Combinatorial Algorithms

Lectures at Summer School on Graph Theory, Xuzhou, July 22 – August 15, 2017

Xujin Chen
xchen@amss.ac.cn
http://people.gucas.ac.cn/∼xchen

Contents

1 Introduction to combinatorial algorithms

The concept of an algorithm can be expressed in terms of a Turing Machine or some other formal model of computation. The intuitive notion of an algorithm is a list of instructions to solve a problem. In this short course, we are going to study *combinatorial algorithms* – the algorithms that investigate combinatorial structures and the algorithms that uses only combinatorial operations.

1.1 Combinatorial structures

Roughly speaking, many structures we study in this course can be described as collection of k-element subsets or permutations from a parent set. Good general textbooks on combinatorics include [1, 2, 35].

Sets and lists. Let X be a (finite) set of cardinality $|X|$, and k be a nonnegative integer at most $|X|$. A *k-subset* of X is a subset of X that has cardinality k.

A (finite) *list* is an ordered collection of objects which are called the *items* of the list. The *length* of a list L is the number of items (not necessarily distinct) in L. A list of length k is often called a *k-tuple*.

The *Cartesian product* of the sets X and Y is $X \times Y = \{[x, y] : x \in X \text{ and } y \in Y\}$. A *permutation* (resp. *k-permutation*) of set X is a list of length $|X|$ (resp. k) such that every element of X occurs exactly once (resp. at most once) in the list.

Graphs and hypergraphs. We are often interested in substructures (subgraphs) such as Hamiltonian cycles, cliques, independent sets ... A graph $G = (V, E)$ is *weighted* if it is associated with edge weight $w : E \to \mathbb{R}$ or/and vertex weight $w : V \to \mathbb{R}$. The weight of a subgraph is the total weight of its edges (vertices). Hypergraphs are also known as set systems.

1.2 Combinatorial problems

The purposes of combinatorial algorithms could be roughly classified into three categories. (1) *Generation*: constructing all the combinatorial structures of a particular type. (2) *Enumeration* (counting): computing the number of different structures of a particular type. (3) *Search*: finding at least one example of a structure of a particular type (if exists). The combinatorial search problems are of various types, depending on the nature of the desired answer. Some recommended books on combinatorial optimization problems include [3, 21, 25].

Decision problem. A question is to be answered "yes" or "no". All that is required for an algorithm to solve a decision problem is to provide the correct answer ("yes" or "no") for any problem instance.

Search problem. It is exactly the same as the corresponding decision problem, except that we are asked to find an *evidence* (*feasible solution*) in the case where the answer to the decision problem is "yes".

Optimal value problem. In this version of the problem, there is no target objective value specified in the problem instance. Instead, it is desired to find the best objective value such that the decision problem will have the answer "yes".

Optimization problem. It is similar to the optimal value problem, except that it is required to find a feasible solution that yields the optimal value.

1.3 Big O notations

Big O notation is a mathematical notation that describes the limiting behavior of a function when the argument tends towards a particular value or infinity. It is a member of a family of notations invented by German number theoretician Edmund Landau (1894) and called *Landau's notation* or *asymptotic notation*. It is a symbolism used in complexity theory, computer science, and mathematics to describe the asymptotic behavior of functions. Basically, it tells you how fast a function grows or declines.

In computer science, big O notation is used to classify algorithms by how they respond to changes in input size, such as how the processing time of an algorithm changes as the problem size becomes extremely large. In analytic number theory it is used to estimate the "error committed" while replacing the asymptotic size of an arithmetical function by the value it takes at a large finite argument.

Formal definition. Suppose $f(x)$ and $g(x)$ are two functions defined on some subset of the real numbers. We write

$$f(x) = O(g(x)) \text{ (or } f(x) = O(g(x)) \text{ as } x \to \infty \text{ to be more precise)}$$

if and only if there exist constants N and C such that $|f(x)| \leq C|g(x)|$ for all $x > N$. Intuitively, this means that f does not grow faster than g. Given real number a, $f(x) = O(g(x))$ as $x \to a$ can be defined similarly.

Related notations. In addition to the big O notations, another Landau symbol is used in mathematics: the little o. Informally, $f(x) = o(g(x))$ means that f grows much slower than g and is insignificant in comparison. Formally, we write

$$f(x) = o(g(x)) \text{ as } x \to \infty$$

if and only if for every $C > 0$ there exists a real number N such that for all $x > N$ we have $|f(x)| < C|g(x)|$. If $g(x) \neq 0$, this is equivalent to

$$\lim_{x \to \infty} \frac{f(x)}{g(x)} = 0.$$

Notation	Definition	Analogy
$f(x) = O(g(x))$	see above	$\leq$
$f(x) = \Theta(g(x))$	$f(x) = O(g(x))$ and $g(x) = O(f(x))$	$=$
$f(x) = \Omega(g(x))$	$g(x) = O(f(x))$	$\geq$
$f(x) = o(g(x))$	see above	$\ll$
$f(x) = \omega(g(x))$	$g(x) = o(f(x))$	$\gg$

Exercise 1.1. *Prove that the function $n!$ is $\Theta(n^{n+1/2}e^{-n})$.*

HINT: It follows from Stirling's formula: $\sqrt{2\pi n}\, e^{\frac{1}{12n+1}} (\frac{n}{e})^n \leq n! \leq \sqrt{2\pi n}\, e^{\frac{1}{12n}} (\frac{n}{e})^n$.

1.4 Algorithm analysis

A good book discussing the analysis of algorithms is [26]. An important part of the analysis of an algorithm is to describe how the running time of the algorithm behave as a function of the size of the input.

Rapid changes in computer architecture make it nearly pointless to measure all running time in terms of a articular machine. For this reason we measure running time on an abstract computer model where we count the number of "elementary" operations in the execution of the algorithm. Roughly speaking,

an elementary operation is one for which the amount of work is bounded by a constant, that is, it is not dependent on the size of problem instance. (However, for the arithmetic operations of addition, multiplication, division, and comparison, we usually make an exception to this rule and count such operations as having unit cost, that is, the length of the numbers involved does not affect the cost of operation.) Take the following sorting algorithm as an example.

Algorithm 1.1. INSERTIONSORT(A, n)
INPUT: *An array* $A = [A[1], ..., A[n]]$ *of* n *numbers.*
OUTPUT: *A sorted array* A *of* n *numbers in increasing order.*

```
for i ← 2 to n do                    //At the end of each iteration, [A[1], ..., A[i]] is a sorted array
  x ← A[i]
  j ← i − 1
  while j ≥ 1 and A[j] > x do
    A[j + 1] ← A[j]                  //Move the jth element backward
    j ← j − 1
  end-while
  A[j + 1] ← x
end-for
```

Suppose that within any iteration of the while loop, a constant amount of time, say c_1, is spent, and that c_2 is the amount of time used within any iteration of the for loop excluding the time spent in the while loop.

Worst-case complexity. The running time of the entire algorithm is at most

$$T(n) = \sum_{i=1}^{n}(c_2 + c_1(i-1)) = c_2(n-1) + \frac{c_1 n(n-1)}{2} = \Theta(n^2).$$

The growth rate of an algorithm's running time is called the (*time*) *complexity* of the algorithm. Thus Algorithm 1.1 has quadratic complexity.

Average-case complexity. This would be determined by looking at the amount of time the algorithm requires for each of the $n!$ possible permutations of n given numbers, and the computing the average of these $n!$ quantities. *Suppose without loss of generality that* A *is a permutation of* $\{1, 2, \ldots, n\}$. For each $i = 2, \ldots, n$, let

$$N(A, i) = |\{j : 1 \le j \le i-1, A[j] > A[i]\}|.$$

The average running time of Algorithm 1.1 over all $n!$ permutations A is

$$\begin{aligned}
\frac{1}{n!}\sum_{A}\sum_{i=2}^{n}(c_2 + c_1 N(A,i)) &= (n-1)c_2 + \frac{c_1}{n!}\sum_{i=2}^{n}\sum_{A} N(A,i) \\
&= (n-1)c_2 + \frac{c_1}{n!}\sum_{i=2}^{n}\sum_{j=1}^{i-1}|\{A : A[j] > A[i]\}| \\
&= (n-1)c_2 + \frac{c_1}{n!}\sum_{i=2}^{n}(i-1)\frac{n!}{2} \\
&= (n-1)c_2 + \frac{c_1 n(n-1)}{4} = \Theta(n^2),
\end{aligned}$$

where the third equation follows from the fact that exactly half of the $n!$ permutations will have $A[j] > A[i]$ (there is a 1-1 correspondence between permutations with $.A[j] > A[i]$ and those with $A[j] < A[i]$).

Input size. An algorithm consists of a set of valid inputs and a sequence of instructions each of which can be composed of elementary operations, such that for each valid input the computation of the algorithm is a uniquely defined finite series of elementary operations which produces a certain output. Usually we are not satisfied with finite computation but rather want a good upper bound on the number of elementary operations performed, depending on the input size.

The input to an algorithm usually consists of a list of numbers. If all these numbers are integers, we can code them in binary representation, using $O(\log(|n|+2))$ bits for sotring an integer n. Rational numbers can be stored by coding the numeration and the denominator separately. The *input size* $\mathsf{size}(x)$ of an instance x with rational data is the total number of bits needed for the binary representation. The *sizes* of a rational number $\alpha = p/q$ (where p and q are relatively prime integers), of a rational vector $\beta = (\beta_1, \ldots, \beta_n)$ and of a rational matrix $A = (\alpha_{ij})_{m \times n}$ are

$$\begin{aligned} \mathsf{size}(\alpha) &= 1 + \lceil \log_2(|p|+1) \rceil + \lceil \log_2(|q|+1) \rceil, \\ \mathsf{size}(\beta) &= n + \sum_{i=1}^{n} \mathsf{size}(\beta_i), \\ \mathsf{size}(A) &= mn + \sum_{i=1}^{m} \sum_{j=1}^{n} \mathsf{size}(\alpha_{ij}). \end{aligned}$$

Exercise 1.2. *Determine the input size of a graph with n vertices and m edges.*

HINT: Adjacency matrix: $\Theta(n^2)$. Incidence matrix: $\Theta(mn)$. Adjacency list: $\Theta(n + m \log n)$.

Polynomial time algorithms. An algorithm with rational input is said to *run in polynomial time* if there is an integer d such that it runs in $O(n^d)$ time, where n is the input size, and all numbers in intermediate computations can be stored with $O(n^d)$ bits. An algorithm with arbitrary input is said to *run in strongly polynomial time* if there is an integer d such that it runs in $O(n^d)$ time for any input consisting of n numbers and it runs in polynomial time for rational input. Polynomial time algorithms are often called "good" or "efficient".

Notation	**Name**	**Example**
$O(1)$	*constant*	Calculating $(-1)^n$
$O(\log \log n)$	*double logarithmic*	Number of comparisons spent finding an item in a sorted array of uniformly distributed values
$O(\log n)$	*logarithmic*	Finding an item in a sorted array
$O(\log^c n)$	*polylogarithmic*	Matrix chain ordering
$O(n)$	*linear*	Finding an item in an unsorted list
$O(n^2)$	*quadratic*	Insertion sort
$O(n^c)$	*polynomial* or *algebraic*	Fining a maximum weight matching
$O(c^n)$	*exponential*	Solving TSP using dynamic programming
$O(n!)$	*factorial*	Solving TSP via brute-force search

1.5 Complexity classes

In general, we hope to find algorithms having polynomial complexity, i.e., $\Theta(n^d)$ for some positive integer d. A *decision problem* can be described as a problem that requires a "yes" or "no" answer. The *class* P refers to the set of all decision problems for which polynomial-time algorithms exist. A larger class of decision problem is called *NP*. The *NP problems* have the property that, for any problem instance for which the answer is "yes", there exists a proof that the answer is "yes" which can be verified by a polynomial-time algorithm. (Note: P stands for "polynomial", while *NP* stands for "non-deterministic polynomial".)

Theorem 1.2. *$P \subseteq NP$.* □

Exercise 1.3. *Give some decision problems that are not in NP.*

HINT: For example "does the problem of Hanoi tower have a solution?"

P vs NP Problem. Is $P \neq NP$? This is one of the seven "Millennium Problems". For a correct solution of the "P versus NP" question, the Clay Mathematics Institute (CMI) will award a prize of $1.000.000. More information on this prize is available at http://www.claymath.org/millennium-problems. The P-versus-NP page http://www.win.tue.nl/∼gwoegi/P-versus-NP.htm collects links around papers that try to settle the "P versus NP" question (in either way).

If it is easy to check that a solution to a problem is correct, is it also easy to solve the problem? This is the essence of the P vs NP question. Typical of the NP problems is that of the Hamiltonian Path Problem: given N cities to visit, how can one do this without visiting a city twice? If you give me a solution, I can easily check that it is correct. But I cannot so easily find a solution.

NP-complete problems. Garey and Johnson [10] provides a very readable treatment of NP-completeness and related topics. The concept of NP-completeness is based on the idea of a polynomial transformation. Suppose D_1 and D_2 are both decision problems. A *polynomial transformation* from D_1 to D_2 is a polynomial-time algorithm, TRANSFORM, which when given any instance I of problem D_1, will construct an instance TRANSFORM(I) of problem D_2 in such a way that I is a yes-instance of D_1 if and only if Transform(I) is a yes-instance of problem D_2. We use notation $D_1 \propto D_2$ to indicate that there is a polynomial transformation from D_1 to D_2.

Property 1.3. *If $D_1 \propto D_2$ and D_2 is polynomial-time solvable, then D_1 is polynomial-time solvable.* □

A decision problem D is *NP-complete* if $D \in NP$, and $D' \propto D$ for any problem $D' \in NP$. Over the years, many decision problem have been shown to be NP-complete. For such problems, we will look for either slower, exponential-time exact algorithms or faster, polynomial-time approximation algorithms.

Reductions between problems. Informally, a *Turing reduction* or simply a *reduction* is a method of using an algorithm A_1 for one problem, say P_1, as a subroutine to solve another problem, say P_2. Note that the two problems P_1 and P_2 need not to be decision problems. The algorithm A_1 can be invoked one or more times, but the resulting algorithm, say A_2, should have the property that A_1 is polynomial-time if and only if A_2 is polynomial-time. Roughly, a Turing reduction establishes that P_1 is no more difficult to solve than P_2. The existence of a Turing reduction is written notationally as $P_1 \propto_T P_2$.

Property 1.4. *If $D_1 \propto D_2$, then $D_1 \propto_T D_2$.* □

An interesting exercise is to find Turing reductions between the different flavors of the KNAPSACK problem.

Problem 1.5. KNAPSACK (decision)

INPUT: Profits $p_1, p_2, \ldots, p_n$, weights $w_1, w_2, \ldots, w_n$, capacity W, and targeted profit P
QUESTION: Does there exist an n-tuple $[x_1, x_2, \ldots, x_n] \in \{0,1\}^n$ satisfying

$$\sum_{i=1}^{n} w_i x_i \leq W \text{ and } \sum_{i=1}^{n} p_i x_i \geq P?$$

Intuitively, the above decision version of KNAPSACK is easier than the following optimization version. A further investigation show that the two versions are actually equivalent.

Problem 1.6. KNAPSACK (optimization)

INPUT: Profits $p_1, p_2, \ldots, p_n$, weights $w_1, w_2, \ldots, w_n$, and capacity W.
GOAL: Find an n-tuple $[x_1, x_2, \ldots, x_n] \in \{0,1\}^n$ satisfying $\sum_{i=1}^{n} w_i x_i \leq W$ such that $\sum_{i=1}^{n} p_i x_i$ is maximized.

Property 1.7. KNAPSACK *(optimization)* $\propto_T$ KNAPSACK *(decision).* □

Assuming an algorithm for solving KNAPSACK (decision), an algorithm for solving the search version of KNAPSACK problem can be constructed to show a Turing reduction stated in the next property.

Problem 1.8. KNAPSACK (search)

INPUT: Profits $p_1, p_2, \ldots, p_n$, weights $w_1, w_2, \ldots, w_n$, capacity W, and targeted profit P.
GOAL: Find an n-tuple $[x_1, x_2, \ldots, x_n] \in \{0,1\}^n$ satisfying $\sum_{i=1}^{n} w_i x_i \leq W$ and $\sum_{i=1}^{n} p_i x_i \geq P$, if such an n-tuple exists

Property 1.9. KNAPSACK *(search)* $\propto_T$ KNAPSACK *(decision).* □

NP-hard problems. The analog of *NP*-complete problems among search and optimization problem are the *NP*-hard problems. A problem R is *NP-hard* if there exists a *NP*-complete problem D such that $D \propto_T R$. Since *NP*-complete problem KNAPSACK (decision) Turing reduces to KNAPSACK (optimization), it follows that KNAPSACK (optimization) is *NP*-hard.

1.6 Data structures

A data structure is an implementation or machine representation of a mathematical structure. In combinatorial algorithms, the choice of data structure can greatly affect the efficiency of an algorithm. Data structures and algorithms are discussed in numerous textbooks. A good general reference is [4].

1.6.1 Data structures for sets and lists

Consider a finite set $X = \{1, 2, \ldots, n\}$, where $|X| = n$. We want to perform on subsets of X the following operations:

- test membership of an element of $x \in X$ in a subset $S \subseteq X$;
- insert and delete elements from a subset S;
- compute intersections and unions of subsets;
- compute cardinality of subsets; and
- list the elements of a subset.

Sorted arrays. One obvious way to store a subset $S \subseteq X$ is as a sorted array. That is, we write $S = [S[1], S[2], ...]$ with $S[1] < S[2] < \cdots$.

We can keep track of the value $|S|$ in a separate auxiliary variable.

Since $|S|$ is kept up-to-date every time an element is inserted or deleted from S, it is clear that no separate computation is required to determine $|S|$. Listing the elements of S can be done in time $O(|S|)$. Testing the membership in S can be accomplished using a binary search, which takes time $O(\log |S|)$. For the insertion or deletion of an element y, a binary or linear search, followed by a shit of the elements greater than y, will accomplish the task in time $O(|S|)$. Intersection or union of two subsets S_1 and S_2 can be computed in time $O(|S_1| + |S_2|)$ by a simple merging algorithm.

Balanced trees. There are other implementations of sets, based on balanced trees, in which all the operations can be performed in time $O(\log |S|)$ (excepting for listing all the elements of the set). One of the most popular date structures to do this is the so-called red-black tree.

Bit arrays. Let $S \subseteq X$. Construct a bit array $B = [B[1], B[2], \ldots, B[n]]$ such that $B[i] = 1$ if $i \in S$, and $B[i] = 0$ if $i \notin S$. This representation requires one unit for each element in the ground set X, which is sometimes not so efficient.

Suppose that β is the number of bits in an unsigned integer word. In general, this is a machine-dependent quantity.

$$\text{For every } u \in X, \text{ let } i_u = \lfloor u/\beta \rfloor + 1,\ j_u = \beta - (u \mod \beta).$$

A *bit array representation* of a subset $S \subseteq X$ is actually an array A of $\omega = \lceil (n+1)/\beta \rceil$ unsigned integers, satisfying

$$u \in S \text{ if and only if the } j_u\text{-th bit of } A[i_u] \text{ is 1.}$$

As an example, consider $n = 20$, $\beta = 8$ and $S = \{1, 3, 11, 16\}$. We have $\omega = 3$, $i_1 = i_3 = 1$, $j_1 = 7$, $j_3 = 5$, $i_{11} = 2$, $j_{11} = 5$, $i_{16} = 3$, $j_{16} = 8$, and thus $A[1] = 01010000$, $A[2] = 00010000$, $A[3] = 10000000$. Note that the last (8th) bit of $A[3]$, which is 1, corresponds to $16 \in S$.

Given unsigned integers m and n, let $m \wedge n$ and $m \vee n$ denote the "bitwise boolean and" and "bitwise boolean or" of m and n, respectively. Let $m \ll j$ (resp. $m \gg j$) denote the shift left (resp. right) of m by j bits – filling the rightmost (resp. leftmost) j bits with 0's.

The following algorithm performs an operation that replaces S with $S \cup \{u\}$, where u may or may not belong to S before the operation is performed.

Algorithm 1.10. SETINSERT(A, u)
INPUT: *A bit array* $A = [A[1], ..., A[\omega]]$ *that represents subset* S *of* X*, and* $u \in X$*.*
OUTPUT: *A bit array* A *that represents* $S \cup \{u\}$*.*

$i \leftarrow \lfloor u/\beta \rfloor + 1$
$j \leftarrow \beta - (u \mod \beta)$
$A[i] \leftarrow A[i] \vee (1 \ll (j-1))$

The deletion operation of $S \leftarrow S \setminus \{u\}$ is accomplished by SETDELETE(A, u), which uses $A[i] \leftarrow A[i] \wedge \neg(1 \ll (j-1))$. Similar to the insertion and deletion, testing membership can also be done in $O(1)$ time.

Exercise 1.4. *Design an algorithm for membership testing with bit array representation.*

HINT:

$i \leftarrow \lfloor u/\beta \rfloor + 1$
$j \leftarrow \beta - (u \mod \beta)$
`Return` $A[i] \wedge (1 \ll (j-1))$

For two sets with bit array representations A and B, respectively, their union (resp. intersection) can be produced by the implementing

for $i \leftarrow 1$ **to** ω
  $C[i] \leftarrow A[i] \vee B[i]$ (resp. $A[i] \wedge B[i]$)
end-for
`Return` C

To compute the cardinality of a set S, we could run over all elements of X and count which ones are in S. This requires $\Omega(n)$ operations of membership testing. A more efficient approach is to precompute an array T (often called a *table*)

whose ith entry is the number of 1 bits in the unsigned integer with value i for $i = 0, 1, 2, \ldots, 2^\alpha - 1$,

where $\alpha \in [1, \beta]$ is a positive integer parameter at most β. Let

$$chunk = \underbrace{00\cdots0}_{\beta-\alpha}\underbrace{11\cdots1}_{\alpha}$$

be the unsigned integer whose rightmost α bits are all 1's and whose remaining $\beta - \alpha$ bits are all 0's. By utilizing the date stored in T, we obtain the following more time-efficient algorithm for computing the cardinality of a set.

Algorithm 1.11. SETSIZE(A)
INPUT: *A bit array $A = [A[1], ..., A[\omega]]$ that represents subset S of X.*
OUTPUT: *An integer size that equals $|S|$.*

```
size ← 0
for i ← 1 to ω do
    x ← A[i]
    while x ≠ 0 do
      size ← size + T[x ∧ chunk)
      x ← (x ≫ α)
    end-while
end-for
Return size
```

There is of course a space-time tradeoff with this approach. The algorithm runs faster as α becomes larger, but an increase in the size of α gives an exponential increase in the size of the table T. In practise $\alpha = 8$ is a convenient compromise.

Data structures for lists. We often use an array to store the elements of a list. If the elements of a list are distinct, the list is often treated as a set.

1.6.2 Data structures of graphs and hypergraphs

There are several convenient data structures for storing graphs and hypergraphs. The simplest on is a list (or set) of (hyper)edges. The *incidence matrix* of a hypergraph $\mathcal{G} = (\mathcal{V}, \mathcal{E})$ is the $|\mathcal{V}| \times |\mathcal{E}|$ matrix whose $[v, e]$-entry is 1 if $v \in e$, and 0 otherwise.

The next two data structures are only useful for graphs. The *adjacency matrix* of a graph $G = (V, E)$ is a $|V| \times |V|$ matrix whose $[u, v]$-entry is 1 if $\{u, v\} \in E$, and 0 otherwise. An *adjacency list* of a graph $G = (V, E)$ is a list A of $|V|$ items, corresponding to the vertices $v \in V$. Each item $A[v]$ is itself a list, consisting of the vertices adjacent to v. Usually, the order of the items within each list $A[v]$ is irrelevant. Hence in this case each $A[v]$ can be represented as a set.

1.7 Algorithm design techniques

In this course, we will mainly study five popular and useful design techniques for combinatorial algorithms: greedy strategies, dynamic programming, divide-and-conquer, primal-dual method, and iterative approaches.

Greedy strategies. The idea of a greedy algorithm is to build up a particular feasible solution by improving the objective function, or some other measure, as much as possible during each stage of the algorithm. This can be done, for example, when a feasible solution is defined as a list of items chosen from an underlying set. A feasible solution is constructed step by step, making sure at each stage that none of the constraints are violated. At the end of the algorithm, we should have constructed a feasible solution, which may or may not be optimal.

Even though the outcome of a greedy algorithm may not be very close to an optimal solution, greedy algorithms are still useful since they can provide nontrivial bounds on optimal solution.

Dynamic programming. Another method of solving optimization problems is dynamic programming. It requires being able to express or compute the optimal solution to a given problem instance I in terms of optimal solutions to smaller instances of the same problem. This is called a *problem decomposition*. The optimal solutions to all the relevant smaller problem instances are then computed and stored in a tabular form. The smallest instances are solved first, and at the end of the algorithm, the optimal solution to the original instance I is obtained.

Problem 1.12. KNAPSACK (optimization value)

INPUT: Profits $p_1, p_2, \ldots, p_n$, weights $w_1, w_2, \ldots, w_n \in \mathbb{Z}_+$, and capacity $W \in \mathbb{Z}_+$.
GOAL: Find the maximum value of $\sum_{i=1}^{n} p_i x_i$ subject to $[x_1, x_2, \ldots, x_n] \in \{0,1\}^n$ and $\sum_{i=1}^{n} w_i x_i \leq W$.

For each integer $j \in [1, n]$ and each integer $C \in [0, W]$, let $P[j, C]$ denote the optimal solution to KNAPSACK (optimal value) for the instance $(p_1, \ldots, p_j; w_1, \ldots, w_j; C)$. The basis of the dynamic programming algorithm is the following recurrence relation

$$P[j,C] = \begin{cases} P[j-1,C] & \text{if } j \geq 2 \text{ and } w_j > C; \\ \max\{P[j-1,C], P[j-1,C-w_j] + p_j\} & \text{if } j \geq 2 \text{ and } w_j \leq C; \\ 0 & \text{if } j = 1 \text{ and } w_j > C; \\ p_1 & \text{if } j = 1 \text{ and } w_j \leq C. \end{cases}$$

The dynamic programming algorithm proceeds to compute the following table of values:

$$\begin{bmatrix} P[1,0] & P[1,1] & \cdots & P[1,W] \\ P[2,0] & P[2,1] & \cdots & P[2,W] \\ \vdots & \vdots & & \vdots \\ P[n,0] & P[n,1] & \cdots & P[n,W] \end{bmatrix}$$

The value $P[n, W]$ is the solution to the problem instance. Note that each entry in the table is computed in time $O(1)$ using the recurrence relation, and hence the running time of the algorithm is $O(nW)$. The algorithm is not be practical if W is too large.

Exercise 1.5. *Prove that* KNAPSACK *(optimization)* $\propto_T$ KNAPSACK *(optimal value).*

HINT: Let ALG be an (exact) algorithm for KNAPSACK (optimal value). Given any instance I of KNAPSACK (optimization), ALG computes the optimal value of the instance, denoted as ALG(I). For each item i in I, let $I \setminus i$ denote the instance obtained from I by removing item i (and keeping the knapsack capacity unchanged). The following gives an algorithm for solving KNAPSACK (optimization) exactly.

---

for $i \leftarrow 1$ **to** n **do**
  if ALG(I) = ALG$(I \setminus i)$ **then** $I \leftarrow I \setminus i$
end-for
`Output the items in` I

Exercise 1.6. *Use a dynamic programming algorithm to solve the following instance of* KNAPSACK *(optimal value):*

profits	$1, 2, 3, 5, 7, 10$;
weights	$2, 3, 5, 8, 13, 16$;
capacity	30.

Then, using the table of values $P[j, C]$, solve KNAPSACK *(optimization) for the same instance.*

HINT: The optimal solution for the optimization version is $\{3, 4, 6\}$ with optimal value 18. See the following table for the values of $P[j, C]$ computed step by step.

	$j=1$	$j=2$	$j=3$	$j=4$	$j=5$	$j=6$
$C=1$	0	0	0	0	0	0
$C=2$	1	1	1	1	1	1
$C=3$	1	2	2	2	2	2
$C=4$	1	2	2	2	2	2
$C=5$	1	3	3	3	3	3
$C=6$	1	3	3	3	3	3
$C=7$	1	3	4	4	4	4
$C=8$	1	3	5	5	5	5
$C=9$	1	3	5	5	5	5
$C=10$	1	3	6	6	6	6
$C=11$	1	3	6	7	7	7
$C=12$	1	3	6	7	7	7
$C=13$	1	3	6	8	8	8
$C=14$	1	3	6	8	8	8
$C=15$	1	3	6	9	9	9
$C=16$	1	3	6	10	10	10
$C=17$	1	3	6	10	10	10
$C=18$	1	3	6	11	11	11
$C=19$	1	3	6	11	11	12
$C=20$	1	3	6	11	11	12
$C=21$	1	3	6	11	11	13
$C=22$	1	3	6	11	11	13
$C=23$	1	3	6	11	11	14
$C=24$	1	3	6	11	11	15
$C=25$	1	3	6	11	11	15
$C=26$	1	3	6	11	15	16
$C=27$	1	3	6	11	15	17
$C=28$	1	3	6	11	16	17
$C=29$	1	3	6	11	17	18
$C=30$	1	3	6	11	17	18

Divide-and-conquer The *divide-and-conquer* design strategy also utilizes a problem decomposition. In general, a solution to a problem instance I should be obtained by combining in some way solutions to one or more smaller instances of the same problem.

Algorithm 1.13. BINARYSEARCH(X, y, l, u)
INPUT: *A sorted list $X = [X[1], X[2], \ldots, X[n]]$ of integers with $X[1] < X[2] < \cdots < X[n]$, integer y, and integers l, u with $l \geq 1$ and $u \leq n$.*
OUTPUT: *The index m such that $X[m] = y$ or the declaration that y does not occur in X.*

```
if l > u then Return "y ∉ X"
  else m ← ⌊(l + u)/2⌋
       if X[m] = y then Return m
       else if X[m] < y then BinarySearch(X, y, m + 1, u)
            else BinarySearch(X, y, l, m − 1)
```

The time complexity of the algorithm is $O(\log n)$. Note that other divide-and-conquer algorithms may require solving more than one smaller instances in order to solve the original problem instance. The following algorithm is a divide-and-conquer algorithm that sorts an array in increasing order.

Algorithm 1.14. MergeSort(X, n)
INPUT: *An array X of n numbers.*
OUTPUT: *A sorted array $X = [X[1], X[2], \ldots, X[n]]$ such that $X[1] \le X[2] \le \cdots \le X[n]$*

```
if n = 1 then Return X
if n = 2
  then if X[1] > X[2]
         then T ← X[1], X[1] ← X[2], X[2] ← T                    //Swap X[1] and X[2]
  else m ← ⌊n/2⌋
       for i ← 1 to m do
         A[i] ← X[i]                                             //A holds the first half of X
       end-for
       for i ← m + 1 to n do
         B[i − m] ← X[i]                                         //B holds the second half of X
       end-for
       A ← MergeSort(A, m), B ← MergeSort(B, n − m)              //Recursions
       A[m + 1] ← ∞, B[n − m + 1] ← ∞
       i ← 1, j ← 1
       for k ← 1 to n
          if A[i] ≤ B[j] then X[k] ← A[i], i ← i + 1 else X[k] ← B[j], j ← j + 1
       end-for
Return X
```

Exercise 1.7. *Give a wort-case analysis of the running time for Algorithm 1.14.*

HINT: It follows from $T(n) = 2T(n/2) + \Theta(n)$ that $T(n) = \Theta(n \log n)$. □

Exercise 1.8. *Prove that* Knapsack *(optimal value)* $\propto_T$ Knapsack *(decision).*

HINT: Use the algorithm for Knapsack (decision) to conduct a binary search for the solution of Knapsack (optimal value). □

Exercise 1.9. Knapsack *(optimization)* $\propto_T$ Knapsack *(decision).*

HINT: Combining Exercises 1.5 and 1.8. □

2 Combinatorial generation

Often it is necessary to find good algorithms to solve problems such as generating all the subsets of a given set S. Related problems include generating all the permutations of S, or all the k-subsets of S.

Suppose that $\mathcal{S}$ is a finite set of cardinality $N = |\mathcal{S}|$. A *ranking function* or simply a rank function is a bijection

$$\mathsf{rank} : \mathcal{S} \to \{1, 2, \ldots, N\}.$$

A rank function defines a total ordering on the elements of $\mathcal{S}$ by the rule

$$s < t \text{ if and only if } \mathsf{rank}(s) < \mathsf{rank}(t).$$

Conversely, there is a unique rank function associated with any total ordering defined on $\mathcal{S}$. If rank is a ranking function defined on $\mathcal{S}$, there is a unique *unranking function* associated with the function rank – a bijection $\mathsf{unrank} : \{1, 2, \ldots, N\} \to \mathcal{S}$ which is the inverse of rank, satisfying

$$\mathsf{rank}(s) = i \Leftrightarrow \mathsf{unrank}(i) = s \text{ for all } s \in \mathcal{S} \text{ and all } i \in \{1, 2, \ldots, N\}.$$

Given a ranking function rank defined on $\mathcal{S}$, the *successor function*, which we denote as $\mathsf{successor}$, satisfies:

$$\mathsf{successor}(s) = t \Leftrightarrow \mathsf{rank}(t) = \mathsf{rank}(s) + 1.$$

We will follow the convention that $\mathsf{successor}(s)$ is undefined if $\mathsf{rank}(s) = N$. Once we have constructed a successor function, it is a simple matter to generate all the elements of $\mathcal{S}$.

2.1 Subsets

Suppose that $S = \{1, \ldots, n\}$ for some positive integer n. Let $\mathcal{S}$ consist of the 2^n subsets of S. Each $T \in \mathcal{S}$ is associated with its *characteristic vector* $\chi(T) = (x_1, \ldots, x_n)$ satisfying $x_i = 1$ if $i \in T$ and $x_i = 0$ otherwise for all $i = 1, \ldots, n$.

Lexicographic ordering. We are going to generate all the elements of $\mathcal{S}$ in lexicographic order. The *lexicographic ordering* on subsets of S is the one induced by the lexicographic ordering of the characteristic vectors of these subsets. With respect to this ordering, the rank of T could be defined as the integer whose binary representation is $\chi(T)$, i.e.,

$$\mathsf{rank}(T) = 1 + \sum_{i=1}^{n} x_i 2^{n-i}.$$

Conversely, unranking an integer $r \in [1, 2^n]$ requires coumputation of the subset T with $\mathsf{rank}(T) = r$.

Algorithm 2.1. SUBSETUNRANK(n, r)
INPUT: *The cardinality n of the ground set S, and integer $r \in [1, n]$.*
OUTPUT: *The rth subset T in the lexicographic ordering on $\mathcal{S} = 2^S$.*

$T \leftarrow \emptyset$, $r' \leftarrow r - 1$
for $i \leftarrow n$ **downto** 1
  if $r' \equiv 1 \pmod 2$ **then** $T \leftarrow T \cup \{i\}$
  $r' \leftarrow \lfloor r'/2 \rfloor$
end-for
`Return` T

Exercise 2.1. *Suppose that $n = 8$. Compute* $\mathsf{rank}(\{1, 2, 4, 6\})$ *and* $\mathsf{unrank}(181)$.

HINT: $\mathsf{rank}(\{1, 2, 4, 6\}) = 1 + 2^7 + 2^6 + 2^4 + 2^2 = 213$, $\mathsf{unrank}(181) = \{1, 3, 4, 6\}$

We have assumed that the ground set is $\{1, \ldots, n\}$. What would we do if we want to rank and unrank the subsets of some other n-element set?

Gray codes. Given two subsets $X, Y \subseteq S$, let $X \Delta Y$ denote the symmetric difference of X and Y. The *distance* between X and Y is defined as the Hamming distance between the vectors $\chi(X)$ and $\chi(Y)$, i.e.,

$$\mathbf{dist}(X, Y) = |X \Delta Y|.$$

A *minimal change ordering* on $\mathcal{S}$ is the one in which any two consecutive subsets have distance one. The characteristic vectors of the subsets in a minimal change ordering form a structure that is known as *Gray code*. There is another way to formulate the concept of minimal change ordering or Cray codes. Consider the n-dimensional unit cube, whose 2^n vertices are labeled by the 2^n binary vectors. The edges of this cube join vertices having Hamming distance equal to 1. Thus, a Cray code can be viewed as a Hamiltonian path in the n-dimensional unit cube. A good starting point for learning more abut Gray codes is [28].

Next we take a look at a nice class of Gray codes called the *binary reflected Gray codes*. We use $\mathsf{G}^n = [\mathsf{G}_1^n, \mathsf{G}_2^n, \ldots, \mathsf{G}_{2^n}^n]$ to denote the binary reflected Gray code for the 2^n binary n-tuples. The codes G^n are defined recursively. The base case

$$\mathsf{G}^1 = [0, 1].$$

Given G^{n-1}, set

$$\mathsf{G}^n = [0\mathsf{G}_1^{n-1}, \ldots, 0\mathsf{G}_{2^{n-1}}^{n-1}, 1\mathsf{G}_{2^{n-1}}^{n-1}, \ldots, 1\mathsf{G}_1^{n-1}]$$

Theorem 2.2. *For any positive integer n, G^n is a Cray code.* □

Exercise 2.2. *Prove Theorem 2.2.*

HINT: The hamming distance between two adjacent vectors is 1.

Algorithm 2.3. GRAYSUCCESSOR(T, n)
INPUT: *A subset T of $\{1, \ldots, n\}$.*
OUTPUT: *The successor U of T for the binary reflected Gray code G^n.*

```
if |T| is even then U ← TΔ{n}
  else j ← n
       while j ∉ T and j ≥ 2
         j ← j − 1
       end-while
       if j = 1 then Return "undefined"
                else U ← TΔ{j − 1}
Return U
```

Exercise 2.3. *Prove that Algorithm 2.3 computes the function* successor *for the Gray code G^n for every positive integer n.*

HINT: By induction on n, the base case can be easily checked. Assuming the correctness for G^{n-1}, we proceed to G^n. If $\mathsf{rank}(T) \leq 2^{n-1} - 1$, we are done by induction hypothesis. If $\mathsf{rank}(T) = 2^{n-1}$, then $\chi(T) = 01\underbrace{00\cdots0}_{n-2}$, i.e., $T = \{2\}$, and Algorithm 2.3 returns $U = \{1, 2\}$ with $\chi(U) = 11\underbrace{00\cdots0}_{n-2}$, which is indeed the successor of T. It remains to consider the case where $\mathsf{rank}(T) \geq 2^{n-1} + 1$ and $\chi(T) = 1t_2t_2 \ldots t_n$. Let S be the successor of T with $\chi(S) = 1s_2s_3 \ldots s_n$ (under G^n). Then $T \setminus \{1\}$ with $\chi(T \setminus \{1\}) = t_2t_2 \ldots t_n$ is the successor of $S \setminus \{1\}$ with $\chi(S \setminus \{1\}) = s_2s_2 \ldots s_n$ (under G^{n-1}).

- Case 1. $|S|$ is odd, i.e., $|S \setminus \{1\}|$ is even: By hypothesis, $T \setminus \{1\} = (S \setminus \{1\})\Delta\{n\}$. Hence $|T|$ is even, and $S = T\Delta\{n\} = U$.

- Case 2. $|S|$ is even, i.e., $|S \setminus \{1\}|$ is odd: By hypothesis, $T \setminus \{1\} = (S \setminus \{1\})\Delta\{j-1\}$, where $\{j, j+1, \ldots, n\} \cap (S \setminus \{1\}) = \{j\}$. It follows that $|T|$ is odd, and $S = T\Delta\{j-1\} = U$.

Lemma 2.1. *Given $n, r \in \mathbb{N}$ with $r \le 2^n$, if the binary representations of G_r^n and $r-1$ are $a_1 a_2 \ldots a_n$ and $b_0 b_1 \ldots b_{n-1} b_n$ respectively,*[1] *then for every $j = 1, 2, \ldots, n$,*

$$a_j \equiv b_j + b_{j-1} \pmod 2 \tag{2.1}$$

and

$$b_j \equiv \sum_{i=1}^{j} a_i \pmod 2 \tag{2.2}$$

Proof. We first prove (2.1) by induction on n. The base case where $n = 1$ is trivial. Assume that (2.1) is true for $n = k-1 \ge 1$ and $j = 1, \ldots, k-1$. We consider $n = k$.

Case 1. If $r \le 2^{k-1}$ (i.e., we consider the first half of G^k), then a_1, b_1, b_0 are all zero, and (2.1) holds for $j = 1$. For $2 \le j \le k$, it reduces to the case of G^{k-1}, and (2.1) is true by induction hypothesis.

Case 2. It remains to consider the case where $2^{k-1} + 1 \le r \le 2^k$ (the second half of G^k). Note that

$$a_1 = 1,\ b_1 = 1,\ b_0 = 0,\ \mathsf{G}^{k-1}_{2^k+1-r} = a_2 \ldots a_{k-1} a_k \text{ (i.e., } \mathsf{G}^k_r = 1\mathsf{G}^{k-1}_{2^k-r+1}),$$

and the binary representation of $(2^k + 1 - r) - 1 = 2^k - 1 - (r-1) = (\sum_{i=0}^{k-1} 2^i) - (r-1)$ is

$$(11\cdots1) - (b_1 b_2 \cdots b_k) = 0(1-b_1)\cdots(1-b_{k-1})(1-b_k).$$

By induction hypothesis, we have $a_j \equiv (1-b_j) + (1-b_{j-1}) \equiv b_j + b_{j-1} \pmod 2$ for $j = 2, \ldots, k$. It follows from $a_1 = b_1 + b_0$ that (2.1) is true for $n = k$.

Now the validity of (2.1) implies $\sum_{i=1}^{j} a_i \equiv \sum_{i=1}^{j} (b_i + b_{i+1}) \equiv b_1 + b_{j+1} \equiv b_{j+1} \pmod 2$. □

The relations in Lemma 2.1 give rise to the ranking and unranking algorithms for the binary reflected Gray code.

Algorithm 2.4. GRAYRANK(T, n)
INPUT: *A subset T of $\{1, \ldots, n\}$.*
OUTPUT: *The rank r of T for the binary reflected Gray code G^n.*

```
r ← 1, b ← 0
for i ← 1 to n
   if i ∈ T then b ← 1 − b                    //b holds the value of current b_i
   if b = 1 then r ← r + 2^{n−i}
end-for
Return r
```

To see the correctness of Algorithm 2.4, it suffices note that for $i = 1, 2, \ldots, n$, $i \in T$ if and only if $a_i = 1$ (by the definition of G^n), and $b_i = 1 - b_{i-1}$ if and only if $a_i = 1$ (by (2.1)).

Algorithm 2.5. GRAYUNRANK(n, r)
INPUT: *The cardinality n of the ground set S, and integer $r \in [1, 2^n]$.*
OUTPUT: *The rth subset T for the binary reflected Gray code G^n.*

[1] Note that $b_0 = 0$ always.

$T \leftarrow \emptyset$, $b \leftarrow 0$, $r \leftarrow r - 1$
for $i \leftarrow 1$ **to** n
  $b' \leftarrow \lfloor r/2^{n-i} \rfloor$ //b' holds the value of current b_i
  if $b' \neq b$ **then** $T \leftarrow T \cup \{i\}$ //b holds the value of b_{i-1}
  $b \leftarrow b'$, $r \leftarrow r - b \cdot 2^{n-i}$ //Update b for the use of next iteration
end-for
`Return` T

Exercise 2.4. *Suppose $n = 8$. Compute* $\mathsf{unrank}(170)$ *w.r.t. binary reflected Gray code.*

HINT: $\mathsf{unrank}(170) = \{1, 2, 3, 4, 5, 6, 8\}$

i	r	b	b'	$i \in T$?
1	169	0	1	yes
2	41	1	0	yes
3	41	0	1	yes
4	9	1	0	yes
5	9	0	1	yes
6	1	1	0	yes
7	1	0	0	no
8	1	0	1	yes

Another way to order the subsets of an n-element set is to order them first in increasing size, and then in lexicographic order for each fixed size. For example, when $n = 3$, this ordering for the subsets of $\{1, 2, 3\}$ is $\emptyset, \{1\}, \{2\}, \{3\}, \{1, 2\}, \{1, 3\}, \{2, 3\}, \{1, 2, 3\}$.

Exercise 2.5. *Develop unranking, ranking and successor algorithms for the subsets with respect to the above ordering.*

HINT: Note that T is a k-subset if and only if

$$\sum_{i=0}^{k-1} \binom{n}{i} < \mathsf{rank}(T) \leq \sum_{i=0}^{k} \binom{n}{i}$$

We can reduce the problem to the one for unranking, ranking and finding successor on k-subsets.

2.2 k-subsets

Suppose that $S = \{1, \ldots, n\}$ for some $n \in \mathbb{N}$, and $\mathcal{S}$ consists of all k-subsets of S for a fixed positive integer $k \leq n$. So $|\mathcal{S}| = \binom{n}{k}$.

Lexicographic ordering. We may write each $T \in \mathcal{S}$ in a natural way as a list $\acute{T} = [t_1, t_2, \ldots, t_k]$ with $t_1 < t_2 < \cdots < t_k$. The lexicographic ordering on $\mathcal{S}$ is induced by the lexicographic ordering on the sequences $\acute{T}$, $T \in \mathcal{S}$.

Algorithm 2.6. kSUBSETLEXSUCC$(n, k, \acute{T})$
INPUT: *The cardinality n of the ground set S, integer $k \in [1, n]$, and k-subset $\acute{T} = [t_1, t_2, \ldots, t_k]$ of S.*
OUTPUT: *The successor $\acute{U}$ of $\acute{T}$ in the lexicographic ordering on $\mathcal{S}$.*

$\acute{U} \leftarrow \acute{T}$, $i \leftarrow k$
while $i \geq 1$ `and` $t_i = n - k + i$ **do** //from right to left, find the first t_i with $t_i < n - (k - i)$
$i \leftarrow i - 1$
end-while
if $i = 0$ **then** `Return "undefined"`
else for $j \leftarrow i$ **to** k **do** //set $u_i, u_{i+1}, \ldots, u_k$ to $t_i + 1, t_i + 2, \ldots, t_i + 1 + (k - i)$
$u_j \leftarrow t_i + 1 + j - i$
end-for
`Return` $\acute{U}$

For example, the successor of $[\ldots, \boldsymbol{n-8}, n-2, n-1, n]$ is $[\ldots, \boldsymbol{n-7, n-6, n-5, n-4}]$.

To construct a ranking algorithm, we need to count the number of k-subsets preceding a given set $\acute{T} = [t_1, \ldots, t_k]$ in the lexicographic ordering on $\mathcal{S}$. It is easy to see that for any $i \leq k$ there are exactly $\binom{n-t_i}{k-i}$ subsets $\acute{X} = [x_1, \ldots, x_k] \in \mathcal{S}$ such that $x_j = t_j$ for $j = 1, \ldots, i$. For convenience, we define $t_0 = 0$. Then

$$\mathsf{rank}(T) = \mathsf{rank}(\acute{T}) = 1 + \sum_{i=1}^{k} \sum_{j=t_{i-1}+1}^{t_i - 1} \binom{n-j}{k-i}. \tag{2.3}$$

To design an unranking algorithm, we consider integer $r \in [1, \binom{n}{k}]$, and suppose that $\mathsf{unrank}(r) = T$ with $\acute{T} = [t_1, \ldots, t_k]$. Observe that

$$t_1 = x \Leftrightarrow \sum_{j=1}^{x-1} \binom{n-j}{k-1} < r \leq \sum_{j=1}^{x} \binom{n-j}{k-1}.$$

Having determined t_1, we can compute t_2 in a similar way. The pattern continues.

Algorithm 2.7. kSUBSETUNR(n, k, r)
INPUT: *The cardinality n of the ground set S, integer $k \in [1, n]$, and integer $r \in [1, \binom{n}{k}]$.*
OUTPUT: *The rth subset $\acute{T}$ in the lexicographic ordering on $\mathcal{S}$.*

$x \leftarrow 1$
for $i \leftarrow 1$ **to** k **do**
while $\binom{n-x}{k-i} < r$ **do**
$r \leftarrow r - \binom{n-x}{k-i}$, $x \leftarrow x + 1$
end-while
$t_i \leftarrow x$, $x \leftarrow x + 1$
end-for
`Return` $\acute{T} = [t_1, \ldots, t_k]$

Co-lex ordering. To each $T \in \mathcal{S}$ we associate a list $\grave{T} = [t_1, t_2, \ldots, t_k]$ with $t_1 > t_2 > \cdots > t_k$. The *co-lex ordering* on S is induced by the lexicographic ordering on the sequences $\grave{T}$, $T \in \mathcal{S}$.

Exercise 2.6. *Design a successor algorithm for the co-lex ordering of k-subsets of an n-element set.*

HINT:

$\grave{U} \leftarrow \grave{T}$, $i \leftarrow k$
while $i \geq 2$ `and` $t_{i-1} = t_i + 1$ **do** //from right to left, find the first t_i with $t_{i-1} > t_i + 1$
$i \leftarrow i - 1$
end-while

if $i = 1$ `and` $t_1 = n$ **then** `Return "undefined"`
  else for $j \leftarrow k$ **downto** i **do**
    $u_j \leftarrow k - j + 1$
  end-for
  $u_i = t_i + 1$
  `Return` $\grave{U}$

Similar to the ranking in lexicographic ordering, for the co-lex ordering the rank function has the following formula (i.e., there are exactly $\binom{t_i-1}{k-(i-1)}$ k-subsets $\grave{X} = [t_1, \ldots, t_{i-1}, x_i, \ldots]$ with $x_i < t_i$)

$$\mathsf{rank}(T) = \mathsf{rank}(\grave{T}) = 1 + \sum_{i=1}^{k} \binom{t_i - 1}{k + 1 - i}, \tag{2.4}$$

which does not depend on the value of n, standing in contrast to (2.3). The unranking algorithm is an analog to Algorithm 2.12.

Exercise 2.7. *According to (2.4), design an unranking algorithm for the co-lex ordering of k-subsets of an n-element set.*

HINT:

$x \leftarrow n$
for $i \leftarrow 1$ **to** k **do**
  while $\binom{x}{k-i+1} \geq r$ **do**
    $x \leftarrow x - 1$
  end-while
$r \leftarrow r - \binom{x}{k-i+1}$, $t_i \leftarrow x + 1$
end-for
`Return` $\grave{T} = [t_1, \ldots, t_k]$

The following relation between the co-lex ordering and the lexicographic ordering suggests a more efficient way for computing lexicographic rank.

Theorem 2.8. *Let* rank_l *and* rank_c *denote the ranks in lexicographic ordering and co-lex ordering respectively. Then for any k-subset $T \in \mathcal{S}$, it holds that* $\mathsf{rank}_\mathsf{l}(T) + \mathsf{rank}_\mathsf{c}(T') = \binom{n}{k} + 1$, *where* $T' = \{n + 1 - i : i \in T\}$.

Exercise 2.8. *Prove Theorem 2.8.*

HINT:

$$\begin{aligned}
& \mathsf{rank}_\mathsf{l}(T) + \mathsf{rank}_\mathsf{c}(T') \\
= \;& 2 + \sum_{i=1}^{k} \sum_{j=t_{i-1}+1}^{t_i-1} \binom{n-j}{k-i} + \sum_{i=1}^{k} \binom{n - t_i}{k + 1 - i} \\
= \;& 2 + \sum_{i=1}^{k} \left[\binom{n - t_{i-1}}{k - i + 1} - \binom{n - t_i + 1}{k - i + 1} + \binom{n - t_i}{k - i + 1} \right] \\
= \;& 2 + \sum_{i=1}^{k} \left[\binom{n - t_{i-1}}{k - i + 1} - \binom{n - t_i}{k - i} \right] \\
= \;& 2 + \binom{n}{k} - 1 = \binom{n}{k} + 1
\end{aligned}$$

Minimal change ordering. Let $\mathcal{S}_k^n$ denote the set of k-subsets of $S = \{1, \ldots, n\}$. Then $\mathsf{dist}(X, Y) \geq 2$ for any distinct $X, Y \in \mathcal{S}_k^n$. A minimal change ordering on S_k^n is the one in which any two consecutive subsets have distance 2. We study a special minimal change ordering called the *revolving door ordering* on S_k^n. It is written as a list

$$\mathsf{A}^{n,k} = \left[\mathsf{A}_1^{n,k}, \mathsf{A}_2^{n,k}, \ldots, \mathsf{A}_{\binom{n}{k}}^{n,k}\right]$$

of $\binom{n}{k}$ k-subsets. The lists

$$\mathsf{A}^{n,0} = [\emptyset] \text{ and } \mathsf{A}^{n,n} = [\{1, \ldots, n\}]$$

are given as initial conditions to start the recursion for defining $\mathsf{A}^{n,k}$. Given $\mathsf{A}^{n-1,k-1}$ and $\mathsf{A}^{n-1,k}$,

$$\mathsf{A}^{n,k} = \left[\mathsf{A}_1^{n-1,k}, \ldots, \mathsf{A}_{\binom{n-1}{k}}^{n-1,k}, \mathsf{A}_{\binom{n-1}{k-1}}^{n-1,k-1} \cup \{n\}, \ldots, \mathsf{A}_1^{n-1,k-1} \cup \{n\}\right].$$

The revolving door algorithm is due to Nijenbuis and Wilf [24].

Lemma 2.2. *For $k, n \in \mathbb{N}$ with $k \leq n$. The following hold:*

$$\mathsf{A}_{\binom{n}{k}}^{n,k} = \{1, \ldots, k-1, n\} \tag{2.5}$$

$$\mathsf{A}_1^{n,k} = \{1, \ldots, k\}. \tag{2.6}$$

Proof. By definition $\mathsf{A}^{1,1} = [\{1\}]$ and $\mathsf{A}_{\binom{n}{k}}^{n,k} = \mathsf{A}_1^{n-1,k-1} \cup \{n\}$ for all $n \geq k \geq 2$. It suffices to justify (2.6).

By induction on n, the base where $n = 1$ is trivial. Suppose that (2.6) holds for $n = j - 1 \geq 1$ and all k with $1 \leq k \leq j - 1$. We consider $n = j$ and any positive integer $k \leq j$. If $k = j$, then $\mathsf{A}_1^{j,j} = \{1, \ldots, j\}$ by definition, and (2.6) holds. So we assume $k \leq j - 1$. By definition and induction hypothesis, we obtain

$$\mathsf{A}_1^{j,k} = \mathsf{A}_1^{j-1,k} = \{1, \ldots, k\},$$

establishing the lemma. □

Theorem 2.9. *For any integers k and n such that $1 \leq k \leq n$, $\mathsf{A}^{n,k}$ is a minimal change ordering on $\mathcal{S}_k^n$.*

Proof. By induction, the base where $n = 1$ is trivial. Suppose that $n \geq 2$ and $\mathsf{A}^{n-1,k}$ is a minimal change ordering for all integer k with $1 \leq k \leq n - 1$. The case of $k \in \{1, n\}$ is easy, since each set in the list is a 1-set.

We consider $2 \leq k \leq n - 1$, and two consecutive sets $\mathsf{A}_i^{n,k}$ and $\mathsf{A}_{i+1}^{n,k}$. If $1 \leq i \leq \binom{n-1}{k} - 1$ or $\binom{n-1}{k} + 1 \leq i \leq \binom{n}{k} - 1$, then we are done by induction hypothesis. It remains to consider the case of $i = \binom{n-1}{k}$, for which we have

$$\mathsf{A}_{\binom{n-1}{k}}^{n,k} = \mathsf{A}_{\binom{n-1}{k}}^{n-1,k} = \{1, \ldots, \boldsymbol{k-1}, n-1\} \text{ and } \mathsf{A}_{\binom{n-1}{k}+1}^{n,k} = \mathsf{A}_{\binom{n-1}{k-1}}^{n-1,k-1} \cup \{n\} = \{1, \ldots, k-2, n-1, \boldsymbol{n}\}$$

by Lemma 2.2. The theorem is proved. □

Exercise 2.9. *For $T \in \mathcal{S}_k^n$ with $\acute{T} = [t_1, \ldots, t_k]$, $t_1 < \cdots < t_k$, its rank in the revolving door ordering is given by the formula*

$$\mathsf{rank}(T) = 1 + \sum_{i=1}^{k} (-1)^{k-i} \left(\binom{t_i}{i} - 1\right).$$

HINT: By induction on n, the case of $t_k < n$ is straightforward from the induction hypothesis. In case of $t_k = n$, we have $\mathsf{rank}(T) = \binom{n}{k} - \sum_{i=1}^{k-1} (-1)^{k-1-i} (\binom{t_i}{i} - 1)$. □

Exercise 2.10. *Determine the 28th subset $\acute{T}$ in the revolving door ordering on $\mathcal{S}_3^7$.*

HINT: $\{7, 5, 3\}$. □

In the following successor algorithm for the revolving door ordering, the successor of the last k-subset is the first one: $\mathsf{successor}(\{1, \ldots, k-1, n\}) = \{1, \ldots, k\}$. This is well-defined. In the following pseudocode, we begin with $t_{k+1} = n + 1$, which is the notational convenience for handling the case of $j = k$.

Algorithm 2.10. kSUBSETREVSUCC$(n, k, \acute{T})$
INPUT: *The cardinality n of the ground set S, integer $k \in [1, n]$, and k-subset $\acute{T} = [t_1, t_2, \ldots, t_k]$ of S.*
OUTPUT: *The successor $\acute{U}$ of $\acute{T}$ in revolving door ordering on $\mathcal{S}_k^n$.*

$\acute{U} \leftarrow \acute{T}$, $t_{k+1} \leftarrow n + 1$, $j \leftarrow 1$
while $j \leq k$ `and` $t_j = j$ **do**
  $j \leftarrow j + 1$
end-while
if $k \not\equiv j \pmod 2$
  then if $j = 1$
    then $u_1 \leftarrow t_1 - 1$ //This is Case B.2 of the slides [2]
    else $u_{j-1} \leftarrow j$ //This is Case B.1 of the slides [3]
      if $j \geq 3$ **then** $u_{j-2} \leftarrow j - 1$
  else if $t_{j+1} \neq t_j + 1$
    then $u_{j-1} \leftarrow t_j$, $u_j \leftarrow t_j + 1$ //This is Case A.2 of the slides [4]
    else $u_{j+1} \leftarrow t_j$, $u_j \leftarrow j$ //This is Case A.1 of the slides [5]
`Return` $\acute{U} = [u_1, \ldots, u_k]$

Exercise 2.11. *Prove the correctness of Algorithm 2.10.*

2.3 Permutations

We now look at the generation of all $n!$ permutations of the set $\{1, \ldots, n\}$. A permutation $\pi : \{1, \ldots, n\} \to \{1, \ldots, n\}$ is often represented by the list $[\pi[1], \ldots, \pi[n]]$.

Lexicographic ordering. To find the successor of a given permutation π in the lexicographic ordering of (list representations of) permutations, we set $\pi[0] = 0$ for notational convenience.

Algorithm 2.11. PMLEXSUCC(n, π)
INPUT: *A permutation π on $\{1, \ldots, n\}$.*
OUTPUT: *The successor σ of π in the lexicographic ordering of permutations.*

$\sigma \leftarrow \pi$, $\pi[0] \leftarrow 0$, $i \leftarrow n - 1$
while $\pi[i + 1] < \pi[i]$ **do**
  $i \leftarrow i - 1$
end-while
if $i = 0$ **then** `Return "undefined"`
$j \leftarrow n$
while $\pi[j] < \pi[i]$ **do**
  $j \leftarrow j - 1$

[2] Case B.2 is the case of $k \not\equiv j \pmod 2$ and $j = 1$, for which we decreases t_1 by 1.
[3] Case B.1 is the case of $k \not\equiv j \pmod 2$ and $j \geq 2$, for which we increase t_{j-1} and t_{j-2} (if any) by 1, respectively.
[4] Case A.2 is the case of $k \equiv j \pmod 2$ and $t_{j+1} \neq t_j + 1$, for which we insert $t_j + 1$ after t_j, and remove $t_{j-1} = j - 1$.
[5] Case A.1 is the case of $k \equiv j \pmod 2$ and $t_{j+1} = t_j + 1$, for which we insert j before t_j, and remove $t_{\min\{j+1,n\}}$.

end-while
$\sigma[j] \leftarrow \pi[i]$, $\sigma[i] \leftarrow \pi[j]$
for $h \leftarrow i+1$ **to** n **do**
  $\sigma[h] \leftarrow \pi[n+i+1-h]$
end-for
`Return` σ

Exercise 2.12. *Prove the correctness of Algorithm 2.11.*

HINT: From right to left, find the first i such that $\pi[i] < \pi[i+1]$ and $\pi[i+1] > \cdots > \pi[n]$. Among $\pi[i+1], \ldots, \pi[n]$, find the smallest (rightmost) $\pi[j]$ such that $\pi[j] > \pi[i]$. Swap $\pi[i]$ and $\pi[j]$. Reverse the order of the elements with indices $i+1, i+2, \ldots, n$. □

In the lexicographic ordering of permutations of $\{1, 2, \ldots, n\}$, we first have $(n-1)!$ permutations that begin with 1, followed by $(n-1)!$ permutations that begin with 2, etc. Hence, given any permutation π, we have

$$(\pi[1]-1)(n-1)! + 1 \le \mathsf{rank}(\pi, n) \le \pi[1](n-1)!.$$

More precisely, let permutation π of $\{1, \ldots, n-1$ be defined by $\pi'[i] = \pi[i+1] - 1$ if $\pi[i+1] > \pi[1]$ and $\pi[i+1]$ otherwise for $i = 1, \ldots, n-1$, then

$$\mathsf{rank}(\pi, n) = (\pi[1]-1)(n-1)! + \mathsf{rank}(\pi', n-1). \tag{2.7}$$

This recurrence relation is reduced to the initial condition $\mathsf{rank}([1], 1) = 1$.

Exercise 2.13. *Convert the recursive formula (2.7) into a non-recursive algorithm.*

HINT:

$r \leftarrow 1$
for $i \leftarrow 1$ **to** n **do**
  $r \leftarrow r + (\pi[i]-1)(n-i)!$
  for $j \leftarrow i+1$ **to** n **do**
    if $\pi[j] > \pi[i]$ **then** $\pi[j] \leftarrow \pi[j] - 1$
  end-for
end-for
`Return` r

On the other hand, every integer $r - 1 \in [0, n! - 1]$ has a unique factorial representation of form

$$r - 1 = \sum_{i=1}^{n-1} (d_i \cdot i!), \text{ where } 0 \le d_i \le i \text{ for } i = 1, \ldots, n-1.$$

Exercise 2.14. *Prove the existence and uniqueness of the factorial representation of* $r - 1$.

HINT: $(k+1)! > \sum_{i=1}^{k} i \times i!$. (Uniqueness!) □

Suppose that $\pi = \mathsf{unrank}(r)$ in the lexicographic ordering. Note that

$$(\pi[1]-1)(n-1)! \le r - 1 = (\pi[1]-1)(n-1)! + \mathsf{rank}(\pi', n-1) - 1 < (\pi[1]-2)(n-1)!.$$

It follows that $\pi[1] = d_{n-1} + 1$. We may recurse by unranking $r' = r - d_{n-1}(n-1)!$.

Algorithm 2.12. PmLexUnr(n, r)
INPUT: *Positive integers n and r with $r \leq n!$.*
OUTPUT: *The rth permutation π in the lexicographic ordering of permutations of $\{1, \ldots, n\}$.*

$\pi[n] \leftarrow 1$
for $j \leftarrow 1$ **to** $n-1$ **do**
  $d \leftarrow \frac{(r-1) \mod (j+1)!}{j!}$, $\pi[n-j] \leftarrow d+1$ //Compute temporary $\pi[n-1], \pi[n-2], \ldots, \pi[1]$ sequentially
  $r \leftarrow r - d \cdot j!$
  for $i \leftarrow n-j+1$ **to** n **do**
    if $\pi[i] > d$ **then** $\pi[i] \leftarrow \pi[i]+1$ //The condition is equivalent to $\pi[i] \geq d+1 = \pi[n-j]$
  end-for
end-for
`Return` π

Exercise 2.15. *Compute* $\mathsf{unrank}(4, 9)$ *and* $\mathsf{unrank}(4, 22)$ *for the lexicographic ordering of permutations of* $\{1, 2, 3, 4\}$.

HINT:

- $9 - 1 = 1 \times 3! + 1 \times 2!$
  $\pi[1] = 2,\ \pi[2] = 3,\ \pi[3] = 1,\ \pi[4] = 4$
- $22 - 1 = 3 \times 3! + 1 \times 2! + 1 \times 1!$
  $\pi[1] = 4,\ \pi[2] = 2,\ \pi[3] = 3,\ \pi[4] = 1$

Minimal change ordering. A minimal change ordering of permutations is produced by Trotter-Johnson algorithm [34, 18]. Suppose we have a list of the $(n-1)!$ permutations of $\{1, \ldots, n-1\}$ in minimal change order (more specifically in Trotter-Johnson ordering), say $\mathsf{T}^{n-1} = [\pi_1, \pi_2, \ldots, \pi_{(n-1)!}]$. Form a new list by repeating each permutation in T^{n-1} n times. If i is odd, then we first insert element n in the first π_i after the element in position $n-1$, then in the second π_i after the element in position $n-2$, etc., and finally in the last π_i preceding the element in position 1. If i is even, then we proceed in the opposite order, inserting element n into the n copies of π_i from the beginning to the end of π_i. This gives rise to *Trotter-Johnson ordering* T^n.

Given permutation π=$[\pi[1], \ldots, \pi[n]]$ with $\pi[k] = n$. Define a permutation π' of $\{1, \ldots, n-1\}$ by $\pi' = [\pi[1], \ldots, \pi[k-1], \pi[k+1], \ldots, \pi[n]]$. Note from the construction that there are at least $n(\mathsf{rank}(\pi', n-1)-1)$ before π and the rank of π is at most $n(\mathsf{rank}(\pi', n-1)-1)+n$, i.e.,

$$n(\mathsf{rank}(\pi', n-1)-1)+1 \leq \mathsf{rank}(\pi, n) \leq n\mathsf{rank}(\pi', n-1).$$

The exact value of π's rank is determined from the position of π' into which the element n was inserted, giving

$$\mathsf{rank}(\pi, n) = n(\mathsf{rank}(\pi', n-1)-1)+\ell, \tag{2.8}$$

where $\ell = n-k+1$ if $\mathsf{rank}(\pi', n-1)$ is odd and $\ell = k$ otherwise.

Exercise 2.16. *Convert the recursive formula (2.8) into a non-recursive ranking algorithm for Trotter-Johnson ordering.*

HINT:

$r \leftarrow 1$
for $i \leftarrow 2$ **to** n **do**

$j \leftarrow 1$, $k \leftarrow 1$
while $\pi[j] \neq i$ **do**
if $\pi[j] < i$ **then** $k \leftarrow k + 1$ //k denote the position of i in the projection of π to $\{1, \ldots, i\}$
$j \leftarrow j + 1$
end-while
if $r \equiv 0 \pmod 2$ **then** $r \leftarrow i(r-1) + k$ **else** $r \leftarrow ir - k + 1$
end-for
`Return` r

Unranking can also be investigated in a recursive way. Rewrite (2.8) as

$$\mathsf{rank}(\pi', n-1) = \frac{\mathsf{rank}(\pi, n) - \ell}{n} + 1 = \left\lfloor \frac{\mathsf{rank}(\pi, n) - 1}{n} \right\rfloor + 1,$$

where the last equation is guaranteed by $1 \leq \ell \leq n$. Given positive integers n and r with $r \leq n!$. Let

$$r' = \left\lfloor \frac{r-1}{n} \right\rfloor + 1.$$

Suppose that the permutation π' of $\{1, \ldots, n-1\}$ has rank r'. We compute $\ell = r - n(r'-1)$, and insert n into π' into position ℓ if r' is even, and into position $n - \ell + 1$ otherwise.

The successor algorithm for Trotter-Johnson ordering is somewhat more complicated. It needs subroutine PmParity that computes the parity of a given permutation. The algorithm changes π to its successor by locating indices $1 \leq s \leq u < u+1 \leq t \leq n$ and swapping $\pi[u]$ and $\pi[u+1]$,

$$\sigma = [\pi[s], \ldots, \pi[t]] \text{ is a permutation on } \{1, \ldots, t-s+1\},$$

$\max\{\pi[u], \pi[u+2]\} = t-s+1$, and $\mathsf{successor}(\sigma) = [\pi[s], \ldots, \pi[u+1], \pi[u], \ldots, \pi[t]]$. The reader is referred to [20] for details.

Algorithm 2.13. TrotterJohnsonSucc(n, π)
Input: *A permutation π on $\{1, \ldots, n\}$.*
Output: *The successor of π in Trotter-Johnson ordering of permutations.*

$s \leftarrow 1$, $t \leftarrow n$ //s (resp. t) holds the starting (resp. ending) position (in the original π) of current σ
done $\leftarrow$FALSE //*done* is used to indicate whether a swap of two adjacent elements is done
while $t > s$ `and` $\neg$*done* **do**
$v \leftarrow s$
while $\pi[v] \neq t - s + 1$ **do** //Determine the position v (in π) of the largest element $t-s+1$ in σ
$v \leftarrow v + 1$
end-while
parity $\leftarrow (1+\text{PmParity}(s-t, [\pi[s], \ldots, \pi[v-1], \pi[v+1], \ldots, \pi[t]]) \mod 2$
if *parity* $= 0$ `and` $v = t$
then $t \leftarrow t - 1$ //$\mathsf{successor}(\sigma) = [\mathsf{successor}(\sigma'), t-s+1]$
if *parity* $= 0$ `and` $k < n$
then *temp* $\leftarrow \pi[v]$, $\pi[v] \leftarrow \pi[v+1]$, $\pi[v+1] \leftarrow$ *temp* //Swap $\pi[v]$ and $\pi[v+1]$
done $\leftarrow$TRUE //We can stop (don't need recurse) as soon as a swap is done
if *parity* $= 1$ `and` $v = s$
then $s \leftarrow s + 1$ //$\mathsf{successor}(\sigma) = [t-s+1, \mathsf{successor}(\sigma')]$
if *parity* $= 1$ `and` $v > s$
then *temp* $\leftarrow \pi[v]$, $\pi[v] \leftarrow \pi[v-1]$, $\pi[v-1] \leftarrow$ *temp* //Swap $\pi[v]$ and $\pi[v-1]$
done $\leftarrow$TRUE //We can stop as soon as a swap is done
end-while
if *done* $=$TRUE **then** `Return` π **else** `Return "undefined"`

Exercise 2.17. *Determine the rank of the permutation* $[2, 4, 6, 7, 5, 3, 1]$ *in lexicographic and Trotter-Johnson order.*

HINT:

$$\begin{aligned}
&\mathsf{rank}(\{1\}, 1) = 1\\
&\mathsf{rank}(\{2, 1\}, 2) = 2\\
&\mathsf{rank}(\{2, 3, 1\}, 3) = 3 + 2 = 5\\
&\mathsf{rank}(\{2, 4, 3, 1\}, 4) = 16 + 4 = 20\\
&\mathsf{rank}(\{2, 4, 5, 3, 1\}, 5) = 95 + 3 = 98\\
&\mathsf{rank}(\{2, 4, 6, 5, 3, 1\}, 6) = 582 + 3 = 585\\
&\mathsf{rank}(\{2, 4, 6, 7, 5, 3, 1\}, 7) = 584 \times 7 + 4 = 4092
\end{aligned}$$

3 Maximum flows

A *flow network* is a directed graph $G = (V, A)$ with distinguished vertices s (the *source*) and t (the *sink*), in which each arc $(i, j) \in A$ has a nonnegative integer capacity u_{ij}. Throughout this section, let $n = |V|$ and $m = |A|$. An s-t flow is defined as a nonnegative real-valued function on the arcs in A satisfying the "flow conservation law" at each vertex $\neq s, t$. Formally, an s-t *flow* is a function $f : A \to \mathbb{R}_+$ satisfying

(i) *capacity constraints*: $f_{ij} \leq u_{ij}$ for all $(i, j) \in A$;

(ii) *conservation constraints*: $\sum_{k:(i,k)\in A} f_{ik} = \sum_{k:(k,i)\in A} f_{ki}$ for all $i \in V \setminus \{s, t\}$

We first consider the problem of finding a maximum-value flow subject to capacity constrains, where the *value* of a flow f is

$$|f| \equiv \sum_{k:(s,k)\in A} f_{sk} - \sum_{k:(k,s)\in A} f_{ks}.$$

Problem 3.1. MAXIMUM s-t FLOW

INPUT: Flow network $G = (V, A)$ with source s, sink t, and capacity $u \in \mathbb{Z}_+^A$
GOAL: Find an s-t flow of maximum value.

Question. Is this a maximum s-t flow? (The numbers inside the brackets are capacities.)

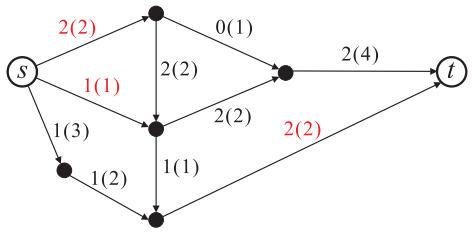


Figure 1: An s-t flow.

3.1 Minimum cuts

For any $S \subseteq V$, let $\delta^+(S) = \{(i,j) \in A : i \in S, j \notin S\}$ denote the set of outgoing arcs from S. An *s-t cut* is a set $S \subseteq V$ such that $s \in S$ and $t \notin S$. The *capacity* of an *s-t* cut S is

$$u(\delta^+(S)) \equiv \sum_{(i,j)\in\delta^+(S)} u_{ij}.$$

A *minimum s-t cut* is the one of the minimum capacity.

Exercise 3.1. *Prove that for any s-t cut S and any s-t flow f, it holds that $|f| \leq u(\delta^+(S))$.*

The following theorem was proved by Ford and Fulkerson [9] for the undirected case and by Dantzig and Fulkerson [5] for the directed case. (According to Robacker [27], the max-flow min-cut theorem was conjectured first by D.R. Fulkerson.)

Theorem 3.2 (max-flow min-cut theorem)**.** *The value of a maximum s-t flow equals the capacity of a minimum s-t cut.*

Exercise 3.2. *Find a maximum s-t flow and a minimum s-t cut for the network given in Figure 2.*

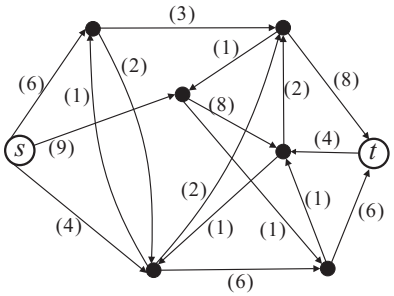


Figure 2: Maximum flow and minimum cut exercise.

For notational simplicity, we often use an alternative definition of flow. In this alternate definition, we assume: If $(i,j) \in A$, then $(j,i) \in A$. Given a flow $\tilde{f}_{ij}$ on arc (i,j), the flow on the reverse arc (j,i) is $\tilde{f}_{ji} = -\tilde{f}_{ij}$. If (j,i) is not in the original instance, then we set $u_{ji} = 0$ (note that since $-\tilde{f}_{ij} = \tilde{f}_{ji} \leq u_{ji} = 0$, this enforces that the flow on the "original" arc is nonnegative.) To summarize, the *alternate definition of network flow* is as follows: An *s-t flow* is a function $\tilde{f} : \tilde{A} \to \mathbb{R}$ satisfying

(i) *capacity constraints*: $\tilde{f}_{ij} \leq u_{ij}$ for all $(i,j) \in \tilde{A}$;

(ii) *anti-symmetry*: $\tilde{f}_{ji} = -\tilde{f}_{ij}$ for all $(i,j) \in \tilde{A}$;

(iii) *conservation constraints*: $\sum_{k:(i,k)\in\tilde{A}} \tilde{f}_{ik} = 0$ for all $i \in V \setminus \{s,t\}$

The value of $\tilde{f}$ is

$$|\tilde{f}| \equiv \sum_{k:(s,k)\in\tilde{A}} \tilde{f}_{sk}.$$

The following concepts are useful in proving Theorem 3.2.

Given a flow $\tilde{f}$, the *residual graph* $G_{\tilde{f}}$ is the graph $(V, A_{\tilde{f}}, u^{\tilde{f}})$, where $A_{\tilde{f}} = \{(i,j) \in \tilde{A} : \tilde{f}_{ij} < u_{ij}\}$ and $u^{\tilde{f}}_{ij} = u_{ij} - \tilde{f}_{ij}$. We call an arc $(i,j) \in A_{\tilde{f}}$ a *residual arc* and call $u^{\tilde{f}}_{ij}$ the *residual capacity* of arc

(i,j). A directed path from s to t in the residual graph $G_{\tilde{f}}$ is called an *augmenting path*. The set $A_{\tilde{f}}$ is therefore the subset of arcs with positive residual capacity. Figure 3 shows the flow on a network and the associated residual graph. By our convention that $\tilde{f}_{ji} = -\tilde{f}_{ij}$, if a flow uses an arc (i,j) less than its full capacity, the residual graph has a residual arc (i,j) of residual capacity $u_{ij} - \tilde{f}_{ij}$ and a residual arc (j,i) of residual capacity $u_{ji} - \tilde{f}_{ji} = 0 - \tilde{f}_{ji} = \tilde{f}_{ij}$.

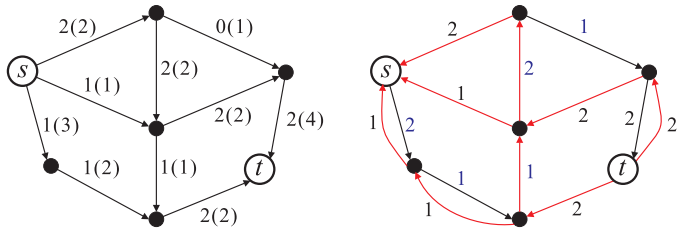


Figure 3: An s-t flow and its associated residual graph.

Given a flow f, residual graph G_f and an augmenting path P, let $\delta = \min_{(i,j)\in P} u^f_{ij}$, i.e., the smallest residual capacity of arcs in the augmenting path P. In the above example (Figure 3), $\delta = 1$. Define a new flow f' such that:

$$f'_{ij} = \begin{cases} f_{ij} + \delta & \text{if } (i,j) \in P, \\ f_{ij} - \delta & \text{if } (j,i) \in P, \\ f_{ij} & \text{otherwise.} \end{cases}$$

Property 3.3. *f' is an s-t flow with $|f'| = |f| + \delta$.* □

Theorem 3.4. *The following are equivalent:*

(i) f is a maximum s-t flow;

(ii) There is no augmenting path in G_f;

(iii) $|f| = u(\delta^+(S))$ for some s-t cut S.

Proof. "(i) ⇒ (ii)" is trivial. To prove "(ii) ⇒ (iii)", let S be the set of all nodes reachable from the source s in G_f . Note that $t \notin S$ and $f_{ij} = u_{ij}$ for any $(i,j) \in \delta^+(S)$. Therefore

$$|f| = \sum_{k:(s,k)\in A} f_{sk} + \sum_{i\in S\setminus\{s\}} \sum_{k:(i,k)\in A} f_{ik} = \sum_{i\in S} \sum_{k:(i,k)\in A} f_{ik} = \sum_{(i,k)\in\delta^+(S)} f_{ik} = u(\delta^+(S)).$$

"(iii) ⇒ (i)" follows from Exercise 3.1. □

We note that the number of s-t cuts is exponentially large ($= 2^{n-2}$). So, it is not a good idea to find the maximum flow value by calculating all possible cut capacities. The theorem above motivates the following algorithm for finding the maximum flow [22].

Algorithm 3.5 (Augmenting path)**.**
INPUT: *Flow network $G = (V, A)$ with source s, sink t, and capacity $u \in \mathbb{Z}^A_+$.*
OUTPUT: *The rth permutation π in the lexicographic ordering of permutations of $\{1, \ldots, n\}$.*

$f \leftarrow \mathbf{0}$
while ∃ `an augmenting path` P `in` G_f **do**
  `Push flow along` P
  `Update` f
end-while

`Output` f

The correctness of the algorithm immediately follows from Theorem 3.4. Under the assumption that all capacities are integer, we push an integral amount of flow at each step of the algorithm. This yields the following result:

Theorem 3.6 (Integrality property)**.** *If all capacities are integers, there is a maximum flow f such that f_{ij} are integers for all $(i,j) \in A$.* □

Corollary 3.7. *If all capacities are rational, Algorithm 3.5 terminates with a maximum flow.*

Surprisingly, if we allow irrational numbers as capacities, the algorithm can fail to terminate. In fact, Ford and Fulkerson [23] showed it can converge to a flow value different from the maximum one (see also [29]).

3.2 Algorithms

Next we focus on polynomial time algorithm for finding maximum flow. Although Algorithm 3.5 correctly finds a maximum flow, it is not a polynomial time algorithm, because it is possible that there will be too little progress made for each augmentation. How about one picking a path to make substantial progress in each augmentation?

3.2.1 Augmenting path algorithms

One natural choice is to employ a greedy strategy: picking path P_0 with largest possible capacity; i.e. $\max_P \min_{(i,j)\in P}\{u_{ij}^f\}$. Then Algorithm 3.5 becomes:

Algorithm 3.8 (Maximum capacity augmenting path)**.**

$f \leftarrow 0$
while $\exists$ `augmenting path in` G_f **do**
  `Pick augmenting path` $P_0 = \arg\max\{\min_{(i,j)\in P}\{u_{ij}^f\} : \text{augmenting path } P\}$
  `Push flow along` P_0
  `Update` f
end-while

To analyze the algorithm, we need some lemmas. The next lemma states that any flow can be decomposed into no more than m weighted s-t paths and cycles so that the flow on an arc is the sum of weights on paths and cycles using the arc, and the value of the s-t flow is the sum of weights along all s-t paths.

Lemma 3.1. *Given an s-t flow f, there are a set $\mathcal{P}$ of s-t paths, a set $\mathcal{C}$ of cycles, and a weight function $w : \mathcal{P} \cup \mathcal{C} \to \mathbb{R}_+$ such that the following hold:*

(i) $f_{ij} = \sum_{Q\in\mathcal{P}\cup\mathcal{C}:(i,j)\in Q} w(Q)$ *for all* $(i,j) \in A$ *with* $f_{ij} > 0$.
(ii) $|f| = \sum_{P\in\mathcal{P}} w(P)$.
(iii) $|\mathcal{P}| + |\mathcal{C}| \le m$.

Proof. By induction on the number of arcs with positive flow. The base case where $f = \mathbf{0}$ trivially holds with $\mathcal{P} \cup \mathcal{C} = \emptyset$. Inductively, we consider an arc $(i,j) \in A$ with $f_{ij} > 0$. If $j \neq t$, by flow conservation at node j, there is a node k such that $f_{jk} > 0$. Similarly, if $i \neq s$, there is a node h such that $f_{hi} > 0$. Proceeding in this manner, we either find a cycle or an s-t path. Denote it by Q. Let

$$w(Q) = \min_{(i,j)\in Q} f_{ij}$$

Update the flow:

$$f'_{ij} = \begin{cases} f_{ij} - w(P) & \text{if } (i,j) \in Q, \\ f_{ij} + w(P) & \text{if } (j,i) \in Q, \\ f_{ij} & \text{otherwise.} \end{cases}$$

Now, the number of paths/cycles with positive weight increases by 1, while the number of arcs with positive flow decreases by at least 1 and the proof follows. □

Lemma 3.2. *Let f be an s-t flow, f^* be a maximum s-t flow in G. Then the maximum flow in the residual graph G_f has value $|f^*| - |f|$.*

Proof. Given any flow g in G_f, let $f'_{ij} = f_{ij} + g_{ij}$ for all $(i,j) \in A$. Then f' is a flow on G, and $|f'| = |f| + |g| \leq |f^*|$ giving $|g| \leq |f^*| - |f|$. Thus the value of any flow in G_f is bounded above by $|f^*| - |f|$.

Also define $\hat{f}_{ij} = f^*_{ij} - f_{ij}$ for all $(i,j) \in A$. Then $\hat{f}$ is a flow on G_f, since $\hat{f}_{ij} = f^*_{ij} - f_{ij} \leq u_{ij} - f_{ij} = u^f_{ij}$, and $\hat{f}$ is a maximum flow since $|\hat{f}| = |f^*| - |f|$. □

It can be deduced from Lemma 3.1 that there is a maximum flow of G_f which can be decomposed into at most m weighted s-t paths (i.e., augmenting paths). Therefore, by Lemma 3.2, some augmenting path, and hence P_0, will have capacity at least $\frac{|f^*|-|f|}{m}$. Let us consider $2m$ iterations of the while-loop in Algorithm 3.8.

- Either (i): all $2m$ iterations augment flow value by an amount $\geq \frac{|f^*|-|f|}{2m}$;
- Or (ii): at least one iteration augments flow value by an amount $< \frac{|f^*|-|f|}{2m}$.

If (i) happens, we are done, since we have a flow of value at least $|f^*|$. If (ii) happens, then the capacity C of the maximum capacity augmenting path has dropped by a factor of 2. Clearly

$$1 \leq C \leq U \equiv \max_{(i,j)\in A} u_{ij}.$$

Therefore, there can be at most $O(\log U)$ decreases of the capacity C of the maximum capacity augmenting path by a factor of 2. Since every $2m$ iterations either the algorithm terminates with a maximum flow or the capacity drops by a factor of 2, there are at most $O(m \log U)$ iterations of the while-loop overall. This gives a polynomial time algorithm.

To calculate the exact running time, we would have to determine the time needed to find the maximum capacity augmenting path P_0.

Exercise 3.3. *Design an algorithm for finding P_0 as efficiently as you can.*

Alternatively, we will consider a variation of Algorithm 3.8. The idea of the this algorithm is to look for paths in which each edge is "big". If such a path exists, then we can increase the flow by a significant amount. If there is no such path, then we will show that we must be closer to the maximum flow value. We need the following definition. Let δ be a nonnegative real. An augment path P is called a *δ-capacity augmenting path* if $u^f_{ij} \geq \delta$ for all $(i,j) \in P$.

Algorithm 3.9 (Capacity scaling)**.**

$f \leftarrow 0$, $\delta \rightarrow 2^{\lfloor \log_2 U \rfloor}$
while $\exists$ `augmenting path in` G_f **do**
  if $\exists$ δ-`capacity augmenting path` P
    then `augment flow along` P, `update` f
    else $\delta \leftarrow \delta/2$
end-while

Theorem 3.10. *Algorithm 3.9 runs in $O(m^2 \log U)$ time.*

Proof. Suppose there does not exist a δ-capacity augmenting path in G_f . Let $\delta' \leftarrow \delta/2$. We know some maximum flow in G_f can be decomposed into at most m paths. The non-existence of δ-capacity augmenting paths implies that each such path has capacity $< \delta$. Thus the maximum flow in G_f has value $< m\delta = 2m\delta'$.

Thus at the beginning of the while-loop in the algorithm we can find at most a number $2m$ of δ-capacity augmenting paths until either we have found a maximum flow or the value of δ is halved. We know δ can be halved at most $O(\log U)$ times, which implies that there are at most $O(m \log U)$ augmentations overall. Each augmentation requires finding a path in the graph G_f in which only edges with capacity at least δ are retained. Thus finding a δ-augmenting path takes at most $O(m)$ time. Therefore capacity scaling gives an $O(m^2 \log U)$ time algorithm. □

Algorithm 3.9 assumes all input data are integer (or equivalently rational). Dinits [7] and Edmonds and Karp [8] restricted their attention to shortest (in terms of number of arcs) augmenting paths, which also improved the running time of the basic augmenting path algorithm dramatically. Moreover, their algorithms are strongly polynomial time.

Theorem 3.11. *If each augmentation of augmenting path algorithm is on a shortest augmenting path, then there are at most nm augmentations. A maximum s-t flow can be found in $O(nm^2)$ time.* □

The proof can be found in [29] and [3]. Note that the theorem makes no hypothesis about either the existence of a maximum flow or the rationality of the components of capacity u. We next turn to an algorithm for finding a maximum flow that takes $O(n^3)$ time.

3.2.2 Push-relabel algorithms

So far we have considered augmenting path algorithms. These algorithms are primal feasible, because capacity constraints and flow conservation constraints are obeyed. We maintain a feasible flow and work towards finding a maximum flow. Goldberg and Tarjan [16] proposed a different, in a sense dual, method, so called Push-Relabel. It is primal infeasible, because it does not obey flow conservation constraints. Here we will maintain a "flow" that has value at least that of the maximum, and work towards finding a feasible flow. The algorithm will maintain a "preflow". A *preflow* is a function $f : A \to \mathbb{R}$ that obeys capacity constraints, antisymmetry constraints (i.e. $f_{ij} = -f_{ji}$) and

$$\sum_{j:(j,i)\in A} f_{ji} \geq 0 \text{ for all } i \in V \setminus \{s, t\}.$$

The *excess* at vertex i is

$$e_i \equiv \sum_{j:(j,i)\in A} f_{ji}.$$

If every vertex (aside from the source and sink) has zero excess, then the preflow is a flow. Given a preflow, we try to reach a feasible flow by pushing excess e_i to sink t and the remainder to source s along shortest paths. Maintaining shortest path lengths is expensive, so instead we maintain a distance labelling d_i which gives us estimates on the shortest path to the sink.

Definition 3.12. *A* distance labelling *is a set of d_i for all $i \in V$ such that (i) d_i is a non-negative integer for each $i \in V$; (ii) $d_t = 0$; (iii) $d_s = n$; and (iv) $d_i \leq d_j + 1$ for all $(i, j) \in A_f$.*

For a path P, we use $|P|$ to denote the length (i.e., the number of arcs) of P.

Property 3.13. *For each $i \in V$, label d_i is a lower bound on the distance from i to t in G_f.*

Proof. To see this, consider the shortest path P from i to t. Any arc (k, j) on this path has the relation $d_k \leq d_j + 1$. Thus, $d_i \leq |P|$, and d_i is the lower bound on distance of i to t. □

The basic algorithm. If we want to push flow along shortest paths, then an arc (i,j) is in the shortest path if $d_i = d_j + 1$. So we will only modify flow on arcs where this condition holds. What if we have excess at a vertex i, and the condition does not hold for any arc $(i,j) \in A_f$? Then our distance estimates for i must be incorrect, since $d_i \leq d_j$ for all $(i,j) \in A_f$. So we will update the label of i to maintain a distance labelling by setting $d_i \leftarrow \min\{d_j : (i,j) \in A_f\} + 1$. We call a vertex $i \in V \setminus \{s,t\}$ *active* if $e_i > 0$.

Algorithm 3.14 (Push-Relabel, Coldberg, Targin 1988)**.**

$f \leftarrow \mathbf{0}$, $e \leftarrow \mathbf{0}$
$f_{sj} \leftarrow u_{sj}$, $f_{js} \leftarrow -f_{sj}$, $e_j \leftarrow u_{sj}$ `for all` $(s,j) \in A_f$
$d_s \leftarrow n$, $d_{V\setminus\{s\}} \leftarrow \mathbf{0}$
while $\exists$ `active vertex` i **do**
  if $\exists$ j `such that` $u_{ij}^f > 0$ `and` $d_i = d_j + 1$
    then `Push` (i,j): $\delta \leftarrow \min\{e_i, u_{ij}^f\}$
      $f_{ij} \leftarrow f_{ij} + \delta$, $f_{ji} \leftarrow f_{ji} - \delta$, $e_i \leftarrow e_i - \delta$, $e_j \leftarrow e_j + \delta$
    else `Relabel` i: $d_i \leftarrow \min\{d_j : (i,j) \in A_f\} + 1$
end-while

Lemma 3.3. *Algorithm 3.14 maintains a valid distance labeling d.*

Proof. We prove by induction on the algorithm that d always satisfies the four conditions in Definition 3.12. Note that the algorithm has never relabled s or t. The validity of conditions (i) – (iii) is obvious.

Base case: This is trivial for $i \neq s$. For s, there is no condition on d_s, because there is no edge out of s in A_f (cf. the second line of the algorithm).

Inductive step: Note that relabelling does not invalidate the distance labels. What about pushes? If we push on arc (i,j), then two things might happen to cause the distance labelling to be invalid. First, (j,i) might enter A_f. In that case, $d_j = d_i - 1 \leq d_i + 1$, so the distance labelling is valid. Second, (i,j) might be deleted from A_f. This is fine since there is one less condition to worry about. □

Lemma 3.4. *If Algorithm 3.14 terminates (in a finite number of steps), then f is a maximum flow.*

Proof. If the algorithm terminates, then f is a flow, since there will not be any excess at any vertex other than the source and sink. Suppose f is not a maximum flow. Then there exists an augmenting path P in G_f. By Property 3.13, this implies that $d_s \leq n-1$, which contradicts $d_s = n$ in Definition 3.12(iii). □

The following lemma is useful in showing that the distance labels stay finite.

Lemma 3.5. *If f is a preflow and i is active, then s is reachable from i in G_f.*

Proof. Let S be the set of vertices reachable from i in G_f. Suppose on the contrary that $s \notin S$. Clearly, for $j \in S, k \notin S$, $f_{kj} \leq 0$ because (j,k) reaches its capacity. Thus,

$$\sum_{j\in S} e_j = \sum_{j\in S} \sum_{k:(k,j)\in A_f} f_{kj} = \sum_{j\in S, k\notin S, (k,j)\in A_f} f_{kj} \leq 0$$
$$\Rightarrow \quad e_j = 0 \text{ for all } j \in S \Rightarrow e_i = 0 \Rightarrow i \text{ is not active,}$$

which causes a contradiction. □

Lemma 3.6. *At any time point in Algorithm 3.14, $d_i \leq 2n-1$ for all $i \in V$.*

Proof. Note that $d_s = n$ never changes, and d_i increases only when i is active. By Lemma 3.5, vertex i is active implies there exists a path P in G_f from i to s. The path in G_f has length at most $n-1$. So $d_i \le d_s + |P| \le d_s + n - 1 = 2n - 1$. □

Lemma 3.7. *There are at most $2n^2$ executions of relabel.*

Proof. Note that $0 \le d_i \le 2n-1$, d_i is integer, d_i never decreases, and relabel increases it by at least 1. So each vertex needs at most $2n-1$ executions. Since there are n vertices, there are at most $n(2n-1) \le 2n^2$ executions of relabel. □

In Algorithm 3.14, there are two types of pushes: (i) push is *saturating* if $\delta = u_{ij}^f$, and (ii) push is *nonsaturating* if $\delta < u_{ij}^f$, implying $\delta = e_i$.

Lemma 3.8. *There are at most mn saturating pushes.*

Proof. Picking an arc $(i, j) \in A$, we need $d_i = d_j + 1$ to push from i to j. To do it again, we need to push back from j to i and $d_j = d_i + 1$. It follows that d_j has to increases by at least 2. Hence here are at most $n - 1$ saturating pushes from i to j by Lemma 3.6. Since there are m arcs, there are in total at most $m(n - 1)$ saturating pushes. □

Lemma 3.9. *There are at most $4n^2m$ nonsaturating pushes.*

Proof. Let $\Phi \equiv \sum_{\text{active } i} d_i$. Note that $\Phi = 0$ at both the start and the end of Algorithm 3.14. So Φ must decrease by the amount Φ increases. What makes Φ increase? Relabel will increase it by at most $2n^2$ (cf. Lemma 3.6). One saturating push may create a new active node i and increase Φ by $d_i \le 2n$. So by Lemma 3.8, Φ can increase by at most $2n^2 + 2n(mn) \le 4n^2m$. What makes Φ decrease? Any nonsaturating push can only make Φ decrease and the decreasing amount is 1. Therefore, there are no more than $4n^2m$ nonsaturating pushes. □

Theorem 3.15. *Algorithm 3.14 runs in $O(n^2m)$ time.*

Proof. The combination of Lemmas 3.6–3.8 says that the algorithm takes $O(n^2m)$ push/relabel operations. □

Finding a minimum s-t cut. Notice that Algorithm 3.14 is typically able to find a minimum s-t cut before the maximum flow, since the excess flow must be pushed back to the source before the maximum flow is revealed. We claim that to find the minimum s-t cut only, one can simply change the definition of active vertices: Let i be active if $e_i > 0$ and $d_i < n$. When the algorithm terminates, any node with excess will have $d_i \ge n$. By Property 3.13, this implies that such a node has no path to the sink. Let S be the set of all vertices that cannot reach the sink. Then it must be the case that all arcs in $\delta^+(S)$ are at full capacity.

Property 3.16. *S is a minimum s-t cut.*

Proof. Indeed, if the algorithm were to continue to run, it would push all remaining excesses back to the source, and all arcs in $\delta^+(S)$ continue to be at full capacity. □

Exercise 3.4. *Find a maximum s-t flow and a minimum s-t cut for the network in Figure 2 using the push-relabel algorithm.*

Faster implementation In Algorithm 3.14, we did not specify how pushes and relabels should be executed. As shown by Goldberg [15] if we are a little more careful, we can obtain a better bound on the non-saturating pushes.

Algorithm 3.17 (FIFO Push-Relabel)**.**

$f \leftarrow \mathbf{0}$, $e \leftarrow \mathbf{0}$
Saturate arcs out of s: $f_{sj} \leftarrow u_{sj}$, $f_{js} \leftarrow -f_{sj}$, $e_j \leftarrow u_{sj}$ for all $(s,j) \in A_f$
$d_s \leftarrow n$, $d_{V\setminus\{s\}} \leftarrow \mathbf{0}$
Put all active vertices in queue Q
while $Q \neq \emptyset$ **do**
  Let i be the node at the front of Q, Pop i
  while $e_i > 0$ and $\exists$ j such that $u^f_{ij} > 0$ and $d_i = d_j + 1$ **do**
    Push (i,j): $\delta \leftarrow \min\{e_i, u^f_{ij}\}$, $f_{ij} \leftarrow f_{ij} + \delta$, $f_{ji} \leftarrow f_{ji} - \delta$, $e_i \leftarrow e_i - \delta$, $e_j \leftarrow e_j + \delta$
    if j becomes active **then** Add j to the end of Q
  end-while
  if $e_i > 0$ **then** Relabel i: $d_i \leftarrow \min\{d_j : (i,j) \in A_f\} + 1$
    Add i to the end of Q
end-while

We call an iteration of the outer while-loop of Algorithm 3.17 an *examination* of vertex i. We partition the total number of vertex examinations into different *passes*. The first pass consist of vertex examinations for those vertices that are put into Q by the fourth line of the algorithm. The second pass consists of the vertex examinations of all vertices that are in Q after the algorithm has examined the vertices in the first pass. Similarly, the third pass consists of the vertex examinations of all the vertices that are in Q after the algorithm has examined the vertices in the second pass, and so on.

Lemma 3.10. *The number of passes over the queue in Algorithm 3.17 is at most* $4n^2$.

Proof. We use the potential function $\Phi = \max\{d_i : i \text{ active}\}$. Let us divide the passes into two types: passes in which some distance label increases, and passes in which no distance label changes.

- By Lemma 3.6, each $d_i \leq 2n-1$, the total number of passes in which some distance label increases is at most $2n^2$.
- If no distance label changes during the pass, then each active node has excess moved to lower-labeled vertices, and Φ decreases by at least 1 during the pass.

Let $\Delta\Phi_\ell$ be the change in Φ from the beginning of pass ℓ until the end of pass ℓ. When $\Delta\Phi_\ell > 0$, we know that some distance labels have increased by at least $\Delta\Phi_\ell$. Thus by Lemma 3.6,

$$\sum_{\ell:\Delta\Phi_\ell>0} \Delta\Phi_\ell \leq 2n^2.$$

Because Φ is initially zero, stays non-negative, and is zero at the end of the algorithm, the total number of passes in which Φ decreases cannot be more than $\sum_{\ell:\Delta\Phi_\ell>0} \Delta\Phi_\ell \leq 2n^2$, that is, since we know that the total increase in Φ over all passes in which it increases is at most $2n^2$, the total number number of passes in which it decreases is also at most $2n^2$.

Thus the total number of passes can be bounded by $4n^2$: $2n^2$ for the passes in which no distance label changes (and thus Φ decreases), and $2n^2$ for the passes in which some distance label increases. □

Theorem 3.18 (Goldberg 1985)**.** *Algorithm 3.17 runs in* $O(n^3)$ *time.*

Proof. Since there is at most one non-saturating push per vertex per pass, it follows from Lemma 3.10 that there are at most a number $4n^3$ of non-saturating pushes. Recalling Lemmas 3.7 and 3.8, the result follows. □

By using the data structure of *dynamic trees*, Push-Relabel can be implemented even more efficiently [16], as stated in the following theorem.

Theorem 3.19 (Goldberg, Tarjan, 1988)**.** *Push-Relabel can be implemented in $O(nm\log(n^2/m))$ time.*

Exercise 3.5. *Suppose that $u_{sv} = \infty$ for some arc (s, v). Show how a maximum flow can be constructed from the solution of a maximum flow problem on a smaller digraph. Can the method be extended to the case where an arbitrary arc has infinite capacity?*

3.2.3 Blocking flow algorithms

Dinits [7] observed that one can speed up the maximum flow algorithm by not augmenting simply along paths, but along flows in the residual network. An s-t flow f in G is *blocking* if every s-t path in G, the original graph, has some arc saturated.

Every maximum flow is obviously also a blocking flow. Is every blocking flow a maximum flow? No, we have seen a counterexample in Figure 1. However, it is useful in order to compute maximum flows.

Lemma 3.11. *A blocking flow in acyclic graphs can be found in $O(mn)$ time.* □

Exercise 3.6. *Prove Lemma 3.11.*

Dinits' algorithm. In the following algorithm, we effectively saturate all the shortest paths in the residual network.

Algorithm 3.20 (Dinits' Blocking Flow Algorithm)**.**

$f \leftarrow \mathbf{0}$
while $\exists$ s-t `path in` G_f **do**
  `Compute distances` d_i `to sink` t `in` G_f `for all` $i \in V$
  `Find blocking flow` $\bar{f}$ `in` $(V, \bar{A})$ `with` $\bar{A} = \{(i,j) \in A_f : d_i = d_j + 1\}$ `& capacity` $u^f_{ij}, (i,j) \in \bar{A}$
  $f \leftarrow f + \bar{f}$
end-while

An arc in $\bar{A} = \{(i, j) \in A_f : d_i = d_j + 1\}$ is called *admissible*. The set of admissible arcs is acyclic, otherwise we would have an inconsistency of the $d_i = d_j + 1$ equations. The following observation is the key to proving a bound on the running time of Dinits' algorithm.

Lemma 3.12. *The distance d_s from the source to the sink strictly increases in each iteration (while-loop) of Algorithm 3.20.*

Proof. Let f be the flow and d_i be distance labels in one iteration. Let f' be the flow and d'_i be distance labels in the next iteration. To begin, we claim that the d_i is a valid distance labelling for the flow in the next iteration; that is,

$$d_i \leq d_j + 1 \text{ for all } (i, j) \in A_{f'}.$$

To see this, we first consider how an arc (i, j) can be in the residual graph of the flow in the next iteration. It could be because it was in the residual graph of the previous iteration; that is $(i, j) \in A_{f'}$, because $(i, j) \in A_f$, in which case the statement holds. Or it could be that the arc is in the residual graph of the next iteration because we pushed flow along the reverse of the arc; that is, $(i, j) \in A_{f'}$ since $(j, i) \in A_f$. In this case, we know that $d_j = d_i + 1$, which implies that $d_i = d_j - 1 \leq d_j + 1$. Thus the claim holds.

We want to show that $d'_s > d_s$. Consider any s-t path P in $A_{f'}$. By definition $|P| = d'_s$. By the properties of a blocking flow, there exists an arc $(p, q) \in P$ that was not admissible in the previous iteration: $(p, q) \notin \bar{A}_f$. In other words, $d_p \neq d_q + 1$, which along with $d_p \leq d_q + 1$ implies that

$$d_p \leq d_q.$$

Since $d_i \leq d_j + 1$ for all arc $(i, j) \in P$, it follows from $d_p \leq d_q$ that $d_s \leq |P| - 1$, which gives $d'_s > d_s$ as desired. □

Clearly this implies that the algorithm takes at most n iterations. Given the two blocking flow algorithms mentioned in Lemmas 3.11 and 3.12, we get the following results [7, 31, 30].

Theorem 3.21. *(i) (Dinits 1970) A maximum flow via blocking flows can be computed in $O(mn^2)$ time. (ii) (Sleator, Tarjan 1980): A maximum flow via blocking flows can be computed in $O(mn \log n)$ time by using dynamic tree data structure.* □

Blocking flows in unit capacity graphs. In some cases, we can show that blocking flow algorithms give a much better result. We consider the special case of unit capacity graphs. A graph has *unit capacity* if $u_{ij} \in \{0, 1\}$ for all arcs $(i, j) \in A$. In this case, we can give the following result. Let

$$\Lambda \equiv \min\{m^{1/2}, n^{2/3}\}.$$

Lemma 3.13. *For unit capacity graphs, Algorithm 3.20 takes $O(\Lambda)$ iterations.*

Proof. We define a number of distance level $D_k = \{i : d_i = k\}$, $k = 0, 1, \ldots, d_s$, and a number of s-t cuts $S_k = \{i : d_i \geq k\}$. Note that for $k > 0$, $s \in S_k$ and $t \notin S_k$.

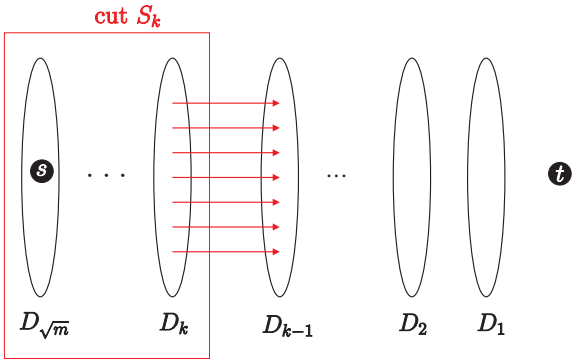


Figure 4: Apply pigeonhole principle by considering all arcs from D_k to D_{k-1}.

Suppose first that $d_s \geq \sqrt{m}$. Then *there exists a distance level D_k such that there are at most $\sqrt{m}$ arcs in $\delta^+(S_k)$*. As can be seen in Figure 4, arcs from D_k to D_{k-1} are distinct for all available distance levels and there are at most m arcs. By the Pigeonhole Principle, this implies that if there are at least $\sqrt{m}$ distance levels, then there exists a D_k such that there are at most $\sqrt{m}$ arcs from D_k to D_{k-1}. Therefore, the residual capacity of the cut S_k is at most $\sqrt{m}$ since the graph is unit capacitated, which implies that the maximum flow value is at most $|f| + \sqrt{m}$ (recalling Lemma 3.2). Hence

- the algorithm takes at most $\sqrt{m}$ iterations until the distance from the source to the sink is $d_s \geq \sqrt{m}$ (by Lemma 3.12); and
- $O(\sqrt{m})$ more augmentations are enough until the algorithm finds a maximum flow.

In total there are $O(\sqrt{m})$ iterations.

Let us now suppose that $d_s \geq 2n^{2/3}$. Then, *there exists k such that* $|D_k| \leq \sqrt[3]{n}$ *and* $|D_{k-1}| \leq \sqrt[3]{n}$ by the pigeonhole principle. Since there are n vertices that are partitioned into the distinct distance levels, we can not have $2n^{2/3}$ distance levels such that every D_k with $|D_k| \leq \sqrt[3]{n}$ is followed by $|D_{k-1}| > \sqrt[3]{n}$. If we now consider all possible arcs from D_k to D_{k-1}, there can be at most $n^{2/3}$ arcs. Thus the residual capacity of the cut S_k is $u^f(\delta^+(S_k)) \leq n^{2/3}$, since all arcs have unit capacity. This proves that only $n^{2/3}$ more iterations are needed to find the maximum flow. Thus as in the previous case, after at most $2n^{2/3}$ iterations, $d_s \geq 2n^{2/3}$, and after $O(n^{2/3})$ iterations, the maximum flow is achieved. There are in total $O(n^{2/3})$ iterations. □

Exercise 3.7. *Show that a blocking flow in an acyclic unit capacity graph can be found in $O(m)$ time.*

Theorem 3.22. *In unit capacity graphs, a maximum flow can be found in $O(\Lambda m)$ time.* □

The Goldberg-Rao algorithm. IDEAS: Suppose that the arcs from D_k to D_{k-1} all have residual capacities no more than Δ for all k. Then (by Lemma and pigeonhole principle) after Λ blocking flow augmentations we will have a cut with residual capacity no more than $\Lambda\Delta$. This seems that it is useful; the amount of remaining flow is reduced significantly with a relatively few blocking flow computations. Then we will reduce Δ and repeat.

PROBLEM 1. How do we get manage to obtain all arcs from D_k to D_{k-1} to have residual capacity at most Δ?

SOLUTION 1. Change our notion of distance. For all $(i,j) \in A_f$, set:

$$l'_{ij} = \begin{cases} 1 & \text{if } u^f_{ij} < \Delta \\ 0 & \text{otherwise.} \end{cases}$$

Revise our definitions to d'_i = distance form i to t using arc length l'_{ij}, $D'_i = \{i : d'_i = k\}$. Arc $(i,j) \in A_f$ is admissible if and only if $d'_i = d'_j + l'_{ij}$.

Note that this does exactly what we want: now any arc from distance level k to distance level $k-1$ must in fact have length 1, and therefore it must have residual capacity no more than Δ. This new idea of lengths causes its own set of problems, however. In particular we have:

PROBLEM 2. The graph of admissible arcs might have cycles (since some l'_{ij} may equal 0). This is an issue since the efficient algorithms we know for blocking flows only run on acyclic graphs.

SOLUTION 2 Suppose the graph of admissible arcs has cycles. Then we will "shrink" (contract) each strongly-connected component of admissible arcs to a single node and then run the blocking flow algorithm. See Figure 5.

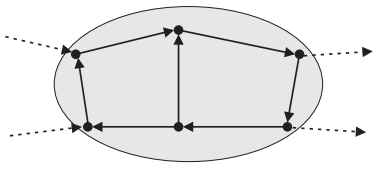


Figure 5: A set of strongly-connected components of admissible arcs to be contracted.

PROBLEM 3. How do we route flow in the "unshrunken" arcs? The arcs inside the strongly connected components have capacity at least Δ (since they had length zero), but it is possible that the total flow going in and coming out of a strongly connected component overwhelms the capacity of those arcs.

SOLUTION 3. Limit the flow so that flow in/ flow out (the dotted arcs in Figure 5) is less than $\Delta/4$ We can do this by changing the basic step in each iteration from "find a blocking flow" to "either find a flow of value $\Delta/4$ or find a blocking flow of value at most $\Delta/4$". Then given a flow on the graph with shrunken components, we can easily route the flow on the graph with unshrunken components as follows:

(i) For each of the unshrunken strongly-connected components, pick some root node, say r;

(ii) Build two trees: an *intree* to r, and an *outtree* from r (see Figure 6);

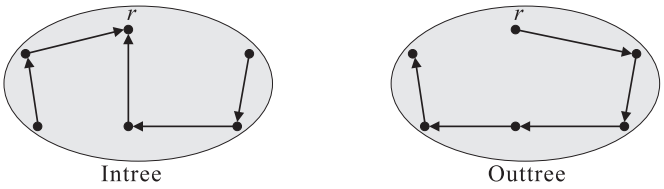


Figure 6: Intree and outtree.

(iii) Use the intree to route incoming flow to r and the outtree to route outgoing flow from r.

Each arc is then used at most twice (at most once in the intree and at most once in the outtree) so by routing at most $\Delta/4$ flow on each of these trees, we are only using $\Delta/2$ capacity of the the arcs, each of which by definition has capacity at least Δ. Recall $U \equiv \max_{i,j)\in A} u_{ij}$. We are now ready to present the algorithm by Goldberg and Rao [13]:

Algorithm 3.23 (Goldberg-Rao, 1998)**.**

$f \leftarrow 0$, $F \leftarrow mU$
while $F \geq 1$ **do**
  $\Delta \leftarrow F/(2\Lambda)$
  repeat 5Λ `times`
    `for all` $(i,j) \in A_f$ `set` $l'_{ij} \leftarrow 1$ `if` $u^f_{ij} < \Delta$ `and` $l'_{ij} \leftarrow 0$ `otherwise`
    `Compute distance` d'_i `to sink` t `for all vertices` i
    `Shrink strongly-connected components of admissible arcs`
    `Find` $\tilde{f}$ `in the shrunken graph:`
      `either a flow of value` $\Delta/4$
      `or a blocking flow of value` $\leq \Delta/4$
    $\hat{f} \leftarrow \tilde{f}$ `with flows routed in the shrunken components`
    $f \leftarrow f + \hat{f}$
  end-repeat
  $F \leftarrow F/2$
end-while

We now prove a lemma that we will need to bound the running time.

Lemma 3.14. *F is an upper bound on the flow value in G_f.*

Proof. We prove the statement by induction on the algorithm. First note that it is true initially; $F = mU$ is an upper bound on the total amount of flow. Now consider the repeat loop. After 5Λ times,

- Either $d'_s \geq \Lambda$ (i.e. we compute a blocking flow at least Λ times). By the blocking flow arguments used in unit capacity graphs, this implies that there is a cut in G_f of residual capacity at most $\Lambda\Delta = F/2$ and hence the remaining flow in G_f is less than or equal to $F/2$.

- Or flow has increased by $\Lambda\Delta = F/2$ (i.e. we found a $\Delta/4$ flow at least 4Λ times). This implies that there will be no more than $F/2$ units of flow remaining in G_f, since there were at most F initially.

In either case, we can legitimately reduce F by a factor of 2 after we have repeated the main step of the algorithm 5Λ times. □

In order for the blocking flow style proof to work, we need to have d'_s increase in each iteration, and it is not obvious that it will under the new definitions. However, we will need to make a slight change to the algorithm to get this to work out.

Lemma 3.15. *If we compute a blocking flow, then d'_s strictly increases.*

Proof. Let $\tilde{A} = \{(i,j) \in A_{f'} : d'_i = d'_j + l'_{ij}\}$ be the set of admissible arcs, where f' is the flow, d'_i are distance labels, and l'_{ij} are length labels in one iteration. Let f'' be the flow, d''_i be distance labels, and l''_{ij} length labels in the next iteration.

The proof structure we want to follow is the same that we used for Dinic's algorithm (Algorithm 3.20): we first show that the distance d' is a valid distance labelling for arcs in the residual graph of the flow in the next iteration: that is, we show that for all $(i.j) \in A_{f''}$ it is the case that $d'_i \leq d'_j + l''_{ij}$. Then we observe that by the properties of a blocking flow, in any s-t path in the residual graph of f'', there must be some arc that was not admissible in the previous iteration; that is, there is some arc (i,j) in any s-t path of $A_{f''}$ such that $d'_i < d'_j + l'_{ij}$. From this we hope to infer that the length of the path must in fact be greater than d'_s.

First, we show that *d' is a valid distance labelling for $A_{f''}$*. If (i,j) is in the residual graph for f'', then it must be the case that either (i,j) was in the residual graph for f', or that (j,i) was in the residual graph for f' and (j,i) was admissible. In the latter case, if (j,i) was admissible, then $d'_j = d'_i + l'_{ji}$, which implies that $d'_i = d'_j - l'_{ji} \leq d'_j + l''_{ij}$ and we are done. If on the other hand $(i,j) \in A_{f'}$ and $d'_i \leq d'_j + l'_{ij}$, the only bad case is if $d'_i = d'_j + l'_{ij}$, $l'_{ij} = 1$ and $l''_{ij} = 0$. Suppose we run into this case. But, $d'_i = d'_j + l''_{ij} = d'_j + 1$ implies that (j,i) is not admissible. Thus we can not have pushed any flow on (j,i), so the residual capacity of (i,j) in f' must be no less than that of (i,j) in f''. Therefore, $u^{f'}_{ij} \geq u^{f''}_{ij} \geq \Delta$, where the last inequality is due to $l''_{ij} = 0$. But $u^{f'}_{ij} \geq \Delta$ contradicts our assumption that $l'_{ij} = 1$.

Now by the properties of a blocking flow, we know that for any s-t path P in $A_{f''}$ there exists $(i,j) \in P$ such that (i,j) was not admissible; that is, $d'_i < d'_j + l'_{ij}$. We want to show that $d'_i < d'_j + l''_{ij}$. This will imply that the length of P under lengths l''_{ij} must be strictly greater than d'_s.

Suppose it is not the case that $d'_i < d'_j + l''_{ij}$, and thus $d'_i = d'_j + l''_{ij}$. What could happen so that this occurs? This can happen only if $l'_{ij} = 1$, $l''_{ij} = 0$, and $d'_i = d'_j$. However, the case that $l'_{ij} = 1$ and $l''_{ij} = 0$ can only happen if the flow was sent from j to i. This implies that (j,i) is admissible, which further implies that $l'_{ji} = 0$, since $d'_i = d'_j$. Unfortunately, nothing in the algorithm so far prevents this bad case from happening. So we make one *final change* to the algorithm to ensure that this case cannot happen. We change the definition of edge lengths as follows:

$$l'_{ij} = \begin{cases} 0, & \text{if } u^f_{ij} \geq \Delta \\ 0, & \text{if } \Delta/2 \leq u^f_{ij} < \Delta,\ d'_i = d'_j, \text{ and } u^f_{ji} \geq \Delta \text{[6]} \\ 1, & \text{otherwise} \end{cases}$$

Before continuing with the proof, we quickly observe that this change does not break the rest of the algorithm. What changes?

- Note that special arcs are admissible; since $d'_i = d'_j$ and $l'_{ij} = 0$ for special arc (i,j). Also, we have that $d'_i = d'_j + l'_{ij}$.

[6] In this case, (i,j) is called a "special arc".

- Note also that distances do not change, since for special arcs (i, j) it is already the case that $d'_i = d'_j$.
- Finally, since $l'_{ij} = 0$, special arcs (i, j) could be in shrunken strongly connected components. But we will still be able to route flow through them in the way that we mentioned earlier since the total amount of flow routed through an arc in a shrunken component was at most $\Delta/2$ (see SOLUTION 3), and the capacity of any special arc is at least $\Delta/2$.

Now let us finish the proof, taking into account the "fix". We have a bad case when (i, j) is not admissible, $l'_{ij} = 1$, $l''_{ij} = 0$, and $d'_i = d'_j$. In order to have $l'_{ij} = 1$ and $l''_{ij} = 0$ (i.e. the capacity increasing from one iteration to the next), it must have been the case that we pushed flow across (j, i) and that (j, i) is admissible. Since $d'_i = d'_j$, (j, i) admissible implies that $l'_{ji} = 0$. Since (i, j) is not admissible, it cannot be a special arc. Thus, $u^f_{ij} < \Delta/2$.

To make $l'_{ij} = 0$ (i.e. $u^{f''}_{ij} \geq \Delta$), more than $\Delta/2$ units of flow must have been pushed across (j, i). But this cannot happen, since in one iteration flow is never increased by more than $\Delta/2$ on any arc. □

We can now give the running time of the algorithm. Observe that the while-loop is executed $\log(mU)$ times; in each execution of the repeat-loop we execute a blocking flow algorithm $O(\Lambda)$ times; and we can run a blocking flow algorithm in $O(m \log n)$ time.

Theorem 3.24 (Goldberg, Rao 1998)**.** *A maximum s-t flow can be computed in $O(\Lambda m(\log n) \log(mU))$ time.* □

Note that for reasonable values of U, the running time is $o(mn)$. It is a big open question if this algorithm can be made to have a strongly polynomial running time that is also $o(mn)$.

Exercise 3.8. *How to find a minimum cut with fewest arcs?*

3.3 Applications

We discuss a couple of applications of maximum flow and minimum cut. The opening example is König's theorem. Given an undirected graph $G = (V, E)$, a subset of E is called a *matching* if no pair among them has common end. A subset of V is called a *vertex cover* if every edge of G has at least one end in it.

Exercise 3.9. *Prove König's Theorem: Let G be a bipartite undirected graph, the maximum size of a matching in G is equality to the minimum size of a vertex cover of G.*

3.3.1 Carpool fairness.

There are k people who are sharing a carpool for ℓ days. Each announces their schedule in advance.

Ex.- (4 People, 5 Days)

person	days 1	2	3	4	5
1	X	X	X		
2	X		X		
3	X	X	X	X	X
4		X	X	X	X

Table 1: Schedules of carpool.

Problem: Every day someone has to drive... We want to allocate driving responsibilities "fairly".

A possible approach/objective is to split the responsibilities equally among the people using the car on a given day. Thus on a day with h people using the carpool, each driver is responsible for a share of $1/h$.

Ex.- (Responsibilities)

person	days 1	2	3	4	5
1	1/3	1/3	1/4		
2	1/3		1/4		
3	1/3	1/3	1/4	1/2	1/2
4		1/3	1/4	1/2	1/2
$\sum$	1	1	1	1	1

Table 2: Responsibility shares.

We use these shares to calculate a driving obligation, o_i for person i. In the example, we have $o_1 = 1/3 + 1/3 + 1/4 = 11/12$. We can then require that person i drives no more than $\lceil o_i \rceil$ times (days) every ℓ days. To see if this can even be done, we formulate the problem as a network in Figure 7.

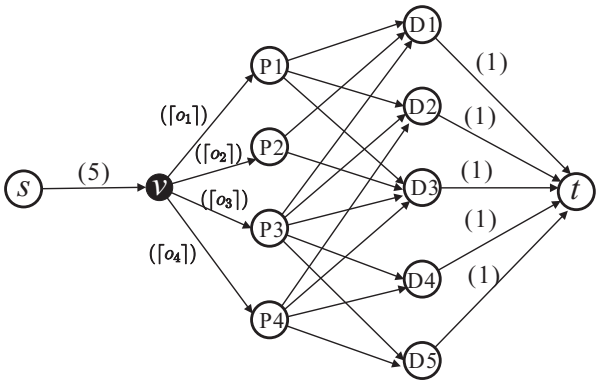


Figure 7: An s-t flow and its associated residual graph.

Exercise 3.10. *Prove the following: (i) If flow of value* 5 *exists, then a fair driving schedule exists. (ii) Such a flow always exists and a fair driving allocation always exists.*

3.3.2 Sport team elimination.

A team *wins* their division if it wins more games than the other teams in the division. A team is *eliminated* if they can not finish first given any outcome of the remaining games.

Ex.- (4 team division)

For example, team D is clearly eliminated because they will end the season with at most 91 wins while team A already have 93 wins.

Claim 3.1. *Team* B *is also eliminated.*

Proof. Team B can still win 93 games but: either the A win one more game and have 94 wins, or C wins all 6 of their games versus the A giving them 94 wins. □

Team	Wins	Remaining Games	A	B	C	D
A	93	8	-	1	6	1
B	89	4	1	-	0	3
C	88	7	6	0	-	1
D	86	5	1	3	1	-

Table 3: 4 team division.

Let T denote the set of teams in the division. For each $i \in T$, set w_i = the number of wins for team i, g_i = the number of games left to play for team i, g_{ij} = the number of games left for team i to play team j. For $R \subseteq T$, we define:

$$w(R) = \sum_{i \in R} w_i,\ g(R) = \sum_{i,j \in R, i<j} g_{ij}, \text{ and } a(R) = \frac{w(R) + g(R)}{|R|}.$$

Claim 3.2. *For any $\emptyset \neq R \subseteq T$, some team $i \in R$ wins at least $a(R)$ games.*

Proof. $w(R)$ is the number of wins of the teams in R, and $g(R)$ represents the number of games in which some team in R must win. Therefore, the average number of wins by teams in R is $a(R)$, which some team surely obtains. □

Corollary 3.25. *If $i \in T$, $R \subseteq T \setminus \{i\}$ and $a(R) > w_i + g_i$, then team i is eliminated.*

Ex.- Let $R = \{\mathsf{A}, \mathsf{C}\}$ and $i = \mathsf{B}$. Then $a(R) = \frac{(93+88)+6}{2} = 93.5 > 93$. So B is eliminated.

Now let x_{ij} denote the number of times team i defeats team j (in the remaining games). Then team k is not eliminated if: there exists $x \in \mathbb{Z}_+^{T \times T}$ satisfying

$$\begin{aligned} & x_{ij} + x_{ji} = g_{ij} && \text{for any } i, j \in T \\ & w_k + \sum_{j \in T} x_{kj} \geq w_i + \sum_{j \in T} x_{ij} && \text{for any } i \in T \\ & x_{ij} \in \mathbb{Z}_+ && \text{for any } i, j \in T \end{aligned}$$

If such an x exists, then there exists $x' \in \mathbb{Z}_+^{T \times T}$ such that team k wins all its remaining games. Therefore we only need to check that

$$w_k + g_k \geq w_i + \sum_{j \in T \setminus \{k\}} x_{ij} \text{ for all } i \in T \setminus \{k\}.$$

We can use the maximum flow instance $\mathcal{I}_k$ illustrated in Figure 8 to decide if team k has *not* been eliminated. Note that we can assume that the capacities on the arcs going from the team nodes to the sink t are nonnegative, since if $w_k + g_k - w_j < 0$, then $w_j > g_k + w_k$, and we know that team k is eliminated.

Lemma 3.16. *If a flow of value $g(T \setminus \{k\})$ exists, then team k is not eliminated.*

Proof. If a flow of value $g(T - \{k\})$ exists, then the arcs from s to the pair nodes are at full capacity. Let x_{ij} denote the flow from pair node $\{i, j\}$ to team node i. Then $x_{ij} + x_{ji} = g_{ij}$ by flow conservation at the pair node $\{i, j\}$. By the integrality property of flow, we know that the x_{ij}'s are integer. Flow conservation and capacity constraints for team node i give:

$$\sum_{j \in T \setminus \{k\}} x_{ij} \leq w_k + g_k - w_i \Rightarrow w_k + g_k \geq w_i + \sum_{j \in T \setminus \{k\}} x_{ij}.$$

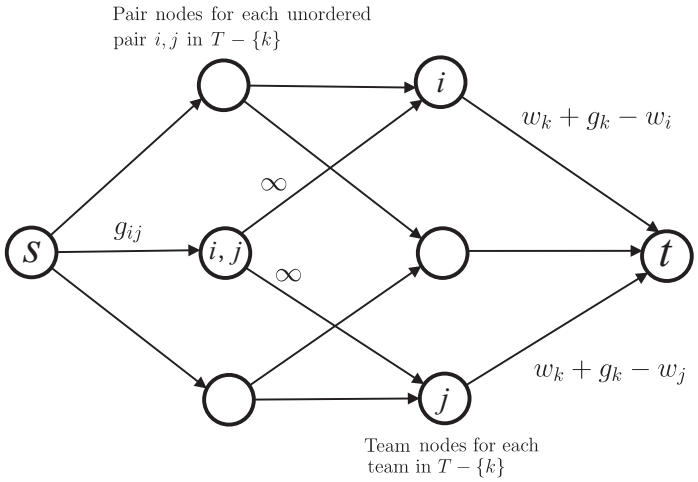


Figure 8: Flow instance for deciding if team k is not eliminated.

So x_{ij}'s satisfy the conditions given above, and k is not eliminated. □

To prove the reverse implication, we recall from Claim 3.2 the following:

Lemma 3.17. *If there exists $R \subseteq T \setminus \{k\}$ such that $a(R) > w_k + g_k$, then team k is eliminated.* □

Lemma 3.18. *If a flow of value $g(T \setminus \{k\})$ does not exist, then team k is eliminated.*

Proof. Let S be a minimum s-t cut, R be the set of team nodes in S, and P be the set of pair nodes. We can give the following expression for the capacity of S:

$$\begin{aligned} u(\delta^+(S)) &= \sum_{\{i,j\} \in P \setminus S} g_{ij} + \sum_{i \in R} (w_k + g_k - w_i) \\ &= \sum_{\{i,j\} \in P \setminus S} g_{ij} + |R|(w_k + g_k) - w(R). \end{aligned}$$

Since there exists no flow of value $g(T \setminus \{k\}) = \sum_{\{i,j\} \in P} g_{ij}$, we know that $g(T \setminus \{k\}) > u(\delta^+(S))$, and therefore

$$\sum_{\{i,j\} \in P} g_{ij} > \sum_{\{i,j\} \in P \setminus S} g_{ij} + |R|(w_k + g_k) - w(R) \Rightarrow \sum_{\{i,j\} \in S \cap P} g_{ij} > |R|(w_k + g_k) - w(R).$$

If pair node $\{i, j\}$ is in S, then both team nodes i and j are in R, otherwise the cut has infinite capacity. So the sum of g_{ij} for pair nodes $\{i, j\} \in S$ cannot be more than $g(R)$. Thus we have $g(R) > |R|(w_k + g_k) - w(R)$; or, rearranging terms once again, $\frac{w(R)+g(R)}{|R|} > w_k + g_k$. By Lemma 3.17, team k is eliminated. □

Theorem 3.26. *Team k is eliminated if and only if the maximum flow instance $\mathcal{I}_k$ has no s-t flow of value $g(T \setminus \{k\})$.* □

Exercise 3.11. *Prove that $O(\log |T|)$ maximum flow computations determine all eliminated teams.*

HINT: It suffices to prove: if k is eliminated, then for any ℓ with $w_\ell + g_\ell \leq w_k + g_k$, team ℓ is also eliminated. □

3.3.3 Market-clearing pricing.

Let us consider a problem from economics: the problem of finding prices that will make a market clear. The *market clears* if all buyers buy only goods that maximize happiness, all money is spent, and no goods remain unpurchased. We refer to this problem as the Market-Clearing Pricing Problem. This turns out to be a nice application of maximum flow techniques.

Problem 3.27. MARKET-CLEARING PRICING

INPUT: Set A of unit amounts of divisible goods;
Set B of buyers;
Integer amount of money m_i, $i \in B$;
Integer utilities u_{ij}, $i \in B, j \in A$.[7]
GOAL: Find prices p_j, $j \in A$ such that the market clears

It has been long known that prices that clear the market exist. A result of Arrow and Debreu from 1954 implies the existence of market-clearing prices, though this may not be earliest work that establishes the existence of such prices. The previous proofs that market-clearing prices exist, however, were non-constructive.

The Market-Clearing Pricing Problem was defined in 1891 by Fisher, who invented a hydraulic machine to solve it (in the case of three goods). Recently, in 2002, a polynomial time algorithm was given for the problem, demonstrating that there still exist nice problems, which are solvable in polynomial time, for which no polynomial time algorithm was previously known. We present this algorithm for computing market-clearing prices, which was developed by Devanur, Papadimitriou, Saberi, and Vazirani [6].

Characterizing market clearance using maximum flow. First, we formalize the notion that all the buyers must buy only goods that maximize their happiness in order for the market to clear. Given prices $p_j, j \in A$, the "bang per buck" that a buyer i derives from a good j is the ratio of the utility u_{ij} to the price p_j. Figure 9(a) depicts sample data and the corresponding bang per buck ratios. Buyers try to maximize the bang per buck they get for the goods they buy, and so we define

$$\alpha_i = \max_{j \in A} \frac{u_{ij}}{p_j}$$

to represent the best bang per buck that a buyer i can obtain.

A buyer i will only buy goods j such that $u_{ij}/p_j = \alpha_i$. We define a graph that represents the goods that each buyer may purchase.

Definition 3.28. *The* equality subgraph $G = (A, B, E)$ *is a bipartite graph (with node sets A and B) where $ij \in E$ if and only if $u_{ij}/p_j = \alpha_i$.*

Given a particular set of prices $p_j, j \in A$, we can determine whether the prices clear the market by performing a maximum flow computation. We add a source vertex s and a sink vertex t to the equality subgraph. For each good $j \in A$, we add an arc (s, j) with capacity p_j. For each buyer $i \in B$, we add an arc (i, t) with capacity m_i. We orient each edge ij corresponding to a buyer $i \in B$ and a good $j \in A$ in the equality subgraph as an arc (j, i) with capacity ∞. We use G' to denote this flow network. Figure 9(b) shows an example of G' for a particular collection of buyers and goods.

In this graph, flow from the source to the sink represents the transfer of money in the market. A unit of flow on an arc (j, i) from a good j to a buyer i represents a dollar spent by buyer i on good j. The total amount of flow from the source to the sink is the total amount of money spent by the buyers on goods. Therefore, the market clears (the buyers spend all their money) if and only if the maximum flow value is $\sum_{i \in B} m_i$.

[7] Utility u_{ij} is an important factor of the happiness buyer i derives from one unit of good j.

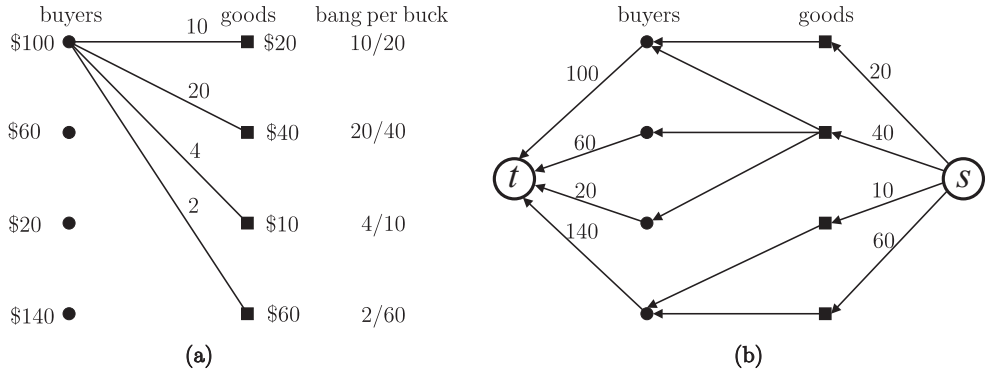


Figure 9: (a) An example of the computation of the bang per buck that a buyer obtains from different goods. The amounts of money the buyers have are shown on the left, the prices of the goods are shown on the right, and the label for an edge ij indicates the utility u_{ij}. (b) A graph in which we can compute a maximum flow to determine whether a set of prices clears a market. The arcs from goods to buyers have infinite capacity.

An exponential time algorithm. The idea behind this algorithm for the Market-Clearing Pricing Problem is to start with small prices, and to raise the prices over the course of the execution of the algorithm. We will keep the prices sufficiently low to ensure that all the goods are sold, but the buyers have left-over money (a surplus). We will maintain

INVARIANT: The singleton set $\{s\}$ is a minimum s-t cut.

This corresponds to all goods being sold. The goal will be to find prices such that $V - \{t\}$ is also a minimum s-t cut, because the capacity of the arcs crossing this cut is the total amount of money the buyers have. When this cut becomes a minimum s-t cut, the value of the maximum flow is $\sum_{i \in B} m_i$, and the market clears. We raise the prices gradually, decreasing the surplus of the buyers until it reaches zero.

`Initialization of prices.` We want to assign small initial values to the prices to ensure that $\{s\}$ is a minimum s-t cut. To initialize the prices, we set $p_j = 1/|A|$ for all $j \in A$. Under these prices, $\{s\}$ is a minimum s-t cut with value 1. We also need at least one buyer for each good. If there are no buyers for good j, we compute α_i for all buyers i. Then, we reduce the price p_j to the value $\max_{i \in B} u_{ij}/\alpha_i$. The initial setting guarantees that every single buyer can buy the totality of the goods and every single good is bought by at least one buyer.

`Raising prices.` When we raise the prices to decrease the surplus of the buyers, we would like to ensure that all edges remain in the equality subgraph. Consider a buyer i for which the edges ij and ik are both in the equality subgraph. By the definition of the equality subgraph, we have $u_{ij}/p_j = u_{ik}/p_k$, which implies that $p_k/p_j = u_{ik}/u_{ij}$. Multiplying both p_j and p_k by the same factor will leave this ratio unchanged. As such, we increase the prices from p_j to p'_j by setting $p'_j = p_j x$, $j \in A$ for some factor x.

To determine the factor x that we will use to raise the prices, we consider the different ways in which the equality subgraph may change when we raise the prices.

Event type (1): By increasing x, the invariant that $\{s\}$ is a minimum s-t cut becomes violated.

In the previous example, multiplying the prices by the factor $x = 2$ causes another minimum s-t cut to emerge, as shown in Figure 10(a). If we multiply the prices by a factor $x > 2$, then we violate the invariant, because $\{s\}$ is no longer a minimum s-t cut.

Note that in the example, the emergence of the new minimum s-t cut when the prices are raised creates a desirable scenario for the last buyer, because all the money \$140 available to that buyer can be spent on goods. In general, the market clears in the subgraph involved in the new minimum s-t cut. As a result, we can "freeze" the subgraph involved in the cut, and consider only the remaining graph when we raise the prices again. At any point in the algorithm, we refer to the subgraph in which we are increasing the prices as *active*, and to the rest of the graph as *frozen*.

Event type (2): A new edge from an active buyer to a frozen good enters the equality subgraph.

Continuing the example from above, if we take the prices that caused the event of type (1) to occur and multiply the prices for the active goods by $x = 1.25$, then an edge between an active buyer and a frozen good is created in the equality subgraph, as shown in Figure 10(b). To address this type of event, we unfreeze the good incident on the new edge, and the connected component containing the good.

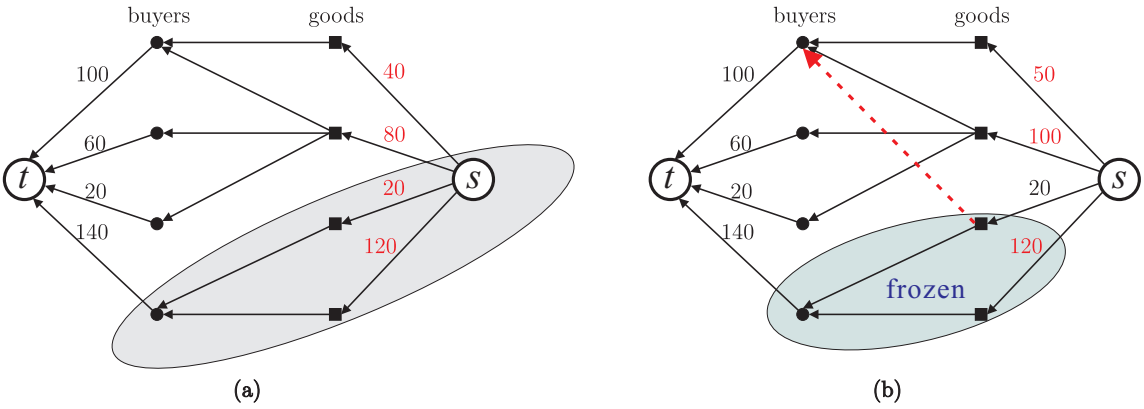


Figure 10: (a) An example of event type (1). If the prices are multiplied by a factor $x > 2$, then the cut shown becomes the minimum s-t cut. (b) An example of event type (2). Multiplying the prices of the active goods in (a) by a factor $x = 1.25$ causes the dashed edge shown between an active buyer and a frozen good to enter the equality subgraph.

Description and analysis of the algorithm. Before describing the algorithm, we prove a curial result. For any $S \subseteq A$ and $T \subseteq B$, we write $N(S) = \{i \in B : \exists\, j \in S \text{ with } ij \in E\}$, $p(S) = \sum_{j \in S} p_j$ and $\mathsf{m}(T) = \sum_{i \in T} m_i$.

Lemma 3.19. *The invariant ($\{s\}$ is a minimum s-t cut in G') holds if and only if $p(S) \leq \mathsf{m}(N(S))$ for every $S \subseteq A$.*

Proof. .($\Rightarrow$). Suppose the invariant holds. Then the value of the minimum s-t cut is $p(A)$ and every arc (s, j) of capacity p_j carries flow at full capacity. Thus, given any $S \subseteq A$, $p(S)$ units of flow are shipped from the nodes of S to the nodes of B connected to them, that is, to $N(S)$. Hence, there must be enough capacity among the buyers in $N(S)$ to ship this flow to the sink. Thus $p(S) \leq \mathsf{m}(N(S))$, as desired.

($\Leftarrow$). Suppose $p(S) \leq \mathsf{m}(N(S))$ for every $S \subseteq A$. Let $\{s\} \cup A_1 \cup B_1$ be any cut, with $A_1 \subseteq A$ and $B_1 \subseteq B$. We will prove that its capacity is at least that of $\{s\}$. For this let $A_2 = A - A_1$ and $B_2 = B - B_1$. The arcs coming out of this cut can be classified into three groups: arcs going from s to A_2, arcs going from A_1 to B_2, arcs going from B_1 to t. Notice that if there are any arcs of the second type then the capacity of the cut is infinite and there is nothing to prove. So we may assume there are no arcs of this

type. This also implies that $B_1 \supseteq N(A_1)$, and correspondingly, $\mathsf{m}(B_1) \geq \mathsf{m}(N(A_1))$. The capacity of the remaining arcs of the first and third types is clearly $p(A_2) + \mathsf{m}(B_1)$. The inequality just deduced and the hypothesis give $p(A_2) + \mathsf{m}(B_1) \geq p(A_2) + \mathsf{m}(N(A_1)) \geq p(A_2) + p(A_1) = p(A)$, as desired. □

The last lemma implies that the algorithm's invariant is *near* violation if for some factor x and some set S we have $x \cdot p(S) = \mathsf{m}(N(S))$. This motivates the following definition.

Definition 3.29. *We call a set $S \subseteq A$* tight *(with respect to a set of prices) if $p(S) = \mathsf{m}(N(S))$.*

It is easy to see that S is tight if the market clears in the part of the graph determined by $(S, N(S))$. We now state the algorithm for the Market-Clearing Pricing Problem.

Algorithm 3.30 (Market-Clearing Prices, Devanur, Papadimitriou, Saberi, Vazirani 2002)**.**

$p_j \leftarrow 1/|A|$ `for all` $j \in A$
$\alpha_i \leftarrow \max_{j \in A} u_{ij}/p_j$ `for all` $i \in B$
$p_j \leftarrow \max_{i \in B} u_{ij}/\alpha_i$ `for all` $j \in A$ `such that` $\nexists\ i \in B$ `with` $ij \in E$
`Recompute` $G = (A, B, E)$
`Frozen graph` $(F, F') \leftarrow (\emptyset, \emptyset)$, `Active graph` $(H, H') \leftarrow (A, B)$
while $H \neq \emptyset$ **do**
  `Raise prices` $p_j \leftarrow p_j x$ $(x > 1)$ `uniformly for all` $j \in H$ `until`
  `either (1) some` $S \subseteq H$ `becomes tight`
    `Move` $(S, N(S))$ `from` (H, H') `to` (F, F')
    `Remove edges between frozen buyers` F' `and active goods` H
  `or (2)` $\exists\ i \in H'$ `and` $j \in F$ `such that` $\alpha_i = u_{ij}/p_j$
    `Add` ij `to` E
    `Move the connected component containing` j `from` (F, F') `to` (H, H')
end-while
`Output` p_j, $j \in A$

We have not yet specified how to determine the minimal value of x such that either event (1) or event (2) occurs. Determining the minimum x such that event (2) occurs is not very difficult. For this, we just have to consider all pairs (i, j) of a buyer in $i \in H'$ and a product in $j \in F$ and determine $x_{ij} = p_j \alpha_i / u_{ij}$ is the minimum factor x for which the bang-per-buck factor of a good $j^* \notin F$ maximizing the happiness of i equals the bang-per-buck factor of the frozen good j:

$$\frac{u_{ij^*}}{x p_{j^*}} = \frac{\alpha_i}{x} = \frac{u_{ij}}{p_j}.$$

The minimum of these x_{ij} values is clearly the minimum x for which event (2) occurs. The discussion above implies that it can be calculated in $O(n^2)$ time, where $n = |A| + |B| + 2$.

The following lemma assures us that we can efficiently determine the minimum x for which event (1) occurs.

Lemma 3.20. *The minimum x for which event (1) occurs can be determined by means of $|A|$ maximum flow computations.*

Proof. Without loss of generality, we may assume that (A, B) is active. The same argument applies to arbitrary active subgraph. To determine such an x, we need to determine

$$x^* \equiv \min_{\emptyset \neq S \subseteq A} \frac{\mathsf{m}(N(S))}{p(S)}.$$

Let S^* denote the set that minimizes the above ratio.

We start with $x \equiv \mathsf{m}(B)/p(A) \geq x^*$, and compute maximum flow for prices $xp_j, j \in A$. If $\{s\}$ turns out to be a minimum s-t cut, then by Lemma 3.19 we know that $x = x^*$ and we are done.

If $\{s\}$ is not a minimum s-t cut, then by Lemma 3.19 we have $x > x^*$. Let $\{s\} \cup A_1 \cup B_1$ be a minimum s-t cut. Note that the value of the cut $\{s\} \cup A_1 \cup B_1$ is $x \cdot p(A - A_1) + \mathsf{m}(B_1)$. If we can show that $S^* \subseteq A_1 \subset A$, then the lemma is proven because we can recurse on $(A_1, N(A_1))$.

Claim 1: $A_1 \subset A$. If $A_1 = A$, then we must have $B_1 = B$ because the arcs between A and B have infinite capacity. But, the cut $\{s\} \cup A \cup B$ has value $\mathsf{m}(B)$ while the cut $\{s\}$ has value $x \cdot p(A)$, and we have $x \cdot p(A) = \mathsf{m}(B)$. This implies that $\{s\}$ is a minimum s-t cut, contradicting our assumption. Therefore, $A_1 \subset A$.

Claim 2: $S^* \subseteq A_1$. Let $S_1 = S^* \cap A_1$ and $S_2 = S^* - A_1$. Note that we must have $N(S_1) \subseteq B_1$ since otherwise the cut will have infinite capacity.

First observe that it must be the case that $\mathsf{m}(N(S_2) - B_1) \geq x \cdot p(S_2)$. Otherwise consider the cut $\{s\} \cup A_1 \cup S_2 \cup B_1 \cup N(S_2)$. It has value $x(p(A - A_1) - p(S_2)) + \mathsf{m}(B_1) + \mathsf{m}(N(S_2) - B_1) < x \cdot p(A - A_1) + \mathsf{m}(B_1)$, which contradicts the fact that $\{s\} \cup A_1 \cup B_1$ is a minimum s-t cut.

Note that this observation implies that it cannot be the case that $S^* = S_2$ since then $x^* < x$ implies that $\mathsf{m}(N(S^*) - B_1) \leq \mathsf{m}(N(S^*)) = x^* \cdot p(S^*) < x \cdot p(S^*)$, a contradiction to $\mathsf{m}(N(S_2) - B_1) \geq x \cdot p(S_2)$.

Thus $S_1 \neq \emptyset$. Furthermore, if $S_2 \neq \emptyset$, then we have $\mathsf{m}(N(S_2) - B_1) \geq x \cdot p(S_2) > x^* \cdot p(S_2)$, and $\mathsf{m}(N(S_2) - B_1) + \mathsf{m}(N(S_1)) \leq \mathsf{m}(N(S^*)) = x^*(p(S_1) + p(S_2))$. Subtracting $\mathsf{m}(N(S_2) - B_1) > x^* \cdot p(S_2)$ from $\mathsf{m}(N(S_2) - B_1) + \mathsf{m}(N(S_1)) \leq x^*(p(S_1) + p(S_2))$ we obtain that $\mathsf{m}(N(S_1)) < x^* \cdot p(S_1)$, which contradicts the definition of x^*. Thus, it must be the case that $S_2 = \emptyset$, implying $S^* \subseteq A_1$ as desired. □

How can we be sure that Algorithm 3.30 finishes at all? The fact that the prices never decrease gives us a hint. Nevertheless, in order to guarantee that every time a price is raised we are making some non-negligible amount of progress is made, we need something like the following lemma:

Lemma 3.21. *For any good j in a tight set S, price p_j has denominator no greater than $\Delta \equiv |A|U^{|A|}$, where $U \equiv \max_{kl} u_{kl}$.*

Proof. We begin with the observation that if S is a tight set, then every connected component[8] of S is also a tight set.

For this, suppose $C_1, C_2, \ldots, C_c$ are the connected components of S. Then $N(S) = \cup_{k=1}^{c} N(C_k)$ where the union is disjoint (by the definition of connected components). This implies $\mathsf{m}(N(S)) = \sum_{k=1}^{c} \mathsf{m}(N(C_k)) \geq \sum_{k=1}^{c} p(C - k) = p(S)$ since $\mathsf{m}(N(C_k)) \geq p(C_k)$, $k = 1, 2, \ldots, c$ by Lemma 3.19. If it were the case that $\mathsf{m}(N(C_k)) > p(C_k)$ for some k, then $\mathsf{m}(N(S)) > p(S)$ would contradict the assumption that S is tight. Hence $\mathsf{m}(N(C_k)) = p(C_k)$ for all $k = 1, 2, \ldots, c$.

Now, going back to our problem, consider any $j \in S$ and let $C \subseteq S$ be the connected component of S containing j, and $T = (A', C, E')$ a minimal tree in G than spans C. For every other $l \in C$ there is path in T connecting j and l. This path has the form $P_{jl} = \langle j, i_1, j_1, i_2, j_2, \ldots, l \rangle$, that is, it goes back and forth between C and B'. For each time the path touches a buyer $i \in B'$ between two goods j' and j^* we can write $u_{ij'}/p_{j'} = u_{ij^*}/p_{j^*}$, or $p_{j^*} = (u_{ij^*}/u_{ij'})p_{j'}$. Iterating this relation along the path we can show that $p_l = p_j(a_l/b_l)$, where each of a_l and b_l is a product of $|P_{jl}|/2$ utilities. Let us root T at j, and denote by E'' the set of edges connecting nodes in C and their parents. Clearly $|E''| = |C|$. It can be shown that b_l is the products of $|P_{jl}|/2$ edges in E''. (Exercise!)

Since we can do the same for at every good l in C, we get $\mathsf{m}(N(C)) = p(C) = \sum_{l \in C} p_l = p_j \sum_{l \in C}(a_l/b_l)$ and

$$p_j = \frac{\mathsf{m}(N(C))}{\sum_{l \in C} \frac{a_l}{b_l}} = \frac{(\prod_{e \in E''} u_e)\mathsf{m}(N(C))}{\sum_{l \in C} a_l \frac{\prod_{e \in E''} u_e}{b_l}} \leq \frac{(\prod_{e \in E''} u_e)\mathsf{m}(N(C))}{\sum_{l \in C} U^{|E''|}} \leq \frac{(\prod_{e \in E''} u_e)\mathsf{m}(N(C))}{\Delta}$$

[8] We here abuse of the terminology a bit and call $S \subseteq A$ a *connected component* if S results from the intersection of a connected component of the bipartite graph G with A.

Note that $(\prod_{e \in E''} u_e)/b_l$ is an integer for each $l \in C$. The lemma follows. □

Before we proceed to discussing the overall running time of Algorithm 3.30, we *remark that if the price p_j in iteration $k+l$ is strictly greater than the price p_j in iteration k, then the difference must be at least* $1/\Delta^2$. This result follows from the fact that for any positive integers a, b, c, d, if $a/b > c/d$ and $b, d \leq \hat{\Delta}$, then $(a/b) - (c/d) \geq 1/(\hat{\Delta}^2)$. Now we are ready to bound the overall running time of the algorithm.

Theorem 3.31. *Algorithm 3.30 runs in $O(\mathsf{m}(B)|A|^2\Delta^2 MF)$ time, where MF is the time required by a maximum flow computation.*

Proof. First, we observe that, by lemma 3.20, the time per iteration is no more than that of $|A|$ maximum flows, i.e., $O(|A| \cdot MF)$. We proceed to bound the number of iterations. By the Lemma and observation above, each time good j is frozen because of event (1), its price p_j has increased by $1/\Delta^2$. Each time event (1) happens, some good's price has increased, so we assign it to this freezing. Thus after k executions of event (1), the total surplus is at most $\mathsf{m}(B) - (k/\Delta^2)$. Thus event (1) can occur at most $\mathsf{m}(B)\Delta^2$ times. On the other hand, there can be at most $|V|$ consecutive iterations of the main loop in which event (2) occurs instead of event (1), simply because there are at most $|A|$ goods, and each time event (2) occurs one good gets unfrozen. We conclude that the total number of iterations is at most $(|A|+1)\mathsf{m}(B)\Delta^2 = O(|A| \cdot \mathsf{m}(B)\Delta^2)$ and the total running is as stated. □

Polynomial running time. Let f be the maximum flow computed in network G' at current prices p. Then $\mathsf{m}(B) - f$ is the *surplus money* with the buyers. Let us partition the running of the algorithm into *phases*; each phase terminates with the occurrence of event (1). Each phase is partitioned into *iterations* which conclude with event (2): a new edge entering the equality subgraph. The algorithm is speeded up by ensuring that in a phase, prices increase substantially. The key idea is to preemptively freeze sets that are "almost tight". Let $\epsilon > 0$ be fixed. The actions taken in case of the two events are modified as follows.

Let S be the newly tight set at the end of a phase and let p be the current prices. The algorithm executes the following extra step: Add ϵ to the price of each good in the active subgraph (the resulting prices is denoted as p') and find a minimum s-t cut S^* in the network (associated with p') that maximizes $|S^*|$. Let $S' = S^* \cap A$. Freeze $(S', N(S'))$. Clearly $S \subseteq S'$ and $\mathsf{m}(N(S')) + p'(A - S') \leq p'(A)$ gives

$$\mathsf{m}(N(S')) \leq p(S') + |S'|\epsilon.$$

Hence the surplus in this frozen subgraph $(S', N(S'))$ is

$$\mathsf{m}(N(S')) - p(S') \leq |S'|\epsilon. \tag{3.1}$$

The algorithm will continue the next phase with prices p (i.e., ϵ was added only for the purpose of finding S'). Hence, the algorithm still maintains the Invariant. Next suppose that a new edge ij enters the equality subgraph, i.e., begins satisfying $\alpha_i = u_{ij}/p_j$. The subgraph to be unfrozen is determined as follows. Add ϵ to the prices of all goods in the frozen part and compute a minimum s-t cut maximize the s-side (with edge ij added). Move the part of this subgraph that is on the t-side to the active subgraph.

Algorithm 3.32 (Market-Clearing Prices, Devanur, Papadimitriou, Saberi, Vazirani 2002)**.**

$p_j \leftarrow 1/|A|$ `for all` $j \in A$
$\alpha_i \leftarrow \max_{j \in A} u_{ij}/p_j$ `for all` $i \in B$
$p_j \leftarrow \max_{i \in B} u_{ij}/\alpha_i$ `for all` $j \in A$ `such that` $\nexists\ i \in B$ `with` $ij \in E$
`Recompute` $G = (A, B, E)$
$\epsilon \leftarrow \mathsf{m}(B)/(ne)$
while $\epsilon \geq 1/(n\Delta^2)$ **do**

```
Frozen graph (F, F') ← (∅, ∅), Active graph (H, H') ← (A, B)
while H ≠ ∅ do
  Raise prices p_j ← p_j x (x > 1) uniformly for all j ∈ H until
  either (1) some S ⊆ H becomes tight
         Add ε to prices of active goods
         Find minimum s-t cut S' in (H, H') such that |S'| is maximized
         Move S' from (H, H') to (F, F')
  or (2) ∃ i ∈ H' and j ∈ F such that α_i = u_ij/p_j
         Add ij to E
         Add ε to prices of frozen goods
         Find minimum s-t cut S'' in (F, F') such that |S''| is maximized
         Move (F ∪ F') − S'' from (F, F') to (H, H')
  Remove all edges between F' and H
end-while
ε ← ε/e end-while
Output p_j, j ∈ A
```

Let S be a set of goods in the active subgraph at any point in the current phase. Say that S is ϵ-*loose* if $\mathsf{m}(N(S)) \geq p(S) + \epsilon$.

Lemma 3.22. *At any point of the current phase of Algorithm 3.32, every subset of goods in the active subgraph is ϵ-loose.*

Proof. Consider set S of goods in the active subgraph. Partition S into subsets depending on the time at which they were added to the active subgraph in the current phase (or if they were in the active subgraph at the start of the phase). Let S_1 be the part that was came earliest. Then, by the freezing operations done by previous phases, $\mathsf{m}(N(S_1)) \geq p(S_1) + |S_1|\epsilon$. On the other hand, because of the Inavirant, $\mathsf{m}(N(S) - N(S_1)) \geq p(S - S_1)$. The lemma follows. □

As a consequence of the above lemma, when the current phase ends with a new tight set, flow must increase by at least ϵ.

Consider the situation when all goods are frozen. Let us call the running of Algorithm 3.32 till this stage an *epoch*. Since the surplus at the start of the epoch was $\mathsf{m}(B)$, the number of phases executed in the epoch is at most $\mathsf{m}(B)/\epsilon$. By the observation made above about the surplus in each frozen subgraph (see (3.1)), the total surplus at the end of the epoch (w.r.t. current prices) is no more than $|A|\epsilon$. At this stage, the algorithm reduces ϵ by a constant factor e (base of natural logs), and executes the next epoch, starting with the current prices.

In the first epoch, $\epsilon = \mathsf{m}(B)/(|A|e)$. Consider the first epoch when ϵ drops below $1/(|A|\Delta^2)$. At the end of this epoch, the surplus is less than $1/\Delta^2$. The next, and final, epoch is run with $\epsilon = 0$. By the remark following the proof of Lemma 3.21, during this epoch there cannot be a good j that appears in the newly tight set of two different phases. Hence, this epoch can have at most $|A|$ phases and terminates with market clearing prices. It is easy to see that the number of epochs is $O(\ln(B(m)\Delta^2)) = O(|A| \log U + \log(\mathsf{m}(B))$. Clearly, each epochs can consist of at most $|A|$ phases. Hence from Lemma 3.20 we obtain the main result:

Theorem 3.33 (Devanur, Papadimitriou, Saberi, Vazirani 2002)**.** *Algorithm 3.32 executes $O(|A|^3 \log U + |A|^2 \log(\mathsf{m}(B))$ maximum flow computations and finds market clearing prices.* □

Exercise 3.12. *Let $G = (V, A)$ be a digraph in which each vertex $v \in V$ is associated with a integer weight $w(v)$. Note that $w(v)$ could be positive, negative or zero. Design an algorithm for finding a subset $S \subseteq V$ such that $\delta^+(S) = \emptyset$, and $\sum_{v \in S} w(v)$ is maximized.*

3.4 Global minimum cuts

We now turn to the following global minimum cut problem, which is closely related to the minimum cut problems that we have discussed thus far. In several cases, the structure of the problem allows us to solve it more efficiently than we could by a direct application of maximum flow methods.

Problem 3.34. GLOBAL MINIMUM CUT

INPUT: Flow network $G = (V, A)$ with capacity $u \in \mathbb{Z}_+^A$
GOAL: Find nonempty $S \subset V$ that minimize $u(\delta^+(S))$.

Preliminary ideas. We will make use of the minimum s-cut when solving for the global minimum cut problem. The *minimum s-cut* for a specified $s \in V$ is a cut $S \subset V$ with $s \in S$ such that $u(\delta^+(S))$ is minimum.

Lemma 3.23. *We can find a minimum s-cut in $n-1$ maximum flows.*

Proof. Observe that there must exist some $i \notin S$. Thus if we find minimum s-i cuts for all possible $i \neq s$, one of these must also be the minimum s-cut; we just take the s-i cut that has the smallest value. □

Lemma 3.24. *We can find the global minimum cut by running a minimum s-cut algorithm twice.*

Proof. Pick any $v \in V$ and set $s = v$. Obviously if we find a minimum s-cut, we find the minimum cut among all those such that $s \in S$. We now need to find the minimum cut among all those such that $s \notin S$. To do this, we construct directed graph G' from G by reversing all its arcs; that is, for each (i, j) in G with capacity u_{ij} , add arc (j, i) to G' with the same capacity. Now find a minimum s-cut in G'. Note that any cut S in G' with $s \in S$ has the same capacity as the cut $V \setminus S$ in G. Thus the minimum s-cut in G' has the same value as the minimum cut in G that does not contain s. We take the smaller of the two cuts to obtain a global minimum cut. □

A cubic time implementation. In fact, we can also find the minimum s-cut by finding something more exotic, called a minimum X-t cut. Given nonempty $X \subset V$ and $t \in V \setminus X$, a *minimum X-t cut* is a cut S with minimum $u(\delta^+(S))$ such that $X \subseteq S$ and $t \notin S$.

Claim 3.3. *If we can find the minimum X-t cut for any X and t, we can find a minimum s-cut.*

Proof. Name the nodes as $1(= s), 2, \ldots, n$. For $i = 2, \ldots, n$, let $X = \{1, \ldots, i-1\}$ and find a minimum X-i cut. We claim that one of these X-i cuts is a minimum s-cut. To see this, let S be a minimum s-cut, and j be the smallest vertex not in S. So $\{1, 2, \ldots, j-1\} \subseteq S$, and S is a minimum X-j cut for $X = \{1, \ldots, j-1\}$. □

We will show how we can implement the idea given in the proof above to find a minimum s-cut. First we need a few definitions. Recall the distance labelling $d_i, i \in V$ defined in Definition 3.12 and constructed in Push-Relabel algorithms. A *distance level k* is the set $D_k = \{i \in V : d_i = k\}$. Distance level k is *empty* if $D_k = \emptyset$. We call distance level k a *cut level* if for $\{i\} = D_k$ and any $(i, j) \in A_f$ we have $d_i \leq d_j$. We next establish why cut levels are useful when finding minimum cuts.

Lemma 3.25. *If the distance level k is a cut level, then for $S = \{i : d_i \geq k\}$, all arcs in $\delta^+(S)$ are saturated.*

Proof. Pick any $(i, j) \in \delta^+(S)$. It follows from the definition of S that $d_i \geq k$ and $d_j < k$. If $d_i = k$ (i.e., $D_k = \{i\}$), then the definition of cut level enforces $(i, j) \notin A_f$ which implies that (i, j) is saturated. If $d_i > k$, then $d_i > d_j + 1$, and thus $(i, j) \notin A_f$ by Definition 3.12 (iv) , which again implies that (i, j) is saturated. □

The intuition is that the minimum cut can be found when there are no active nodes strictly below the cut level. Here is the implementation of the push-relabel algorithm to find the minimum s-cut.

Algorithm 3.35 (Push-Relabel Minimum s-Cut, Hao and Orlin, 1993)**.**

$X \leftarrow \{s\}$, $d_s \leftarrow n$, $d_{V\setminus\{s\}} \leftarrow \mathbf{0}$
Pick any vertex in $V \setminus X$ as t //This is equivalent to setting $t \leftarrow \arg\min_{i \in V\setminus X} d_i$
$cutval \leftarrow \infty$, $cut \leftarrow \emptyset$
Let k be the lowest cut level ($n-1$ if no cut level)
while $X \neq V$ **do**
  Run Push-Relable which selects only active nodes i with $d_i < k$
  Let k be the lowest cut level ($n-1$ if no cut level)[9]
  $S \leftarrow \{i : d_i \geq k\}$
  if $u(\delta^+(S)) < cutval$ **then** $cutval \leftarrow u(\delta^+(S))$, $cut \leftarrow S$
  $X \leftarrow X \cup \{t\}$, $d_t \leftarrow n$, saturate all arcs out of t
  $t \leftarrow \arg\min_{i \in V\setminus X} d_i$
end-while
Output $cutval$, cut

Lemma 3.26. *The nonempty distance levels k for $k < n$ are consecutive.*

Proof. This is clearly true at the beginning of the algorithm. If some distance level D_l with $l < n$ becomes empty, let i be the last node in D_l. We will consider two cases for i to leave D_l.

Case (1): i is relabelled. This happens if $e_i > 0$ and $d_i = l < k$ for the lowest cut level k. Consequently, by the condition of relabel, $d_i \leq d_j$ for any $(i,j) \in A_f$. However, $|D_l| = 1$ implies that that l, instead of k, is the lowest cut level, a contradiction.

Case (2): i is sink t and d_i is set to n at the end of the execution of the while-loop. Then, in the previous iteration, i had the minimum distance d_i. Also, since i is a sink, its distance has not increased. Therefore, i is still the minimum d_i and setting d_i to n does not contradict the lemma. □

Lemma 3.27. *If $i \notin X$, $d_i \leq n-2$.*

Proof. By induction on $|X|$. Let $i \notin X$ have the maximum distance label. Noting that at each iteration the current sink t has the lowest distance label, Lemma 3.26 implies that the distance levels between d_t and d_i are all non-empty.

Initially, $X = \{s\}$, $d_t = 0$, and since each distance level between d_i and d_t must contain at least one vertex from the remaining $n - |X| - 1$ vertices,

$$d_i \leq d_t + (n - |X| - 1) \leq n - 2.$$

Now assume that $d_i \leq d_t + (n - |X| - 1) \leq n - 2$ at the start of an iteration. At the end of this iteration, $|X|$ increases by 1 as the current sink t is added to X. The distance label of the new sink, denoted as t', increases by at most 1. This is because even if the lowest distance level becomes empty after t has been added to X, there must be a node in the next higher distance level (by the property that the non-empty distance levels are consecutive). Letting d' denote the distance labels in the next iteration and $X' = X \cup \{t\}$, we have

$$d'_i \leq d'_{t'} + (n - |X'| - 1) \leq d_t + 1 + (n - (|X| + 1) - 1) \leq n - 2.$$

The first inequality follows from Lemma 3.26, while the last one comes from the inductive hypothesis. □

[9] Note that there are no active nodes i with $d_i < k$.

Lemma 3.28. *Each time through the while-loop in Algorithm 3.35, the cut S that the algorithm finds is a minimum X-t cut.*

Proof. We know that $d_i = n$ for all $i \in X$, and $d_i \leq n-2$ for all $i \notin X$, and the sink t has the minimum distance label. Now, the way in which S is chosen implies that $X \subseteq S$ while $t \notin S$. Also, since any node with an excess is inside S and all arcs in $\delta^+(S)$ are saturated (proved in Lemma 3.25), S is a minimum X-t cut. □

Lemma 3.29. *The cut output by Algorithm 3.35 is a minimum s-cut.*

Proof. Let S^* be a minimum s-cut with capacity $c(\delta^+(S^*))$. Consider the first iteration of the while-loop for which the current sink is not in the minimum s-cut, i.e., $t \notin S^*$. Then, in this iteration $X \subseteq S^*$. The minimum X-t cut found in this execution (cf. Lemma 3.28) can have capacity at most $c(\delta^+(S^*))$. This is because S^* is also an X-t cut. Also since any X-t cut is an s-cut, its capacity is at least $c(\delta^+(S^*))$. The above statements imply that the minimum X-t cut found at the end of this iteration is a minimum s-cut. □

Lemma 3.30. *In Algorithm 3.35, there are*

(i) at most $O(n^2)$ relabels,

(ii) at most $O(mn)$ saturating pushes,

(iii) at most $O(n^3)$ non-saturating pushes.

Proof. (i) is true since each time a node other than the sink is relabelled, its distance label increases by at least 1 and the total increase is bounded by $n-2$ (by Lemma 3.27). The distance label of the sink is set to n at the end of the iteration and it is not relabelled further.

(ii) holds since between two saturating pushes on an arc, the distance labels of its end nodes must have increased by 2. Again, as the distance labels are bounded, the number of saturating pushes is $O(n)$ for any arc and $O(mn)$ overall.

(iii) can be shown using the FIFO implementation of the push-relabel algorithm and a modified potential function. □

Recall that we also showed last time that two executions of a minimum s-cut algorithm can be used to find a global minimum cut. Then the above lemmas lead to the following theorem [17]:

Theorem 3.36 (Hao, Orlin, 1994)**.** *A minimum s-cut and also a global minimum cut can be found in $O(n^3)$ time.* □

In fact, it can be shown that the algorithm can run in $O(mn \log n)$ time. We contrast this running time with the earlier "crude" estimates of $(n-1)$ and $n(n-1)$ maximum flow computations required to find a minimum s-cut and global minimum cut, respectively.

So, in directed graphs an algorithm for finding a global minimum cut is based on a maximum flow computation. Next, we look at an algorithm for finding a global minimum cut in undirected graphs which has almost nothing to do with flows.

The undirected variants. Given an undirected graph $G = (V, E)$, and nonempty $X, Y \subset V$ with $X \cap Y = \emptyset$, define $\delta(X) = \{ij \in E : |\{i,j\} \cap X| = 1\}$ and $\delta(X,Y) = \{ij \in E, |\{i,j\} \cap X| = |\{i,j\} \cap Y| = 1\}$.

Problem 3.37. Undirected Global Minimum Cut

Input: Undirected graph $G = (V, E)$ with capacity $u \in \mathbb{Z}_+^A$.
Goal: Find nonempty $S \subset V$ that minimize $u(\delta(S))$.

Given some arbitrarily chosen vertex v_1, the following procedure returns an ordering of the vertices. In each iteration, the algorithm looks at all vertices not in the set S and picks the one which maximizes the capacity of edges connecting it to nodes in S.

Procedure 3.38 (Maximum Adjacency Ordering)**.**

$S \leftarrow \{v_1\}$
for $i \leftarrow 2$ **to** n **do**
  `Choose` $v_i \leftarrow \arg\max_{v \in V \setminus S} u(\delta(S, \{v\})$
  $S \leftarrow S \cup \{v_i\}$
end-for

The ordering produced by the procedure has a very nice property (see Property 3.39 below) which at first glance looks very surprising. To prove the property, we need the following lemma. Let $\lambda(G, s, t)$ denote the value of a minimum s-t cut in G.

Lemma 3.31. *For any three vertices $p, q, r \in V$, it holds that $\lambda(G, p, q) \geq \min\{\lambda(G, r, q), \lambda(G, p, r)\}$.*

Proof. Let S be the minimum p-q cut of the graph and $p \in S$. If $r \in S$, then $\lambda(G, p, q) \geq \lambda(G, r, q)$, since S is also an r-q cut. If $r \notin S$, then $\lambda(G, p, q) \geq \lambda(G, p, r)$. In either case, the result holds. □

Property 3.39. *For MA ordering $v_1, \ldots, v_n$, $\{v_n\}$ is a minimum v_{n-1}-v_n cut (v_n-v_{n-1} cut, the order does not matter in an undirected graph).*

Proof. We know from the definition of the minimum cut that $\lambda(G, v_{n-1}, v_n) \leq u(\delta(v_n))$. We need to prove the reverse inequality. We do this through an induction on the number of nodes and edges, $|E| + |V|$.

The base case, i.e. when either $|E| = 0$ or $|V| = 2$, holds trivially. For the inductive case, there are two possibilities

(i) $v_{n-1}v_n \in E$: Let $v_{n-1}v_n = e$, $G' = G \setminus e$, $\delta' = \delta$. Observe that $v_1, v_2, \ldots, v_n$ is still an MA ordering of G', and

$$u(\delta(v_n)) = u(\delta'(v_n)) + c_e = \lambda(G', v_{n-1}, v_n) + c_e = \lambda(G, v_{n-1}, v_n).$$

The second equality is by induction and the final equality is because for each any $v_{n-1} - v_n$ cut in G' has the same value in G (adding in edge e) and vice versa.

(ii) $v_{n-1}v_n \notin E$: In this case, we need to apply the inductive hypothesis twice. First, let $G' = G \setminus v_{n-1}$. Note that $v_1, v_2, \ldots, v_{n-2}, v_n$ is an MA ordering in G', and by the inductive hypothesis,

$$u(\delta(v_n)) = u(\delta'(v_n)) = \lambda(G', v_{n-2}, v_n) \leq \lambda(G', v_{n-2}, v_n) \leq \lambda(G, v_{n-2}, v_n).$$

Now, let $G' = G \setminus v_n$. Again, $v_1, v_2, \ldots, v_{n-1}$ is an MA ordering in G', and by the construction of the ordering, and the inductive hypothesis, we have

$$u(\delta(v_n)) \leq u(\delta(v_{n-1})) = \lambda(G'; v_{n-2}, v_n) \leq \lambda(G', v_{n-2}, v_n) \leq \lambda(G, v_{n-2}, v_n).$$

Now using Lemma 3.31, we obtain

$$\lambda(G, v_{n-1}, v_n) \geq \min\{\lambda(G, v_{n-2}, v_{n-1}), \lambda(G, v_{n-2}, v_n)\} \geq u(\delta(v_n)).$$

Therefore by the principle of mathematical induction, $\lambda(G, v_{n-1}, v_n) \geq u(\delta(v_n))$ holds for any number of vertices and edges. This proves the result. □

Remarks. (i) We do not know in advance what the nodes v_{n-1} and v_n will be. (ii) The minimum v_{n-1}-v_n cut is special in that one side of the cut just consists of a single vertex.(iii) The MA ordering algorithm can be used as a subroutine to find a global minimum cut. To see this, let S be a global minimum cut. Consider two cases:

Case 1: $v_n \in S$ and $v_{n-1} \notin S$. Then Claim 3.39 implies that $u(\delta(S)) \geq u(\delta(\{v_n\}))$ since $(S, V - S)$ is a v_n-v_{n-1} cut. On the other hand, $u(\delta(\{v_n\})) \geq u(\delta(S))$ since S is a global min-cut. So, $\{v_n\}$ is a global minimum cut and we are done.

Case 2: v_{n-1} and v_n are on the same of side of the global minimum cut. Then we glue v_n and v_{n-1} to a single vertex and repeat MA ordering on a graph with one fewer vertices.

Algorithm 3.40 (Minimum Cut using Maximum Adjacency Ordering)**.**

$MC \leftarrow \infty$, $S \leftarrow \emptyset$
while $|V| > 1$ **do**
  `Compute MA ordering` $v_1, v_2, \ldots, v_{|V|}$
  if $u(\delta(v_{|V|})) < MC$ **then** $MC \leftarrow u(\delta(v_{|V|}))$, $S \leftarrow \{v_{|V|}\}$
  `Glue` $v_{|V|-1}$ `and` $v_{|V|}$ `into a single node`
end-while
`Output` S

Theorem 3.41. *Algorithm 3.40 finds a global minimum cut in $O(mn)$ time.*

Proof. Note that Procedure 3.38 runs in $O(m)$ time. □

Exercise 3.13. *Let $G = (V, E)$ be an undirected graph with nonnegative capacity $c \in \mathbb{Z}_+^A$. For each pair of vertices $u, v \in V$, let $\sigma(u, v)$ denote the capacity of a minimum u-v cut. Show that σ satisfies $\sigma(v_0, v_k) \geq \min\{\sigma(v_0, v_1), \sigma(v_1, v_2), \ldots, \sigma(v_{k-1}, v_k)\}$ for any choices of vertices $v_0, v_1, \ldots, v_k$.*

3.5 Multicommodity flows

We now move on to an more complex type of network problem: multicommodity flow. As suggested by the name, in this problem we wish to move multiple commodities between different source-sink pairs in the graph.

The input of multicommodity flow problem includes (1) a directed graph $G = (V, A)$; (2) a set of k source-sink pairs: s^a-t^a for $a = 1, \ldots, k$; (3) integer capacities $u_{ij} \geq 0$ for all $(i, j) \in A$, and (4)(optional) a set of k demands d^a for $a = 1, \ldots, k$.

For each $a = 1, \ldots, k$, let f^a be a valid s^a-t^a flow in G (f^a satisfies capacity constraints and flow conservation at vertices other than s^a and t^a). Then the f^a are a *multicommodity flow* if $\sum_{a=1}^k f_{ij}^a \leq u_{ij}$ for all $(i, j) \in A$. We define the *value* of flows in the usual way: $|f^a| = \sum_{(s^a,j)\in A} f_{s^a i}^a - \sum_{i,s^a)\in A} f_{is^a}^a$ There are several potential goals for this problem:

- Feasibility: Determine if there exist flows f^a such that $|f^a| = d^a$ for all a.
- Maximum concurrent flow: Find the maximum λ such that $|f^a| \geq \lambda d^a$ for all a.
- Maximum multicommodity flow: Maximize the total flow value $\sum_{a=1}^k |f^a|$, where the demands d^a, $a = 1, \ldots, k$ are ignored.

We shall focus on the last, the (*maximum*) *multicommodity flow problem.*

Exercise 3.14. *Show that the feasibility multicommodity flow problem reduces to the maximum multicommodity flow problem.*

3.5.1 Linear programming formulation

The multicommodity flow problem has a very simple formulation as a linear program if we use the path formulation for flows. The resulting LP will actually contain exponentially many variables and we will not worry about solving it directly. Its usefulness comes from the fact that the dual problem has only m variables and its feasibility is easily checked. The dual will also provide us some intuition about the algorithm we are going to present.

Before proceeding we introduce some notation. Let the variable X_P represent the total flow along path P. We let $\mathcal{P}^a$ be the set of all paths P from s^a to t^a, and $\mathcal{P} = \cup_{a=1}^{k} \mathcal{P}^a$. The LP formulation of the maximum commodity flow problem is the following:

$$\begin{aligned} \max & \sum_{a=1}^{k} \left(\sum_{P \in \mathcal{P}^a} X_P \right) \\ \text{s.t.} & \sum_{a=1}^{k} \sum_{P \in \mathcal{P}^a, (i,j) \in P} X_P \leq u_{ij}, \quad \forall\, (i,j) \in A \\ & X_P \geq 0 \qquad\qquad\qquad\qquad \forall\, P \in \mathcal{P} \end{aligned}$$

It's dual is

$$\begin{aligned} \min & \sum_{(i,j) \in A} u_{ij} l_{ij} \\ \text{s.t.} & \sum_{(i,j) \in P} l_{ij} \geq 1, \quad \forall\, P \in \mathcal{P} \\ & l_{ij} \geq 0, \qquad\qquad \forall\, (i,j) \in A \end{aligned}$$

In this program l might be viewed as an arc length function, in which case $l(P) = \sum_{(i,j) \in P} l_{ij}$ is the length of path P. Checking feasibility of the dual is equivalent to checking, for every a, that the length of the shortest path between s^a and t^a is at least 1. This can be easily done in polynomial time.

The comments above imply also that the dual problem is solvable in polynomial time using the ellipsoid method. Nevertheless, those methods are computationally expensive and the approximation algorithm that we are about to present might perform better in practice.

3.5.2 The Garg-Könemann approximation algorithm

The preceding discussion motivates the following algorithm for finding an approximate solution to our problem [11].

Algorithm 3.42 (ϵ-approximate maximum multicommodity flow (Garg, Könemann, 1998)**.**

$X_P \leftarrow 0$ `for all` $P \in \mathcal{P}$
$l_{ij} \leftarrow \delta$ `for all` $(i,j) \in A$ `(*)`
while $\exists\ P \in \mathcal{P}$ `s.t.` $l(P) < 1$ **do**
  $P \leftarrow \arg\min_{P' \in \mathcal{P}} l(P')$ [10]
  $u \leftarrow \min_{(i,j) \in P} u_{ij}$
  $X_P \leftarrow X_P + u$
  $l_{ij} \leftarrow (1 + \epsilon u / u_{ij}$ `for all` $(i,j) \in P$
end-for
`Pick` M `such that` X/M `is feasible` `(*)`
`Return` X/M

[10] Thus $l(P) < 1$.

To simplify the initial exposition we have left three steps of the algorithm (marked with an (*))unspecified. Later we will describe in detail how to determine optimal values for δ and M, and how to pick the path P so as to guarantee both polynomial running time of the algorithm and $(1-2\epsilon)$ optimality of the resulting multicommodity flow.

Note that this algorithm is quite different from previous flow algorithms that we have considered. We are not using the notion of a residual graph. Our solution X while running the while-loop is not even necessarily feasible; it is quite possible that the flow on an arc exceeds its capacity. Thus we scale down the flow at the end of the algorithm to ensure that the solution we return is a feasible flow.

We also notice that our problem has no integrality property. Thus we have to use some other argument to guarantee that the algorithm will end. This is what we do next.

Lemma 3.32. *Algorithm 3.42 terminates after at most $m \log_{1+\epsilon}((1+\epsilon)/\delta)$ iterations.*

Proof. Initially, for all $(i,j) \in A$, $l_{ij} = \delta$. At any point in the algorithm $l_{ij} \leq 1+\epsilon$. Indeed, l_{ij} only changes if it is in a path P of length $l(P) < 1$. Since all arcs have positive length, this means that $l_{ij} < 1$. Furthermore, l_{ij} is increased by a factor that is not above $1+\epsilon$ (since by definition $\leq u_{ij}$) so it can not become greater than $1+\epsilon$.

Also, at each iteration at least one arc has its length augmented by a factor of $1+\epsilon$. Call this arc a *tight* arc for that iteration. If a given arc e is the tight arc for i_e iterations then its length after the i_e-th such iteration is $\delta(1+\varepsilon)^{i_e}$ and since this quantity is no bigger than $(1+\epsilon)$ we conclude that $i_e \leq \log_{1+\epsilon}((1+\epsilon)/\delta)$, which imposes a bound on the total number of iterations for which e can be the tight arc. Since this bound is the same for all arcs we conclude that the total number of iterations is no more than $m \log_{1+\epsilon}((1+\epsilon)/\delta)$ □

We now show that if we scale the flow by a fixed quantity, the flow becomes feasible.

Lemma 3.33. *If we scale flows f^a by $M = \log_{1+\epsilon}((1+\epsilon)/\delta)$, then the total flow becomes feasible.*

Proof. Fix an arc (i,j). At each iteration h, if $(i,j) \in P_h$, where P_h is the selected path. The flow on this arc (i,j) is increased by u_h. If we set $a_h = u_h/u_{ij} \leq 1$, the length l_{ij} is increased by a factor of $1+a_h\epsilon$. At the end, l_{ij} is increased by a factor of $\prod_{h:(i,j)\in P_h}(1+a_h\epsilon)$. The flow on these arcs, on the other hand, is increased by $\sum_{h:(i,j)\in P_h} u_h = u_{ij}\sum_{h:(i,j)\in P_h} a_h$, starting from 0. Since initially $l_{ij} = \delta$, and at the end $l_{ij} < 1+\epsilon$, we have

$$\delta \prod_{h:(i,j)\in P_h} (1+a_h\epsilon) < 1+\varepsilon.$$

Since $a_h \leq 1$, we have $1+a_h\epsilon \geq (1+\epsilon)^{a_h}$ and

$$\delta(1+\epsilon)^{\sum_{h:(i,j)\in P_h} a_h} < \delta \prod_{h:(i,j)\in P_h} (1+a_h\epsilon) < 1+\varepsilon \Rightarrow \sum_{h:(i,j)\in P_h} a_h < \log_{1+\epsilon}\frac{1+\epsilon}{\delta} = M$$

Thus since the total amount of flow on arc (i,j) is $u_{ij}\sum_{h:(i,j)\in P_h} a_h$, if we divide the flows by M, the total amount of flow on arc (i,j) will be no more than u_{ij}, and the flow will be feasible. □

For length function l, we write $D(l) = \sum_{(i,j)\in A} u_{ij}l_{ij}$ for the dual objective function and $\alpha(l) = \min_{P'\in\mathcal{P}} l(P')$. We use l^s denote the length function, and $X^s = \sum_{P\in\mathcal{P}} X_P^s$ the primal value at the end of iteration s. We define $D(s) = D(l^s)$ and $\alpha(s) = \alpha(l^s)$. Note that the optimal objecive value

$$\beta \equiv \min_{l \text{ feasible} D(l)} = \min_{l\geq 0, \alpha(l)\neq 0} D(l)/\alpha(l).$$

This equality comes from the fact that if you divide a positive length function by its corresponding shortest path, the new shortest path becomes 1 so the length function becomes feasible.

Lemma 3.34. *Let t be the index of the last iteration. Then $1 \leq \alpha(t) \leq \delta n e^{\epsilon X^t/\beta}$.*

Proof. The termination of the algorithm implies $\alpha(t) \geq 1$. To show the other inequality, we consider how the dual objective function changes from iteration to iteration. Let P_s be the shortest path on which we augment in iteration s. For an arbitrary iteration s,

$$\begin{aligned} D(s) = \sum_{(i,j)\in P_s} u_{ij} l_{ij}^s &= \sum_{(i,j)\in P_s} u_{ij} l_{ij}^{s-1}(1+\epsilon u/u_{ij}) \\ &= D(s-1) + \epsilon u \sum_{(i,j)\in P_s} u_{ij} l_{ij}^{s-1} = D(s-1) + \epsilon(X^s - X^{s-1})\alpha(s-1) \end{aligned}$$

Thus

$$D(s) = D(0) + \epsilon \sum_{h=1}^{s} (X^h - X^{h-1})\alpha(h-1).$$

We are looking for a bound on β. If we consider length function $l^s - l^0$, since β is the minimum, we have

$$\beta \leq D(l^s - l^0)/\alpha(l^s - l^0).$$

D is linear so $D(l^s - l^0) = D(s) - D(0)$. Now, $\alpha(l^s - l^0)$ is the length of some path $Q \in \mathcal{P}$. Then $\alpha(l^s - l^0) = l^s(Q) - l^0(Q) \geq \alpha(s) - \delta n$. Then we have

$$\beta \leq \frac{D(s) - D(0)}{\alpha(s) - \delta n} \leq \frac{\epsilon \sum_{h=1}^{s} (X^h - X^{h-1})\alpha(h-1)}{\alpha(s) - \delta n}.$$

Rearranging terms gives

$$\alpha(s) \leq \delta n + \frac{\epsilon}{\beta} \sum_{h=1}^{s} (X^h - X^{h-1})\alpha(h-1).$$

Let $\alpha'(s)$ be the maximum possible value of $\alpha(s)$ given the above inequalities for $1 \leq s \leq t$. Let $\alpha'(0) = \delta n$. Then

$$\begin{aligned} \alpha'(0) &= \delta n \\ \alpha'(1) &= \delta n + \frac{\epsilon}{\beta}(X^1 - X^0)\alpha'(0) = \left(1 + \frac{\epsilon}{\beta}(X^1 - X^0)\right)\alpha'(0) \\ \alpha'(2) &= \delta n + \frac{\epsilon}{\beta}((X^2 - X^1)\alpha'(1) + (X^1 - X^0)\alpha'(0)) \\ &= \left(1 + \frac{\epsilon}{\beta}(X^1 - X^0)\right)\alpha'(0) + \frac{\epsilon}{\beta}(X^2 - X^1)\alpha'(1) \\ &= \left(1 + \frac{\epsilon}{\beta}(X^2 - X^1)\right)\alpha'(1) \end{aligned}$$

In general we obtain

$$\alpha'(s) = \left(1 + \frac{\epsilon}{\beta}(X^s - X^{s-1})\right)\alpha'(s-1) \leq e^{\epsilon(X^s - X^{s-1})/\beta}\alpha'(s-1).$$

Then applying the bound repeatedly, we get $\alpha'(s) \leq \alpha'(0)e^{\epsilon(X^s - X^0)/\beta}$, and in particular $\alpha(t) \leq \alpha'(t) \leq \alpha'(0)e^{\epsilon(X^t - X^0)/\beta}$. Since $X^0 = 0$ and $\alpha'(0) = \delta n$, we have $\alpha(t) \leq \delta n e^{\epsilon X^t/\beta}$ as desired. □

Theorem 3.43. *Algorithm 3.42 computes a $1/(1-2\epsilon)$ approximate flow for the maximum multicommodity flow problem.*

Proof. Since $\delta n e^{\epsilon X^t/\beta} \geq 1$ by Lemma 3.34, we have $X^t/\beta \geq (\ln(\frac{1}{\delta n}))/\epsilon$. Set $\delta = (1+\epsilon)((1+\epsilon)n)^{-1/\epsilon}$. This value is chosen so that $(\ln(\frac{1}{\delta n}))/M = (1-\epsilon)\ln(1+\epsilon)$. By substitution,

$$\frac{X^t}{M\beta} \geq \frac{\ln(\frac{1}{\delta n})}{M\epsilon} = \frac{(1-\epsilon)\ln(1+\epsilon)}{\epsilon}.$$

By a Taylor series argument,

$$\frac{X^t/M}{\beta} \geq \frac{(1-\epsilon)\ln(1+\epsilon)}{\epsilon} \geq \frac{(1-\epsilon)\epsilon - \epsilon^2/2}{\epsilon} \geq 1 - 2\epsilon,$$

establishing the theorem. □

4 Minimum-cost flows

A minimum-cost flow problem is to find a feasible flow of minimum cost. The feasibility is defined with respect to any of a number of equivalent models, several of which will be discussed or mentioned.

4.1 Minimum-cost circulations

Problem 4.1. Minimum-Cost Circulation

Input: A directed graph $G = (V, A)$ with cost $c \in \mathbb{Z}_+^A$, demand $l \in \mathbb{Z}_+^A$, capacity $u \in \mathbb{Z}_+^A$, where $l_{ij} \leq u_{ij}$ for all $(i,j) \in A$

Goal: Find a circulation f that minimizes the cost $\sum_{(i,j)\in A} c_{ij} f_{ij}$, where a *circulation* is a function $f : A \to \mathbb{R}_+$ satisfying $l_{ij} \leq f_{ij} \leq u_{ij}$ for all $(i,j) \in A$ and $\sum_{k:(i,k)\in A} f_{ik} = \sum_{k:(k,i)\in A} f_{ki}$ for all $i \in V$.

We will show below that this is equivalent to the more commonly studied *minimum-cost flow problem.* In the minimum-cost flow problem the input is the same (a directed graph $G = (V, A)$ with integer costs $c_{ij} \geq 0$ and integer capacities $u_{ij} \geq 0$ for each arc $(i,j) \in A$. The difference is that there are no demands l_{ij} but instead, there are integer demands b_i for each $i \in V$, such that the sum of demands over all the vertices is zero: $\sum_{i\in V} b_i = 0$. The goal of the minimum-cost flow problem is to find a flow that minimizes the cost $\sum_{(i,j)\in A} c_{ij} f_{ij}$ such that $0 \leq f_{ij} \leq u_{ij}$ for all $(i,j) \in A$ and

$$\sum_{k:(k,i)\in A} f_{ki} - \sum_{k:(i,k)\in A} f_{ik} = b_i$$

for all $i \in V$.

Theorem 4.2. *The minimum-cost flow problem and the minimum-cost circulation problem are equivalent.*

Proof. (*flow* ⇒ *circulation*) Given an instance of the minimum-cost flow problem, add a node s to the graph. For $i \in V$ with $b_i > 0$, we attach an arc (i,s) with cost 0, and set $l_{is} = u_{is} = b_i$. For $i \in V$ with $b_i < 0$, we attach an arc (s,i) of cost 0 and set $l_{si} = u_{si} = -b_i$ (See Figure 11). Note that given a feasible flow in the original problem we can get a circulation of the same cost in the modified instance since the flow coming into each node is equal to the flow going out of each node (including the node s, since $\sum_{i:b_i>0} b_i = \sum_{i:b_i<0}(-b_i)$). The reverse is also true – given a circulation in the modified instance, the flow on the arcs of the original problem is a feasible flow of the same cost. So by finding a minimum-cost circulation in the modified instance we can find a minimum-cost flow in the original instance.

(*circulation* ⇒ *flow*) For this part, we change variables. Set $f'_{ij} = f_{ij} - l_{ij}$, and $u'_{ij} = u_{ij} - l_{ij}$. Set $b_i = \sum_{k:(i;k)\in A} l_{ik} - \sum_{k:(k,i)\in A} l_{ki}$. This provides a direct transformation between the two problems. Given a feasible circulation f in the original problem, we have a feasible flow f' in the modified problem of the same cost, and vice versa. Thus by finding a minimum-cost flow in the modified instance we can find a minimum-cost circulation in the original instance. □

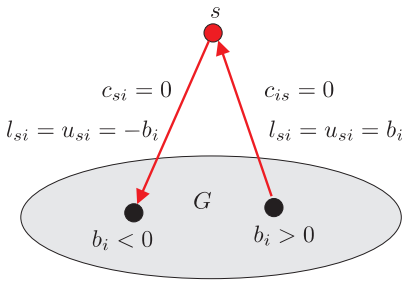


Figure 11: Transformation of minimum-cost flow instance to minimum-cost circulation instance.

From here on we will consider only the minimum-cost circulation problem.

Exercise 4.1. *Show that via one maximum flow computation, we can tell if the circulation problem is feasible and find a feasible circulation if one exists.*

Exercise 4.2. *Show that there exists a feasible circulation if and only if* $u(\delta^+(S)) \geq l(\delta^-(S))$ *holds for all nonempty proper subsets* S *of* V*.*

We will now change our notation slightly for the problem, as we did for the maximum flow problem, since it will make our algorithms and proofs simpler. Replace each arc by two arcs of opposite orientations. If f_{ij} is the flow in (i, j), then we force $f_{ji} = -f_{ij}$. This is called antisymmetry. Also set $u_{ji} = -l_{ij}$. This removes the lower bound constraints, since $f_{ji} \leq u_{ji} \Rightarrow -f_{ij} \leq -l_{ij} \Rightarrow f_{ij} \geq l_{ij}$. We make the costs antisymmetric, too: $c_{ji} = -c_{ij}$. Thus the total cost for the two edges with flow f is $c_{ji}f_{ji}+c_{ij}f_{ij} = 2c_{ij}f_{ij}$. Hence optimizing for the total cost for this new graph is the same as optimizing for the total cost for the original graph. Thus our definition of a feasible circulation becomes the following. A *circulation* is a function $\tilde{f} : \tilde{A} \to \mathbb{R}$ such that

(i) $\tilde{f}_{ij} \leq u_{ij}$ for all $(i, j) \in \tilde{A}$,

(ii) $\tilde{f}_{ij} = -\tilde{f}_{ji}$ for all $(i, j) \in \tilde{A}$, and

(iii) $\sum_{k:(i,k)\in\tilde{A}} \tilde{f}_{ik} = 0$ for all $i \in V$.

4.2 Optimality conditions

In the case of the maximum flow problem, we had conditions that told us when a flow was optimal; i.e. we knew a flow was maximum if and only if there was no augmenting path. We would like to give similar conditions for the minimum-cost circulation problem. We need a few definitions first.

A *residual graph* for a circulation f is $G_f = (V, A_f)$, where $A_f = \{(i, j) \in \tilde{A} : \tilde{f}_{ij} < u_{ij}\}$ with residual capacity

$$u^f_{ij} = u_{ij} - f_{ij}$$

and cost

$$c^f_{ij} = \begin{cases} c_{ij}, & \text{if } (i, j) \in A; \\ -c_{ji}, & \text{if } (i, j) \notin A. \end{cases}$$

Clearly $u^f_{ij} > 0$ for all $(i, j) \in A_f$. Let $p : V \to \mathbb{R}$. We often call p a *potential* on V, and $p(i)$ the potential or price of vertex $i \in V$. The potential plays the role of the dual variable. We shall show this formally latter. The *reduced cost* of (i, j) with respect to potentials p is $c^p_{ij} = c_{ij} + p_i - p_j$.

Observe that $c^p_{ij} = -c^p_{ji}$ for all $(i,j) \in A$, and that the cost of a cycle Γ and the reduced cost of a cycle Γ is the same for any set of potentials p, that is,

$$c(\Gamma) = c^p(\Gamma), \tag{4.1}$$

since the potentials cancel out (see Figure 12). Given a circulation f, we write $\sum_{(i,j)\in A} c_{ij} f_{ij}$ as $c \cdot f$.

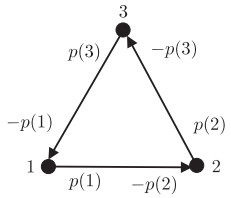


Figure 12: An example showing cost of cycle is same as the reduced cost of the cycle.

Lemma 4.1. *If f is a circulation, then $c \cdot f = c^p \cdot f$.*

Proof. Note that

$$\begin{aligned} c^p \cdot f &= c \cdot f + \sum_{(i,j)\in A} (p_i - p_j) f_{ij} \\ &= c \cdot f + \sum_{i \in V} p_i \left(\sum_{k:(i,k)\in A} f_{ik} - \sum_{k:(k,i)\in A} f_{ki} \right) \\ &= c \cdot f, \end{aligned}$$

where the last equation follows since the term in parentheses is zero because of flow conservation. □

Theorem 4.3. *The following are equivalent:*

(i) f is a minimum cost circulation.

(ii) There are no negative cost cycles in G_f.

(iii) There exists a potential p such that $c^p_{ij} \geq 0$ for all $(i,j) \in A_f$.

Proof. $\neg(ii) \Rightarrow \neg(i)$: Let Γ be a negative cost cycle in A_f. Define $\delta = \min_{(i,j)\in\Gamma} u^f_{ij}$. Then $\delta > 0$. Let

$$f'_{ij} = \begin{cases} f_{ij} + \delta, & (i,j) \in \Gamma, \\ f_{ij} - \delta, & (j,i) \in \Gamma, \\ f_{ij}, & \text{otherwise.} \end{cases}$$

Thus $f'_{ij} = -f'_{ji}$ and f' is a feasible circulation if f is. Also, $f'_{ij} \leq u_{ij}$ for all $(i,j) \in A$. Furthermore $c \cdot f' = c \cdot f + 2\delta c(\Gamma) < c \cdot f$ since Γ is a negative cost cycle. Therefore, f is not of minimum cost.
Note: In $G_{f'}$, Γ does not exist. This is so because $f'_{ij} = u_{ij}$ for some $(i,j) \in \Gamma$ by the definition of δ. We say that Γ has been *cancelled*.

$(ii) \Rightarrow (iii)$: Add a vertex s to G_f, and add arcs of cost 0 from s to each $i \in V$. Then let $p(i)$ be the length of the shortest path from s to i using costs c as the edge lengths. These paths are well defined since there are no negative-cost cycles, by assumption. Moreover, by properties of shortest paths, for any $(i,j) \in A_f$, $p_j \leq p_i + c_{ij}$, so that $c^p_{ij} = c_{ij} + p_i - p_j \geq 0$.

$(iii) \Rightarrow (i)$: Suppose f^* is any other valid circulation. We want to show that $c \cdot f \leq c \cdot f^*$. Consider the circulation f', where $f'_{ij} = f^*_{ij} - f_{ij}$. It is easy to see that f' is a feasible circulation. Let p be a

potential such that $c^p_{ij} \geq 0$ for all $(i,j) \in A_f$. Note that if $f'_{ij} > 0$ then $f_{ij} < f^*_{ij} \leq u_{ij}$. This implies $(i,j) \in A_f$ and $c^p_{ij} \geq 0$. Recalling Lemma 4.1, we have

$$\begin{aligned} c \cdot f' = c^p \cdot f' = \sum_{(i,j)\in A} c^p_{ij} f'_{ij} &= \sum_{(i,j)\in A, f'_{ij}>0} c^p_{ij} f'_{ij} + \sum_{(i,j)\in A, f'_{ij}<0} (-c^p_{ji})(-f'_{ji}) \\ &= 2\left(\sum_{(i,j)\in A, f'_{ij}>0} c^p_{ij} f'_{ij}\right) \geq 0 \end{aligned}$$

Thus $c \cdot f^* = c \cdot (f' + f) \geq c \cdot f$. Therefore f is a minimum-cost circulation. □

Exercise 4.3. *A directed Eular tour of a digraph G is a closed dipath that uses every vertex at least once and every arc exactly once. Show that G has a directed Euler tour if and only if G is connected and $|\delta^+(v)| = |\delta^-(v)|$ holds for every vertex v.*

4.3 A pseudo-polynomial time algorithm

Theorem 4.3 yields a natural algorithm for computing a minimum-cost circulation [19], whose correctness follows immediately from the theorem.

Algorithm 4.4 (Cycle-Cancelling, Klein, 1967)**.**

```
f ← a feasible circulation
while A_f contains a negative cycle Γ do
   Cancel Γ, Update f
end-while
```

The correctness of the algorithm follows immediately from the above theorem. Note that we can always find a feasible circulation, if one exists, by running one maximum flow computation (see Exercise 4.1).

Exercise 4.4. *Show that a negative cycle, if one exists, can be found in $O(mn)$ time.*

Also, notice that the algorithm implies that minimum-cost circulations, like maximum flows, satisfies an *integrality property*: If u_{ij} and c_{ij} are integer for all $(i,j) \in A$, then if a feasible circulation exists, there is always integer-valued minimum-cost circulation. This is true, since we can always cancel a cycle with integer flow during each iteration of the cycle-cancelling algorithm.

To get a bound on the running time of the algorithm, we define $U = \max_{(i,j)\in A} u_{ij}$ and $C = \max_{(i,j)\in A} |c_{ij}|$. Then any feasible circulation can cost at most mCU and must cost at least $-mCU$. Therefore, since a cycle cancellation improves the cost of a circulation by at least 1, at most $O(mCU)$ cancellations are needed in order to find an optimal circulation. This gives us a pseudo-polynomial running time of $O(m^2nCU)$.

4.4 The minimum mean-cost cycle cancelling algorithm

As with the augmenting path algorithm for the maximum flow problem, we can obtain a polynomial time algorithm by a better choice of cycle at each iteration. It turns out we can get a polynomial time algorithm [14] by cancelling the *minimum mean-cost cycle*, defined as follows. The *mean cost* of a cycle Γ is $c(\Gamma)/|\Gamma|$, where $|\Gamma|$ represents the number of arcs in Γ. The minimum mean cost cycle in A_f is given by $\mu(f) = \min\left\{\frac{c(\Gamma)}{|\Gamma|} : \Gamma \text{ is a cycle in } A_f\right\}$.

Algorithm 4.5 (Minimum Mean-Cost Cycle Cancelling, Goldberg-Tarjan, 1989)**.**

$f \leftarrow$ `a feasible circulation`
while $\mu(f) < 0$ **do**
  `Cancel a minimum mean-cost cycle in` G_f`, Update` f
end-while

Exercise 4.5. *We can compute $\mu(f)$ and find the corresponding cycle in $O(mn)$ time.*

A circulation f is ϵ-optimal if there exists potential p such that $c^p_{ij} \geq -\epsilon$ for all $(i,j) \in A_f$. By Theorem 4.3, f is 0-optimal if and only if f is a minimum cost circulation. Further, any circulation f must be Coptimal, since assigning $p(i) = 0$ for all $i \in V$ gives $c^p_{ij} \geq -C$ for all $(i,j) \in A_f$. Let $\epsilon(f)$ to be the minimum ϵ such that f is ϵ-optimal. Interestingly, the two values of $\epsilon(f)$ and $\mu(f)$ are closely related.

Lemma 4.2. *$\mu(f) = -\epsilon(f)$ for any circulation f.*

Proof. We first show that $\mu(f) \geq -\epsilon(f)$. Since there exists potential p such that $c^p_{ij} \geq -\epsilon(f)$ for all $(i,j) \in A_f$, by summing over all arcs in a minimum-mean cost cycle Γ, we obtain that $c^p(\Gamma) \geq -\epsilon(f)|\Gamma|$. Thus $\mu(f) = c(\Gamma)/|\Gamma| = j = c^p(\Gamma)/|\Gamma| \geq -\epsilon(f)$.

We now show that $\mu(f) \leq -\epsilon(f)$. Set $\bar{c}_{ij} = c_{ij} - \mu(f)$. Then for any cycle Γ in G_f, $\bar{c}(\Gamma) = c(?\Gamma) - |\Gamma|\mu(f)$. As $\mu(f) \leq c(\Gamma)/|\Gamma|$, we have $\bar{c}(\Gamma) \geq 0$. We introduce a source vertex s, connected to all vertices i with arcs of cost $\bar{c}_{si} = 0$, and define the potential $p(i)$ of vertex i to be the length of shortest path from s to i using costs c_{ij}. By the definition of a shortest path, for all $(i,j) \in A_f$, $p(j) \leq p(i) + \bar{c}_{ij} = p_i + c_{ij} - \mu(f)$, which implies $c^p_{ij} = c_{ij} + p_i - p_j \geq \mu(f)$ for all $(i,j) \in A_f$. Hence we have $\mu(f) \leq -\epsilon(f)$. □

Given circulation f, let $f^{(i)}$ denote the circulation i iterations later in Algorithm 4.5. Next show that the Goldberg-Tarjan algorithm runs in polynomial time.

Lemma 4.3. $\epsilon(f^{(1)}) \leq \epsilon(f)$.

Proof. We know there exists potential p such that $c^p_{ij} \geq \epsilon(f)$ for all $(i,j) \in A_f$. For the minimum-mean cost cycle Γ in G_f, combining (4.1) and Lemma 4.6 we have $\mu(f) = c^p(\Gamma)/|\Gamma| = -\epsilon(f)$. It follows that $c^p_{ij} = -\epsilon(f)$ for all $(i,j) \in \Gamma$. We now claim that $c^p_{ij} \geq -\epsilon(f)$ for all $(i,j) \in A_{f^{(1)}}$. We have $(i,j) \in A_{f^{(1)}}$ if either (i,j) was in A_f, or if $(j,i) \in \Gamma$. In the former case, $c^p_{ij} \geq -\epsilon(f)$. In the latter case, $c^p_{ij} = -c^p_{ji} = \epsilon(f) \geq 0 \geq -\epsilon(f)$ In both cases, it follows that $f^{(1)}$ is $\epsilon(f)$-optimal. □

Lemma 4.4. $\epsilon(f^{(m)}) \leq (1 - \frac{1}{n})\epsilon(f)$.

Proof. We know there exists potentials p such that uch that $c^p_{ij} \geq \epsilon(f)$ for all $(i,j) \in A_f$. Suppose that in some iteration k of Algorithm 4.5 we cancel cycle Γ such that there exists $(i,j) \in \Gamma$ with $c^p_{ij} \geq 0$. Then:

$$-\epsilon(f^{(k)}) = \mu(f^{(k)}) = \frac{c^p(\Gamma)}{|\Gamma|} \geq \frac{|\Gamma| - 1}{|\Gamma|}(-\epsilon(f)) \geq (1 - 1/n)(-\epsilon(f)).$$

Thus $\epsilon(f^{(k)}) \leq (1 - 1/n)\epsilon(f)$.

How many consecutive iterations can there be the case that cycle Γ that is cancelled has $c^p_{ij} < 0$ for all $(i,j) \in \Gamma$? Cancelling the cycle removes at least one arc (i,j) with $c^p_{ij} < 0$ from the residual graph and creates only arcs a with nonnegative reduced costs $c^p_a \geq 0$. So we need no more than m iterations before we cancel such a cycle Γ. Now we are done by applying Lemma 4.3. □

Lemma 4.5. *When $\epsilon(f) < 1/n$, circulation f is optimal.*

Proof. Since $\epsilon(f) < 1/n$, there exists a potential p such that $c^p_{ij} > -1/n$ for all $(i,j) \in A_f$. Thus for all cycles $\Gamma \in G_f$, the integrality of c and (4.1) imply $c(\Gamma) = c^p(\Gamma) > -|\Gamma|/n \geq -1$ and $c(\Gamma) \geq 0$. The result follows from Theorem 4.3. □

We now prove using the previous three lemmas that the Goldberg-Tarjan algorithm (Algorithm 4.5) terminates in time bounded by a polynomial in the input size.

Theorem 4.6 (Goldberg-Tarjan, 1989)**.** *Algorithm 4.5 requires at most $O(mn\log(nC))$ iterations (while-loops), and runs in $O(m^2n^2\log(nC))$ time.*

Proof. Any initial circulation is C-optimal. After $k = mn\log(nC)$ iterations, we deduce from Lemma 4.4 that

$$\epsilon(f^{(k)}) \leq \epsilon(f)(1-1/n)^{n\log(nC)} \leq C(1-1/n)^{n\log(nC)} < Ce^{-\log(nC)} = 1/n,$$

using the fact that $(1-1/n)^n < e^{-1}$. This proves the optimality of $f(k)$ by Lemma 4.5. The running time follows from the fact that minimum mean cost cycle computation takes $O(mn)$ time (Exercise 4.5). □

Strongly polynomial time analysis If an algorithm is strongly polynomial for minimum-cost circulations, its running time depends only on m and n. The first such algorithm is due to Éva Tardos [32]. She won the Fulkerson Prize for it in 1988.

An arc $(i,j) \in A$ is *ϵ-fixed* if the flow on it is the same for all ϵ-optimal circulations. Before we begin discussing conditions under which an arc becomes ϵ-fixed, we give a lemma that we will need.

Lemma 4.6. *For any circulation f, any $\emptyset \neq S \subseteq V$, it holds that $\sum_{i\in S, j\notin S,(i,j)\in A} f_{ij} = 0$.*

Proof. The flow conservation constraints $\sum_{i:(i,j)\in A} f_{ij} = 0$ give $\sum_{j\in S}\sum_{i:(i,j)\in A} f_{ij} = 0$. Recall the antisymmetry conditions that $d_{ij}+f_{ji} = 0$ for all $(i,j) \in A(S)$. It follows that $\sum_{i\in S,j\notin S,(i,j)\in A} f_{ij} = 0$. □

Lemma 4.7. *Let $\epsilon > 0$, let f be a circulation, and let p be potentials such that f is ϵ-optimal with respect to p. If $|c^p_{ij}| \geq 2n\epsilon$, then (i,j) is ϵ-fixed.*

Proof. Suppose on the contrary that f' is an optimal circulation such that $f'_{ij} = f_{ij}$. Assume that $c^p_{ij} \leq -2n\epsilon$; this is without loss of generality since costs are antisymmetric. The idea is that $f'_{ij} = f_{ij}$ will imply that there exists a cycle Γ in $G_{f'}$ containing (i,j). The cost of (i,j) is so negative that $c(\Gamma)/|\Gamma| < -\epsilon$, contradicting the ϵ-optimality of f'.

First, we show that there exists a cycle Γ in $G_{f'}$ such that $(i,j) \in \Gamma$? Since $c^p_{ij} \leq -2n\epsilon$ we know that $(i,j) \notin A_f$ because of ϵ-optimality of f. Therefore $f_{ij} = u_{ij}$. Thus we must have $f'_{ij} < f_{ij} = u_{ij}$.

Let $E_< = \{(k,l) \in A : f'_{kl} < f_{kl}\}$. Observe that $E_< \subseteq A_{f'}$ since $f'_{kl} < f_{kl} \leq u_{kl}$. Let S be the set of vertices reachable from j in $E_<$. We will show that $i \in S$S therefore a cycle Γ exists as claimed. Note that $E_< \subseteq A_{f'}$. Suppose by contradiction that $i \notin S$. Lemma 4.6 tells us that $\sum_{k\in S,l\notin S} f_{kl} = 0$ and $\sum_{k\in S,l\notin S} f'_{kl} = 0$. These together imply that

$$\sum_{k\in S,l\notin S} (f_{kl} - f'_{kl}) = 0$$

Recall that $f'_{ij} < f_{ij}$, which implies $f'_{ji} > f'_{ji}$. Therefore there is a term in the sum that is negative. Then there must be a term that is positive. So there exists $(k,l), k \in S, l \notin S$ such that $f'_{kl} < f_{kl}$, which implies $(k,l) \in E_<$. However, since $k \in S$, it must be that $l \in S$, which is a contradiction.

Therefore we know that if $|c^p_{ij}| \geq 2n\epsilon$, the (i,j) is part of a cycle Γ in the set of edges (k,l) with $f'_{kl} < f_{kl}$. Note that this implies that the reverse cycle $\Lambda = \{(l,k) : (k,l) \in \Gamma\}$ g exists in the set of arcs (l,k) with $f'_{lk} > f_{lk}$, which implies that Λ exists in G_f since flow on the arcs in this cycle cannot be at

their upper bounds. Since f is ϵ-optimal we know that $c^p_{lk} \geq -\epsilon$ for all $(l,k) \in A_f$. Therefore for any $(k,l) \in \Gamma$ we know that $c^p_{kl} \leq \epsilon$. Recall that $\mu(f') = -\epsilon(f') \geq -\epsilon$. Thus

$$\frac{c(\Gamma)}{|\Gamma|} = \frac{c^p(\Gamma)}{|\Gamma|} = \frac{1}{|\Gamma|}\left(c^p_{ij} + \sum_{(k.l)\in\Gamma\setminus(i,j)} c^p_{kl}\right) \leq \frac{-2n\epsilon + (|\Gamma|-1)\epsilon}{|\Gamma|} < \frac{-n\epsilon}{|\Gamma|} < -\epsilon.$$

Therefore there exists a cycle in $A_{f'}$ whose mean cost is less than $-\epsilon$, which is a contradiction. Therefore the flow on the arc (i,j) must be fixed. □

We now show that this leads to a strongly polynomial time running time analysis for the minimum mean-cost cycle cancelling algorithm.

Theorem 4.7. *Algorithm 4.5 terminates after $O(m^2 n \log n)$ iterations.*

Proof. Once an arc is fixed, it will always remain fixed since $\epsilon(f)$ is non-increasing. We now claim that a new arc will be fixed after at most $k = mn\log(2n)$ iterations. Let f be the current circulation and Γ be the cycle cancelled in this iteration of Algorithm 4.5. Then by Lemma 4.4

$$\epsilon(f^{(k)}) \leq (1-1/n)^{n\log(2n)}\epsilon(f) < e^{-\log(2n)}\epsilon(f) = \epsilon(f)/(2n).$$

Let p^k be the potential associated with the flow $f^{(k)}$ such that the flow is $\epsilon(f^{(k)})$-optimal. Then $-\epsilon(f) = c^{p^k}(\Gamma)/|\Gamma| < -2n \cdot \epsilon(f^{(k)})$. Therefore, there exists $(i,j) \in \Gamma$ such that $c^{p^k}_{ij} < 2n \cdot \epsilon(f^{(k)})$. It follows from Lemma 4.7 that (i,j) is $\epsilon(f^{(k)})$-fixed.

Further, note that (i,j) was not $\epsilon(f)$-fixed since $(i,j) \in T$ and the flow on it changed when we cancelled Γ. But if it was $\epsilon(f)$-fixed, the flow on it would not have changed. Therefore we fixed a new arc. □

4.5 A primal-dual algorithm

So far, the algorithm for the minimum-cost circulation problem that we have studied has been a primal algorithm. The algorithm starts with some feasible circulation and moves towards optimality. One could also consider a dual algorithm, which maintains a dual feasible solution, and moves towards optimality. We will discuss a special case of dual algorithms known (in combinatorial optimization) as primal-dual algorithms. They start with some dual feasible solution and a primal infeasible solution. The algorithm moves to reduce the infeasibility of the primal and increase the value of the dual while maintaining complimentary slackness.

To have a primal-dual method, we need first a primal and a dual. Let us go back to the original notation for the circulation problem in which we had lower bounds on the flows for each arc and did not have the antisymmetry condition. A primal LP for the minimum cost circulation problem is as follows.

$$\begin{array}{lll} & \min \sum_{(i,j)\in A} c_{ij} f_{ij} & \\ \text{s.t.} & \sum_{k:(k,i)\in A} f_{ki} - \sum_{k:(i,k)\in A} f_{ik} = 0 & \forall\, i \in V \\ & l_{ij} \leq f_{ij} \leq u_{ij} & \forall\, (i,j) \in A \end{array}$$

The dual of the LP is

$$\begin{array}{lll} & \max \sum_{(i,j)\in A} l_{ij} w_{ij} - \sum_{(i,j)\in A} u_{ij} z_{ij} & \\ \text{s.t.} & p_j - p_i + w_{ij} - z_{ij} = c_{ij} & \forall\, (i,j) \in A \\ & w_{ij} \geq 0, z_{ij} \geq 0 & \forall\, (i,j) \in A \end{array}$$

Now, suppose that the vertex potential p is given. The reduced cost is $c^p_{ij} = c_{ij} + p_i - p_j = w_{ij} - z_{ij}$. If we know the potential, then we can compute the best possible setting of the dual variables w and z (that is, the ones that will maximize the objective function) by setting

$$w_{ij} = \max\{c^p_{ij}, 0\} \equiv (c^p_{ij})^+ \text{ and } -z_{ij} = \min\{c^p_{ij}, 0\} \equiv (c^p_{ij})^-.$$

Therefore, finding potentials p yields a solution to the dual and the following LP is equivalent to the dual LP:

$$\begin{array}{lll} & \max \sum_{(i,j)\in A} l_{ij}(c^p_{ij})^+ + \sum_{(i,j)\in A} u_{ij}(c^p_{ij})^- & \\ \text{s.t.} & c_{ij} + p_i - p_j = c^p_{ij} & \forall\, (i,j) \in A \\ & w_{ij} \geq 0, z_{ij} \geq 0 & \forall\, (i,j) \in A \end{array}$$

Then by complementary slackness we have

$$\begin{array}{rcl} c^p_{ij} > 0 & \Leftrightarrow & w_{ij} > 0 \Rightarrow f_{ij} = l_{ij} \\ c^p_{ij} < 0 & \Leftrightarrow & z_{ij} > 0 \Rightarrow f_{ij} = u_{ij} \end{array}$$

In general, the primal-dual method works as follows. We start with some dual feasible solution. We then check whether or not we can find a feasible primal solution that obeys the complementary slackness conditions with respect to the current dual. If so, then we have a feasible primal and feasible dual that obey complementary slackness with respect to each other, and thus must be optimal. If not, then we claim that we can find some way to modify the dual so that the dual objective function increases, and we repeat the step of checking for a feasible primal solution that obeys complementary slackness with respect to the current dual.

Algorithm 4.8 (Primal-Dual)**.**

```
Find feasible dual (p ← 0)
while current primal doesn't obey complementary slackness w.r.t. current dual do
   Get a direction of dual increase and update
end-while
```

Our primal-dual algorithm for the minimum cost circulation problem will start with a dual feasible solution by setting all potentials equal to 0. We will then determine whether there exists a primal feasible solution that obeys complimentary slackness by defining a new circulation problem with modified upper and lower bounds $\tilde{u}$ and $\tilde{l}$.

$$\begin{array}{rcl} c^p_{ij} > 0 & \Rightarrow & \tilde{l}_{ij} = \tilde{u}_{ij} = l_{ij} \\ c^p_{ij} < 0 & \Rightarrow & \tilde{l}_{ij} = \tilde{u}_{ij} = u_{ij} \\ c^p_{ij} = 0 & \Rightarrow & \tilde{l}_{ij} = l_{ij}, \tilde{u}_{ij} = u_{ij} \end{array}$$

As with most primal dual approaches, we have reduced a problem with cost to a problem without cost where we only need to check for feasibility. If we can find a feasible circulation in the problem with bounds $\tilde{l}$ and $\tilde{u}$, we are finished, since then we will have a primal feasible solution and a dual feasible solution that obey the complementary slackness conditions, and thus are optimal.

If not, then by Hoffman's circulation theorem (see Exercise 4.2) we can find a cut S such that $\tilde{l}(\delta^+(S)) > \tilde{u}(\delta^-(S))$. We could check for feasibility and find such an S with one maximum flow computation. We will modify the dual to increase the dual objective function. To do this, we will increase the potentials of vertices in the cut S by a value β. This will lead to an increase in the dual objective function.

Lemma 4.8. *If there is no feasible circulation with respect to bounds $\tilde{l}$ and $\tilde{u}$, then we can increase the dual objective function.*

Proof. We want to adjust the reduced cost c^p so that the c^p do not flip sign for all $(i,j) \in A$. We do this by increasing p_i by β for all $i \in S$, where S is the cut satisfying $\tilde{l}(\delta^+(S)) > \tilde{u}(\delta^-(S))$. We then have:

$$c_{ij}^p = \begin{cases} c_{ij}^p + \beta, & \text{if } (i,j) \in \delta^+(S) \\ c_{ij}^p - \beta, & \text{if } (i,j) \in \delta^-(S) \\ c_{ij}^p, & \text{otherwise} \end{cases}$$

So in order to preserve the signs of c_{ij}^p, we set

$$\beta = \min\{\min\{|c_{ij}^p| : (i,j) \in \delta^+(S), c_{ij}^p < 0\}, \min\{c_{ij}^p : (i,j) \in \delta^-(S), c_{ij}^p > 0\}\} > 0.$$

We now consider the change, denoted as Δ, in the dual objective function:

$$\Delta = \beta \left(\sum_{(i,j)\in\delta^+(S), c_{ij}^p \geq 0} l_{ij} - \sum_{(i,j)\in\delta^-(S), c_{ij}^p > 0} l_{ij} + \sum_{(i,j)\in\delta^+(S), c_{ij}^p < 0} u_{ij} - \sum_{(i,j)\in\delta^-(S), c_{ij}^p \leq 0} u_{ij} \right)$$

Observe that

$$\sum_{(i,j)\in\delta^+(S), c_{ij}^p \geq 0} l_{ij} + \sum_{(i,j)\in\delta^+(S), c_{ij}^p < 0} u_{ij} = \sum_{(i,j)\in\delta^+(S)} \tilde{l}_{ij} = \tilde{l}(\delta^+(S)),$$

$$\sum_{(i,j)\in\delta^-(S), c_{ij}^p > 0} l_{ij} + \sum_{(i,j)\in\delta^-(S), c_{ij}^p \leq 0} u_{ij} = \sum_{(i,j)\in\delta^+(S)} \tilde{u}_{ij} = \tilde{u}(\delta^-(S)).$$

Therefore $\Delta = \beta(\tilde{l}(\delta^+(S)) - \tilde{u}(\delta^-(S))) > 0$, and we have a dual objective function increase. □

Since the costs and initial potentials are integral, we have that β is integral and so the next potential will remain integral. Furthermore, if the bounds l and u are integral, then the dual objective function increase will also be integral. This will give a pseudo-polynomial time algorithm that requires $O(mCU)$ maximum flow computations. In fact, clever analysis will give an algorithm that requires $O(\min\{nC, nU\})$ maximum flow computations.

4.6 A cost scaling algorithm

The primal-dual algorithm discussed above has its running linearly dependent on the input numbers. We show how to reduce, and eventually eliminate this dependence by cost scaling.

Algorithm 4.9 (Cost Scaling, Goldberg, Targan, 1990)**.**

```
Let f be any feasible circulation
ε ← C,  p ← 0
while ε ≥ 1/n do
        ε ← ε/2
        (f, p) ← Run Subroutine:  find ε-optimal circulation given input (f, ε, p)
end-while
```

The idea is that given a (2ϵ)-optimal circulation f with respect to potentials p, the sub- routine will find an ϵ-optimal circulation f' with respect to potentials p'. Since the initial circulation is C-optimal and the final f is $< (1/n)$-optimal (and hence optimal by Lemma 4.5), we will require $\log(nC)$ iterations of the while-loop. We can also show that the number of iterations is strongly polynomial by tweaking one of our previous theorems.

Lemma 4.9. *For circulation f and $\epsilon' < \epsilon(f)/(2n)$, the set of ϵ'-fixed arcs strictly contains the set of $\epsilon(f)$-fixed arcs.*

Proof. Clearly if an arc is ϵ'-fixed, it is also $\epsilon(f)$-fixed. We want to show that there exists an arc that is ϵ'-fixed, but not $\epsilon(f)$-fixed. Let p be the potential such that f is $\epsilon(f)$-fixed. Then there exists a cycle $\Gamma \in G_f$ such that $-\epsilon(f) = c^p(\Gamma)/|\Gamma|$ by Lemma . We also know that $c^p_{ij} \geq -\epsilon(f)$ for all $(i,j) \in A_f$ by definition. Hence $c^p_{ij} = -\epsilon(f)$ for all $(i,j) \in \Gamma$

If we cancel cycle Γ, the resulting circulation, denoted as $\hat{f}$, is still $\epsilon(f)$-optimal (by Lemma 4.6. Thus no arc in Γ is $\epsilon(f)$-fixed. Now let f' be any ϵ'-optimal circulation with respect to potential p'. Then $-\epsilon(f) = c^p(\Gamma)/|\Gamma| = c^{p'}(\Gamma)/|\Gamma| < -2n\epsilon'$, and thus there exists $(i,j) \in \Gamma$ such that $c^{p'}_{ij} < -2n\epsilon'$. Therefore (i,j) is ϵ'-fixed (by Lemma 4.7), but not $\epsilon(f)$-fixed. □

We now want to claim the following corollary: "Every $\log(2n)$ iterations of the while-loop of Algorithm 4.9, a new arc is fixed". But note that the lemma states that ϵ' must be a factor of $2n$ less than $\varepsilon(f)$, not just any ϵ such that f is ϵ-optimal. In order to make this true, we need add one more step to the Cost Scaling algorithm.

Algorithm 4.10 (Cost Scaling, Goldberg, Targan, 1990)**.**

```
Let f be any feasible circulation
ε ← C, p ← 0
while ε ≥ 1/n do
    Find potential p such that f is ε(f)-optimal, ε ← ε(f)
    ε ← ε/2
    (f, p) ← Run Subroutine[11]: find ε-optimal circulation given input (f, ε, p)
end-while
```

This additional step will only decrease ϵ as the algorithm continues. Then we can claim the corollary above. Since we can fix at most m arcs, we have the following lemma.

Lemma 4.10. *After* $\min(m\log(2n), \log(nC))$ *iterations, Algorithm 4.10 finds a minimum cost circulation.*

The subroutine for finding ϵ-optimal circulation We present an algorithm for the subroutine find-2-opt-circ(f, ϵ, p) based on the ideas from the push-relabel algorithm that we saw for the maximum flow problem.

FIND-ϵ-OPT-CIRC

INPUT: 2ϵ-opt circulation f, potential p s.t. $c^p_{ij} \geq -2\epsilon$ for all $(i,j) \in A_f$

OUTPUT: ϵ-opt circulation f', potential p' s.t. $c^{p'}_{ij} \geq -\epsilon$ for all $(i,j) \in A_{f'}$

The basic idea of the algorithm is that we will first convert the 2ϵ-optimal circulation to an ϵ-optimal *pseudoflow*, and then convert the ϵ-optimal pseudoflow to an ϵ-optimal circulation. A *pseudoflow* $f : A \to \mathbb{R}$ satisfies: $f_{ij} = -f_{ji}$ and $f_{ij} \leq u_{ij}$ for all $(i,j) \in A$.

Note that a pseudoflow obeys antisymmetry and capacity constraints but not flow conservation. For pseudoflow f, the *excess* at vertex $i \in V$ is $e^f_i = \sum_{k:(k,i)\in A} f_{ki}$. Note that this quality may be negative. If so, then negative excess is sometimes called a *deficit*. A vertex with positive excess is called *active*.

How can we convert a 2ϵ-optimal circulation to an ϵ-optimal pseudoflow? It is easy; we just saturate every edge with negative cost. That is, for $(i,j) \in A$ with $c^p_{ij} < 0$, set f_{ij} to u_{ij} . Then f is a 0-optimal pseudoflow.

[11] See Procedure 4.11.

To use a push/relabel scheme, we need to specify the conditions needed (and actions taken) for doing a push operation and a relabel operation. Obviously, in order to get from a pseudoflow to a circulation, we would like to get rid of all excesses; following the idea of the push/relabel algorithm for maximum flow, we will do a push on vertices with positive excess. Recall that in the maximum flow case, we only pushed along admissible arcs that met some criterion with their distance label. What should be the concept of an admissible arc in this case? Here we say an arc (i, j) is *admissible* if $c^p_{ij} < 0$. Thus we push from vertex i with $e^f_i > 0$ if there exists j such that $u^f_{ij} > 0$ and $c^p_{ij} < 0$. As in the maximum flow case, we will push $\delta = \min\{e^f_i, u^f_{ij}\}$ units of flow along (i, j).

Observe that ϵ-optimality is maintained during a push operation on (i, j) since if (j, i) is created in the residual graph, it will have reduced cost $c^p_{ji} = -c^p_{ij} > 0$.

What happens during a relabel operation? We need to relabel if there is excess at a vertex i, but there are no admissible arcs leaving i. In this case, all arcs with residual capacity must have non-negative reduced cost. To create some admissible arc, we will simply alter the potential $p(i)$ at vertex i. In particular, we set

$$p_i \leftarrow \max_{(i,j)\in A_f} (p_k - c_{ij} - \epsilon).$$

Note that after a relabel operation, we have $c_{ij} + p_i - p_j \geq -\epsilon$ for all $(i, j) \in A_f$, and $c_{ij} + p_i - p_j = -\epsilon$ for some $(i, j) \in A_f$. Therefore, p_i is decreased by at least ϵ, and f maintains ϵ-optimality. Putting these together, we obtain the following subroutine.

Procedure 4.11 (Push-Relabel find-ϵ-opt-circ(f, ϵ, p))**.**

$f_{ij} \leftarrow u_{ij}$ `for all` $(i, j) \in A_f$ `with` $c^p_{ij} < 0$
while $\exists$ `active` $i \in V$ **do**
  if $\exists$ j `such that` $u^f_{ij} > 0$ `and` $c^p_{ij} < 0$ **then** Push $\delta = \min\{e^f_i, u^f_{ij}\}$ `flow on` (i, j)
  else Relabel $p_i \leftarrow \max_{(i,j)\in A_f}\{p_j - c_{ij} - \epsilon\}$
end-while
`Return` (f, p)

Lemma 4.11. *Let f be a pseudoflow, f' a circulation. For any i with $e^f_i > 0$, there exists j such that $e^f_j < 0$ and there exists a path P from i to j with $(k, l) \in A_f$, $(l, k) \in A_{f'}$ for all $(k, l) \in P$.*

Proof. We first claim that we can find P in set of arcs $A_< = \{(i, j) : f_{ij} < f'_{ij}\}$. Note $A_< \subseteq A_f$ since $f_{ij} < f'_{ij} \leq u_{ij}$. Further note that if $(i, j) \in A_<$, then $(j, i) \in A_{f'}$ since then $f'_{ji} = -f'_{ij} < -f_{ij} = f_{ji} \leq u_{ji}$. Thus given a vertex i such that $e^f_i > 0$, it will be sufficient to find a path in $A_<$ to some j such that $e^f_j < 0$.

To do this, let S be all vertices reachable from i using arcs in $A_<$. Then,

$$-\sum_{k\in S} e^f_k = \sum_{k\in S}\sum_{j:(k,j)\in A} f_{kj} = \sum_{k\in S, j\notin S, (k,j)\in A} f_{kj} \geq \sum_{k\in S, j\notin S, (k,j)\in A} f'_{kj}$$

The inequality holds because $(k, j) \notin A_<$. Since f' is a circulation, $\sum_{k\in S, j\notin S, (k,j)\in A} f'_{kj} = 0$ (see Lemma 4.6). It follows that $\sum_{k\in S} e^f_k \leq 0$. Since $e^f_i > 0$ and $i \in S$, there must be some $j \in S$ such that $e^f_j < 0$. Furthermore, j is reachable from i using arcs of $A_<$. □

Using the lemma above, we can now bound the amount that the potential of any vertex changes during the course of Procedure 4.11.

Lemma 4.12. *For any $i \in V$, potential p_i decreases by at most $3n\epsilon$ during Procedure 4.11.*

Proof. Let f' be the initial 2ϵ-optimal circulation, and p' the initial potential. We consider the last point in the procedure during which p_i is relabelled. Note that if p_i is relabelled, then $e_i^f > 0$. By Lemma 4.11, we know there is $j \in V$ such that $e_j^f < 0$ and there is a path P from i to j in A_f, with the reverse of the path in $A_{f'}$.

First, observe that f being ϵ-optimal implies

$$-|P|\epsilon \leq \sum_{(k,l)\in P} c_{kl}^p = \sum_{(k,l)\in P} (c_{kl} + p_k - p_l) = \left(\sum_{(k,l)\in P} c_{kl} \right) + p_i - p_j.$$

Next, observe that since f' is 2ϵ-optimal and the reverse of P from j to i is in $A_{f'}$ implies that

$$-2|P|\epsilon \leq \sum_{(k,l)\in P} c_{lk}^{p'} = \left(\sum_{(k,l)\in P} c_{lk} \right) + p'_j - p'_i$$

Finally, observe that by our definition of costs $\sum_{(k,l)\in P} c_{kl} = -\sum_{(k,l)\in P} c_{lk}$. Thus by adding the previous inequalities, we get

$$-3|P|\epsilon \leq (p_i - p'_i) + (p'_j - p_j)$$

Because $e_j^f < 0$, the vertex j must not have been relabelled to this point in the procedure, and thus $p_j = p'_j$. Therefore we have that $-3n\epsilon \leq p_i - p'_i$. Since we assumed that this was the last point in the procedure during which i was relabelled, the lemma statement follows. □

The corollary below follows immediately from the fact that every relabelling decreases the potential at a vertex by at least ϵ.

Corollary 4.12. *The number of relabels per vertex is at most $3n$, and there are at most $3n^2$ relabels in total.*

As with the push/relabel algorithm for maximum flows, the bounds on the remaining push operations follow almost immediately from this. Recall that a *Push* operation is said to be saturating if $\delta = u_{ij}^f$, or non-saturating otherwise (in which case $\delta = e_i^f$). As in the case of the push/relabel algorithm for the maximum flow problem, we now bound the number of push operations by considering the two types of pushes separately.

Lemma 4.13. *The number of saturating pushes in Procedure 4.11 is at most $3nm$.*

Proof. Pick any arc (i, j). Initially, $c_{ij}^p \geq 0$ if $u_{ij}^f > 0$. Therefore, we have to relabel i before we can push on (i, j), since for (i, j) to be admissible, we need $c_{ij}^p < 0$. Having had a saturating push on (i, j), in order to push flow again on it, we must first push flow back on (j, i), which implies $c_{ji}^p < 0$, which in turn implies $c_{ij}^p > 0$. Therefore, we need to relabel i once more to push flow on (i, j) again. This together with Lemma 4.12 leads directly to a bound of at most $3n$ saturating pushes on (i, j). Thus for all m arcs in the graph, there can be at most $3nm$ saturating pushes. □

Now we wish to find an upper bound for the total number of non-saturating pushes in Procedure 4.11. We need the following lemma to help us with this bound.

Lemma 4.14. *The set of admissible arcs is acyclic.*

Proof. We prove this lemma by induction on the procedure. The base case of the procedure is simple since initially no admissible arcs exist. Now suppose that the claim holds in the middle of the procedure. Each time a push is executed, it can only remove admissible arcs from the residual graph, but cannot

add them, so the claim holds. Each time a relabel is executed, it adds admissible outgoing arcs of vertex i, but removes all of i's admissible incoming arcs because all of the reduced costs of the arcs entering i are increased by at least ϵ. Thus no cycles can be created by the new admissible arcs coming out of vertex i. □

Lemma 4.15. *The number of non-saturating pushes in Procedure 4.11 is $O(n^2m)$.*

Proof. Define Φ_i to be the number of vertices reachable from i via the admissible arcs, and let $\Phi = \sum_{\text{active } i} \Phi_i$. Initially $\Phi \leq n$ (since every vertex can reach only itself); when the procedure terminates, Φ =0, since there are no active vertices.

What makes Φ increase? A saturating push on the arc (i, j) could result in a new active vertex j, and therefore Φ can increase by at most n. In addition, a relabel can increase Φ_i by at most n, but for a vertex j such that $j \neq i$, the relabel does not increase Φ_j, since all arcs entering i are no longer admissible. So, by Lemma 4.13 and Corollary 4.12 the amount that Φ increases is at most $n(3nm+3n^2)$.

What makes Φ decrease? From the procedure, we see that a non-saturating push decreases Φ by at least 1: after such a push, i has turned inactive, and even if some other vertex j became active as a result of the non-saturating push, it would still reach fewer vertices than i by the acyclicity of the admissible arcs (see Lemma 4.14). So, the total number of non-saturating pushes in this procedure has to be at most $n(3nm + 3n^2) = O(n^2m)$. □

From the above lemmas, we see that the total number of push/relabel operations of the algorithm is at most $O(n^2m)$. Given an implementation with $O(1)$ time per operation (which we will not discuss), we may obtain the overall computational time of the Push/Relabel find-opt-circ subroutine:

Theorem 4.13. *Procedure 4.11 takes $O(n^2m)$ time. Furthermore, with a FIFO implementation of Push/Relabel, the procedure runs in $O(n^3)$ time.*

The overall running time Combining Theorem 4.13 with the bound on the number of iterations of the cost-scaling algorithm (see Lemma 4.10), we obtain the following.

Theorem 4.14 (Goldberg, Tarjan, 1990)**.** *Algorithm 4.10 (the cost-scaling algorithm for the minimum-cost circulation problem) can be implemented in $O(n^3 \min\{\log(nC), m \log n\})$ time.*

Note that if we replace Push/Relabel find-opt-circ with a subroutine based on blocking flows, the cost-scaling algorithm can be shown to run in $O(mn \log n \min\{\log(nC), m \log n\})$ time.

We close our performance analysis of the cost-scaling algorithm with two open questions. First, is a minimum cost circulation problem solvable with $O(\min\{m \log n, \log(nC)\})$ iterations of any maximum flow algorithm? It looked like the push/relabel algorithm could be used as the find-opt-circ subroutine with only minor modifications; this is also the case for the blocking flow variant of this subroutine.

More to the point, can the Goldberg-Rao maximum flow algorithm be used for this subroutine? This would then give us a minimum cost circulation algorithm that runs in $O(\Lambda m \log n(\log(mU))(\log(nC)))$ time, which would be the fastest known algorithm.

To these questions we have no definitive answers. The current best strongly polynomial time bound is due to Orlin, whose minimum cost circulation algorithm runs in $O(m \log n((m + n \log n))$ time.

4.7 An application to optimal loading

We look at a problem on the optimal loading of a hopping aircraft. Consider an airplane that makes stops at Beijing, Shanghai, Hangzhou, and Guangzhou, at each stop picking up some passengers. Our goal is to maximize the revenue while obeying the airplane's seating capacity constraints.

To solve this problem we first make some definitions as follows: b_{ij} is the number of passengers who want to travel from i to j; f_{ij} is the fare for passengers travelling from i to j; u is the capacity of the

airplane. Figure 13 below shows how we may transform this problem into a minimum cost flow problem. In the figure vertex 1 is Beijing, vertex 2 Shanghai, vertex 3 Hangzhou, and vertex 4 Guangzhou. We create arcs from vertex i to vertex $i+1$ of capacity u for each i; this corresponds to the capacity of the aircraft. For each value b_{ij}, we create a vertex with a demand $-b_{ij}$ at the vertex; each vertex has two arcs, one pointing to vertex i and one pointing to vertex j. The cost of the arc to vertex i is $-f_{ij}$; this corresponds to the revenue we get for each passenger traveling from i to j. The cost of the arc to vertex j is zero; we get no revenue from passengers we do not transport to vertex j in the aircraft. We put demands at each of the four city vertices corresponding to the number of passengers who want to end up there; that is $b_1 = 0$, $b_2 = b_{12}$, $b_3 = b_{13} + b_{23}$ and $b_4 = b_{14} + b_{24} + b_{34}$.

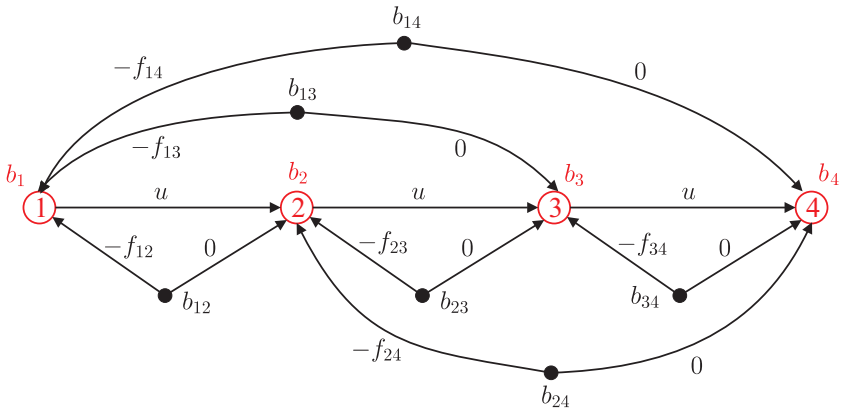


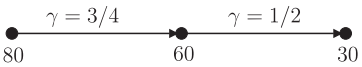
Figure 13: The optimal loading a hopping aircraft.

Exercise 4.6. *Formulate the problem of finding a minimum-cost perfect matching in a bipartite graph as a minim-cost flow problem.*

Exercise 4.7. *Suppose that we are given a connected digraph $G = (V, E)$ in which $|\delta^+(v)| + |\delta^-(v)| \equiv 0 \pmod 2$ for every vertex $v \in V$, and each arc $(u, v) \in A$ is associated with a cost c_{uv} of reversing (u, v) to be (v, u). Find a minimum cost subset of arcs to reverse, so that the resulting digraph has a directed Euler tour.*

5 Generalized flows

We discuss a model in which the edges are also "lossy", so the flow is no longer conserved, but transformed along edges. This models leaks, theft, taxes, etc.



In the above graph, if we start with 80 units of flow, we obtain 60 units after following the first arc and 30 units after the second arc. We call the parameter γ the "gain" of the arc.

Another application for this model would be converting currency. Consider, for instance, the graph below in which we want to convert, say, \$1000 into Hungarian forints. Besides the "gain" factor we can also add, as before, capacity constraints to (some) arcs, for example we can convert at most \$800 directly into forints. Note that some paths lead better rates than others; for example, the dollar → euro → forint path gives an exchange rate of 6 forints per dollar as opposed to the direct path for which the rate is just 5.

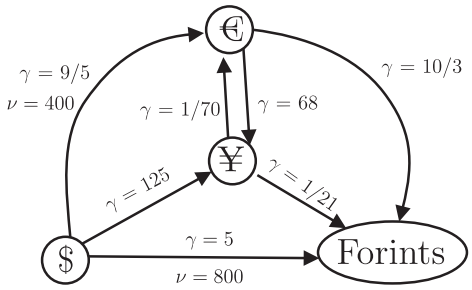


Figure 14: Currency conversion.

Problem 5.1. GENERALIZED CIRCULATION

INPUT: A symmetric directed graph $G = (V, A)$, i.e. $(i, j) \in A \Leftrightarrow (j, i) \in A$;
Designated sink $t \in V$;
Integer capacities u_{ij} for all $(i, j) \in A$;
Gains γ_{ij} (ratios of integers): $\gamma_{ji} = 1/\gamma_{ij}$ for all $(i, j) \in A$;
All input integers are bounded by B.
OUTPUT: Find a circulation g that maximizes the excess e_t^g, denoted by $|g|$, and also called the *value* of the flow.

The model is a generalization of the maximum flow problem in the sense that if every gain has value 1, then the generalized network flow model becomes the maximum flow problem. The following definitions will help us clarify what we mean by excess of a flow in the context of the generalized circulation problem. A flow $g : A \to \mathbb{R}$ is a *generalized pseudoflow* if it satisfies

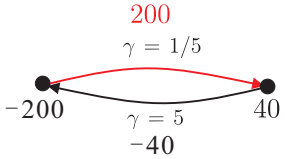
- Capacity constraints: $g_{ij} \leq u_{ij}$ for all $(i, j) \in A$;
- Anti-symmetry condition $g_{ij} = -\gamma_{ji} g_{ji}$, i.e., $g_{ij}\gamma_{ij} = -g_{ji}$, for all $(i, j) \in A$.

The *residual excess* of a flow g at a vertex i is given by

$$e_i^g = -\sum_{j:(i,j)\in A} g_{ij} = \sum_{j:(j,i)\in A} \gamma_{ji} g_{ji}$$

If $e_i^g > 0$ we say we have an *excess* at vertex i. If $e_i^g < 0$ we say we have a *deficit* at vertex i.

For example, if the flow on the upper arc of the figure below is 200 units, then the flow on the lower reverse arc is -40 by antisymmetry. Note that the definition of excess, although somewhat unintuitive, is capturing the notion of the total amount of flow entering a vertex minus that leaving the vertex.



A *flow* g is a pseudoflow such that $e_i^g \geq 0$ for all $i \in V$. A *circulation* is a flow such that $e_i^g = 0$ for all $i \in V \setminus \{t\}$. Thus our goal is to find a circulation that maximizes the excess at the sink vertex t. Given a pseudoflow g in G, we define the *residual graph* $G_g = (V, A_g)$ by $A_g = \{(i, j) \in A : g_{ij} < u_{ij}\}$ and $u_{ij}^g = u_{ij} - g_{ij}$ for all $(i, j) \in A_g$.

For a path or a cycle P, we define the *gain* of P as $\gamma(P) = \prod_{(i,j)\in P} \gamma_{ij}$. Let C be a cycle. If $\gamma(C) > 1$, then C is a *flow-generating cycle*. If $\gamma(C) < 1$, then C is a *flow-absorbing cycle*. A *labelling function*

$\mu : V \to \mathbb{R}_+ \cup \{\infty\}$ such that $\mu_t = 1$, represents the change in units of measurement of a vertex. Namely $\mu_i =$ (new units)/(old units).

For example if we wanted to perform the currency conversion (from Figure 14) in cents instead of dollars, we would need $\mu_\$ = 100$. The conversion rates involving the relabelled vertex would be affected (5 forints per dollar becomes .05 forints per cent), and also the capacity of the edges incident to the vertex (800 would become 80000 on the lowest edge, for instance).

In general we would have to perform the following changes for the gains, capacities, and excess at each relabelled vertex:

$$u^\mu_{ij} = u_{ij}\mu_i,\ \gamma^\mu_{ij} = \gamma_{ij} \cdot \mu_j/\mu_i \text{ and } e^\mu_i = e_i\mu_i.$$

If we already have some pseudoflow g, note that we also have to relabel it:

$$g^\mu_{ij} = g_{ij}\mu_i.^{12}$$

Note that relabelling does not change the value of $|g|$ since $\mu_t = 1$ by definition. Thus

$$|g^\mu| = |g|. \tag{5.1}$$

The antisymmetry is also preserved, namely,

$$g^\mu_{ij} = -\gamma^\mu_{ji} g^\mu_{ji} \text{ if and only if } g_{ij} = -\gamma g_{ji}.$$

An *augmenting path* P in G_g is a path from a vertex with excess to the sink t. A *generalized augmenting path* (GAP) is a flow generating cycle in the residual graph G_g with a (possibly trivial) path from a vertex on cycle to the sink t. For each $(i,j) \in A$, we set its *length* to be $c_{ij} = -\log(\gamma_{ij})$. Note that

$$c_{ij} = -\log(\gamma_{ij}) = \log(1/\gamma_{ij}) = \log(\gamma_{ji}) = -c_{ji}.$$

For path or cycle P, its length equals

$$\sum_{(i,j)\in P} c_{ij} = -\log(\gamma(P)). \tag{5.2}$$

Negative-length cycles are equivalent to having flow generating cycles, since

$$\sum_{(i,j)\in C} c_{ij} < 0 \Leftrightarrow \log(\gamma(C)) > 0 \Leftrightarrow \gamma(C) > 1.$$

5.1 Optimality conditions

The algorithm to be discussed is built on the following optimality conditions for the generalized circulation problem.

Theorem 5.2. *The following are equivalent for a generalized circulation g:*

(i) g is optimal.

(ii) G_g has no generalized augmenting paths (no GAPs).

(iii) There exist labelling μ such that the relabelled gains satisfy $\gamma^\mu_{ij} \le 1$ for all $(i,j) \in A_g$.

[12] The implicit assumption implies that a feasibel circulation is still feasible after relabelling.

Proof. $(\neg(ii) \Rightarrow \neg(i))$. Assume that a GAP exists in G_g. Let C be the flow generating cycle, and P be the path from a vertex i on the cycle to the sink t. Now consider a flow of δ coming into i (ignore for now the source of this flow). If we push this flow around the cycle C we end up back at i with a flow of $\delta \cdot \gamma(C)$. Since $\gamma(C) > 1$ we can pay back the original δ flow, and still remain with $\delta(\gamma(C) - 1) > 0$ amount of flow at i. Pushing forward this flow from i to t on the path P, we add an extra $\delta(\gamma(C) - 1)\gamma(P)$ flow at t. Set δ such that residual capacities (along C and P) are obeyed, and we get a circulation g' such that $|g'| > |g|$. Thus g was not optimal. Note that flow constraints are satisfied for every vertex and hence no vertices with excess are created.

$((ii) \Rightarrow (iii))$. Let S be the set of vertices that can reach t in G_g. We have no GAPs (by assumption) in S, thus there are no negative cost cycles in S for costs $c_{ij} = -\log \gamma_{ij}$. Set C_i to be the cost of shortest path P_i from i to t w.r.t. costs c, and $\mu_i = e^{-C_i} = \prod_{(k.l)\in P_i} \gamma_{kl}$. If $(i, j) \in A_g$ then, by definition of the C_i's we have that $C_i \leq c_{ij} + C_j$. This implies that

$$\mu_i = e^{-C_i} \geq e^{-c_{ij}-C_j} = \mu_j e^{-c_{ij}} = \gamma_{ij}\mu_j.$$

Thus $\gamma^\mu_{ij} = \gamma_{ij}\mu_j/\mu_i \leq 1$. By setting $\mu_i = 0$ for all $i \in V \setminus S$ we ensure that our labeling satisfies the conditions of (iii). To see this, note that by the definition of S, there are no arcs in A_g with $i \not\in S$ and $j \in S$. If $i \in S$ and $j \notin S$, then $\gamma^\mu_{ij} = \gamma_{ij} \cdot \mu_j/\mu_i = \gamma_{ij} \cdot 0/\mu_i = 0 \leq 1$. If $i, j \notin S$, then using the convention that $0/0 = 0$, we have $\gamma^\mu_{ij} = 0 \leq 1$.

$((iii) \Rightarrow (ii))$. Given labelling μ and circulation g, consider any other circulation $\tilde{g}$. Let us focus on an arc $(i, j) \in A$.

- If $g^\mu_{ij} < \tilde{g}^\mu_{ij}$, then $g^\mu_{ij} < u_{ij}$ as $\tilde{g}^\mu_{ij} \leq u_{ij}$. So $(i, j) \in A_f$, which implies $\gamma^\mu_{ij} \leq 1$ (by the assumption of (iii)).
- If $g^\mu_{ij} > \tilde{g}^\mu_{ij}$, then $-\gamma^\mu_{ji} g^\mu_{ji} > -\gamma^\mu_{ji}\tilde{g}^\mu_{ji}$ (by antisymmetry) $\Rightarrow g^\mu_{ji} < \tilde{g}^\mu_{ji} \Rightarrow (j, i) \in A_g \Rightarrow \gamma^\mu_{ji} \leq 1 \Rightarrow \gamma^\mu_{ij} \geq 1$.

So for any arc $(i, j) \in A$, $(\gamma^\mu_{ij} - 1)(g^\mu_{ij} - \tilde{g}^\mu_{ij}) \geq 0$. Summing over all arcs in A, we obtain $\sum_{(i,j)\in A}(\gamma^\mu_{ij} - 1)(g^\mu_{ij} - \tilde{g}^\mu_{ij}) \geq 0$. We can rewrite this as $\sum_{(i,j)\in A} \gamma^\mu_{ij}(g^\mu_{ij} - \tilde{g}^\mu_{ij}) - \sum_{(i,j)\in A}(g^\mu_{ij} - \tilde{g}^\mu_{ij}) \geq 0$. By antisymmetry $\gamma^\mu_{ij} g^\mu_{ij} = -g^\mu_{ji}$ and $-\gamma^\mu_{ij}\tilde{g}^\mu_{ij} = \tilde{g}^\mu_{ji}$, we have

$$\sum_{(i,j)\in A} (\tilde{g}^\mu_{ji} - g^\mu_{ji}) - \sum_{(i,j)\in A} (g^\mu_{ij} - \tilde{g}^\mu_{ij}) \geq 0.$$

The symmetry of G implies

$$\begin{aligned}
& \sum_{(i,j)\in A} (\tilde{g}^\mu_{ij} - g^\mu_{ij}) - \sum_{(i,j)\in A} (g^\mu_{ij} - \tilde{g}^\mu_{ij}) \geq 0 \\
\Rightarrow \quad & \sum_{(i,j)\in A} \tilde{g}^\mu_{ij} \geq \sum_{(i,j)\in A} g^\mu_{ij} \\
\Rightarrow \quad & \sum_{i\in V} \sum_{j:(i,j)\in A} \tilde{g}^\mu_{ij} \geq \sum_{i\in V} \sum_{j:(i,j)\in A} g^\mu_{ij}
\end{aligned}$$

Since in any circulation $\bar{g}$, excess $e^{\bar{g}}_i = -\sum_{j:(i,j)\in A} \bar{g}_{ij} = 0$ for all $i \in V \setminus \{t\}$, we deduce that $\sum_{j:(i,j)\in A} \bar{g}^\mu_{ij} = \sum_{j:(i,j)\in A} \mu_i \bar{g}_{ij} = 0$ for all $i \in V \setminus \{t\}$. Then we can reduce in the previous expression, for both g and $\tilde{g}$, all arcs that are not leaving t. We obtain

$$\sum_{j:(t,j)\in A} \tilde{g}^\mu_{tj} \geq \sum_{j:(t,j)\in A} g^\mu_{tj} \Rightarrow |g^\mu| \equiv -\sum_{j:(t,j)\in A} g^\mu_{tj} \geq -\sum_{j:(t,j)\in A} \tilde{g}^\mu_{tj} \equiv |\tilde{g}^\mu|.$$

Recalling (5.1), the last expression gives $|g| \geq |\tilde{g}|$, and it is true for any arbitrary circulation $\tilde{g}$. Thus we can conclude that g is optimal; so (i) holds. □

5.2 Truemper's algorithm

We will look at a primal-dual style algorithm which decouples GAPs by (1) pushing flows along flow generating cycles to create excesses at vertices, and (2) pushing these excesses to the sink.

IDEA: Look at *costs* $c_{ij} = -\log \gamma_{ij}, (i,j) \in A_g$. Then flow generating cycles are equivalent to negative cost cycles with respect to c.

Exercise 5.1. *By using minimum mean cost cycle cancelling, we can cancel all flow generating cycles (i.e., negative cost cycles) in $O(m^2n^3log(nB))$ time.*[13]

Definition 5.3. *A labelling μ is* canonical *if $\mu_i = \max_{i\text{-}t \text{ path } P} \gamma(P)$ for all $i \in V$, where $\gamma(P) = \prod_{(i,j)\in P} \gamma_{ij}$.*

Recalling (5.2), we can compute the canonical labels by finding a minimum cost path from each vertex to the sink t, since all negative cycles w.r.t. c have been cancelled.

Let μ be a canonical labelling. Then for any $(i,j) \in A_g$, Definition 5.3 implies $\mu_i \geq \mu_j \gamma_{ij}$, giving $\gamma^\mu_{ij} = \gamma_{ij}\mu_j/\mu_i \leq 1$. Note that if an arc (i,j) is on the shortest path from i to t, then we have its relabelled gain $\gamma^\mu_{ij} = 1$.

Algorithm 5.4 (Truemper, 1977)**.**

$g \leftarrow 0$
`Cancel all flow generating cycles`
while $\exists\ i \in V$ `with` $e^g_i > 0$ `that can reach` t `in` G_g **do**
  `Compute canonical labels` μ
  `Compute a maximum flow` f `that pushes flow from` $\{i \in V : e^g_i > 0\}$
    `in graph` $(V, \{(i,i) \in A_g : \gamma^\mu_{ij} = 1\}$ `with capacity` $(u^g)^\mu$
  $g_{ij} \leftarrow g_{ij} + f_{ij}/\mu_i$ `for all` $(i,j) \in A$ [14]
end-while

By the discussion above, the algorithm finds maximum flows from the vertices with excess along the highest-gain paths to the sink. Note that we will not go into how the algorithm handles situations where excesses cannot reach the sink; we assume that we can "undo" the creation of any excess by pushing flow back along the flow-generating cycle that created it.

Lemma 5.1. *No flow generating cycles are created by augmenting g^μ by the maximum flow f.*

Proof. All arcs (i,j) initially have $\gamma^\mu_{ij} \leq 1$. The maximum flow creates only arcs (i,j) with $\gamma^\mu_{ij} = 1$. So reverse arcs (j,i) that appear in the residual graph have $\gamma^\mu_{ji} = 1/\gamma^\mu_{ij} = 1$. □

Lemma 5.2. *The number of iterations of the while-loop of Algorithm 5.4 is no more than the number of different possible gains of paths.*

Proof. After augmentation, there exists no augmenting path P from a vertex with excess to the sink with $\gamma^\mu(P) = 1$. So $\gamma^\mu(P) < 1$ for any path a vertex with excess to the sink in the new residual graph. Let μ_i be the old canonical label for some vertex i with excess, and let its new canonical label be $\mu' = \gamma(P)$ for some path P. Then

$$\frac{\mu'_i}{\mu_i} = \frac{\gamma(P)}{\mu_i} = \frac{\mu_t}{\mu_i} \prod_{(k,l)\in P} \frac{\gamma_{kl}\mu_l}{\mu_l} = \prod_{(k,l)\in P} \frac{\gamma_{kl}\mu_l}{\mu_k} = \gamma^\mu(P) < 1 \Rightarrow \mu'_i < \mu_i.$$

Now, since the canonical label of vertex i is equal to the gain of some path, the fact that the canonical label of i is strictly decreasing in each iteration implies that there can be no more iterations than the number of different gains of paths. □

[13] An $O(mn^2 \log n \log B)$ time algorithm for flow generating cycle cancelling can be found in [12]
[14] This is equivalent to $g^\mu_{ij} \leftarrow g^\mu_{ij} + f_{ij}$ for all $(i,j) \in A$.

5.3 Gain scalling

Given Lemma 5.2, we just need to make sure that the number of different possible gains is polynomially bounded. In general, though, this is not the case. However, we can force the desired condition by modifying the gains so that there are only a polynomial number of different gains of paths by rounding the reduced gains as follows. Let $b = (1+\epsilon)^{1/n}$. Then for $\gamma_{ij} \leq 1$ define

$$\bar{\gamma}_{ij} = b^{\lfloor \log_b \gamma_{ij} \rfloor} \text{ and } \bar{\gamma}_{ji} = 1/\bar{\gamma}_{ij}.$$

Note that rounding down is consistent for both $\bar{\gamma}_{ij}$ and $\bar{\gamma}_{ji}$ since either $\gamma_{ij} = 1$ (which then implies that $\bar{\gamma}_{ij} = \bar{\gamma}_{ji} = b^0 = 1$) or only one of γ_{ij}^{μ} and γ_{ji} is greater than 1.

How many different gain values of paths are now possible? We can bound the gain of a path P by $B^{-n} \leq \bar{\gamma}(P) \leq B^n$. Given that all gains are powers of b, then only

$$O(\log_b B^{2n}) = O(n \log_b B) = O(n^2 \log_{1+\epsilon} B) \tag{5.3}$$

paths with different gains are possible. Thus, if we let H denote a network with gains $\bar{\gamma}$, then we may use Truemper's algorithm (Algorithm 5.4) to find an optimal flow h in H in polynomial time. To obtain an approximate solution to the original generalized flow problem, we interpret h in G as follows:

$$g_{ij} = \begin{cases} h_{ij} & \text{if } h_{ij} \geq 0, \\ -\gamma_{ji} h_{ji} & \text{if } h_{ij} < 0. \end{cases}$$

A flow g is *ϵ-optimal* if $|g| \geq (1-\epsilon)|g^*|$ for an optimal flow g^*.

Theorem 5.5. *For an optimal flow h in H, its interpretation in G is ϵ-optimal.*

Proof. Let g^* be the optimal flow in G. For each path P pushing δ units of excess to the sink t gives $\gamma(P)\delta$ units at the sink. In H, the same path gives

$$\bar{\gamma}(P) \geq \delta \prod_{(i,j)\in P} b^{(\log_b \gamma_{ij})-1} = \delta\gamma(P)/b^{|p|} \geq \delta\gamma(P)/b^n = \delta\gamma(P)/(1+\epsilon) > \gamma(P)(1-\epsilon)\delta$$

units of flow at the sink. Thus, the total flow pushed to the sink in H by g is $\sum_P \bar{\gamma}(P)\delta_P > \sum_P \gamma(P)\delta_P(1-\epsilon) = (1-\epsilon)|g^*|$. So the optimal flow h must have value greater than $(1-\epsilon)|g^*|$ in the network H. Since the gains in G are only no less than those in H, the interpretation of h in G will only have larger value, and thus is at least $(1-\epsilon)|g^*|$. □

This gives a polynomial time ϵ-optimal approximation algorithm for the generalized flow problem. In other words, this algorithm achieves a $1/(1-\epsilon)$-approximation.

Procedure 5.6 (Rounded Truemper(G, ϵ))**.**

`Round down gains using` $b = (1+\epsilon)^{1/n}$ `to get graph` H
$h \leftarrow$ `Truemper`(H)
`Return interpretation of` h `in` G

Let CC denote the time complexity for cycle cancelling in Truemper's algorithm. i.e., Algorithm 5.4 (see Exercise 5.1). It follows follows from (5.3) that

Lemma 5.3. *Procedure 5.6 finds an ϵ-optimal flow of G in $O(CC + (n^2 \log_{1+\epsilon} B)MP)$ time.* □

5.4 Error scaling

Lemma 5.4. *Suppose we have a flow g and labels μ such that $\gamma_{ij}^{\mu} \leq 1$ for all $(i,j) \in A_g$. If $|g^*| - |g| < B^{-2m}$ for an optimal flow g^*, then we can compute the optimal flow by one maximum flow computation.*

Proof. Let D be the product of all the denominators of input gains. Then $D \leq B^m$. Each μ_i must be an integral multiple of $1/D$. This in turn implies that u_{ij}^{μ} is an integral multiple of $1/D$. Let

$$h_{ij}^{\mu} = \begin{cases} 0 & \text{if } \gamma_{ij}^{\mu} = 1, \\ g_{ij}^{\mu} & \text{otherwise.} \end{cases}$$

Note that

$$\begin{aligned} e_i^{h,\mu} &= -\sum_{j:(i,j)\in A} h_{ij}^{\mu} \\ &= -\sum_{j:(i,j)\in A,\gamma_{ij}^{\mu}\neq 1} h_{ij}^{\mu} \\ &= -\sum_{j:(i,j)\in A,\gamma_{ij}^{\mu}>1} u_{ij}^{\mu} + \sum_{j:(i,j)\in A,\gamma_{ij}^{\mu}<1} \gamma_{ji}^{\mu} g_{ji}^{\mu} \\ &= -\sum_{j:(i,j)\in A,\gamma_{ij}^{\mu}>1} u_{ij}^{\mu} + \sum_{j:(i,j)\in A,\gamma_{ji}^{\mu}>1} \gamma_{ji}^{\mu} g_{ji}^{\mu} \\ &= -\sum_{j:(i,j)\in A,\gamma_{ij}^{\mu}>1} u_{ij}^{\mu} + \sum_{j:(i,j)\in A,\gamma_{ji}^{\mu}>1} \gamma_{ji}^{\mu} u_{ji}^{\mu} \end{aligned}$$

is an integral multiple of $1/D^2$. This implies that $|h^{\mu}| = e_t^{h,\mu}$ is also an integral multiple of $1/D^2$.

Now we set up the network K as is shown in Figure 15. Add a dummy source vertex s and a sink vertex t', and all the arcs with $\gamma_{ij}^{\mu} = 1$. We compute flow f from s, which satisfies all the deficits ($e_i^{h,\mu} < 0$) and maximizes the flow into t. We know that a flow satisfying all deficits exists, since g^{μ} satisfies them. By the integrality property of the maximum flow problem, we know that the flow f^{μ} has value that must be an integral multiple of $1/D^2$, since all the capacities, supplies, and demands are multiples of this factor.

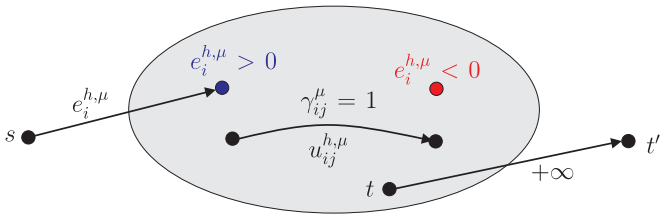


Figure 15: Network K.

By a similar argument, we can show that $|g^*|$ must be an integral multiple of $1/D$. Now we have the following inequalities

$$|g^*| \geq |h^{\mu}| + |f^{\mu}| \geq |g^{\mu}| = |g| > |g^*| - B^{-2m},$$

where $|h^{\mu}|$, $|f^{\mu}|$ and $|g^*|$ are all integral multiples of $1/D^2 \geq B^{-2m}$. This enforces $|h^{\mu}| + |f^{\mu}| = |g^*|$, proving the theorem. □

Corollary 5.7. *Given a B^{-4m}-optimal flow with no flow generating cycles, we can compute an optimal flow with one maximum flow computation.*

Proof. Let g be a B^{-4m}-optimal flow with no flow generating cycles. Since there are no flow-generating cycles, we can compute the canonical labels μ, which thus satisfies $\gamma_{ij}^{\mu} \leq 1$ for all $(i,j) \in A_g$. Furthermore, for any optimal flow g^*, the flow g satisfies $|g| \geq (1 - B^{4m})|g^*|$. On the other hand, $|g^*| \leq mU \leq mB \leq B^m$, which implies $|g^*| - |g| \leq |g^*|/B^{4m} \leq B^{-3m} < B^{-2m}$. This enables us to apply Lemma 5.4. □

Setting $\epsilon = B^{-4m}$, we can obtain a B^{-4m}-approximation using the Truemper algorithm (Algorithm 5.4) with gain scaling. Unfortunately, this method is not polynomial in B, since $\log_{1+\epsilon} B$ for $\epsilon = B^{-4m}$ is $O(B^{4m} \log B)$. This is exponential in the size of the input. It is possible, however, to modify the Truemper gain scaling approach to derive an actual polynomial time algorithm for computing exact generalized flows. The basic idea is that we will invoke the Truemper gain scaling algorithm to iteratively obtain half of the remaining flow in the residual graph by setting $\epsilon = 1/2$. Then only $\log_2 B^{4m}$ iterations of the Truemper gain scaling algorithm are needed to get a B^{-4m}-optimal flow. We introduce the algorithm below [33].

Algorithm 5.8 (Tardos, Vayne, 1998)**.**

$g \leftarrow 0$
for $i = 1$ `to` $\log_2 B^{4m}$ **do**
  $f \leftarrow$ `Rounded Truemper`$(G_g, 1/2)$
  $g \leftarrow g + f$
end-for

Theorem 5.9. *Algorithm 5.8 computes a B^{-4m}-optimal flow in $O((m \log B)(CC + (n^2 \log B)MF))$ time.*

Proof. The initial zero flow is 1-optimal. Each iteration finds a (1/2)-optimal flow in G_g, so the i-th iteration is 2^{-i}-optimal. In $\log_2 B^{4m}$ iterations, the flow is B^{-4m}-optimal. The result is instant from Lemma 5.3. □

6 Matroids

Matroids form a combinatorial abstraction which generalizes spanning trees and a host of other combinatorial structures. Let us recall Kruskal's algorithm for finding a maximum wight spanning tree.

Algorithm 6.1 (Kruskal, 1956)**.**
INPUT: *Connected graph $G = (V, E)$ with edge weight $w \in \mathbb{R}^E$.*
OUTPUT: *A spanning tree T of G.*

$T \leftarrow \emptyset$
while $\exists\ e \in E \setminus T$ `such that` $T \cup \{e\}$ `is a forest` **do**
  `Choose such` e `with` $w(e)$ `maximum`
  $T \leftarrow T \cup \{e\}$
end-while
`Output` T

Exercise 6.1. *Given an algorithm to find a maximum weight forest of a graph, where edge weights might be positive, negative or zero.*

6.1 Independent systems and matroids

An *independence system* is a pair $(S, \mathcal{I})$, where S is an arbitrary finite set, called *ground set* or *universe set* and $\mathcal{I} \subseteq 2^S$ is a family (collection) of subsets of S, whose members are called the *independent sets* of S and satisfy the following properties:

(i) $\emptyset \in \mathcal{I}$.

(ii) If $X \in \mathcal{I}$ and $Y \subseteq X$, then $Y \in \mathcal{I}$.

Maximal independent sets (with resect to inclusion) are called *bases*. In particular, for $X \subset S$, any maximal independent subset of X is called a *basis* of X. Furthermore, there is a weight function $w : 2^S \leftarrow \mathbb{R}$, which is *modular*, i.e., for any $X \subseteq S$ we have $w(X) = \sum_{x \in X} w(x)$, where we use $w(x)$ as a short hand of $w(\{x\})$. For any $X \subseteq S$, the *rank* of X is defined as $\mathsf{rank}(X) = \max\{|Y| : Y \subseteq X \text{ and } Y \in \mathcal{I}\}$, i.e., the maximum cardinality of a basis of X.

There are two optimization problems associated with independence systems, a minimization for finding a minimum weight basis of the ground set and a maximization problem for finding a maximum weight independent set.

In general, $(S, \mathcal{I})$ is not given explicitly. Instead, we assume that we have an *independence oracle* that decides if a given subset $X \subseteq S$ is a member of the independence system, i.e., if $X \in \mathcal{I}$. Also notice that the maximization problem asks for any member of the system $(S, \mathcal{I})$, while the minimization problem asks for a basis. Three examples of combinatorial optimization problems that can be expressed in terms of independence systems are: maximum weight forest problem, minimum spanning tree problem, and maximum weight matching.

An independent system $\mathcal{M} = (S, \mathcal{I})$ is a *matroid* if it satisfies the following property: If $X, Y \in \mathcal{I}$ and $|Y| > |X|$, then there exists $y \in Y \setminus X$ such that $X \cup \{x\} \in \mathcal{I}$. A maximal set $X \in \mathcal{I}$ is called a *basis* of matroid $\mathcal{M}$.

Exercise 6.2. *Prove that all bases of a matroid have the the same cardinality.*

Let $(S, \mathcal{I})$ be an independence system. For $X \subseteq S$ we define the *lower rank* of X as the minimum cardinality of a basis of X, i.e.,

$$\mathsf{lrank}(X) = \min\{|Y| : Y \subseteq X,\ Y \in \mathcal{I} \text{ and } Y \cup \{X\} \notin \mathcal{I} \text{ for all } x \in X \setminus Y\}.$$

The *rank quotient* of $(S, \mathcal{I})$ is defined as

$$q(S, \mathcal{I}) = \min_{X \subseteq S} \frac{\mathsf{lrank}(X)}{\mathsf{rank}(X)}.$$

Lemma 6.1. *Let $(S, \mathcal{I})$ be an independence system. Then $q(S, \mathcal{I}) \leq 1$. Furthermore, $(S, \mathcal{I})$ is a matroid if and only if $q(S, \mathcal{I}) = 1$.*

Exercise 6.3. *Show the following five examples are all matroids.*

(i) Given an undirected graph $G = (V, E)$, the *graphic matroid* of G is defined as $\mathcal{M}_G = (E, \mathcal{I}_G)$, where $\mathcal{I}_G = \{F \subseteq E : F \text{ contains no cycles}\}$.

(ii) Given a set S and a nonnegative integer k, the *uniform matroid of rank k* is defined as $\mathcal{M}_S^k = (S, \mathcal{I}^k)$, where $\mathcal{I}^k = \{T \subseteq S : |T| \leq k\}$.

(iii) Given a set $S = \uplus_{i=1}^k S_i$, where $\uplus$ is the operation of disjoint union, and nonnegative integers $n_1, \ldots, n_k$, the *partition matroid* is defined as $\mathcal{M} = (S, \mathcal{I})$, where $\mathcal{I} = \{\uplus_{i=1}^k T_i \subseteq S_i : |T_i| \leq n_i\}$.

(iv) Let A be an $m \times n$ matrix and $S = \{1, \ldots, , n\}$. For any i with $1 \leq i \leq n$, let A_i denote the ith column of A. The *linear matroid over matrix A* is defined as $\mathcal{M}_A = (S, \mathcal{I}_A)$, where $\mathcal{I}_A = \{T \subseteq S : A_i, i \in T$ are linearly independent$\}$.

(v) Let $\mathcal{M} = (S, \mathcal{I})$ be a matroid and $T \subseteq S$. Then the matroid restriction of $\mathcal{M}$ to the set T is the matroid $\mathcal{M}_T = (T, \mathcal{I}_T)$, where $\mathcal{I}_T = \{R : R \in \mathcal{I} \text{ and } R \subseteq T\}$.

6.2 Greedy algorithm

Let $(S, \mathcal{I})$ be an independence system with weight function $w : S \leftarrow \mathbb{R}_+$ and consider the problem for finding a maximum independent set of $\mathcal{I}$. Notice that the restriction to nonnegative weights is without loss of generality, because negative weight never appear in an optimum solution. Again, we assume that we have access to an independence oracle. This allows us to formulate the following greedy algorithm.

Algorithm 6.2 (Greedy)**.**
INPUT: *$(S, \mathcal{I})$ and $w \in \mathbb{R}_+^S$.*
OUTPUT: *A set $J \in \mathcal{I}$ such that $\sum_{e \in J} w_e$ is maximized.*

$J \leftarrow \emptyset$
while $\exists\ e \in S \setminus J$ `such that` $J \cup \{e\} \in \mathcal{I}$ **do**
  `Choose such` e `with` $w(e)$ `maximum`
  $J \leftarrow J \cup \{e\}$
end-while
`Output` J

The following theorem states that the approximation ratio of the greedy Algorithm 6.2 is at least the rank quotient (and of course at most one). Notice that the algorithm runs in polynomial time if the independence oracle is polynomial time (which is often the case in the applications).

Theorem 6.3. *Given any instance $I = (S, \mathcal{I}, w)$ for maximum weight independent set, let* GREEDY(I) *denote the weight of the solution returned by Algorithm 6.2, and let* OPT(I) *denote the optimal value of instance I. Then $q(S, \mathcal{I}) \leq \frac{\text{GREEDY}(I)}{\text{OPT}(I)} \leq 1$.*

Independent systems for which the greedy Algorithm 6.2 always returns an optimal solution are called "matroids".

Theorem 6.4. *Let $(S, \mathcal{I})$ be an independence system. Then the greedy Algorithm 6.2 finds a maximum weight independent set for every $w \in \mathbb{R}^S$ if and only if $(S, \mathcal{I})$ is a matroid.*

Matroids are plentiful enough that the greedy Algorithm 6.2 has a number of applications. In addition, other optimization problems on matroids (not just the optimal independent set problem) can be solved efficiently.

6.3 Matroid intersection

Given two independence systems $(S, \mathcal{I}_1)$ and $(S, \mathcal{I}_2)$, we define their intersection by $(S, \mathcal{I}_1 \cap \mathcal{I}_2)$. The intersection of a finite number of independence systems is defined analogously.

Lemma 6.2. *Any independence system is the intersection of a finite number of matroids.*

Thus, the intersection of matroids is not a matroid in general. Hence we can not expect that a greedy algorithm finds an optimum common independent set. However, the following result implies a lower bound on the approximation ratio of the greedy Algorithm 6.2.

Lemma 6.3. *If $(S, \mathcal{I})$ is the intersection of p matroids, then $q(S, \mathcal{I}) \geq 1/p$.*

As we have seen in Theorem 6.3 that the lower bound is sharp, the greedy Algorithm 6.2 can work arbitrarily bad when used for optimizing over arbitrary independence systems, i.e., the intersection of p matroids. However, for optimizing over the intersection of two matroids, there is a polynomial time algorithm (see Algorithm 6.8 below), provided that the independence oracle is polynomial. The important property of the algorithm is that it generalizes the concept of an augmenting path. The intersection problem becomes *NP*-hard for more than two matroids.

Problem 6.5. MAXIMUM MATROID INTERSECTION

INPUT: Matroids $(S, \mathcal{I}_1)$ and $(S, \mathcal{I}_2)$ with independence oracles.
GOAL: Find $X \in \mathcal{I}_1 \cap \mathcal{I}_2$ such that $|X|$ is maximized.

An important special case is that the problem of maximum cost matching in bipartite graphs can be formulated as matroid intersection of two matroids. The remainder of this section is devoted to the development of Edmonds' algorithm for solving Problem 6.5 in polynomial time. We start our discussion with basic facts on the submodularity of the rank function of a matroids.

Exercise 6.4. *Let S be a finite set and $r : 2^S \leftarrow \mathbb{N}$. Then the following are equivalent.*

(i) r is the rank function of a matroid $(S, \mathcal{I})$ (and $\mathcal{I} = \{X \subseteq S : r(X) = |X|\}$).

(ii) For all $X, Y \subseteq S$, it holds that (a) $r(X) \leq |X|$; (b) if $X \subseteq Y$ then $r(X) \leq r(Y)$; (c) $r(X \cup Y) + r(X \cap Y) \leq r(X) + r(Y)$.

(iii) $r(\emptyset) = 0$, and for all $X \subseteq S$ and $x, y \in S$ it holds that (a) $r(X) \leq r(X \cup \{x\}) \leq r(X) + 1$; (b) if $r(X \cup \{x\}) = r(X \cup \{y\}) = r(X)$, then $r(X \cup \{x, y\}) = r(X)$.

For any independence system $(S, \mathcal{I})$, the sets in $2^S \setminus \mathcal{I}$ are *dependent*. A minimal dependent set is called a *circuit*.

Exercise 6.5. *Let S be a finite set and $\mathcal{C} \subseteq 2^S$ is the set of circuits of an independence system $(S, \mathcal{I})$, where $\mathcal{I} = \{T \subseteq S :$ there is no $C \in \mathcal{C}$ with $C \subseteq T$, if and only if the following conditions hold:*

(i) $\emptyset \notin \mathcal{C}$;

(ii) For any $C, D \in \mathcal{C}$ it holds that if $C \subseteq D$ then $C = D$.

Theorem 6.6. *Let $(S, \mathcal{I})$ be an independence system. If for any $X \in \mathcal{I}$ and $x \in S$ the set $X \cup \{x\}$ contains at most p circuits, then $q(S, \mathcal{I}) \geq 1/p$.*

Theorem 6.7. *If $\mathcal{C}$ is the set of circuits of an independence system $(S, \mathcal{I})$, then the following statements are equivalent:*

(i) $(S, \mathcal{I})$ is a matroid.

(ii) For any $X \in \mathcal{I}$ and $x \in S$ it holds that $X \cup \{x\}$ contains at most one circuit.

(iii) For any distinct $C, D \in \mathcal{C}$ and $a \in C \cap D$, there exists a circuit contained in $(C \cup D) \setminus \{a\}$.

(iv) For any $C, D \in \mathcal{C}$, $a \in C \cap D$, $b \in C \setminus D$ it holds that b belongs to a circuit which is contained in $C \cup D \setminus \{a\}$.

For any $X \in \mathcal{I}$ and $x \in S$, let $C(X, x)$ denote the unique circuit in $X \cup \{x\}$ if $X \cup \{x\} \notin \mathcal{I}$, and $C(X, x) = \emptyset$ otherwise.

Lemma 6.4. *Let $(S, \mathcal{I})$ be a matroid and $X \in \mathcal{I}$. Let $x_1, \ldots, x_s \in X$ and $y_1, \ldots, y_s \notin X$ satisfying $x_j \in C(X, y_j)$ for all $1 \leq j \leq s$ and $x_i \notin C(X, y_j)$ for all $1 \leq i < j \leq s$. Then $(X \setminus \{x_1, \ldots, x_s\}) \cup \{y_1, \ldots, y_s\} \in \mathcal{I}$.*

The idea behind the Edmonds' algorithm is the following: Starting with $X = \emptyset$, we augment X by one element in each iteration. Since in general we cannot hope for an element x such that $x \in \mathcal{I}_1 \cap \mathcal{I}_2$, we shall look for "alternating paths". To make this convenient, we define an auxiliary graph. We apply the notion $C(X, x)$ to $(S, \mathcal{I}_i)$ and write $C_i(X, x)$ for $i = 1, 2$.

Given a set $X \in \mathcal{I}_1 \cap \mathcal{I}_2$, we define a directed *auxiliary graph* $G_X = (S, A_X^1 \cup A_X^2)$ by $A_X^1 = \{(x, y) : y \in S \setminus X, x \in C_1(X, y) \setminus \{y\}\}$ and $A_X^2 = \{(y, x) : y \in S \setminus X, x \in C_2(X, y) \setminus \{y\}\}$. We set $B_X^i = \{y \in S \setminus X : X \cup \{y\} \in \mathcal{I}_i$ for $i = 1, 2$, and look for a shortest path from B_X^1 to B_X^2. Such a path will enable us to augment the set X. In particular, if $B_X^1 \cap B_X^2 \neq \emptyset$, we have a path of length zero and we can argument X by any element in $B_X^1 \cap B_X^2$.

Lemma 6.5. *Let $X \in \mathcal{I}_1 \cup \mathcal{I}_2$. Let $y_0, x_1, y_1, \ldots, x_s, y_s$ be the vertices of a shortest y_0-y_s path in G_X (in this order), with $y_0 \in B^1_X$ and $y_s \in B^2_Z$. Then $(X \cup \{y_0, \ldots, y_s\}) \setminus \{x_1, \ldots, x_s\} \in \mathcal{I}_1 \cap \mathcal{I}_2$.*

Algorithm 6.8 (Edmonds)**.**
INPUT: *Matroids $(S, \mathcal{I}_1)$ and $(S, \mathcal{I}_2)$ with independence oracles.*
OUTPUT: *An independent set $X \in \mathcal{I}_1 \cap \mathcal{I}_2$*

$X \leftarrow \emptyset$
repeat
  Compute G_X, B^1_X and B^2_X
  if $\exists$ a B^1_X-B^2_X path P in G_X
    then Find a shortest B^1_X-B^2_X path P in G_X
      $X \leftarrow X \Delta P$
until there is no B^1_X-B^2_X path in G_X
Output X

Theorem 6.9. *Algorithm 6.8 correctly solves matroid intersection Problem 6.5 in $O(|S|^3\tau)$, where τ is the maximum complexity of the two independence oracles.*

References

[1] Richard A Brualdi. *Introductory combinatorics*. Prentice-Hall, 1992.

[2] Peter J Cameron. *Combinatorics: topics, techniques, algorithms*. Cambridge University Press, 1994.

[3] W.J. Cook, W.H. Cunningham, W.R. Pulleyblank, and A. Schrijver. *Combinatorial Optimization*. Wiley, New York, 1998.

[4] Thomas H. Cormen, Charles Eric Leiserson, Ronald L Rivest, and Clifford Stein. *Introduction to algorithms*, volume 6. MIT press Cambridge, 2001.

[5] G Dantzig and Delbert Ray Fulkerson. On the max flow min cut theorem of networks. *Linear inequalities and related systems, Annals of Mathematics Studies*, 38:225–231, 1956.

[6] Nikhil R Devanur, Christos H Papadimitriou, Amin Saberi, and Vijay V Vazirani. Market equilibrium via a primal-dual-type algorithm. In *2013 IEEE 54th Annual Symposium on Foundations of Computer Science*, pages 389–389. IEEE Computer Society, 2002.

[7] E.A. Dinits. Algorithm for solution of a problem of maximum flow in a network with power estimation. *Soviet Mathematics Doklady*, 11:1277–1280, 1970.

[8] J. Edmonds and R.M. Karp. Theoretical improvements in algorithmic efficiency for network flow problems. *Journal of the Association for Computing Machinery 19 (1972) 248C264.*, 19:248–264, 1972.

[9] L. R. Ford and D. R. Fulkerson. Maximal Flow through a Network. *Canadian Journal of Mathematics*, 8:399–404, 1956.

[10] Michael R. Garey and David S. Johnson. *Computers and Intractability; A Guide to the Theory of NP-Completeness*. W. H. Freeman & Co., New York, NY, USA, 1990.

[11] Naveen Garg and Jochen Koenemann. Faster and simpler algorithms for multicommodity flow and other fractional packing problems. *SIAM Journal on Computing*, 37(2):630–652, 2007.

[12] Andrew V Goldberg, Serge A Plotkin, and Éva Tardos. Combinatorial algorithms for the generalized circulation problem. *Mathematics of Operations Research*, 16(2):351–381, 1991.

[13] Andrew V Goldberg and Satish Rao. Beyond the flow decomposition barrier. *Journal of the ACM (JACM)*, 45(5):783–797, 1998.

[14] Andrew V Goldberg and Robert E Tarjan. Finding minimum-cost circulations by canceling negative cycles. *Journal of the ACM (JACM)*, 36(4):873–886, 1989.

[15] A.V. Goldberg. A new max-flow algorithm, technical report mit/lcs/tm-291. Technical report, Massachusetts Institute of Technology, Cambridge, Massachusetts, 1985.

[16] A.V. Goldberg and R.E. Tarjan. A new approach to the maximum-flow problem. *Journal of the Association for Computing Machinery*, 35:921–940, 1988.

[17] Jianxiu Hao and James B Orlin. A faster algorithm for finding the minimum cut in a graph. In *Proceedings of the third annual ACM-SIAM symposium on Discrete algorithms*, pages 165–174. Society for Industrial and Applied Mathematics, 1992.

[18] Selmer M Johnson. Generation of permutations by adjacent transposition. *Mathematics of computation*, 17(83):282–285, 1963.

[19] Morton Klein. A primal method for minimal cost flows with applications to the assignment and transportation problems. *Management Science*, 14(3):205–220, 1967.

[20] Donald L Kreher and Douglas R Stinson. *Combinatorial algorithms: generation, enumeration, and search*, volume 7. CRC press, 1998.

[21] Eugene L Lawler. *Combinatorial optimization: networks and matroids*. Courier Corporation, 2001.

[22] Jr L.R. Ford and D.R. Fulkerson. A simple algorithm for finding maximal network flows and an application to the hitchcock problem. *Canadian Journal of Mathematics*, 9:210–218, 1957.

[23] Jr L.R. Ford and D.R. Fulkerson. *Flows in Networks*. Princeton University Press, Princeton, New Jersey, 1962.

[24] Albert Nijenhuis and Herbert S Wilf. *Combinatorial algorithms: for computers and calculators*. Elsevier, 2014.

[25] Christos H Papadimitriou and Kenneth Steiglitz. *Combinatorial optimization: algorithms and complexity*. Courier Corporation, 1982.

[26] Paul Walton Purdom and Cynthia A Brown. *The analysis of algorithms*. Oxford University Press, USA, 1995.

[27] J.T. Robacker. *On Network Theory, Research Memorandum RM-1498*. The RAND Corporation, Santa Monica, California, May 1955.

[28] Carla Savage. A survey of combinatorial gray codes. *SIAM review*, 39(4):605–629, 1997.

[29] Alexander Schrijver. *Combinatorial optimization: polyhedra and efficiency*, volume 24. Springer, 2003.

[30] D.D. Sleator and R.E. Tarjan. A data structure for dynamic trees. In *Proceedings of the Thirteenth Annual ACM Symposium on Theory of Computing (13th STOC)*, pages 114–122, Milwaukee, Wisconsin, 1981. he Association for Computing Machinery.

[31] D.D.K. Sleator. *An $O(nm \log n)$ Algorithm for Maximum Network Flow.* Ph.d. thesis, report no. stan-cs-80-831, Department of Computer Science, Stanford University, Stanford, California, 1980.

[32] Éva Tardos. A strongly polynomial minimum cost circulation algorithm. *Combinatorica*, 5(3):247–255, 1985.

[33] Éva Tardos and Kevin D Wayne. Simple generalized maximum flow algorithms. In *Integer Programming and Combinatorial Optimization*, pages 310–324. Springer, 1998.

[34] HF Trotter. Algorithm 115: Perm. *Communications of the ACM*, 5(8):434–435, 1962.

[35] Jacobus Hendricus van Lint and Richard Michael Wilson. *A course in combinatorics.* Cambridge university press, 2001.