \cap V^+} w_i - \sum_{j \in S \cap V^-} w_j \\
&= W^+ - \sum_{i \in S} w_i
\end{aligned}
$$

(Where $W^+ = \sum_{i \in V^+} w_i$, which is a constant.) This implies that minimizing $C\left(s \cup S, t \cup T\right)$ is equivalent to minimizing $W^+ - \sum_{i \in S} w_i$, which is in turn equivalent to $\max_{S \subseteq V} \sum_{i \in S} w_i$. Therefore, any source set S that minimizes the cut capacity also maximizes the sum of the weights of the nodes in S. Since by Claim 15.2 any source set of an $s-t$ cut in $G_{s,t}$ is closed, we conclude that S is a maximum closed set on G. $\square$

Variants/Special Cases

- In the minimum closure problem we seek to find a closed set with minimum total weight. This can be solved by negating the weights on the nodes in G to obtain G^-, constructing $G_{s,t}^-$ just as before, and solving for the maximum closure. Under this construction, the source set of a minimum $s-t$ cut on $G_{s,t}$ is a minimum closed set on G. See Figure 46 for a numerical example.

- The selection problem is a special case of the maximum closure problem.

Numerical Example

(a) Minimum closure problem

(b) Negated graph

(c) Modified (max flow) graph

(d) Source of min cut = min closure set on G
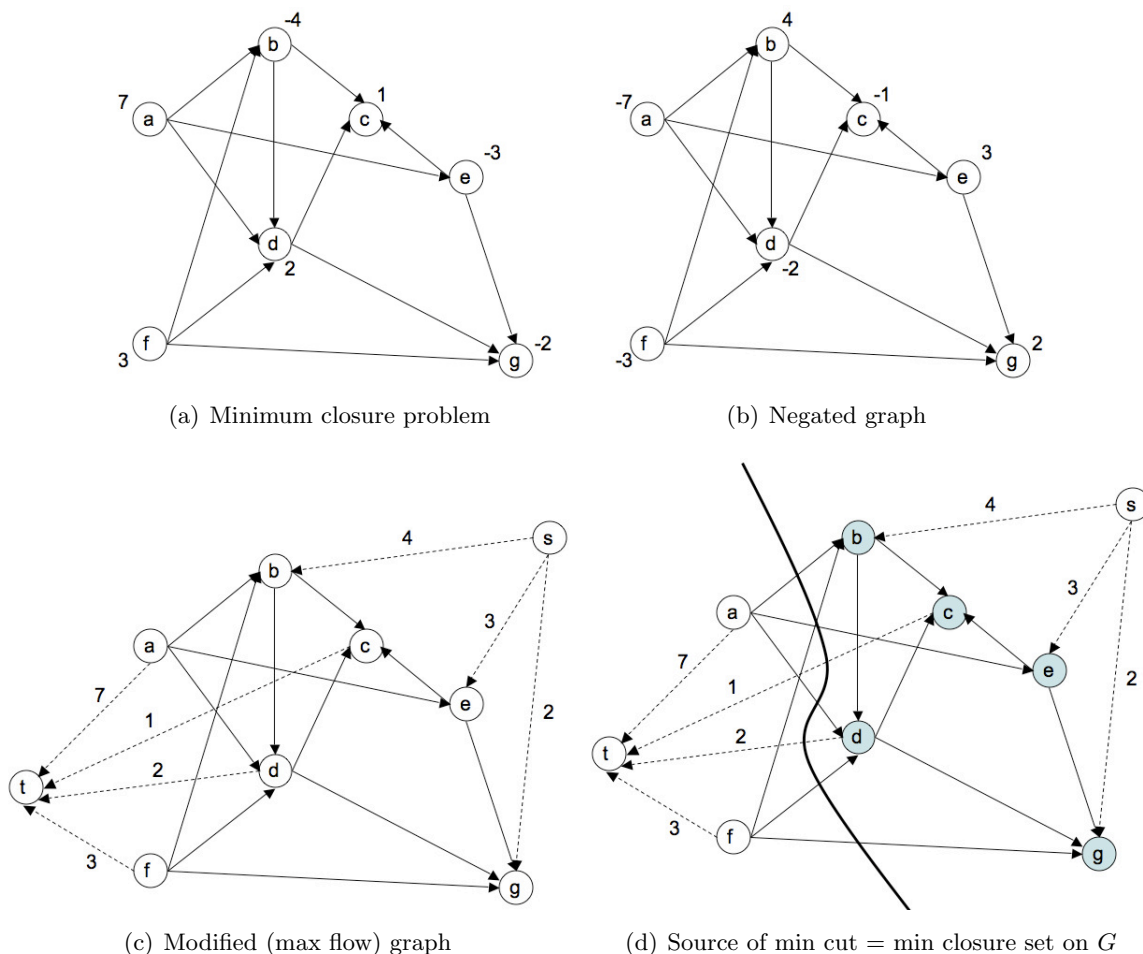
Figure 46: Converting a minimum closure problem to a maximum closure problem. Under this transformation, the source set of a minimum cut is also a minimum closed set on the original graph.

15.3 Monotone IP2

We first focus on integer programming with monotone inequalities (Monotone IP2).

Definition 15.4. *An inequality is said to be* monotone *if it is of the form* $ax - by \geq c$ *or* $ax - by \leq c$, *where* x, y *are variables and* a *and* b *of the same sign.*

A typical Monotone IP2 is shown below:

$$\text{(Monotone IP2)} \quad \max \quad \sum_{i \in V} w_i x_i$$

$$\text{s.t.} \quad a_{ij} x_i - b_{ij} x_j \geq c_{ij} \quad \forall (i,j) \in E$$

$$l_i \leq x_i \leq u_i, \ integer \quad \forall i \in V$$

where $a_{ij} \geq 0, b_{ij} \geq 0, \forall (i,j) \in E$.

Note that assuming a_{ij}, b_{ij} in all monotone inequalities of the problem to be positive and the inequality relationship to be "$\geq$" does not impair the generality of Monotone IP2. It is easy to see that monotone inequality $ax - by \leq c$ can be transformed into $by - ax \geq -c$, and monotone inequality $ax - by \geq c$ with $a \leq 0, b \leq 0$ can be transformed into $(-b)y - (-a)x \geq c$ with $-b \geq 0, -a \geq 0$.

Monotone IP2 is *NP* -hard. This was proved by Lagarias [Lag85] for the simultaneous diophantine approximation problem which is a special case of monotone IP2. However Hochbaum and Naor showed, in [HN94], that monotone IP2 is solved in pseudopolynomial time that depends on the the range of the variables only. This means that monotone IP2 is *weakly* NP-hard. Note that for all interesting applications the variables have small range - typically the variables are binary. Hence the procedure described for monotone IP2 runs in polynomial time for small range variables. We next describe how to solve Monotone IP2 as a maximum closure problem.

We first make a change of variables in our original problem. In particular we write variable x_i as a summation of binary variables $x_i = l_i + \sum_{p=l_i+1}^{u_i} x_i^{(p)}$, and impose the restriction that $x_i^{(p)} = 1 \implies x_i^{(p-1)} = 1$ for all $p = l_i + 1, \ldots, u_i$.

With this change of variables we rewrite Monotone IP2 as follows:

$$\max \quad \sum_{i \in V} w_i l_i + \sum_{i \in V} w_i \sum_{p=l_i+1}^{u_i} x_i^{(p)} \tag{16a}$$

$$\text{s.t.} \quad x_j^{(p)} \leq x_i^{(q(p))} \quad \forall \, (i,j) \in E \quad \text{for } p = l_j + 1, \ldots, u_j \tag{16b}$$

$$x_i^{(p)} \leq x_i^{(p-1)} \quad \forall \, i \in V \quad \text{for } p = l_i + 1, \ldots, u_i \tag{16c}$$

$$x_i^{(p)} \in \{0, 1\} \quad \forall \, i \in V \quad \text{for } p = l_i + 1, \ldots, u_i, \tag{16d}$$

where $q(p) \equiv \left\lceil \frac{c_{ij} + b_{ij} p}{a_{ij}} \right\rceil$.

Inequality (16c) guarantees the restriction that $x_i^{(p)} = 1 \implies x_i^{(p-1)} = 1$ for all $p = l_i + 1, \ldots, u_i$; Inequality (16b) follows from the monotone inequalities in the original problem. In particular, for any monotone inequality $a_{ij} x_i - b_{ij} x_j \geq c_{ij}$ with $a_{ij}, b_{ij} \geq 0$, $a_{ij} x_i - b_{ij} x_j \geq c_{ij} \iff x_i \geq \frac{c_{ij} + b_{ij} x_j}{a_{ij}} \overset{x_i \text{ integer}}{\implies} x_i \geq \left\lceil \frac{c_{ij} + b_{ij} x_j}{a_{ij}} \right\rceil$. Equivalently, if $x_j \geq p$, we must have $x_i \geq q(p) = \left\lceil \frac{c_{ij} + b_{ij} p}{a_{ij}} \right\rceil$. In terms of the newly defined binary variables, this is further equivalent to $x_j^p = 1 \implies x_i^{q(p)} = 1$, i.e., $x_j^{(p)} \leq x_i^{(q(p))}$.

Now it's obvious that Monotone IP2 is the maximum closure problem of an s, t-Graph G_{st} defined as follows. First still define $V^+ \equiv \{i \in V | w_i > 0\}$ and $V^- \equiv \{j \in V | w_j \leq 0\}$ as before, and the sets of nodes and arcs are constructed below.

Set of nodes:

Add a source s, a sink t, and nodes $x_i^{(l_i)}, x_i^{(l_i+1)}, \ldots, x_i^{(u_i)}$ for each $i \in V$.

Set of arcs:

1) For any $i \in V^+$, connect s to $x_i^{(p)}, p = l_i + 1, \ldots, u_i$ by an arc with capacity w_i.

2) For any $i \in V^-$, connect $x_i^{(p)}, p = l_i + 1, \ldots, u_i$ to t by an arc with capacity $|w_i| = -w_i$.

3) For any $i \in V$, connect $x_i^{(p)}$ to $x_i^{(p-1)}$, $p = l_i + 1, \ldots, u_i$, by an arc with capacity ∞, and connect s to $x_i^{(l_i)}$ with an arc with capacity ∞.

4) For any $(i,j) \in E$, connect $x_j^{(p)}$ to $x_i^{q(p)}$ by an arc with capacity ∞ for all $p = l_i + 1, \ldots, u_j$. (Note that for situations where $q(p) > u_i$, we must have $x_j^{(p)} = 0$. Therefore, we can either remove

the node $x_j^{(p)}$ by redefining a tighter upper bound for x_j, or simply fix $x_j^{(p)}$ to be zero by introducing an arc from $x_j^{(p)}$ to t with capacity ∞.)

This construction is illustrated in Figure 47.
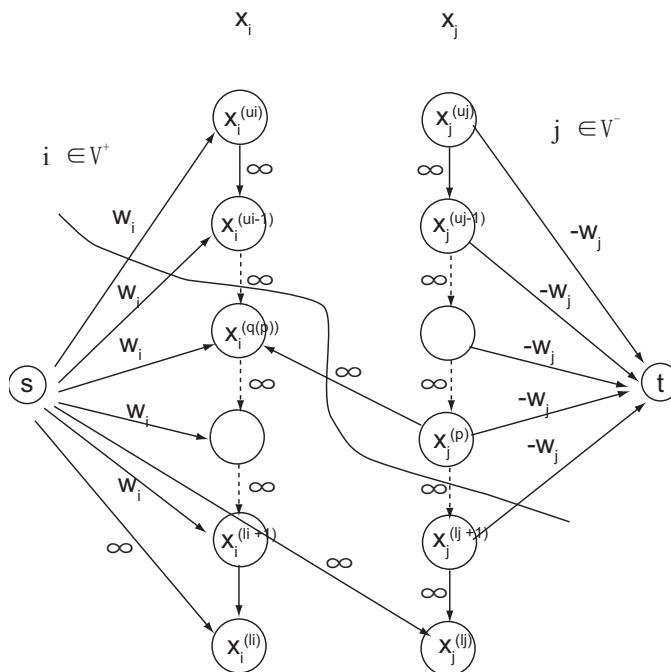
Figure 47: Illustration of G_{st} for Monotone IP2 and example of a finite $s-t$ cut

Remarks:

- It should be noted that the maximum closure problem is defined on an s, t-Graph with $2 + \sum_{i \in V}(u_i - l_i)$ nodes. The size of the graph is not a polynomial function of the length of the input. Therefore, the original Monotone IP2 is weakly NP-hard and can be solved by pseudo-polynomial time algorithm based on the construction of a maximum closure problem.

- For the min version of Monotone IP2, we can construct the s, t-Graph in the same way and define closure with respect to the sink set instead.

15.4 Non–monotone IP2

Now let's look at the non–monotone integer program with two variables per inequality:

$$\text{(IP2)} \quad \min \quad \sum_{i \in V} w_i x_i$$
$$\text{s.t.} \quad a_{ij} x_i + b_{ij} x_j \geq c_{ij} \quad \forall (i, j) \in E$$
$$l_i \leq x_i \leq u_i, \; integer \quad \forall i \in V$$

This problem is more general than Monotone IP2, as we don't impose any restrictions on the signs of a_{ij} and b_{ij}. The problem is clearly NP-hard, since vertex cover is a special case. We now give a 2-approximation algorithm for it.

We first "monotonize" IP2 by replacing each variable x_i in the objective by $x_i = \frac{x_i^+ - x_i^-}{2}$ where $l_i \leq x_i^+ \leq u_i$ and $-u_i \leq x_i^- \leq -l_i$.

Each non-monotone inequality $a_{ij}x_i + b_{ij}x_j \leq c_{ij}$ is replaced by the following two inequalities:

$$a_{ij}x_i^+ - b_{ij}x_j^- \leq c_{ij}$$
$$-a_{ij}x_i^- + b_{ij}x_j^+ \leq c_{ij}$$

And each monotone inequality $a'_{ij}x_i - b'_{ij}x_j \leq c'_{ij}$ is replaced by the two inequalities:

$$a'_{ij}x_i^+ - b'_{ij}x_j^+ \leq c_{ij}$$
$$-a'_{ij}x_i^- + b'_{ij}x_j^- \leq c_{ij}$$

With such transformations we get the following problem:

$$(\text{IP2'}) \min \quad \frac{1}{2}\left(\sum_{i \in V} w_i x_i^+ + \sum_{i \in V}(-w_i)x_i^- \right)$$

$$\text{s.t.} \quad a_{ij}x_i^+ - b_{ij}x_j^- \leq c_{ij} \quad \forall\, (i,j) \in E$$
$$-a_{ij}x_i^- + b_{ij}x_j^+ \leq c_{ij} \quad \forall\, (i,j) \in E$$
$$l_i \leq x_i^+ \leq u_i, \; integer \quad \forall\, i \in V$$
$$-u_i \leq x_i^- \leq -l_i, \; integer \quad \forall\, i \in V$$

Let's examine the relationship between IP2 and IP2'. Given any feasible solution $\{x_i\}_{i \in V}$ for IP2, we can construct a corresponding feasible solution for IP2' with the same objective value, by letting $x_i^+ = x_i, x_i^- = -x_i, i \in V$; On the other hand, given any feasible solution $\{x_i^+, x_i^-\}_{i \in V}$ for IP2', $\{x_i = \frac{x_i^+ - x_i^-}{2}\}_{i \in V}$ satisfies the inequality constraint in IP2 (To see this, simply sum up the two inequality constraints in IP2' and divide the resulted inequality by 2.) but may not be integral. Therefore, IP2' provides a lower bound for IP2.

Since IP2' is a monotone IP2, we can solve it in integers (we can reduce it to maximum closure, which in turn reduces to the minimum s,t-cut problem). Its solution is a lower bound on IP2. In general it is not trivial to round the solution obtained and get an integer solution for IP2. However we can prove that there always exists a way to round the variables and get our desired 2-approximation.

15.5 2-approximations for integer programs with two variables per inequality

We have now established that for any integer program on two variables per inequality it is possible to generate a superoptimal half integral solution in polynomial time. The following was shown in [HNMT93, Hoc97].

15.5.1 The quality of the bound provided by the monotonized system

Given a system of inequalities with two variables per inequality, let the set of feasible solutions for this system be
$$S = \{\mathbf{x} \in \mathcal{R}^n \mid \mathcal{A}\mathbf{x} \leq \mathbf{c}\},$$

and the feasible solutions to the monotone system resulting from the transformation above,

$$S^{(2)} = \{(\mathbf{x}^+, \mathbf{x}^-) \mid \mathcal{A}^{(2)}(\mathbf{x}^+, \mathbf{x}^-) \le \mathbf{c}^{(2)} \ , \ \mathbf{x}^+, \mathbf{x}^- \in \mathcal{R}^n\} \ .$$

If $\mathbf{x} \in S$, $\mathbf{x}^+ = \mathbf{x}$, and $\mathbf{x}^- = -\mathbf{x}$, then $(\mathbf{x}^+, \mathbf{x}^-) \in S^{(2)}$. So, for every feasible solution in S, there exists a feasible solution in $S^{(2)}$. Conversely, if $(\mathbf{x}^+, \mathbf{x}^-) \in S^{(2)}$, then $\mathbf{x}^{(2)} = \frac{1}{2}(\mathbf{x}^+ - \mathbf{x}^-) \in S$. Hence, for every feasible solution in $S^{(2)}$, there is a feasible solution in S.

Let $S_I = \{\mathbf{x} \in S \mid \mathbf{x} \text{ integer }\}$, and let

$$S_I^{(2)} = \left\{ \frac{1}{2}(\mathbf{x}^+ - \mathbf{x}^-) \mid (\mathbf{x}^+, \mathbf{x}^-) \in S^{(2)} \text{ and } \mathbf{x}^+, \mathbf{x}^- \text{ integer} \right\} \ .$$

If $\mathbf{x} \in S_I$, then $\mathbf{x} \in S_I^{(2)}$. Thus, $S_I \subseteq S_I^{(2)} \subseteq S$.

In fact, the set of solutions $S_I^{(2)}$ is even smaller than the set of feasible solutions that are integer multiples of $\frac{1}{2}$. To see that, let

$$S^{(\frac{1}{2})} = \{\mathbf{x} \mid \mathcal{A}\mathbf{x} \le \mathbf{c} \text{ and } \mathbf{x} \in \frac{1}{2}Z^n\} \ .$$

The claim is that $S_I^{(2)} \subset S^{(\frac{1}{2})}$, and $S^{(\frac{1}{2})}$ may contain points not in $S_I^{(2)}$. The following example illustrates such a case:

$$5x + 2y \le 6$$
$$0 \le x, y \le 1 \ .$$

Obviously, $(x = 1, \ y = \frac{1}{2})$ is a feasible solution in $S^{(\frac{1}{2})}$. But there is no corresponding integer solution in $S_I^{(2)}$ as $x^+ = -x^- = 1$ implies that $y^+ = y^- = 0$. It follows that the bound derived from optimizing over $S_I^{(2)}$ is tighter than a bound derived from optimizing over $S^{(\frac{1}{2})}$. Not only is this latter optimization weaker, but it is also in general NP-hard as stated in the following Lemma (proved in [HNMT93]).

Lemma 15.5. *Minimizing over a system of inequalities with two variables per inequality for $\mathbf{x} \in \frac{1}{2} \cdot Z^n$, is NP-hard.*

Proof. To see that this is NP-hard, we use a reduction from the vertex cover problem proposed by O. Goldschmidt. Consider a vertex cover problem with nonnegative weights of nodes,

$$
\begin{aligned}
\text{(VC)} \quad \min \quad & \sum_{i \in V} w_i x_i \\
\text{s.t.} \quad & x_i + x_j \ge 1 \quad \forall \, [i,j] \in E \\
& 0 \le x_j \le 1 \quad x_j \text{ integer} \quad j \in V.
\end{aligned}
$$

Due to the nonnegativity of w_j, the problem does not change if we increase the bounds to 2:

$$
\begin{aligned}
\text{(VC(2))} \quad \min \quad & \sum_{i \in V} w_i x_i \\
\text{s.t.} \quad & x_i + x_j \ge 1 \quad \forall \, [i,j] \in E \\
& 0 \le x_j \le 2 \quad x_j \text{ integer} \quad j \in V.
\end{aligned}
$$

Using the substitution $y_j = \frac{1}{2}x_j$, it is evident that this problem is equivalent to the following problem in which the integrality requirement is replaced by the restriction that the variables are integer multiples of $\frac{1}{2}$:

$$\begin{aligned} \min \quad & \sum_{i \in V} w_i y_i \\ \text{s.t.} \quad & 2y_i + 2y_j \geq 1 \quad \forall \, [i,j] \in E \\ & 0 \leq y_j \leq 1 \quad y_j \text{ is an integer multiple of } \frac{1}{2} \quad j \in V. \end{aligned}$$

Hence, to solve in integer multiples of $\frac{1}{2}$ is at least as difficult as to solve the vertex cover problem.

$\square$

15.5.2 Computing an approximate solution

Here we show how to obtain a 2-approximation for the optimum of a bounded integer program with two variables per inequality in $O(mnU^2 \log(Un^2/m))$ time. Assume that the given integer program has a feasible integer solution denoted by $z_1, \ldots, z_n$. (This can be tested in polynomial time).

We first transform the integer program into a monotone integer system and compute an optimal solution for the monotone system as in the previous section. For every variable x_i $(i = 1, \ldots, n)$, let m_i^+ and m_i^- denote the respective values of x_i^+ and x_i^- in the optimal solution of the monotone system. For $i = 1, \ldots, n$, let $m_i^* = \frac{1}{2}(m_i^+ - m_i^-)$. We define the following solution vector, denoted by $\boldsymbol{\ell} = (\ell_1, \ldots, \ell_n)$, where for $i = 1, \ldots, n$:

$$\ell_i = \begin{cases} \min\{m_i^+, -m_i^-\} & \text{if } z_i \leq \min\{m_i^+, -m_i^-\}, \\ z_i & \text{if } \min\{m_i^+, -m_i^-\} \leq z_i \leq \max\{m_i^+, -m_i^-\}, \\ \max\{m_i^+, -m_i^-\} & \text{if } z_i \geq \max\{m_i^+, -m_i^-\}. \end{cases} \tag{17}$$

Lemma 15.6. *The vector $\boldsymbol{\ell}$ is a feasible solution of the given integer program.*

Proof. Let $ax_i + bx_j \geq c$ be an inequality where a and b are nonnegative. We check all possible cases. If ℓ_i is equal to z_i or $\min\{m_i^+, -m_i^-\}$, and ℓ_j is equal to z_j or $\min\{m_j^+, -m_j^-\}$, then clearly,

$$a\ell_i + b\ell_j \geq az_i + bz_j \geq c .$$

Suppose $\ell_i \geq z_i$ and $\ell_j = \max\{m_j^+, -m_j^-\}$. By construction, we know that

$$am_i^+ - bm_j^- \geq c \quad \text{and} \quad -am_i^- + bm_j^+ \geq c .$$

If $\ell_i \geq -m_i^-$, then,

$$a\ell_i + b\ell_j \geq -am_i^- + bm_j^+ \geq c .$$

Otherwise,

$$a\ell_i + b\ell_j \geq am_i^+ - bm_j^- \geq c .$$

The last case is when $\ell_i = \max\{m_i^+, -m_i^-\}$, and $\ell_j = \max\{m_j^+, -m_j^-\}$. In this case,

$$a\ell_i + b\ell_j \geq am_i^+ - bm_j^- \geq c .$$

The other types of inequalities are handled similarly. $\square$

We showed that vector $\boldsymbol{\ell}$ is a feasible solution. We now argue that it also approximates the optimum.

Theorem 15.7.

1. *The vector $\boldsymbol{\ell}$ is a 2-approximate solution of the bounded integer program.*

2. *The value of the objective function at the vector $\mathbf{m}^*$ is at least a half of the value of the objective function of the best integer solution.*

Proof. By construction, $\boldsymbol{\ell} \leq 2\mathbf{m}^*$. From the previous subsection we know that the vector $\mathbf{m}^*$ provides a lower bound on the value of the objective function for any integral solution. Hence, the theorem follows. $\square$

The complexity of the algorithm is dominated by the complexity of the procedure in [HN94] for optimizing over a monotone system. The running time is $O(mnU^2 \log(Un^2/m))$.

15.6 Vertex cover problem

The vertex cover problem is defined as follows: Given an undirected graph $G = (V, E)$ we want to find the set $S \subseteq V$ with minimum cardinality such that every edge in E is adjacent to (at least) one node in S. In the weighted version, each node $i \in V$ has an associated weight $w_i \geq 0$ and we want to minimize the total weight of the set S. Note that, if the graph has negative or zero weights, then we simply include all those nodes in S and remove their neighbors from V.

Let $x_i = \begin{cases} 1 & \text{if } i \in V \text{ is in the cover} \\ 0 & \text{otherwise} \end{cases}$.

The (weighted) vertex cover problem can be formulated as the integer program below.

$$\min \sum_{i \in V} w_i x_i \qquad (\mathcal{VC}.0)$$

$$\text{s.t. } x_i + x_j \geq 1 \quad \forall \, (i, j) \in E \qquad (\mathcal{VC}.1)$$

$$x_i \in \{0, 1\} \quad \forall \, i \in V$$

15.6.1 Vertex cover on bipartite graphs

The vertex cover problem on bipartite graphs $G = (V_1 \cup V_2, E)$ can be solved in polynomial time. This follows since constraint matrix of the vertex cover problem on bipartite graphs is totally unimodular. To see this, observe that the constraint matrix can be separated in columns into two parts corresponding to V_1 and V_2 respectively, each row within which contains exactly one 1.

That the vertex cover problem on bipartite graph is polynomially solvable can also be seen from our discussion on monotone IP2.

To see this, let

$x_i = \begin{cases} 1 & \text{if } i \in V_1 \text{ is in the cover} \\ 0 & \text{otherwise} \end{cases}$,

$y_i = \begin{cases} -1 & \text{if } i \in V_2 \text{ is in the cover} \\ 0 & \text{otherwise} \end{cases}$.

The integer programming formulation for the vertex cover problem on bipartite graph can then be transformed to the following:

$$\min \sum_{j \in V_1} w_j x_j + \sum_{j \in V_2} (-w_j) y_j \qquad (\mathcal{VC}_B)$$

$$\text{s.t. } x_i - y_j \geq 1 \quad \forall\ (i,j) \in E$$
$$x_i \in \{0,1\} \quad \forall\ i \in V_1$$
$$y_j \in \{-1,0\} \quad \forall\ j \in V_2$$

Evidently, this formulation is a Monotone IP2. Furthermore since both x_i and y_j can take only two values, the size of the resulted s,t-Graph is polynomial. In fact, the following s,t-Graph (Figure 48) can be constructed to solve the vertex cover problem on bipartite graph.
1. Given a bipartite graph $G = (V_1 \cup V_2, E)$, set the capacity of all edges in A equal to ∞.
2. Add a source s, a sink t, set A_s of arcs from s to all nodes $i \in V_1$ (with capacity $u_{s,i} = w_i$), and set A_t of arcs from all nodes $j \in V_2$ to t (with capacity $u_{j,t} = w_j$).
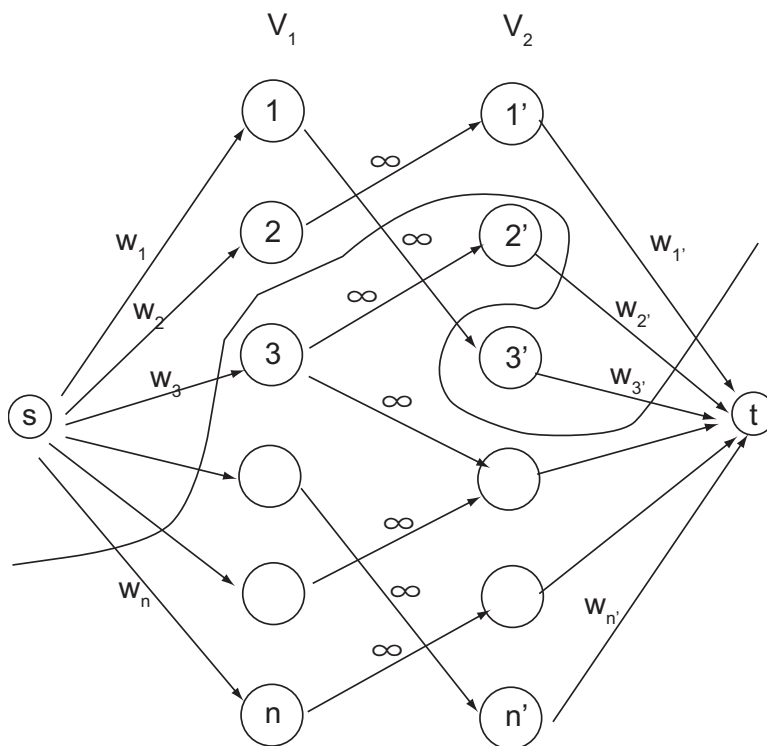
Figure 48: An illustration of the s,t-Graph for the vertex cover problem on bipartite graph and an example of a finite $s-t$ cut

It is easy to see that if $(s \cup S, t \cup T)$ is a finite $s-t$ cut on $G_{s,t}$, then $(V_1 \cap T) \cup (V_2 \cap S)$ is a vertex cover. Furthermore, if $(s \cup S, t \cup T)$ is an optimal solution to the minimum $s-t$ cut problem on $G_{s,t}$, then $(V_1 \cap T) \cup (V_2 \cap S)$ is a vertex cover with the minimum weights, where

$$C(s \cup S, t \cup T) = \sum_{i \in V_1 \cap T} w_i + \sum_{j \in V_2 \cap S} w_j = \sum_{j \in VC^*} w_j$$

15.6.2 Vertex cover on general graphs

The vertex cover problem on general graphs is *NP*-hard. For the unweighted version, there is a 2-approximation algorithm due to Gavril for finding a solution at most twice as large as the minimum solution.

The algorithm goes as follows.
1. Find a maximal (but not necessarily maximum) matching in G, i.e., a maximal subset of edges, M, such that no two edges in M have an endpoint in common.
2. Return $VC^M = \cup_{(i,j) \in M} \{i \cup j\}$, i.e., the set of all endpoints of edges in M.

The correctness of the algorithm can be easily shown. Clearly, since M is maximal, there cannot be an edge in $E \backslash M$ with no endpoint in VC^M. Thus, VC^M is a vertex cover. Obviously, $|VC^M| = 2|M|$. Also, for each of the edges in M, one of the endpoints must be in every vertex cover. Therefore, $VC_{opt} \geq |M|$. In conclusion, $|VC^M| = 2|M| \leq 2|VC_{opt}|$.

Moreover, the LP-relaxation of the vertex cover problem on general graphs can be solved by solving the vertex cover problem in a related bipartite graph. Specifically, as suggested by Edmonds and Pulleyblank and noted in [NT75], the LP-relaxation can be solved by finding an optimal cover C in a bipartite graph $G_b = (V_{b1} \cup V_{b2}, E_b)$ having vertices $a_j \in V_{b1}$ and $b_j \in V_{b2}$ of weight w_j for each vertex $j \in V$, and two edges $(a_i, b_j), (a_j, b_i)$ for each edge $(i,j) \in E$. Given the optimal cover C on G_b, the optimal solution to the LP-relaxation of our original problem is given by:

$$
x_j = \begin{cases} 1 & \text{if} \quad a_j \in C \text{ and } b_j \in C, \\ \frac{1}{2} & \text{if} \quad a_j \in C \text{ and } b_j \notin C, \text{ or} \quad a_j \notin C \text{ and } b_j \in C, \\ 0 & \text{if} \quad a_j \notin C \text{ and } b_j \notin C. \end{cases}
$$

In turn, the problem of solving the vertex cover problem in G_b (or, on any bipartite graph) can be reduced to the minimum cut problem as we showed previously. For this purpose we create a (directed) *st*-graph $G_{st} = (V_{st}, A_{st})$ as follows: (1) $V_{st} = V_b \cup \{s\} \cup \{t\}$, (2) A_{st} contains an infinite-capacity arc (a_i, b_j) for each edge $(a_i, b_j) \in E_b$, (3) A_{st} contains an arc (s, a_i) of capacity w_i for each node $i \in V_{b1}$, and (4) A_{st} contains an arc (b_i, t) of capacity w_i for each node $i \in V_{b2}$.

Given a minimum (S, T) cut we obtain the optimal vertex cover as follows: let $a_i \in C$ if $a_i \in T$ and let $b_j \in C$ if $b_j \in S$.

We now present an alternative method of showing that the LP-relaxation of the vertex cover problem can be reduced to a minimum cut problem, based on our discussion on integer programming with two variables per inequality (IP2).

As we did for the non–monotone integer program with two variables per inequality, we can "monotonize" ($\mathcal{VC}$) by replacing each binary variable x_i in the objective by $x_i = \frac{x_i^+ - x_i^-}{2}$ where $x_i^+ \in \{0, 1\}$ and $x_i^- \in \{-1, 0\}$, and each inequality $x_i + x_j \geq 1$ by the following two inequalities:

$$
x_i^+ - x_j^- \geq 1
$$
$$
-x_i^- + x_j^+ \geq 1
$$

The "monotonized" formulation is shown below:

$$\min \frac{1}{2} \left\{ \sum_{i \in V} w_i x_i^+ + \sum_{i \in V} (-w_i) x_i^- \right\} \qquad (\mathcal{VC}'.0)$$

$$\text{s.t. } x_i^+ - x_j^- \geq 1 \quad \forall (i,j) \in E \qquad (\mathcal{VC}'.1)$$

$$-x_i^- + x_j^+ \geq 1 \quad \forall (i,j) \in E \qquad (\mathcal{VC}'.2)$$

$$x_i^+ \in \{0,1\} \quad \forall i \in V$$

$$x_i^- \in \{-1,0\} \quad \forall i \in V$$

The relationship between $(\mathcal{VC})$ and $(\mathcal{VC}')$ can be easily seen. For any feasible solution $\{x_i\}_{i \in V}$ for $(\mathcal{VC})$, a corresponding feasible solution for $(\mathcal{VC}')$ with the same objective value can be constructed by letting $x_i^+ = x_i, x_i^- = -x_i, \forall i \in V$; On the other hand, for any feasible solution $\{x_i^+, x_i^-\}_{i \in V}$ for $(\mathcal{VC}')$, $x_i = \frac{x_i^+ - x_i^-}{2}$ satisfies inequality $(\mathcal{VC}.1)$ (To see this, simply sum up inequalities $(\mathcal{VC}.1)$ and $(\mathcal{VC}.2)$ and divide the resulted inequality by 2.) but may not be integral. Therefore, given an optimal solution $\{x_i^+, x_i^-\}_{i \in V}$ for $(\mathcal{VC}')$, the solution $\{x_i = \frac{x_i^+ - x_i^-}{2}\}$ is feasible, half integral (i.e., $x_i \in \{0, \frac{1}{2}, 1\}$) and super-optimal (i.e., its value provides a lower bound) for $(\mathcal{VC})$.

Comparing $(\mathcal{VC}')$ with $(\mathcal{VC})_{\mathcal{B}}$, we can see that without the coefficient $\frac{1}{2}$ in the objective $(\mathcal{VC}')$ can be treated as a vertex cover problem on a bipartite graph $G(V_{b_1} \cup V^{b_2}, E_b)$ where $V_{b_1} = a_j, \forall j \in V, V_{b_2} = b_j, \forall i \in V, E_b = \{(a_i, b_j), (a_j, b_i), \forall (i,j) \in E\}$.

Therefore, we obtain the following relationship

$$VC^{+-} \leq 2VC^*$$

where VC^* is the optimal value of the original vertex cover problem on a general graph $G(V, E)$, and VC^{+-} is the optimal value of the vertex cover problem defined on the bipartite graph $G(V_{b_1} \cup V^{b_2}, E_b)$.

15.7 The s-excess problem

The s-excess problem is another generalization of the closure problem. Here we introduce edge weights e_{ij} and relax the closure requirement. However, the objective receives a penalty for any pair of nodes that do not satisfy the requirement, which is proportional to the amount of the violation. Let z_{ij} be 1 if $i \in S^*$ and $j \in \bar{S}^*$, and 0 otherwise. Then the minimum s-excess problem is to find a subset of nodes $S \subseteq V$ that minimizes

$$\sum_{i \in S} w_i + \sum_{i \in S, j \in \bar{S}} e_{ij} ,$$

which can be represented by the following integer linear program.

$$\min \quad \sum_{i \in V} w_i x_i + \sum_{(i,j) \in A} e_{ij} z_{ij}$$

$$\begin{array}{lll}
\text{subject to} & x_i - x_j \leq z_{ij} & (i,j) \in A \\
& 0 \leq x_i \leq 1 & i \in V \\
& 0 \leq z_{ij} \leq 1 & (i,j) \in A \\
& x_i, \ z_{ij} \text{ integer}
\end{array}$$

[BP89] and [Hoc08] show that this problem is equivalent to solving the minimum $s - t$ cut problem on a modified graph, G_{st}, defined as follows. We add nodes s and t to the graph, with an arc from s to every negative weight node i (with capacity $u_{si} = -w_i$), and an arc from every positive weight node j to t (with capacity $u_{jt} = w_j$).

Lemma 15.8. S^* *is a set of minimum s-excess capacity in the original graph G if and only if S^* is the source set of a minimum cut in G_{st}.*

Proof. As before, let $V^+ \equiv \{i \in V | w_i > 0\}$, and let $V^- \equiv \{j \in V | w_j < 0\}$. Let $(s \cup S, t \cup T)$ define an $s - t$ cut on G_{st}. Then the capacity of this cut is given by

$$
\begin{aligned}
C\left(s \cup S, t \cup T\right) &= \sum_{(s,i) \in A_s,\ i \in T} u_{s,i} + \sum_{(j,t) \in A_t,\ j \in S} u_{j,t} + \sum_{i \in S,\ j \in T} e_{ij} \\
&= \sum_{i \in T \cap V^-} -w_i + \sum_{j \in S \cap V^+} w_j + \sum_{i \in S,\ j \in T} e_{ij} \\
&= \sum_{i \in V^-} -w_i - \sum_{i \in S \cap V^-} -w_i + \sum_{j \in S \cap V^+} w_j + \sum_{i \in S,\ j \in T} e_{ij} \\
&= W^- + \sum_{j \in S} w_j + \sum_{i \in S,\ j \in T} e_{ij}
\end{aligned}
$$

Where W^- is the sum of all negative weights in G, which is a constant. Therefore, minimizing $C\left(s \cup S, t \cup T\right)$ is equivalent to minimizing $\sum_{j \in S} w_j + \sum_{i \in S,\ j \in T} e_{ij}$, and we conclude that the source set of a minimum $s - t$ cut on G_{st} is also a minimum s-excess set of G.

$\square$

References

[BP89]   D. Greig B. Porteous, A. Seheult. Exact maximum a posteriori estimation for binary images. *Journal of the Royal Statistical Society, Series B*, 51(2):271–279, 1989.

[HN94]   D. S. Hochbaum and J. Naor. Simple and fast algorithms for linear and integer programs with two variables per inequality. *SIAM Journal on Computing*, 23(6):1179–1192, 1994.

[HNMT93] D. S. Hochbaum, J. Naor N. Megiddo, and A. Tamir. Tight bounds and 2-approximation algorithms for integer programs with two variables per inequality. *Mathematical Programming*, 62:69–83, 1993.

[Hoc97]   D. S. Hochbaum. Approximating covering and packing problems: set cover, vertex cover, independent set and related problems. chapter 3. In *Approximation algorithms for NP-hard problems*, pages 94–143. PWS Boston, 1997.

[Hoc02]   D. S. Hochbaum. Solving integer programs over monotone inequalities in three variables: A framework for half integrality and good approximations. *European Journal of Operational Research*, 140(2):291–321, 2002.

[Hoc08]   D. S. Hochbaum. The pseudoflow algorithm: A new algorithm for the maximum flow problem. *Operations Research*, 58(4):992–1009, July-Aug 2008.

[Joh68]   T. B. Johnson. *Optimum Open Pit Mine Production Technology.* PhD thesis, Operations Research Center, University of California, Berkeley, 1968.

[Lag85]   J. C. Lagarias. The computational complexity of simultaneous diophantine approximation problems. *SIAM Journal on Computing*, 14:196–209, 1985.

[Pic76]   J. C. Picard. Maximal closure of a graph and applications to combinatorial problems. *Management Science*, 22:1268–1272, 1976.

16   Solving IP3 as an s-excess problem

The notes in this section, on IP3, are based on extracts from the paper [3].
The s-excess problem is a generalization of the closure problem in that it allows us to violate the closure constraints, with penalties. The process of solving IP3 relies on an analogous binarizing process which transforms an IP3 problem to an s-excess problem, which in turn, is also solvable as a minimum s, t-cut on a related graph. The process of binarizing leads to the following theorem:

Theorem 16.1. *[2] A monotone* IP3 *is equivalent to an integer program in $O(nU)$ binary variables over $O(mU)$ totally unimodular constraints.*

In [2] it is also shown how to solve, with the same complexity, also the more general convex monotone IP3 problem with an objective function which is convex and separable in the variables x_j and z_{ij}.

The process of "Binarizing" when applied to an IP3 results in an equivalent problem in binary variables with constraint coefficients that are in $\{0, -1, 1\}$ and inequalities of the form:

$$
\begin{aligned}
x_i - x_j &\leq 0 \qquad \text{or} \\
x_i - x_j &\leq z_{ij}.
\end{aligned}
$$

More precisely, "Binarizing" converts any monotone IP3 to an instance of the *minimum s-excess problem*. That problem was introduced in [1] in the context of the pseudoflow algorithm. The problem is defined as follows:

Problem Name: *Minimum s-Excess*

Instance: *Given a directed graph $G = (V, A)$, node weights (positive or negative) w_i for all $i \in V$, and nonnegative arc weights e_{ij} for all $(i,j) \in A$.*

Optimization Problem: *Find a subset of nodes $S \subseteq V$ such that $\sum_{i \in S} w_i + \sum_{i \in S, j \in \bar{S}} e_{ij}$ is minimum.*

The minimum s-excess problem is a generalization of the maximum closure, or rather the respective minimum closure, problem. The constraints of the s-excess problem are a relaxation of the closure constraints, in that a violation of the closure constraint $x_i \leq x_j$ is permitted at a penalty of e_{ij} per unit violation. Indeed, when the quantities e_{ij} are infinite, the problem reduces to the closure problem. The s-excess problem is formulated as a binary optimization problem

$$
\begin{aligned}
(s\text{-excess}) \quad \min \quad & \sum_{j \in V} w_j x_j + \sum_{(i,j) \in A} e_{ij} z_{ij} \\
\text{subject to} \quad & x_i - x_j \leq z_{ij} \quad \text{for } (i,j) \in A \\
& x_j \quad \text{binary } j = 1, \dots, n \\
& z_{ij} \quad \text{binary } (i,j) \in A.
\end{aligned}
$$

Although the constraints of the type $x_i - x_j \leq 0$ do not seem to appear in this formulation, these are written as $x_i - x_j \leq z_{ij}$ where the cost coefficient e_{ij} of the respective variable z_{ij} (and the capacity of the corresponding arc) is infinite. It is shown next that the minimum s-excess set in a graph G is the source set of a minimum cut in an associated graph G_{st}, defined as follows. We add nodes s and t to the graph G, with an arc from s to every negative weight node i (with capacity $u_{si} = -w_i$), and an arc from every positive weight node j to t (with capacity $u_{jt} = w_j$).

Lemma 16.1. S^* *is a set of minimum s-excess capacity in the original graph G if and only if S^* is the source set of a minimum cut in an associated graph G_{st}.*

Proof. Recall the notation, $V^+ \equiv \{i \in V | w_i > 0\}$, and $V^- \equiv \{j \in V | w_j < 0\}$. Let $(\{s\} \cup S, \{t\} \cup T)$ define an s, t-cut on G_{st}. Then the capacity of this cut is given by

$$
\begin{aligned}
C\left(\{s\} \cup S, \{t\} \cup T\right) &= \sum_{(s,i) \in A_{st}, \, i \in T} u_{s,i} + \sum_{(j,t) \in A_{st}, \, j \in S} u_{j,t} + \sum_{i \in S, \, j \in T} e_{ij} \\
&= \sum_{i \in T \cap V^-} -w_i + \sum_{j \in S \cap V^+} w_j + \sum_{i \in S, \, j \in T} e_{ij} \\
&= \sum_{i \in V^-} -w_i - \sum_{j \in S \cap V^-} -w_i + \sum_{j \in S \cap V^+} w_j + \sum_{i \in S, \, j \in T} e_{ij} \\
&= W^- + \sum_{j \in S} w_j + \sum_{i \in S, \, j \in T} e_{ij},
\end{aligned}
$$

where W^- is the sum of all negative weights in G, which is a constant. Therefore, minimizing $C\left(s \cup S, t \cup T\right)$ is equivalent to minimizing $\sum_{j \in S} w_j + \sum_{i \in S, \, j \in T} e_{ij}$, and we conclude that the source set of a minimum s, t-cut on G_{st} is also a minimum s-excess set of G.

$\square$

It was shown, in the previous section, how to represent a monotone inequality with two variables, $ax_i - bx_j \leq c$, as a graph in which any feasible finite cut corresponds to a feasible solution. Similarly, for a monotone inequality on three variables, $ax_i - bx_j \leq c_{ij} + z_{ij}$, the construction of the x_i-chain is exactly the same as in the case of IP2. The difference is in the setting of the arcs and the capacities assigned to these arcs. The algorithm presented here applies for the more general nonlinear extension of IP3 where each $w_i x_i$ may be replaced by any nonlinear function $w_i(x_i)$, and each term $e_{ij} z_{ij}$ may be replaced by $e_{ij}(z_{ij})$ for $e_{ij}()$ a convex nondecreasing function.

Consider the monotone inequality of the type $ax_i - bx_j \leq c_{ij} + z_{ij}$. For notational convenience and without risk of ambiguity we let, in the following discussion, $c = c_{ij}$ and $z = z_{ij}$. Let $e_{ij}(z)$ be the (convex nondecreasing) cost function associated with z, and $0 \leq z \leq \gamma_{ij}$.

The inequality is equivalent to the set of implications:

$$
\text{If} \quad x_i \geq p \quad \text{then} \quad x_j \geq \lceil \frac{ap - c - z}{b} \rceil.
$$

We denote

$$
j(p, z) \equiv \lceil \frac{ap - c - z}{b} \rceil.
$$

Several more notational conventions will streamline the exposition: Instead of listing the inequalities we list the capacities of the arcs in the associated graph G. We let the capacity of arc

(p, q) be denoted by $c(p, q)$. Stating that $c(p, q) = \infty$ will be taken to be equivalent to generating an inequality

$$x_i^{(p)} \le x_j^{(q)}.$$

Stating that $c(p, q) = K$ will be taken to be equivalent to generating an inequality

$$x_i^{(p)} \le x_j^{(q)} + z(p, q)$$

where the cost of the binary variable $z(p, q)$ in the objective function is K. Therefore the objective function term corresponding to the z-variables is

$$\sum_{(i,j) \in E_1} \sum_{p_i = \ell_i}^{u_i} \sum_{p_j = \ell_j}^{u_j} c(p_i, p_j) z(p_i, p_j).$$

We will demonstrate that the capacities and costs indeed correspond to the correct cost of the z variables.

The description is restricted to the case where $a \le b$. The procedure for $a > b$ follows similar principles and will be omitted. This assumption allows us to conclude that when the value of x_i is increased by 1, the value of $j(p, z)$ can increase by one unit at most.

16.1 z is binary

We first describe the procedure for generating inequalities and arcs for z binary, i.e., $\gamma_{ij} = 1$. The challenge here is to track whether the setting of z to be equal to 1 does indeed relax the inequality. In general, there should be an arc $(p, j(p, 0))$ of cost $e_{ij} = e_{ij}(1)$. We need however to avoid the situation when $j(p, 0) = j(p - 1, 0)$ and charge twice the value e_{ij} when $x_i = p$ and $x_j = j(p, 0) - 1$. In this case both $x_i^{(p)}$ and $x_i^{(p-1)}$ are in the source set of the cut and $x_i^{(j(p,0))}$ is in the sink set. So only arc $(p - 1, j(p - 1, 0))$ exists and the charge of e_{ij} is applied once.

Generate arcs $\left(ax_i - bx_j \le c + z,\ 1,\ e_{ij}()\right)$
_____
Let p be smallest so that $p \ge \ell_i$ and $j(p, 0) \ge \ell_j + 1$.
If $p > u_i$, then stop: Output "constraint is always satisfied".
If $j(p, 1) > u_j$, then stop: "problem is infeasible".
Set $c(p, j(p, 1)) = \infty$.
If $j(p, 0) > j(p, 1)$ then $c(p, j(p, 0)) = e_{ij}$.
while $p \le u_i - 1$, do
    $p \leftarrow p + 1$.
    $c(p, j(p, 1)) = \infty$.
    If $j(p, 0) > j(p, 1)$ and $j(p, 0) > j(p - 1, 0)$ then $c(p, j(p, 0)) = e_{ij}$
    Else, continue
end
Proof of correctness: Let $x_i = p$ and $x_j = q$ in some solution.
If $q < j(p, 1)$ then the solution is infeasible as it violates the inequality $ax_i - bx_j \le c + z$. The arc $(p, j(p, 1))$ is in the cut and it has infinite capacity, as required. Thus such a solution has infinite s-excess value.
If $q = j(p, 1) < j(p, 0)$ then $c < ap + b \le c + 1$. Therefore the value of z must be equal to 1 with a charge of e_{ij} to the objective function. The only arc in the cut between the x_i and the x_j chains is from the lowest node $p' \le p$ in the x_i-chain with $j(p', 1) < j(p', 0) = j(p, 0)$. This arc is of capacity

e_{ij} and thus $\sum_{p_i=\ell_i}^{u_i} \sum_{p_j=\ell_j}^{u_j} c(p_i, p_j) z(p_i, p_j) = e_{ij}$.

If $q > j(p, 1)$ then, since $a \leq b$, $j(p, 0) = j(p, 1) + 1$, $q \geq j(p, 0)$ in which case there is no arc in the cut associated with this inequality. $\square$

16.2 z is integer

The generation of arcs/inequalities for the case of z integer obviously generalizes the case of z binary.

Let $x_i = p$ and define q_p to be the smallest value such that $j(p, q) = j(p, 0) - 1$. That is,

$$\lceil \frac{ap - c - q_p}{b} \rceil = \frac{ap - c - q_p}{b} = \lceil \frac{ap - c}{b} \rceil - 1.$$

We will introduce arcs adjacent to $x_i = p$ sequentially for $p = \ell_i, \ldots, u_i$. The nodes in the x_j-chain will have at any point in the process a subset of arcs from nodes $p = \ell_i, \ldots, p - 1$ with certain incapacity already generated. We denote the total incumbent sum of incapacity into node p' by $C(p')$.

Generate arcs $(ax_i - bx_j \leq c + z, \gamma_{ij}, e_{ij}())$

Let p be smallest so that $p \geq \ell_i$ and $j(p, 0) \geq \ell_j + 1$.

If $p > u_i$, then stop: Output "constraint is always satisfied".

If $j(p, \gamma_{ij}) > u_j$, then stop: "problem is infeasible".

while $p \leq u_i$, do

Let q_p be smallest so that $j(p, q_p) = j(p, 0) - 1$. Set $c(p, j(p, 0)) = e_{ij}(q_p) - C(j(p, 0))$, $k = 1$

    until $j(p, 0) - k = \max\{\ell_j, j(p, \gamma_{ij})\} + 1$, do:

        set $c(p, j(p, 0) - k) = e_{ij}(kb + q_p) - e_{ij}((k - 1)b + q_p) - C(j(p, 0) - k)$.

        $k \leftarrow k + 1$.

    end

Let $c(p, j(p, \gamma_{ij})) = \infty$.

    $p \leftarrow p + 1$.

end

Since the functions $e_{ij}()$ are convex and monotone nondecreasing then it is guaranteed that every capacity generated is nonnegative. Another property of the generated capacities is that the total sum of capacities in a cut between x_i and x_j is precisely $e_{ij}(ax_i - bx_j - c)$.

<u>Proof of correctness</u>: We first establish the nonnegativity of the capacities.

When done with the assignment of arc capacities adjacent to node $x_i = p$ then the new total capacities are

$$
\begin{aligned}
C(j(p, 0)) \quad &= e_{ij}(q_p) \\
C(j(p, 0) - 1) \quad &= e_{ij}(b + q_p) - e_{ij}(q_p) \\
C(j(p, 0) - 2) \quad &= e_{ij}(2b + q_p) - e_{ij}(b + q_p) \\
&\quad . \\
&\quad . \\
&\quad . \\
&\quad . \\
C(p, j(p, 0) - k) \quad &= e_{ij}(kb + q_p) - e_{ij}((k - 1)b + q_p).
\end{aligned}
$$

First we show the nonnegativity of the capacities of all arcs. Suppose this is true by induction for all arcs adjacent to nodes $x_i = \ell_i, \ldots, p$ and consider the capacities of arcs adjacent to node $x_i = p + 1$. There are two cases:

$j(p, 0) = j(p + 1, 0)$, or
$j(p, 0) < j(p + 1, 0)$.

In the first case there are arcs of total capacity $e_{ij}(q_p)$ adjacent to $j(p, 0)$ when we assign the capacity to the arc $(p, j(p + 1, 0)$. Necessarily $q_{p+1} > q_p$ (the difference is in fact a units) thus recalling that $e_{ij}()$ are monotone nondecreasing, $e_{ij}(q_p) \leq e_{ij}(q_{p+1})$. Therefore $e_{ij}(q_{p+1}) \geq C(j(p, 0))$. Also, for any k,

$$C(p, j(p, 0) - k) = e_{ij}(kb + q_p) - e_{ij}((k - 1)b + q_p) \leq e_{ij}(kb + q_{p+1}) - e_{ij}((k - 1)b + q_{p+1}).$$

Thus all arc capacities are nonnegative.

Consider now the second case where $j(p, 0) < j(p + 1, 0)$. The incumbent capacity adjacent to $j(p + 1, 0)$ is 0. Also $q_{p+1} + b > q_p$ thus for all k,

$$C(p, j(p, 0) - k) \leq e_{ij}((k - 1)b + q_p) - e_{ij}((k - 2)b + q_p) \leq e_{ij}(kb + q_{p+1}) - e_{ij}((k - 1)b + q_{p+1}),$$

Thus all arc capacities are nonnegative.

Let $x_i = p$ and $x_j = q$ in some solution.
If $q < j(p, \gamma_{ij})$ then the solution is infeasible as it violates the inequality in its most relaxed form $ax_i - bx_j \leq c + \gamma_{ij}$. The arc $(p, j(p, \gamma_{ij}))$ is then in the cut and it has infinite capacity, as required. Thus such a solution has infinite s-excess value.

If $q \geq j(p, 0)$ then the inequality is satisfied and there is no arc associated with this inequality (and with z_{ij}) in the cut.

So suppose that $j(p, 0) > q \geq j(p, \gamma_{ij})$. In this case the value of z_{ij} in an associated optimal assignment is $\lceil ap - bq - c \rceil$.

Claim 16.2. $z_{ij} = q_p + b(j(p, 0) - q - 1)$

Proof: Assume for convenience that c is integer, then $q_p = ap - c - b(j(p, 0) - 1)$. Thus,

$$q_p + b(j(p, 0) - q - 1) = ap - c + bq.$$

□

Since the total capacity adjacent to nodes $x_j = q + 1, \ldots, j(p, 0)$ is $C(p, x_j)$, the total charge for arcs in the cut is $\sum_{x_j=q+1}^{j(p,0)} C(p, x_j)$. This sum is a telescopic sequence adding up to $e_{ij}(q_p + b(j(p, 0) - q - 1))$:

$$
\begin{aligned}
\sum_{x_j=q+1}^{j(p,0)} C(p, x_j) \quad &= e_{ij}(q_p) \\
&+ e_{ij}(b + q_p) - e_{ij}(q_p) \\
&+ e_{ij}(2b + q_p) - e_{ij}(b + q_p) \\
&\quad . \\
&\quad . \\
&\quad . \\
&\quad . \\
&+ e_{ij}((j(p, 0) - q - 1)b + q_p) - e_{ij}((j(p, 0) - q - 2)b + q_p) \\
&= e_{ij}((j(p, 0) - q - 1)b + q_p).
\end{aligned}
$$

So the total charge for arcs in the cut is precisely $e_{ij}(z_{ij})$ as required.

16.2.1 An example

We present an example for the case when $a_{ij} = b_{ij} = 1$.

We define recursively the excess unit increments of the functions e_{ij} (a discrete equivalent of the second derivative):

$$
\begin{aligned}
\Delta_{ij}(0) &= 0 \\
\Delta_{ij}(1) &= e_{ij}(1) - e_{ij}(0) \\
\Delta_{ij}(2) &= e_{ij}(2) - e_{ij}(1) - \Delta_{ij}(1) \\
&\quad . \\
&\quad . \\
&\quad . \\
&\quad . \\
\Delta_{ij}(\gamma_{ij}) &= e_{ij}(\gamma_{ij}) - e_{ij}(\gamma_{ij} - 1) - \Delta_{ij}(\gamma_{ij} - 1).
\end{aligned}
$$

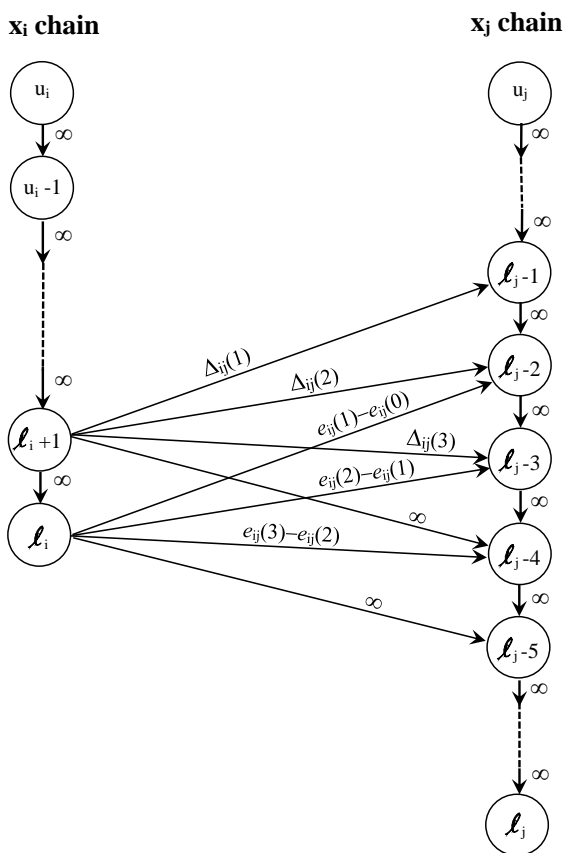Figure 49: Arcs and their capacities when $e_{ij}()$ are convex

Figure 49 depicts the generated graph for an inequality $x_i - x_j \leq 2 + z_{ij}$ where $\gamma_{ij} = 3$. The illustration shows the arcs associated with the lowest nodes in the x_i chain. Note that the arcs

for the first node, ℓ_i, follow a different pattern from that of the other nodes. That is because the higher valued nodes in the chain are guaranteed that when they are in the source set, then so are all the nodes under them and thus the arcs adjacent to a particular valued node in the x_j chain is also originating from lower valued nodes. Thus such an arc contributes only to the incremental cost.

In this example:

$c(\ell_i, \ell_i - 2) = e_{ij}(1) - e_{ij}(0)$. If this arc is in the cut then $z_{ij} \geq 1$.

$c(\ell_i, \ell_i - 3) = e_{ij}(2) - e_{ij}(1)$. If this arc is in the cut then $z_{ij} \geq 2$.

$c(\ell_i, \ell_i - 4) = e_{ij}(3) - e_{ij}(2)$. If this arc is in the cut then $z_{ij} \geq 3$.

$c(\ell_i, \ell_i - 5) = \infty$. It is infeasible for x_j to be $\leq \ell_i - 5$.

$c(\ell_i + 1, \ell_i - 1) = \Delta_{ij}(1) = e_{ij}(1) - e_{ij}(0)$. If this arc is in the cut then $z_{ij} \geq 1$.

$c(\ell_i + 1, \ell_i - 2) = \Delta_{ij}(2)$. If this arc is in the cut then also the arc $(\ell_i, \ell_i - 2)$ and the arc $(\ell_i + 1, \ell_i - 1)$ are in the cut and $z_{ij} \geq 2$. The total capacity of these three arcs is $\Delta_{ij}(1) + \Delta_{ij}(2) + e_{ij}(1) - e_{ij}(0) = e_{ij}(2) - e_{ij}(1) + e_{ij}(1) - e_{ij}(0) = e_{ij}(2) - e_{ij}(0)$ as required.

$c(\ell_i + 1, \ell_i - 3) = \Delta_{ij}(3)$. If this arc is in the cut then $z_{ij} \geq 3$.

$c(\ell_i + 1, \ell_i - 4) = \infty$. It is infeasible for $x_j \leq \ell_i - 4$ if $x_i \geq \ell_i + 1$.

References

[1] D.S. Hochbaum. *The pseudoflow algorithm and the pseudoflow-based simplex for the maximum flow problem.* Proc. of IPCO 98, Lecture Notes in Computer Science, vol. 1412, 1998, pp. 325–337.

[2] D.S. Hochbaum. *Solving integer programs over monotone inequalities in three variables: A framework for half integrality and good approximations.* Euro. J. of Oper. Res. **140**:2 (2002), 291–321.

[3] Dorit S. Hochbaum. *Applications and efficient algorithms for integer programming problems on monotone constraints.* Networks , 77: 1 (2021), 21–49. DOI:10.1002/net.21983

17 Necessary and Sufficient Condition for Feasible Flow in a Network

First the discussion is for zero lower bounds. We assume throughout that the total supply in the graph is equal to the total demand, else there is no feasible solution.

17.1 For a network with zero lower bounds

A cut is a partition of the nodes, $(D, \bar{D})$. We first demonstrate a necessary condition, and then show that it is also sufficient for flow feasibility.

The set D contains some supply nodes and some demand nodes. Let the supply and demand of each node $j \in V$ be denoted by b_j, where b_j is positive if j is a supply node, and negative, if j is a demand node. The cut $(D, \bar{D})$ must have sufficient capacity to accommodate the net supply (which could be negative) that must go out of D.

$$\sum_{j \in D} b_j \leq C(D, \bar{D}) = \sum_{i \in D, j \in \bar{D}} u_{ij}, \qquad \forall D \subset V. \tag{18}$$

This condition must be satisfied for every partition, namely, for every subset of nodes $D \subset V$.

The interesting part is that this condition is also sufficient, and that it can be verified by solving a certain minimum cut problem.

In order to show that the condition is sufficient we construct an auxiliary s, t graph. Let S be the set of all supply nodes, and B be the set of all demand nodes. We append to the graph a source node s with arcs going to all supply nodes with capacity equal to their supplies. We place a sink node t with arcs going from all nodes of B to the sink with capacity equal to the absolute value of the demand at each node. There exists a feasible flow in the original graph only if the minimum s, t cut in this auxiliary graph is at least equal to the total supply $sup = \sum_{b_j > 0} b_j$ (or total demand, which is equal to it). In other words, since the total supply value is the capacity of some minimum cut – the one adjacent to the source, or the one adjacent to the sink – either one of these cuts must be a minimum cut. In particular, for any partition of V, $(D, \bar{D})$, the cut capacity $C(\{s\} \cup D, \{t\} \cup \bar{D})$ must be at least as big as the total supply:

$$
\begin{aligned}
sup \leq C(\{s\} \cup D, \{t\} \cup \bar{D}) \ &= C(\{s\}, D) + C(D, \bar{D}) + C(D, \{t\}) \\
&= \textstyle\sum_{j \in \bar{D}, b_j > 0} b_j + C(D, \bar{D}) + \sum_{j \in D, b_j < 0} -b_j.
\end{aligned}
$$

Reordering the terms yields,

$$
\sum_{j \in D, b_j > 0} b_j + \sum_{j \in D, b_j < 0} b_j \leq C(D, \bar{D}).
$$

The left hand side is simply the sum of weights of nodes in D. This necessary condition is therefore identical to the condition stated in (18).

Verification of condition (18).

A restatement of condition (18) is that for each subset of nodes $D \subset V$,

$$
\sum_{j \in D} b_j - C(D, \bar{D}) \leq 0.
$$

The term on the left is called the *s-excess* of the set D. In Section 15.7 it is shown that the maximum *s-excess* set in a graph can be found by solving a minimum cut problem. The condition is thus satisfied if and only if the maximum *s-excess* in the graph is of value 0.

17.2 In the presence of positive lower bounds

In this case condition (18) still applies where the cut value $C(D, \bar{D}) = \sum_{i \in D, j \in \bar{D}} u_{ij} - \sum_{u \in \bar{D}, v \in D} \ell_{uv}$

17.3 For a circulation problem

Here all supplies and demands are zero. The condition is thus,
For all $D \subset V$

$$
\sum_{i \in D, j \in \bar{D}} u_{ij} - \sum_{u \in \bar{D}, v \in D} \ell_{uv} \geq 0.
$$

17.4 For the transportation problem

Here all arc capacities are infinite. The arcs capacities in the s, t graph we construct, as before, are only finite for source adjacent and sink adjacent arcs. Let the transportation problem be given on the bipartite graph $(V_1 \cup V_2, A)$ for V_1 a set of supply nodes, and V_2 a set of demand nodes.

Consider an partition in the graph $(D, \bar{D})$. Since all arc capacities in A are infinite, Condition (18) is trivially satisfied whenever an arc of A is included in the cut. So we restrict our attention to cuts that do not include an arc of A. Such cuts have the set D closed, namely all the successors of the supply nodes in D are also included in D. For closed sets D the cut capacity is 0. The condition is then for each closed set D,

$$\sum_{j \in D b_j > 0} b_j \leq \sum_{j \in D b_j < 0} -b_j.$$

This condition is an extension of Hall's theorem for the existence of perfect matching in bipartite graphs.

(See also pages 194-196 of text)

18 Algorithms for MCNF

18.1 Network Simplex

Consider the linear programming formulation of the minimum cost flow problem on a network $G = (V, A)$, with x_{ij} the amount of flow on arc $(i, j) \in A$:

$$
\begin{aligned}
\text{Minimize} \quad & z = \sum_{(i,j) \in A} c_{ij} x_{ij} \\
\text{Subject to:} \quad & \sum_{(i,j) \in A} x_{ij} - \sum_{(k,i) \in A} x_{ki} = b_i \quad i \in V \\
& 0 \leq x_{ij} \leq u_{ij} \quad\quad\quad (i,j) \in A
\end{aligned}
$$

It is easily seen that the constraint matrix is not of full rank. In fact, the matrix is of rank n-1. This can be remedied by adding an artificial variable corresponding to an arbitrary node. Another way is just to throw away one constraint, say the one corresponding to node 1. For the uncapacitated version, where the capacities u_{ij} are infinite, the dual is :

$$
\begin{aligned}
\text{Maximize} \quad & z = \sum_{j \in V} b_j \pi_j \\
\text{Subject to:} \quad & \pi_i - \pi_j \leq c_{ij} \quad (i,j) \in A.
\end{aligned}
$$

We note that any MCNF problem can be formulated as an uncapacitated network flow problem. The reduction is in the textbook, page 40. An arc (i, j) with capacity u_{ij} is replaced by an extra node ij and two arcs, (i, ij) and (j, ij). The first is of cost c_{ij} and capacity ∞. The second is of cost 0 and capacity ∞. The supply of node i remains unchanged, b_i. The supply of node j becomes $b_j + u_{ij}$, and the supply (demand) of node ij is $-u_{ij}$. This transformation increases the number of nodes to $n + m$ and the number of arcs to $2m$.

We denote by c_{ij}^{π} the reduced cost $c_{ij} - \pi_i + \pi_j$ which are the slacks in the dual constraints. As proved earlier, the sum of the reduced costs along a cycle is equal to the sum of the original costs.

Simplex method assumes that an initial "basic" feasible flow is given by solving phase I or by solving another network flow problem. By basis, we mean the $n - 1$ x_{ij} variables whose columns in the constraint matrix are linearly independent. The non-basic variables are always set to zero. (or

in the case of finite upper bounds, nonbasic variables may be set to their respective upper bounds.) We will prove in Section 18.3 that a basic solution corresponds to a spanning tree. The concept of Simplex algorithm will be illustrated in the handout but first we need to define some important terminologies used in the algorithm first.

Given a flow vector x, a basic arc is called *free* iff $0 < x_{ij} < u_{ij}$. An arc (i,j) is called *restricted* iff $x_{ij} = 0$ or $x_{ij} = u_{ij}$. From the concept of basic solution and its corresponding spanning tree we can recognize a characteristic of an optimal solution. An optimal solution is said to be *cycle-free* iff in every cycle at least one arc assumes its lower or upper bound.

Proposition 18.1. *If there exists a finite optimal solution, then there exists an optimal solution which is cycle-free.*

Proof: Prove by contradiction. Suppose we have an optimal solution which is not cycle-free and has the minimum number of free arcs. Then we can identify a cycle such that we can send flow in either clockwise or counter-clockwise direction. By linearity, one direction will not increase the objective value as sum of the costs of arcs along this direction is nonpositive. From our finiteness assumption, we will make at least one arc achieve its lower or upper bound after sending a certain amount of flow along the legitimate direction. This way, we get another optimal solution having fewer free arcs and the contradiction is found. □

18.2 (T,L,U) structure of an optimal solution

There are three types of arcs in any feasible solution: free arcs, arcs with zero flow, and arcs with the flow at upper bound. We simply state the corollary which stems from the property 18.1 that there exists an optimal solution that is cycle-free.

Corollary 18.2. *There exists an optimal solution that has the structure (T,L,U), where T is a spanning tree containing all free arcs, L is the set of arcs at lower bounds, and U the set of arcs at upper bounds.*

Proof: Since there exists an optimal solution which is cycle-free, then the free arcs form a spanning forest. It is then possible to add restricted arcs to form a spanning tree. □

18.3 Simplex' basic solutions and the corresponding spanning trees

The following is an important observation for applying the simplex method to the minimum cost network flow problem.

Theorem 18.1. *There is a one-to-one correspondence between a basic solution and a spanning tree.*

Proof: A basic solution corresponds to linearly independent columns. Recall that we can add an artificial variable to make the constraint matrix full rank. Clearly, the fictitious column we add is always in a basis for otherwise the original matrix is of full rank. So a basis can be characterized by n nodes and n-1 arcs. If we can show that there are no cycles, then we are done, since an acyclic graph on n nodes and $n-1$ edges is a spanning tree. For the sake of contradiction, suppose there exists a cycle in the graph corresponding to a basis. Select some arc (i,j) in the cycle and orient the cycle in the direction of that arc. Then for each column in the basis associated with an arc in the cycle, assign a coefficient 1 if the arc is in the direction of the orientation of the cycle and -1 otherwise. Applying these coefficients to respective columns in the basis we find that the weighted

sum of these columns is the zero vector. Thus the columns of the basis could not have been linearly independent. □

Note that not every basis is feasible, the same as for general linear programming. Furthermore, it can be shown that we can always permute the columns of a basis in a way such that it is in lower-triangular form, which is ideal for Gaussian elimination.

The simplex algorithm works as follows: Given a feasible spanning tree, an entering arc is a nonbasic arc with negative reduced cost, or equivalently, we identify a negative cost cycle by adding it to the spanning tree of basic arcs, all of which have zero reduced costs. The leaving arc, the counterpart of the entering arc, is the bottleneck arc on the residual cycle.

This algorithm corresponds to optimality condition 1 that says that a solution is optimal if the residual graph contains no negative cost cycle (the sum of *reduced* costs along a cycle is equal to the sum of costs along a cycle).

18.4 Optimality Conditions

For a flow x^*, we define a residual network $G(x^*)$ such that every arc (i, j) with capacity u_{ij}, flow x_{ij}, and arc cost c_{ij} in the original network corresponds to two arcs in the residual network: one going from i to j, with cost c_{ij} and flow $u_{ij} - x_{ij}$, the other from j to i with cost $-c_{ij}$ and flow x_{ij}.

There are three equivalent statements of the optimality of flow x^* for the minimum cost flow problems.

1. The following are equivalent to optimality of x^*.

   - x^* is feasible.
   - There is no negative cost cycle in $G(x^*)$.

2. The following are equivalent to optimality of x^*.
   There exists a potentials vector π such that,

   - x^* is feasible.
   - $c_{ij}^\pi \geq 0$ for each arc (i,j) in $G(x^*)$.

3. The following are equivalent to optimality of x^*.
   There exists a potentials vector π such that,

   - x^* is feasible,
   - $c_{ij}^\pi > 0 \Rightarrow x_{ij}^* = 0$.
   - $c_{ij}^\pi = 0 \Rightarrow 0 \leq x_{ij}^* \leq u_{ij}$.
   - $c_{ij}^\pi < 0 \Rightarrow x_{ij}^* = u_{ij}$.

Theorem 18.2 (Optimality condition 1)**.** A feasible flow x^* is optimal iff the residual network $G(x^*)$ contains no negative cost cycle.

Proof: $\Leftarrow$ direction.
Let x^0 be an optimal flow and $x^0 \neq x^*$. $(x^0 - x^*)$ is a feasible circulation flow vector. A circulation is decomposable into at most m primitive cycles. The sum of the costs of flows on these cycles is $c(x^0 - x^*) \geq 0$ since all cycles in $G(x^*)$ have non-negative costs. But that means $cx^0 \geq cx^*$. Therefore x^* is a min cost flow.

$\Rightarrow$ direction.
Suppose not, then there is a negative cost cycle in $G(x^*)$. But then we can augment flow along that cycle and get a new feasible flow of lower cost, which is a contradiction. $\qquad\square$

Theorem 18.3 (Optimality condition 2). A feasible flow x^* is optimal iff there exist node potentials π such that $c_{ij}^\pi \geq 0$ for each arc (i,j) in $G(x^*)$.

Proof: $\Leftarrow$ direction.
If $c_{ij}^\pi \geq 0$ for each arc (i,j) in $G(x^*)$, then in particular for any cycle C in $G(x^*)$, $\sum_{(i,j)\in C} c_{ij}^\pi = \sum_{(i,j)\in C} c_{ij}$. Thus every cycle's cost is nonnegative. From optimality condition 1 it follows that x^* is optimal.
$\Rightarrow$ direction.
x^* is optimal, thus (Opt cond 1) there is no negative cost cycle in $G(x^*)$. Therefore shortest paths distances are well defined. We add a source node s with arcs going to all nodes of the graph, each of cost 0 (say, any constant will work). Let $d(i)$ be the shortest paths distances from node s to node $i \in V$. Let the node potentials be $\pi_i = -d(i)$. From the validity of distance labels it follows that,

$$c_{ij} + d(i) \geq d(j)$$

Thus, $c_{ij}^\pi = c_{ij} - \pi_i + \pi_j \geq 0$.
$\qquad\square$

Theorem 18.4 (Optimality condition 3). A feasible flow x^* is optimal iff there exist node potentials π such that,

- $c_{ij}^\pi > 0 \Rightarrow x_{ij}^* = 0$

- $c_{ij}^\pi = 0 \Rightarrow 0 \leq x_{ij}^* \leq u_{ij}$

- $c_{ij}^\pi < 0 \Rightarrow x_{ij}^* = u_{ij}$.

Proof: $\Leftarrow$ direction.
If these conditions are satisfied then in particular for all residual arcs $(i,j) \in G(x^*)$, $c_{ij}^\pi \geq 0$. Hence, optimality condition 2 is satisfied.
$\Rightarrow$ direction.
From Optimality condition 2 $c_{ij}^\pi \geq 0$ for all (i,j) in $G(x^*)$. Suppose that for some (i,j) $c_{ij}^\pi > 0$ but $x_{ij}^* > 0$. But then $(j,i) \in G(x^*)$ and thus $c_{ji}^\pi = -c_{ij}^\pi < 0$ which is a contradiction (to opt con 2). Therefore the first condition on the list is satisfied. The second is satisfied due to feasibility. Suppose now that the third holds, and $c_{ij}^\pi < 0$. So then arc (i,j) is not residual, as for all residual arcs the reduced costs are nonnegative. So then the arc must be saturated and $x_{ij}^* = u_{ij}$ as stated. $\qquad\square$

Dual optimality: The node potentials vector π in Optimality conditions 2 or 3, is also a dual optimal solution. Specifically, for a node potentials vector π, if there exists a feasible flow x^*, such that in the residual network $G(x^*)$, $c_{ij}^\pi \geq 0$ for each arc $(i,j) \in G(x^*)$, then π is an optimal dual solution. Similarly, if there exists a feasible flow x^*, such that

- $c_{ij}^\pi > 0 \Rightarrow x_{ij}^* = 0$

- $c_{ij}^\pi = 0 \Rightarrow 0 \leq x_{ij}^* \leq u_{ij}$

- $c_{ij}^\pi < 0 \Rightarrow x_{ij}^* = u_{ij}$,

then π is an optimal dual solution.

The proof of that relies on the weak and strong duality theorems for $x*$ and π which makes an important use of property 2.4 (in the book), that says that for any feasible primal and dual vector, $x*$ and π,

$$\sum_{(i,j) \in A} c_{ij}^{\pi} x_{ij} = \sum_{(i,j) \in A} c_{ij} x_{ij} + \sum_{j \in V} b_j \pi(j).$$

18.5   Implied algorithms – Cycle cancelling

Optimality condition 1 implies an algorithm of negative cycle canceling: whenever a negative cost cycle is detected in the residual graph it is "canceled" by augmenting flow along the cycle in the amount of its bottleneck residual capacity.

Finding a negative cycle in the residual graph is done by applying the Bellman-Ford algorithm. If no negative cost cycle is detected then the current flow is optimal. In the simplex method it is easier to detect a negative cost cycle. Since all in-tree arcs have reduced cost 0, it is only necessary to find a negative reduced cost arc out of tree. So it seems that simplex is a more efficient form of negative cycle canceling algorithm, isn't it?

The answer is no. The reason is, that while simplex identifies a negative cost cycle more easily than by Bellman Ford algorithm, this cycle is not necessarily in the residual graph. In other words, there can be in the tree and in the basis, arcs that are at their lower or their upper bound. This is precisely the case of *degeneracy*. In that case, although a negative cost cycle was found, the bottleneck capacity along that cycle can be 0. In that case the simplex pivot leads to another solution of equal value.

The complexity of the **cycle canceling algorithm**, is finite. Every iteration reduces the cost of the solution by at least one unit. For maximum arc cost equal to C and maximum capacity value equal to U the number of iterations cannot exceed $2mUC$. The complexity is thus $O(m^2 nUC)$, which is not polynomial, but is only exponential in the length of the numbers in the input.

In order to make this complexity polynomial we can apply a *scaling algorithm*. The scaling can be applied to either capacities, or costs, or both.

18.6   Implied Algorithm: The Successive Shortest Path Algorithm for MCNF

The Successive Shortest Path algorithm maintains the invariant that the reduced costs, $c_{ij}^{\pi} = c_{ij} - \pi_i + \pi_j$, of all arcs in the residual graph are non-negative. Using optimality conditions 2, it remains to attain feasibility of the flow. When all costs c_{ij} are nonnegative, there is no further pre-processing. If not, we apply Bellman-Ford to find the shortest paths labels d_i from an added source node s that has 0 cost arcs to all nodes. The negative of these labels are then node potentials such that the reduced costs w.r.t. these node potentials are nonnegative. We then replace the costs in the original graph by these attained reduced costs. That is $c_{ij} \leftarrow c_{ij} + d_i - d_j$.

The initial flow is set to 0. The infeasibility is due to some node having excess (supply) and some nodes having deficit (demand). The algorithm iteratively sends flow from supply node to a demand node along a shortest path in the residual graph.

At each iteration, the total excess is reduced by at least one unit. Because the shortest path in the residual graph is computed for the reduced costs, that are nonnegative, one can employ Dijkstra's algorithm. Let $|D|$ denote the total initial deficit (equal to the total initial excess). The complexity is therefore $O(|D|(m + n \log n))$. Scaling algorithms that scale the amount of deficit/excess improve the complexity to depend on $\log |D|$ instead of $|D|$.

Algorithm 1 Successive Shortest Path Algorithm

1: $x \leftarrow 0$ and $\pi \leftarrow 0$

2: **for** each $i \in N$ **do**

3:     $e(i) \leftarrow b(i)$

4: **end for**

5: Initialize the sets $E \leftarrow \{i : e(i) > 0\}$ and $D \leftarrow \{i : e(i) < 0\}$

6: **while** $E \neq \emptyset$ **do**

7:     Select a node $k \in E$ and a node $l \in D$

8:     Determine shortest path distances $d(j)$ from node s to all other nodes in $G(x)$ with respect to the reduced costs c_{ij}^π

9:     Let P denote a shortest path from node k to node l

10:     Update $\pi \leftarrow \pi - d$

11:     $\delta \leftarrow \min[e(k) - e(l), \min\{c_{ij} : (i,j) \in P\}]$

12:     Augment δ units of flow along the path P

13:     Update x, $G(x)$, E, D, and the reduced costs

14: **end while**

The Successive Shortest Path algorithm is very efficient for problems that have a polynomial amount of excess. In particular, the complexity of solving the assignment problem using the successive shortest path algorithm is $O(n(m + n \log n))$.

18.7 Solving maximum flow as MCNF

We know how to represent the maximum flow problem as a minimum cost network flow. We add an arc of cost -1 and infinite capacity between t and s. Since all other costs in the network are 0 the total cost will be minimum when the flow on the arc from t to s is maximum. Maximum flow as minimum cost network flow is described in Figure 50
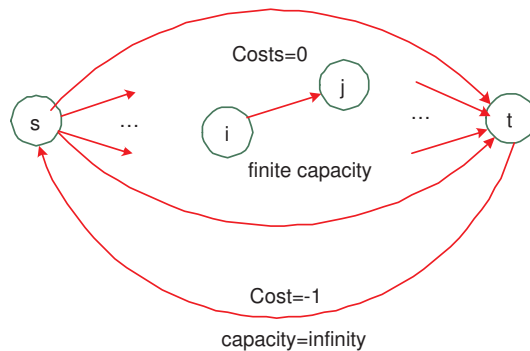

Figure 50: Maximum flow as MCNF problem

If we solve this MCNF problem by cycle canceling algorithm, we observe that the only negative cost arc is (t, s) and therefore each negative cost cycle is a path from s to t augmented with this arc. When we look for negative cost cycle of minimum mean, that is, the ratio of the cost to the number of arcs, it translates in the context of maximum flow to -1 divided by the length of the s, t path in terms of the number of arcs. The minimum mean corresponds to a path of smallest

number number of arcs – shortest augmenting path. You can easily verify the other analogies in the following table.

Minimum cost network flow	maximum flow
Negative cost cycle in $G(x)$	Augmenting path in $G(x)$
Cycle canceling algorithm	Augmenting path algorithm
Minimum mean cost cycle canceling	Shortest augmenting path
Most negative cost cycle	Augmenting path (all paths have same cost)

Notice that finding a most negative cost cycle in a network is NP-hard as it is at least as hard as finding a hamiltonian path if one exists. (set all costs to 1 and all capacities to 1 and then finding most negative cost cycle is equivalent to finding a longest cycle.) But finding a minimum mean cost cycle can be done in polynomial time. The mean cost of a cycle is the cost divided by the number of arcs on the cycle.

18.8 The gap between the primal and dual of uncapacitated MCNF

For the max flow min cut problem the gap between the primal and dual is significant, as discussed in the max-flow min-cut section. We now investigate the gap between primal and dual for uncapacitated MCNF.

We want to show that for an optimal flow $\mathbf{x}^*$ there is a corresponding set of potentials π_i and vice versa.

Flow $\rightarrow$ potentials: Let $\mathbf{x}^*$ be the flow. The residual graph $G(x^*)$ is connected as there must be at least one arc in the residual graph for each $(i, j) \in A$.

Let the length of the arc $(i, j) \in G(\mathbf{x}^*)$ be:

$$d_{ij} = \begin{cases} c_{ij} & \text{if } (i,j) \in A \\ -c_{ji} & \text{if } (j,i) \in A. \end{cases}$$

We now add to the graph $G(\mathbf{x}^*)$ a source node s with arcs of length 0 to all nodes $i \in V$. This does not introduce any negative cost cycle. In this graph the shortest path distances from node s, $d(i)$, are well defined since there are no negative cost cycles in $G(\mathbf{x}^*)$. Let $\pi_i = -d(i)$. Using Bellman-Ford algorithm the complexity of the shortest paths procedure is $O(mn)$.

Potentials $\rightarrow$ flow :
If $c_{ij}^\pi > 0$ then set $x_{ij}^* = 0$,
If $c_{ij}^\pi < 0$ then set $x_{ij}^* = u_{ij}$.
As for the remaining flows, they can be anywhere between the lower and upper bounds. By assigning the flows guaranteed to be at the upper and lower bounds, the supplies of nodes will be modified. We now use the maximum flow procedure to find a flow that is feasible on these arcs: connect excess nodes to source and deficit nodes to sink and find a flow that saturates these arcs, that is, – maximum. The complexity of this process is that of finding maximum flow, e.g. $O(mn \log \frac{n^2}{m})$.

If the potentials are a *basic* (optimal) solution, in the sense that the arcs with $c_{ij}^\pi = 0$ form an acyclic graph – a forest, then the potentials provide a basis that corresponds to the (T,L,U) structure. In this case the maximum flow needs to be solved on a tree-network. This can be done more efficiently than solving max-flow on a general network: Hochbaum's Pseudoflow algorithm, [Hoc08], was shown to solve the maximum-flow problem on s, t-tree networks in linear time, $O(n)$,

where s, t-tree networks are formed by joining a forest of capacitated arcs, with nodes s and t adjacent to any subset of the nodes. Notice that such linear gap, if the basis is given, holds also for the maximum flow reconstructed from min-cut solution. Therefore, solving a MCNF problem, and identifying the (T,L,U) basis structure have different complexities (the latter more than the former).

References

[Hoc08]   D.S. Hochbaum. The pseudoflow algorithm: A new algorithm for the maximum-flow problem. *Operations Research*, 56(4):992–1009, 2008.

19   Planar Graphs

Planar graph : A graph is planar if it can be drawn in the plane so no edges intersect. Such a drawing is called planar embedding.

Kuratowsky theorem(1927) *A graph is planar iff it contains no subgraphs homeomorphic to $K_{3,3}$ or K_5.*

Face : Faces are regions defined by a planar embedding, where two points in the same face can be reachable from each other along a continuous curve (that does not intersect any edges).

Chromatic number : The minimum number k for which G is k-colorable.

Critical graph : G is called critical if chromatic number of any proper subgraph of G is smaller than that of G.

k-critical graph : A k-critical graph is one that is k-chromatic and critical.

Theorem 19.1. *If G(not necessarily planar) is k-critical, then minimum degree $\geq k - 1$.*

Proof. See J.A.Bondy and U.S.R.Murty(1976)

Proposition 19.1. *In a planar graph, there is a node of degree ≤ 5. (Proof in next subsection)*

Theorem 19.2 (Heawood's 5-color theorem(1890))**.** *Every planar graph is 5-colorable.*

Proof. By induction on n, the number of nodes. If $n \leq 5$ then the graph is 5-colorable.

For a graph $G = (V, E)$ with $n > 5$, assume that $G - \{v\}$ is 5-colorable for any $v \in V$. Now, if for all v, $\deg(v) \leq 4$ or less than 5 colors are used by the neighbors of v, then G is 5-colorable and we are done.

Suppose G is not 5-colorable, that is, G is 6-critical. By above property and theorem, there exists a node, say v, of degree 5. Because G is 6-critical, there exists a 5-vertex coloring, that is a partitioning of into 5 sets each of a different color, of $G - \{v\}$, say $(V_1, V_2, V_3, V_4, V_5)$. Since G is not 5-colorable, v must be adjacent to a vertex of each of the five colors. Therefore we can assume that the neighbors of v in clockwise order about v are v_1, v_2, v_3, v_4 and v_5, where $v_i \in V_i$ for $1 \leq i \leq 5$. Let $G_{1,3}$ be the subgraph of nodes of V_1 and V_3. Now v_1 and v_3 must belong to the same connected component of $G_{1,3}$. For, otherwise, consider the component of $G_{1,3}$ that contains v_1 (see Figure 51). By interchanging the colors 1 and 3 in this component we obtain a new coloring of $G - \{v\}$
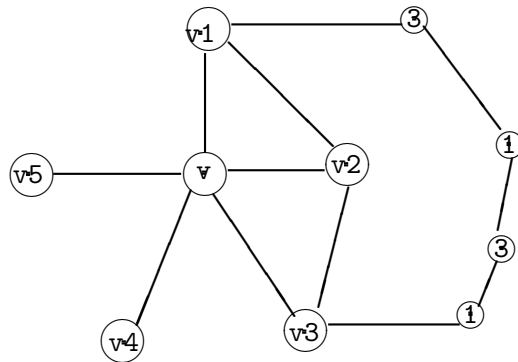
Figure 51: Illustration for proof of 5-Color theorem

in which only 4 colors are assigned to the neighbors of v and G is 5-colorable. Therefore v_1 and v_3 must belong to the same component of $G_{1,3}$.

Let $P_{1,3}$ be a (v_1, v_3)-path in $G_{1,3}$, and let C denote the cycle $vv_1P_{1,3}v_3v$. Since C separates v_2 and v_4, $G_{2,4}$ has at least two separate components, one inside C and another outside C. By interchanging the colors of the component of $G_{2,4}$ in C that contains v_2 we can color v by color 2.

19.1   Properties of Planar Graphs

Proposition 19.2 (Euler's Formula). *For a connected planar graph on n nodes, m edges, and f faces, $f = m - n + 2$ (8.6 in text)*

Proof. By induction on the number of faces. If $f = 1$ then the graph is a spanning tree, and it contains $n - 1$ edges. The equation is then satisfied. Assume, inductively, the formula is true for $f \leq k$

Now, we show that the formula is valid for every graph with $k + 1$ faces. Consider a graph G with $k + 1$ faces, n nodes, and $m + 1$ edges. Because the graph is not a tree, there exists a cycle. Let edge e be in a cycle, then it separates two faces, the one contained in the cycle and the other one. Removing edge e results in a graph having one fewer faces and one fewer edges. Therefore the resulting graph has k faces, n nodes, and m edges. By the induction hypothesis, $k = m - n + 2$. Therefore, if we add e back into the resulting graph, $k + 1 = m + 1 - n + 2$.

Proposition 19.3 (Sparsity). *$m < 3n$ (8.7 in text)*

Proof. This property is proved by contradiction

Suppose $(m \geq 3n)$

Since the network is a simple graph, the boundary of each face contains at least three arcs. If we count arcs on the boundaries of all the faces one by one, there are at least $3f$ arcs. Also, each arc is counted at most twice because it belongs to at most two faces. Therefore, $3f \leq 2m$, or equivalently, $f \leq \frac{2m}{3}$.

By Euler's formula $f = m - n + 2$, so $\frac{2m}{3} \geq m - n + 2$. This, however, implies that $3n \geq m + 6$ which contradicts the supposition that $m \geq 3n$.
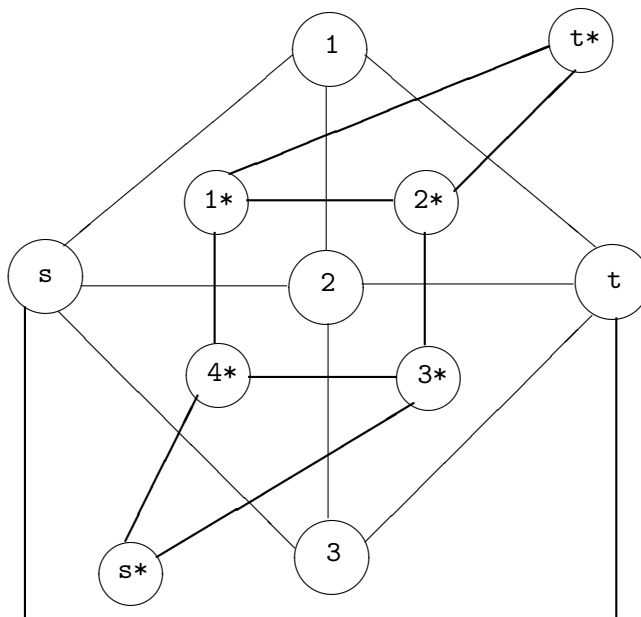
Figure 52: Geometric Dual of a Graph

Corollary 19.4. *In a planar graph, exist a node of degree ≤ 5*

Proof. Planar graphs are sparse, so $m < 3n$. Let d_i is the degree of node i. $\sum_{i=1}^{n} d_i = 2m < 6n$ Let $d = \frac{\sum_{i=1}^{n} d_i}{n}$ be the average degree. Sparsity ensures that $d < 6$. Since there must be at least one node j such that $d_j \leq d < 6$. Since d_j is an integer, it follows that $d_j \leq 5$.

Planarity is a hereditary property - after removing a node, or removing an edge, from a planar graph, the remaining graph is still planar.

19.2   Geometric Dual of Planar Graph

The geometric dual of a planar graph is defined by construction:

- 1. Place a node in each face, including the exterior face

- 2. Two nodes are adjacent if the corresponding faces share an edge

The original planar graph is called the primal and the newly constructed graph is called the dual. The dual of a planar graph is still planar.

The followings are some interesting points of primal and dual

- 1. # of nodes in dual = # of faces in primal

- 2. For simple graphs,
  # edges in the primal $\leq$ # edges in the dual

- 3. If no path of length ≥ 2, separates 2 faces,
  then the dual of dual is the primal.

19.3 s-t Planar Flow Network

For an undirected s-t planar flow network s and t are on the exterior face (or on the same face which can then be embedded as an exterior face)

- 1. Add directed arc (t, s) with infinite capacity.

- 2. take the dual of the graph.

- 3. call the two nodes in the two faces created by the new edge e s^* and t^*.

19.4 Min Cut in an Undirected Weighted s-t Planar Flow Network

Weight of edge in dual = capacity of edge in primal

Any path between s^* and t^* corresponds to a cut separating s and t

capacity of cut in primal = cost of path in dual

Shortest path between s^* and t^* corresponds to a minimum cut separating s and t

Since all capacities in primal are nonnegative, all edge costs in dual are nonnegative. Dijkstra's Algorithm can be applied to find the shortest path. The running time for Dijkstra's Algorithm is $O(m + nlogn) = O(3n + nlogn) = O(nlogn)$.

The above algorithm for min cut in planar graph makes no use of the max flow.

19.5 Max Flow in a Undirected Weighted s-t Planar Flow Network

Shown by Hassin in 1981 that max flow can also be derived from shortest path labels.

In the original graph, let U_{ij} be the flow capacity of arc(i,j) X_{ij} be the flow from node i to node j. In an undirected graph, the flow capacity for an edge is $(0, U_{ij})$ for both directions. Therefore $-U_{ij} \le X_{ij} \le U_{ij}$

Consider the edge (i,j) in the direction from i to j, let i^* be a dual node in the face to the left of the arc and j^* to the right and define $d(i^*)$ to be the shortest path from s^* to i^* in the dual graph, then set:

$X_{ij} = d(i^*) - d(j^*)$.

Claim 19.5. $X_{ij} = d(i^*) - d(j^*)$ *are feasible flows.*

Proof.

The capacity constraints are satisfied:

Since $d(i^*)$ is the shortest path from s^* to i^* in the dual graph, $d(i^*)$ satisfies the shortest path optimality condition:

$d(i^*) \le d(j^*) + c_{i^*j^*}$. Since cost of path in dual = capacity of cut in primal

$$d(i^*) \le d(j^*) + U_{ij}$$
$$d(j^*) \le d(i^*) + U_{ij}$$
$$-U_{ij} \le X_{ij} \le U_{ij}$$

The flow balance constraints are satisfied:

For a node i, let's label it's neighbors 1, 2, ...,p and create the dual graph with nodes 1^*, 2^*, ... p^* (see Figure 53).

$$X_{i1} = d(1^*) - d(2^*)$$
$$X_{i2} = d(2^*) - d(3^*)$$
$$\dots$$
$$X_{ip} = d(p^*) - d(1^*).$$

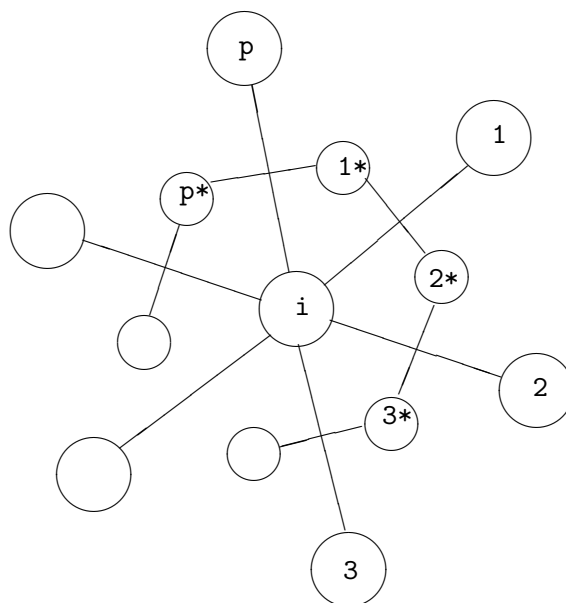Sum up all the equations we have: $\sum_{k=1}^{p} X_{ik} = 0$.

Figure 53: Dual graph for max-flow

For edges on the shortest path in dual, identify the min cut, the flow on the cut edges is saturated, therefore, flow value = cut capacity = max flow

Again we can use Dijkstra's Algorithm to find the shortest path, which is also the max flow. The running time is $O(nlogn)$ as we justified in min cut algorithm.

20 Cut Problems and Algorithms

20.1 Network Connectivity

Network connectivity problems consist of arc connectivity problems and node connectivity problems.

1. The arc (edge) connectivity of a connected network is the minimum number of arcs whose removal from the network disconnects it into two or more components. Unweighted arc (edge) connectivity has arc (edge) weights=1.

2. The node connectivity of a network equals the minimum number of nodes whose deletion from the network disconnects it into two or more components.

Minimum 2-Cut A Minimum 2 cut in a directed network is the min weight arc-connectivity in the network, which is the smallest collection of arcs that separates one non-empty component of the network from the other. That is for each pair of nodes (u, v), find the minimum weight $u - v$ cut and select the minimum of these minimum cuts from among all pairs (u, v).

An Algorithm for Finding a Minimum 2-Cut Find for all pairs $v, u \in V$, the (v-u) - min cut in the network using $O(n^2)$ applications of Max-Flow. This can, however, be done with $O(n)$ applications of Min-Cut. Pick a node, say 1, and find the min-cut that separates 1 and j, for all $j \neq 1$. From among these pairs, take the smallest minimum cut. This requires considering $n - 1$ pairs and apply min-cut to each.

20.2 Matula's Algorithm

Given an unweighted, undirected graph $G = (V, E)$. We introduce the notation $\alpha(G)$ is the connectivity of the graph, which is also the capacity of the minimum 2-cut in the graph. $\alpha(A, B)$ for $A, B \subset V$ disjoint subsets, is the minimum cut separating A and B. Such cut can be found by shrinking A into a source node s, and b into a sink node t and solve the resulting s, t cut problem.

For $\delta = \min_{v \in V} degree(v)$, δ is obviously an upper bound on the connectivity of the graph. Let $nonneighbor(S)$ be the set of nodes that are *not* adjacent to nodes of S, and let $\alpha(G) = \alpha(S^*, \bar{S}^*)$. The following lemma is the basis of Matula's algorithm for the unweighted min 2-cut problem.

Lemma 20.1 (8.12 in Ahuja, Magnanti, Orlin). *Let $(S^*, \bar{S}^*)$ be a minimum unweighted 2-cut, then either $\alpha(G) = \delta$ or, $\forall S \subseteq S^*$ nonneighbor$(S) \neq \emptyset$*

Proof: Assume that $\alpha(G) < \delta$. Then,

$$\delta|\bar{S}^*| \leq \sum_{v \in \bar{S}^*} degree(v) \leq |\bar{S}^*|(|\bar{S}^*| - 1) + \alpha(G).$$

Subtract δ from both sides,

$$\delta(|\bar{S}^*| - 1) \leq |\bar{S}^*|(|\bar{S}^*| - 1) + \alpha(G) - \delta.$$

Rearranging the terms,

$$1 \leq \delta - \alpha(G) \leq (|\bar{S}^*| - 1)(|\bar{S}^*| - \delta).$$

Therefore both terms on the right must be greater or equal to 1. In particular, $|\bar{S}^*| \geq \delta + 1$. But the number of neighbors of S^* is less than or equal to δ, thus, $nonneighbor(S) \neq \emptyset$. $\qquad\square$

Matula's Algorithm for undirected unweighted min 2-cut

$p :=$ min degree node
$\delta \leftarrow degree(p)$
$S^* = \{p\}$, $S = \{p\}$, $\alpha^* \leftarrow \delta$.

While nonneighbor$(S) \neq 0$
  Let $k \in$ nonneighbor(S)
  Find min-cut $(S, k) = (S_k, \bar{S}_k)$ of value $\alpha[S, k]$
  If $\alpha^* > \alpha[S, k]$, then Do
    $\alpha^* \leftarrow \alpha[S, k]$, $S^* \leftarrow S_k$
  Else continue
  $S \leftarrow S \cup \{k\}$
  update in-degrees of neighbors of v

Lemma 20.2. *The algorithm is correct and its complexity is $O(mn)$.*

Proof. First we demonstrate correctness. If $\alpha(G) = \delta$ then done, as we never reduce the value of the cut. Assume then that $\alpha(G) < \delta$. We need to demonstrate that at some iteration $S \subseteq S^*$ and the selected nonneighbor k satisfies that $k \in \bar{S}^*$. This will suffice as for that iteration $\alpha[S, k] = \alpha^*$.

Initially $S = \{p\} \subseteq S^*$; when the algorithm terminates $S \not\subseteq S^*$, otherwise *nonneighbor*$(S) \neq \emptyset$, (see lemma). Therefore there must be an iteration i which is the last one in which $S \subseteq S^*$. Consider that iteration and the vertex $k \in noneighbor(S)$ selected. Since $S \cup \{k\} \not\subseteq S^* \Rightarrow k \in \bar{S}^*$. Thus, $\alpha[S, k] = \alpha^*$.

Consider now the complexity of the algorithm: Consider augmentations along shortest augmenting path (such as in Dinic's algorithm). Start a BFS in the graph from the source "node" S. The first layer of children is neighbor(S), denoted $N(S)$. Let k be the first node identified in the next layer of children of neighbor(S). Obviously k is a nonneighbor. Also there are augmenting paths of length 2 from S. Scan the set of neighbors of k, $A(k) = \{j_1, \ldots, j_{n_k}\}$. Each node in $A(k)$ that is also in $N(S)$ corresponds to an augmenting path of length 2 and we augment the flow and update the residual graph at $O(1)$ per augmentation. Therefore, throughout the algorithm, all augmentations along paths of length 2 can be accomplished at $O(m)$.

Overall, there are two types of augmenting paths:
-type1: its last internal node (before k) $\in neighbor(S)$
-type2: its last internal node (before k) $\in noneighbor(S)$.

Above we showed that all type1 augmentations are done in $O(m)$. Now, there are at most $O(n)$ type2 path augmentations throughout the algorithm: Consider a path $\{S\}, \ldots, \ell, k$ for ℓ the node just preceding k. When finding min-cut (S, k), node ℓ can be used just once since as soon as we are done with k it is added to S and ℓ becomes a neighbor. Therefore ℓ can be before the last node only in one type2 path. The complexity of finding an augmenting path is $O(m)$, so the total work for a type2 path is $O(mn)$. The total complexity of the algorithm is hence $O(mn)$.

20.3 Brief Review of Additional Results

Undirected weighted 2-cut: $O(mn + n^2 logn)$ by Nagamochi and Ibaraki, [NI92], [NI92a]. There is no use of the concepts of the max-flow min-cut theorem in their algorithm: It applies preprocessing that removes from the graph all but $O(kn)$ arcs without changing the value of the min 2-cut and then proceed by constructing a collection of $O(k)$ spanning trees.

Directed and undirected weighted 2-cut: $O(n^3)$ or $O(mn \log(n^2/m))$ by Hao and Orlin [HO92]. The idea is to use preflow while maintaining distance labels when values are updated.

References

[HO92]   J. Hao and J. B. Orlin. A faster algorithm for finding the minimum cut of a graph. In *Proc. 3rd ACM-SIAM Symp. On Discrete Alg.* 165–174, 1992.

[NI92]   H. Nagamochi and T. Ibaraki. Linear time algorithms for finding sparse k-connected spanning subgraph of a k-connected graph. *Algorithmica,* 7 (5/6):583-596, 1992.

[NI92a] H. Nagamochi and T. Ibaraki. Computing edge connectivity in multigraphs and capacitated. *SIAM J. Discrete Math,* 5:54–66, 1992.