CS 573 Algorithms[①]

Sariel Har-Peled

May 29, 2013

Contents

Preface

This manuscript is a collection of class notes for the (semi-required graduate) course "473G Algorithms" taught in the University of Illinois, Urbana-Champaign, in the spring of 2006 and fall 2007.

There are without doubt errors and mistakes in the text and I would like to know about them. Please email me about any of them you find.

Class notes for algorithms class are as common as mushrooms after a rain. I have no plan of publishing them in any form except on the web. In particular, Jeff Erickson has class notes for 473 which are better written and cover some of the topics in this manuscript (but naturally, I prefer my exposition over his).

My reasons in writing the class notes are to (i) avoid the use of a (prohibitly expensive) book in this class, (ii) cover some topics in a way that deviates from the standard exposition, and (iii) have a clear description of the material covered. In particular, as far as I know, no book covers all the topics discussed here. Also, this manuscript is also available (on the web) in more convenient lecture notes form, where every lecture has its own chapter.

Most of the topics covered are core topics that I believe every graduate student in computer science should know about. This includes NP-Completeness, dynamic programming, approximation algorithms, randomized algorithms and linear programming. Other topics on the other hand are more additional topics which are nice to know about. This includes topics like network flow, minimum-cost network flow, and union-find. Nevertheless, I strongly believe that knowing all these topics is useful for carrying out any worthwhile research in any subfield of computer science.

Teaching such a class always involve choosing what not to cover. Some other topics that might be worthy of presentation include fast Fourier transform, the Perceptron algorithm, advanced data-structures, computational geometry, etc – the list goes on. Since this course is for general consumption, more theoretical topics were left out.

In any case, these class notes should be taken for what they are. A short (and sometime dense) tour of some key topics in theoretical computer science. The interested reader should seek other sources to pursue them further.

Acknowledgements

(No preface is complete without them.) I would like to thank the students in the class for their input, which helped in discovering numerous typos and errors in the manuscript.[1] Furthermore, the content was greatly effected by numerous insightful discussions with Jeff Erickson, Edgar Ramos and Chandra Chekuri.

Getting the source for this work

This work was written using LaTeX. Figures were drawn using either `xfig` (older figures) or ipe (newer figures). You can get the source code of these class notes from `http://valis.cs.uiuc.edu/~sariel/teach/05/b/`. See below for detailed copyright notice.

In any case, if you are using these class notes and find them useful, it would be nice if you send me an email.

Copyright

— Sariel Har-Peled
December 2007, Urbana, IL

[1] Any remaining errors exist therefore only because they failed to find them, and the reader is encouraged to contact them and complain about it. Naaa, just kidding.

Part I

NP Completeness

Chapter 1

NP Completeness I

> "Then you must begin a reading program immediately so that you man understand the crises of our age," Ignatius said solemnly. "Begin with the late Romans, including Boethius, of course. Then you should dip rather extensively into early Medieval. You may skip the Renaissance and the Enlightenment. That is mostly dangerous propaganda. Now, that I think about of it, you had better skip the Romantics and the Victorians, too. For the contemporary period, you should study some selected comic books."
>
> "You're fantastic."
>
> "I recommend Batman especially, for he tends to transcend the abysmal society in which he's found himself. His morality is rather rigid, also. I rather respect Batman."
>
> – A confederacy of Dunces, John Kennedy Toole.

1.1 Introduction

The question governing this course, would be the development of efficient algorithms. Hopefully, what is an algorithm is a well understood concept. But what is an *efficient* algorithm? A natural answer (but not the only one!) is an algorithm that runs quickly.

What do we mean by quickly? Well, we would like our algorithm to:

(A) Scale with input size. That is, it should be able to handle large and hopefully huge inputs.

(B) Low level implementation details should not matter, since they correspond to small improvements in performance. Since faster CPUs keep appearing it follows that such improvements would (usually) be taken care of by hardware.

(C) What we will really care about are asymptotic running time. Explicitly, polynomial time.

In our discussion, we will consider the input size to be n, and we would like to bound the overall running time by a function of n which is asymptotically as small as possible. An algorithm with better asymptotic running time would be considered to be *better*.

Example 1.1.1. It is illuminating to consider a concrete example. So assume we have an algorithm for a problem that needs to perform $c2^n$ operations to handle an input of size

Input size	n^2 ops	n^3 ops	n^4 ops	2^n ops	$n!$ ops
5	0 secs	0 secs	0 secs	0 secs	0 secs
20	0 secs	0 secs	0 secs	0 secs	16 mins
30	0 secs	0 secs	0 secs	0 secs	$3 \cdot 10^9$ years
50	0 secs	0 secs	0 secs	0 secs	never
60	0 secs	0 secs	0 secs	7 mins	never
70	0 secs	0 secs	0 secs	5 days	never
80	0 secs	0 secs	0 secs	15.3 years	never
90	0 secs	0 secs	0 secs	15,701 years	never
100	0 secs	0 secs	0 secs	10^7 years	never
8000	0 secs	0 secs	1 secs	never	never
16000	0 secs	0 secs	26 secs	never	never
32000	0 secs	0 secs	6 mins	never	never
64000	0 secs	0 secs	111 mins	never	never
200,000	0 secs	3 secs	7 days	never	never
2,000,000	0 secs	53 mins	202.943 years	never	never
10^8	4 secs	12.6839 years	10^9 years	never	never
10^9	6 mins	12683.9 years	10^{13} years	never	never

Figure 1.1: Running time as function of input size. Algorithms with exponential running times can handle only relatively small inputs. We assume here that the computer can do $2.5 \cdot 10^{15}$ operations per second, and the functions are the exact number of operations performed. Remember – never is a long time to wait for a computation to be completed.

n, where c is a small constant (say 10). Let assume that we have a CPU that can do 10^9 operations a second. (A somewhat conservative assumption, as currently [Jan 2006][①], the blue-gene supercomputer can do about $3 \cdot 10^{14}$ floating-point operations a second. Since this super computer has about $131,072$ CPUs, it is not something you would have on your desktop any time soon.) Since $2^{10} \approx 10^3$, you have that our (cheap) computer can solve in (roughly) 10 seconds a problem of size $n = 27$.

But what if we increase the problem size to $n = 54$? This would take our computer about 3 million years to solve. (It is better to just wait for faster computers to show up, and then try to solve the problem. Although there are good reasons to believe that the exponential growth in computer performance we saw in the last 40 years is about to end. Thus, unless a substantial breakthrough in computing happens, it might be that solving problems of size, say, $n = 100$ for this problem would forever be outside our reach.)

The situation dramatically change if we consider an algorithm with running time $10n^2$. Then, in one second our computer can handle input of size $n = 10^4$. Problem of size $n = 10^8$ can be solved in $10n^2/10^9 = 10^{17-9} = 10^8$ which is about 3 years of computing (but blue-gene might be able to solve it in less than 20 minutes!).

[①]But the recently announced Super Computer that would be completed in 2012 in Urbana, is naturally way faster. It supposedly would do 10^{15} operations a second (i.e., petaflop). Blue-gene probably can not sustain its theoretical speed stated above, which is only slightly slower.

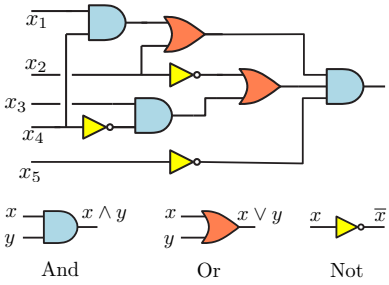
Thus, algorithms that have asymptotically a polynomial running time (i.e., the algorithms running time is bounded by $O(n^c)$ where c is a constant) are able to solve large instances of the input and can solve the problem even if the problem size increases dramatically.

Can we solve all problems in polynomial time? The answer to this question is unfortunately no. There are several synthetic examples of this, but it is believed that a large class of important problems can not be solved in polynomial time.

Circuit Satisfiability

Instance: A circuit C with m inputs
Question: Is there an input for C such that C returns true for it.

As a concrete example, consider the circuit depicted on the right.



Currently, all solutions known to Circuit Satisfiability require checking all possibilities, requiring (roughly) 2^m time. Which is exponential time and too slow to be useful in solving large instances of the problem.

This leads us to the most important open question in theoretical computer science:

Question 1.1.2. *Can one solve Circuit Satisfiability in polynomial time?*

The common belief is that Circuit Satisfiability can **NOT** be solved in polynomial time. Circuit Satisfiability has two interesting properties.

(A) Given a supposed positive solution, with a detailed assignment (i.e., proof): $x_1 \leftarrow 0$, $x_2 \leftarrow 1, ..., x_m \leftarrow 1$ one can verify in polynomial time if this assignment really satisfies C. This is done by computing what every gate in the circuit what its output is for this input. Thus, computing the output of C for its input. This requires evaluating the gates of C in the right order, and there are some technicalities involved, which we are ignoring. (But you should verify that you know how to write a program that does that efficiently.)
Intuitively, this is the difference in hardness between coming up with a proof (hard), and checking that a proof is correct (easy).

(B) It is a ***decision problem***. For a specific input an algorithm that solves this problem has to output either TRUE or FALSE.

1.2 Complexity classes

Definition 1.2.1 (P: Polynomial time). Let P denote is the class of all decision problems that can be solved in polynomial time in the size of the input.

Definition 1.2.2 (NP: Nondeterministic Polynomial time). Let NP be the class of all decision problems that can be verified in polynomial time. Namely, for an input of size n, if the solution to the given instance is true, one (i.e., an oracle) can provide you with a proof (of polynomial length!) that the answer is indeed TRUE for this instance. Furthermore, you can verify this proof in polynomial time in the length of the proof.

Clearly, if a decision problem can be solved in polynomial time, then it can be verified in polynomial time. Thus, $\mathrm{P} \subseteq \mathrm{NP}$.

Remark. The notation NP stands for Non-deterministic Polynomial. The name come from a formal definition of this class using Turing machines where the machines first guesses (i.e., the non-deterministic stage) the proof that the instance is TRUE, and then the algorithm verifies the proof.

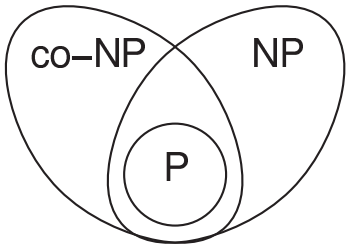


Figure 1.2: The relation between the different complexity classes P, NP, and co-NP.

Definition 1.2.3 (co-NP). The class co-NP is the opposite of NP – if the answer is FALSE, then there exists a short proof for this negative answer, and this proof can be verified in polynomial time.

See Figure 1.2 for the currently *believed* relationship between these classes (of course, as mentioned above, $\mathrm{P} \subseteq \mathrm{NP}$ and $\mathrm{P} \subseteq$ co-NP is easy to verify). Note, that it is quite possible that $\mathrm{P} = \mathrm{NP} =$ co-NP, although this would be extremely surprising.

Definition 1.2.4. A problem Π is NP-Hard, if being able to solve Π in polynomial time implies that $\mathrm{P} = \mathrm{NP}$.

Question 1.2.5. *Are there any problems which are* NP-Hard*?*

Intuitively, being NP-Hard implies that a problem is ridiculously hard. Conceptually, it would imply that proving and verifying are equally hard - which nobody that did CS 573 believes is true.

In particular, a problem which is NP-Hard is at least as hard as ALL the problems in NP, as such it is safe to assume, based on overwhelming evidence that it can not be solved in polynomial time.

Theorem 1.2.6 (Cook's Theorem). *Circuit Satisfiability is* NP-Hard.

Definition 1.2.7. A problem Π is NP-Complete (NPC in short) if it is both NP-Hard and in NP.

Clearly, Circuit Satisfiability is NP-Complete, since we can verify a positive solution in polynomial time in the size of the circuit,

By now, thousands of problems have been shown to be NP-Complete. It is extremely unlikely that any of them can be solved in polynomial time.

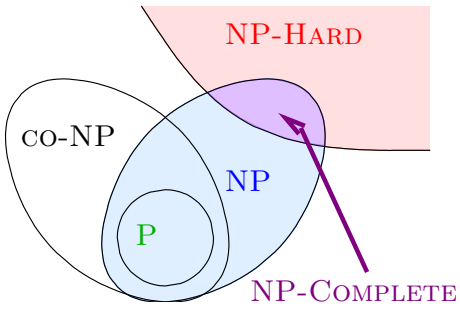


Figure 1.3: The relation between the complexity classes.

Definition 1.2.8. In the formula satisfiability problem, (a.k.a. SAT) we are given a formula, for example:

$$\left(a \vee b \vee c \vee \overline{d}\right) \iff \left((b \wedge \overline{c}) \vee \overline{(\overline{a} \Rightarrow d)} \vee (c \neq a \wedge b)\right)$$

and the question is whether we can find an assignment to the variables $a, b, c, \ldots$ such that the formula evaluates to TRUE.

It seems that SAT and Circuit Satisfiability are "similar" and as such both should be NP-Hard.

1.2.1 Reductions

Let A and B be two decision problems.

Given an input I for problem A, a *reduction* is a transformation of the input I into a new input I', such that

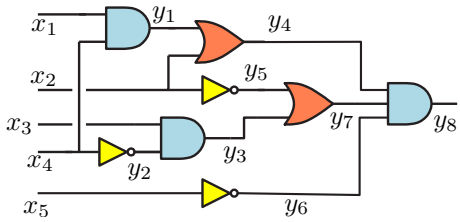
$$A(I) \ \ is \ \ \text{TRUE} \quad \Leftrightarrow \quad B(I') \ \ is \ \ \text{TRUE}.$$

Thus, one can solve A by first transforming and input I into an input I' of B, and solving $B(I')$.

This idea of using reductions is omnipresent, and used almost in any program you write.

Let $T : I \to I'$ be the input transformation that maps A into B. How fast is T? Well, for our nefarious purposes we need ***polynomial reductions***; that is, reductions that take polynomial time.

For example, given an instance of Circuit Satisfiability, we would like to generate an equivalent formula. We will explicitly write down what the circuit computes in a formula form. To see how to do this, consider the following example.



$$y_1 = x_1 \wedge x_4 \quad y_2 = \overline{x_4} \quad y_3 = y_2 \wedge x_3$$
$$y_4 = x_2 \vee y_1 \quad y_5 = \overline{x_2} \quad y_6 = \overline{x_5}$$
$$y_7 = y_3 \vee y_5 \quad y_8 = y_4 \wedge y_7 \wedge y_6 \quad y_8$$

We introduced a variable for each wire in the circuit, and we wrote down explicitly what each gate computes. Namely, we wrote a formula for each gate, which holds only if the gate computes correctly the output for its given input.

The circuit is satisfiable **if and only if** there is an assignment such that all the above formulas hold. Alternatively, the circuit is satisfiable if and only if the following (single) formula is satisfiable

$$\begin{aligned}(y_1 = x_1 \wedge x_4) \wedge (y_2 = \overline{x_4}) \wedge (y_3 = y_2 \wedge x_3) \\ \wedge (y_4 = x_2 \vee y_1) \wedge (y_5 = \overline{x_2}) \\ \wedge (y_6 = \overline{x_5}) \wedge (y_7 = y_3 \vee y_5) \\ \wedge (y_8 = y_4 \wedge y_7 \wedge y_6) \wedge y_8.\end{aligned}$$

It is easy to verify that this transformation can be done in polynomial time.

The resulting reduction is depicted in Figure 1.4.

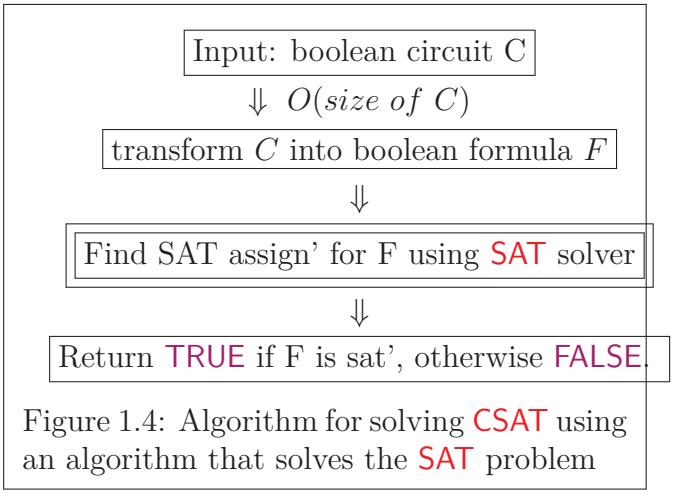


Figure 1.4: Algorithm for solving CSAT using an algorithm that solves the SAT problem

Namely, given a solver for SAT that runs in $T_{\mathsf{SAT}}(n)$, we can solve the CSAT problem in time

$$T_{CSAT}(n) \leq O(n) + T_{SAT}(O(n)),$$

where n is the size of the input circuit. Namely, if we have polynomial time algorithm that solves SAT then we can solve CSAT in polynomial time.

Another way of looking at it, is that we believe that solving CSAT requires exponential time; namely, $T_{\mathsf{CSAT}}(n) \geq 2^n$. Which implies by the above reduction that

$$2^n \leq T_{CSAT}(n) \leq O(n) + T_{SAT}(O(n)).$$

Namely, $T_{\mathsf{SAT}}(n) \geq 2^{n/c} - O(n)$, where c is some positive constant. Namely, if we believe that we need exponential time to solve CSAT then we need exponential time to solve SAT.

This implies that if SAT $\in$ P then CSAT $\in$ P.

We just proved that SAT is as hard as CSAT. Clearly, SAT $\in$ NP which implies the following theorem.

Theorem 1.2.9. *SAT (formula satisfiability) is* NP-Complete.

1.3 More NP-Complete problems

1.3.1 3SAT

A boolean formula is in conjunctive normal form (CNF) if it is a conjunction (AND) of several *clauses*, where a clause is the disjunction (or) of several *literals*, and a literal is either a variable or a negation of a variable. For example, the following is a CNF formula:

$$\overbrace{(a \vee b \vee \overline{c})}^{clause} \wedge (a \vee \overline{e}) \wedge (c \vee e).$$

Definition 1.3.1. 3CNF formula is a CNF formula with *exactly* three literals in each clause.

The problem 3SAT is formula satisfiability when the formula is restricted to be a 3CNF formula.

Theorem 1.3.2. *3SAT is* NP-Complete.

Proof: First, it is easy to verify that 3SAT is in NP.

Next, we will show that 3SAT is NP-Complete by a reduction from CSAT (i.e., Circuit Satisfiability). As such, our input is a circuit C of size n. We will transform it into a 3CNF in several steps:

(A) Make sure every AND/OR gate has only two inputs. If (say) an AND gate have more inputs, we replace it by cascaded tree of AND gates, each one of degree two.

(B) Write down the circuit as a formula by traversing the circuit, as was done for SAT. Let F be the resulting formula.
A clause corresponding to a gate in F will be of the following forms: (i) $a = b \wedge c$ if it corresponds to an AND gate, (ii) $a = b \vee c$ if it corresponds to an OR gate, and (iii) $a = \overline{b}$ if it corresponds to a NOT gate. Notice, that except for the single clause corresponding to the output of the circuit, all clauses are of this form. The clause that corresponds to the output is a single variable.

(C) Change every gate clause into several CNF clauses.

(i) For example, an AND gate clause of the form $a = b \wedge c$ will be translated into

$$\left(a \vee \overline{b} \vee \overline{c}\right) \wedge (\overline{a} \vee b) \wedge (\overline{a} \vee c)\,. \tag{1.1}$$

Note that Eq. (1.1) is true if and only if $a = b \wedge c$ is true. Namely, we can replace the clause $a = b \wedge c$ in F by Eq. (1.1).

(ii) Similarly, an OR gate clause the form $a = b \vee c$ in F will be transformed into

$$(\overline{a} \vee b \vee c) \wedge (a \vee \overline{b}) \wedge (a \vee \overline{c}).$$

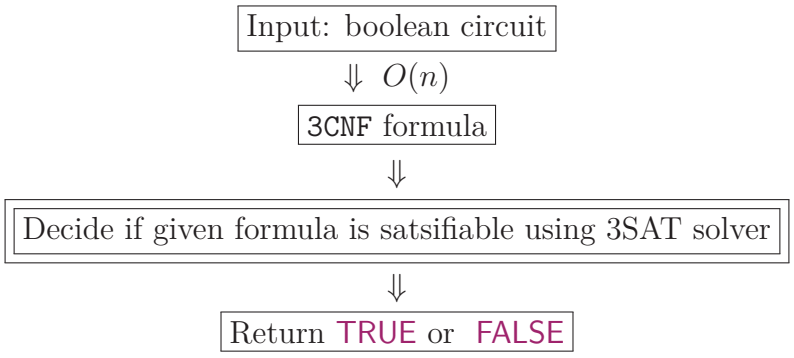


Figure 1.5: Reduction from CSAT to 3SAT

(iii) Finally, a clause $a = \overline{b}$, corresponding to a NOT gate, will be transformed into

$$(a \vee b) \wedge (\overline{a} \vee \overline{b}).$$

(D) Make sure every clause is exactly three literals. Thus, a single variable clause a would be replaced by

$$(a \vee x \vee y) \wedge (a \vee \overline{x} \vee y) \wedge (a \vee x \vee \overline{y}) \wedge (a \vee \overline{x} \vee \overline{y}),$$

by introducing two new dummy variables x and y. And a two variable clause $a \vee b$ would be replaced by

$$(a \vee b \vee y) \wedge (a \vee b \vee \overline{y}),$$

by introducing the dummy variable y.

This completes the reduction, and results in a new `3CNF` formula G which is satisfiable if and only if the original circuit C is satisfiable. The reduction is depicted in Figure 1.5. Namely, we generated an equivalent `3CNF` to the original circuit. We conclude that if $T_{\mathsf{3SAT}}(n)$ is the time required to solve 3SAT then

$$T_{CSAT}(n) \leq O(n) + T_{3SAT}(O(n)),$$

which implies that if we have a polynomial time algorithm for 3SAT, we would solve CSAT is polynomial time. Namely, 3SAT is NP-Complete. ∎

1.4 Bibliographical Notes

Cook's theorem was proved by Stephen Cook (`http://en.wikipedia.org/wiki/Stephen_Cook`). It was proved independently by Leonid Levin (`http://en.wikipedia.org/wiki/Leonid_Levin`) more or less in the same time. Thus, this theorem should be referred to as the Cook-Levin theorem.

The standard text on this topic is [GJ90]. Another useful book is [ACG+99], which is a more recent and more updated, and contain more advanced stuff.

Chapter 2

NP Completeness II

2.1 Max-Clique

We remind the reader, that a ***clique*** is a complete graph, where every pair of vertices are connected by an edge. The MaxClique problem asks what is the largest clique appearing as a subgraph of G. See Figure 2.1.

MaxClique

Instance: A graph G
Question: What is the largest number of nodes in G forming a complete subgraph?

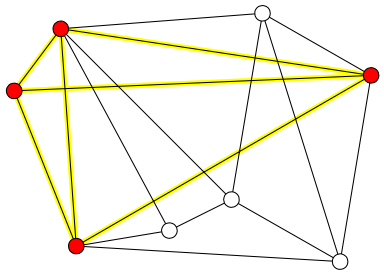


Figure 2.1: A clique of size 4 inside a graph with 8 vertices.

Note that MaxClique is an *optimization* problem, since the output of the algorithm is a number and not just true/false.

The first natural question, is how to solve MaxClique. A naive algorithm would work by enumerating all subsets $S \subseteq V(\mathsf{G})$, checking for each such subset S if it induces a clique in G (i.e., all pairs of vertices in S are connected by an edge of G). If so, we know that G_S is a clique, where G_S denotes the ***induced subgraph*** on S defined by G; that is, the graph formed by removing all the vertices are not in S from G (in particular, only edges that have both endpoints in S appear in G_S). Finally, our algorithm would return the largest S encountered, such that G_S is a clique. The running time of this algorithm is $O(2^n n^2)$ as can be easily verified.

Suggestion 2.1.1. When solving any algorithmic problem, always try first to find a simple (or even naive) solution. You can try optimizing it later, but even a naive solution might give you useful insight into a problem structure and behavior. TIP

We will prove that MaxClique is NP-Hard. Before dwelling into that, the simple algorithm we devised for MaxClique shade some light on why intuitively it should be NP-Hard: It does not seem like there is any way of avoiding the brute force enumeration of all possible subsets of the vertices of G. Thus, a problem is NP-Hard or NP-Complete, *intuitively*, if the only way we know how to solve the problem is to use naive brute force enumeration of all relevant possibilities.

How to prove that a problem X is NP-Hard? Proving that a given problem X is NP-Hard is usually done in two steps. First, we pick a known NP-Complete problem A. Next, we show how to solve any instance of A in polynomial time, assuming that we are given a polynomial time algorithm that solves X.

Proving that a problem X is NP-Complete requires the additional burden of showing that is in NP. Note, that only decision problems can be NP-Complete, but optimization problems can be NP-Hard; namely, the set of NP-Hard problems is much bigger than the set of NP-Complete problems.

Theorem 2.1.2. *MaxClique is* NP-Hard.

Proof: We show a reduction from 3SAT. So, consider an input to 3SAT, which is a formula F defined over n variables (and with m clauses).

We build a graph from the formula F by scanning it, as follows:

(i) For every literal in the formula we generate a vertex, and label the vertex with the literal it corresponds to.
Note, that every clause corresponds to the three such vertices.

(ii) We connect two vertices in the graph, if they are: (i) in different clauses, and (ii) they are *not* a negation of each other.

Let G denote the resulting graph. See Figure 2.2 for a concrete example. Note, that this reduction can be easily be done in quadratic time in the size of the given formula.

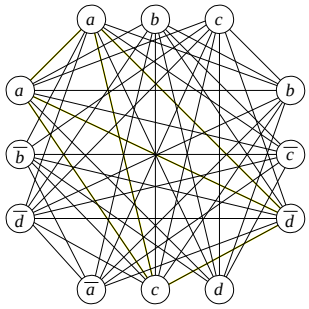


Figure 2.2: The generated graph for the formula $(a \vee b \vee c) \wedge (b \vee \overline{c} \vee \overline{d}) \wedge (\overline{a} \vee c \vee d) \wedge (a \vee \overline{b} \vee \overline{d})$.

We claim that F is satisfiable iff there exists a clique of size m in G.

$\Longrightarrow$ Let $x_1, \ldots, x_n$ be the variables appearing in F, and let $v_1, \ldots, v_n \in \{0, 1\}$ be the satisfying assignment for F. Namely, the formula F holds if we set $x_i = v_i$, for $i = 1, \ldots, n$.
For every clause C in F there must be at least one literal that evaluates to TRUE. Pick a vertex that corresponds to such TRUE value from each clause. Let W be the resulting set of vertices. Clearly, W forms a clique in G. The set W is of size m, since there are m clauses and each one contribute one vertex to the clique.

$\Longleftarrow$ Let U be the set of m vertices which form a clique in G.
We need to translate the clique G_U into a satisfying assignment of F.
(i) set $x_i \leftarrow$ TRUE if there is a vertex in U labeled with x_i.
(ii) set $x_i \leftarrow$ FALSE if there is a vertex in U labeled with $\overline{x_i}$.
This is a valid assignment as can be easily verified. Indeed, assume for the sake of contradiction, that there is a variable x_i such that there are two vertices u, v in U labeled with x_i and $\overline{x_i}$; namely, we are trying to assign to contradictory values of x_i. But then, u and v, by construction will not be connected in G, and as such G_S is not a clique. A contradiction.
Furthermore, this is a satisfying assignment as there is at least one vertex of U in each clause. Implying, that there is a literal evaluating to TRUE in each clause. Namely, F evaluates to TRUE.

Thus, given a polytime (i.e., polynomial time) algorithm for MaxClique, we can solve 3SAT in polytime. We conclude that MaxClique in NP-Hard. ■

MaxClique is an optimization problem, but it can be easily restated as a decision problem.

Clique

Instance: A graph G, integer k
Question: Is there a clique in G of size k?

Theorem 2.1.3. *Clique is* NP-Complete.

Proof: It is NP-Hard by the reduction of Theorem 2.1.2. Thus, we only need to show that it is in NP. This is quite easy. Indeed, given a graph G having n vertices, a parameter k, and a set W of k vertices, verifying that every pair of vertices in W form an edge in G takes $O(u + k^2)$, where u is the size of the input (i.e., number of edges + number of vertices). Namely, verifying a positive answer to an instance of Clique can be done in polynomial time.
Thus, Clique is NP-Complete. ■

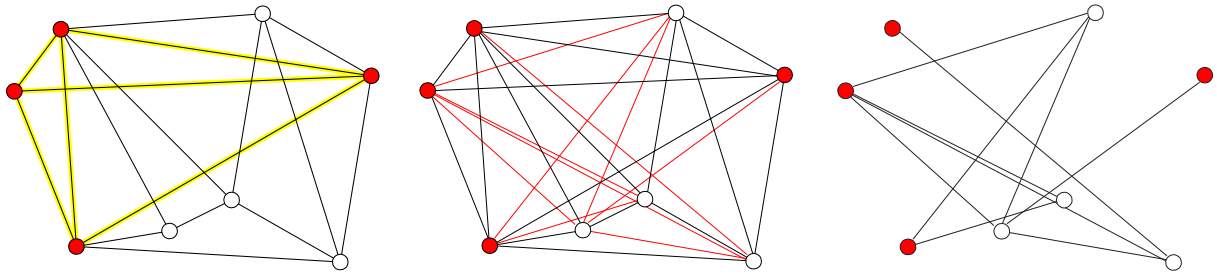
2.2 Independent Set

Definition 2.2.1. A set S of nodes in a graph $\mathsf{G} = (V, E)$ is an ***independent set***, if no pair of vertices in S are connected by an edge.

Independent Set

Instance: A graph G, integer k
Question: Is there an independent set in G of size k?

Theorem 2.2.2. *Independent Set is* NP-Complete.



(a) (b) (c)

Figure 2.3: (a) A clique in a graph G, (b) the complement graph is formed by all the edges not appearing in G, and (c) the complement graph and the independent set corresponding to the clique in G.

Proof: This readily follows by a reduction from Clique. Given G and k, compute the complement graph $\overline{G}$ where we connected two vertices u, v in $\overline{G}$ iff they are independent (i.e., not connected) in G. See Figure 2.3. Clearly, a clique in G corresponds to an independent set in $\overline{G}$, and vice versa. Thus, Independent Set is NP-Hard, and since it is in NP, it is NPC. ■

2.3 Vertex Cover

Definition 2.3.1. For a graph G, a set of vertices $S \subseteq V(\mathsf{G})$ is a ***vertex cover*** if it touches every *edge* of G. Namely, for every edge $uv \in E(\mathsf{G})$ at least one of the endpoints is in S.

Vertex Cover

Instance: A graph G, integer k
Question: Is there a vertex cover in G of size k?

Lemma 2.3.2. *A set S is a vertex cover in G iff $V \setminus S$ is an independent set in G.*

Proof: If S is a vertex cover, then consider two vertices $u, v \in V \setminus S$. If $uv \in E(\mathsf{G})$ then the edge uv is not covered by S. A contradiction. Thus $V \setminus S$ is an independent set in G.

Similarly, if $V \setminus S$ is an independent set in G, then for any edge $uv \in E(\mathsf{G})$ it must be that either u or v are not in $V \setminus G$. Namely, S covers all the edges of G. ■

Theorem 2.3.3. *Vertex Cover is* NP-Complete.

Proof: Vertex Cover is in NP as can be easily verified. To show that it NP-Hard we will do a reduction from Independent Set. So, we are given an instance of Independent Set which is a graph G and parameter k, and we want to know whether there is an independent set in G of size k. By Lemma 2.3.2, G has an independent set of k iff it has a vertex cover of size $n - k$. Thus, feeding G and $n - k$ into (the supposedly given) black box that can solves vertex cover in polynomial time, we can decide if G has an independent set of size k in polynomial time. Thus Vertex Cover is NP-Complete. ■

2.4 Graph Coloring

Definition 2.4.1. A ***coloring***, by c colors, of a graph $\mathsf{G} = (V, E)$ is a mapping $C : V(\mathsf{G}) \to \{1, 2, \ldots, c\}$ such that every vertex is assigned a color (i.e., an integer), such that no two vertices that share an edge are assigned the same color.

Usually, we would like to color a graph with a minimum number of colors. Deciding if a graph can be colored with two colors is equivalent to deciding if a graph is bipartite and can be done in linear time using DFS or BFS[①].

Coloring is useful for resource allocation (used in compilers for example) and scheduling type problems.

Surprisingly, moving from two colors to three colors make the problem much harder.

3Colorable

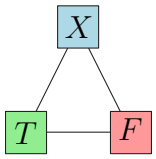
Instance: A graph G.
Question: Is there a coloring of G using three colors?

Theorem 2.4.2. *3Colorable is* NP-Complete.

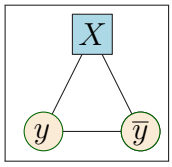
Proof: Clearly, 3Colorable is in NP.

We prove that it is NP-Complete by a reduction from 3SAT. Let $\mathcal{F}$ be the given 3SAT instance. The basic idea of the proof is to use gadgets to transform the formula into a graph. Intuitively, a ***gadget*** is a small component that corresponds to some feature of the input.

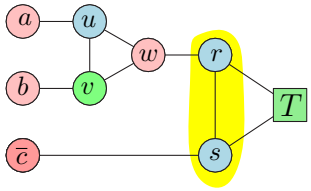
The first gadget will be the ***color generating gadget***, which is formed by three special vertices connected to each other, where the vertices are denoted by X, F and T, respectively. We will consider the color used to color T to correspond to the TRUE value, and the color of the F to correspond to the FALSE value.



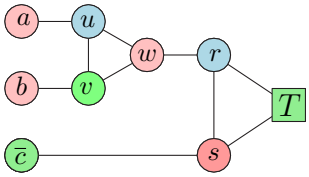
For every variable y appearing in $\mathcal{F}$, we will generate a ***variable gadget***, which is (again) a triangle including two new vertices, denoted by x and $\overline{y}$, and the third vertex is the auxiliary vertex X from the color generating gadget. Note, that in a valid 3-coloring of the resulting graph either y would be colored by T (i.e., it would be assigned the same color as the color as the vertex T) and $\overline{y}$ would be colored by F, or the other way around. Thus, a valid coloring could be interpreted as assigning TRUE or FALSE value to each variable y, by just inspecting the color used for coloring the vertex y.



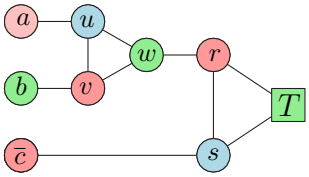
[①]If you do not know the algorithm for this, please read about it to fill this monstrous gap in your knowledge.



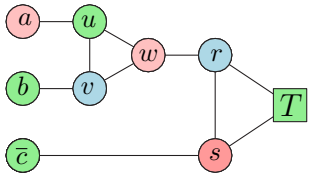
(1) $a \Leftarrow F$, $b \Leftarrow F$, $\overline{c} \Leftarrow F$



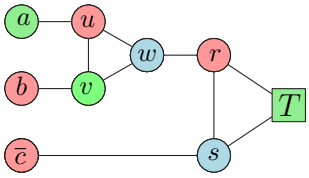
(2) $a \Leftarrow F$, $b \Leftarrow F$, $\overline{c} \Leftarrow T$



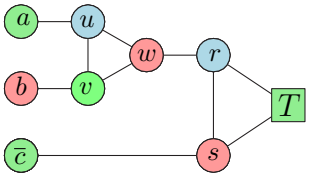
(3) $a \Leftarrow F$, $b \Leftarrow T$, $\overline{c} \Leftarrow F$



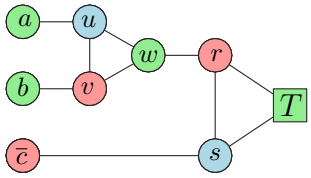
(4) $a \Leftarrow F$, $b \Leftarrow T$, $\overline{c} \Leftarrow T$



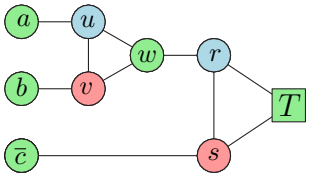
(5) $a \Leftarrow T$, $b \Leftarrow F$, $\overline{c} \Leftarrow F$



(6) $a \Leftarrow T$, $b \Leftarrow F$, $\overline{c} \Leftarrow T$



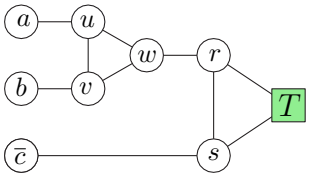
(7) $a \Leftarrow F$, $b \Leftarrow T$, $\overline{c} \Leftarrow T$



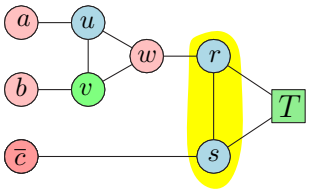
(8) $a \Leftarrow T$, $b \Leftarrow F$, $\overline{c} \Leftarrow F$

Figure 2.4: The clause $a \vee b \vee \overline{c}$ and all the three possible colorings to its literals. If all three literals are colored by the color of the special node F, then there is no valid coloring of this component, see case (1).

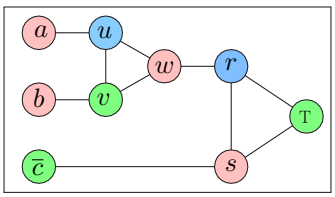
Finally, for every clause we introduce a ***clause gadget***. See the figure on the right – for how the gadget looks like for the clause $a \vee b \vee \bar{c}$. Note, that the vertices marked by a, b and $\bar{c}$ are the corresponding vertices from the corresponding variable gadget. We introduce five new variables for every such gadget. The claim is that this gadget can be colored by three colors if and only if the clause is satisfied. This can be done by brute force checking all 8 possibilities, and we demonstrate it only for two cases. The reader should verify that it works also for the other cases.



Indeed, if all three vertices (i.e., three variables in a clause) on the left side of a variable clause are assigned the F color (in a valid coloring of the resulting graph), then the vertices u and v must be either be assigned X and T or T and X, respectively, in any valid 3-coloring of this gadget (see figure on the left). As such, the vertex w must be assigned the color F. But then, the vertex r must be assigned the X color. But then, the vertex s has three neighbors with all three different colors, and there is no valid coloring for s.



As another example, consider the case when one of the variables on the left is assigned the T color. Then the clause gadget can be colored in a valid way, as demonstrated on the figure on the right.



This concludes the reduction. Clearly, the generated graph can be computed in polynomial time. By the above argumentation, if there is a valid 3-coloring of the resulting graph G, then there is a satisfying assignment for $\mathcal{F}$. Similarly, if there is a satisfying assignment for $\mathcal{F}$ then the G be colored in a valid way using three colors. For how the resulting graph looks like, see Figure 2.5.

This implies that 3Colorable is NP-Complete. ■

Here is an interesting related problem. You are given a graph G as input, and you know that it is 3-colorable. In polynomial time, what is the minimum number of colors you can use to color this graph legally? Currently, the best polynomial time algorithm for coloring such graphs, uses $O\left(n^{3/14}\right)$ colors.

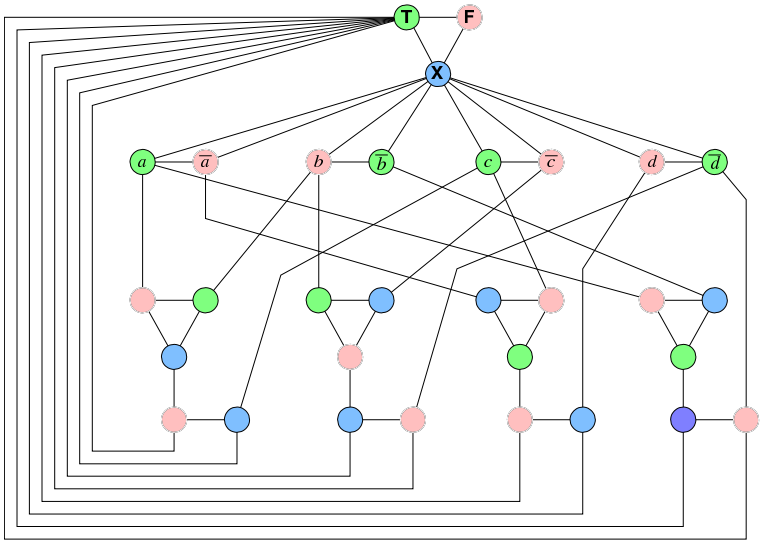


Figure 2.5: The formula $(a \vee b \vee c) \wedge (b \vee \overline{c} \vee \overline{d}) \wedge (\overline{a} \vee c \vee d) \wedge (a \vee \overline{b} \vee \overline{d})$ reduces to the depicted graph.

Chapter 3

NP Completeness III

3.1 Hamiltonian Cycle

Definition 3.1.1. A ***Hamiltonian cycle*** is a cycle in the graph that visits every vertex exactly once.

Definition 3.1.2. An ***Eulerian cycle*** is a cycle in a graph that uses every edge exactly once.

Finding Eulerian cycle can be done in linear time. Surprisingly, finding a Hamiltonian cycle is much harder.

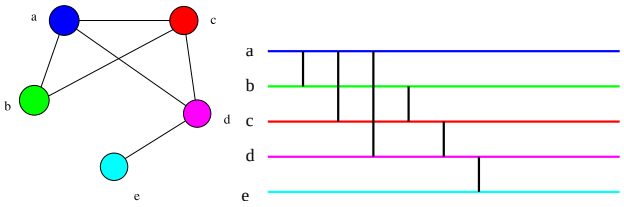
Hamiltonian Cycle

Instance: A graph G.
Question: Is there a Hamiltonian cycle in G?

Theorem 3.1.3. *Hamiltonian Cycle is* NP-Complete.

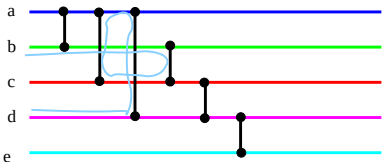
Proof: Hamiltonian Cycle is clearly in NP.

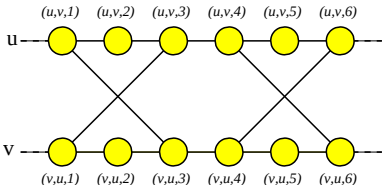
We will show a reduction from Vertex Cover. Given a graph G and integer k we redraw G in the following way: We turn every vertex into a horizontal line segment, all of the same length. Next, we turn an edge in the original graph G into a *gate*, which is a vertical segment connecting the two relevant vertices.



Note, that there is a Vertex Cover in G of size k if and only if there are k horizontal lines that stabs all the gates in the resulting graph H (a line stabs a gate if one of the endpoints of the gate lies on the line).

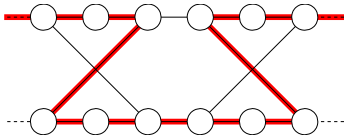
Thus, computing a vertex cover in G is equivalent to computing k disjoints paths through the graph G that visits all the gates. However, there is a technical problem: a path might change venues or even go back. See figure on the right.

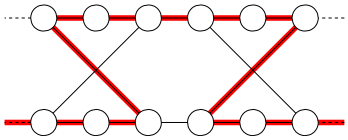


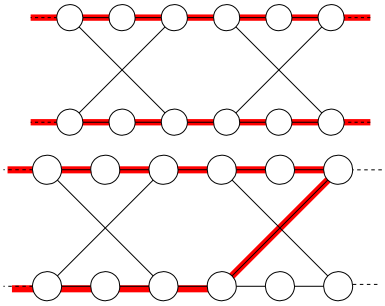


To overcome this problem, we will replace each gate with a component that guarantees, that if you visit all its vertices, you have to go forward and can NOT go back (or change "lanes"). The new component is depicted on the left.

There only three possible ways to visit *all* the vertices of the components by paths that do not start/end inside the component, and they are the following:

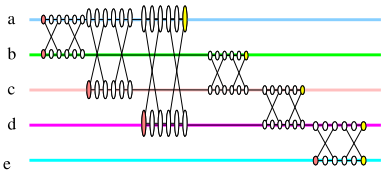




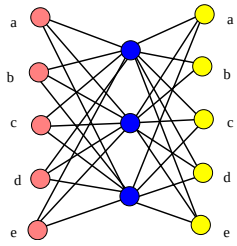


The proof that this is the only three possibilities is by brute force. Depicted on the right is one impossible path, that tries to backtrack by entering on the top and leaving on the bottom. Observe, that there are vertices left unvisited. Which means that not all the vertices in the graph are going to be visited, because we add the constraint, that the paths start/end outside the gate-component (this condition would be enforced naturally by our final construction).

The resulting graph H_1 for the example graph we started with is depicted on the right. There exists a Vertex Cover in the original graph **iff** there exists k paths that start on the left side and end on the right side, in this weird graph. And these k paths visits all the vertices.



The final stroke is to add connection from the left side to the right side, such that once you arrive to the right side, you can go back to the left side. However, we want connection that allow you to travel exactly k times. This is done by adding to the above graph a "routing box" component H_2 depicted on the right, with k new middle vertices. The ith vertex on the left of the routing component is the left most vertex of the ith horizontal line in the graph, and the ith vertex on the right of the component is the right most vertex of the ith horizontal line in the graph.



It is now easy (but tedious) to verify that the resulting graph $H_1 \cup H_2$ has a Hamiltonian path **iff** H_1has k paths going from left to right, which happens, **iff** the original graph has a Vertex Cover of size k. It is easy to verify that this reduction can be done in polynomial time. ■

3.2 Traveling Salesman Problem

A traveling salesman tour, is a Hamiltonian cycle in a graph, which its price is the price of all the edges it uses.

TSP

Instance: $G = (V, E)$ a complete graph - n vertices, $c(e)$: Integer cost function over the edges of G, and k an integer.
Question: Is there a traveling-salesman tour with cost at most k?

Theorem 3.2.1. *TSP is* NP-Complete.

Proof: Reduction from Hamiltonian cycle. Consider a graph $G = (V, E)$, and let H be the complete graph defined over V. Let

$$c(e) = \begin{cases} 1 & e \in E(G) \\ 2 & e \notin E(G). \end{cases}$$

Clearly, the cheapest TSP in H with cost function equal to n iff G is Hamiltonian. Indeed, if G is not Hamiltonian, then the TSP must use one edge that does not belong to G, and then, its price would be at least $n + 1$. ■

3.3 Subset Sum

We would like to prove that the following problem, Subset Sum is NPC.

Subset Sum

Instance: S - set of positive integers,t: - an integer number (Target)
Question: Is there a subset $X \subseteq S$ such that $\sum_{x \in X} x = t$?

How does one prove that a problem is NP-Complete? First, one has to choose an appropriate NPC to reduce from. In this case, we will use 3SAT. Namely, we are given a 3CNF formula with n variables and m clauses. The second stage, is to "play" with the problem and understand what kind of constraints can be encoded in an instance of a given problem and understand the general structure of the problem.

The first observation is that we can use very long numbers as input to Subset Sum. The numbers can be of polynomial length in the size of the input 3SAT formula F.

The second observation is that in fact, instead of thinking about Subset Sum as adding numbers, we can think about it as a problem where we are given vectors with k components each, and the sum of the vectors (coordinate by coordinate, must match. For example, the input might be the vectors $(1, 2), (3, 4), (5, 6)$ and the target vector might be $(6, 8)$. Clearly, $(1, 2) + (5, 6)$ give the required target vector. Lets refer to this new problem as Vec Subset Sum.

Vec Subset Sum

Instance: S - set of n vectors of dimension k, each vector has non-negative numbers for its coordinates, and a target vector $\overrightarrow{t}$.
Question: Is there a subset $X \subseteq S$ such that $\sum_{\overrightarrow{x} \in X} \overrightarrow{x} = \overrightarrow{t}$?

Given an instance of Vec Subset Sum, we can covert it into an instance of Subset Sum as follows: We compute the largest number in the given instance, multiply it by $n^2 \cdot k \cdot 100$, and compute how many digits are required to write this number down. Let U be this number of digits. Now, we take every vector in the given instance and we write it down using U digits, padding it with zeroes as necessary. Clearly, each vector is now converted into a huge integer number. The property is now that a sub of numbers in a specific column of the given instance can not spill into digits allocated for a different column since there are enough zeroes separating the digits corresponding to two different columns.

Target	??	??	01	???
a_1	??	??	01	??
a_2	??	??	01	??

Next, let us observe that we can force the solution (if it exists) for Vec Subset Sum to include exactly one vector out of two vectors. To this end, we will introduce a new coordinate (i.e., a new column in the table on the right) for all the vectors. The two vectors a_1 and a_2 will have 1 in this coordinate, and all other vectors will have zero in this coordinate. Finally, we set this coordinate in the target vector to be 1. Clearly, a solution is a subset of vectors that in this coordinate add up to 1. Namely, we have to choose either a_1 or a_2 into our solution.

In particular, for each variable x appearing in F, we will introduce two rows, denoted by x and $\overline{x}$ and introduce the above mechanism to force choosing either x or $\overline{x}$ to the optimal solution. If x (resp. $\overline{x}$) is chosen into the solution, we will interpret it as the solution to F assigns TRUE (resp. FALSE) to x.

numbers	...	$C \equiv a \vee b \vee \overline{c}$	...
a	...	01	...
$\overline{a}$	...	00	...
b	...	01	...
$\overline{b}$	...	00	...
c	...	00	...
$\overline{c}$	...	01	...
C fix-up 1	000	07	000
C fix-up 2	000	08	000
C fix-up 3	000	09	000
TARGET		10	

Next, consider a clause $C \equiv a \vee b \vee \overline{c}$.appearing in F. This clause requires that we choose at least one row from the rows corresponding to a, b to $\overline{c}$. This can be enforced by introducing a new coordinate for the clauses C, and setting 1 for each row that if it is picked then the clauses is satisfied. The question now is what do we set the target to be? Since a valid solution might have any number between 1 to 3 as a sum of this coordinate. To overcome this, we introduce three new dummy rows, that store in this coordinate, the numbers 7, 8 and 9, and we set this coordinate in the target to be 10. Clearly, if we pick to dummy rows into the optimal solution then sum in this coordinate would exceed 10. Similarly, if we do not pick one of these three dummy rows to the optimal solution, the maximum sum in this coordinate would be $1 + 1 + 1 = 3$, which is smaller than 10. Thus, the only possibility is to pick one dummy row, and some subset of the rows such that the sum is 10. Notice, this "gadget" can accommodate any (non-empty) subset of the three rows chosen for a, b and $\overline{c}$.

We repeat this process for each clause of F. We end up with a set U of $2n+3m$ vectors with $n+m$ coordinate, and the question if there is a subset of these vectors that add up to the target vector. There is such a subset if and only if the original formula F is satisfiable, as can be easily verified. Furthermore, this reduction can be done in polynomial time.

Finally, we convert these vectors into an instance of Subset Sum. Clearly, this instance of Subset Sum has a solution if and only if the original instance of 3SAT had a solution. Since Subset Sum is in NP as an be easily verified, we conclude that that Subset Sum is NP-Complete.

Theorem 3.3.1. *Subset Sum is* NP-Complete.

For a concrete example of the reduction, see Figure 3.1.

3.4 3 dimensional Matching (3DM)

3DM

Instance: X, Y, Z sets of n elements, and T a set of triples, such that $(a,b,c) \in T \subseteq X \times Y \times Z$.
Question: Is there a subset $S \subseteq T$ of n disjoint triples, s.t. every element of $X \cup Y \cup Z$ is covered exactly once.?

Theorem 3.4.1. *3DM is* NP-Complete.

The proof is long and tedious and is omitted.

BTW, $2DM$ is polynomial (later in the course?).

3.5 Partition

Partition

Instance: A set S of n numbers.
Question: Is there a subset $T \subseteq S$ s.t. $\sum_{t \in T} t = \sum_{s \in S \setminus T} s$.?

Theorem 3.5.1. *Partition is* NP-Complete.

Proof: Partition is in NP, as we can easily verify that such a partition is valid.

Reduction from Subset Sum. Let the given instance be n numbers $a_1, \ldots, a_n$ and a target number t. Let $S = \sum_{i=}^{n} a_i$, and set $a_{n+1} = 3S - t$ and $a_{n+2} = 3S - (S-t) = 2S + t$. It is easy to verify that there is a solution to the given instance of subset sum, iff there is a solution to the following instance of partition:

$$a_1, \ldots, a_n, a_{n+1}, a_{n+2}.$$

Clearly, Partition is in NP and thus it is NP-Complete. ■

numbers	$a \vee \overline{a}$	$b \vee \overline{b}$	$c \vee \overline{c}$	$d \vee \overline{d}$	$D \equiv \overline{b} \vee c \vee \overline{d}$	$C \equiv a \vee b \vee \overline{c}$
a	1	0	0	0	00	01
$\overline{a}$	1	0	0	0	00	00
b	0	1	0	0	00	01
$\overline{b}$	0	1	0	0	01	00
c	0	0	1	0	01	00
$\overline{c}$	0	0	1	0	00	01
d	0	0	0	1	00	00
$\overline{d}$	0	0	0	1	01	01
C fix-up 1	0	0	0	0	00	07
C fix-up 2	0	0	0	0	00	08
C fix-up 3	0	0	0	0	00	09
D fix-up 1	0	0	0	0	07	00
D fix-up 2	0	0	0	0	08	00
D fix-up 3	0	0	0	0	09	00
TARGET	1	1	1	1	10	10

numbers
010000000001
010000000000
000100000001
000100000100
000001000100
000001000001
000000010000
000000010101
000000000007
000000000008
000000000009
000000000700
000000000800
000000000900
010101011010

Figure 3.1: The Vec Subset Sum instance generated for the 3SAT formula $F = \left(\overline{b} \vee c \vee \overline{d}\right) \wedge (a \vee b \vee \overline{c})$ is shown on the left. On the right side is the resulting instance of Subset Sum.

3.6 Some other problems

It is not hard to show that the following problems are NP-Complete:

SET COVER

Instance: $(\mathsf{S}, \mathcal{F}, k)$:
S: A set of n elements
$\mathcal{F}$: A family of subsets of S, s.t. $\bigcup_{X \in \mathcal{F}} X = \mathsf{S}$.
k: A positive integer.
Question: Are there k sets $S_1, \ldots, S_k \in \mathcal{F}$ that cover S. Formally, $\bigcup_i S_i = \mathsf{S}$?

Chapter 4

Dynamic programming

The events of 8 September prompted Foch to draft the later legendary signal: "My centre is giving way, my right is in retreat, situation excellent. I attack." It was probably never sent.
– – The first world war, John Keegan..

4.1 Basic Idea - Partition Number

Definition 4.1.1. For a positive integer n, the ***partition number*** of n, denoted by $p(n)$, is the number of different ways to represent n as a decreasing sum of positive integers.

The different number of partitions of 6 are shown on the right.

$6 = 6$		
6=5+1		
6=4+2	6=4+1+1	
$6 = 3 + 3$	$6 = 3 + 2 + 1$	6+3+1+1+1
6=2+2+2	6=2+2+1+1	6=2+1+1+1+1
6=1+1+1+1+1+1		

It is natural to ask how to compute $p(n)$. The "trick" is to think about a recursive solution and observe that once we decide what is the leading number d, we can solve the problem recursively on the remaining budget $n - d$ under the constraint that no number exceeds d..

Suggestion 4.1.2. Recursive algorithms are one of the main tools in developing algorithms (and writing programs). If you do not feel comfortable with recursive algorithms you should spend time playing with recursive algorithms till you feel comfortable using them. Without the ability to think recursively, this class would be a long and painful torture to you. Speak with me if you need guidance on this topic. TIP

The resulting algorithm is depicted on the right. We are interested in analyzing its running time. To this end, draw the recursion tree of PARTITIONS and observe that the amount of work spend at each node, is proportional to the number of children it has. Thus, the overall time spend by the algorithm is proportional to the size of the recurrence tree, which is proportional (since every node is either a leaf or has at least two children) to the number of leafs in the tree, which is $\Theta(p(n))$.

This is not very exciting, since it is easy verify that $3^{\sqrt{n}/4} \leq p(n) \leq n^n$.

```
PartitionsI(num, d)    //d-max digit
  if (num ≤ 1) or (d = 1)
      return 1
  if d > num
      d ← num
  res ← 0
  for i ← d down to 1
      res = res + PartitionsI(num − i,i)
  return res

Partitions(n)
  return PartitionsI(n, n)
```

Exercise 4.1.3. Prove the above bounds on $p(n)$ (or better bounds).

Suggestion 4.1.4. Exercises in the class notes are a natural easy questions for inclusions in exams. You probably want to spend time doing them. TIP

Hardy and Ramanujan (in 1918) showed that $p(n) \approx \frac{e^{\pi\sqrt{2n/3}}}{4n\sqrt{3}}$ (which I am sure was your first guess).

It is natural to ask, if there is a faster algorithm. Or more specifically, why is the algorithm **Partitions** so slowwwwwwwwwwwwwwwwww? The answer is that during the computation of **Partitions**(n) the function **PartitionsI**(num, max_digit) is called a lot of times with the *same* parameters.

An easy way to overcome this problem is cache the results of **PartitionsI** using a hash table.[①] Whenever **PartitionsI** is being called, it checks in a cache table if it already computed the value of the function for this parameters, and if so it returns the result. Otherwise, it computes the value of the function and before returning the value, it stores it in the cache. This simple (but powerful) idea is known as ***memoization***.

What is the running time of **PartitionS_C**? Analyzing recursive algorithm that have been transformed by memoization are usually analyzed as follows: (i) bound the number of values stored in the hash table, and (ii) bound

```
PartitionsI_C(num, max_digit)
  if (num ≤ 1) or (max_digit = 1)
      return 1
  if max_digit > num
      d ← num
  if ⟨num, max_digit⟩ in cache
      return cache(⟨num, max_digit⟩)
  res ← 0
  for i ← max_digit down to 1 do
      res += PartitionsI_C(num − i,i)
  cache(⟨num, max_digit⟩) ← res
  return res

PartitionS_C(n)
  return PartitionsI_C(n, n)
```

[①] Throughout the course, we will assume that a hash table operation can be done in constant time. This is a reasonable assumption using randomization and perfect hashing.

the amount of work involved in storing one
value into the hash table (ignoring recursive calls).

Here is the argument in this case:

(A) If a call to **PartitionsI_C** takes (by itself) more than constant time, then this call performs a store in the cache.

(B) Number of store operations in the cache is $O(n^2)$, since this is the number of different entries stored in the cache. Indeed, for **PartitionsI_C**(num, max_digit), the parameters num and max_digit are both integers in the range $1, \ldots, n$.

(C) We charge the work in the loop to the resulting store. The work in the loop is at most $O(n)$ time (since $max\_digit \leq n$).

(D) As such, the overall running time of **PartitionS_C**(n) is $O(n^2) \times O(n) = O(n^3)$.

Note, that this analysis is naive but it would be sufficient for our purposes (verify that the bound of $O(n^3)$ on the running time is tight in this case).

4.1.1 A Short sermon on memoization

This idea of memoization is generic and nevertheless very useful. To recap, it works by taking a recursive function and caching the results as the computations goes on. Before trying to compute a value, check if it was already computed and if it is already stored in the cache. If so, return result from the cache. If it is not in the cache, compute it and store it in the cache (for the time being, you can think about the cache as being a hash table).

- **When does it work:** There is a lot of inefficiency in the computation of the recursive function because the same call is being performed repeatedly.
- **When it does NOT work:**
  (A) The number of different recursive function calls (i.e., the different values of the parameters in the recursive call) is "large".
  (B) When the function has side effects.

Tidbit 4.1.5. Some functional programming languages allow one to take a recursive function $f(\cdot)$ that you already implemented and give you a memorized version $f'(\cdot)$ of this function without the programmer doing any extra work. For a nice description of how to implement it in Scheme see [ASS96]. tidbit

It is natural to ask if we can do better than just using caching? As usual in life – more pain, more gain. Indeed, in a lot of cases we can analyze the recursive calls, and store them directly in an (sometime multi-dimensional) array. This gets rid of the recursion (which used to be an important thing long time ago when memory, used by the stack, was a truly limited resource, but it is less important nowadays) which usually yields a slight improvement in performance in the real world.

This technique is known as ***dynamic programming***[②]. We can sometime save space and improve running time in dynamic programming over memoization.

[②] As usual in life, it is not dynamic, it is not programming, and its hardly a technique. To overcome this, most texts find creative ways to present this topic in the most opaque way possible.

Dynamic programing made easy:

(A) Solve the problem using recursion - easy (?).
(B) Modify the recursive program so that it caches the results.
(C) Dynamic programming: Modify the cache into an array.

4.2 Example – Fibonacci numbers

Let us revisit the classical problem of computing Fibonacci numbers.

4.2.1 Why, where, and when?

To remind the reader, in the Fibonacci sequence, the first two numbers $F_0 = 0$ and $F_1 = 1$, and $F_i = F_{i-1} + F_{i-2}$, for $i > 1$. This sequence was discovered independently in several places and times. From Wikipedia:

> "The Fibonacci sequence appears in Indian mathematics, in connection with Sanskrit prosody. In the Sanskrit oral tradition, there was much emphasis on how long (L) syllables mix with the short (S), and counting the different patterns of L and S within a given fixed length results in the Fibonacci numbers; the number of patterns that are m short syllables long is the Fibonacci number F_{m+1}."

(To see that, imagine that a long syllable is equivalent in length to two short syllables.) Surprisingly, the credit for this formalization goes back more than 2000 years (!)

Fibonacci was a decent mathematician (1170âĂŞ-1250 AD), and his most significant and lasting contribution was spreading the Hindu-Arabic numerical system (i.e., zero) in Europe. He was the son of a rich merchant that spend much time growing up in Algiers, where he learned the decimal notation system. He traveled throughout the Mediterranean world to study mathematics. When he came back to Italy he published a sequence of books (the first one "Liber Abaci" contained the description of the decimal notations system). In this book, he also posed the following problem:

> Consider a rabbit population, assuming that: A newly born pair of rabbits, one male, one female, are put in a field; rabbits are able to mate at the age of one month so that at the end of its second month a female can produce another pair of rabbits; rabbits never die and a mating pair always produces one new pair (one male, one female) every month from the second month on. The puzzle that Fibonacci posed was: how many pairs will there be in one year?

(The above is largely based on Wikipedia.)

4.2.2 Computing Fibonacci numbers

The recursive function for computing Fibonacci numbers is depicted on the right. As before, the running time of **FibR**(n) is proportional to $O(F_n)$, where F_n is the nth Fibonacci number. It is known that

```
FibR(n)
    if n = 0
        return 1
    if n = 1
        return 1
    return FibR(n − 1) + FibR(n − 2)
```

$$F_n = \frac{1}{\sqrt{5}}\left[\left(\frac{1+\sqrt{5}}{2}\right)^n + \left(\frac{1-\sqrt{5}}{2}\right)^n\right] = \Theta(\phi^n),$$

where $\phi = \frac{1+\sqrt{5}}{2}$.

We can now use memoization, and with a bit of care, it is easy enough to come up with the dynamic programming version of this procedure, see **FibDP** on the right. Clearly, the running time of **FibDP**(n) is linear (i.e., $O(n)$).

```
FibDP(n)
    if n ≤ 1
        return 1
    if F[n] initialized
        return F[n]
    F[n] ⇐= FibDP(n − 1)+FibDP(n − 2)
    return F[n]
```

A careful inspection of **FibDP** exposes the fact that it fills the array $F[...]$ from left to right. In particular, it only requires the last two numbers in the array.

As such, we can get rid of the array all together, and reduce space needed to $O(1)$: This is a phenomena that is quite common in dynamic programming: By carefully inspecting the way the array/table is being filled, sometime one can save space by being careful about the implementation.

```
FibI(n)
    prev ← 0, curr ← 1
    for i = 1 to n do
        next ← curr + prev
        prev ← curr
        curr ← next
    return curr
```

The running time of **FibI** is identical to the running time of **FibDP**. Can we do better?

Surprisingly, the answer is yes, to this end observe that

$$\begin{pmatrix} y \\ x+y \end{pmatrix} = \begin{pmatrix} 0 & 1 \\ 1 & 1 \end{pmatrix} \begin{pmatrix} x \\ y \end{pmatrix}.$$

As such,

$$\begin{pmatrix} F_{n-1} \\ F_n \end{pmatrix} = \begin{pmatrix} 0 & 1 \\ 1 & 1 \end{pmatrix} \begin{pmatrix} F_{n-2} \\ F_{n-1} \end{pmatrix} = \begin{pmatrix} 0 & 1 \\ 1 & 1 \end{pmatrix}^2 \begin{pmatrix} F_{n-3} \\ F_{n-2} \end{pmatrix} = \begin{pmatrix} 0 & 1 \\ 1 & 1 \end{pmatrix}^{n-3} \begin{pmatrix} F_2 \\ F_1 \end{pmatrix}.$$

Thus, computing the nth Fibonacci number can be done by computing $\begin{pmatrix} 0 & 1 \\ 1 & 1 \end{pmatrix}^{n-3}$.

How to this quickly? Well, we know that $a*b*c = (a*b)*c = a*(b*c)$[3], as such one can compute a^n by repeated squaring, see pseudo-code on the right. The running time of **FastExp** is $O(\log n)$ as can be easily verified. Thus, we can compute in F_n in $O(\log n)$ time.

```
FastExp(a, n)
    if n = 0 then
        return 1
    if n = 1 then
        return a
    if n is even then
        return (FastExp(a, n/2))^2
    else
        return a * (FastExp(a, (n-1)/2))^2
```

But, something is very strange. Observe that F_n has $\approx \log_{10} 1.68...^n = \Theta(n)$ digits. How can we compute a number that is that large in logarithmic time? Well, we assumed that the time to handle a number is $O(1)$ independent of its size. This is not true in practice if the numbers are large. Naturally, one has to be very careful with such assumptions.

4.3 Edit Distance

We are given two strings A and B, and we want to know how close the two strings are too each other. Namely, how many edit operations one has to make to turn the string A into B?

We allow the following operations: (i) insert a character, (ii) delete a character, and (iii) replace a character by a different character. Price of each operation is one unit.

For example, consider the strings A ="har-peled" and B ="sharp eyed". Their ***edit distance*** is 4, as can be easily seen.

	h	a	r	–	p		e	l	e	d
s	h	a	r		p	<space>	e	y	e	d
1	0	0	0	1	0	1	0	1	0	0

Insert: [ ] / [s] delete: [–] / [ ] replace: [l] / [y]

(still) insert: [ ] / [<space>] ignore: [e] / [e]

Figure 4.1: Interpreting edit-distance as a alignment task. Aligning identical characters to each other is free of cost. The price in the above example is 4. There are other ways to get the same edit-distance in this case.

[3] Associativity of multiplication...

But how do we compute the edit-distance (min # of edit operations needed)?

The idea is to list the edit operations from left to right. Then edit distance turns into a an alignment problem. See Figure 4.1.

In particular, the idea of the recursive algorithm is to inspect the last character and decide which of the categories it falls into: insert, delete or ignore. See pseudo-code on the right.

```
ed(A[1..m], B[1..n])
    if m = 0 return n
    if n = 0 return m
    p_insert = ed(A[1..m], B[1..(n − 1)]) + 1
    p_delete = ed(A[1..(m − 1)], B[1..n]) + 1
    p_r/i = ed(A[1..(m − 1)], B[1..(n − 1)])
                + [A[m] ≠ B[n]]
    return min(p_insert, p_delete, p_replace/ignore)
```

The running time of **ed**(...)? Clearly exponential, and roughly 2^{n+m}, where $n+m$ is the size of the input.

So how many **different** recursive calls **ed** performs? Only:$O(m*n)$ different calls, since the only parameters that matter are n and m.

So the natural thing is to introduce memoization. The resulting algorithm **edM** is depicted on the right. The running time of **edM**(n, m) when executed on two strings of length n and m respective is $O(nm)$, since there are $O(nm)$ store operations in the cache, and each store requires $O(1)$ time (by charging one for each recursive call).

```
edM(A[1..m], B[1..n])
  if m = 0 return n
  if n = 0 return m
  if T[m, n] is initialized then return T[m, n]
  p_insert = edM(A[1..m], B[1..(n − 1)]) + 1
  p_delete = edM(A[1..(m − 1)], B[1..n]) + 1
  p_r/i = edM(A[1..(m − 1)], B[1..(n − 1)]) + [A[m] ≠ B[n]]
  T[m, n] ← min(p_insert, p_delete, p_replace/ignore)
  return T[m, n]
```

Looking on the entry $T[i, j]$ in the table, we realize that it depends only on $T[i-1, j]$, $T[i, j-1]$ and $T[i-1, j-1]$. Thus, instead of recursive algorithm, we can fill the table T row by row, from left to right.

```
edDP(A[1..m], B[1..n])
  for i = 1 to m do T[i, 0] ← i
  for j = 1 to n do T[0, j] ← j
  for i ← 1 to m do
      for j ← 1 to n do
          p_insert = T[i, j − 1] + 1
          p_delete = T[i − 1, j] + 1
          p_r/ignore = T[i − 1.j − 1] + [A[i] ≠ B[j]]
          T[i, j] ← min(p_insert, p_delete, p_r/ignore)
  return T[m, n]
```

The dynamic programming version that uses a two dimensional array is pretty simple now to derive and is depicted on the left. Clearly, it requires $O(nm)$ time, and $O(nm)$ space. See the pseudo-code of the resulting algorithm **edDP** on the left.

It is enlightening to think about the algorithm as computing for each $T[i, j]$ the cell it got the value from. What you get is a tree encoded in the table. See Figure 4.2. It is now easy to extract from the table the sequence of edit operations that realizes the minimum edit distance between

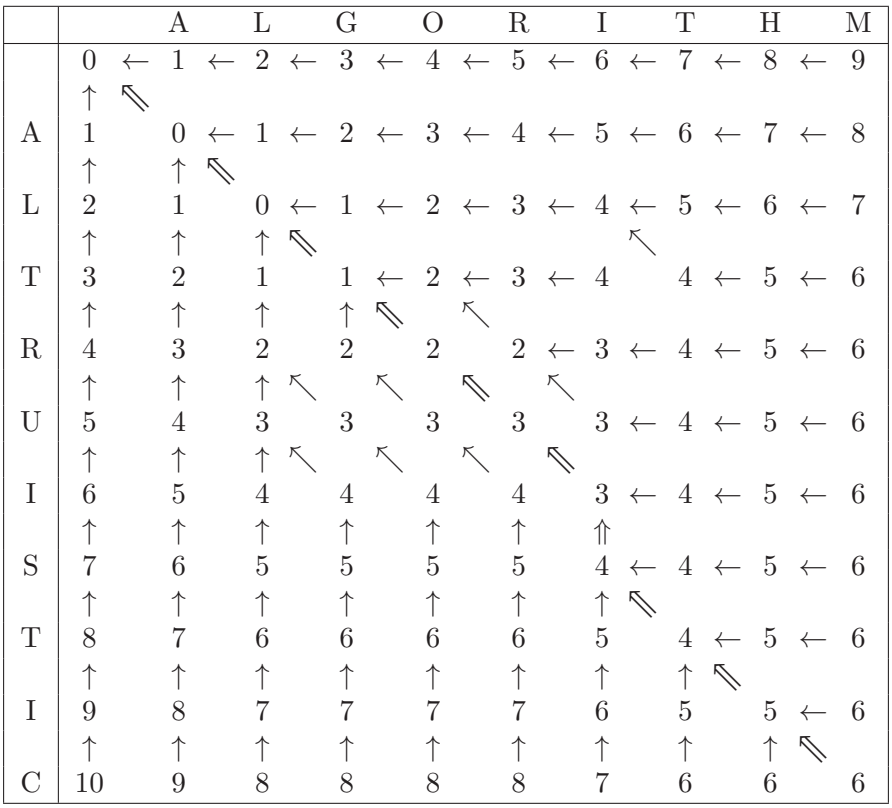


Figure 4.2: Extracting the edit operations from the table.

A and B. Indeed, we start a walk on this graph from the node corresponding to $T[n, m]$. Every time we walk left, it corresponds to a deletion, every time we go up, it corresponds to an insertion, and going sideways corresponds to either replace/ignore.

Note, that when computing the ith row of $T[i, j]$, we only need to know the value of the cell to the left of the current cell, and two cells in the row above the current cell. It is thus easy to verify that the algorithm needs only the remember the current and previous row to compute the edit distance. We conclude:

Theorem 4.3.1. *Given two strings A and B of length n and m, respectively, one can compute their edit distance in $O(nm)$. This uses $O(nm)$ space if we want to extract the sequence of edit operations, and $O(n+m)$ space if we only want to output the price of the edit distance.*

Exercise 4.3.2. Show how to compute the sequence of edit-distance operations realizing the edit distance using only $O(n + m)$ space and $O(nm)$ running time. (Hint: Use a recursive algorithm, and argue that the recursive call is always on a matrix which is of size, roughly, half of the input matrix.)

4.3.1 Shortest path in a DAG and dynamic programming

Given a dynamic programming problem and its associated recursive program, one can consider all the different possible recursive calls, as ***configurations***. We can create graph, every configuration is a node, and an edge is introduced between two configurations if one configuration is computed from another configuration, and we put the additional price that might be involved in moving between the two configurations on the edge connecting them. As such, for the edit distance, we have directed edges from the vertex (i, j) to $(i, j-1)$ and $(i-1, j)$ both with weight 1 on them. Also, we have an edge between (i, j) to $(i-1, j-1)$ which is of weight 0 if $A[i] = B[j]$ and 1 otherwise. Clearly, in the resulting graph, we are asking for the shortest path between (n, m) and $(0, 0)$.

And here are where things gets interesting. The resulting graph G is a DAG (***directed acyclic graph***[4]). DAG can be interpreted as a partial ordering of the vertices, and by topological sort on the graph (which takes linear time), one can get a full ordering of the vertices which agrees with the DAG. Using this ordering, one can compute the shortest path in a DAG in linear time (in the size of the DAG). For edit-distance the DAG size is $O(nm)$, and as such this algorithm takes $O(nm)$ time.

This interpretation of dynamic programming as a shortest path problem in a DAG is a useful way of thinking about it, and works for many dynamic programming problems.

More surprisingly, one can also compute the longest path in a DAG in linear time. Even for negative weighted edges. This is also sometime a problem that solving it is equivalent to dynamic programming.

[4] No cycles in the graph – its a miracle!

Chapter 5

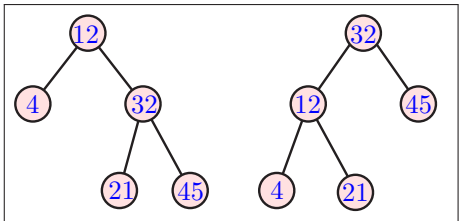
Dynamic programming II - The Recursion Strikes Back

> "No, mademoiselle, I don't capture elephants. I content myself with living among them. I like them. I like looking at them, listening to them, watching them on the horizon. To tell you the truth, I'd give anything to become an elephant myself. That'll convince you that I've nothing against the Germans in particular: they're just men to me, and that's enough."
>
> – – The roots of heaven, Romain Gary.

5.1 Optimal search trees

Given a binary search tree $\mathcal{T}$, the time to search for an element x, that is stored in $\mathcal{T}$, is $O(1 + \text{depth}(\mathcal{T}, x))$, where $\text{depth}(\mathcal{T}, x)$ denotes the depth of x in $\mathcal{T}$ (i.e., this is the length of the path connecting x with the root of $\mathcal{T}$).

Problem 5.1.1. Given a set of n (sorted) keys $A[1 \ldots n]$, build the best binary search tree for the elements of A.



Two possible search trees for the set $A = [4, 12, 21, 32, 45]$.

Note, that we store the values in the internal node of the binary trees. The figure on the right shows two possible search trees for the same set of numbers. Clearly, if we are accessing the number 12 all the time, the tree on the left would be better to use than the tree on the right.

Usually, we just build a balanced binary tree, and this is good enough. But assume that we have additional information about what is the frequency in which we access the element $A[i]$, for $i = 1, \ldots, n$. Namely, we know that $A[i]$ is going be accessed $\mathsf{f}[i]$ times, for $i = 1, \ldots, n$.

In this case, we know that the total search time for a tree $\mathcal{T}$ is $S(\mathcal{T}) = \sum_{i=1}^{n} (\text{depth}(\mathcal{T}, i) + 1)\, \mathsf{f}[i]$, where $\text{depth}(\mathcal{T}, i)$ is the depth of the node in $\mathcal{T}$ storing the value $A[i]$. Assume that $A[r]$ is

the value stored in the root of the tree $\mathcal{T}$. Clearly, all the values smaller than $A[r]$ are in the subtree $\text{left}_{\mathcal{T}}$, and all values larger than r are in $\text{right}_{\mathcal{T}}$. Thus, the total search time, in this case, is

$$S(\mathcal{T}) = \sum_{i=1}^{r-1} (\text{depth}(\text{left}_{\mathcal{T}}, i) + 1)\,\mathsf{f}[i] + \overbrace{\sum_{i=1}^{n} \mathsf{f}[i]}^{\text{price of access to root}} + \sum_{i=r+1}^{n} (\text{depth}(\text{right}_{\mathcal{T}}, i) + 1)\,\mathsf{f}[i].$$

Observe, that if $\mathcal{T}$ is the optimal search tree for the access frequencies $\mathsf{f}[1], \ldots, \mathsf{f}[n]$, then the subtree left_T must be *optimal* for the elements accessing it (i.e., $A[1 \ldots r-1]$ where r is the root).

Thus, the price of $\mathcal{T}$ is

$$S(\mathcal{T}) \quad = \quad S(\text{left}_T) + S(\text{right}_T) + \sum_{i=1}^{n} \mathsf{f}[i],$$

where $S(Q)$ is the price of searching in Q for the frequency of elements stored in Q.

This recursive formula naturally gives rise to a recursive algorithm, which is depicted on the right. The naive implementation requires $O(n^2)$ time (ignoring the recursive call). But in fact, by a more careful implementation, together with the tree $\mathcal{T}$, we can also return the price of searching on this tree with the given frequencies. Thus, this modified algorithm. Thus, the running time for this function takes $O(n)$ time (ignoring recursive calls). The running time of the resulting algorithm is

```
CompBestTreeI( A[i...j], f[i...j] )
  for r = i...j do
      T_left  ← CompBestTreeI(A[i...r-1], f[i...r-1])
      T_right ← CompBestTreeI(A[r+1...j], f[r+1...j])
      T_r ← Tree(T_left , A[r] , T_right)
      P_r ← S(T_r)

  return cheapest tree out of T_i, ..., T_j.

CompBestTree( A[1...n], f[1...n] )
      return CompBestTreeI( A[1...n], f[1...n])
```

$$\alpha(n) = O(n) + \sum_{i=0}^{n-1} (\alpha(i) + \alpha(n-i-1)),$$

and the solution of this recurrence is $O(n3^n)$.

We can, of course, improve the running time using memoization. There are only $O(n^2)$ different recursive calls, and as such, the running time of **CompBestTreeMemoize** is $O(n^2) \cdot O(n) = O(n^3)$.

Theorem 5.1.2. *Can can compute the optimal binary search tree in $O(n^3)$ time using $O(n^2)$ space.*

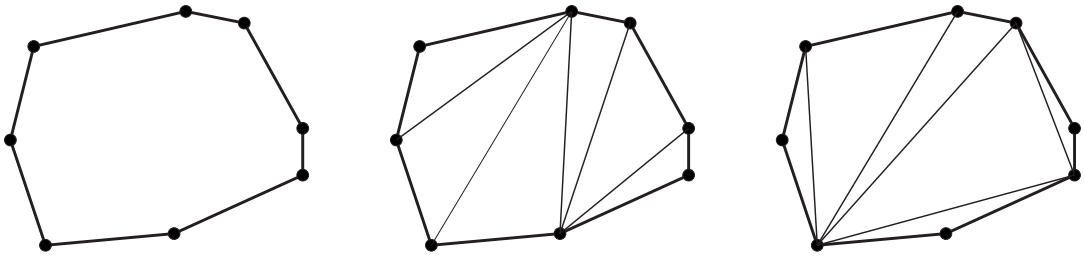


Figure 5.1: A polygon and two possible triangulations of the polygon.

A further improvement raises from the fact that the root location is "monotone". Formally, if $R[i, j]$ denotes the location of the element stored in the root for the elements $A[i \ldots j]$ then it holds that $R[i, j-1] \leq R[i, j] \leq R[i, j+1]$. This limits the search space, and we can be more efficient in the search. This leads to $O(n^2)$ algorithm. Details are in Jeff Erickson class notes.

5.2 Optimal Triangulations

Given a convex polygon P in the plane, we would like to find the triangulation of P of minimum total length. Namely, the total length of the diagonals of the triangulation of P, plus the (length of the) perimeter of P are minimized. See Figure 5.1.

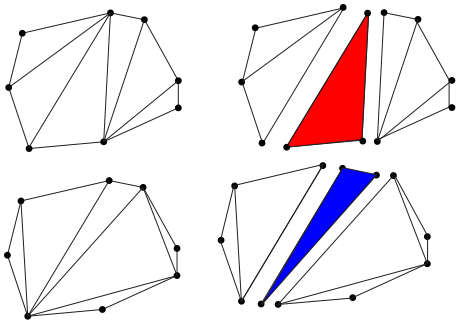
Definition 5.2.1. A set $S \subseteq \mathbb{R}^d$ is ***convex*** if for any to $x, y \in S$, the segment xy is contained in S.

A ***convex polygon*** is a closed cycle of segments, with no vertex pointing inward. Formally, it is a simple closed polygonal curve which encloses a convex set.

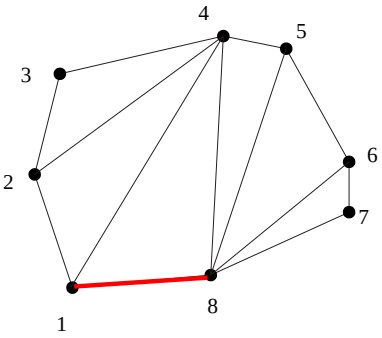
A ***diagonal*** is a line segment connecting two vertices of a polygon which are not adjacent. A *triangulation* is a partition of a convex polygon into (interior) disjoint triangles using diagonals.

Observation 5.2.2. *Any triangulation of a convex polygon with n vertices is made out of exactly $n-2$ triangles.*

Our purpose is to find the triangulation of P that has the minimum total length. Namely, the total length of diagonals used in the triangulation is minimized. We would like to compute the optimal triangulation using divide and conquer. As the figure on the right demonstrate, there is always a triangle in the triangulation, that breaks the polygon into two polygons. Thus, we can try and guess such a triangle in the optimal triangulation, and recurse on the two polygons such created. The only



difficulty, is to do this in such a way that the recursive subproblems can be described in succinct way.



To this end, we assume that the polygon is specified as list of vertices $1 \ldots n$ in a clockwise ordering. Namely, the input is a list of the vertices of the polygon, for every vertex, the two coordinates are specified. The key observation, is that in any triangulation of P, there exist a triangle that uses the edge between vertex 1 and n (red edge in figure on the left).

In particular, removing the triangle using the edge $1 - n$ leaves us with two polygons which their vertices are *consecutive* along the original polygon.

Let $M[i, j]$ denote the price of triangulating a polygon starting at vertex i and ending at vertex j, where every diagonal used contributes its length twice to this quantity, and the perimeter edges contribute their length exactly once. We have the following "natural" recurrence:

$$M[i,j] = \begin{cases} 0 & j \leq i \\ 0 & j = i+1 \\ \min_{i<k<j}(\Delta(i,j,k) + M[i,k] + M[k,j]) & Otherwise \end{cases}.$$

Where $Dist(i,j) = \sqrt{(x[i]-x[j])^2 + (y[i]-y[j])^2}$ and $\Delta(i,j,k) = Dist(i,j) + Dist(j,k) + Dist(i,k)$, where the ith point has coordinates $(x[i], y[i])$, for $i = 1, \ldots, n$. Note, that the quantity we are interested in is $M[1, n]$, since it the triangulation of P with minimum total weight.

Using dynamic programming (or just memoization), we get an algorithm that computes optimal triangulation in $O(n^3)$ time using $O(n^2)$ space.

5.3 Matrix Multiplication

We are given two matrix: (i) A is a matrix with dimensions $p \times q$ (i.e., p rows and q columns) and (ii) B is a matrix of size $q \times r$. The product matrix AB, with dimensions $p \times r$, can be computed in $O(pqr)$ time using the standard algorithm.

A	1000×2
B	2×1000
C	1000×2

Things becomes considerably more interesting when we have to multiply a chain for matrices. Consider for example the three matrices A, B and C with dimensions as listed on the left. Computing the matrix $ABC = A(BC)$ requires $2 \cdot 1000 \cdot 2 + 1000 \cdot 2 \cdot 2 = 8,000$ operations. On the other hand, computing the same matrix using $(AB)C$ requires $1000 \cdot 2 \cdot 1000 + 1000 \cdot 1000 \cdot 2 = 4,000,000$. Note, that matrix multiplication is associative, and as such $(AB)C = A(BC)$.

Thus, given a chain of matrices that we need to multiply, the exact ordering in which we do the multiplication matters as far to multiply the order is important as far as efficiency.

Problem 5.3.1. The input is n matrices $M_1, \ldots, M_n$ such that M_i is of size $D[i-1] \times D[i]$ (i.e., M_i has $D[i-1]$ rows and $D[i]$ columns), where $D[0 \ldots n]$ is array specifying the sizes. Find the ordering of multiplications to compute $M_1 \cdot M_2 \cdots M_{n-1} \cdot M_n$ most efficiently.

Again, let us define a recurrence for this problem, where $M[i,j]$ is the amount of work involved in computing the product of the matrices $M_i \cdots M_j$. We have

$$M[i,j] = \begin{cases} 0 & j = i \\ D[i-1] \cdot D[i] \cdot D[i+1] & j = i+1 \\ \min_{i \le k < j} (M[i,k] + M[k+1,j] + D[i-1] \cdot D[k] \cdot D[j]) & j > i+1. \end{cases}$$

Again, using memoization (or dynamic programming), one can compute $M[1,n]$, in $O(n^3)$ time, using $O(n^2)$ space.

5.4 Longest Ascending Subsequence

Given an array of numbers $A[1 \ldots n]$ we are interested in finding the ***longest ascending subsequence***. For example, if $A = [6,3,2,5,1,12]$ the longest ascending subsequence is $2,5,12$. To this end, let $M[i]$ denote longest increasing subsequence having $A[i]$ as the last element in the subsequence. The recurrence on the maximum possible length, is

$$M[n] = \begin{cases} 1 & n = 1 \\ 1 + \max\limits_{1 \le k < n, A[k] < A[n]} M[k] & \text{otherwise.} \end{cases}$$

The length of the longest increasing subsequence is $\max_{i=1}^{n} M[i]$. Again, using dynamic programming, we get an algorithm with running time $O(n^2)$ for this problem. It is also not hard to modify the algorithm so that it outputs this sequence (you should figure out the details of this modification). A better $O(n \log n)$ solution is possible using some data-structure magic.

5.5 Pattern Matching

Assume you have a string $S =$ "$Magna\ Carta$" and a pattern $P =$ "$?ag * at * a *$" where "?" can match a single character, and "*" can match any substring. You would like to decide if the pattern matches the string.

We are interested in solving this problem using dynamic programming. This is not too hard since this is similar to the edit-distance problem that was already covered.

Tidbit

Magna Carta or *Magna Charta* - the great charter that King John of England was forced by the English barons to grant at Runnymede, June 15, 1215, traditionally interpreted as guaranteeing certain civil and political liberties.

```
IsMatch(S[1...n], P[1...m])
  if m = 0 and n = 0 then return TRUE.
  if m = 0 then return FALSE.
  if n = 0 then
      if P[1...m] is all stars then return TRUE
      else return FALSE
  if (P[m] = '?') then
      return IsMatch(S[1...n-1], P[1...m-1])
  if (P[m] ≠ '*') then
      if P[m] ≠ S[n] then return FALSE
      else return IsMatch(S[1...n-1], P[1...m-1])
  for i = 0 to n do
      if IsMatch(S[1...i], P[1...m-1]) then
          return TRUE
  return FALSE
```

The resulting code is depicted on the left, and as you can see this is pretty tedious. Now, use memoization together with this recursive code, and you get an algorithm with running time $O(mn^2)$ and space $O(nm)$, where the input string of length n, and m is the length of the pattern.

Being slightly more clever, one can get a faster algorithm with running time $O(nm)$.

BTW, one can do even better. A $O(m+n)$ time is possible but it requires Knuth-Morris-Pratt algorithm, which is a fast string matching algorithm.

Chapter 6

Approximation algorithms

6.1 Greedy algorithms and approximation algorithms

A natural tendency in solving algorithmic problems is to locally do whats seems to be the right thing. This is usually referred to as ***greedy algorithms***. The problem is that usually these kind of algorithms do not really work. For example, consider the following optimization version of Vertex Cover:

VertexCoverMin

Instance: A graph G, and integer k.
Question: Return the smallest subset $S \subseteq V(\mathsf{G})$, s.t. S touches all the edges of G.

For this problem, the greedy algorithm will always take the vertex with the highest degree (i.e., the one covering the largest number of edges), add it to the cover set, remove it from the graph, and repeat. We will refer to this algorithm as **GreedyVertexCover**.

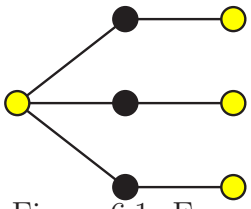


Figure 6.1: Example.

It is not too hard to see that this algorithm does not output the optimal vertex-cover. Indeed, consider the graph depicted on the right. Clearly, the optimal solution is the black vertices, but the greedy algorithm would pick the four white vertices.

This of course still leaves open the possibility that, while we do not get the optimal vertex cover, what we get is a vertex cover which is "relatively good" (or "good enough").

Definition 6.1.1. A ***minimization problem*** is an optimization problem, where we look for a valid solution that minimizes a certain target function.

Example 6.1.2. In the VertexCoverMin problem the (minimization) target function is the size of the cover. Formally $\mathrm{Opt}(\mathsf{G}) = \min_{S \subseteq V(\mathsf{G}), S \text{ cover of } G} |S|$.

The VertexCover(G) is just the set S realizing this minimum.

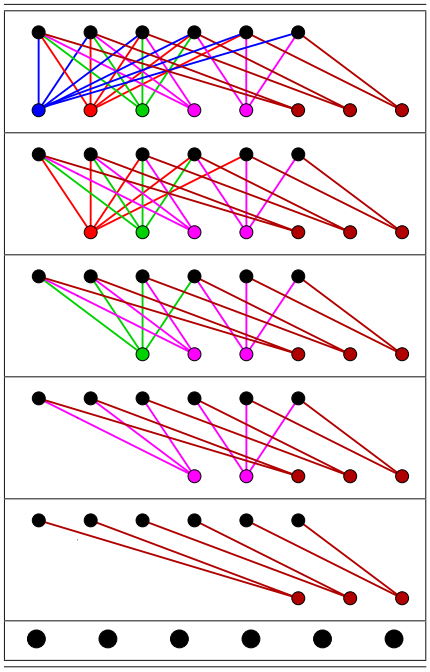
Definition 6.1.3. Let $\mathrm{Opt}(\mathsf{G})$ denote the value of the target function for the optimal solution.

Intuitively, a vertex-cover of size "close" to the optimal solution would be considered to be good.

Definition 6.1.4. Algorithm **Alg** for a minimization problem Min achieves an approximation factor $\alpha \geq 1$ if for all inputs G, we have:

$$\frac{\mathbf{Alg}(\mathsf{G})}{\mathrm{Opt}(\mathsf{G})} \leq \alpha.$$

We will refer to **Alg** as an ***α-approximation algorithm*** for Min.



As a concrete example, an algorithm is a 2-approximation for VertexCoverMin, if it outputs a vertex-cover which is at most twice the size of the optimal solution for vertex cover.

So, how good (or bad) is the **GreedyVertexCover** algorithm described above? Well, the graph in Figure 6.1 shows that the approximation factor of **GreedyVertexCover** is *at least* $4/3$.

It turns out that **GreedyVertexCover** performance is considerably worse. To this end, consider the following bipartite graph: $G_n = (L \cup R, E)$, where L is a set of n vertices. Next, for $i = 2, \ldots, n$, we add a set R_i of $\lfloor n/i \rfloor$ vertices, to R, each one of them of degree i, such that all of them (i.e., all vertices of degree i at L) are connected to distinct vertices in R. The execution of **GreedyVertexCover** on such a graph is shown on the right.

Clearly, in G_n all the vertices in L have degree at most $n-1$, since they are connected to (at most) one vertex of R_i, for $i = 2, \ldots, n$. On the other hand, there is a vertex of degree n at R (i.e., the single vertex of R_n). Thus, **GreedyVertexCover** will first remove this vertex. We claim, that **GreedyVertexCover** will remove all the vertices of $R_2, \ldots, R_n$ and put them into the vertex-cover. To see that, observe that if $R_2, \ldots, R_i$ are still active, then all the nodes of R_i have degree i, all the vertices of L have degree at most $i-1$, and all the vertices of $R_2, \ldots, R_{i-1}$ have degree strictly smaller than i. As such, the greedy algorithms will use the vertices of R_i. Easy induction now implies that all the vertices of R are going to be picked by **GreedyVertexCover**. This implies the following lemma.

Lemma 6.1.5. *The algorithm* **GreedyVertexCover** *is* $\Omega(\log n)$ *approximation to the optimal solution to* VertexCoverMin.

Proof: Consider the graph G_n above. The optimal solution is to pick all the vertices of L to the vertex cover, which results in a cover of size n. On the other hand, the greedy algorithm picks the set R. We have that

$$|R| = \sum_{i=2}^{n} |R_i| = \sum_{i=2}^{n} \left\lfloor \frac{n}{i} \right\rfloor \geq \sum_{i=2}^{n} \left(\frac{n}{i} - 1 \right) \geq n \sum_{i=1}^{n} \frac{1}{i} - 2n = n(H_n - 2) \,.$$

Here, $H_n = \sum_{i=1}^{n} 1/i = \lg n + \Theta(1)$ is the nth harmonic number. As such, the approximation ratio for **GreedyVertexCover** is $\geq \dfrac{|R|}{|L|} = \dfrac{n(H_n - 2)}{n} = \Omega(\log n)$. ■

Theorem 6.1.6. *The greedy algorithm for VertexCover achieves $\Theta(\log n)$ approximation, where n is the number of vertices in the graph. Its running time is $O(mn^2)$.*

Proof: The lower bound follows from Lemma 6.1.5. The upper bound follows from the analysis of the greedy of Set Cover, which will be done shortly.

As for the running time, each iteration of the algorithm takes $O(mn)$ time, and there are at most n iterations. ■

6.1.1 Alternative algorithm – two for the price of one

One can still do much better than the greedy algorithm in this case. In particular, let **ApproxVertexCover** be the algorithm that chooses an edge from G, add both endpoints to the vertex cover, and removes the two vertices (and all the edges adjacent to these two vertices) from G. This process is repeated till G has no edges. Clearly, the resulting set of vertices is a vertex-cover, since the algorithm removes an edge only if it is being covered by the generated cover.

Theorem 6.1.7. **ApproxVertexCover** *is a 2-approximation algorithm for VertexCoverMin that runs in $O(n^2)$ time.*

Proof: Every edge picked by the algorithm contains at least one vertex of the optimal solution. As such, the cover generated is at most twice larger than the optimal. ■

6.2 Fixed parameter tractability, approximation, and fast exponential time algorithms (to say nothing of the dog)

6.2.1 A silly brute force algorithm for vertex cover

So given a graph $\mathsf{G} = (\mathsf{V}, \mathsf{E})$ with n vertices, we can approximate VertexCoverMin up to a factor of two in polynomial time. Let K be this approximation – we know that any vertex cover in G must be of size at least $K/2$, and we have a cover of size K. Imagine the case

```
fpVertexCoverInner (X, β)
        // Computes minimum vertex cover for the induced graph G_X
        // β: size of VC computed so far.
    if X = ∅ or G_X has no edges then return β
    e ← any edge uv of G_X.
    β_1 = fpVertexCoverInner (X \ {u, v}, β + 2)
    // Only take u to the cover, but then we must also take
    //  all the vertices that are neighbors of v,
    //  to cover their edges with v
    β_2 = fpVertexCoverInner (X \ ({u} ∪ N_{G_X}(v)), β + |N_{G_X}(v)|)
    // Only take v to the cover...
    β_3 = fpVertexCoverInner (X \ ({v} ∪ N_{G_X}(u)), β + |N_{G_X}(u)|)
    return min(β_1, β_2, β_3).

algFPVertexCover (G = (V, E))
    return fpVertexCoverInner (V, 0)
```

Figure 6.2: Fixed parameter tractable algorithm for VertexCoverMin.

that K is truly small – can we compute the optimal vertex-cover in this case quickly? Well, of course, we could just try all possible subsets of vertices size at most K, and check for each one whether it is a cover or not. Checking if a specific set of vertices is a cover takes $O(m) = O(n^2)$ time, where $m = |\mathsf{E}|$. So, the running time of this algorithm is

$$\sum_{i=1}^{K} \binom{n}{i} O\left(n^2\right) =\leq \sum_{i=1}^{K} O\left(n^i \cdot n^2\right) = O\left(n^{K+2}\right),$$

where $\binom{n}{i}$ is the number of subsets of the vertices of G of size exactly i. Observe that we do not require to know K – the algorithm can just try all sizes of subsets, till it finds a solution. We thus get the following (not very interesting result).

Lemma 6.2.1. *Given a graph $\mathsf{G} = (\mathsf{V}, \mathsf{E})$ with n vertices, one can solve VertexCoverMin in $O(n^{\alpha+2})$ time, where α is the size the minimum vertex cover.*

6.2.2 A fixed parameter tractable algorithm

As before, our input is a graph $\mathsf{G} = (\mathsf{V}, \mathsf{E})$, for which we want to compute a vertex-cover of minimum size. We need the following definition:

Definition 6.2.2. Let $\mathsf{G} = (\mathsf{V}, \mathsf{E})$ be a graph. For a subset $S \subseteq \mathsf{V}$, let G_S be the ***induced subgraph*** over S. Namely, it is a graph with the set of vertices being S. For any pair of vertices $x, y \in \mathsf{V}$, we have that the edge $xy \in \mathsf{E}(\mathsf{G}_S)$ if and only if $xy \in \mathsf{E}(\mathsf{G})$, and $x, y \in S$.

Also, in the following, for a vertex v, let $N_{\mathsf{G}}(v)$ denote the set of vertices of G that are adjacent to v.

Consider an edge $\mathsf{e} = uv$ in G. We know that either u or v (or both) must be in any vertex cover of G, so consider the brute force algorithm for VertexCoverMin that tries all these possibilities. The resulting algorithm **algFPVertexCover** is depicted in Figure 6.2.

Lemma 6.2.3. *The algorithm* **algFPVertexCover** *(depicted in Figure 6.2) returns the optimal solution to the given instance of VertexCoverMin.*

Proof: It is easy to verify, that if the algorithm returns β then it found a vertex cover of size β. Since the depth of the recursion is at most n, it follows that this algorithm always terminates.

Consider the optimal solution $Y \subseteq \mathsf{V}$, and run the algorithm, where every stage of the recursion always pick the option that complies with the optimal solution. Clearly, since in every level of the recursion at least one vertex of Y is being found, then after $O(|Y|)$ recursive calls, the remaining graph would have no edges, and it would return $|Y|$ as one of the candidate solution. Furthermore, since the algorithm always returns the minimum solution encountered, it follows that it would return the optimal solution. ∎

Lemma 6.2.4. *The depth of the recursion of* **algFPVertexCover**(G) *is at most α, where α is the minimum size vertex cover in G.*

Proof: The idea is to consider all the vertices that can be added to the vertex cover being computed without covering any new edge. In particular, in the case the algorithm takes both u and v to the cover, then one of these vertices must be in the optimal solution, and this can happen at most α times.

The more interesting case, is when the algorithm picks $N_{\mathsf{G}_X}(v)$ (i.e., β_2) to the vertex cover. We can add v to the vertex cover in this case without getting any new edges being covered (again, we are doing this only conceptually – the vertex cover computed by the algorithm would not contain v [only its neighbors]). We do the same thing for the case of β_3.

Now, observe that in any of these cases, the hypothetical set cover being constructed (which has more vertices than what the algorithm computes, but covers exactly the same set of edges in the original graph) contains one vertex of the optimal solution picked into itself in each level of the recursion. Clearly, the algorithm is done once we pick all the vertices of the optimal solution into the hypothetical vertex cover. It follows that the depth the recursion is $\leq \alpha$. ∎

Theorem 6.2.5. *Let G be a graph with n vertices, and with the minimal vertex cover being of size α. Then, the algorithm* **algFPVertexCover** *(depicted in Figure 6.2) returns the optimal vertex cover for G and the running time of this algorithm is $O(3^{\alpha} n^2)$.*

Proof: By Lemma 6.2.4, the recursion tree has depth α. As such, it contains at most $2 \cdot 3^{\alpha}$ nodes. Each node in the recursion requires $O(n^2)$ work (ignoring the recursive calls), if implemented naively. Thus, the bound on the running time follows. ∎

Algorithms where the running time is of the form $O(n^c f(\alpha))$, where α is some parameter that depends on the problem are ***fixed parameter tractable*** algorithms for the given problem.

6.2.2.1 Remarks

Currently, the fastest algorithm known for this problem has running time $O(1.2738^{\alpha} + \alpha n)$ [CKX10]. This algorithm uses similar ideas, but is considerably more complicated.

It is known that no better approximation than 1.3606 is possible for VertexCoverMin, unless P = NP. The currently best approximation known is $2 - \Theta\left(1/\sqrt{\log n}\right)$. If the ***Unique Games Conjecture*** is true, then no better constant approximation is possible in polynomial time.

6.3 Traveling Salesman Person

We remind the reader that the optimization variant of the TSP problem is the following.

TSP-Min

Instance: $\mathsf{G} = (V, E)$ a complete graph, and $\omega(e)$ a cost function on edges of G.
Question: The cheapest tour that visits all the vertices of G exactly once.

Theorem 6.3.1. *TSP-Min can not be approximated within **any** factor unless* NP = P.

Proof: Consider the reduction from Hamiltonian Cycle into TSP. Given a graph G, which is the input for the Hamiltonian cycle, we transform it into an instance of TSP-Min. Specifically, we set the weight of every edge to 1 if it was present in the instance of the Hamiltonian cycle, and 2 otherwise. In the resulting complete graph, if there is a tour price n then there is a Hamiltonian cycle in the original graph. If on the other hand, there was no cycle in G then the cheapest TSP is of price $n + 1$.

Instead of 2, let us assign the missing edges in H a weight of cn, for c an arbitrary number. Let H denote the resulting graph. Clearly, if G does not contain any Hamiltonian cycle in the original graph, then the price of the TSP-Min in H is at least $cn + 1$.

Note, that the prices of tours of H are either (i) equal to n if there is a Hamiltonian cycle in G, or (ii) larger than $cn + 1$ if there is no Hamiltonian cycle in G. As such, if one can do a c-approximation, in polynomial time, to TSP-Min, then using it on H would yield a tour of price $\leq cn$ if a tour of price n exists. But a tour of price $\leq cn$ exists if and only if G has a Hamiltonian cycle.

Namely, such an approximation algorithm would solve a NP-Complete problem (i.e., Hamiltonian Cycle) in polynomial time. ■

Note, that Theorem 6.3.1 implies that TSP-Min can not be approximated to within any factor. However, once we add some assumptions to the problem, it becomes much more manageable (at least as far as approximation).

What the above reduction did, was to take a problem and reduce it into an instance where this is a huge gap, between the optimal solution, and the second cheapest solution. Next, we argued that if had an approximation algorithm that has ratio better than the ratio between the two endpoints of this empty interval, then the approximation algorithm, would in polynomial time would be able to decide if there is an optimal solution.

6.3.1 TSP with the triangle inequality

6.3.1.1 A 2-approximation

Consider the following special case of TSP:

$\mathsf{TSP}_{\triangle\neq}$-Min

Instance: $\mathsf{G} = (V, E)$ is a complete graph. There is also a cost function $\omega(\cdot)$ defined over the edges of G, that complies with the triangle inequality.
Question: The cheapest tour that visits all the vertices of G exactly once.

We remind the reader that the ***triangle inequality*** holds for $\omega(\cdot)$ if

$$\forall u, v, w \in \mathsf{V}(\mathsf{G}), \qquad \omega(u, v) \leq \omega(u, w) + \omega(w, v).$$

The triangle inequality implies that if we have a path σ in G, that starts at s and ends at t, then $\omega(st) \leq \omega(\sigma)$. Namely, ***shortcutting***, that is going directly from s to t, is always beneficial if the triangle inequality holds (assuming that we do not have any reason to visit the other vertices of σ).

Definition 6.3.2. A cycle in a graph G is ***Eulerian*** if it visits every edge of G exactly once.

Unlike Hamiltonian cycle, which has to visit every vertex exactly once, an Eulerian cycle might visit a vertex an arbitrary number of times. We need the following classical result:

Lemma 6.3.3. *A graph G has a cycle that visits every edge of G exactly once (i.e., an Eulerian cycle) if and only if G is connected, and all the vertices have even degree. Such a cycle can be computed in $O(n + m)$ time, where n and m are the number of vertices and edges of G, respectively.*

Our purpose is to come up with a 2-approximation algorithm for $\mathsf{TSP}_{\triangle\neq}$-Min. To this end, let C_{opt} denote the optimal TSP tour in G. Observe that C_{opt} is a spanning graph of G, and as such we have that

$$\omega(C_{\mathrm{opt}}) \geq \text{weight}\Big(\text{cheapest spanning graph of } \mathsf{G}\Big).$$

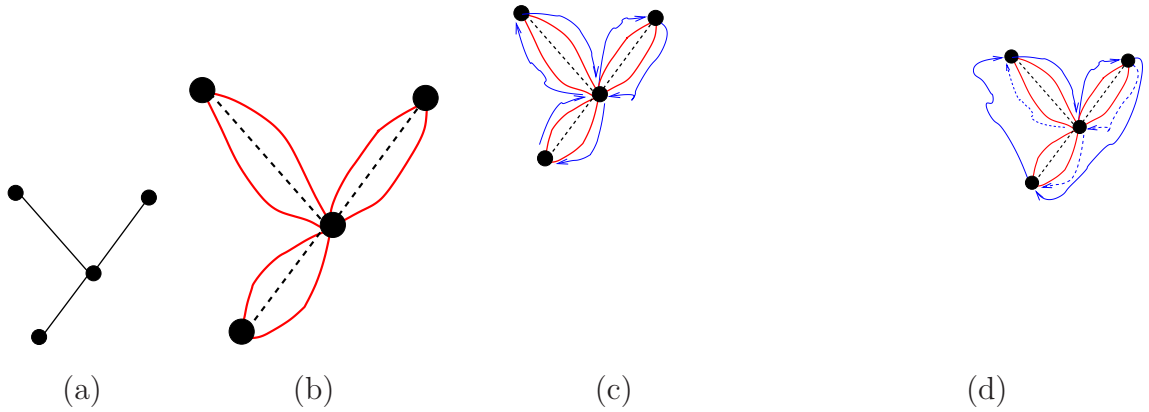


Figure 6.3: The TSP approximation algorithm: (a) the input, (b) the duplicated graph, (c) the extracted Eulerian tour, and (d) the resulting shortcut path.

But the cheapest spanning graph of G, is the minimum spanning tree (MST) of G, and as such $\omega(C_{\text{opt}}) \geq \omega(\text{MST}(\mathsf{G}))$. The MST can be computed in $O(n \log n + m) = O(n^2)$ time, where n is the number of vertices of G, and $m = \binom{n}{2}$ is the number of edges (since G is the complete graph). Let T denote the MST of G, and covert T into a tour by duplicating every edge twice. Let H denote the new graph. We have that H is a connected graph, every vertex of H has even degree, and as such H has an Eulerian tour (i.e., a tour that visits every edge of H exactly once).

As such, let C denote the Eulerian cycle in H. Observe that

$$\omega(\mathsf{C}) = \omega(H) = 2\omega(T) = 2\omega(MST(\mathsf{G})) \leq 2\omega(C_{\text{opt}}) .$$

Next, we traverse C starting from any vertex $v \in V(\mathsf{C})$. As we traverse C, we skip vertices that we already visited, and in particular, the new tour we extract from C will visit the vertices of $V(\mathsf{G})$ in the order they first appear in C. Let π denote the new tour of G. Clearly, since we are performing shortcutting, and the triangle inequality holds, we have that $\omega(\pi) \leq \omega(\mathsf{C})$. The resulting algorithm is depicted in Figure 6.3.

It is easy to verify, that all the steps of our algorithm can be done in polynomial time. As such, we have the following result.

Theorem 6.3.4. *Given an instance of TSP with the triangle inequality (TSP$_{\triangle\neq}$-Min) (namely, a graph G with n vertices and $\binom{n}{2}$ edges, and a cost function $\omega(\cdot)$ on the edges that comply with the triangle inequality), one can compute a tour of G of length $\leq 2\omega(C_{\text{opt}})$, where C_{opt} is the minimum cost TSP tour of G. The running time of the algorithm is $O(n^2)$.*

6.3.1.2 A 3/2-approximation to TSP$_{\triangle\neq}$-Min

Let us revisit the concept of matchings.

Definition 6.3.5. Given a graph $\mathsf{G} = (V, E)$, a subset $M \subseteq E$ is a ***matching*** if no pair of edges of M share endpoints. A ***perfect matching*** is a matching that covers all the vertices of G. Given a weight function w on the edges, a ***min-weight perfect matching***, is the minimum weight matching among all perfect matching, where

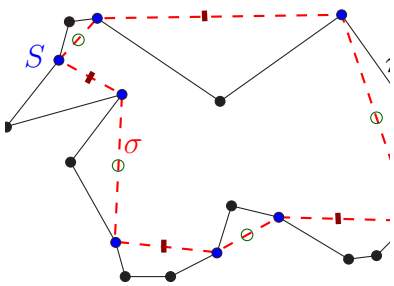
$$\omega(M) = \sum_{e \in M} \omega(e) .$$

The following is a known result, and we will see a somewhat weaker version of it in class.

Theorem 6.3.6. *Given a graph G and weights on the edges, one can compute the min-weight perfect matching of G in polynomial time.*

Lemma 6.3.7. *Let $\mathsf{G} = (V, E)$ be a complete graph, S a subset of the vertices of V of even size, and $\omega(\cdot)$ a weight function over the edges. Then, the weight of the min-weight perfect matching in G_S is $\leq \omega(\mathrm{TSP}(\mathsf{G}))/2$.*

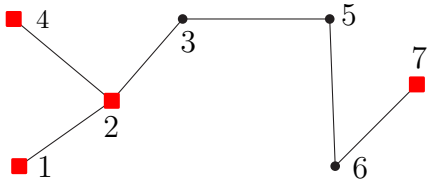
Proof: Let π be the cycle realizing the TSP in G. Let σ be the cycle resulting from shortcutting π so that it uses only the vertices of S. Clearly, $\omega(\sigma) \leq \omega(\pi)$. Now, let M_e and M_o be the sets of even and odd edges of σ respectively. Clearly, both M_o and M_e are perfect matching in G_S, and



$$\omega(M_o) + \omega(M_e) = \omega(\sigma) .$$

We conclude, that $min(w(M_o), w(M_e)) \leq \omega(\mathrm{TSP}(\mathsf{G}))/2$. ■

We now have a creature that has the weight of half of the TSP, and we can compute it in polynomial time. How to use it to approximate the TSP? The idea is that we can make the MST of G into an Eulerian graph by being more careful. To this end, consider the tree on the right. Clearly, it is almost Eulerian, except for these pesky odd degree vertices. Indeed, if all the vertices of the spanning tree had even degree, then the graph would be Eulerian (see Lemma 6.3.3).



In particular, in the depicted tree, the "problematic" vertices are $1, 4, 2, 7$, since they are all the odd degree vertices in the MST T.

Lemma 6.3.8. *The number of odd degree vertices in any graph G' is even.*

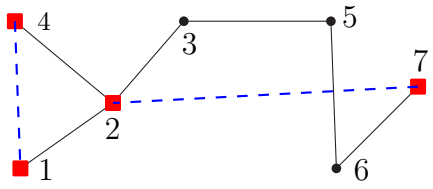
Proof: Observe that $\mu = \sum_{v \in V(G')} d(v) = 2|E(G')|$, where $d(v)$ denotes the degree of v. Let $U = \sum_{v \in V(G'), d(v) \text{ is even}} d(v)$, and observe that U is even as it is the sum of even numbers.

Thus, ignoring vertices of even degree, we have

$$\alpha = \sum_{v \in V, d(v) \text{ is odd}} d(v) = \mu - U = \text{even number},$$

since μ and U are both even. Thus, the number of elements in the above sum of all odd numbers must be even, since the total sum is even. ■

So, we have an even number of problematic vertices in T. The idea now is to compute a minimum-weight perfect matching M on the problematic vertices, and add the edges of the matching to the tree. The resulting graph, for our running example, is depicted on the right.



Let $H = (V, E(M) \cup E(T))$ denote this graph, which is the result of adding M to T.

We observe that H is Eulerian, as all the vertices now have even degree, and the graph is connected. We also have

$$\omega(H) \quad = \quad \omega(\mathrm{MST}(\mathsf{G})) + \omega(M) \leq \omega(\mathrm{TSP}(\mathsf{G})) + \omega(\mathrm{TSP}(\mathsf{G}))/2 = (3/2)\omega(TSP(\mathsf{G})),$$

by Lemma 6.3.7. Now, H is Eulerian, and one can compute the Euler cycle for H, shortcut it, and get a tour of the vertices of G of weight $\leq (3/2)\omega(\mathrm{TSP}(\mathsf{G}))$.

Theorem 6.3.9. *Given an instance of TSP with the triangle inequality, one can compute in polynomial time, a* $(3/2)$*-approximation to the optimal TSP.*

6.4 Biographical Notes

The 3/2-approximation for TSP with the triangle inequality is due to Christofides [Chr76].

Chapter 7

Approximation algorithms II

7.1 Max Exact 3SAT

We remind the reader that an instance of 3SAT is a boolean formula, for example $F = (x_1 + x_2 + x_3)(x_4 + \overline{x_1} + x_2)$, and the decision problem is to decide if the formula has a satisfiable assignment. Interestingly, we can turn this into an optimization problem.

Max 3SAT

Instance: A collection of clauses: $C_1, \ldots, C_m$.
Question: Find the assignment to $x_1, ..., x_n$ that satisfies the maximum number of clauses.

Clearly, since 3SAT is NP-Complete it implies that Max 3SAT is NP-Hard. In particular, the formula F becomes the following set of two clauses:

$$x_1 + x_2 + x_3 \qquad \text{and} \qquad x_4 + \overline{x_1} + x_2.$$

Note, that Max 3SAT is a ***maximization problem***.

Definition 7.1.1. Algorithm **Alg** for a maximization problem achieves an approximation factor α if for all inputs, we have:

$$\frac{\mathbf{Alg}(G)}{\mathrm{Opt}(G)} \geq \alpha.$$

In the following, we present a ***randomized algorithm*** – it is allowed to consult with a source of random numbers in making decisions. A key property we need about random variables, is the linearity of expectation property, which is easy to derive directly from the definition of expectation.

Definition 7.1.2 (*Linearity of expectations*.). Given two random variables X, Y (not necessarily independent, we have that $\mathbf{E}\Big[X + Y\Big] = \mathbf{E}\Big[X\Big] + \mathbf{E}\Big[Y\Big]$.

Theorem 7.1.3. *One can achieve (in expectation)* $(7/8)$*-approximation to Max 3SAT in polynomial time. Namely, if the instance has* m *clauses, then the generated assignment satisfies* $(7/8)m$ *clauses in* expectation.

Proof: Let $x_1, \ldots, x_n$ be the n variables used in the given instance. The algorithm works by randomly assigning values to $x_1, \ldots, x_n$, independently, and equal probability, to 0 or 1, for each one of the variables.

Let Y_i be the indicator variables which is 1 if (and only if) the ith clause is satisfied by the generated random assignment and 0 otherwise, for $i = 1, \ldots, m$. Formally, we have

$$Y_i = \begin{cases} 1 & C_i \text{ is satisfied by the generated assignment,} \\ 0 & \text{otherwise.} \end{cases}$$

Now, the number of clauses satisfied by the given assignment is $Y = \sum_{i=1}^{m} Y_i$. We claim that $\mathbf{E}[Y] = (7/8)m$, where m is the number of clauses in the input. Indeed, we have

$$\mathbf{E}\Big[Y\Big] = \mathbf{E}\left[\sum_{i=1}^{m} Y_i\right] = \sum_{i=1}^{m} \mathbf{E}\Big[Y_i\Big]$$

by linearity of expectation. Now, what is the probability that $Y_i = 0$? This is the probability that all three literals appear in the clause C_i are evaluated to FALSE. Since the three literals are instance of three distinct variable, these three events are independent, and as such the probability for this happening is

$$\mathbf{Pr}\Big[Y_i = 0\Big] = \frac{1}{2} * \frac{1}{2} * \frac{1}{2} = \frac{1}{8}.$$

(Another way to see this, is to observe that since C_i has exactly three literals, there is only one possible assignment to the three variables appearing in it, such that the clause evaluates to FALSE. Now, there are eight (8) possible assignments to this clause, and thus the probability of picking a FALSE assignment is 1/8.) Thus,

$$\mathbf{Pr}\Big[Y_i = 1\Big] = 1 - \mathbf{Pr}\Big[Y_i = 0\Big] = \frac{7}{8},$$

and

$$\mathbf{E}\Big[Y_i\Big] = \mathbf{Pr}\Big[Y_i = 0\Big] * 0 + \mathbf{Pr}\Big[Y_i = 1\Big] * 1 = \frac{7}{8}.$$

Namely, $\mathbf{E}[\#\text{ of clauses sat}] = \mathbf{E}[Y] = \sum_{i=1}^{m} \mathbf{E}[Y_i] = (7/8)m$. Since the optimal solution satisfies at most m clauses, the claim follows. ∎

Curiously, Theorem 7.1.3 is stronger than what one usually would be able to get for an approximation algorithm. Here, the approximation quality is independent of how well the

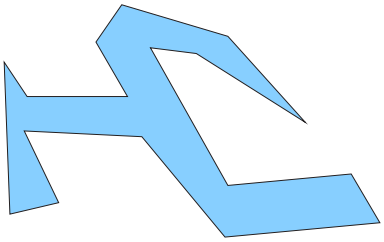
optimal solution does (the optimal can satisfy at most m clauses, as such we get a (7/8)-approximation. Curiouser and curiouser[1], the algorithm does not even look on the input when generating the random assignment.

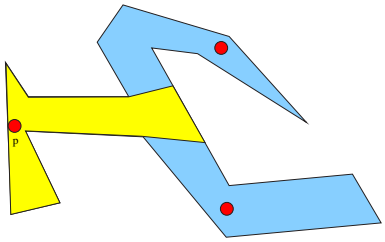
Håstad [Hås01a] proved that one can do no better; that is, for any constant $\varepsilon > 0$, one can not approximate 3SAT in polynomial time (unless P = NP) to within a factor of $7/8+\varepsilon$. It is pretty amazing that a trivial algorithm like the above is essentially optimal.

7.2 Approximation Algorithms for Set Cover

7.2.1 Guarding an Art Gallery

You are given the floor plan of an art gallery, which is a two dimensional simple polygon. You would like to place guards that see the whole polygon. A guard is a point, which can see all points around it, but it can not see through walls. Formally, a point p can ***see*** a point q, if the segment pq is contained inside the polygon. See figure on the right, for an illustration of how the input looks like.





A ***visibility polygon*** at p (depicted as the yellow polygon on the left) is the region inside the polygon that p can see. WE would like to find the *minimal* number of guards needed to guard the given art-gallery? That is, all the points in the art gallery should be visible from at least one guard we place.

The art-gallery problem is a set-cover problem. We have a ground set (the polygon), and family of sets (the set of all visibility polygons), and the target is to find a minimal number of sets covering the whole polygon.

It is known that finding the minimum number of guards needed is NP-Hard. No approximation is currently known. It is also known that a polygon with n corners, can be guarded using $n/3+1$ guards. Note, that this problem is harder than the classical set-cover problem because the number of subsets is infinite and the underlining base set is also infinite.

An interesting *open problem* is to find a polynomial time approximation algorithm, such that given P, it computes a set of guards, such that $\#guards \leq \sqrt{n}k_{opt}$, where n is the number of vertices of the input polygon P, and k_{opt} is the number of guards used by the optimal solution.

7.2.2 Set Cover

The optimization version of Set Cover, is the following:

[1] "Curiouser and curiouser!" Cried Alice (she was so much surprised, that for the moment she quite forgot how to speak good English). – Alice in wonderland, Lewis Carol

Set Cover

Instance: $(\mathsf{S}, \mathcal{F})$:
S - a set of n elements
$\mathcal{F}$ - a family of subsets of S, s.t. $\bigcup_{X \in \mathcal{F}} X = \mathsf{S}$.
Question: The set $\mathcal{X} \subseteq \mathcal{F}$ such that $\mathcal{X}$ contains as few sets as possible, and $\mathcal{X}$ covers S. Formally, $\bigcup_{X \in \mathcal{X}} X = \mathsf{S}$.

The set S is sometime called the ***ground set***, and a pair $(\mathsf{S}, \mathcal{F})$ is either called a ***set system*** or a ***hypergraph***. Note, that Set Cover is a minimization problem which is also NP-Hard.

Example 7.2.1. Consider the set $\mathsf{S} = \{1, 2, 3, 4, 5\}$ and the following family of subsets

$$\mathcal{F} = \Big\{\{1,2,3\},\ \{2,5\},\ \{1,4\},\ \{4,5\}\Big\}.$$

Clearly, the smallest cover of S is $\mathcal{X}_{opt} = \Big\{\{1,2,3\},\ \{4,5\}\Big\}$.

The greedy algorithm **GreedySetCover** for this problem is depicted on the right. Here, the algorithm always picks the set in the family that covers the largest number of elements not covered yet. Clearly, the algorithm is polynomial in the input size. Indeed, we are given a set S of n elements, and m subsets. As such, the input size is at least $\Omega(m+n)$ (and at most of size $O(mn)$), and the algorithm takes time polynomial in m and n. Let $\mathcal{X}_{opt} = \{V_1, \ldots, V_k\}$ be the optimal solution.

GreedySetCover$(\mathsf{S}, \mathcal{F})$
  $\mathcal{X} \leftarrow \emptyset$; $T \leftarrow \mathsf{S}$
  while T is not empty **do**
    $U \leftarrow$ set in $\mathcal{F}$ covering largest # of elements in T
    $\mathcal{X} \leftarrow \mathcal{X} \cup \{U\}$
    $T \leftarrow T \setminus U$
  return $\mathcal{X}$.

Let T_i denote the elements not covered in the beginning ith iteration of **GreedySetCover**, where $T_1 = \mathsf{S}$. Let U_i be the set added to the cover in the ith iteration, and $\alpha_i = |U_i \cap T_i|$ be the number of new elements being covered in the ith iteration.

Claim 7.2.2. *We have $\alpha_1 \geq \alpha_2 \geq \ldots \geq \alpha_k \geq \ldots \geq \alpha_m$.*

Proof: If $\alpha_i < \alpha_{i+1}$ then U_{i+1} covers more elements than U_i and we can exchange between them, as we found a set that in the ith iteration covers more elements that the set used by **GreedySetCover**. Namely, in the ith iteration we would use U_{i+1} instead of U_i. This contradicts the greediness of **GreedySetCover** of choosing the set covering the largest number of elements not covered yet. A contradiction. ■

Claim 7.2.3. *We have $\alpha_i \geq |T_i|/k$. Namely, $|T_{i+1}| \leq (1 - 1/k)\,|T_i|$.*

Proof: Consider the optimal solution. It is made out of k sets and it covers S, and as such it covers $T_i \subseteq S$. This implies that one of the subsets in the optimal solution cover at least $1/k$ fraction of the elements of T_i. Finally, the greedy algorithm picks the set that covers the largest number of elements of T_i. Thus, U_i covers at least $\alpha_i \geq |T_i|/k$ elements.

As for the second claim, we have that $|T_{i+1}| = |T_i| - \alpha_i \leq (1 - 1/k)\,|T_i|$. ■

Theorem 7.2.4. *The algorithm* **GreedySetCover** *generates a cover of S using at most $O(k \log n)$ sets of $\mathcal{F}$, where k is the size of the cover in the optimal solution.*

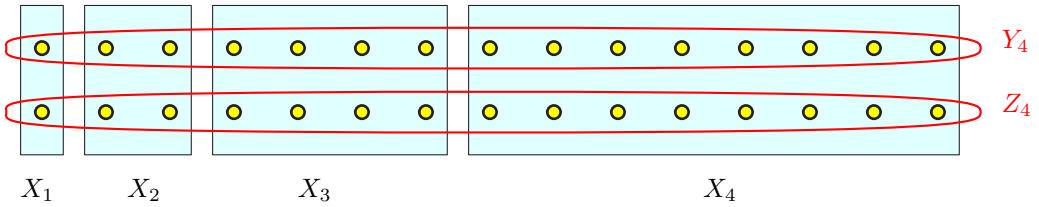
Proof: We have that $|T_i| \leq (1-1/k)\,|T_{i-1}| \leq (1-1/k)^i\,|T_0| = (1-1/k)^i n$. In particular, for $M = \lceil 2k \ln n \rceil$ we have

$$|T_M| \leq \left(1 - \frac{1}{k}\right)^M n \leq \exp\left(-\frac{1}{k} M\right) n = \exp\left(-\frac{\lceil 2k \ln n \rceil}{k}\right) n \leq \frac{1}{n} < 1,$$

since $1 - x \leq e^{-x}$, for $x \geq 0$. Namely, $|T_M| = 0$. As such, the algorithm terminates before reaching the Mth iteration, and as such it outputs a cover of size $O(k \log n)$, as claimed. ■

7.2.3 Lower bound

The lower bound example is depicted in the following figure.



We provide a more formal description of this lower bound next, and prove that it shows $\Omega(\log n)$ approximation to **GreedySetCover**.

We want to show here that the greedy algorithm analysis is tight. To this end, consider the set system $\Lambda_i = (\mathsf{S}_i, \mathcal{F}_i)$, where $\mathsf{S}_i = Y_i \cup Z_i$, $Y_i = \{y_1, \ldots, y_{2^i-1}\}$ and $Z_i = \{z_1, \ldots, z_{2^i-1}\}$. The family of sets $\mathcal{F}_i$ contains the following sets

$$X_j = \left\{y_{2^{j-1}}, \ldots, y_{2^j-1},\ z_{2^{j-1}}, \ldots, z_{2^j-1}\right\},$$

for $j = 1, \ldots, i$. Furthermore, $\mathcal{F}_i$ also contains the two special sets Y_i and Z_i. Clearly, minimum set cover for Λ_i is the two sets Y_i and Z_i.

However, sets Y_i and Z_i have size $2^i - 1$. But, the set X_i has size

$$|X_i| = 2\left(2^i - 1 - 2^{i-1} + 1\right) = 2^i,$$

and this is the largest set in Λ_i. As such, the greedy algorithm **GreedySetCover** would pick X_i as first set to its cover. However, once you remove X_i from Λ_i (and from its ground set), you remain with the set system Λ_{i-1}. We conclude that **GreedySetCover** would pick the sets $X_i, X_{i-1}, \ldots, X_1$ to the cover, while the optimal cover is by two sets. We conclude:

Lemma 7.2.5. *Let $n = 2^{i+1} - 2$. There exists an instance of Set Cover of n elements, for which the optimal cover is by two sets, but* **GreedySetCover** *would use $i = \lfloor \lg n \rfloor$ sets for the cover. That is,* **GreedySetCover** *is a $\Theta(\log n)$ approximation to SetCover.*

7.2.4 Just for fun – weighted set cover

Weighted Set Cover

Instance: $(\mathsf{S}, \mathcal{F}, \rho)$:
S: a set of n elements
$\mathcal{F}$: a family of subsets of S, s.t. $\bigcup_{X \in \mathcal{F}} X = \mathsf{S}$.
$\rho(\cdot)$: A price function assigning price to each set in $\mathcal{F}$.
Question: The set $\mathcal{X} \subseteq \mathcal{F}$, such that $\mathcal{X}$ covers S. Formally, $\bigcup_{X \in \mathcal{X}} X = \mathsf{S}$, and $\rho(\mathcal{X}) = \sum_{X \in \mathcal{X}} \rho(X)$ is minimized.

The greedy algorithm in this case, **WGreedySetCover**, repeatedly picks the set that pays the least cover each element it cover. Specifically, if a set $X \in \mathcal{F}$ covered t new elements, then the ***average price*** it pays per element it cover is $\alpha(X) = \rho(X)/t$. **WGreedySetCover** as such, picks the set with the lowest average price. Our purpose here to prove that this greedy algorithm provides $O(\log n)$ approximation.

7.2.4.1 Analysis

Let U_i be the set of elements that are not covered yet in the end of the ith iteration. As such, $U_0 = \mathsf{S}$. At the beginning of the ith iteration, the ***average optimal cost*** is $\alpha_i = \rho(\mathrm{Opt})/n_i$, where Opt is the optimal solution and $n_i = |U_{i-1}|$ is the number of uncovered elements.

Lemma 7.2.6. *We have that:*
(A) $\alpha_1 \leq \alpha_2 \leq \cdots$.
(B) For $i < j$, *we have* $2\alpha_i \leq \alpha_j$ *only if* $n_j \leq n_i/2$.

Proof: (A) is hopefully obvious – as the number of elements not covered decreases, the average price to cover the remaining elements using the optimal solution goes up.

(B) $2\alpha_i \leq \alpha_j$ implies that $2\rho(\mathrm{Opt})/n_i \leq \rho(\mathrm{Opt})/n_j$, which implies in turn that $2n_j \leq n_i$. ■

So, let k be the first iteration such that $n_k \leq n/2$. The basic idea is that total price that **WGreedySetCover** paid during these iterations is at most $2\rho(\mathrm{Opt})$. This immediately implies $O(\log n)$ iteration, since this can happen at most $O(\log n)$ times till the ground set is fully covered.

To this end, we need the following technical lemma.

Lemma 7.2.7. *Let* U_{i-1} *be the set of elements not yet covered in the beginning of the* i*th iteration, and let* $\alpha_i = \rho(\mathrm{Opt})/n_i$ *be the average optimal cost per element. Then, there exists a set* X *in the optimal solution, with lower average cost; that is,* $\rho(X)/|U_{i-1} \cap X| \leq \alpha_i$.

Proof: Let $X_1, \ldots, X_m$ be the sets used in the optimal solution. Let $s_j = |U_{i-1} \cap X_j|$, for $j = 1, \ldots, m$, be the number of new elements covered by each one of these sets. Similarly,

let $\rho_j = \rho(X_j)$, for $j = 1, \ldots, m$. The average cost of the jth set is ρ_j / s_j (it is $+\infty$ if $s_j = 0$). It is easy to verify that

$$\min_{j=1}^{m} \frac{\rho_j}{s_j} \leq \frac{\sum_{j=1}^{m} \rho_j}{\sum_{j=1}^{m} s_j} = \frac{\rho(\mathrm{Opt})}{\sum_{j=1}^{m} s_j} \leq \frac{\rho(\mathrm{Opt})}{|U_{i-1}|} = \alpha_i.$$

The first inequality follows from the fact that if $a/b \leq c/d$ (all positive numbers), then $\frac{a}{b} \leq \frac{a+c}{b+d} \leq \frac{c}{d}$. In particular, for any such numbers $\min\left(\frac{a}{b}, \frac{c}{d}\right) \leq \frac{a+c}{b+d}$, and applying this repeatedly implies this inequality. The second inequality follows as $\sum_{j=1}^{m} s_j \geq |U_{i-1}|$. This implies that the optimal solution must contain a set with an average cost smaller than the average optimal cost. ■

Lemma 7.2.8. *Let k be the first iteration such that $n_k \leq n/2$. The total price of the sets picked in iteration 1 to $k-1$, is at most $2\rho(\mathrm{Opt})$.*

Proof: By Lemma 7.2.7, at each iteration the algorithm picks a set with average cost that is smaller than the optimal average cost (which goes up in each iteration). However, the optimal average cost iterations, 1 to $k-1$, is at most twice the starting cost, since the number of elements not covered is at least half the total number of elements. It follows, that the for each element covered, the greedy algorithm paid at most twice the initial optimal average cost. So, if the number of elements covered by the beginning of the kth iteration is $\beta \geq n/2$, then the total price paid is $2\alpha_1\beta = 2(\rho(\mathrm{Opt})/n)\beta \leq 2\rho(\mathrm{Opt})$, implying the claim. ■

Theorem 7.2.9. **WGreedySetCover** *computes a $O(\log n)$ approximation to the optimal weighted set cover solution.*

Proof: **WGreedySetCover** paid at most twice the optimal solution to cover half the elements, by Lemma 7.2.8. Now, you can repeat the argument on the remaining uncovered elements. Clearly, after $O(\log n)$ such halving steps, all the sets would be covered. In each halving step, **WGreedySetCover** paid at most twice the optimal cost. ■

7.3 Biographical Notes

The Max 3SAT remains hard in the "easier" variant of MAX 2SAT version, where every clause has 2 variables. It is known to be NP-Hard and approximable within 1.0741 [FG95], and is not approximable within 1.0476 [Hås01a]. Notice, that the fact that MAX 2SAT is hard to approximate is surprising as 2SAT can be solved in polynomial time (!).

Chapter 8

Approximation algorithms III

8.1 Clustering

Consider the problem of ***unsupervised learning***. We are given a set of examples, and we would like to partition them into classes of similar examples. For example, given a webpage X about "The reality dysfunction", one would like to find all webpages on this topic (or closely related topics). Similarly, a webpage about "All quiet on the western front" should be in the same group as webpage as "Storm of steel" (since both are about soldier experiences in World War I).

The hope is that all such webpages of interest would be in the same cluster as X, if the clustering is good.

More formally, the input is a set of examples, usually interpreted as points in high dimensions. For example, given a webpage W, we represent it as a point in high dimensions, by setting the ith coordinate to 1 if the word w_i appears somewhere in the document, where we have a prespecified list of $10,000$ words that we care about. Thus, the webpage W can be interpreted as a point of the $\{0,1\}^{10,000}$ hypercube; namely, a point in $10,000$ dimensions.

Let X be the resulting set of n points in d dimensions.

To be able to partition points into similar clusters, we need to define a notion of similarity. Such a similarity measure can be any distance function between points. For example, consider the "regular" Euclidean distance between points, where

$$\|p-q\| = \sqrt{\sum_{i=1}^{d}(p_i-q_i)^2},$$

where $p=(p_1,\ldots,p_d)$ and $q=(q_1,\ldots,q_d)$.

As another motivating example, consider the ***facility location problem***. We are given a set X of n cities and distances between them, and we would like to build k hospitals, so that the maximum distance of a city from its closest hospital is minimized. (So that the maximum time it would take a patient to get to the its closest hospital is bounded.)

Intuitively, what we are interested in is selecting good representatives for the input point-set X. Namely, we would like to find k points in X such that they represent X "well".

Formally, consider a subset S of k points of X, and a p a point of X. The ***distance of*** p ***from the set*** S is

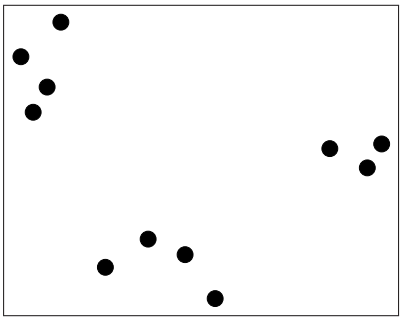
$$\mathbf{d}(p, S) = \min_{q \in S} \|p - q\| ;$$

namely, $\mathbf{d}(p, S)$ is the minimum distance of a point of S to p. If we interpret S as a set of centers then $\mathbf{d}(p, S)$ is the distance of p to its closest center.

Now, the ***price of clustering*** X by the set S is

$$\nu(X, S) = \max_{p \in X} \mathbf{d}(p, S) .,$$

This is the maximum distance of a point of X from its closest center in S.



It is somewhat illuminating to consider the problem in the plane. We have a set X of n points in the plane, we would like to find k smallest discs centered at input points, such that they cover all the points of P. Consider the example on the right.

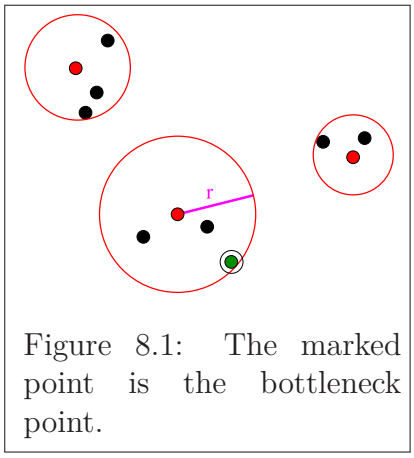


Figure 8.1: The marked point is the bottleneck point.

In this example, assume that we would like to cover it by 3 disks. One possible solution is being shown in Figure 8.1. The quality of the solution is the radius r of the largest disk. As such, the clustering problem here can be interpreted as the problem of computing an optimal cover of the input point set by k discs/balls of minimum radius. This is known as the ***k-center*** problem.

It is known that k-center clustering is NP-Hard, even to approximate within a factor of (roughly) 1.8. Interestingly, there is a simple and elegant 2-approximation algorithm. Namely, one can compute in polynomial time, k centers, such that they induce balls of radius at most twice the optimal radius.

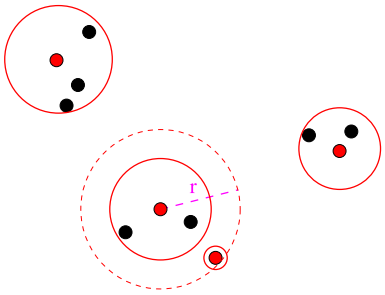
Here is the formal definition of the k-center clustering problem.

k-center clustering

Instance: A set P a of n points, a distance function $\mathbf{d}(p, q)$, for $p, q \in P$, with triangle inequality holding for $\mathbf{d}(\cdot, \cdot)$, and a parameter k.
Question: A subset S that realizes $r_{opt}(P, k) = \min_{S \subseteq P, |S| = k} \mathsf{D}_S(P)$, where $\mathsf{D}_S(P) = \max_{x \in X} \mathbf{d}(x, S)$ and $\mathbf{d}(x, S) = \min_{s \in S} \mathbf{d}(s, x)$.

8.1.1 The approximation algorithm for k-center clustering

To come up with the idea behind the algorithm, imagine that we already have a solution with $m = 3$ centers. We would like to pick the next $m + 1$ center. Inspecting the examples above, one realizes that the solution is being determined by a bottleneck point; see Figure 8.1. That is, there is a single point which determine the quality of the clustering, which is the point furthest away from the set of centers. As such, the natural step is to find a new center that would better serve this bottleneck point. And, what can be a better service for this point, than make it the next center? (The resulting clustering using the new center for the example is depicted on the right.)



Namely, we always pick the bottleneck point, which is furthest away for the current set of centers, as the next center to be added to the solution.

The resulting approximation algorithm is depicted on the right. Observe, that the quantity r_{i+1} denotes the (minimum) radius of the i balls centered at $u_1, \ldots, u_i$ such that they cover P (where all these balls have the same radius). (Namely, there is a point $p \in P$ such that $\mathbf{d}(p, \{u_1, \ldots, u_i\}) = r_{i+1}$.

It would be convenient, for the sake analysis, to imagine that we run **AprxKCenter** one additional iteration, so that the quantity r_{k+1} is well defined.

Observe, that the running time of the algorithm **AprxKCenter** is $O(nk)$ as can be easily verified.

AprxKCenter(P, k)
  $P = \{p_1, \ldots, p_n\}$
  $S = \{p_1\}$, $u_1 \leftarrow p_1$
  while $|S| < k$ **do**
    $i \leftarrow |S|$
    for $j = 1 \ldots n$ **do**
      $d_j \leftarrow \min(d_j, \mathbf{d}(p_j, u_i))$
    $r_{i+1} \leftarrow \max(d_1, \ldots, d_n)$
    $u_{i+1} \leftarrow$point of P realizing r_i
    $S \leftarrow S \cup \{u_{i+1}\}$

  return S

Lemma 8.1.1. *We have that* $r_2 \geq \ldots \geq r_k \geq r_{k+1}$.

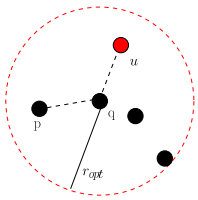
Proof: At each iteration the algorithm adds one new center, and as such the distance of a point to the closest center can not increase. In particular, the distance of the furthest point to the centers does not increase. ∎

Observation 8.1.2. *The radius of the clustering generated by* **AprxKCenter** *is* r_{k+1}.

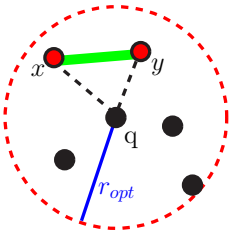
Lemma 8.1.3. *We have that* $r_{k+1} \leq 2r_{opt}(P, k)$, *where* $r_{opt}(P, k)$ *is the radius of the optimal solution using* k *balls.*

Proof: Consider the k balls forming the optimal solution: $D_1, \ldots, D_k$ and consider the k center points contained in the solution S computed by **AprxKCenter**.

If every disk D_i contain at least one point of S, then we are done, since every point of P is in distance at most $2r_{opt}(P,k)$ from one of the points of S. Indeed, if the ball D_i, centered at q, contains the point $u \in S$, then for any point $p \in P \cap D_i$, we have that



$$\mathbf{d}(p,u) \leq \mathbf{d}(p,q) + \mathbf{d}(q,u) \leq 2r_{opt}.$$



Otherwise, there must be two points x and y of S contained in the same ball D_i of the optimal solution. Let D_i be centered at a point q.

We claim distance between x and y is at least r_{k+1}. Indeed, imagine that x was added at the αth iteration (that is, $u_\alpha = x$), and y was added in a later βth iteration (that is, $u_\beta = y$), where $\alpha < \beta$. Then,

$$r_\beta = \mathbf{d}(y, \{u_1, \ldots, u_{\beta-1}\}) \leq \mathbf{d}(x,y)\,,$$

since $x = u_\alpha$ and $y = u_\beta$. But $r_\beta \geq r_{k+1}$, by Lemma 8.1.1. Applying the triangle inequality again, we have that $r_{k+1} \leq r_\beta \leq \mathbf{d}(x,y) \leq \mathbf{d}(x,q) + \mathbf{d}(q,y) \leq 2r_{opt}$, implying the claim. ■

Theorem 8.1.4. *One can approximate the k-center clustering up to a factor of two, in time $O(nk)$.*

Proof: The approximation algorithm is **AprxKCenter**. The approximation quality guarantee follows from Lemma 8.1.3, since the furthest point of P from the k-centers computed is r_{k+1}, which is guaranteed to be at most $2r_{opt}$. ■

8.2 Subset Sum

Subset Sum

Instance: $X = \{x_1, \ldots, x_n\}$ – n integer positive numbers, t - target number
Question: Is there a subset of X such the sum of its elements is t?

Subset Sum is (of course) NPC, as we already proved. It can be solved in polynomial time if the numbers of X are small. In particular, if $x_i \leq M$, for $i = 1, \ldots, n$, then $t \leq Mn$ (otherwise, there is no solution). Its reasonably easy to solve in this case, as the algorithm on the right shows. The running time of the resulting algorithm is $O(Mn^2)$.

SolveSubsetSum (X, t, M)
  $b[0 \ldots Mn]$ - boolean array init to FALSE.
    // $b[x]$ is TRUE if x can be realized by a subset of X.
  $b[0] \leftarrow$ TRUE.
  for $i = 1, \ldots, n$ **do**
    for $j = Mn$ down to x_i **do**
      $b[j] \leftarrow B[j - x_i] \vee B[j]$

  return $B[t]$

Note, that M my be prohibitly large, and as such, this algorithm is not polynomial in n. In particular, if $M = 2^n$ then this

algorithm is prohibitly slow. Since the relevant decision problem is NPC, it is unlikely that an efficient algorithm exist for this problem. But still, we would like to be able to solve it quickly and efficiently. So, if we want an efficient solution, we would have to change the problem slightly. As a first step, lets turn it into an optimization problem.

Subset Sum Optimization

Instance: (X, t): A set X of n positive integers, and a target number t.
Question: The largest number γ_{opt} one can represent as a subset sum of X which is smaller or equal to t.

Intuitively, we would like to find a subset of X such that it sum is smaller than t but very close to t.

Next, we turn problem into an approximation problem.

Subset Sum Approx

Instance: (X, t, ε): A set X of n positive integers, a target number t, and parameter $\varepsilon > 0$.
Question: A number z that one can represent as a subset sum of X, such that $(1 - \varepsilon)\gamma_{opt} \leq z \leq \gamma_{opt} \leq t$.

The challenge is to solve this approximation problem efficiently. To demonstrate that there is hope that can be done, consider the following simple approximation algorithm, that achieves a constant factor approximation.

Lemma 8.2.1. *If there is a subset sum that adds up to t one can find a subset sum that adds up to at least $\gamma_{opt}/2$ in $O(n \log n)$ time.*

Proof: Add the numbers from largest to smallest, whenever adding a number will make the sum exceed t, we throw it away. We claim that the generate sum $\gamma_{opt}/2 \leq s \leq t$. Clearly, if the total sum of the numbers is smaller than t, then no number is being rejected and $s = \gamma_{opt}$.

Otherwise, let u be the first number being rejected, and let s' be the partial subset sum, just before u is being rejected. Clearly, $s' > u > 0$, $s' < t$, and $s' + u > t$. This implies $t < s' + u < s' + s' = 2s'$, which implies that $s' \geq t/2$. Namely, the subset sum output is larger than $t/2$. ∎

8.2.1 On the complexity of ε-approximation algorithms

Definition 8.2.2 (PTAS). For a maximization problem PROB, an algorithm $\mathcal{A}(I, \varepsilon)$ (i.e., A receives as input an instance of PROB, and an approximation parameter $\varepsilon > 0$) is a ***polynomial time approximation scheme*** (***PTAS***) if for any instance I we have

$$(1 - \varepsilon) \left|\mathsf{opt}(I)\right| \leq \left|\mathcal{A}(I, \varepsilon)\right| \leq \left|\mathsf{opt}(I)\right|,$$

where $|\mathsf{opt}(I)|$ denote the price of the optimal solution for I, and $|\mathcal{A}(I,\varepsilon)|$ denotes the price of the solution outputted by $\mathcal{A}$. Furthermore, the running time of the algorithm $\mathcal{A}$ is polynomial in n (the input size), when ε is fixed.

For a minimization problem, the condition is that $|\mathsf{opt}(I)| \leq |\mathcal{A}(I,\varepsilon)| \leq (1+\varepsilon)|\mathsf{opt}(I)|$.

Example 8.2.3. An approximation algorithm with running time $O(n^{1/\varepsilon})$ is a PTAS, while an algorithm with running time $O(1/\varepsilon^n)$ is not.

Definition 8.2.4 (FPTAS.). An approximation algorithm is ***fully polynomial time approximation scheme*** (***FPTAS***) if it is a PTAS, and its running time is polynomial both in n and $1/\varepsilon$.

Example 8.2.5. A PTAS with running time $O(n^{1/\varepsilon})$ is not a FPTAS, while a PTAS with running time $O(n^2/\varepsilon^3)$ is a FPTAS.

8.2.2 Approximating subset-sum

Let $S = \{a_1, \ldots, a_n\}$ be a set of numbers. For a number x, let $x + S$ denote the translation of S by x; namely, $x + S = \{a_1 + x, a_2 + x, \ldots a_n + x\}$. Our first step in deriving an approximation algorithm for Subset Sum is to come up with a slightly different algorithm for solving the problem exactly. The algorithm is depicted on the right.

ExactSubsetSum(S, t)
  $n \leftarrow |S|$
  $P_0 \leftarrow \{0\}$
  for $i = 1 \ldots n$ **do**
    $P_i \leftarrow P_{i-1} \cup (P_{i-1} + x_i)$
    Remove from P_i all elements $> t$
  return largest element in P_n

Note, that while **ExactSubsetSum** performs only n iterations, the lists P_i that it constructs might have exponential size.

Trim(L', δ)
  $L \leftarrow$ **Sort**(L')
  $L = \langle y_1 \ldots y_m \rangle$
    // $y_i \leq y_{i+1}$, for $i = 1, \ldots, n-1$.
  $curr \leftarrow y_1$
  $L_{out} \leftarrow \{y_1\}$
  for $i = 2 \ldots m$ **do**
    if $y_i > curr \cdot (1+\delta)$
      Append y_i to L_{out}
      $curr \leftarrow y_i$
  return L_{out}

Thus, if we would like to turn **ExactSubsetSum** into a faster algorithm, we need to somehow to make the lists L_l smaller. This would be done by removing numbers which are very close together.

Definition 8.2.6. For two positive real numbers $z \leq y$, the number y is a δ-approximation to z if $\frac{y}{1+\delta} \leq z \leq y$.

The procedure **Trim** that trims a list L' so that it removes close numbers is depicted on the left.

Observation 8.2.7. *If $x \in L'$ then there exists a number $y \in L_{out}$ such that $y \leq x \leq y(1+\delta)$, where $L_{out} \leftarrow$ **Trim**(L', δ).*

We can now modify **ExactSubsetSum** to use **Trim** to keep the candidate list shorter. The resulting algorithm **ApproxSubsetSum** is depicted on the right.

Let E_i be the list generated by the algorithm in the ith iteration, and P_i be the list of numbers without any trimming (i.e., the set generated by **ExactSubsetSum** algorithm) in the ith iteration.

ApproxSubsetSum(S, t)
  //Assume $S = \{x_1, \ldots, x_n\}$, where
  // $x_1 \leq x_2 \leq \ldots \leq x_n$
  $n \leftarrow |S|$, $L_0 \leftarrow \{0\}$, $\delta = \varepsilon/2n$
  for $i = 1 \ldots n$ **do**
    $E_i \leftarrow L_{i-1} \cup (L_{i-1} + x_i)$
    $L_i \leftarrow$ **Trim**(E_i, δ)
    Remove from L_i all elements $> t$.
  return largest element in L_n

Claim 8.2.8. *For any $x \in P_i$ there exists $y \in L_i$ such that $y \leq x \leq (1+\delta)^i y$.*

Proof: If $x \in P_1$ the claim follows by Observation 8.2.7 above. Otherwise, if $x \in P_{i-1}$, then, by induction, there is $y' \in L_{i-1}$ such that $y' \leq x \leq (1+\delta)^{i-1} y'$. Observation 8.2.7 implies that there exists $y \in L_i$ such that $y \leq y' \leq (1+\delta)y$, As such,

$$y \leq y' \leq x \leq (1+\delta)^{i-1} y' \leq (1+\delta)^i y$$

as required.

The other possibility is that $x \in P_i \setminus P_{i-1}$. But then $x = \alpha + x_i$, for some $\alpha \in P_{i-1}$. By induction, there exists $\alpha' \in L_{i-1}$ such that

$$\alpha' \leq \alpha \leq (1+\delta)^{i-1} \alpha'.$$

Thus, $\alpha' + x_i \in E_i$ and by Observation 8.2.7, there is a $x' \in L_i$ such that

$$x' \leq \alpha' + x_i \leq (1+\delta) x'.$$

Thus,

$$x' \leq \alpha' + x_i \leq \alpha + x_i = x \leq (1+\delta)^{i-1} \alpha' + x_i \leq (1+\delta)^{i-1} (\alpha' + x_i) \leq (1+\delta)^i x'.$$

Namely, for any $x \in P_i \setminus P_{i-1}$, there exists $x' \in L_i$, such that $x' \leq x \leq (1+\delta)^i x'$. ∎

8.2.2.1 Bounding the running time of ApproxSubsetSum

We need the following two easy technical lemmas. We include their proofs here only for the sake of completeness.

Lemma 8.2.9. *For $x \in [0,1]$, it holds $\exp(x/2) \leq (1+x)$.*

Proof: Let $f(x) = \exp(x/2)$ and $g(x) = 1 + x$. We have $f'(x) = \exp(x/2)/2$ and $g'(x) = 1$. As such,

$$f'(x) = \frac{\exp(x/2)}{2} \leq \frac{\exp(1/2)}{2} \leq 1 = g'(x), \qquad \text{for } x \in [0,1].$$

Now, $f(0) = g(0) = 1$, which immediately implies the claim. ∎

Lemma 8.2.10. *For $0 < \delta < 1$, and $x \geq 1$, we have $\log_{1+\delta} x \leq \dfrac{2 \ln x}{\delta} = .O\left(\dfrac{\ln x}{\delta}\right)$.*

Proof: We have, by Lemma 8.2.9, that $\log_{1+\delta} x = \dfrac{\ln x}{\ln(1+\delta)} \leq \dfrac{\ln x}{\ln \exp(\delta/2)} = \dfrac{2 \ln x}{\delta}$. ∎

Observation 8.2.11. *In a list generated by* **Trim**, *for any number x, there are no two numbers in the trimmed list between x and $(1+\delta)x$.*

Lemma 8.2.12. *We have $|L_i| = O\Big((n/\varepsilon^2) \log n\Big)$, for $i = 1, \ldots, n$.*

Proof: The set $L_{i-1} + x_i$ is a set of numbers between x_i and ix_i, because x_i is larger than $x_1 \ldots x_{i-1}$ and L_{i-1} contains subset sums of at most $i-1$ numbers, each one of them smaller than x_i. As such, the number of different values in this range, stored in the list L_i, after trimming is at most

$$\log_{1+\delta} \frac{ix_i}{x_i} = O\left(\frac{\ln i}{\delta}\right) = O\left(\frac{\ln n}{\delta}\right),$$

by Lemma 8.2.10. Thus, as $\delta = \varepsilon/2n$, we have

$$|L_i| \leq |L_{i-1}| + O\left(\frac{\ln n}{\delta}\right) \leq |L_{i-1}| + O\left(\frac{n \ln n}{\varepsilon}\right) = O\left(\frac{n^2 \log n}{\varepsilon}\right).$$

∎

Lemma 8.2.13. *The running time of* **ApproxSubsetSum** *is $O\Big(\frac{n^3}{\varepsilon} \log^2 n\Big)$.*

Proof: Clearly, the running time of **ApproxSubsetSum** is dominated by the total length of the lists $L_1, \ldots, L_n$ it creates. Lemma 8.2.12 implies that $\sum_i |L_i| = O\left(\dfrac{n^3}{\varepsilon} \log n\right)$. Now, since **Trim** have to sort the lists, its running time is $O(|L_i| \log |L_i|)$ in the ith iteration. Overall, the running time is $O(\sum_i |L_i| \, |L_i|) = O\Big(\frac{n^3}{\varepsilon} \log^2 n\Big)$. ∎

8.2.2.2 The result

Theorem 8.2.14. **ApproxSubsetSum** *returns a number $u \leq t$, such that*

$$\frac{\gamma_{opt}}{1+\varepsilon} \leq u \leq \gamma_{opt} \leq t,$$

where γ_{opt} is the optimal solution (i.e., largest realizable subset sum smaller than t).

The running time of **ApproxSubsetSum** *is $O\Big((n^3/\varepsilon) \log^2 n\Big)$.*

Proof: The running time bound is by Lemma 8.2.13.

As for the other claim, consider the optimal solution $\mathsf{opt} \in P_n$. By Claim 8.2.8, there exists $z \in L_n$ such that $z \leq \mathsf{opt} \leq (1+\delta)^n z$. However,

$$(1+\delta)^n = (1+\varepsilon/2n)^n \leq \exp\left(\frac{\varepsilon}{2}\right) \leq 1+\varepsilon,$$

since $1 + x \leq e^x$ for $x \geq 0$. Thus, $\mathsf{opt}/(1+\varepsilon) \leq z \leq \mathsf{opt} \leq t$, implying that the output of **ApproxSubsetSum** is within the required range. ∎

8.3 Approximate Bin Packing

Consider the following problem.

Min Bin Packing

Instance: $a_1 \ldots a_n$ – n numbers in $[0, 1]$
Question: Q: What is the minimum number of unit bins do you need to use to store all the numbers in S?

Bin Packing is NP-Complete because you can reduce Partition to it. Its natural to ask how one can approximate the optimal solution to Bin Packing.

One such algorithm is ***next fit***. Here, we go over the numbers one by one, and put a number in the current bin if that bin can contain it. Otherwise, we create a new bin and put the number in this bin. Clearly, we need at least

$$\lceil A \rceil \ \textit{bins} \quad \textit{where} \ \ A = \sum_{i=1}^{n} a_i$$

Every two consecutive bins contain numbers that add up to more than 1, since otherwise we would have not created the second bin. As such, the number of bins used is $\leq 2\lceil A \rceil$. As such, the next fit algorithm for bin packing achieves a $\leq 2\lceil A \rceil / \lceil A \rceil = 2$ approximation.

A better strategy, is to sort the numbers from largest to smallest and insert them in this order, where in each stage, we scan all current bins, and see if can insert the current number into one of those bins. If we can not, we create a new bin for this number. This is known as ***first fit***. We state the approximation ratio for this algorithm without proof.

Theorem 8.3.1. *Decreasing first fit is a* 1.5*-approximation to Min Bin Packing.*

8.4 Bibliographical notes

One can do 2-approximation for the k-center clustering in low dimensional Euclidean space can be done in $\Theta(n \log k)$ time [FG88]. In fact, it can be solved in linear time [Har04].

Part II

Randomized Algorithms

Chapter 9

Randomized Algorithms

9.1 Some Probability

Definition 9.1.1. (Informal.) A ***random variable*** is a measurable function from a probability space to (usually) real numbers. It associates a value with each possible atomic event in the probability space.

Definition 9.1.2. The ***conditional probability*** of X given Y is

$$\mathbf{Pr}\Big[X = x \,\Big|\, Y = y\Big] = \frac{\mathbf{Pr}\Big[(X = x) \cap (Y = y)\Big]}{\mathbf{Pr}\Big[Y = y\Big]}.$$

An equivalent and useful restatement of this is that

$$\mathbf{Pr}\Big[(X = x) \cap (Y = y)\Big] = \mathbf{Pr}\Big[X = x \,\Big|\, Y = y\Big] * \mathbf{Pr}\Big[Y = y\Big].$$

Definition 9.1.3. Two events X and Y are ***independent***, if $\mathbf{Pr}[X = x \cap Y = y] = \mathbf{Pr}[X = x] \cdot \mathbf{Pr}[Y = y]$. In particular, if X and Y are independent, then

$$\mathbf{Pr}\Big[X = x \,\Big|\, Y = y\Big] = \mathbf{Pr}[X = x].$$

Definition 9.1.4. The ***expectation*** of a random variable X is the average value of this random variable. Formally, if X has a finite (or countable) set of values, it is

$$\mathbf{E}\Big[X\Big] = \sum_x x \cdot \mathbf{Pr}[X = x],$$

where the summation goes over all the possible values of X.

One of the most powerful properties of expectations is that an expectation of a sum is the sum of expectations.

Lemma 9.1.5 (Linearity of expectation.). *For any two random variables X and Y, we have* $\mathbf{E}\Big[X+Y\Big] = \mathbf{E}\Big[X\Big] + \mathbf{E}\Big[Y\Big]$.

Proof: For the simplicity of exposition, assume that X and Y receive only integer values. We have that

$$\begin{aligned}
\mathbf{E}\Big[X+Y\Big] &= \sum_x \sum_y (x+y)\,\mathbf{Pr}\Big[(X=x)\cap(Y=y)\Big] \\
&= \sum_x \sum_y x * \mathbf{Pr}\Big[(X=x)\cap(Y=y)\Big] + \sum_x \sum_y y * \mathbf{Pr}\Big[(X=x)\cap(Y=y)\Big] \\
&= \sum_x x * \sum_y \mathbf{Pr}\Big[(X=x)\cap(Y=y)\Big] + \sum_y y * \sum_x \mathbf{Pr}\Big[(X=x)\cap(Y=y)\Big] \\
&= \sum_x x * \mathbf{Pr}\Big[X=x\Big] + \sum_y y * \mathbf{Pr}\Big[Y=y\Big] \\
&= \mathbf{E}\Big[X\Big] + \mathbf{E}\Big[Y\Big].
\end{aligned}$$

■

Another interesting creature, is the conditional expectation; that is, it is the expectation of a random variable given some additional information.

Definition 9.1.6. Given random variables X and Y, the ***conditional expectation*** of X given Y, is the quantity $\mathbf{E}\Big[X \;\Big|\; Y\Big]$. Specifically, you are given the value y of the random variable Y, and the $\mathbf{E}\Big[X \;\Big|\; Y\Big] = \mathbf{E}\Big[X \;\Big|\; Y=y\Big] = \sum_x x * \mathbf{Pr}\Big[X=x \;\Big|\; Y=y\Big]$.

Note, that for a random variable X, the expectation $\mathbf{E}[X]$ is a number. On the other hand, the conditional probability $f(y) = \mathbf{E}\Big[X \;\Big|\; Y=y\Big]$ is a function. The key insight why conditional probability is the following.

Lemma 9.1.7. *For any two random variables X and Y (not necessarily independent), we have that* $\mathbf{E}[X] = \mathbf{E}\Big[\mathbf{E}\Big[X \;\Big|\; Y\Big]\Big]$.

Proof: We use the definitions carefully:

$$\begin{aligned}
\mathbf{E}\Big[\mathbf{E}\Big[X \;\Big|\; Y\Big]\Big] &= \mathop{\mathbf{E}}_y\Big[\mathbf{E}\Big[X \;\Big|\; Y=y\Big]\Big] = \mathop{\mathbf{E}}_y\left[\sum_x x * \mathbf{Pr}\Big[X=x \;\Big|\; Y=y\Big]\right] \\
&= \sum_y \mathbf{Pr}\Big[Y=y\Big] * \left(\sum_x x * \mathbf{Pr}\Big[X=x \;\Big|\; Y=y\Big]\right) \\
&= \sum_y \mathbf{Pr}\Big[Y=y\Big] * \left(\sum_x x * \frac{\mathbf{Pr}\Big[(X=x)\cap(Y=y)\Big]}{\mathbf{Pr}[Y=y]}\right) \\
&= \sum_y \sum_x x * \mathbf{Pr}\Big[(X=x)\cap(Y=y)\Big] = \sum_x \sum_y x * \mathbf{Pr}\Big[(X=x)\cap(Y=y)\Big] \\
&= \sum_x x * \left(\sum_y \mathbf{Pr}\Big[(X=x)\cap(Y=y)\Big]\right) = \sum_x x * \mathbf{Pr}\Big[X=x\Big] = \mathbf{E}[X].
\end{aligned}$$

■

9.2 Sorting Nuts and Bolts

Problem 9.2.1 (Sorting Nuts and Bolts). You are given a set of n nuts and n bolts. Every nut have a matching bolt, and all the n pairs of nuts and bolts have different sizes. Unfortunately, you get the nuts and bolts separated from each other and you have to match the nuts to the bolts. Furthermore, given a nut and a bolt, all you can do is to try and match one bolt against a nut (i.e., you can not compare two nuts to each other, or two bolts to each other).

When comparing a nut to a bolt, either they match, or one is smaller than other (and you known the relationship after the comparison).

How to match the n nuts to the n bolts quickly? Namely, while performing a small number of comparisons.

The naive algorithm is of course to compare each nut to each bolt, and match them together. This would require a quadratic number of comparisons. Another option is to sort the nuts by size, and the bolts by size and then "merge" the two ordered sets, matching them by size. The only problem is that we can not sorts only the nuts, or only the bolts, since we can not compare them to each other. Indeed, we sort the two sets simultaneously, by simulating **QuickSort**. The resulting algorithm is depicted on the right.

MatchNutsAndBolts(N: nuts, B: bolts)
  Pick a random nut n_{pivot} from N
  Find its matching bolt b_{pivot} in B
  $B_L \leftarrow$ All bolts in B smaller than n_{pivot}
  $N_L \leftarrow$ All nuts in N smaller than b_{pivot}
  $B_R \leftarrow$ All bolts in B larger than n_{pivot}
  $N_R \leftarrow$ All nuts in N larger than b_{pivot}
  MatchNutsAndBolts(N_R,B_R)
  MatchNutsAndBolts(N_L,B_L)

9.2.1 Running time analysis

Definition 9.2.2. Let $\mathcal{RT}$ denote the random variable which is the running time of the algorithm. Note, that the running time is a random variable as it might be different between different executions on the *same input*.

Definition 9.2.3. For a randomized algorithm, we can speak about the expected running time. Namely, we are interested in bounding the quantity $\mathbf{E}[\mathcal{RT}]$ for the worst input.

Definition 9.2.4. The ***expected running-time*** of a randomized algorithm for input of size n is

$$T(n) = \max_{U \text{ is an input of size } n} \mathbf{E}\Big[\mathcal{RT}(U)\Big],$$

where $\mathcal{RT}(U)$ is the running time of the algorithm for the input U.

Definition 9.2.5. The ***rank*** of an element x in a set S, denoted by $\mathrm{rank}(x)$, is the number of elements in S of size smaller or equal to x. Namely, it is the location of x in the sorted list of the elements of S.

Theorem 9.2.6. *The expected running time of* **MatchNutsAndBolts** *(and thus also of* **QuickSort***) is $T(n) = O(n \log n)$, where n is the number of nuts and bolts. The worst case running time of this algorithm is $O(n^2)$.*

Proof: Clearly, we have that $\mathbf{Pr}[\text{rank}(n_{pivot}) = k] = \frac{1}{n}$. Furthermore, if the rank of the pivot is k then

$$\begin{aligned} T(n) &= \mathop{\mathbf{E}}_{k=\text{rank}(n_{pivot})}[O(n) + T(k-1) + T(n-k)] = O(n) + \mathop{\mathbf{E}}_{k}[T(k-1) + T(n-k)] \\ &= T(n) = O(n) + \sum_{k=1}^{n} \mathbf{Pr}[Rank(Pivot) = k] * (T(k-1) + T(n-k)) \\ &= O(n) + \sum_{k=1}^{n} \frac{1}{n} \cdot (T(k-1) + T(n-k)), \end{aligned}$$

by the definition of expectation. It is not easy to verify that the solution to the recurrence $T(n) = O(n) + \sum_{k=1}^{n} \frac{1}{n} \cdot (T(k-1) + T(n-k))$ is $O(n \log n)$. ■

9.2.1.1 Alternative incorrect solution

The algorithm **MatchNutsAndBolts** is lucky if $\frac{n}{4} \leq \text{rank}(n_{pivot}) \leq \frac{3}{4}n$. Thus, $\mathbf{Pr}[\text{“lucky”}] = 1/2$. Intuitively, for the algorithm to be fast, we want the split to be as balanced as possible. The less balanced the cut is, the worst the expected running time. As such, the “Worst” lucky position is when $\text{rank}(n_{pivot}) = n/4$ and we have that

$$T(n) \leq O(n) + \mathbf{Pr}[\text{“lucky”}] * (T(n/4) + T(3n/4)) + \mathbf{Pr}[\text{“unlucky”}] * T(n).$$

Namely, $T(n) = O(n) + \frac{1}{2} * \left(T(\frac{n}{4}) + T(\frac{3}{4}n)\right) + \frac{1}{2}T(n)$. Rewriting, we get the recurrence $T(n) = O(n) + T(n/4) + T((3/4)n)$, and its solution is $O(n \log n)$.

While this is a very intuitive and elegant solution that bounds the running time of **QuickSort**, it is also incomplete. The interested reader should try and make this argument complete. After completion the argument is as involved as the previous argument. Nevertheless, this argumentation gives a good back of the envelope analysis for randomized algorithms which can be applied in a lot of cases.

9.2.2 What are randomized algorithms?

Randomized algorithms are algorithms that use random numbers (retrieved usually from some unbiased source of randomness [say a library function that returns the result of a random coin flip]) to make decisions during the executions of the algorithm. The running time becomes a random variable. Analyzing the algorithm would now boil down to analyzing the behavior of the random variable $\mathcal{RT}(n)$, where n denotes the size of the input.In particular, the expected running time $\mathbf{E}[\mathcal{RT}(n)]$ is a quantity that we would be interested in.

It is useful to compare the expected running time of a randomized algorithm, which is

$$T(n) = \max_{U \text{ is an input of size } n} \mathbf{E}\Big[\mathcal{RT}(U)\Big],$$

to the worst case running time of a deterministic (i.e., not randomized) algorithm, which is

$$T(n) = \max_{U \text{ is an input of size } n} \mathcal{RT}(U),$$

Caveat Emptor:[1] Note, that a randomized algorithm might have exponential running time in the worst case (or even unbounded) while having good expected running time. For example, consider the algorithm **FlipCoins** depicted on the right. The expected running time of **FlipCoins** is a geometric random variable with probability 1/2, as such we have that $\mathbf{E}[\mathcal{RT}(\text{ FlipCoins })] = O(2)$. However, **FlipCoins** can run forever if it always gets 1 from the **RandBit** function.

```
FlipCoins
    while RandBit= 1 do
        nothing;
```

This is of course a ludicrous argument. Indeed, the probability that **FlipCoins** runs for long decreases very quickly as the number of steps increases. It can happen that it runs for long, but it is extremely unlikely.

Definition 9.2.7. The running time of a randomized algorithm **Alg** is $O(f(n))$ with ***high probability*** if

$$\mathbf{Pr}\Big[\mathcal{RT}(\mathbf{Alg}(n)) \geq c \cdot f(n)\Big] = o(1).$$

Namely, the probability of the algorithm to take more than $O(f(n))$ time decreases to 0 as n goes to infinity. In our discussion, we would use the following (considerably more restrictive definition), that requires that

$$\mathbf{Pr}\Big[\mathcal{RT}(\mathbf{Alg}(n)) \geq c \cdot f(n)\Big] \leq \frac{1}{n^d},$$

where c and d are appropriate constants. For technical reasons, we also require that $\mathbf{E}[\mathcal{RT}(\mathbf{Alg}(n))] = O(f(n))$.

9.3 Analyzing QuickSort

The previous analysis works also for **QuickSort**. However, there is an alternative analysis which is also very interesting and elegant. Let $a_1, ..., a_n$ be the n given numbers (in sorted order – as they appear in the output).

It is enough to bound the number of comparisons performed by **QuickSort** to bound its running time, as can be easily verified. Observe, that two specific elements are compared to each other by **QuickSort** at most once, because **QuickSort** performs only comparisons against the pivot, and after the comparison happen, the pivot does not being passed to the two recursive subproblems.

Let X_{ij} be an indicator variable if **QuickSort** compared a_i to a_j in the current execution, and zero otherwise. The number of comparisons performed by **QuickSort** is **exactly** $Z = \sum_{i<j} X_{ij}$.

[1] Caveat Emptor - let the buyer beware (i.e., one buys at one's own risk)

Observation 9.3.1. *The element a_i is compared to a_j iff one of them is picked to be the pivot and they are still in the same subproblem.*

Also, we have that $\mu = \mathbf{E}[X_{ij}] = \mathbf{Pr}[X_{ij} = 1]$. To quantify this probability, observe that if the pivot is smaller than a_i or larger than a_j then the subproblem still contains the block of elements $a_i, \ldots, a_j$. Thus, we have that

$$\mu = \mathbf{Pr}[a_i \text{ or } a_j \text{ is first pivot} \in a_i, \ldots, a_j] = \frac{2}{j-i+1}.$$

Another (and hopefully more intuitive) explanation for the above phenomena is the following: Imagine, that before running **QuickSort** we choose for every element a random priority, which is a real number in the range $[0, 1]$. Now, we reimplement **QuickSort** such that it always pick the element with the lowest random priority (in the given subproblem) to be the pivot. One can verify that this variant and the standard implementation have the same running time. Now, a_i gets compares to a_j if and only if all the elements $a_{i+1}, \ldots, a_{j-1}$ have random priority larger than both the random priority of a_i and the random priority of a_j. But the probability that one of two elements would have the lowest random-priority out of $j - i + 1$ elements is $2 * 1/(j - i + 1)$, as claimed.

Thus, the running time of **QuickSort** is

$$\begin{aligned}\mathbf{E}\Big[\mathcal{RT}(n)\Big] &= \mathbf{E}\left[\sum_{i<j} X_{ij}\right] = \sum_{i<j} \mathbf{E}\Big[X_{ij}\Big] = \sum_{i<j} \frac{2}{j-i+1} = 2\sum_{i=1}^{n-1}\sum_{j=i+1}^{n} \frac{1}{j-i+1} \\ &= 2\sum_{i=1}^{n-1}\sum_{\Delta=2}^{n-i+1} \frac{1}{\Delta} \leq 2\sum_{i=1}^{n-1}\sum_{\Delta=1}^{n} \frac{1}{\Delta} \leq 2\sum_{i=1}^{n-1} H_n = 2nH_n.\end{aligned}$$

by linearity of expectations, where $H_n = \sum_{i=1}^{n} \frac{1}{i} \leq \ln n + 1$ is the nth harmonic number,

As we will see in the near future, the running time of **QuickSort** is $O(n \log n)$ with high-probability. We need some more tools before we can show that.

9.4 QuickSelect – median selection in linear time

Consider the problem of given a set X of n numbers, and a parameter k, to output the kth smallest number (which is the number with ***rank*** k in X). This can be easily be done by modifying **QuickSort** only to perform one recursive call. See Figure 9.1 for a pseud-code of the resulting algorithm.

Intuitively, at each iteration of **QuickSelect** the input size shrinks by a constant factor, leading to a linear time algorithm.

Theorem 9.4.1. *Given a set X of n numbers, and any integer k, the expected running time of* **QuickSelect**(X, n) *is $O(n)$.*

```
QuickSelect(X, k)
  // Input:  X = {x_1, ..., x_n} numbers, k.
  // Assume x_1, ..., x_n are all distinct.
  // Task:  Return kth smallest number in X.
  y ← random element of X.
  r ← rank of y in X.
  if r = k then return y
  X_< = all elements in X < than y
  X_> = all elements in X > than y
  // By assumption |X_<| + |X_>| + 1 = |X|.
  if r < k then
    return QuickSelect( X_>, k − r )
  else
    return QuickSelect( X_≤, k )
```

Figure 9.1: **QuickSelect** pseudo-code.

Proof: Let $X_1 = X$, and X_i be the set of numbers in the ith level of the recursion. Let y_i and r_i be the random element and its rank in X_i, respectively, in the ith iteration of the algorithm. Finally, let $n_i = |X_i|$. Observe that the probability that the pivot y_i is in the "middle" of its subproblem is

$$\alpha = \mathbf{Pr}\left[\frac{n_i}{4} \leq r_i \leq \frac{3}{4}n_i\right] \geq \frac{1}{2},$$

and if this happens then

$$n_{i+1} \leq \max(r_i - 1, n_i - r_i) \leq \frac{3}{4}n_i.$$

We conclude that

$$\begin{aligned}\mathbf{E}\left[n_{i+1} \,\middle|\, n_i\right] &\leq \mathbf{Pr}\left[y_i \text{ in the middle}\right]\frac{3}{4}n_i + \mathbf{Pr}\left[y_i \text{ not in the middle}\right] n_i \\ &\leq \alpha\frac{3}{4}n_i + (1-\alpha)\, n_i = n_i(1-\alpha/4) \leq n_i(1-(1/2)/4) = (7/8)n_i.\end{aligned}$$

Now, we have that

$$\begin{aligned}m_{i+1} &= \mathbf{E}\left[n_{i+1}\right] = \mathbf{E}\left[\mathbf{E}\left[n_{i+1} \,\middle|\, n_i\right]\right] \leq \mathbf{E}[(7/8)n_i] = (7/8)\,\mathbf{E}[n_i] = (7/8)m_i \\ &= (7/8)^i m_0 = (7/8)^i n,\end{aligned}$$

since for any two random variables we have that $\mathbf{E}[X] = \mathbf{E}\left[\mathbf{E}\left[X \,\middle|\, Y\right]\right]$. In particular, the expected running time of **QuickSelect** is proportional to

$$\mathbf{E}\left[\sum_i n_i\right] = \sum_i \mathbf{E}\left[n_i\right] \leq \sum_i m_i = \sum_i (7/8)^i n = O(n),$$

as desired. ■

Chapter 10

Randomized Algorithms II

10.1 QuickSort and Treaps with High Probability

You must be asking yourself what are treaps. For the answer, see Section 10.3$_{\text{p90}}$.

One can think about **QuickSort** as playing a game in rounds. Every round, **QuickSort** picks a pivot, splits the problem into two subproblems, and continue playing the game recursively on both subproblems.

If we track a single element in the input, we see a sequence of rounds that involve this element. The game ends, when this element find itself alone in the round (i.e., the subproblem is to sort a single element).

Thus, to show that **QuickSort** takes $O(n \log n)$ time, it is enough to show, that every element in the input, participates in at most $32 \ln n$ rounds with high enough probability.

Indeed, let X_i be the event that the ith element participates in more than $32 \ln n$ rounds.

Let C_{QS} be the number of comparisons performed by **QuickSort**. A comparison between a pivot and an element will be always charged to the element. And as such, the number of comparisons overall performed by **QuickSort** is bounded by $\sum_i r_i$, where r_i is the number of rounds the ith element participated in (the last round where it was a pivot is ignored). We have that

$$\alpha = \mathbf{Pr}\Big[C_{QS} \geq 32n \ln n\Big] \leq \mathbf{Pr}\left[\bigcup_i X_i\right] \leq \sum_{i=1}^{n} \mathbf{Pr}\Big[X_i\Big] .$$

Here, we used the ***union rule***, that states that for any two events A and B, we have that $\mathbf{Pr}[A \cup B] \leq \mathbf{Pr}[A] + \mathbf{Pr}[B]$. Assume, for the time being, that $\mathbf{Pr}[X_i] \leq 1/n^3$. This implies that

$$\alpha \leq \sum_{i=1}^{n} \mathbf{Pr}\Big[X_i\Big] \leq \sum_{i=1}^{n} \frac{1}{n^3} = \frac{1}{n^2}.$$

Namely, **QuickSort** performs at most $32n \ln n$ comparisons with high probability. It follows, that **QuickSort** runs in $O(n \log n)$ time, with high probability, since the running time of **QuickSort** is proportional to the number of comparisons it performs.

To this end, we need to prove that $\mathbf{Pr}[X_i] \leq 1/n^3$.

10.1.1 Proving that an elements participates in small number of rounds.

Consider a run of **QuickSort** for an input made out of n numbers. Consider a specific element x in this input, and let $S_1, S_2, \ldots$ be the subsets of the input that are in the recursive calls that include the element x. Here S_j is the set of numbers in the jth round (i.e., this is the recursive call at depth j which includes x among the numbers it needs to sort).

The element x would be considered to be ***lucky***, in the jth iteration, if the call to the **QuickSort**, splits the current set S_j into two parts, where both parts contains at most $(3/4)\,|S_j|$ of the elements.

Let Y_j be an indicator variable which is 1 if and only if x is lucky in jth round. Formally, $Y_j = 1$ if and only if $|S_j|/4 \leq |S_{j+1}| \leq 3\,|S_j|/4$. By definition, we have that

$$\mathbf{Pr}\Big[Y_j\Big] = \frac{1}{2}.$$

Furthermore, $Y_1, Y_2, \ldots, Y_m$ are all independent variables.

Note, that x can participate in at most

$$\rho = \log_{4/3} n \leq 3.5 \ln n \tag{10.1}$$

rounds, since at each successful round, the number of elements in the subproblem shrinks by at least a factor $3/4$, and $|S_1| = n$. As such, if there are ρ successful rounds in the first k rounds, then $|S_k| \leq (3/4)^\rho n \leq 1$.

Thus, the question of how many rounds x participates in, boils down to how many coin flips one need to perform till one gets ρ heads. Of course, in expectation, we need to do this 2ρ times. But what if we want a bound that holds with high probability, how many rounds are needed then?

In the following, we require the following lemma, which we will prove in Section 10.2.

Lemma 10.1.1. *In a sequence of M coin flips, the probability that the number of ones is smaller than $L \leq M/4$ is at most* $\exp(-M/8)$.

To use Lemma 10.1.1, we set

$$M = 32 \ln n \geq 8\rho,$$

see Eq. (10.1). Let Y_j be the variable which is one if x is lucky in the jth level of recursion, and zero otherwise. We have that $\mathbf{Pr}[Y_j = 0] = \mathbf{Pr}[Y_j = 1] = 1/2$ and that $Y_1, Y_2, \ldots, Y_M$ are independent. By Lemma 10.1.1, we have that the probability that there are only $\rho \leq M/4$ ones in $Y_1, \ldots, Y_M$, is smaller than

$$\exp\left(-\frac{M}{8}\right) \leq \exp(-\rho) \leq \frac{1}{n^3}.$$

We have that the probability that x participates in M recursive calls of **QuickSort** to be at most $1/n^3$.

There are n input elements. Thus, the probability that depth of the recursion in **QuickSort** exceeds $32 \ln n$ is smaller than $(1/n^3) * n = 1/n^2$. We thus established the following result.

Theorem 10.1.2. *With high probability (i.e., $1-1/n^2$) the depth of the recursion of* **QuickSort** *is $\leq 32 \ln n$. Thus, with high probability, the running time of* **QuickSort** *is $O(n \log n)$.*

More generally, for any constant c, there exist a constant d, such that the probability that **QuickSort** *recursion depth for any element exceeds $d \ln n$ is smaller than $1/n^c$.*

Specifically, for any $t \geq 1$, we have that probability that the recursion depth for any element exceeds $t \cdot d \ln n$ is smaller than $1/n^{t \cdot c}$.

Proof: Let us do the last part (but the reader is encouraged to skip this on first reading). Setting $M = 32t \ln n$, we get that the probability that an element has depth exceeds M, requires that in M coin flips we get at most $h = 4 \ln n$ heads. That is, if Y is the sum of the coin flips, where we get $+1$ for head, and -1 for tails, then Y needs to be smaller than $-(M-h)+h = -M+2h$. By symmetry, this is equal to the probability that $Y \geq \Delta = M - 2h$. By Theorem 10.2.3 below, the probability for that is

$$\begin{aligned}\mathbf{Pr}\Big[Y \geq \Delta\Big] &\leq \exp\Big(-\Delta^2/2M\Big) = \exp\left(-\frac{(M-2h)^2}{2M}\right) = \exp\left(-\frac{(32t-8)^2 \ln^2 n}{128t \ln n}\right) \\ &= \exp\left(-\frac{(4t-1)^2 \ln n}{2t}\right) \leq \exp\left(-\frac{3t^2 \ln n}{t}\right) \leq \frac{1}{n^{3t}}.\end{aligned}$$

■

Of course, the same result holds for the algorithm **MatchNutsAndBolts** for matching nuts and bolts.

10.1.2 An alternative proof of the high probability of QuickSort

Consider a set of T of the n items to be sorted, and consider a specific element $t \in T$. Let X_i be the size of the input in the ith level of recursion that contains t. We know that $X_0 = n$, and

$$\mathbf{E}\Big[X_i \;\Big|\; X_{i-1}\Big] \leq \frac{1}{2}\frac{3}{4}X_{i-1} + \frac{1}{2}X_{i-1} \leq \frac{7}{8}X_{i-1}.$$

Indeed, with probability $1/2$ the pivot is the middle of the subproblem; that is, its rank is between $X_{i-1}/4$ and $(3/4)X_{i-1}$ (and then the subproblem has size $\leq X_{i-1}(3/4)$), and with probability $1/2$ the subproblem might has not shrank significantly (i.e., we pretend it did not shrink at all).

Now, observe that for any two random variables we have that $\mathbf{E}\Big[X\Big] = \mathbf{E}_y\Big[\mathbf{E}\Big[X\,\Big|Y=y\Big]\,\Big]$, see Lemma 9.1.7$_{\text{p78}}$.. As such, we have that

$$\mathbf{E}\Big[X_i\Big] = \mathop{\mathbf{E}}_{y}\left[\mathbf{E}\Big[X_i \;\Big|\; X_{i-1}=y\Big]\right] \leq \mathop{\mathbf{E}}_{X_{i-1}=y}\Big[\frac{7}{8}y\Big] = \frac{7}{8}\,\mathbf{E}\Big[X_{i-1}\Big] \leq \left(\frac{7}{8}\right)^i \mathbf{E}[X_0] = \left(\frac{7}{8}\right)^i n.$$

In particular, consider $M = 8\log_{8/7} n$. We have that

$$\mu = \mathbf{E}\Big[X_M\Big] \leq \left(\frac{7}{8}\right)^M n \leq \frac{1}{n^8} n = \frac{1}{n^7}.$$

Of course, t participates in more than M recursive calls, if and only if $X_M \geq 1$. However, by Markov's inequality (Theorem 10.2.1), we have that

$$\mathbf{Pr}\left[\begin{array}{c}\text{element } t \text{ participates}\\ \text{in more than } M \text{ recursive calls}\end{array}\right] \leq \mathbf{Pr}[X_M \geq 1] \leq \frac{\mathbf{E}[X_M]}{1} \leq \frac{1}{n^7},$$

as desired. That is, we proved that the probability that any element of the input T participates in more than M recursive calls is at most $n(1/n^7) \leq 1/n^6$.

10.2 Chernoff inequality

10.2.1 Preliminaries

Theorem 10.2.1 (*Markov's Inequality*.). *For a non-negative variable X, and $t > 0$, we have:*

$$\mathbf{Pr}\Big[X \geq t\Big] \leq \frac{\mathbf{E}[X]}{t}.$$

Proof: Assume that this is false, and there exists $t_0 > 0$ such that $\mathbf{Pr}[X \geq t_0] > \dfrac{\mathbf{E}[X]}{t_0}$. However,

$$\begin{aligned}\mathbf{E}\Big[X\Big] &= \sum_x x \cdot \mathbf{Pr}\Big[X=x\Big] = \sum_{x<t_0} x \cdot \mathbf{Pr}\Big[X=x\Big] + \sum_{x\geq t_0} x \cdot \mathbf{Pr}\Big[X=x\Big]\\ &\geq 0 + t_0 \cdot \mathbf{Pr}\Big[X \geq t_0\Big] > 0 + t_0 \cdot \frac{\mathbf{E}[X]}{t_0} = \mathbf{E}\Big[X\Big],\end{aligned}$$

a contradiction. ■

We remind the reader that two random variables X and Y are *independent* if for any x, y we have that

$$\mathbf{Pr}\Big[(X=x)\cap(Y=y)\Big] = \mathbf{Pr}\Big[X=x\Big]\cdot\mathbf{Pr}\Big[Y=y\Big].$$

The following claim is easy to verify, and we omit the easy proof.

Claim 10.2.2. *If X and Y are independent, then $\mathbf{E}[XY] = \mathbf{E}[X]\,\mathbf{E}[Y]$.*

If X and Y are independent then $Z = e^X$ and $W = e^Y$ are also independent variables.

10.2.2 Chernoff inequality

Theorem 10.2.3 (*Chernoff inequality*). *Let $X_1, \ldots, X_n$ be n independent random variables, such that $\mathbf{Pr}[X_i = 1] = \mathbf{Pr}[X_i = -1] = \frac{1}{2}$, for $i = 1, \ldots, n$. Let $Y = \sum_{i=1}^{n} X_i$. Then, for any $\Delta > 0$, we have*

$$\mathbf{Pr}\Big[Y \geq \Delta\Big] \leq \exp\Big(-\Delta^2/2n\Big).$$

Proof: Clearly, for an arbitrary t, to be specified shortly, we have

$$\mathbf{Pr}\Big[Y \geq \Delta\Big] = \mathbf{Pr}\Big[tY \geq t\Delta\Big] = \mathbf{Pr}\Big[\exp(tY) \geq \exp(t\Delta)\Big] \leq \frac{\mathbf{E}\Big[\exp(tY)\Big]}{\exp(t\Delta)}, \tag{10.2}$$

where the first part follows since $\exp(\cdot)$ preserve ordering, and the second part follows by Markov's inequality (Theorem 10.2.1).

Observe that, by the definition of $\mathbf{E}[\cdot]$ and by the Taylor expansion of $\exp(\cdot)$, we have

$$\begin{aligned}
\mathbf{E}\Big[\exp(tX_i)\Big] &= \frac{1}{2}e^{t} + \frac{1}{2}e^{-t} = \frac{e^t + e^{-t}}{2} \\
&= \frac{1}{2}\left(1 + \frac{t}{1!} + \frac{t^2}{2!} + \frac{t^3}{3!} + \cdots\right) \\
&\quad + \frac{1}{2}\left(1 - \frac{t}{1!} + \frac{t^2}{2!} - \frac{t^3}{3!} + \cdots\right) \\
&= \left(1 + \frac{t^2}{2!} + + \cdots + \frac{t^{2k}}{(2k)!} + \cdots\right).
\end{aligned}$$

Now, $(2k)! = k!(k+1)(k+2)\cdots 2k \geq k!2^k$, and thus

$$\mathbf{E}\Big[\exp(tX_i)\Big] = \sum_{i=0}^{\infty} \frac{t^{2i}}{(2i)!} \leq \sum_{i=0}^{\infty} \frac{t^{2i}}{2^i(i!)} = \sum_{i=0}^{\infty} \frac{1}{i!}\left(\frac{t^2}{2}\right)^i = \exp\left(\frac{t^2}{2}\right),$$

again, by the Taylor expansion of $\exp(\cdot)$. Next, by the independence of the X_is, we have

$$\begin{aligned}
\mathbf{E}\Big[\exp(tY)\Big] &= \mathbf{E}\left[\exp\left(\sum_i tX_i\right)\right] = \mathbf{E}\left[\prod_i \exp(tX_i)\right] = \prod_{i=1}^{n} \mathbf{E}\Big[\exp(tX_i)\Big] \\
&\leq \prod_{i=1}^{n} \exp\left(\frac{t^2}{2}\right) = \exp\left(\frac{nt^2}{2}\right).
\end{aligned}$$

We have, by Eq. (10.2), that

$$\mathbf{Pr}\Big[Y \geq \Delta\Big] \leq \frac{\mathbf{E}\Big[\exp(tY)\Big]}{\exp(t\Delta)} \leq \frac{\exp\Big(\frac{nt^2}{2}\Big)}{\exp(t\Delta)} = \exp\left(\frac{nt^2}{2} - t\Delta\right).$$

Next, we select the value of t that minimizes the right term in the above inequality. Easy calculation shows that the right value is $t = \Delta/n$. We conclude that

$$\mathbf{Pr}\Big[Y \geq \Delta\Big] \leq \exp\left(\frac{n}{2}\left(\frac{\Delta}{n}\right)^2 - \frac{\Delta}{n}\Delta\right) = \exp\left(-\frac{\Delta^2}{2n}\right).$$

■

Note, the above theorem states that

$$\mathbf{Pr}\Big[Y \geq \Delta\Big] = \sum_{i=\Delta}^{n} \mathbf{Pr}\Big[Y = i\Big] = \sum_{i=n/2+\Delta/2}^{n} \frac{\binom{n}{i}}{2^n} \leq \exp\left(-\frac{\Delta^2}{2n}\right),$$

since $Y = \Delta$ means that we got $n/2 + \Delta/2$ times $+1$s and $n/2 - \Delta/2$ times (-1)s.

By the symmetry of Y, we get the following corollary.

Corollary 10.2.4. *Let $X_1, \ldots, X_n$ be n independent random variables, such that $\mathbf{Pr}[X_i = 1] = \mathbf{Pr}[X_i = -1] = \frac{1}{2}$, for $i = 1, \ldots, n$. Let $Y = \sum_{i=1}^{n} X_i$. Then, for any $\Delta > 0$, we have*

$$\mathbf{Pr}\Big[|Y| \geq \Delta\Big] \leq 2\exp\left(-\frac{\Delta^2}{2n}\right).$$

By easy manipulation, we get the following result.

Corollary 10.2.5. *Let $X_1, \ldots, X_n$ be n independent coin flips, such that $\mathbf{Pr}[X_i = 1] = \mathbf{Pr}[X_i = 0] = \frac{1}{2}$, for $i = 1, \ldots, n$. Let $Y = \sum_{i=1}^{n} X_i$. Then, for any $\Delta > 0$, we have*

$$\mathbf{Pr}\left[\frac{n}{2} - Y \geq \Delta\right] \leq \exp\left(-\frac{2\Delta^2}{n}\right) \quad \text{and} \quad \mathbf{Pr}\left[Y - \frac{n}{2} \geq \Delta\right] \leq \exp\left(-\frac{2\Delta^2}{n}\right).$$

In particular, we have $\mathbf{Pr}\left[\left|Y - \frac{n}{2}\right| \geq \Delta\right] \leq 2\exp\left(-\frac{2\Delta^2}{n}\right)$.

Proof: Transform X_i into the random variable $Z_i = 2X_i - 1$, and now use Theorem 10.2.3 on the new random variables $Z_1, \ldots, Z_n$. ■

Lemma 10.1.1 (Restatement.) *In a sequence of M coin flips, the probability that the number of ones is smaller than $L \leq M/4$ is at most* $\exp(-M/8)$.

Proof: Let $Y = \sum_{i=1}^{m} X_i$ the sum of the M coin flips. By the above corollary, we have:

$$\mathbf{Pr}\Big[Y \leq L\Big] = \mathbf{Pr}\left[\frac{M}{2} - Y \geq \frac{M}{2} - L\right] = \mathbf{Pr}\left[\frac{M}{2} - Y \geq \Delta\right],$$

where $\Delta = M/2 - L \geq M/4$. Using the above Chernoff inequality, we get

$$\mathbf{Pr}\Big[Y \leq L\Big] \leq \exp\left(-\frac{2\Delta^2}{M}\right) \leq \exp(-M/8).$$

■

10.2.2.1 The Chernoff Bound — General Case

Here we present the Chernoff bound in a more general settings.

Problem 10.2.6. Let $X_1, \ldots X_n$ be n independent Bernoulli trials, where

$$\mathbf{Pr}\Big[X_i = 1\Big] = p_i \qquad \text{and} \qquad \mathbf{Pr}\Big[X_i = 0\Big] = 1 - p_i,$$

and let denote

$$Y = \sum_i X_i \qquad \mu = \mathbf{E}[Y]\,.$$

Question: what is the probability that $Y \geq (1+\delta)\mu$.

Theorem 10.2.7 (Chernoff inequality). *For any $\delta > 0$,*

$$\mathbf{Pr}\Big[Y > (1+\delta)\mu\Big] < \left(\frac{e^{\delta}}{(1+\delta)^{1+\delta}}\right)^{\mu}.$$

Or in a more simplified form, for any $\delta \leq 2e-1$,

$$\mathbf{Pr}\Big[Y > (1+\delta)\mu\Big] < \exp\Big(-\mu\delta^2/4\Big)\,, \tag{10.3}$$

and

$$\mathbf{Pr}\Big[Y > (1+\delta)\mu\Big] < 2^{-\mu(1+\delta)},$$

for $\delta \geq 2e-1$.

Theorem 10.2.8. *Under the same assumptions as the theorem above, we have*

$$\mathbf{Pr}\Big[Y < (1-\delta)\mu\Big] \leq \exp\left(-\mu\frac{\delta^2}{2}\right).$$

The proofs of those more general form, follows the proofs shown above, and are omitted. The interested reader can get the proofs from:

http://www.uiuc.edu/~sariel/teach/2002/a/notes/07_chernoff.ps

10.3 Treaps

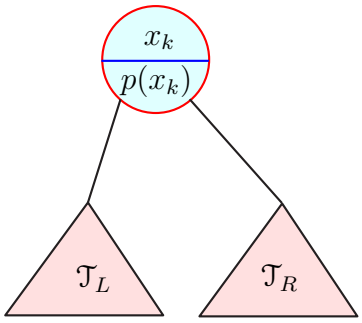
Anybody that ever implemented a balanced binary tree, knows that it can be very painful. A natural question, is whether we can use randomization to get a simpler data-structure with good performance.

10.3.1 Construction

The key observation is that many of data-structures that offer good performance for balanced binary search trees, do so by storing additional information to help in how to balance the tree. As such, the key Idea is that for every element x inserted into the data-structure, randomly choose a priority $p(x)$; that is, $p(x)$ is chosen uniformly and randomly in the range $[0, 1]$.

So, for the set of elements $X = \{x_1, \ldots, x_n\}$, with (random) priorities $p(x_1), \ldots, p(x_n)$, our purpose is to build a binary tree which is "balanced". So, let us pick the element x_k with the lowest priority in X, and make it the root of the tree. Now, we partition X in the natural way:

(A) L: set of all the numbers smaller than x_k in X, and
(B) R: set of all the numbers larger than x_k in X.



We can now build recursively the trees for L and R, and let denote them by $\mathcal{T}_L$ and $\mathcal{T}_R$. We build the natural tree, by creating a node for x_k, having $\mathcal{T}_L$ its left child, and $\mathcal{T}_R$ as its right child.

We call the resulting tree a ***treap***. As it is a tree over the elements, and a heap over the priorities; that is, TREAP = TREE + HEAP.

Lemma 10.3.1. *Given n elements, the expected depth of a treap $\mathcal{T}$ defined over those elements is $O(\log(n))$. Furthermore, this holds with high probability; namely, the probability that the depth of the treap would exceed $c \log n$ is smaller than $\delta = n^{-d}$, where d is an arbitrary constant, and c is a constant that depends on d.*[①]

Furthermore, the probability that $\mathcal{T}$ has depth larger than $ct \log(n)$, for any $t \geq 1$, is smaller than n^{-dt}.

Proof: Observe, that every element has equal probability to be in the root of the treap. Thus, the structure of a treap, is identical to the recursive tree of **QuickSort**. Indeed, imagine that instead of picking the pivot uniformly at random, we instead pick the pivot to be the element with the lowest (random) priority. Clearly, these two ways of choosing pivots are equivalent. As such, the claim follows immediately from our analysis of the depth of the recursion tree of **QuickSort**, see Theorem 10.1.2$_{\text{p86}}$. ■

10.3.2 Operations

The following innocent observation is going to be the key insight in implementing operations on treaps:

Observation 10.3.2. *Given n distinct elements, and their (distinct) priorities, the treap storing them is uniquely defined.*

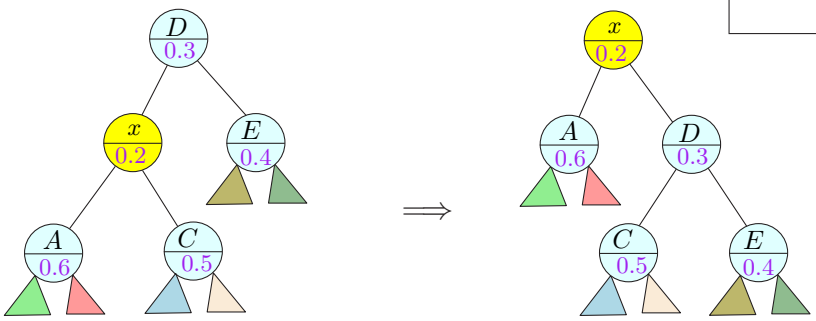
[①] That is, if we want to decrease the probability of failure, that is δ, we need to increase c.

10.3.2.1 Insertion

Given an element x to be inserted into an existing treap $\mathcal{T}$, insert it in the usual way into $\mathcal{T}$ (i.e., treat it a regular search binary tree). This takes $O(\text{height}(\mathcal{T}))$. Now, x is a leaf in the treap. Set x priority $p(x)$ to some random number $[0, 1]$. Now, while the new tree is a valid search tree, it is not necessarily still a valid treap, as x's priority might be smaller than its parent. So, we need to fix the tree around x, so that the priority property holds.

We call **RotateUp**(x) to do so. Specifically, if x parent is y, and $p(x) < p(y)$, we will rotate x up so that it becomes the parent of y. We repeatedly do it till x has a larger priority than its parent. The rotation operation takes constant time and plays around with priorities, and importantly, it preserves the binary search tree order. Here is a rotate right operation **RotateRight**(D):

```
RotateUp(x)
  y ← parent(x)
  while p(y) > p(x) do
      if y.left_child = x then
          RotateRight(y)
      else
          RotateLeft(y)
      y ← parent(x)
```



RotateLeft is the same tree rewriting operation done in the other direction.

In the end of this process, both the ordering property and the priority property holds. That is, we have a valid treap that includes all the old elements, and the new element. By Observation 10.3.2, since the treap is uniquely defined, we have updated the treap correctly. Since every time we do a rotation the distance of x from the root decrease by one, it follows that insertions takes $O(\text{height}(\mathcal{T}))$.

10.3.2.2 Deletion

Deletion is just an insertion done in reverse. Specifically, to delete an element x from a treap $\mathcal{T}$, set its priority to $+\infty$, and rotate it down it becomes a leaf. The only tricky observation is that you should rotate always so that the child with the lower priority becomes the new parent. Once x becomes a leaf deleting it is trivial - just set the pointer pointing to it in the tree to null.

10.3.2.3 Split

Given an element x stored in a treap $\mathcal{T}$, we would like to split $\mathcal{T}$ into two treaps – one treap $\mathcal{T}_{\leq}$ for all the elements smaller or equal to x, and the other treap $\mathcal{T}_{>}$ for all the elements

larger than x. To this end, we set x priority to $-\infty$, fix the priorities by rotating x up so it becomes the root of the treap. The right child of x is the treap $\mathcal{T}_{>}$, and we disconnect it from $\mathcal{T}$ by setting x right child pointer to null. Next, we restore x to its real priority, and rotate it down to its natural location. The resulting treap is $\mathcal{T}_{\leq}$. This again takes time that is proportional to the depth of the treap.

10.3.2.4 Meld

Given two treaps $\mathcal{T}_L$ and $\mathcal{T}_R$ such that all the elements in $\mathcal{T}_L$ are smaller than all the elements in $\mathcal{T}_R$, we would like to merge them into a single treap. Find the largest element x stored in $\mathcal{T}_L$ (this is just the element stored in the path going only right from the root of the tree). Set x priority to $-\infty$, and rotate it up the treap so that it becomes the root. Now, x being the largest element in $\mathcal{T}_L$ has no right child. Attach $\mathcal{T}_R$ as the right child of x. Now, restore x priority to its original priority, and rotate it back so the priorities properties hold.

10.3.3 Summery

Theorem 10.3.3. *Let $\mathcal{T}$ be a treap, initialized to an empty treap, and undergoing a sequence of $m = n^c$ insertions, where c is some constant. The probability that the depth of the treap in any point in time would exceed $d \log n$ is $\leq 1/n^f$, where d is an arbitrary constant, and f is a constant that depends only c and d.*

In particular, a treap can handle insertion/deletion in $O(\log n)$ time with high probability.

Proof: Since the first part of the theorem implies that with high probability all these treaps have logarithmic depth, then this implies that all operations takes logarithmic time, as an operation on a treap takes at most the depth of the treap.

As for the first part, let $\mathcal{T}_1, \ldots, \mathcal{T}_m$ be the sequence of treaps, where $\mathcal{T}_i$ is the treap after the ith operation. Similarly, let X_i be the set of elements stored in $\mathcal{T}_i$. By Lemma 10.3.1, the probability that $\mathcal{T}_i$ has large depth is tiny. Specifically, we have that

$$\alpha_i = \mathbf{Pr}\Big[\mathrm{depth}(\mathcal{T}_i) > tc' \log n^c\Big] = \mathbf{Pr}\left[\mathrm{depth}(\mathcal{T}_i) > c't\left(\frac{\log n^c}{\log |\mathcal{T}_i|}\right) \cdot \log |\mathcal{T}_i|\right] \leq \frac{1}{n^{t \cdot c}},$$

as a tedious and boring but straightforward calculation shows. Picking t to be sufficiently large, we have that the probability that the ith treap is too deep is smaller than $1/n^{f+c}$. By the union bound, since there are n^c treaps in this sequence of operations, it follows that the probability of any of these treaps to be too deep is at most $1/n^f$, as desired. ∎

10.4 Bibliographical Notes

Chernoff inequality was a rediscovery of Bernstein inequality, which was published in 1924 by Sergei Bernstein. Treaps were invented by Siedel and Aragon [SA96]. Experimental evidence suggests that Treaps performs reasonably well in practice, despite their simplicity,

see for example the comparison carried out by Cho and Sahni [CS00]. Implementations of treaps are readily available. An old implementation I wrote in `C` is available here: `http://valis.cs.uiuc.edu/blog/?p=6060`.

Chapter 11

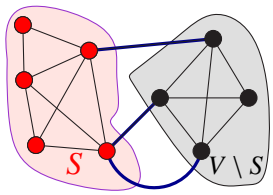
Min Cut

I built on the sand
And it tumbled down,
I built on a rock
And it tumbled down.
Now when I build, I shall begin
With the smoke from the chimney.
– Leopold Staff, Foundations.

11.1 Min Cut

11.1.1 Problem Definition

Let $\mathsf{G} = (V, E)$ be undirected graph with n vertices and m edges. We are interested in ***cuts*** in G.



Definition 11.1.1. A ***cut*** in G is a partition of the vertices of V into two sets S and $V \setminus S$, where the edges of the cut are

$$(S, V \setminus S) = \left\{ uv \;\middle|\; u \in S, v \in V \setminus S, \text{ and } uv \in E \right\},$$

where $S \neq \emptyset$ and $V \setminus S \neq \emptyset$. We will refer to the number of edges in the cut $(S, V \setminus S)$ as the *size of the cut*. For an example of a cut, see figure on the right.

We are interested in the problem of computing the ***minimum cut*** (i.e., ***mincut***), that is, the cut in the graph with minimum cardinality. Specifically, we would like to find the set $S \subseteq V$ such that $(S, V \setminus S)$ is as small as possible, and S is neither empty nor $V \setminus S$ is empty.

11.1.2 Some Definitions

We remind the reader of the following concepts. The ***conditional probability*** of X given Y is $\mathbf{Pr}\Big[X = x \,|Y = y\Big] = \mathbf{Pr}\Big[(X = x) \cap (Y = y)\Big] / \mathbf{Pr}\Big[Y = y\Big]$. An equivalent, useful restatement of this is that

$$\mathbf{Pr}\Big[(X = x) \cap (Y = y)\Big] = \mathbf{Pr}\Big[X = x \;\Big|\; Y = y\Big] \cdot \mathbf{Pr}[Y = y]\,. \tag{11.1}$$

Two events X and Y are ***independent***, if $\mathbf{Pr}\Big[X = x \cap Y = y\Big] = \mathbf{Pr}\Big[X = x\Big] \cdot \mathbf{Pr}\Big[Y = y\Big]$. In particular, if X and Y are independent, then $\mathbf{Pr}\Big[X = x \;\Big|\; Y = y\Big] = \mathbf{Pr}\Big[X = x\Big]$.

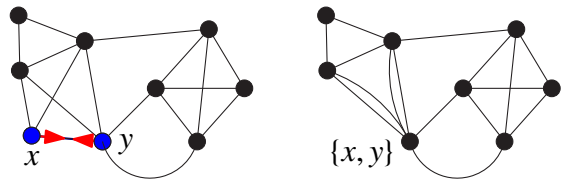
The following is easy to prove by induction using Eq. (11.1).

Lemma 11.1.2. *Let $\mathcal{E}_1, \ldots, \mathcal{E}_n$ be n events which are not necessarily independent. Then,*

$$\mathbf{Pr}\Big[\cap_{i=1}^{n} \mathcal{E}_i\Big] = \mathbf{Pr}\Big[\mathcal{E}_1\Big] * \mathbf{Pr}\Big[\mathcal{E}_2 \,|\mathcal{E}_1\Big] * \mathbf{Pr}\Big[\mathcal{E}_3 \;\Big|\; \mathcal{E}_1 \cap \mathcal{E}_2\Big] * \ldots * \mathbf{Pr}\Big[\mathcal{E}_n \;\Big|\; \mathcal{E}_1 \cap \ldots \cap \mathcal{E}_{n-1}\Big].$$

11.2 The Algorithm

The basic operation used by the algorithm is ***edge contraction***, depicted in Figure 11.1. We take an edge $e = xy$ in G and merge the two vertices into a single vertex. The new resulting graph is denoted by G/xy. Note, that we remove self loops created by the contraction. However, since the resulting graph is no longer a regular graph, it has parallel edges – namely, it is a multi-graph. We represent a multi-graph, as a regular graph with multiplicities on the edges. See Figure 11.2.

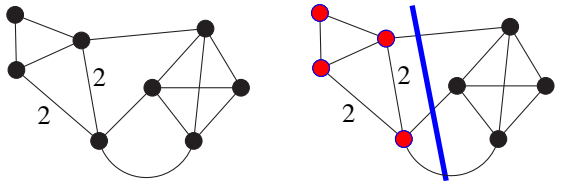


(a) (b)

Figure 11.1: (a) A contraction of the edge xy. (b) The resulting graph.

The edge contraction operation can be implemented in $O(n)$ time for a graph with n vertices. This is done by merging the adjacency lists of the two vertices being contracted, and then using hashing to do the fix-ups (i.e., we need to fix the adjacency list of the vertices that are connected to the two vertices).

Note, that the cut is now computed counting multiplicities (i.e., if e is in the cut and it has weight w, then the contribution of e to the cut weight is w).



(a) (b)

Figure 11.2: (a) A multi-graph. (b) A minimum cut in the resulting multi-graph.

Observation 11.2.1. *A set of vertices in G/xy corresponds to a set of vertices in the graph G. Thus a cut in G/xy always corresponds to a valid cut in G. However, there are cuts in G that do not exist in G/xy. For example, the cut $S = \{x\}$, does not exist in G/xy. As such, the size of the minimum cut in G/xy is at least as large as the minimum cut in G (as long as G/xy has at least one edge). Since any cut in G/xy has a corresponding cut of the same cardinality in G.*

Our algorithm works by repeatedly performing edge contractions. This is beneficial as this shrinks the underlying graph, and we would compute the cut in the resulting (smaller)

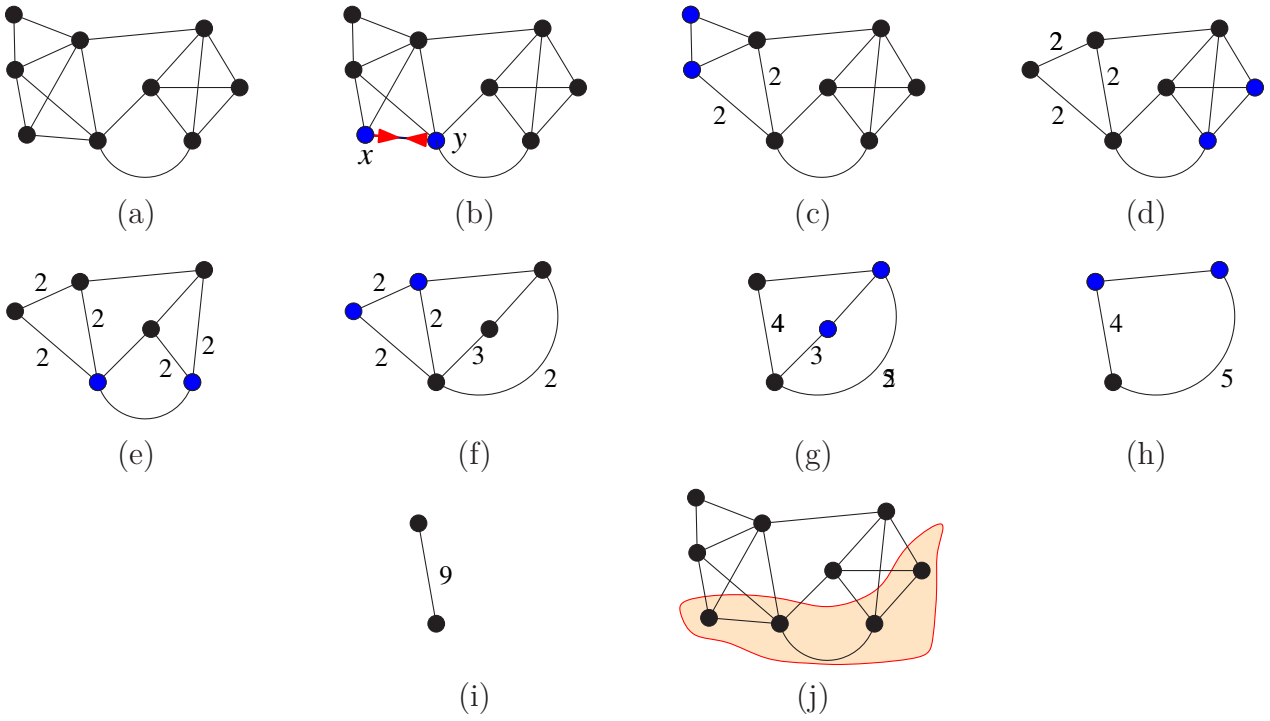


Figure 11.3: (a) Original graph. (b)–(j) a sequence of contractions in the graph, and (h) the cut in the original graph, corresponding to the single edge in (h). Note that the cut of (h) is not a mincut in the original graph.

graph. An "extreme" example of this, is shown in Figure 11.3, where we contract the graph into a single edge, which (in turn) corresponds to a cut in the original graph. (It might help the reader to think about each vertex in the contracted graph, as corresponding to a connected component in the original graph.)

Figure 11.3 also demonstrates the problem with taking this approach. Indeed, the resulting cut is not the minimum cut in the graph.

So, why did the algorithm fail to find the minimum cut in this case?[①] The failure occurs because of the contraction at Figure 11.3 (e), as we had contracted an edge in the minimum cut. In the new graph, depicted in Figure 11.3 (f), there is no longer a cut of size 3, and all cuts are of size 4 or more. Specifically, the algorithm succeeds only if it does not contract an edge in the minimum cut.

11.2.1 The resulting algorithm

Observation 11.2.2. *Let $e_1, \ldots, e_{n-2}$ be a sequence of edges in G, such that none of them is in the minimum cut, and such that $\mathsf{G}' = G/\{e_1, \ldots, e_{n-2}\}$ is a single multi-edge. Then,*

[①] Naturally, if the algorithm had succeeded in finding the minimum cut, this would have been **our** success.

this multi-edge corresponds to a minimum cut in G.

Note, that the claim in the above observation is only in one direction. We might be able to still compute a minimum cut, even if we contract an edge in a minimum cut, the reason being that a minimum cut is not unique. In particular, another minimum cut might survived the sequence of contractions that destroyed other minimum cuts.

Algorithm **MinCut**(G)
  $\mathsf{G}_0 \leftarrow G$
  $i = 0$
  while G_i has more than two vertices **do**
    $e_i \leftarrow$ random edge from $\mathsf{E}(\mathsf{G}_i)$
    $\mathsf{G}_{i+1} \leftarrow G_i / e_i$
    $i \leftarrow i + 1$
  Let $(S, V \setminus S)$ be the cut in the original graph corresponding to the single edge in G_i
  return $(S, V \setminus S)$.

Figure 11.4: The minimum cut algorithm.

Using Observation 11.2.2 in an algorithm is problematic, since the argumentation is circular, how can we find a sequence of edges that are not in the cut without knowing what the cut is? The way to slice the Gordian knot here, is to randomly select an edge at each stage, and contract this random edge.

See Figure 11.4 for the resulting algorithm **MinCut**.

11.2.1.1 On the art of randomly picking an edge

Every edge has a weight associated with it (which is the number of edges in the original graph it represents). A vertex weight is the total weight associated with it. We maintain during the contraction for every vertex the total weight of the edges adjacent to it. We need the following easy technical lemma.

Lemma 11.2.3. *Let $X \{x_1, \ldots, x_n\}$ be a set of n elements, and let $\omega(x_i)$ be an integer positive weight. One can pick randomly, in $O(n)$ time, an element from the set X, with the probability of picking x_i being $\omega(x_i)/W$, where $W = \sum_{i=1}^{n} \omega(x_i)$.*

Proof: Pick randomly a real number r in the range 0 to W. We also precompute the prefix sums $\beta_i = \sum_{k=1}^{i} \omega(x_k) = \beta_{i-1} + \omega(x_i)$, for $i = 1, \ldots, n$, which can be done in linear time. Now, find the first index i, such that $\beta_{i-1} < r \leq \beta_i$. Clearly, the probability of x_i to be picked is exactly $\omega(x_i)/W$. ∎

Now, we pick a vertex randomly according to the vertices weight in $O(n)$ time, and we pick randomly an edge adjacent to a vertex in $O(n)$ time (again, according to the weights of the edges). Thus, we uniformly sample an edge of the graph, with probability proportional to its weight, as desired.

11.2.2 Analysis

11.2.2.1 The probability of success

Naturally, if we are extremely lucky, the algorithm would never pick an edge in the mincut, and the algorithm would succeed. The ultimate question here is what is the probability of

success. If it is relatively "large" then this algorithm is useful since we can run it several times, and return the best result computed. If on the other hand, this probability is tiny, then we are working in vain since this approach would not work.

Lemma 11.2.4. *If a graph G has a minimum cut of size k and G has n vertices, then $|\mathsf{E}(\mathsf{G})| \geq kn/2$.*

Proof: Each vertex degree is at least k, otherwise the vertex itself would form a minimum cut of size smaller than k. As such, there are at least $\sum_{v \in V} \text{degree(v)}/2 \geq nk/2$ edges in the graph. ∎

Lemma 11.2.5. *If we pick in random an edge e from a graph G, then with probability at most $2/n$ it belong to the minimum cut.*

Proof: There are at least $nk/2$ edges in the graph and exactly k edges in the minimum cut. Thus, the probability of picking an edge from the minimum cut is smaller then $k/(nk/2) = 2/n$. ∎

The following lemma shows (surprisingly) that **MinCut** succeeds with reasonable probability.

Lemma 11.2.6. **MinCut** *outputs the mincut with probability* $\geq \dfrac{2}{n(n-1)}$.

Proof: Let $\mathcal{E}_i$ be the event that e_i is not in the minimum cut of G_i. By Observation 11.2.2, **MinCut** outputs the minimum cut if the events $\mathcal{E}_0, \ldots, \mathcal{E}_{n-3}$ all happen (namely, all edges picked are outside the minimum cut).

By Lemma 11.2.5, it holds $\mathbf{Pr}\Big[\mathcal{E}_i \Big| \mathcal{E}_0 \cap \mathcal{E}_1 \cap \ldots \cap \mathcal{E}_{i-1}\Big] \geq 1 - \dfrac{2}{|V(G_i)|} = 1 - \dfrac{2}{n-i}$. Implying that

$$\begin{aligned}
\Delta &= \mathbf{Pr}[\mathcal{E}_0 \cap \ldots \cap \mathcal{E}_{n-3}] \\
&= \mathbf{Pr}[\mathcal{E}_0] \cdot \mathbf{Pr}\Big[\mathcal{E}_1 \Big| \mathcal{E}_0\Big] \cdot \mathbf{Pr}\Big[\mathcal{E}_2 \Big| \mathcal{E}_0 \cap \mathcal{E}_1\Big] \cdot \ldots \cdot \mathbf{Pr}\Big[\mathcal{E}_{n-3} \Big| \mathcal{E}_0 \cap \ldots \cap \mathcal{E}_{n-4}\Big]
\end{aligned}$$

As such, we have

$$\begin{aligned}
\Delta &\geq \prod_{i=0}^{n-3} \left(1 - \frac{2}{n-i}\right) = \prod_{i=0}^{n-3} \frac{n-i-2}{n-i} \\
&= \frac{n-2}{n} * \frac{n-3}{n-1} * \frac{n-4}{n-2} \cdots \frac{2}{4} \cdot \frac{1}{3} \\
&= \frac{2}{n \cdot (n-1)}.
\end{aligned}$$

∎

11.2.2.2 Running time analysis.

Observation 11.2.7. **MinCut** *runs in* $O(n^2)$ *time.*

Observation 11.2.8. *The algorithm always outputs a cut, and the cut is not smaller than the minimum cut.*

Definition 11.2.9. (informal) Amplification is the process of running an experiment again and again till the things we want to happen, with good probability, do happen.

Let **MinCutRep** be the algorithm that runs **MinCut** $n(n-1)$ times and return the minimum cut computed in all those independent executions of **MinCut**.

Lemma 11.2.10. *The probability that* **MinCutRep** *fails to return the minimum cut is* < 0.14.

Proof: The probability of failure of **MinCut** to output the mincut in each execution is at most $1 - \frac{2}{n(n-1)}$, by Lemma 11.2.6. Now, **MinCutRep** fails, only if all the $n(n-1)$ executions of **MinCut** fail. But these executions are independent, as such, the probability to this happen is at most

$$\left(1 - \frac{2}{n(n-1)}\right)^{n(n-1)} \leq \exp\left(-\frac{2}{n(n-1)} \cdot n(n-1)\right) = \exp(-2) < 0.14,$$

since $1 - x \leq e^{-x}$ for $0 \leq x \leq 1$. ∎

Theorem 11.2.11. *One can compute the minimum cut in* $O(n^4)$ *time with constant probability to get a correct result. In* $O(n^4 \log n)$ *time the minimum cut is returned with high probability.*

11.3 A faster algorithm

The algorithm presented in the previous section is extremely simple. Which raises the question of whether we can get a faster algorithm[②]?

So, why **MinCutRep** needs so many executions? Well, the probability of success in the first ν iterations is

$$\begin{aligned}\mathbf{Pr}\left[\mathcal{E}_0 \cap \ldots \cap \mathcal{E}_{\nu-1}\right] &\geq \prod_{i=0}^{\nu-1}\left(1 - \frac{2}{n-i}\right) = \prod_{i=0}^{\nu-1}\frac{n-i-2}{n-i} \\ &= \frac{n-2}{n} * \frac{n-3}{n-1} * \frac{n-4}{n-2} \ldots = \frac{(n-\nu)(n-\nu-1)}{n \cdot (n-1)}. \end{aligned} \tag{11.2}$$

[②]This would require a more involved algorithm, that is life.

```
Contract( G, t )
  while |(G)| > t do
      Pick a random edge e in G.
      G ← G/e
  return G
```

```
FastCut(G = (V, E))
  G – multi-graph
begin
  n ← |V(G)|
  if n ≤ 6 then
      Compute (via brute force) minimum cut
      of G and return cut.
  t ← ⌈1 + n/√2⌉
  H_1 ← Contract(G, t)
  H_2 ← Contract(G, t)
  /* Contract is randomized!!! */
  X_1 ← FastCut(H_1),
  X_2 ← FastCut(H_2)
  return minimum cut out of X_1 and X_2.
end
```

Figure 11.5: **Contract**(G, t) shrinks G till it has only t vertices. **FastCut** computes the minimum cut using **Contract**.

Namely, this probability deteriorates very quickly toward the end of the execution, when the graph becomes small enough. (To see this, observe that for $\nu = n/2$, the probability of success is roughly $1/4$, but for $\nu = n - \sqrt{n}$ the probability of success is roughly $1/n$.)

So, the key observation is that as the graph get smaller the probability to make a bad choice increases. So, instead of doing the amplification from the outside of the algorithm, we will run the new algorithm more times when the graph is smaller. Namely, we put the amplification directly into the algorithm.

The basic new operation we use is **Contract**, depicted in Figure 11.5, which also depict the new algorithm **FastCut**.

Lemma 11.3.1. *The running time of* **FastCut**(G) *is* $O(n^2 \log n)$*, where* $n = |V(G)|$.

Proof: Well, we perform two calls to **Contract**(G, t) which takes $O(n^2)$ time. And then we perform two recursive calls on the resulting graphs. We have:

$$T(n) = O\Big(n^2\Big) + 2T\left(\frac{n}{\sqrt{2}}\right)$$

The solution to this recurrence is $O(n^2 \log n)$ as one can easily (and should) verify. ∎

Exercise 11.3.2. Show that one can modify **FastCut** so that it uses only $O(n^2)$ space.

Lemma 11.3.3. *The probability that* **Contract**$(G, n/\sqrt{2})$ *had not contracted the minimum cut is at least* $1/2$.

Namely, the probability that the minimum cut in the contracted graph is still a minimum cut in the original graph is at least $1/2$.

Proof: Just plug in $\nu = n - t = n - \left\lceil 1 + n/\sqrt{2} \right\rceil$ into Eq. (11.2). We have

$$\mathbf{Pr}[\mathcal{E}_0 \cap \ldots \cap \mathcal{E}_{n-t}] \geq \frac{t(t-1)}{n \cdot (n-1)} = \frac{\left\lceil 1 + n/\sqrt{2} \right\rceil \left(\left\lceil 1 + n/\sqrt{2} \right\rceil - 1 \right)}{n(n-1)} \geq \frac{1}{2}.$$

■

The following lemma bounds the probability of success. A more elegant argument is given in Section 11.3.1 below.

Lemma 11.3.4. **FastCut** *finds the minimum cut with probability larger than* $\Omega\left(1/\log n\right)$.

Proof: Do not read this proof – a considerably more elegant argument is given in Section 11.3.1.

Let $P(n)$ be the probability that the algorithm succeeds on a graph with n vertices.

The probability to succeed in the first call on H_1 is the probability that **Contract** did not hit the minimum cut (this probability is larger than $1/2$ by Lemma 11.3.3), times the probability that the algorithm succeeded on H_1 in the recursive call (those two events are independent). Thus, the probability to succeed on the call on H_1 is at least $(1/2) * P(n/\sqrt{2})$, Thus, the probability to fail on H_1 is $\leq 1 - \frac{1}{2} P\left(\frac{n}{\sqrt{2}}\right)$.

The probability to fail on both H_1 and H_2 is smaller than

$$\left(1 - \frac{1}{2} P\left(\frac{n}{\sqrt{2}}\right)\right)^2,$$

since H_1 and H_2 are being computed independently. Note that if the algorithm, say, fails on H_1 but succeeds on H_2 then it succeeds to return the mincut. Thus the above expression bounds the probability of failure. And thus, the probability for the algorithm to succeed is

$$P(n) \geq 1 - \left(1 - \frac{1}{2} P\left(\frac{n}{\sqrt{2}}\right)\right)^2 = P\left(\frac{n}{\sqrt{2}}\right) - \frac{1}{4}\left(P\left(\frac{n}{\sqrt{2}}\right)\right)^2.$$

We need to solve this recurrence. (This is very tedious, but since the details are non-trivial we provide the details of how to do so.) Divide both sides of the equation by $P\left(n/\sqrt{2}\right)$ we have:

$$\frac{P(n)}{P(n/\sqrt{2})} \geq 1 - \frac{1}{4} P(n/\sqrt{2}).$$

It is now easy to verify that this inequality holds for $P(n) \geq c/\log n$ (since the worst case is $P(n) = c/\log n$ we verify this inequality for this value). Indeed,

$$\frac{c/\log n}{c/\log(n/\sqrt{2})} \geq 1 - \frac{c}{4\log(n/\sqrt{2})}.$$

As such, letting $\Delta = \log n$, we have

$$\frac{\log n - \log \sqrt{2}}{\log n} = \frac{\Delta - \log \sqrt{2}}{\Delta} \geq \frac{4(\log n - \log \sqrt{2}) - c}{4(\log n - \log \sqrt{2})} = \frac{4(\Delta - \log \sqrt{2}) - c}{4(\Delta - \log \sqrt{2})}.$$

Equivalently, $4(\Delta - \log \sqrt{2})^2 \geq 4\Delta(\Delta - \log \sqrt{2}) - c\Delta$. Which implies $-8\Delta \log \sqrt{2} + 4\log^2 \sqrt{2} \geq -4\Delta \log \sqrt{2} - c\Delta$. Namely,

$$c\Delta - 4\Delta \log \sqrt{2} + 4\log^2 \sqrt{2} \geq 0,$$

which clearly holds for $c \geq 4 \log \sqrt{2}$.

We conclude, that the algorithm succeeds in finding the minimum cut in probability

$$\geq 2 \log 2 / \log n.$$

(Note that the base of the induction holds because we use brute force, and then $P(i) = 1$ for small i.) ■

Exercise 11.3.5. Prove, that running **FastCut** repeatedly $c \cdot \log^2 n$ times, guarantee that the algorithm outputs the minimum cut with probability $\geq 1 - 1/n^2$, say, for c a constant large enough.

Theorem 11.3.6. *One can compute the minimum cut in a graph G with n vertices in $O(n^2 \log^3 n)$ time. The algorithm succeeds with probability $\geq 1 - 1/n^2$.*

Proof: We do amplification on **FastCut** by running it $O(\log^2 n)$ times. The running time bound follows from Lemma 11.3.1. The bound on the probability follows from Lemma 11.3.4, and using the amplification analysis as done in Lemma 11.2.10 for **MinCutRep**. ■

11.3.1 On coloring trees and min-cut

Let T_h be a complete binary tree of height h. We randomly color its edges by black and white. Namely, for each edge we independently choose its color to be either black or white, with equal probability. We are interested in the event that there exists a path from the root of T_h to one of its leafs, that is all black. Let $\mathcal{E}_h$ denote this event, and let $\rho_h = \mathbf{Pr}[\mathcal{E}_h]$. Observe that $\rho_0 = 1$ and $\rho_1 = 3/4$ (see below).

To bound this probability, consider the root u of T_h and its two children u_l and u_r. The probability that there is a black path from u_l to one of its children is ρ_{h-1}, and as such, the probability that there is a black path from u through u_l to a leaf of the subtree of u_l is $\mathbf{Pr}\Big[\text{the edge } uu_l \text{ is colored black}\Big] \cdot \rho_{h-1} = \rho_{h-1}/2$. As such, the probability that there is no black path through u_l is $1 - \rho_{h-1}/2$. As such, the probability of not having a black path from u to a leaf (through either children) is $(1 - \rho_{h-1}/2)^2$. In particular, there desired probability, is the complement; that is

$$\rho_h = 1 - \left(1 - \frac{\rho_{h-1}}{2}\right)^2 = \frac{\rho_{h-1}}{2}\left(2 - \frac{\rho_{h-1}}{2}\right) = \rho_{h-1} - \frac{\rho_{h-1}^2}{4}.$$

Lemma 11.3.7. *We have that* $\rho_h \geq 1/(h+1)$.

Proof: The proof is by induction. For $h = 1$, we have $\rho_1 = 3/4 \geq 1/(1+1)$.

Observe that $\rho_h = f(\rho_{h-1})$ for $f(x) = x - x^2/4$, and $f'(x) = 1 - x/2$. As such, $f'(x) > 0$ for $x \in [0,1]$ and $f(x)$ is increasing in the range $[0,1]$. As such, by induction, we have that $\rho_h = f(\rho_{h-1}) \geq f\left(\frac{1}{(h-1)+1}\right) = \frac{1}{h} - \frac{1}{4h^2}$. We need to prove that $\rho_h \geq 1/(h+1)$, which is implied by the above if

$$\frac{1}{h} - \frac{1}{4h^2} \geq \frac{1}{h+1} \quad \Leftrightarrow \quad 4h(h+1) - (h+1) \geq 4h^2 \quad \Leftrightarrow \quad 4h^2 + 4h - h - 1 \geq 4h^2 \quad \Leftrightarrow \quad 3h \geq 1,$$

which trivially holds. ∎

The recursion tree for **FastCut** corresponds to such a coloring. Indeed, it is a binary tree as every call performs two recursive calls. Inside such a call, we independently perform two (independent) contractions reducing the given graph with n vertices to have $n/\sqrt{2}$ vertices. If this contraction succeeded (i.e., it did not hit the min-cut), then consider this edge to be colored by black (and white otherwise). Clearly, the algorithm succeeds, if and only if, there is black colored path from the root of the tree to the leaf. Since the tree has depth $H \leq 2 + \log_{\sqrt{2}} n$, and by Lemma 11.3.7, we have that the probability of **FastCut** to succeed is at least $1/(h+1) \geq 1/(3 + \log_{\sqrt{2}} n)$.

Galton-Watson processes. Imagine that you start with a single node. The node is going to have two children, and each child survives with probability half (independently). If a child survives it is going to have two children, and so on. Clearly, a single node give a rise to a random tree. The natural question is what is the probability that the original node has descendants h generations in the future. In the above we proved that this probability is at least $1/(h+1)$. See below for more details on this interpretation.

11.4 Bibliographical Notes

The **MinCut** algorithm was developed by David Karger during his PhD thesis in Stanford. The fast algorithm is a joint work with Clifford Stein. The basic algorithm of the mincut is described in [MR95, pages 7–9], the faster algorithm is described in [MR95, pages 289–295].

Galton-Watson process. The idea of using coloring of the edges of a tree to analyze **FastCut** might be new (i.e., Section 11.3.1). It is inspired by ***Galton-Watson processes*** (which is a special case of a branching process). The problem that initiated the study of these processes goes back to the 19th century [WG75]. Victorians were worried that aristocratic surnames were disappearing, as family names passed on only through the male children. As such, a family with no male children had its family name disappear. So, imagine the number of male children of a person is an independent random variable $X \in \{0, 1, 2, \ldots\}$.

Starting with a single person, its family (as far as male children are concerned) is a random tree with the degree of a node being distributed according to X. We continue recursively in constructing this tree, again, sampling the number of children for each current leaf according to the distribution of X. It is not hard to see that a family disappears if $\mathbf{E}[X] \leq 1$, and it has a constant probability of surviving if $\mathbf{E}[X] > 1$. In our case, X was the number of the two children of a node that their edges were colored black.

Of course, since infant mortality is dramatically down (as is the number of aristocrat males dying to maintain the British empire), the probability of family names to disappear is now much lower than it was in the 19th century. Interestingly, countries with family names that were introduced long time ago have very few surnames (i.e., Koreans have 250 surnames, and three surnames form 45% of the population). On the other hand, countries that introduced surnames more recently have dramatically more surnames (for example, the Dutch have surnames only for the last 200 years, and there are $68,000$ different family names).

Part III

Network Flow

Chapter 12

Network Flow

12.1 Network Flow

We would like to transfer as much "merchandise" as possible from one point to another. For example, we have a wireless network, and one would like to transfer a large file from s to t. The network have limited capacity, and one would like to compute the maximum amount of information one can transfer.

Specifically, there is a network and capacities associated with each connection in the network. The question is how much "flow" can you transfer from a source s into a sink t. Note, that here we think about the flow as being splitable, so that it can travel from the source to the sink along several parallel paths simultaneously. So, think about our network as being a network of pipe moving water from the source the sink (the capacities are how much water can a pipe transfer in a given unit of time). On the other hand, in the internet traffic is packet based and splitting is less easy to do.

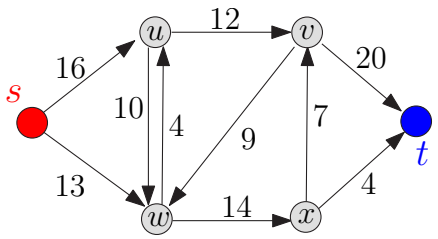
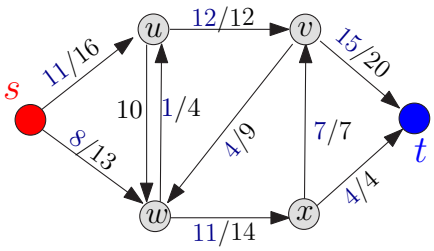


Figure 12.1: A network flow.

Definition 12.1.1. Let $\mathsf{G} = (V, E)$ be a ***directed*** graph. For every edge $(u \to v) \in E(\mathsf{G})$ we have an associated edge ***capacity*** $c(u, v)$, which is a non-negative number. If the edge $(u \to v) \notin G$ then $c(u, v) = 0$. In addition, there is a ***source*** vertex s and a target ***sink*** vertex t.

The entities G, s, t and $c(\cdot)$ together form a ***flow network*** or simply a ***network***. An example of such a flow network is depicted in Figure 12.1.



We would like to transfer as much flow from the source s to the sink t. Specifically, all the flow starts from the source vertex, and ends up in the sink. The flow on an edge is a non-negative quantity that can not exceed the capacity constraint for this edge. One possible flow is depicted on the left figure, where the numbers a/b on an edge denote a flow of a units on an edge with capacity at most b.

We next formalize our notation of a flow.

Definition 12.1.2 (flow). A ***flow*** in a network is a function $f(\cdot,\cdot)$ on the edges of G such that:

(A) ***Bounded by capacity***: For any edge $(u \to v) \in E$, we have $f(u,v) \leq c(u,v)$. Specifically, the amount of flow between u and v on the edge $(u \to v)$ never exceeds its capacity $c(u,v)$.

(B) ***Anti symmetry***: For any u, v we have $f(u,v) = -f(v,u)$.

(C) There are two special vertices: (i) the ***source*** vertex s (all flow starts from the source), and the ***sink*** vertex t (all the flow ends in the sink).

(D) ***Conservation of flow***: For any vertex $u \in V \setminus \{s,t\}$, we have $\sum_v f(u,v) = 0$.[①]

(Namely, for any internal node, all the flow that flows into a vertex leaves this vertex.)

The amount of flow (or simply ***flow***) of f, called the ***value*** of f, is $|f| = \sum_{v \in V} f(s,v)$.

Note, that a flow on edge can be negative (i.e., there is a positive flow flowing on this edge in the other direction).

Problem 12.1.3 (Maximum flow). Given a network G find the ***maximum flow*** in G. Namely, compute a legal flow f such that $|f|$ is maximized.

12.2 Some properties of flows and residual networks

For two sets $X, Y \subseteq V$, let $f(X,Y) = \sum_{x \in X, y \in Y} f(x,y)$. We will slightly abuse the notations and refer to $f\Big(\{v\}, S\Big)$ by $f(v,S)$, where $v \in V(\mathsf{G})$.

Observation 12.2.1. $|f| = f(s,V)$.

Lemma 12.2.2. *For a flow f, the following properties holds:*

(i) $\forall u \in V(\mathsf{G})$ *we have* $f(u,u) = 0$,

(ii) $\forall X \subseteq V$ *we have* $f(X,X) = 0$,

(iii) $\forall X, Y \subseteq V$ *we have* $f(X,Y) = -f(Y,X)$,

[①] This law for electric circuits is known as Kirchhoff's Current Law.

(iv) $\forall X, Y, Z \subseteq V$ *such that* $X \cap Y = \emptyset$ *we have that* $f(X \cup Y, Z) = f(X, Z) + f(Y, Z)$ *and* $f(Z, X \cup Y) = f(Z, X) + f(Z, Y)$.

(v) For all $u \in V \setminus \{s, t\}$, *we have* $f(u, V) = f(V, u) = 0$.

Proof: Property (i) holds since $(u \to u)$ it not an edge in the graph, and as such its flow is zero. As for property (ii), we have

$$\begin{aligned} f(X, X) &= \sum_{\{u,v\} \subseteq X, u \neq v} (f(u, v) + f(v, u)) + \sum_{u \in X} f(u, u) \\ &= \sum_{\{u,v\} \subseteq X, u \neq v} (f(u, v) - f(u, v)) + \sum_{u \in X} 0 = 0, \end{aligned}$$

by the anti-symmetry property of flow (Definition 12.1.2 (B)).

Property (iii) holds immediately by the anti-symmetry of flow, as

$$f(X, Y) = \sum_{x \in X, y \in Y} f(x, y) = - \sum_{x \in X, y \in Y} f(y, x) = -f(Y, X)\,.$$

(iv) This case follows immediately from definition.

Finally (v) is a restatement of the conservation of flow property. ■

Claim 12.2.3. $|f| = f(V, t)$.

Proof: We have:

$$\begin{aligned} |f| &= f(s, V) = f\Big(V \setminus (V \setminus \{s\}), V\Big) \\ &= f(V, V) - f(V \setminus \{s\}, V) \\ &= -f(V \setminus \{s\}, V) = f(V, V \setminus \{s\}) \\ &= f(V, t) + f(V, V \setminus \{s, t\}) \\ &= f(V, t) + \sum_{u \in V \setminus \{s,t\}} f(V, u) \\ &= f(V, t) + \sum_{u \in V \setminus \{s,t\}} 0 \\ &= f(V, t), \end{aligned}$$

since $f(V, V) = 0$ by Lemma 12.2.2 (i) and $f(V, u) = 0$ by Lemma 12.2.2 (iv). ■

Definition 12.2.4. Given capacity c and flow f, the ***residual capacity*** of an edge $(u \to v)$ is

$$c_f(u, v) = c(u, v) - f(u, v).$$

Intuitively, the residual capacity $c_f(u, v)$ on an edge $(u \to v)$ is the amount of unused capacity on $(u \to v)$. We can next construct a graph with all edges that are not being fully used by f, and as such can serve to improve f.

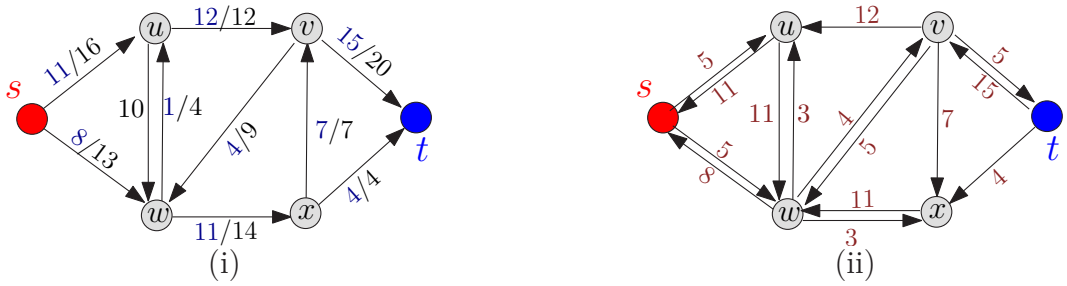


Figure 12.2: (i) A flow network, and (ii) the resulting residual network. Note, that $f(u,w) = -f(w,u) = -1$ and as such $c_f(u,w) = 10 - (-1) = 11$.

Definition 12.2.5. Given f, $\mathsf{G} = (V, E)$ and c, as above, the ***residual graph*** (or ***residual network***) of G and f is the the graph $\mathsf{G}_f = (V, E_f)$ where

$$E_f = \left\{ (u,v) \in V \times V \;\middle|\; c_f(u,v) > 0 \right\}.$$

Note, that by the definition of E_f, it might be that an edge $(u \to v)$ that appears in E might induce two edges in E_f. Indeed, consider an edge $(u \to v)$ such that $f(u,v) < c(u,v)$ and $(v \to u)$ is not an edge of G. Clearly, $c_f(u,v) = c(u,v) - f(u,v) > 0$ and $(u \to v) \in E_f$. Also,

$$c_f(v,u) = c(v,u) - f(v,u) = 0 - (-f(u,v)) = f(u,v),$$

since $c(v,u) = 0$ as $(v \to u)$ is not an edge of G. As such, $(v \to u) \in E_f$. This states that we can always reduce the flow on the edge $(u \to v)$ and this is interpreted as pushing flow on the edge $(v \to u)$. See Figure 12.2 for an example of a residual network.

Since every edge of G induces at most two edges in G_f, it follows that G_f has at most twice the number of edges of G; formally, $|E_f| \leq 2|E|$.

Lemma 12.2.6. *Given a flow f defined over a network G, then the residual network G_f together with c_f form a flow network.*

Proof: One need to verify that $c_f(\cdot)$ is always a non-negative function, which is true by the definition of E_f. ∎

The following lemma testifies that we can improve a flow f on G by finding a any legal flow h in the residual netowrk G_f.

Lemma 12.2.7. *Given a flow network $\mathsf{G}(V,E)$, a flow f in G, and h be a flow in G_f, where G_f is the residual network of f. Then $f + h$ is a (legal) flow in G and its capacity is $|f + h| = |f| + |h|$.*

Proof: By definition, we have $(f+h)(u,v) = f(u,v) + h(u,v)$ and thus $(f+h)(X,Y) = f(X,Y) + h(X,Y)$. We need to verify that $f+h$ is a legal flow, by verifying the properties required to it by Definition 12.1.2.

Anti symmetry holds since $(f+h)(u,v) = f(u,v) + h(u,v) = -f(v,u) - h(v,u) = -(f+h)(v,u)$.

Next, we verify that the flow $f+h$ is bounded by capacity. Indeed,

$$(f+h)(u,v) \leq f(u,v) + h(u,v) \leq f(u,v) + c_f(u,v) = f(u,v) + (c(u,v) - f(u,v)) = c(u,v).$$

For $u \in V - s - t$ we have $(f+h)(u,V) = f(u,V) + h(u,V) = 0 + 0 = 0$ and as such $f+h$ comply with the conservation of flow requirement.

Finally, the total flow is

$$|f+h| = (f+h)(s,V) = f(s,V) + h(s,V) = |f| + |h|.$$

■

Definition 12.2.8. For G and a flow f, a path π in G_f between s and t is an ***augmenting path***.

Note, that all the edges of π has positive capacity in G_f, since otherwise (by definition) they would not appear in E_f. As such, given a flow f and an augmenting path π, we can improve f by pushing a positive amount of flow along the augmenting path π. An augmenting path is depicted on the right, for the network flow of Figure 12.2.

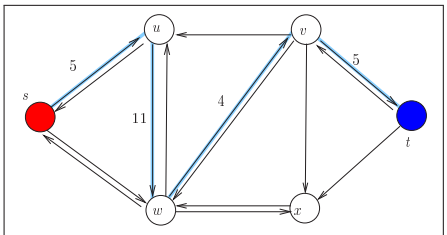


Figure 12.3: An augmenting path for the flow of Figure 12.2.

Definition 12.2.9. For an augmenting path π let $c_f(\pi)$ be the maximum amount of flow we can push through π. We call $c_f(\pi)$ the ***residual capacity*** of π. Formally,

$$c_f(\pi) = \min_{(u \to v) \in \pi} c_f(u,v).$$

We can now define a flow that realizes the flow along π. Indeed:

$$f_\pi(u,v) = \begin{cases} c_f(\pi) & \text{if } (u \to v) \text{ is in } \pi \\ -c_f(\pi) & \text{if } (v \to u) \text{ is in } \pi \\ 0 & \text{otherwise.} \end{cases}$$

Lemma 12.2.10. *For an augmenting path π, the flow f_π is a flow in G_f and $|f_\pi| = c_f(\pi) > 0$.*

We can now use such a path to get a larger flow:

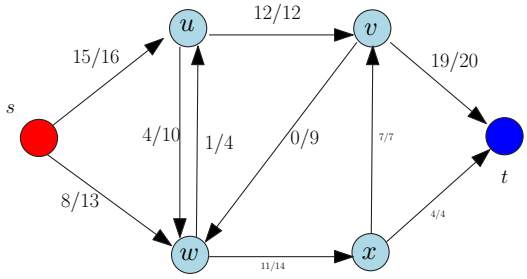
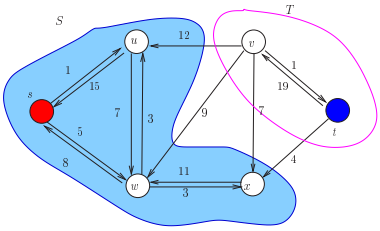


Figure 12.4: The flow resulting from applying the residual flow f_p of the path p of Figure 12.3 to the flow of Figure 12.2.

Lemma 12.2.11. *Let f be a flow, and let π be an augmenting path for f. Then $f + f_\pi$ is a "better" flow. Namely, $|f + f_\pi| = |f| + |f_\pi| > |f|$.*

Namely, $f + f_\pi$ is flow with larger value than f. Consider the flow in Figure 12.4.

Can we continue improving it? Well, if you inspect the residual network of this flow, depicted on the right. Observe that s is disconnected from t in this residual network. So, we are unable to push any more flow. Namely, we found a solution which is a local maximum solution for network flow. But is that a global maximum? Is this the maximum flow we are looking for?



12.3 The Ford-Fulkerson method

mtdFordFulkerson(G, c)
begin
  $f \leftarrow$ Zero flow on G
  while (G_f has augmenting path p) **do**
    (* `Recompute` G_f `for this check` *)
    $f \leftarrow f + f_p$
  return f
end

Given a network G with capacity constraints c, the above discussion suggest a simple and natural method to compute a maximum flow. This is known as the ***Ford-Fulkerson*** method for computing maximum flow, and is depicted on the left, we will refer to it as the **mtdFordFulkerson** method.

It is unclear that this method (and the reason we do not refer to it as an algorithm) terminates and reaches the global maximum flow. We address these problems shortly.

12.4 On maximum flows

We need several natural concepts.

Definition 12.4.1. A ***directed cut*** (S, T) in a flow network $\mathsf{G} = (V, E)$ is a partition of V into S and $T = V - S$, such that $s \in S$ and $t \in T$. We usually will refer to a directed cut as being a ***cut***.

The net ***flow of f across a cut*** (S, T) is $f(S, T) = \sum_{s \in S, t \in T} f(s, t)$.

The ***capacity*** of (S, T) is $c(S, T) = \sum_{s \in S, t \in T} c(s, t)$.

The ***minimum cut*** is the cut in G with the minimum capacity.

Lemma 12.4.2. *Let G,f,s,t be as above, and let (S, T) be a cut of G. Then $f(S, T) = |f|$.*

Proof: We have

$$f(S, T) = f(S, V) - f(S, S) = f(S, V) = f(s, V) + f(S - s, V) = f(s, V) = |f|,$$

since $T = V \setminus S$, and $f(S - s, V) = \sum_{u \in S - s} f(u, V) = 0$ by Lemma 12.2.2 (v) (note that u can not be t as $t \in T$). ■

Claim 12.4.3. *The flow in a network is upper bounded by the capacity of any cut (S, T) in G.*

Proof: Consider a cut (S, T). We have $|f| = f(S, T) = \sum_{u \in S, v \in T} f(u, v) \leq \sum_{u \in S, v \in T} c(u, v) = c(S, T)$. ∎

In particular, the maximum flow is bounded by the capacity of the minimum cut. Surprisingly, the maximum flow is exactly the value of the minimum cut.

Theorem 12.4.4 (Max-flow min-cut theorem). *If f is a flow in a flow network $\mathsf{G} = (V, E)$ with source s and sink t, then the following conditions are equivalent:*

(A) f is a maximum flow in G.

(B) The residual network G_f contains no augmenting paths.

(C) $|f| = c(S, T)$ for some cut (S, T) of G. And (S, T) is a minimum cut *in G.*

Proof: (A) ⇒ (B): By contradiction. If there was an augmenting path p then $c_f(p) > 0$, and we can generate a new flow $f + f_p$, such that $|f + f_p| = |f| + c_f(p) > |f|$. A contradiction as f is a maximum flow.

(B) ⇒ (C): Well, it must be that s and t are disconnected in G_f. Let

$$S = \left\{ v \;\middle|\; \text{Exists a path between } s \text{ and } v \text{ in } \mathsf{G}_f \right\}$$

and $T = V \setminus S$. We have that $s \in S$, $t \in T$, and for any $u \in S$ and $v \in T$ we have $f(u, v) = c(u, v)$. Indeed, if there were $u \in S$ and $v \in T$ such that $f(u, v) < c(u, v)$ then $(u \to v) \in E_f$, and v would be reachable from s in G_f, contradicting the construction of T.

This implies that $|f| = f(S, T) = c(S, T)$. The cut (S, T) must be a minimum cut, because otherwise there would be cut (S', T') with smaller capacity $c(S', T') < c(S, T) = f(S, T) = |f|$, On the other hand, by Lemma 12.4.3, we have $|f| = f(S', T') \leq c(S', T')$. A contradiction.

(C) ⇒ (A) Well, for any cut (U, V), we know that $|f| \leq c(U, V)$. This implies that if $|f| = c(S, T)$ then the flow can not be any larger, and it is thus a maximum flow. ∎

The above max-flow min-cut theorem implies that if **mtdFordFulkerson** terminates, then it had computed the maximum flow. What is still allusive is showing that the **mtdFordFulkerson** method always terminates. This turns out to be correct only if we are careful about the way we pick the augmenting path.

Chapter 13

Network Flow II - The Vengeance

13.1 Accountability

The comic in Figure 13.1 is by Jonathan Shewchuk and is referring to the Calvin and Hobbes comics.

People that do not know maximum flows: essentially everybody.

Average salary on earth $< \$5,000$

People that know maximum flow - most of them work in programming related jobs and make at least $\$10,000$ a year.

Salary of people that learned maximum flows: $> \$10,000$

Salary of people that did not learn maximum flows: $< \$5,000$

Salary of people that know Latin: 0 (unemployed).

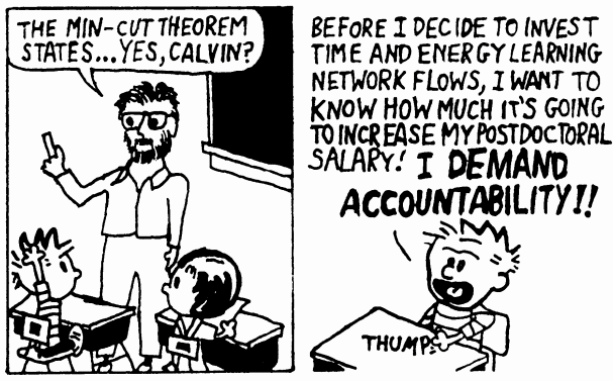


Figure 13.1: `http://www.cs.berkeley.edu/~jrs/`

Thus, by just learning maximum flows (and not knowing Latin) you can double your future salary!

13.2 The Ford-Fulkerson Method

The **mtdFordFulkerson** method is depicted on the right.

mtdFordFulkerson(G,s,t)
  Initialize flow f to zero
  while $\exists$ path π from s to t in G_f **do**
    $c_f(\pi) \leftarrow \min \left\{ c_f(u,v) \;\middle|\; (u \to v) \in \pi \right\}$
    for $\forall\, (u \to v) \in \pi$ **do**
      $f(u,v) \leftarrow f(u,v) + c_f(\pi)$
      $f(v,u) \leftarrow f(v,u) - c_f(\pi)$

Lemma 13.2.1. *If the capacities on the edges of G are integers, then* **mtdFordFulkerson** *runs in $O(m\,|f^*|)$ time, where $|f^*|$ is the amount of flow in the maximum flow and $m = |E(\mathsf{G})|$.*

Proof: Observe that the **mtdFordFulkerson** method performs only subtraction, addition and min operations. Thus, if it finds an augmenting path π, then $c_f(\pi)$ must be a *positive* integer number. Namely, $c_f(\pi) \geq 1$. Thus, $|f^*|$ must be an integer number (by induction), and each iteration of the algorithm improves the flow by at least 1. It follows that after $|f^*|$ iterations the algorithm stops. Each iteration takes $O(m+n) = O(m)$ time, as can be easily verified. ■

The following observation is an easy consequence of our discussion.

Observation 13.2.2 (Integrality theorem). *If the capacity function c takes on only integral values, then the maximum flow f produced by the* **mtdFordFulkerson** *method has the property that $|f|$ is integer-valued. Moreover, for all vertices u and v, the value of $f(u,v)$ is also an integer.*

13.3 The Edmonds-Karp algorithm

The **Edmonds-Karp** algorithm works by modifying the **mtdFordFulkerson** method so that it always returns the shortest augmenting path in G_f (i.e., path with smallest number of edges). This is implemented by finding π using **BFS** in G_f.

Definition 13.3.1. For a flow f, let $\delta_f(v)$ be the length of the shortest path from the source s to v in the residual graph G_f. Each edge is considered to be of length 1.

We will shortly prove that for any vertex $v \in V \setminus \{s,t\}$ the function $\delta_f(v)$, in the residual network G_f, increases monotonically with each flow augmentation. We delay proving this (key) technical fact (see Lemma 13.3.5 below), and first show its implications.

Lemma 13.3.2. *During the execution of the* **Edmonds-Karp** *algorithm, an edge $(u \to v)$ might disappear (and thus reappear) from G_f at most $n/2$ times throughout the execution of the algorithm, where $n = |V(\mathsf{G})|$.*

Proof: Consider an iteration when the edge $(u \to v)$ disappears. Clearly, in this iteration the edge $(u \to v)$ appeared in the augmenting path π. Furthermore, this edge was fully utilized; namely, $c_f(\pi) = c_f(uv)$, where f is the flow in the beginning of the iteration when it disappeared. We continue running **Edmonds-Karp** till $(u \to v)$ "magically" reappears. This

means that in the iteration before $(u \to v)$ reappeared in the residual graph, the algorithm handled an augmenting path σ that contained the edge $(v \to u)$. Let g be the flow used to compute σ. We have, by the monotonicity of $\delta(\cdot)$ [i.e., Lemma 13.3.5 below], that

$$\delta_g(u) = \delta_g(v) + 1 \geq \delta_f(v) + 1 = \delta_f(u) + 2$$

as **Edmonds-Karp** is always augmenting along the shortest path. Namely, the distance of s to u had increased by 2 between its disappearance and its (magical?) reappearance. Since $\delta_0(u) \geq 0$ and the maximum value of $\delta_?(u)$ is n, it follows that $(u \to v)$ can disappear and reappear at most $n/2$ times during the execution of the **Edmonds-Karp** algorithm. ■

The careful reader would observe that $\delta_?(u)$ might become infinity at some point during the algorithm execution (i.e., u is no longer reachable from s). If so, by monotonicity, the edge $(u \to v)$ would never appear again, in the residual graph, in any future iteration of the algorithm.

Observation 13.3.3. *Every time we add an augmenting path during the execution of the* **Edmonds-Karp** *algorithm, at least one edge disappears from the residual graph* $\mathsf{G}_?$*. Indeed, every edge that realizes the residual capacity of the augmenting path will disappear once we push the maximum possible flow along this path.*

Lemma 13.3.4. *The* **Edmonds-Karp** *algorithm handles at most* $O(nm)$ *augmenting paths before it stops. Its running time is* $O(nm^2)$*, where* $n = |V(\mathsf{G})|$ *and* $m = |E(\mathsf{G})|$*.*

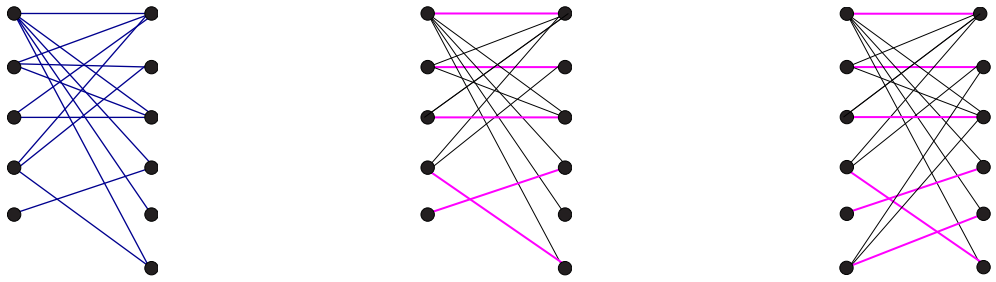
Proof: Every edge might disappear at most $n/2$ times during **Edmonds-Karp** execution, by Lemma 13.3.2. Thus, there are at most $nm/2$ edge disappearances during the execution of the **Edmonds-Karp** algorithm. At each iteration, we perform path augmentation, and at least one edge disappears along it from the residual graph. Thus, the **Edmonds-Karp** algorithm perform at most $O(mn)$ iterations.

Performing a single iteration of the algorithm boils down to computing an Augmenting path. Computing such a path takes $O(m)$ time as we have to perform BFS to find the augmenting path. It follows, that the overall running time of the algorithm is $O(nm^2)$. ■

We still need to prove the aforementioned monotonicity property. (This is the only part in our discussion of network flow where the argument gets a bit tedious. So bear with us, after all, you are going to double your salary here.)

Lemma 13.3.5. *If the* **Edmonds-Karp** *algorithm is run on a flow network* $\mathsf{G} = (V, E)$ *with source* s *and sink* t*, then for all vertices* $v \in V \setminus \{s, t\}$*, the shortest path distance* $\delta_f(v)$ *in the residual network* G_f *increases monotonically with each flow augmentation.*

Proof: Assume, for the sake of contradiction, that this is false. Consider the flow just after the first iteration when this claim failed. Let f denote the flow before this (fatal) iteration was performed, and let g be the flow after.



(i) (ii) (iii)

Figure 13.2: (i) A bipartite graph. (ii) A maximum matching in this graph. (iii) A perfect matching (in a different graph).

Let v be the vertex such that $\delta_g(v)$ is minimal, among all vertices for which the monotonicity fails. Formally, this is the vertex v where $\delta_g(v)$ is minimal and $\delta_g(v) < \delta_f(v)$.

Let $\pi = s \to \cdots \to u \to v$ be the shortest path in G_g from s to v. Clearly, $(u \to v) \in \mathsf{E}(\mathsf{G}_g)$, and thus $\delta_g(u) = \delta_g(v) - 1$.

By the choice of v it must be that $\delta_g(u) \geq \delta_f(u)$, since otherwise the monotonicity property fails for u, and u is closer to s than v in G_g, and this, in turn, contradicts our choice of v as being the closest vertex to s that fails the monotonicity property. There are now two possibilities:

(i) If $(u \to v) \in \mathsf{E}(\mathsf{G}_f)$ then

$$\delta_f(v) \leq \delta_f(u) + 1 \leq \delta_g(u) + 1 = \delta_g(v) - 1 + 1 = \delta_g(v).$$

This contradicts our assumptions that $\delta_f(v) > \delta_g(v)$.

(ii) If $(u \to v)$ is not in $\mathsf{E}(\mathsf{G}_f)$ then the augmenting path π used in computing g from f contains the edge $(v \to u)$. Indeed, the edge $(u \to v)$ reappeared in the residual graph G_g (while not being present in G_f). The only way this can happens is if the augmenting path π pushed a flow in the other direction on the edge $(u \to v)$. Namely, $(v \to u) \in \pi$. However, the algorithm always augment along the shortest path. Thus, since by assumption $\delta_g(v) < \delta_f(v)$, we have

$$\delta_f(u) = \delta_f(v) + 1 > \delta_g(v) = \delta_g(u) + 1,$$

by the definition of u.

Thus, $\delta_f(u) > \delta_g(u)$ (i.e., the monotonicity property fails for u) and $\delta_g(u) < \delta_g(v)$. A contradiction to the choice of v. ■

13.4 Applications and extensions for Network Flow

13.4.1 Maximum Bipartite Matching

Definition 13.4.1. For an undirected graph $\mathsf{G} = (V, E)$ a ***matching*** is a subset of edges $M \subseteq E$ such that for all vertices $v \in V$, at most one edge of M is incident on v.

A ***maximum matching*** is a matching M such that for any matching M' we have $|M| \geq |M'|$.

A matching is ***perfect*** if it involves all vertices. See Figure 13.2 for examples of these definitions.

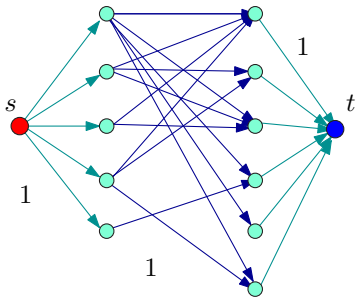


Figure 13.3

Theorem 13.4.2. *One can compute maximum bipartite matching using network flow in $O(nm^2)$ time, for a bipartite graph with n vertices and m edges.*

Proof: Given a bipartite graph G, we create a new graph with a new source on the left side and sink on the right, see Figure 13.3.

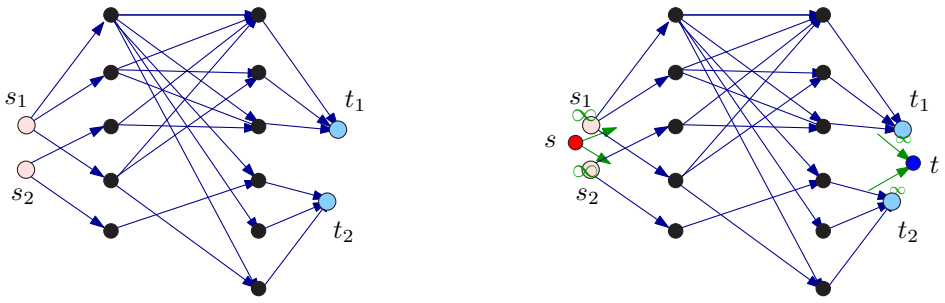
Direct all edges from left to right and set the capacity of all edges to 1. Let H be the resulting flow network. It is now easy to verify that by the Integrality theorem, a flow in H is either 0 or one on every edge, and thus a flow of value k in H is just a collection of k vertex disjoint paths between s and t in G, which corresponds to a matching in G of size k.

Similarly, given a matching of size k in G, it can be easily interpreted as realizing a flow in H of size k. Thus, computing a maximum flow in H results in computing a maximum matching in G. The running time of the algorithm is $O(nm^2)$. ■

13.4.2 Extension: Multiple Sources and Sinks

Given a flow network with several sources and sinks, how can we compute maximum flow on such a network?

The idea is to create a super source, that send all its flow to the old sources and similarly create a super sink that receives all the flow. See Figure 13.4. Clearly, computing flow in both networks in equivalent.



(i) (ii)

Figure 13.4: (i) A flow network with several sources and sinks, and (ii) an equivalent flow network with a single source and sink.

Chapter 14

Network Flow III - Applications

14.1 Edge disjoint paths

14.1.1 Edge-disjoint paths in a directed graphs

Question 14.1.1. *Given a graph* G *(either directed or undirected), two vertices* s *and* t*, and a parameter* k*, the task is to compute* k *paths from* s *to* t *in* G*, such that they are* **edge disjoint***; namely, these paths do not share an edge.*

To solve this problem, we will convert G (assume G is a directed graph for the time being) into a network flow graph H, such that every edge has capacity 1. Find the maximum flow in G (between s and t). We claim that the value of the maximum flow in the network H, is equal to the number of edge disjoint paths in G.

Lemma 14.1.2. *If there are* k *edge disjoint paths in* G *between* s *and* t*, then the maximum flow in* H *is at least* k*.*

Proof: Given k such edge disjoint paths, push one unit of flow along each such path. The resulting flow is legal in h and it has value k. ■

Definition 14.1.3 (0/1-flow). A flow f is a $0/1$***-flow*** if every edge has either no flow on it, or one unit of flow.

Lemma 14.1.4. *Let* f *be a* $0/1$ *flow in a network* H *with flow value* μ*. Then there are* μ *edge disjoint paths between* s *and* t *in* H*.*

Proof: By induction on the number of edges in H that has one unit of flow assigned to them by f. If $\mu = 0$ then there is nothing to prove.

Otherwise, start traversing the graph H from s traveling only along edges with flow 1 assigned to them by f. We mark such an edge as used, and do not allow one to travel on such an edge again. There are two possibilities:

(i) We reached the target vertex t. In this case, we take this path, add it to the set of output paths, and reduce the flow along the edges of the generated path π to 0. Let H' be the resulting flow network and f' the resulting flow. We have $|f'| = \mu - 1$, H' has less edges, and by induction, it has $\mu - 1$ edge disjoint paths in H' between s and t. Together with π this forms μ such paths.

(ii) We visit a vertex v for the second time. In this case, our traversal contains a cycle C, of edges in H that have flow 1 on them. We set the flow along the edges of C to 0 and use induction on the remaining graph (since it has less edges with flow 1 on them). The value of the flow f did not change by removing C, and as such it follows by induction that there are μ edge disjoint paths between s and t in H. ■

Since the graph G is simple, there are at most $n = |V(H)|$ edges that leave s. As such, the maximum flow in H is n. Thus, applying the Ford-Fulkerson algorithm, takes $O(mn)$ time. The extraction of the paths can also be done in linear time by applying the algorithm in the proof of Lemma 14.1.4. As such, we get:

Theorem 14.1.5. *Given a directed graph G with n vertices and m edges, and two vertices s and t, one can compute the maximum number of edge disjoint paths between s and t in H, in $O(mn)$ time.*

As a consequence we get the following cute result:

Lemma 14.1.6. *In a directed graph G with nodes s and t the maximum number of edge disjoint $s-t$ paths is equal to the minimum number of edges whose removal separates s from t.*

Proof: Let U be a collection of edge-disjoint paths from s to t in G. If we remove a set F of edges from G and separate s from t, then it must be that every path in U uses at least one edge of F. Thus, the number of edge-disjoint paths is bounded by the number of edges needed to be removed to separate s and t. Namely, $|U| \leq |F|$.

As for the other direction, let F be a set of edges thats its removal separates s and t. We claim that the set F form a cut in G between s and t. Indeed, let S be the set of all vertices in G that are reachable from s without using an edge of F. Clearly, if F is minimal then it must be all the edges of the cut (S, T) (in particular, if F contains some edge which is not in (S, T) we can remove it and get a smaller separating set of edges). In particular, the smallest set F with this separating property has the same size as the minimum cut between s and t in G, which is by the max-flow mincut theorem, also the maximum flow in the graph G (where every edge has capacity 1).

But then, by Theorem 14.1.5, there are $|F|$ edge disjoint paths in G (since $|F|$ is the amount of the maximum flow). ■

14.1.2 Edge-disjoint paths in undirected graphs

We would like to solve the s-t disjoint path problem for an undirected graph.

Problem 14.1.7. Given undirected graph G, s and t, find the maximum number of edge-disjoint paths in G between s and t.

The natural approach is to duplicate every edge in the undirected graph G, and get a (new) directed graph H. Next, apply the algorithm of Section 14.1.1 to H.

So compute for H the maximum flow f (where every edge has capacity 1). The problem is the flow f might use simultaneously the two edges $(u \to v)$ and $(v \to u)$. Observe, however, that in such case we can remove both edges from the flow f. In the resulting flow is legal and has the same value. As such, if we repeatedly remove those "double edges" from the flow f, the resulting flow f' has the same value. Next, we extract the edge disjoint paths from the graph, and the resulting paths are now edge disjoint in the original graph.

Lemma 14.1.8. *There are k edge-disjoint paths in an undirected graph G from s to t if and only if the maximum value of an $s-t$ flow in the directed version H of G is at least k. Furthermore, the Ford-Fulkerson algorithm can be used to find the maximum set of disjoint s-t paths in G in $O(mn)$ time.*

14.2 Circulations with demands

14.2.1 Circulations with demands

We next modify and extend the network flow problem. Let $G = (V, E)$ be a directed graph with capacities on the edges. Each vertex v has a demand d_v:

- $d_v > 0$: sink requiring d_v flow into this node.
- $d_v < 0$: source with $-d_v$ units of flow leaving it.
- $d_v = 0$: regular node.

Let S denote all the source vertices and T denote all the sink/target vertices.

For a concrete example of an instance of circulation with demands, see figure on the right.

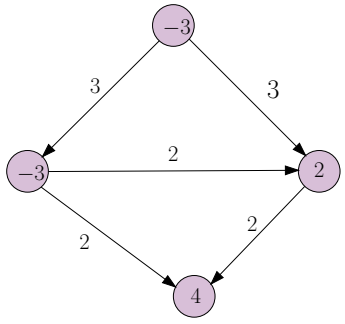


Figure 14.1: Instance of circulation with demands.

Definition 14.2.1. A ***circulation*** with demands $\{d_v\}$ is a function f that assigns nonnegative real values to the edges of G, such that:

- Capacity condition: $\forall e \in E$ we have $f(e) \leq c(e)$.
- Conservation condition: $\forall v \in V$ we have $f^{in}(v) - f^{out}(v) = d_v$.

Here, for a vertex v, let $f^{in}(v)$ denotes the flow into v and $f^{out}(v)$ denotes the flow out of v.

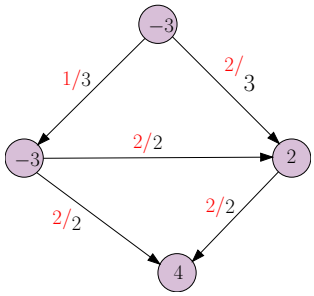


Figure 14.2: A valid circulation for the instance of Figure 14.1.

Problem 14.2.2. Is there a circulation that comply with the demand requirements?

See Figure 14.1 and Figure 14.2 for an example.

Lemma 14.2.3. *If there is a feasible circulation with demands $\{d_v\}$, then $\sum_v d_v = 0$.*

Proof: Since it is a circulation, we have that $d_v = f^{in}(v) - f^{out}(v)$. Summing over all vertices: $\sum_v d_v = \sum_v f^{in}(v) - \sum_v f^{out}(v)$. The flow on every edge is summed twice, one with positive sign, one with negative sign. As such,

$$\sum_v d_v = \sum_v f^{in}(v) - \sum_v f^{out}(v) = 0,$$

which implies the claim. ∎

In particular, this implies that there is a feasible solution only if

$$D = \sum_{v, d_v > 0} d_v = \sum_{v, d_v < 0} -d_v.$$

14.2.1.1 The algorithm for computing a circulation

The algorithm performs the following steps:

(A) $\mathsf{G} = (V, E)$ - input flow network with demands on vertices.
(B) Check that $D = \sum_{v, d_v > 0} d_v = \sum_{v, d_v < 0} -d_v$.
(C) Create a new super source s, and connect it to all the vertices v with $d_v < 0$. Set the capacity of the edge $s \to v$ to be $-d_v$.
(D) Create a new super target t. Connect to it all the vertices u with $d_u > 0$. Set capacity on the new edge $u \to t$ to be d_u.
(E) On the resulting network flow network H (which is a standard instance of network flow). Compute maximum flow on H from s to t. If it is equal to D, then there is a valid circulation, and it is the flow restricted to the original graph. Otherwise, there is no valid circulation.

Theorem 14.2.4. *There is a feasible circulation with demands $\{d_v\}$ in G if and only if the maximum s-t flow in H has value D. If all capacities and demands in G are integers, and there is a feasible circulation, then there is a feasible circulation that is integer valued.*

14.3 Circulations with demands and lower bounds

Assume that in addition to specifying a circulation and demands on a network G, we also specify for each edge a lower bound on how much flow should be on each edge. Namely, for every edge $e \in E(\mathsf{G})$, we specify $\ell(e) \leq c(e)$, which is a lower bound to how much flow must be on this edge. As before we assume all numbers are integers.

We need now to compute a flow f that fill all the demands on the vertices, and that for any edge e, we have $\ell(e) \leq f(e) \leq c(e)$. The question is how to compute such a flow?

Let use start from the most naive flow, which transfer on every edge, exactly its lower bound. This is a valid flow as far as capacities and lower bounds, but of course, it might violate the demands. Formally, let $f_0(e) = \ell(e)$, for all $e \in E(\mathsf{G})$. Note that f_0 does not even satisfy the conservation rule:

$$L_v = f_0^{in}(v) - f_0^{out}(v) = \sum_{e \text{ into } v} \ell(e) - \sum_{e \text{ out of } v} \ell(e).$$

If $L_v = d_v$, then we are happy, since this flow satisfies the required demand. Otherwise, there is imbalance at v, and we need to fix it.

Formally, we set a new demand $d'_v = d_v - L_v$ for every node v, and the capacity of every edge e to be $c'(e) = c(e) - \ell(e)$. Let G' denote the new network with those capacities and demands (note, that the lower bounds had "disappeared"). If we can find a circulation f' on G' that satisfies the new demands, then clearly, the flow $f = f_0 + f'$, is a legal circulation, it satisfies the demands and the lower bounds.

But finding such a circulation, is something we already know how to do, using the algorithm of Theorem 14.2.4. Thus, it follows that we can compute a circulation with lower bounds.

Lemma 14.3.1. *There is a feasible circulation in G if and only if there is a feasible circulation in G'.*

If all demands, capacities, and lower bounds in G are integers, and there is a feasible circulation, then there is a feasible circulation that is integer valued.

Proof: Let f' be a circulation in G'. Let $f(e) = f_0(e) + f'(e)$. Clearly, f satisfies the capacity condition in G, and the lower bounds. Furthermore,

$$f^{in}(v) - f^{out}(v) = \sum_{e \text{ into } v} (\ell(e) + f'(e)) - \sum_{e \text{ out of } v} (\ell(e) + f'(e)) = L_v + (d_v - L_v) = d_v.$$

As such f satisfies the demand conditions on G.

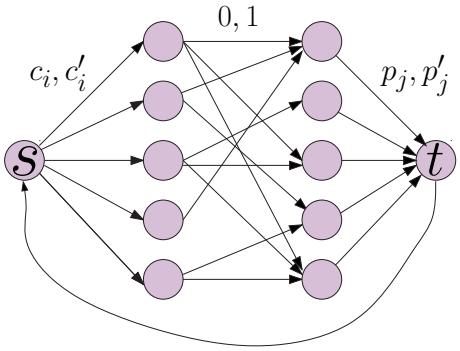
Similarly, let f be a valid circulation in G. Then it is easy to check that $f'(e) = f(e) - \ell(e)$ is a valid circulation for G'. ∎

14.4 Applications

14.4.1 Survey design

We would like to design a survey of products used by consumers (i.e., "Consumer i: what did you think of product j?"). The ith consumer agreed in advance to answer a certain number of questions in the range $[c_i, c'_i]$. Similarly, for each product j we would like to have at least p_j opinions about it, but not more than p'_j. Each consumer can be asked about a subset of the products which they consumed. In particular, we assume that we know in advance all the products each consumer used, and the above constraints. The question is how to assign questions to consumers, so that we get all the information we want to get, and every consumer is being asked a valid number of questions.

The idea of our solution is to reduce the design of the survey to the problem of computing a circulation in graph. First, we build a bipartite graph having consumers on one side, and products on the other side. Next, we insert the edge between consumer i and product j if the product was used by this consumer. The capacity of this edge is going to be 1. Intuitively, we are going to compute a flow in this network which is going to be an integer number. As such, every edge would be assigned either 0 or 1, where 1 is interpreted as asking the consumer about this product.



The next step, is to connect a source to all the consumers, where the edge $(s \to i)$ has lower bound c_i and upper bound c'_i. Similarly, we connect all the products to the destination t, where $(j \to t)$ has lower bound p_j and upper bound p'_j. We would like to compute a flow from s to t in this network that comply with the constraints. However, we only know how to compute a circulation on such a network. To overcome this, we create an edge with infinite capacity between t and s. Now, we are only looking for a valid circulation in the resulting graph G which complies with the aforementioned constraints. See figure on the right for an example of G.

Given a circulation f in G it is straightforward to interpret it as a survey design (i.e., all middle edges with flow 1 are questions to be asked in the survey). Similarly, one can verify that given a valid survey, it can be interpreted as a valid circulation in G. Thus, computing circulation in G indeed solves our problem.

We summarize:

Lemma 14.4.1. *Given n consumers and u products with their constraints $c_1, c'_1, c_2, c'_2, \ldots, c_n, c'_n$, $p_1, p'_1, \ldots, p_u, p'_u$ and a list of length m of which products where used by which consumers. An algorithm can compute a valid survey under these constraints, if such a survey exists, in time $O((n+u)m^2)$.*

Chapter 15

Network Flow IV - Applications II

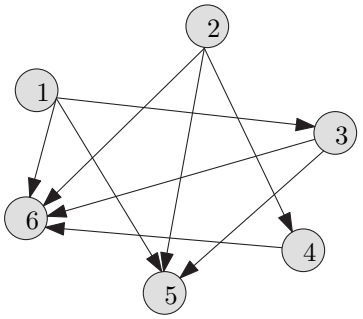
15.1 Airline Scheduling

Problem 15.1.1. Given information about flights that an airline needs to provide, generate a profitable schedule.

The input is a detailed information about "legs" of flight that the airline need to serve. We denote this set of flights by $\mathcal{F}$. We would like to find the minimum number of airplanes needed to carry out this schedule. For an example of possible input, see Figure 15.1 (i).

1: Boston (depart 6 A.M.) - Washington DC (arrive 7 A.M,).
2: Urbana (depart 7 A.M.) - Champaign (arrive 8 A.M.)
3: Washington (depart 8 A.M.) - Los Angeles (arrive 11 A.M.)
4: Urbana (depart 11 A.M.) - San Francisco (arrive 2 P.M.)
5: San Francisco (depart 2:15 P.M.) - Seattle (arrive 3:15 P.M.)
6: Las Vegas (depart 5 P.M.) - Seattle (arrive 6 P.M.).

(i)



(ii)

Figure 15.1: (i) a set $\mathcal{F}$ of flights that have to be served, and (ii) the corresponding graph G representing these flights.

We can use the same airplane for two segments i and j if the destination of i is the origin of the segment j and there is enough time in between the two flights for required maintenance. Alternatively, the airplane can fly from dest(i) to origin(j) (assuming that the time constraints are satisfied).

Example 15.1.2. As a concrete example, consider the flights:

1. Boston (depart 6 A.M.) - Washington D.C. (arrive 7 A.M,).
2. Washington (depart 8 A.M.) - Los Angeles (arrive 11 A.M.)

3. Las Vegas (depart 5 P.M.) - Seattle (arrive 6 P.M.)

This schedule can be served by a single airplane by adding the leg "Los Angeles (depart 12 noon)- Las Vegas (1 P,M.)" to this schedule.

15.1.1 Modeling the problem

The idea is to model the feasibility constraints by a graph. Specifically, G is going to be a directed graph over the flight legs. For i and j, two given flight legs, the edge $(i \to j)$ will be present in the graph G if the same airplane can serve both i and j; namely, the same airplane can perform leg i and afterwards serves the leg j.

Thus, the graph G is acyclic. Indeed, since we can have an edge $(i \to j)$ only if the flight j comes after the flight i (in time), it follows that we can not have cycles.

We need to decide if all the required legs can be served using only k airplanes?

15.1.2 Solution

The idea is to perform a reduction of this problem to the computation of circulation. Specifically, we construct a graph H, as follows:

• For every leg i, we introduce two vertices $u_i, v_i \in V(H)$. We also add a source vertex s and a sink vertex t to H. We set the demand at t to be k, and the demand at s to be $-k$ (i.e., k units of flow are leaving s and need to arrive to t).

• Each flight on the list must be served. This is forced by introducing an edge $e_i = (u_i \to v_i)$, for each leg i. We also set the lower bound on e_i to be 1, and the capacity on e_i to be 1 (i.e., $\ell(e_i) = 1$ and $c(e_i) = 1$).

• If the same plane can perform flight i and j (i.e., $(i \to j) \in \mathsf{E}(\mathsf{G})$) then add an edge $(v_i \to u_j)$ with capacity 1 to H (with no lower bound constraint).

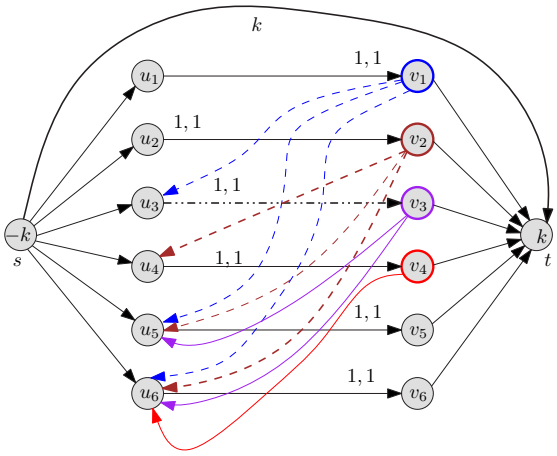


Figure 15.2: The resulting graph H for the instance of airline scheduling from Figure 15.1.

• Since any airplane can start the day with flight i, we add an edge $(s \to u_i)$ with capacity 1 to H, for all flights i.

• Similarly, any airplane can end the day by serving the flight j. Thus, we add edge $(v_j \to t)$ with capacity 1 to G, for all flights j.

• If we have extra planes, we do not have to use them. As such, we introduce a "overflow" edge $(s \to t)$ with capacity k, that can carry over all the unneeded airplanes from s directly to t.

Let H denote the resulting graph. See Figure 15.2 for an example.

Lemma 15.1.3. *There is a way to perform all flights of $\mathcal{F}$ using at most k planes if and only if there is a feasible circulation in the network H.*

Proof: Assume there is a way to perform the flights using $k' \leq k$ flights. Consider such a feasible schedule. The schedule of an airplane in this schedule defines a path π in the network H that starts at s and ends at t, and we send one unit of flow on each such path. We also send $k - k'$ units of flow on the edge $(s \rightarrow t)$. Note, that since the schedule is feasible, all legs are being served by some airplane. As such, all the "middle" edges with lower-bound 1 are being satisfied. Thus, this results is a valid circulation in H that satisfies all the given constraints.

As for the other direction, consider a feasible circulation in H. This is an integer valued circulation by the Integrality theorem. Suppose that k' units of flow are sent between s and t (ignoring the flow on the edge $(s \rightarrow t)$). All the edges of H (except $(s \rightarrow t)$) have capacity 1, and as such the circulation on all other edges is either zero or one (by the Integrality theorem). We convert this into k' paths by repeatedly traversing from the vertex s to the destination t, removing the edges we are using in each such path after extracting it (as we did for the k disjoint paths problem). Since we never use an edge twice, and H is acyclic, it follows that we would extract k' paths. Each of those paths correspond to one airplane, and the overall schedule for the airplanes is valid, since all required legs are being served (by the lower-bound constraint). ■

Extensions and limitations. There are a lot of other considerations that we ignored in the above problem: (i) airplanes have to undergo long term maintenance treatments every once in awhile, (ii) one needs to allocate crew to these flights, (iii) schedule differ between days, and (iv) ultimately we interested in maximizing revenue (a much more fluffy concept and much harder to explicitly describe).

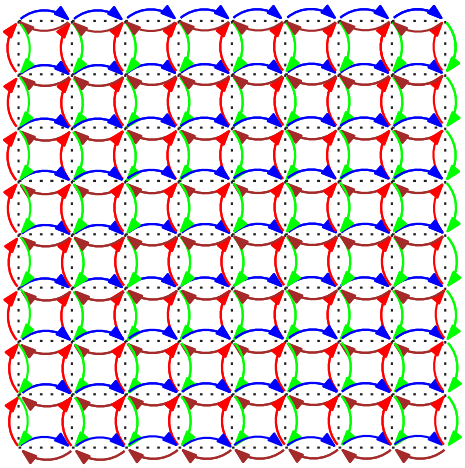
In particular, while network flow is used in practice, real world problems are complicated, and network flow can capture only a few aspects. More than undermining the usefulness of network flow, this emphasize the complexity of real-world problems.

15.2 Image Segmentation

In the ***image segmentation problem***, the input is an image, and we would like to partition it into background and foreground. For an example, see Figure 15.3.

(i) (ii)

Figure 15.3: The (i) input image, and (ii) a possible segmentation of the image.



The input is a bitmap on a grid where every grid node represents a pixel. We covert this grid into a directed graph G, by interpreting every edge of the grid as two directed edges. See the figure on the right to see how the resulting graph looks like.

Specifically, the input for out problem is as follows:

- A bitmap of size $N \times N$, with an associated directed graph $\mathsf{G} = (V, E)$.
- For every pixel i, we have a value $f_i \geq 0$, which is an estimate of the likelihood of this pixel to be in foreground (i.e., the larger f_i is the more probable that it is in the foreground)
- For every pixel i, we have (similarly) an estimate b_i of the likelihood of pixel i to be in background.
- For every two adjacent pixels i and j we have a separation penalty p_{ij}, which is the "price" of separating i from j. This quantity is defined only for adjacent pixels in the bitmap. (For the sake of simplicity of exposition we assume that $p_{ij} = p_{ji}$. Note, however, that this assumption is not necessary for our discussion.)

Problem 15.2.1. Given input as above, partition V (the set of pixels) into two disjoint subsets F and B, such that

$$q(F, B) = \sum_{i \in F} f_i + \sum_{i \in B} b_i - \sum_{(i,j) \in E, |F \cap \{i,j\}|=1} p_{ij}.$$

is maximized.

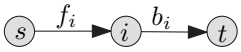
We can rewrite $q(F, B)$ as:

$$\begin{aligned} q(F,B) &= \sum_{i \in F} f_i + \sum_{j \in B} b_j - \sum_{(i,j) \in E, |F \cap \{i,j\}|=1} p_{ij} \\ &= \sum_{i \in v} (f_i + b_i) - \left(\sum_{i \in B} f_i + \sum_{j \in F} b_j + \sum_{(i,j) \in E, |F \cap \{i,j\}|=1} p_{ij} \right). \end{aligned}$$

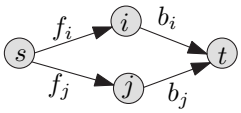
Since the term $\sum_{i \in v}(f_i + b_i)$ is a constant, maximizing $q(F, B)$ is equivalent to minimizing $u(F, B)$, where

$$u(F,B) = \sum_{i \in B} f_i + \sum_{j \in F} b_j + \sum_{(i,j) \in E, |F \cap \{i,j\}|=1} p_{ij}. \tag{15.1}$$

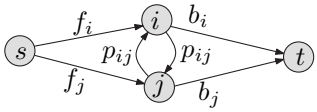
How do we compute this partition. Well, the basic idea is to compute a minimum cut in a graph such that its price would correspond to $u(F, B)$. Before dwelling into the exact details, it is useful to play around with some toy examples to get some intuition. Note, that we are using the max-flow algorithm as an algorithm for computing minimum directed cut.



To begin with, consider a graph having a source s, a vertex i, and a sink t. We set the price of $(s \to i)$ to be f_i and the price of the edge $(i \to t)$ to be b_i. Clearly, there are two possible cuts in the graph, either $(\{s, i\}, \{t\})$ (with a price b_i) or $(\{s\}, \{i, t\})$ (with a price f_i). In particular, every path of length 2 in the graph between s and t forces the algorithm computing the minimum-cut (via network flow) to choose one of the edges, to the cut, where the algorithm "prefers" the edge with lower price.



Next, consider a bitmap with two vertices i an j that are adjacent. Clearly, minimizing the first two terms in Eq. (15.1) is easy, by generating length two parallel paths between s and t through i and j. See figure on the right. Clearly, the price of a cut in this graph is exactly the price of the partition of $\{i, j\}$ into background and foreground sets. However, this ignores the separation penalty p_{ij}.



To this end, we introduce two new edges $(i \to j)$ and $(j \to i)$ into the graph and set their price to be p_{ij}. Clearly, a price of a cut in the graph can be interpreted as the value of $u(F, B)$ of the corresponding sets F and B, since all the edges in the segmentation from nodes of F to nodes of B are contributing their separation price to the cut price. Thus, if we extend this idea to the directed graph G, the minimum-cut in the resulting graph would corresponds to the required segmentation.

Let us recap: Given the directed grid graph $\mathsf{G} = (V, E)$ we add two special source and sink vertices, denoted by s and t respectively. Next, for all the pixels $i \in V$, we add an edge $e_i = (s \to i)$ to the graph, setting its capacity to be $c(e_i) = f_i$. Similarly, we add the edge $e'_i = (j \to t)$ with capacity $c(e'_i) = b_i$. Similarly, for every pair of vertices $i.j$ in that grid that are adjacent, we assign the cost p_{ij} to the edges $(i \to j)$ and $(j \to i)$. Let H denote the resulting graph.

The following lemma, follows by the above discussion.

Lemma 15.2.2. *A minimum cut (F, B) in H minimizes $u(F, B)$.*

Using the minimum-cut max-flow theorem, we have:

Theorem 15.2.3. *One can solve the segmentation problem, in polynomial time, by computing the max flow in the graph H.*

15.3 Project Selection

You have a small company which can carry out some projects out of a set of projects P. Associated with each project $i \in P$ is a revenue p_i, where $p_i > 0$ is a profitable project and $p_i < 0$ is a losing project. To make things interesting, there is dependency between projects. Namely, one has to complete some "infrastructure" projects before one is able to do other projects. Namely, you are provided with a graph $\mathsf{G} = (P, E)$ such that $(i \to j) \in E$ if and only if j is a prerequisite for i.

Definition 15.3.1. A set $X \subset P$ is ***feasible*** if for all $i \in X$, all the prerequisites of i are also in X. Formally, for all $i \in X$, with an edge $(i \to j) \in E$, we have $j \in X$.

The ***profit*** associated with a set of projects $X \subseteq P$ is $\text{profit}(X) = \sum_{i \in X} p_i$.

Problem 15.3.2 (Project Selection Problem). Select a feasible set of projects maximizing the overall profit.

The idea of the solution is to reduce the problem to a minimum-cut in a graph, in a similar fashion to what we did in the image segmentation problem.

15.3.1 The reduction

The reduction works by adding two vertices s and t to the graph G, we also perform the following modifications:

- For all projects $i \in P$ with positive revenue (i.e., $p_i > 0$) add the $e_i = (s \to i)$ to G and set the capacity of the edge to be $c(e_i) = p_i$, where s is the added source vertex.
- Similarly, for all projects $j \in P$, with negative revenue (i.e., $p_j < 0$) add the edge $e'_j = (j \to t)$ to G and set the edge capacity to $c(e'_j) = -p_j$, where t is the added sink vertex.
- Compute a bound on the max flow (and thus also profit) in this network: $C = \sum_{i \in P, p_i > 0} p_i$.
- Set capacity of all other edges in G to $4C$ (these are the dependency edges in the project, and intuitively they are too expensive to be "broken" by a cut).

Let H denote the resulting network.

Let $X \subseteq P$ Be a set of feasible projects, and let $X' = X \cup \{s\}$ and $Y' = (P \setminus X) \cup \{t\}$. Consider the s-t cut (X', Y') in H. Note, that no edge of $\mathsf{E}(\mathsf{G})$ is in (X', Y') since X is a feasible set (i.e., there is no $u \in X'$ and $v \in Y'$ such that $(u \to v) \in \mathsf{E}(\mathsf{G})$).

Lemma 15.3.3. *The capacity of the cut (X', Y'), as defined by a feasible project set X, is $c(X', Y') = C - \sum_{i \in X} p_i = C - \text{profit}(X)$.*

Proof: The edges of H are either:

(i) original edges of G (conceptually, they have price $+\infty$),
(ii) edges emanating from s, and
(iii) edges entering t.

Since X is feasible, it follows that no edges of type (i) contribute to the cut. The edges entering t contribute to the cut the value

$$\beta = \sum_{i \in X \text{ and } p_i < 0} -p_i.$$

The edges leaving the source s contribute

$$\gamma = \sum_{i \notin X \text{ and } p_i > 0} p_i = \sum_{i \in P, p_i > 0} p_i - \sum_{i \in X \text{ and } p_i > 0} p_i = C - \sum_{i \in X \text{ and } p_i > 0} p_i,$$

by the definition of C.

The capacity of the cut (X', Y') is

$$\beta + \gamma = \sum_{i \in X \text{ and } p_i < 0} (-p_i) + \left(C - \sum_{i \in X \text{ and } p_i > 0} p_i \right) = C - \sum_{i \in X} p_i = C - \text{profit}(X),$$

as claimed. ■

Lemma 15.3.4. *If (X', Y') is a cut with capacity at most C in G, then the set $X = X' \setminus \{s\}$ is a feasible set of projects.*

Namely, cuts (X', Y') of capacity $\leq C$ in H corresponds one-to-one to feasible sets which are profitable.

Proof: Since $c(X', Y') \leq C$ it must not cut any of the edges of G, since the price of such an edge is $4C$. As such, X must be a feasible set. ■

Putting everything together, we are looking for a feasible set X that maximizes $\text{profit}(X) = \sum_{i \in X} p_i$. This corresponds to a set $X' = X \cup \{s\}$ of vertices in H that minimizes $C - \sum_{i \in X} p_i$, which is also the cut capacity (X', Y'). Thus, computing a minimum-cut in H corresponds to computing the most profitable feasible set of projects.

Theorem 15.3.5. *If (X', Y') is a minimum cut in H then $X = X' \setminus \{s\}$ is an optimum solution to the project selection problem. In particular, using network flow the optimal solution can be computed in polynomial time.*

Proof: Indeed, we use network flow to compute the minimum cut in the resulting graph H. Note, that it is quite possible that the most profitable project is still a net loss. ■

15.4 Baseball elimination

There is a baseball league taking place and it is nearing the end of the season. One would like to know which teams are still candidates to winning the season.

Example 15.4.1. There 4 teams that have the following number of wins:

NEW YORK: 92, BALTIMORE: 91, TORONTO: 91, BOSTON: 90,

and there are 5 games remaining (all pairs except New York and Boston).

We would like to decide if Boston can still win the season? Namely, can Boston finish the season with as many point as anybody else? (We are assuming here that at every game the winning team gets one point and the losing team gets nada.[①])

First analysis. Observe, that Boston can get at most 92 wins. In particular, if New York wins any game then it is over since New-York would have 93 points.

Thus, to Boston to have any hope it must be that both Baltimore wins against New York and Toronto wins against New York. At this point in time, both teams have 92 points. But now, they play against each other, and one of them would get 93 wins. So Boston is eliminated!

Second analysis. As before, Boston can get at most 92 wins. All three other teams gets $X = 92 + 91 + 91 + (5 - 2)$ points together by the end of the league. As such, one of these three teams will get $\geq \lceil X/3 \rceil = 93$ points, and as such Boston is eliminated.

While the analysis of the above example is very cute, it is too tedious to be done each time we want to solve this problem. Not to mention that it is unclear how to extend these analyses to other cases.

15.4.1 Problem definition

Problem 15.4.2. The input is a set S of teams, where for every team $x \in S$, the team has w_x points accumulated so far. For every pair of teams $x, y \in S$ we know that there are g_{xy} games remaining between x and y. Given a specific team z, we would like to decide if z is eliminated?

Alternatively, is there away such that z would get as many wins as anybody else by the end of the season?

15.4.2 Solution

First, we can assume that z wins all its remaining games, and let m be the number of points z has in this case. Our purpose now is to build a network flow so we can verify that no other team *must* get more than m points.

[①] nada = nothing.

To this end, let s be the source (i.e., the source of wins). For every remaining game, a flow of one unit would go from s to one of the teams playing it. Every team can have at most $m - w_x$ flow from it to the target. If the max flow in this network has value

$$\alpha = \sum_{x,y \neq z, x<y} g_{xy}$$

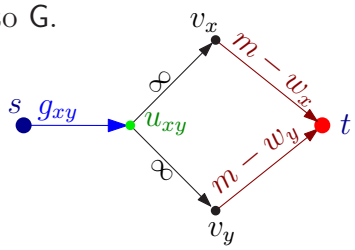
(which is the maximum flow possible) then there is a scenario such that all other teams gets at most m points and z can win the season. Negating this statement, we have that if the maximum flow is smaller than α then z is eliminated, since there must be a team that gets more than m points.

Construction. Let $S' = S \setminus \{z\}$ be the set of teams, and let

$$\alpha = \sum_{\{x,y\} \subseteq S'} g_{xy}. \tag{15.2}$$

We create a network flow G. For every team $x \in S'$ we add a vertex v_x to the network G. We also add the source and sink vertices, s and t, respectively, to G.

For every pair of teams $x, y \in S'$, such that $g_{xy} > 0$ we create a node u_{xy}, and add an edge $(s \to u_{xy})$ with capacity g_{xy} to G. We also add the edge $(u_{xy} \to v_x)$ and $(u_{xy} \to v_y)$ with infinite capacity to G. Finally, for each team x we add the edge $(v_x \to t)$ with capacity $m - w_x$ to G. How the relevant edges look like for a pair of teams x and y is depicted on the right.



Analysis. If there is a flow of value α in G then there is a way that all teams get at most m wins. Similarly, if there exists a scenario such that z ties or gets first place then we can translate this into a flow in G of value α. This implies the following result.

Theorem 15.4.3. *Team z has been eliminated if and only if the maximum flow in G has value strictly smaller than α. Thus, we can test in polynomial time if z has been eliminated.*

15.4.3 A compact proof of a team being eliminated

Interestingly, once z is eliminated, we can generate a compact proof of this fact.

Theorem 15.4.4. *Suppose that team z has been eliminated. Then there exists a "proof" of this fact of the following form:*

1. *The team z can finish with at most m wins.*
2. *There is a set of teams $\widehat{S} \subset S$ so that $\sum_{s \in \widehat{S}} w_x + \sum_{\{x,y\} \subseteq \widehat{S}} g_{xy} > m \left|\widehat{S}\right|$.*

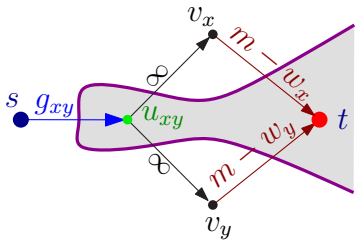
(And hence one of the teams in $\widehat{S}$ must end with strictly more than m wins.)

Proof: If z is eliminated then the max flow in G has value γ, which is smaller than α, see Eq. (15.2). By the minimum-cut max-flow theorem, there exists a minimum cut (S, T) of capacity γ in G, and let $\widehat{S} = \Big\{ x \;\Big|\; v_x \in S \Big\}$

Claim 15.4.5. *For any two teams x and y for which the vertex u_{xy} exists, we have that $u_{xy} \in S$ if and only if both x and y are in $\widehat{S}$.*

Proof: $\underline{\Big(x \notin \widehat{S} \text{ or } y \notin \widehat{S}\Big) \implies u_{xy} \notin S}$: If x is not in $\widehat{S}$ then v_x is in T. But then, if u_{xy} is in S the edge $(u_{xy} \to v_x)$ is in the cut. However, this edge has infinite capacity, which implies this cut is not a minimum cut (in particular, (S, T) is a cut with capacity smaller than α). As such, in such a case u_{xy} must be in T. This implies that if either x or y are *not* in $\widehat{S}$ then it must be that $u_{xy} \in T$. (And as such $u_{xy} \notin S$.)

$\underline{x \in \widehat{S} \text{ and } y \in \widehat{S} \implies u_{xy} \in S}$: Assume that both x and y are in $\widehat{S}$, then v_x and v_y are in S. We need to prove that $u_{xy} \in S$. If $u_{xy} \in T$ then consider the new cut formed by moving u_{xy} to S. For the new cut (S', T') we have



$$c(S', T') = c(S, T) - c\Big((s \to u_{xy})\Big).$$

Namely, the cut (S', T') has a lower capacity than the minimum cut (S, T), which is a contradiction. See figure on the right for this *impossible* cut. We conclude that $u_{xy} \in S$. ■

The above argumentation implies that edges of the type $(u_{xy} \to v_x)$ can not be in the cut (S, T). As such, there are two type of edges in the cut (S, T): (i) $(v_x \to t)$, for $x \in \widehat{S}$, and (ii) $(s \to u_{xy})$ where at least one of x or y is not in $\widehat{S}$. As such, the capacity of the cut (S, T) is

$$c(S, T) = \sum_{x \in \widehat{S}} (m - w_x) + \sum_{\{x,y\} \not\subset \widehat{S}} g_{xy} = m \left|\widehat{S}\right| - \sum_{x \in \widehat{S}} w_x + \left(\alpha - \sum_{\{x,y\} \subseteq \widehat{S}} g_{xy} \right).$$

However, $c(S, T) = \gamma < \alpha$, and it follows that

$$m \left|\widehat{S}\right| - \sum_{x \in \widehat{S}} w_x - \sum_{\{x,y\} \subseteq \widehat{S}} g_{xy} < \alpha - \alpha = 0.$$

Namely, $\sum_{x \in \widehat{S}} w_x + \sum_{\{x,y\} \subseteq \widehat{S}} g_{xy} > m \left|\widehat{S}\right|$, as claimed. ■

Part IV

Min Cost Flow

Chapter 16

Network Flow V - Min-cost flow

16.1 Minimum Average Cost Cycle

Let $\mathsf{G} = (V, E)$ be a ***digraph*** (i.e., a directed graph) with n vertices and m edges, and $\omega : E \to \mathbb{R}$ be a weight function on the edges. A ***directed cycle*** is closed walk $\mathsf{C} = (v_0, v_1, \ldots, v_t)$, where $v_t = v_0$ and $(v_i \to v_{i+1}) \in E$, for $i = 0, \ldots, t-1$. The ***average cost of a directed cycle*** is $\text{AvgCost}(\mathsf{C}) = \omega(C)/t = \left(\sum_{e \in C} \omega(e)\right)/t$.

For each $k = 0, 1, \ldots$, and $v \in V$, let $d_k(v)$ denote the minimum length of a walk with exactly k edges, ending at v (note, that the walk can start anywhere). So, for each v, we have

$$d_0(v) = 0 \quad \text{and} \quad d_{k+1}(v) = \min_{e=(u \to v) \in E} \Big(d_k(u) + \omega(e) \Big).$$

Thus, we can compute $d_i(v)$, for $i = 0, \ldots, n$ and $v \in V(G)$ in $O(nm)$ time using dynamic programming.

Let

$$\text{MinAvgCostCycle}(\mathsf{G}) = \min_{\mathsf{C} \text{ is a cycle in } \mathsf{G}} \text{AvgCost}(\mathsf{C})$$

denote the average cost of the ***minimum average cost cycle*** in G.

The following theorem is somewhat surprising.

Theorem 16.1.1. *The minimum average cost of a directed cycle in* G *is equal to*

$$\alpha = \min_{v \in V} \max_{k=0}^{n-1} \frac{d_n(v) - d_k(v)}{n - k}.$$

Namely, $\alpha = \text{MinAvgCostCycle}(\mathsf{G})$.

Proof: Note, that adding a quantity r to the weight of every edge of G increases the average cost of a cycle $\text{AvgCost}(\mathsf{C})$ by r. Similarly, α would also increase by r. In particular, we can

assume that the price of the minimum average cost cycle is zero. This implies that now all cycles have non-negative (average) cost.

Thus, from this point on we assume that MinAvgCostCycle(G) $= 0$, and we prove that $\alpha = 0$ in this case. This in turn would imply the theorem – indeed, given a graph where MinAvgCostCycle(G) $\neq 0$, then we will shift the costs the edges so that it is zero, use the proof below, and then shift it back.

MinAvgCostCycle(G) $= 0 \implies \alpha \geq 0$: We can rewrite α as $\alpha = \min_{u \in V} \beta(u)$, where

$$\beta(u) = \max_{k=0}^{n-1} \frac{d_n(u) - d_k(u)}{n-k}.$$

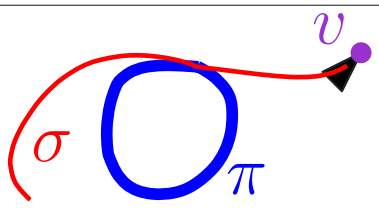


Figure 16.1: Decomposing P_n into a path σ and a cycle π.

Assume, that α is realized by a vertex v; that is $\alpha = \beta(v)$. Let P_n be a walk with n edges ending at v, of length $d_n(v)$. Since there are n vertices in G, it must be that P_n must contain a cycle. So, let us decompose P_n into a cycle π of length $n-k$ and a path σ of length k (k depends on the length of the cycle in P_n). We have that

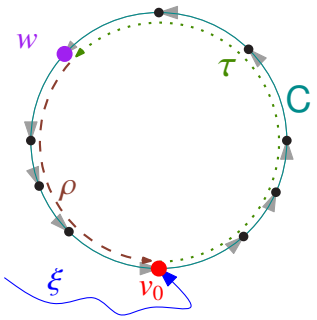
$$d_n(v) = \omega(P_n) = \omega(\pi) + \omega(\sigma) \geq \omega(\sigma) \geq d_k(v),$$

since $\omega(\pi) \geq 0$ as π is a cycle (and we assumed that all cycles have zero or positive cost). As such, we have $d_n(v) - d_k(v) \geq 0$. As such, $\frac{d_n(v)-d_k(v)}{n-k} \geq 0$. Let

$$\beta(v) = \max_{j=0}^{n-1} \frac{d_n(v) - d_j(v)}{n-j} \geq \frac{d_n(v) - d_k(v)}{n-k} \geq 0.$$

Now, $\alpha = \beta(v) \geq 0$, by the choice of v.

MinAvgCostCycle(G) $= 0 \implies \alpha \leq 0$: Let $\mathsf{C} = (v_0, v_1, \ldots, v_t)$ be the directed cycle of weight 0 in the graph. Observe, that $\min_{j=0}^{\infty} d_j(v_0)$ must be realized (for the first time) by an index $r < n$, since if it is longer, we can always shorten it by removing cycles and improve its price (since cycles have non-negative price). Let ξ denote this walk of length r ending at v_0. Let w be a vertex on C reached by walking $n-r$ edges on C starting from v_0, and let τ denote this walk (i.e., $|\tau| = n - r$). We have that



$$d_n(w) \leq \omega\Big(\xi \mid\mid \tau\Big) = d_r(v_0) + \omega(\tau)\,, \tag{16.1}$$

where $\xi \mid\mid \tau$ denotes the path formed by concatenating the path τ to ξ.

Similarly, let ρ be the walk formed by walking on C from w all the way back to v_0. Note that $\tau \mid\mid \rho$ goes around C several times, and as such, $\omega(\tau \mid\mid \rho) = 0$, as $\omega(\mathsf{C}) = 0$. Next, for

any k, since the shortest path with k edges arriving to w can be extended to a path that arrives to v_0, by concatenating ρ to it, we have that

$$d_k(w) + \omega(\rho) \geq d_{k+|\rho|}(v_0) \geq d_r(v_0) \geq d_n(w) - \omega(\tau)\,,$$

by Eq. (16.1). Rearranging, we have that $\omega(\rho) \geq d_n(w) - \omega(\tau) - d_k(w)$. Namely, we have

$$\forall k \qquad 0 = \omega(\tau \mid\mid \rho) = \omega(\rho) + \omega(\tau) \geq \Big(d_n(w) - \omega(\tau) - d_k(w)\Big) + \omega(\tau) = d_n(w) - d_k(w).$$

$$\implies \qquad \forall k \qquad \frac{d_n(w) - d_k(w)}{n-k} \leq 0$$

$$\implies \qquad \beta(w) = \max_{k=0}^{n-1} \frac{d_n(w) - d_k(w)}{n-k} \leq 0.$$

As such, $\alpha = \min_{v \in V(\mathsf{G})} \beta(v) \leq \beta(w) \leq 0$, and we conclude that $\alpha = 0$. ∎

Finding the minimum average cost cycle is now not too hard. We compute the vertex v that realizes α in Theorem 16.1.1. Next, we add $-\alpha$ to all the edges in the graph. We now know that we are looking for a cycle with price 0. We update the values $d_i(v)$ to agree with the new weights of the edges.

Now, v is the vertex realizing the quantity $0 = \alpha = \min_{u \in V} \max_{k=0}^{n-1} \frac{d_n(u) - d_k(u)}{n-k}$. Namely, we have that for the vertex v it holds

$$\max_{k=0}^{n-1} \frac{d_n(v) - d_k(v)}{n-k} = 0 \implies \forall k \in \{0, \ldots, n-1\} \quad \frac{d_n(v) - d_k(v)}{n-k} \leq 0$$
$$\implies \forall k \in \{0, \ldots, n-1\} \quad d_n(v) - d_k(v) \leq 0.$$

This implies that $d_n(v) \leq d_i(v)$, for all i. Now, we repeat the proof of Theorem 16.1.1. Let P_n be the path with n edges realizing $d_n(v)$. We decompose it into a path π of length k and a cycle τ. We know that $\omega(\tau) \geq 0$ (since all cycles have non-negative weights now). Now, $\omega(\pi) \geq d_k(v)$. As such, $\omega(\tau) = d_n(v) - \omega(\pi) \leq d_n(v) - d_k(v) \leq 0$, as π is a path of length k ending at v, and its cost is $\geq d_k(v)$. Namely, the cycle τ has $\omega(\tau) \leq 0$, and it the required cycle and computing it required $O(nm)$ time.

Note, that the above reweighting in fact was not necessary. All we have to do is to compute the node realizing α, extract P_n, and compute the cycle in P_n, and we are guaranteed by the above argumentation, that this is the cheapest average cycle.

Corollary 16.1.2. *Given a direct graph G with n vertices and m edges, and a weight function $\omega(\cdot)$ on the edges, one can compute the cycle with minimum average cost in $O(nm)$ time.*

16.2 Potentials

In general computing the shortest path in a graph that have negative weights is harder than just using the Dijkstra algorithm (that works only for graphs with non-negative weights on its edges). One can use Bellman-Ford algorithm[①] in this case, but it considerably slower (i.e.,

[①] `http://en.wikipedia.org/wiki/Bellman-Ford_algorithm`

it takes $O(mn)$ time). We next present a case where one can still use Dijkstra algorithm, with slight modifications.

The following is only required in the analysis of the minimum-cost flow algorithm we present later in this chapter. We describe it here in full detail since its simple and interesting.

For a directed graph $G = (V, E)$ with weight $w(\cdot)$ on the edges, let $\mathbf{d}_\omega(s, t)$ denote the length of the shortest path between s and t in G under the weight function w. Note, that w might assign negative weights to edges in G.

A ***potential*** $p(\cdot)$ is a function that assigns a real value to each vertex of G, such that if $e = (u \to v) \in G$ then $w(e) \geq p(v) - p(u)$.

Lemma 16.2.1. *(i) There exists a potential $p(\cdot)$ for G if and only if G has no negative cycles (with respect to $w(\cdot)$).*

(ii) Given a potential function $p(\cdot)$, for an edge $e = (u \to v) \in E(G)$, let $\ell(e) = w(e) - p(v) + p(u)$. Then $\ell(\cdot)$ is non-negative for the edges in the graph and for any pair of vertices $s, t \in V(G)$, we have that the shortest path π realizing $\mathbf{d}_\ell(s, t)$ also realizes $\mathbf{d}_\omega(s, t)$.

(iii) Given G and a potential function $p(\cdot)$, one can compute the shortest path from s to all the vertices of G in $O(n \log n + m)$ time, where G has n vertices and m edges

Proof: (i) Consider a cycle C, and assume there is a potential $p(\cdot)$ for G, and observe that

$$w(\mathsf{C}) = \sum_{(u \to v) \in E(\mathsf{C})} w(e) \geq \sum_{(u \to v) \in E(\mathsf{C})} (p(v) - p(u)) = 0,$$

as required.

For a vertex $v \in V(G)$, let $p(v)$ denote the shortest walk that ends at v in G. We claim that $p(v)$ is a potential. Since G does not have negative cycles, the quantity $p(v)$ is well defined. Observe that $p(v) \leq p(u) + w(u \to v)$ since we can always continue a walk to u into v by traversing $(u \to v)$. Thus, $p(v) - p(u) \leq w(u \to v)$, as required.

(ii) Since $\ell(e) = w(e) - p(v) + p(u)$ we have that $w(e) \geq p(v) - p(u)$ since $p(\cdot)$ is a potential function. As such $w(e) - p(v) + p(u) \geq 0$, as required.

As for the other claim, observe that for any path π in G starting at s and ending at t we have that

$$\ell(\pi) = \sum_{e=(u \to v) \in \pi} (w(e) - p(v) + p(u)) = w(\pi) + p(s) - p(t),$$

which implies that $\mathbf{d}_\ell(s, t) = \mathbf{d}_\omega(s, t) + p(s) - p(t)$. Implying the claim.

(iii) Just use the Dijkstra algorithm on the distances defined by $\ell(\cdot)$. The shortest paths are preserved under this distance by (ii), and this distance function is always positive. ■

16.3 Minimum cost flow

Given a network flow $\mathsf{G} = (V, E)$ with source s and sink t, capacities $\mathsf{c}(\cdot)$ on the edges, a real number ϕ, and a cost function $\kappa(\cdot)$ on the edges. The ***cost*** of a flow f is defined to be

$$\mathrm{cost}(\mathsf{f}) = \sum_{e \in E} \kappa(e) * \mathsf{f}(e).$$

The ***minimum-cost s-t flow problem*** ask to find the flow f that minimizes the cost and has value ϕ.

It would be easier to look on the problem of ***minimum-cost circulation problem***. Here, we are given instead of ϕ a lower-bound $\ell(\cdot)$ on the flow on every edge (and the regular upper bound $\mathsf{c}(\cdot)$ on the capacities of the edges). All the flow coming into a node must leave this node. It is easy to verify that if we can solve the minimum-cost circulation problem, then we can solve the min-cost flow problem. Thus, we will concentrate on the min-cost circulation problem.

An important technicality is that all the circulations we discuss here have zero demands on the vertices. As such, a circulation can be conceptually considered to be a flow going around in cycles in the graph without ever stopping. In particular, for these circulations, the conservation of flow property should hold for all the vertices in the graph.

The ***residual graph*** of f is the graph $\mathsf{G}_{\mathsf{f}} = (V, E_{\mathsf{f}})$ where

$$E_{\mathsf{f}} = \left\{ e = (u \to v) \in V \times V \;\middle|\; \mathsf{f}(e) < \mathsf{c}(e) \text{ or } \mathsf{f}\left(e^{-1}\right) > \ell\left(e^{-1}\right) \right\}.$$

where $e^{-1} = (v \to u)$ if $e = (u \to v)$. Note, that the definition of the residual network takes into account the lower-bound on the capacity of the edges.

Assumption 16.3.1. To simplify the exposition, we will assume that if $(u \to v) \in E(\mathsf{G})$ then $(v \to u) \notin E(G)$, for all $u, v \in V(G)$. This can be easily enforced by introducing a vertex in the middle of every edge of G. This is acceptable, since we are more concerned with solving the problem at hand in polynomial time, than the exact complexity. Note, that our discussion can be extended to handle the slightly more general case, with a bit of care.

We extend the cost function to be anti-symmetric; namely,

$$\forall\, (u \to v) \in E_{\mathsf{f}} \qquad \kappa\Big((u \to v)\Big) = -\kappa\Big((v \to u)\Big).$$

Consider a directed cycle C in G_{f}. For an edge $e = (u \to v) \in E$, we define

$$\chi_C(e) = \begin{cases} 1 & e \in C \\ -1 & e^{-1} = (v \to u) \in C \\ 0 & \text{otherwise;} \end{cases}$$

that is, we pay 1 if e is in C and -1 if we travel e in the "wrong" direction.

The ***cost*** of a directed cycle C in G_{f} is defined as

$$\kappa(\mathsf{C}) = \sum_{e \in \mathsf{C}} \kappa(e).$$

We will refer to a circulation that comply with the capacity and lower-bounds constraints as being ***valid***. A function that just comply with the conservation property (i.e., all incoming flow into a vertex leaves it), is a ***weak circulation***. In particular, a weak circulation might not comply with the capacity and lower bounds constraints of the given instance, and as such is not a valid circulation.

We need the following easy technical lemmas.

Lemma 16.3.2. *Let* f *and* g *be two valid circulations in* $\mathsf{G} = (V, E)$*. Consider the function* $\mathsf{h} = \mathsf{g} - \mathsf{f}$*. Then,* h *is a weak circulation, and if* $\mathsf{h}(u \to v) > 0$ *then the edge* $(u \to v) \in \mathsf{G}_\mathsf{f}$*.*

Proof: The fact that h is a circulation is trivial, as it is the difference between two circulations, and as such the same amount of flow that comes into a vertex leaves it, and thus it is a circulation. (Note, that h might not be a valid circulation, since it might not comply with the lower-bounds on the edges.)

Observe, that if $\mathsf{h}(u \to v)$ is negative, then $\mathsf{h}(v \to u) = -\mathsf{h}(u \to v)$ by the anti-symmetry of f and g, which implies the same property holds for h.

Consider an arbitrary edge $e = (u \to v)$ such that $\mathsf{h}(u \to v) > 0$.

There are two possibilities. First, if $e = (u \to v) \in E$, and $\mathsf{f}(e) < \mathsf{c}(e)$, then the claim trivially holds, since then $e \in \mathsf{G}_\mathsf{f}$. Thus, consider the case when $\mathsf{f}(e) = \mathsf{c}(e)$, but then $\mathsf{h}(e) = \mathsf{g}(e) - \mathsf{f}(e) \leq 0$. Which contradicts our assumption that $\mathsf{h}(u \to v) > 0$.

The second possibility, is that $e = (u \to v) \notin E$. But then $e^{-1} = (v \to u)$ must be in E, and it holds $0 > \mathsf{h}(e^{-1}) = \mathsf{g}(e^{-1}) - \mathsf{f}(e^{-1})$. Implying that $\mathsf{f}(e^{-1}) > \mathsf{g}(e^{-1}) \geq \ell(e^{-1})$. Namely, there is a flow by f in G going in the direction of e^{-1} which larger than the lower bound. Since we can return this flow in the other direction, it must be that $e \in \mathsf{G}_\mathsf{f}$. ∎

Lemma 16.3.3. *Let* f *be a circulation in a graph* G*. Then,* f *can be decomposed into at most* m *cycles,* $\mathsf{C}_1, \ldots, \mathsf{C}_m$*, such that, for any* $e \in E(G)$*, we have*

$$\mathsf{f}(e) = \sum_{i=1}^{t} \lambda_i \cdot \chi_{\mathsf{C}_i}(e),$$

where $\lambda_1, \ldots, \lambda_t > 0$ *and* $t \leq m$*, where* m *is the number of edges in* G*.*

Proof: Since f is a circulation, and the amount of flow into a node is equal to the amount of flow leaving the node, it follows that as long as f not zero, one can find a cycle in f. Indeed, start with a vertex which has non-zero amount of flow into it, and walk on an adjacent edge that has positive flow on it. Repeat this process, till you visit a vertex that was already visited. Now, extract the cycle contained in this walk.

Let C_1 be such a cycle, and observe that every edge of C_1 has positive flow on it, let λ_1 be the smallest amount of flow on any edge of C_1, and let e_1 denote this edge. Consider the new flow $\mathsf{g} = \mathsf{f} - \lambda_1 \cdot \chi_{\mathsf{C}_1}$. Clearly, g has zero flow on e_1, and it is a circulation. Thus, we can remove e_1 from G, and let H denote the new graph. By induction, applied to g on H, the flow g can be decomposed into $m - 1$ cycles with positive coefficients. Putting these cycles together with λ_1 and C_1 implies the claim. ∎

Theorem 16.3.4. *A flow* f *is a minimum cost feasible circulation if and only if each directed cycle of* G_f *has nonnegative cost.*

Proof: Let C be a negative cost cycle in G_f. Then, we can circulate more flow on C and get a flow with smaller price. In particular, let $\varepsilon > 0$ be a sufficiently small constant, such that

$\mathsf{g} = \mathsf{f} + \varepsilon * \chi_{\mathsf{C}}$ is still a feasible circulation (observe, that since the edges of C are G_{f}, all of them have residual capacity that can be used to this end). Now, we have that

$$\text{cost}(\mathsf{g}) = \text{cost}(\mathsf{f}) + \sum_{e \in \mathsf{C}} \kappa(e) * \varepsilon = \text{cost}(\mathsf{f}) + \varepsilon * \sum_{e \in \mathsf{C}} \kappa(e) = \text{cost}(\mathsf{f}) + \varepsilon * \kappa(\mathsf{C}) < \text{cost}(\mathsf{f}),$$

since $\kappa(\mathsf{C}) < 0$, which is a contradiction to the minimality of f.

As for the other direction, assume that all the cycles in G_{f} have non-negative cost. Then, let g be any feasible circulation. Consider the circulation $\mathsf{h} = \mathsf{g} - \mathsf{f}$. By Lemma 16.3.2, all the edges used by h are in G_{f}, and by Lemma 16.3.3 we can find $t \leq |E(\mathsf{G}_{\mathsf{f}})|$ cycles $\mathsf{C}_1, \ldots, \mathsf{C}_t$ in G_{f}, and coefficients $\lambda_1, \ldots, \lambda_t$, such that

$$\mathsf{h}(e) = \sum_{i=1}^{t} \lambda_i \chi_{\mathsf{C}_i}(e).$$

We have that

$$\text{cost}(\mathsf{g}) - \text{cost}(\mathsf{f}) = \text{cost}(\mathsf{h}) = \text{cost}\left(\sum_{i=1}^{t} \lambda_i \chi_{\mathsf{C}_i}\right) = \sum_{i=1}^{t} \lambda_i \text{cost}(\chi_{\mathsf{C}_i}) = \sum_{i=1}^{t} \lambda_i \kappa\left(\mathsf{C}_i\right) \geq 0,$$

as $\kappa(\mathsf{C}_i) \geq 0$, since there are no negative cycles in G_{f}. This implies that $\text{cost}(\mathsf{g}) \geq \text{cost}(\mathsf{f})$. Namely, f is a minimum-cost circulation. ■

16.4 A Strongly Polynomial Time Algorithm for Min-Cost Flow

The algorithm would start from a feasible circulation f. We know how to compute such a flow f using the standard max-flow algorithm. At each iteration, it would find the cycle C of minimum average cost cycle in G_{f} (using the algorithm of Section 16.1). If the cost of C is non-negative, we are done since we had arrived to the minimum cost circulation, by Theorem 16.3.4.

Otherwise, we circulate as much flow as possible along C (without violating the lower-bound constraints and capacity constraints), and reduce the price of the flow f. By Corollary 16.1.2, we can compute such a cycle in $O(mn)$ time. Since the cost of the flow is monotonically decreasing the algorithm would terminate if all the number involved are integers. But we will show in fact that his algorithm performs a polynomial number of iterations in n and m.

It is striking how simple is this algorithm, and the fact that it works in polynomial time. The analysis is somewhat more painful.

16.5 Analysis of the Algorithm

To analyze the above algorithm, let f_i be the flow in the beginning of the ith iteration. Let C_i be the cycle used in the ith iteration. For a flow f, let C_f the minimum average-length cycle of G_f, and let $\mu(\mathsf{f}) = \kappa(\mathsf{C}_\mathsf{f})/|\mathsf{C}_\mathsf{f}|$ denote the average "cost" per edge of C_f.

The following lemma, states that we are making "progress" in each iteration of the algorithm.

$\mathsf{f}, \mathsf{g}, \mathsf{h}, \mathsf{i}$	Flows or circulations
G_f	The residual graph for f
$\mathsf{c}(e)$	The capacity of the flow on e
$\ell(e)$	The lower-bound (i.e., demand) on the flow on e
$\mathrm{cost}(\mathsf{f})$	The overall cost of the flow f
$\kappa(e)$	The cost of sending one unit of flow on e
$\psi(e)$	The reduced cost of e

Figure 16.2: Notation used.

Lemma 16.5.1. *Let f be a flow, and let g the flow resulting from applying the cycle $\mathsf{C} = \mathsf{C}_\mathsf{f}$ to it. Then, $\mu(\mathsf{g}) \geq \mu(\mathsf{f})$.*

Proof: Assume for the sake of contradiction, that $\mu(\mathsf{g}) < \mu(\mathsf{f})$. Namely, we have

$$\frac{\kappa(\mathsf{C}_\mathsf{g})}{|\mathsf{C}_\mathsf{g}|} < \frac{\kappa(\mathsf{C}_\mathsf{f})}{|\mathsf{C}_\mathsf{f}|}. \tag{16.2}$$

Now, the only difference between G_f and G_g are the edges of C_f. In particular, some edges of C_f might disappear from G_g, as they are being used in g to their full capacity. Also, all the edges in the opposite direction to C_f will be present in G_g.

Now, C_g must use at least one of the new edges in G_g, since otherwise this would contradict the minimality of C_f (i.e., we could use C_g in G_f and get a cheaper average cost cycle than C_f). Let U be the set of new edges of G_g that are being used by C_g and are not present in G_f. Let $U^{-1} = \left\{ e^{-1} \;\middle|\; e \in U \right\}$. Clearly, all the edges of U^{-1} appear in C_f.

Now, consider the cycle $\pi = \mathsf{C}_\mathsf{f} \cup \mathsf{C}_\mathsf{g}$. We have that the average of π is

$$\alpha = \frac{\kappa(\mathsf{C}_\mathsf{f}) + \kappa(\mathsf{C}_\mathsf{g})}{|\mathsf{C}_\mathsf{f}| + |\mathsf{C}_\mathsf{g}|} < \max\left(\frac{\kappa(\mathsf{C}_\mathsf{g})}{|\mathsf{C}_\mathsf{g}|}, \frac{\kappa(\mathsf{C}_\mathsf{f})}{|\mathsf{C}_\mathsf{f}|} \right) = \mu(\mathsf{f}),$$

by Eq. (16.2). We can write π is a union of k edge-disjoint cycles $\sigma_1, \ldots, \sigma_k$ and some 2-cycles. A 2-cycle is formed by a pair of edges e and e^{-1} where $e \in U$ and $e^{-1} \in U^{-1}$. Clearly, the cost of these 2-cycles is zero. Thus, since the cycles $\sigma_1, \ldots, \sigma_k$ have no edges in U, it follows that they are all contained in G_f. We have

$$\kappa(\mathsf{C}_\mathsf{f}) + \kappa(\mathsf{C}_\mathsf{g}) = \sum_i \kappa(\sigma_i) + 0.$$

Thus, there is some non-negative integer constant c, such that

$$\alpha = \frac{\kappa(\mathsf{C}_\mathsf{f}) + \kappa(\mathsf{C}_\mathsf{g})}{|\mathsf{C}_\mathsf{f}| + |\mathsf{C}_\mathsf{g}|} = \frac{\sum_i \kappa(\sigma_i)}{c + \sum_i |\sigma_i|} \geq \frac{\sum_i \kappa(\sigma_i)}{\sum_i |\sigma_i|},$$

since α is negative (since $\alpha < \mu(\mathsf{f}) < 0$ as otherwise the algorithm would had already terminated). Namely, $\mu(\mathsf{f}) > \left(\sum_i \kappa(\sigma_i)\right) / \left(\sum_i |\sigma_i|\right)$. Which implies that there is a cycle σ_r, such that $\mu(\mathsf{f}) > \kappa(\sigma_r)/|\sigma_r|$ and this cycle is contained in G_f. But this is a contradiction to the minimality of $\mu(\mathsf{f})$. ∎

16.5.1 Reduced cost induced by a circulation

Conceptually, consider the function $\mu(\mathsf{f})$ to be a potential function that increases as the algorithm progresses. To make further progress in our analysis, it would be convenient to consider a reweighting of the edges of G, in such a way that preserves the weights of cycles.

Given a circulation f, we are going to define a different cost function on the edges which is induced by f. To begin with, let $\beta(u \to v) = \kappa(u \to v) - \mu(\mathsf{f})$. Note, that under the cost function α, the cheapest cycle has price 0 in G (since the average cost of an edge in the cheapest average cycle has price zero). Namely, G has no negative cycles under β. Thus, for every vertex $v \in V(G)$, let $d(v)$ denote the length of the shortest walk that ends at v. The function $d(v)$ is a potential in G, by Lemma 16.2.1, and as such

$$d(v) - d(u) \leq \beta(u \to v) = \kappa(u \to v) - \mu(\mathsf{f}) \,. \tag{16.3}$$

Next, let the ***reduced cost*** of $(u \to v)$ (in relation to f) be

$$\psi(u \to v) = \kappa(u \to v) + d(u) - d(v).$$

In particular, Eq. (16.3) implies that

$$\forall\, (u \to v) \in E(\mathsf{G}_\mathsf{f}) \quad \psi(u \to v) = \kappa(u \to v) + d(u) - d(v) \geq \mu(\mathsf{f}) \,. \tag{16.4}$$

Namely, the reduced cost of any edge $(u \to v)$ is at least $\mu(\mathsf{f})$.

Note that $\psi(v \to u) = \kappa(v \to u) + d(v) - d(u) = -\kappa(u \to v) + d(v) - d(u) = -\psi(u \to v)$ (i.e., it is anti-symmetric). Also, for any cycle C in G, we have that $\kappa(\mathsf{C}) = \psi(C)$, since the contribution of the potential $d(\cdot)$ cancels out.

The idea is that now we think about the algorithm as running with the reduced cost instead of the regular costs. Since the costs of cycles under the original cost and the reduced costs are the same, negative cycles are negative in both costs. The advantage is that the reduced cost is more useful for our purposes.

16.5.2 Bounding the number of iterations

Lemma 16.5.2. *Let f be a flow used in the ith iteration of the algorithm, let g be the flow used in the $(i+m)$th iteration, where m is the number of edges in G. Furthermore, assume that the algorithm performed at least one more iteration on g. Then, $\mu(\mathsf{g}) \geq (1 - 1/n)\mu(\mathsf{f})$.*

Proof: Let $\mathsf{C}_0, \ldots, \mathsf{C}_{m-1}$ be the m cycles used in computing g from f. Let $\psi(\cdot)$ be the reduced cost function induced by f.

If a cycle has only negative ***reduced cost*** edges, then after it is applied to the flow, one of these edges disappear from the residual graph, and the reverse edge (with positive reduced cost) appears in the residual graph. As such, if all the edges of these cycles have negative reduced costs, then G_g has no negative reduced cost edge, and as such $\mu(\mathsf{g}) \geq 0$. But the algorithm stops as soon as the average cost cycle becomes positive. A contradiction to our assumption that the algorithm performs at least another iteration.

Let C_h be the first cycle in this sequence, such that it contains an edge e', such that its reduced cost is positive; that is $\psi(e') \geq 0$. Note, that C_h has most n edges. We have that

$$\kappa(\mathsf{C}_h) = \psi(\mathsf{C}_h) = \sum_{e \in \mathsf{C}_h} \psi(e) = \psi(e') + \sum_{e \in \mathsf{C}_h, e \neq e'} \psi(e) \geq 0 + (|\mathsf{C}_h| - 1)\, \mu(\mathsf{f}),$$

by Eq. (16.4). Namely, the average cost of C_h is

$$0 > \mu(\mathsf{f}_h) = \frac{\kappa(\mathsf{C}_h)}{|\mathsf{C}_h|} \geq \frac{|\mathsf{C}_h| - 1}{|\mathsf{C}_h|} \mu(\mathsf{f}) \geq \left(1 - \frac{1}{n}\right) \mu(\mathsf{f}).$$

The claim now easily follows from Lemma 16.5.1. ∎

To bound the running time of the algorithm, we will argue that after sufficient number of iterations edges start disappearing from the residual network and never show up again in the residual network. Since there are only $2m$ possible edges, this would imply the termination of the algorithm.

Observation 16.5.3. *We have that* $(1 - 1/n)^n \leq (\exp(-1/n))^n \leq 1/e$, *since* $1 - x \leq e^{-x}$, *for all* $x \geq 0$, *as can be easily verified.*

Lemma 16.5.4. *Let* f *be the circulation maintained by the algorithm at iteration* ρ. *Then there exists an edge* e *in the residual network* G_f *such that it never appears in the residual networks of circulations maintained by the algorithm, for iterations larger than* $\rho + t$, *where* $t = 2nm \lceil \ln n \rceil$.

Proof: Let g be the flow used by the algorithm at iteration $\rho + t$. We define the reduced cost over the edges of G, as induced by the flow g. Namely,

$$\psi(u \to v) = \kappa(u \to v) + d(u) - d(v),$$

where $d(u)$ is the length of the shortest walk ending at u where the weight of edge $(u \to w)$ is $\kappa(u \to w) - \mu(\mathsf{g})$.

flow	in iteration
f	ρ
g	$\rho + t$
h	$\rho + t + \tau$

Now, conceptually, we are running the algorithm using this reduced cost function over the edges, and consider the minimum average cost cycle at iteration ρ with cost $\alpha = \mu(\mathsf{f})$. There must be an edge $\mathsf{e} \in E(\mathsf{G}_\mathsf{f})$, such that $\psi(\mathsf{e}) \leq \alpha$. (Note, that α is a negative quantity, as otherwise the algorithm would have terminated at iteration ρ.)

We have that, at iteration $\rho + t$, it holds

$$\mu(\mathsf{g}) \geq \alpha * \left(1 - \frac{1}{n}\right)^t \geq \alpha * \exp(-2m \lceil \ln n \rceil) \geq \frac{\alpha}{2n}, \tag{16.5}$$

by Lemma 16.5.2 and Observation 16.5.3 and since $\alpha < 0$. On the other hand, by Eq. (16.4), we know that for all the edges f in $E(\mathsf{G}_\mathsf{g})$, it holds $\psi(\mathsf{f}) \geq \mu(\mathsf{g}) \geq \alpha/2n$. As such, e can not be an edge of G_g since $\psi(\mathsf{e}) \leq \alpha$. Namely, it must be that $\mathsf{g}(\mathsf{e}) = \mathsf{c}(\mathsf{e})$.

So, assume that at a later iteration, say $\rho + t + \tau$, the edge e reappeared in the residual graph. Let h be the flow at the $(\rho + t + \tau)$th iteration, and let G_h be the residual graph. It must be that $\mathsf{h}(\mathsf{e}) < \mathsf{c}(\mathsf{e}) = \mathsf{g}(\mathsf{e})$.

Now, consider the circulation $\mathsf{i} = \mathsf{g} - \mathsf{h}$. It has a positive flow on the edge e, since $\mathsf{i}(\mathsf{e}) = \mathsf{g}(\mathsf{e}) - \mathsf{h}(\mathsf{e}) > 0$. In particular, there is a directed cycle C of positive flow of i in G_i that includes e, as implied by Lemma 16.3.3. Note, that Lemma 16.3.2 implies that C is also a cycle of G_h.

Now, the edges of C^{-1} are present in G_g. To see that, observe that for every edge $\mathsf{g} \in \mathsf{C}$, we have that $0 < \mathsf{i}(\mathsf{g}) = \mathsf{g}(\mathsf{g}) - \mathsf{h}(\mathsf{g}) \leq \mathsf{g}(\mathsf{g}) - \ell(\mathsf{g})$. Namely, $\mathsf{g}(\mathsf{g}) > \ell(\mathsf{g})$ and as such $\mathsf{g}^{-1} \in E(\mathsf{G}_\mathsf{g})$. As such, by Eq. (16.4), we have $\psi(\mathsf{g}^{-1}) \geq \mu(\mathsf{g})$. This implies

$$\forall \mathsf{g} \in \mathsf{C} \qquad \psi(\mathsf{g}) = -\psi\Big(\mathsf{g}^{-1}\Big) \leq -\mu(\mathsf{g}) \leq -\frac{\alpha}{2n},$$

by Eq. (16.5). Since C is a cycle of G_h, we have

$$\kappa(\mathsf{C}) = \psi(C) = \psi(\mathsf{e}) + \psi(C \setminus \{\mathsf{e}\}) \leq \alpha + (|\mathsf{C}| - 1) \cdot \left(-\frac{\alpha}{2n}\right) < \frac{\alpha}{2}.$$

Namely, the average cost of the cycle C, which is present in G_h, is $\kappa(\mathsf{C})/|\mathsf{C}| < \alpha/(2n)$.

On the other hand, the minimum average cost cycle in G_h has average price $\mu(\mathsf{h}) \geq \mu(\mathsf{g}) \geq \frac{\alpha}{2n}$, by Lemma 16.5.1. A contradiction, since we found a cycle C in G_h which is cheaper. ■

We are now ready for the "kill" – since one edge disappears forever every $O(mn \log n)$ iterations, it follows that after $O(m^2 n \log n)$ iterations the algorithm terminates. Every iteration takes $O(mn)$ time, by Corollary 16.1.2. Putting everything together, we get the following.

Theorem 16.5.5. *Given a digraph G with n vertices and m edges, lower bound and upper bound on the flow of each edge, and a cost associated with each edge, then one can compute a valid circulation of minimum-cost in $O(m^3 n^2 \log n)$ time.*

16.6 Bibliographical Notes

The minimum average cost cycle algorithm, of Section 16.1, is due to Karp [Kar78].

The description here follows very roughly the description of [Sch04]. The first strongly polynomial time algorithm for minimum-cost circulation is due to Éva Tardos [Tar85]. The algorithm we show is an improved version due to Andrew Goldberg and Robert Tarjan [GT89]. Initial research on this problem can be traced back to the 1940s, so it took almost fifty years to find a satisfactory solution to this problem.

Part V

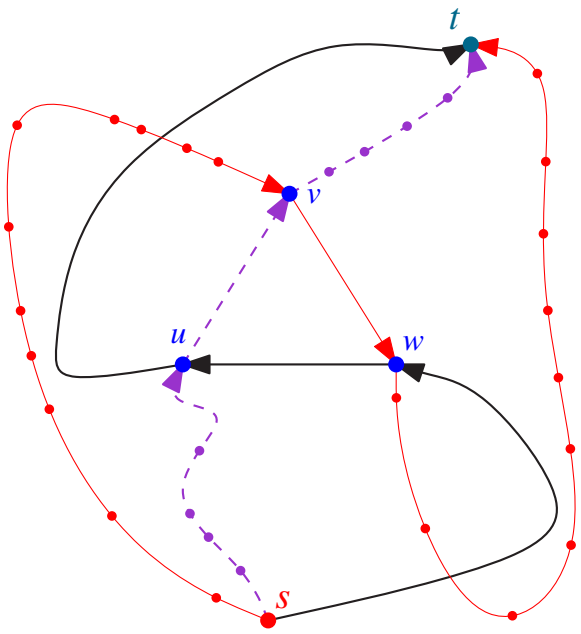
Min Cost Flow

Chapter 17

Network Flow VI - Min-Cost Flow Applications

17.1 Efficient Flow

A flow f would be considered to be ***efficient*** if it contains no cycles in it. Surprisingly, even the **Ford-Fulkerson** algorithm might generate flows with cycles in them. As a concrete example consider the picture on the right. A disc in the middle of edges indicate that we split the edge into multiple edges by introducing a vertex at this point. All edges have capacity one. For this graph, **Ford-Fulkerson** would first augment along $s \to w \to u \to t$. Next, it would augment along $s \to u \to v \to t$, and finally it would augment along $s \to v \to w \to t$. But now, there is a cycle in the flow; namely, $u \to v \to w \to u$.



One easy way to avoid such cycles is to first compute the max flow in G. Let α be the value of this flow. Next, we compute the min-cost flow in this network from s to t with flow α, where every edge has cost one. Clearly, the flow computed by the min-cost flow would not contain any such cycles. If it did contain cycles, then we can remove them by pushing flow against the cycle (i.e., reducing the flow along the cycle), resulting in a cheaper flow with the same value, which would be a contradiction. We got the following result.

Theorem 17.1.1. *Computing an efficient (i.e., acyclic) max-flow can be done in polynomial time.*

(BTW, this can also be achieved directly by removing cycles directly in the flow. Naturally, this flow might be less efficient than the min-cost flow computed.)

17.2 Efficient Flow with Lower Bounds

Consider the problem AFWLB (acyclic flow with lower-bounds) of computing efficient flow, where we have lower bounds on the edges. Here, we require that the returned flow would be integral, if all the numbers involved are integers. Surprisingly, this problem which looks like very similar to the problems we know how to solve efficiently is NP-Complete. Indeed, consider the following problem.

Hamiltonian Path

Instance: A directed graph G and two vertices s and t.
Question: Is there a Hamiltonian path (i.e., a path visiting every vertex exactly once) in G starting at s and ending at t?

It is easy to verify that Hamiltonian Path is NP-Complete[①]. We reduce this problem to AFWLB by replacing each vertex of G with two vertices and a direct edge in between them (except for the source vertex s and the sink vertex t). We set the lower-bound and capacity of each such edge to 1. Let H denote the resulting graph.

Consider now acyclic flow in H of capacity 1 from s to t which is integral. Its 0/1-flow, and as such it defines a path that visits all the special edges we created. In particular, it corresponds to a path in the original graph that starts at s, visits all the vertices of G and ends up at t. Namely, if we can compute an integral acyclic flow with lower-bounds in H in polynomial time, then we can solve Hamiltonian path in polynomial time. Thus, AFWLB is NP-Hard.

Theorem 17.2.1. *Computing an efficient (i.e., acyclic) max-flow with lower-bounds is* NP-Hard *(where the flow must be integral). The related decision problem (of whether such a flow exist) is* NP-Complete.

By this point you might be as confused as I am. We can model an acyclic max-flow problem with lower bounds as min-cost flow, and solve it, no? Well, not quite. The solution returned from the min-cost flow might have cycles and we can not remove them by cycling the cycles. That was only possible when there was no lower bounds on the edge capacities. Namely, the min-cost flow algorithm would return us a solution with cycles in it if there are lower bounds on the edges.

[①] Verify that you know to do this — its a natural question for the exam.

17.3 Shortest Edge-Disjoint Paths

Let G be a directed graph. We would like to compute k-edge disjoint paths between vertices s and t in the graph. We know how to do it using network flow. Interestingly, we can find the shortest k-edge disjoint paths using min-cost flow. Here, we assign cost 1 for every edge, and capacity 1 for every edge. Clearly, the min-cost flow in this graph with value k, corresponds to a set of k edge disjoint paths, such that their total length is minimized.

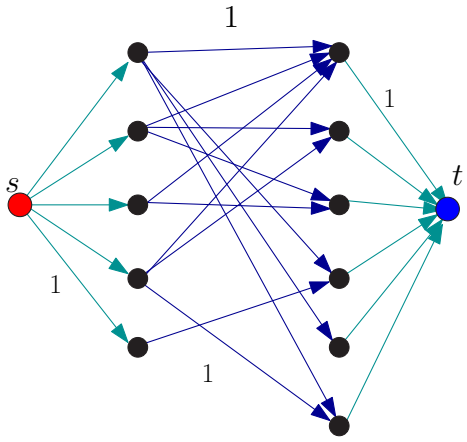
17.4 Covering by Cycles

Given a direct graph G, we would like to cover all its vertices by a set of cycles which are vertex disjoint. This can be done again using min-cost flow. Indeed, replace every vertex u in G by an edge $(u' \to u'')$. Where all the incoming edges to u are connected to u' and all the outgoing edges from u are now starting from u''. Let H denote the resulting graph. All the new edges in the graph have a lower bound and capacity 1, and all the other edges have no lower bound, but their capacity is 1. We compute the minimum cost circulation in H. Clearly, this corresponds to a collection of cycles in G covering all the vertices of minimum cost.

Theorem 17.4.1. *Given a directed graph G and costs on the edges, one can compute a cover of G by a collection of vertex disjoint cycles, such that the total cost of the cycles is minimized.*

17.5 Minimum weight bipartite matching

Given an undirected bipartite graph G, we would like to find the maximum cardinality matching in G that has minimum cost. The idea is to reduce this to network flow as we did in the unweighted case, and compute the maximum flow – the graph constructed is depicted on the right. Here, any edge has capacity 1. This gives us the size ϕ of the maximum matching in G. Next, we compute the min-cost flow in G with this value ϕ, where the edges connected to the source or the sing has cost zero, and the other edges are assigned their original cost in G. Clearly, the min-cost flow in this graph corresponds to a maximum cardinality min-cost flow in the original graph.



Here, we are using the fact that the flow computed is integral, and as such, it is a 0/1-flow.

Theorem 17.5.1. *Given a bipartite graph G and costs on the edges, one can compute the maximum cardinality minimum cost matching in polynomial time.*

17.6 The transportation problem

In the ***transportation problem***, we are given m facilities $f_1, \ldots, f_m$. The facility f_i contains x_i units of some commodity, for $i = 1, \ldots, m$. Similarly, there are $u_1, \ldots, u_n$ customers that would like to buy this commodity. In particular, u_i would like to by d_i units, for $i = 1, \ldots, n$. To make things interesting, it costs c_{ij} to send one unit of commodity from facility i to costumer j. The natural question is how to supply the demands while minimizing the total cost.

To this end, we create a bipartite graph with $f_1, \ldots, f_m$ on one side, and $u_1, \ldots, u_n$ on the other side. There is an edge from $(f_i \to u_j)$ with costs c_{ij}, for $i = 1, \ldots, m$ and $j = 1, \ldots, n$. Next, we create a source vertex that is connected to f_i with capacity x_i, for $i = 1, \ldots, m$. Similarly, we create an edges from u_j to the sink t, with capacity d_i, for $j = 1, \ldots n$. We compute the min-cost flow in this network that pushes $\phi = \sum_j d_k$ units from the source to the sink. Clearly, the solution encodes the required optimal solution to the transportation problem.

Theorem 17.6.1. *The transportation problem can be solved in polynomial time.*

Part VI

Fast Fourier Transform

Chapter 18

Fast Fourier Transform

> "But now, reflecting further, there begins to creep into his breast a touch of fellow-feeling for his imitators. For it seems to him now that there are but a handful of stories in the world; and if the young are to be forbidden to prey upon the old then they must sit for ever in silence."
>
> – – J.M. Coetzee.

18.1 Introduction

In this chapter, we will address the problem of multiplying two polynomials quickly.

Definition 18.1.1. A ***polynomial*** $p(x)$ of degree n is a function of the form $p(x) = \sum_{j=0}^{n} a_j x^j = a_0 + x(a_1 + x(a_2 + \ldots + xa_n))$.

Note, that given x_0, the polynomial can be evaluated at x_0 in $O(n)$ time.

There is a "dual" (and equivalent) representation of a polynomial. We sample its value in enough points, and store the values of the polynomial at those points. The following theorem states this formally. We omit the proof as you should have seen it already at some earlier math class.

Theorem 18.1.2. *For any set* $\Big\{(x_0, y_0), (x_1, y_1), \ldots, (x_{n-1}, y_{n-1})\Big\}$ *of* n **point-value pairs** *such that all the* x_k *values are distinct, there is a* unique *polynomial* $p(x)$ *of degree* $n-1$*, such that* $y_k = p(x_k)$*, for* $k = 0, \ldots, n-1$*.*

An explicit formula for $p(x)$ as a function of those point-value pairs is

$$p(x) = \sum_{i=0}^{n-1} y_i \frac{\prod_{j \neq i}(x - x_j)}{\prod_{j \neq i}(x_i - x_j)}.$$

Note, that the ith term in this summation is zero for $X = x_1, \ldots, x_{i-1}, x_{i+1}, \ldots, x_{n-1}$, and is equal to y_i for $x = x_i$.

It is easy to verify that given n point-value pairs, we can compute $p(x)$ in $O(n^2)$ time (using the above formula).

The point-value pairs representation has the advantage that we can multiply two polynomials quickly. Indeed, if we have two polynomials p and q of degree $n-1$, both represented by $2n$ (we are using more points than we need) point-value pairs

$$\begin{aligned}&\left\{(x_0, y_0), (x_1, y_1), \ldots, (x_{2n-1}, y_{2n-1})\right\} \quad \text{for} \quad p(x),\\ &\text{and} \quad \left\{(x_0, y'_0), (x_1, y'_1), \ldots, (x_{2n-1}, y'_{2n-1})\right\} \quad \text{for} \quad q(x).\end{aligned}$$

Let $r(x) = p(x)q(x)$ be the product of these two polynomials. Computing $r(x)$ directly requires $O(n^2)$ using the naive algorithm. However, in the point-value representation we have, that the representation of $r(x)$ is

$$\begin{aligned}\left\{(x_0, r(x_0)), \ldots, (x_{2n-1}, r(x_{2n-1}))\right\} &= \left\{(x_0, p(x_0)q(x_0)), \ldots, (x_{2n-1}, p(x_{2n-1})q(x_{2n-1}))\right\}\\ &= \left\{(x_0, y_0y'_0), \ldots, (x_{2n-1}, y_{2n-1}y'_{2n-1})\right\}.\end{aligned}$$

Namely, once we computed the representation of $p(x)$ and $q(x)$ using point-value pairs, we can multiply the two polynomials in linear time. Furthermore, we can compute the standard representation of $r(x)$ from this representation.

Thus, if could translate quickly (i.e., $O(n \log n)$ time) from the standard representation of a polynomial to point-value pairs representation, and back (to the regular representation) then we could compute the product of two polynomials in $O(n \log n)$ time. The ***Fast Fourier Transform*** is a method for doing exactly this. It is based on the idea of choosing the x_i values carefully and using divide and conquer.

18.2 Computing a polynomial quickly on n values

In the following, we are going to assume that the polynomial we work on has degree $n-1$, where $n = 2^k$. If this is not true, we can pad the polynomial with terms having zero coefficients.

Assume that we magically were able to find a set of numbers $\Psi = \{x_1, \ldots, x_n\}$, so that it has the following property: $|\mathsf{SQ}(\Psi)| = n/2$, where $\mathsf{SQ}(\Psi) = \left\{x^2 \;\middle|\; x \in \Psi\right\}$. Namely, when we square the numbers of Ψ, we remain with only $n/2$ distinct values, although we started with n values. It is quite easy to find such a set.

What is much harder is to find a set that have this property repeatedly. Namely, $\mathsf{SQ}(\mathsf{SQ}(\Psi))$ would have $n/4$ distinct values, $\mathsf{SQ}(\mathsf{SQ}(\mathsf{SQ}(\Psi)))$ would have $n/8$ values, and $\mathsf{SQ}^i(\Psi)$ would have $n/2^i$ distinct values.

Predictably, maybe, it is easy to show that there is no such set of real numbers (verify...). But let us for the time being ignore this technicality, and fly, for a moment, into the land of fantasy, and assume that we do have such a set of numbers, so that $|\mathsf{SQ}^i(\Psi)| = n/2^i$ numbers, for $i = 0, \ldots, k$. Let us call such a set of numbers ***collapsible***.

Given a set of numbers $\mathcal{X} = \{x_0, \ldots, x_n\}$ and a polynomial $p(x)$, let

$$p\big(\mathcal{X}\big) = \left\langle \big(x_0, p(x_0)\big), \ldots, \big(x_n, p(x_n)\big) \right\rangle.$$

FFTAlg$(p,\ X)$

input: $p(x)$: A polynomial of degree n: $p(x) = \sum_{i=0}^{n-1} a_i x^i$
X: A collapsible set of n elements.

output: $p(X)$

begin

$u(y) = \sum_{i=0}^{n/2-1} a_{2i} y^i$
$v(y) = \sum_{i=0}^{n/2-1} a_{1+2i} y^i$.
$Y = \mathsf{SQ}(X) = \left\{ x^2 \;\middle|\; x \in X \right\}$.
$U =$ **FFTAlg**(u, Y) /* $U = u(Y)$ */
$V =$ **FFTAlg**(v, Y) /* $V = v(Y)$ */

$Out \leftarrow \emptyset$
for $x \in A$ **do**
/* $p(x) = u(x^2) + x * v(x^2)$ */
/* $U[x^2]$ is the value $u(x^2)$ */
$(x, p(x)) \leftarrow (x, U[x^2] + x \cdot V[x^2])$
$Out \leftarrow Out \cup \{(x, p(x))\}$

return Out

end

Figure 18.1: The FFT algorithm.

Furthermore, let us rewrite $p(x) = \sum_{i=0}^{n-1} a_i x^i$ as $p(x) = u(x^2) + x \cdot v(x^2)$, where

$$u(y) = \sum_{i=0}^{n/2-1} a_{2i} y^i \quad \text{and} \quad v(y) = \sum_{i=0}^{n/2-1} a_{1+2i} y^i.$$

Namely, we put all the even degree terms of $p(x)$ into $u(\cdot)$, and all the odd degree terms into $v(\cdot)$. The maximum degree of the two polynomials $u(y)$ and $v(y)$ is $n/2$.

We are now ready for the kill: To compute $p(\Psi)$ for Ψ, which is a collapsible set, we have to compute $u(\mathsf{SQ}(\Psi)), v(\mathsf{SQ}(\Psi))$. Namely, once we have the value-point pairs of $u(\mathsf{SQ}(A)), v(\mathsf{SQ}(A))$ we can, in *linear* time, compute $p(\Psi)$. But, $\mathsf{SQ}(\Psi)$ have $n/2$ values because we assumed that Ψ is collapsible. Namely, to compute n point-value pairs of $p(\cdot)$, we have to compute $n/2$ point-value pairs of two polynomials of degree $n/2$ over a set of $n/2$ numbers.

Namely, we reduce a problem of size n into two problems of size $n/2$. The resulting algorithm is depicted in Figure 18.1.

What is the running time of **FFTAlg**? Well, clearly, all the operations except the recursive calls takes $O(n)$ time (assume, for the time being, that we can fetch $U[x^2]$ in $O(1)$ time). As for the recursion, we call recursively on a polynomial of degree $n/2$ with $n/2$ values (Ψ is collapsible!). Thus, the running time is $T(n) = 2T(n/2) + O(n)$, which is $O(n \log n)$ – exactly what we wanted.

18.2.1 Generating Collapsible Sets

Nice! But how do we resolve this "technicality" of not having collapsible set? It turns out that if we work over the complex numbers (instead of over the real numbers), then generating collapsible sets is quite easy. Describing complex numbers is outside the scope of this writeup, and we assume that you already have encountered them before. Nevertheless a quick reminder is provided in Section 18.4.1. Everything you can do over the real numbers you can do over the complex numbers, and much more (complex numbers are your friend).

In particular, let γ denote a nth root of unity. There are n such roots, and let $\gamma_j(n)$ denote the jth root, see Figure 18.2$_{\text{p161}}$. In particular, let

$$\gamma_j(n) = \cos((2\pi j)/n) + \mathbf{i}\sin((2\pi j)/n) = \gamma^j.$$

Let $\mathcal{A}(n) = \{\gamma_0(n), \ldots, \gamma_{n-1}(n)\}$. It is easy to verify that $|\mathsf{SQ}(\mathcal{A}(n))|$ has exactly $n/2$ elements. In fact, $\mathsf{SQ}(\mathcal{A}(n)) = \mathcal{A}(n/2)$, as can be easily verified. Namely, if we pick n to be a power of 2, then $\mathcal{A}(n)$ is the *required* collapsible set.

Theorem 18.2.1. *Given polynomial $p(x)$ of degree n, where n is a power of two, then we can compute $p(X)$ in $O(n \log n)$ time, where $X = \mathcal{A}(n)$ is the set of n different powers of the nth root of unity over the complex numbers.*

We can now multiply two polynomials quickly by transforming them to the point-value pairs representation over the nth root of unity, but we still have to transform this representation back to the regular representation.

18.3 Recovering the polynomial

This part of the writeup is somewhat more technical. Putting it shortly, we are going to apply the **FFTAlg** algorithm once again to recover the original polynomial. The details follow.

It turns out that we can interpret the FFT as a matrix multiplication operator. Indeed, if we have $p(x) = \sum_{i=0}^{n-1} a_i x^i$ then evaluating $p(\cdot)$ on $\mathcal{A}(n)$ is equivalent to:

$$\begin{pmatrix} y_0 \\ y_1 \\ y_2 \\ \vdots \\ y_{n-1} \end{pmatrix} = \underbrace{\begin{pmatrix} 1 & \gamma_0 & \gamma_0^2 & \gamma_0^3 & \cdots & \gamma_0^{n-1} \\ 1 & \gamma_1 & \gamma_1^2 & \gamma_1^3 & \cdots & \gamma_1^{n-1} \\ 1 & \gamma_2 & \gamma_2^2 & \gamma_2^3 & \cdots & \gamma_2^{n-1} \\ 1 & \gamma_3 & \gamma_3^2 & \gamma_3^3 & \cdots & \gamma_3^{n-1} \\ \vdots & \vdots & \vdots & \vdots & \cdots & \vdots \\ 1 & \gamma_{n-1} & \gamma_{n-1}^2 & \gamma_{n-1}^3 & \cdots & \gamma_{n-1}^{n-1} \end{pmatrix}}_{\text{the matrix } V} \begin{pmatrix} a_0 \\ a_1 \\ a_2 \\ a_3 \\ \vdots \\ a_{n-1} \end{pmatrix},$$

where $\gamma_j = \gamma_j(n) = (\gamma_1(n))^j$ is the jth power of the nth root of unity, and $y_j = p(\gamma_j)$.

This matrix V is very interesting, and is called the ***Vandermonde*** matrix. Let V^{-1} be the inverse matrix of this Vandermonde matrix. And let multiply the above formula from the left. We get:

$$\begin{pmatrix} a_0 \\ a_1 \\ a_2 \\ a_3 \\ \vdots \\ a_{n-1} \end{pmatrix} = V^{-1} \begin{pmatrix} y_0 \\ y_1 \\ y_2 \\ \vdots \\ y_{n-1} \end{pmatrix}.$$

Namely, we can recover the polynomial $p(x)$ from the point-value pairs

$$\Big\{(\gamma_0, p(\gamma_0)), (\gamma_1, p(\gamma_1)), \ldots, (\gamma_{n-1}, p(\gamma_{n-1}))\Big\}$$

by doing a single matrix multiplication of V^{-1} by the vector $[y_0, y_1, \ldots, y_{n-1}]$. However, multiplying a vector with n entries with a matrix of size $n \times n$ takes $O(n^2)$ time. Thus, we had not benefitted anything so far.

However, since the Vandermonde matrix is so well behaved[①], it is not too hard to figure out the inverse matrix.

Claim 18.3.1.

$$V^{-1} = \frac{1}{n} \begin{pmatrix} 1 & \beta_0 & \beta_0^2 & \beta_0^3 & \cdots & \beta_0^{n-1} \\ 1 & \beta_1 & \beta_1^2 & \beta_1^3 & \cdots & \beta_1^{n-1} \\ 1 & \beta_2 & \beta_2^2 & \beta_2^3 & \cdots & \beta_2^{n-1} \\ 1 & \beta_3 & \beta_3^2 & \beta_3^3 & \cdots & \beta_3^{n-1} \\ \vdots & \vdots & \vdots & \vdots & \cdots & \vdots \\ 1 & \beta_{n-1} & \beta_{n-1}^2 & \beta_{n-1}^3 & \cdots & \beta_{n-1}^{n-1} \end{pmatrix},$$

where $\beta_j = (\gamma_j(n))^{-1}$.

Proof: Consider the (u, v) entry in the matrix $C = V^{-1}V$. We have

$$C_{u,v} = \sum_{j=0}^{n-1} \frac{(\beta_u)^j (\gamma_j)^v}{n}.$$

We need to use the fact here that $\gamma_j = (\gamma_1)^j$ as can be easily verified. Thus,

$$C_{u,v} = \sum_{j=0}^{n-1} \frac{(\beta_u)^j ((\gamma_1)^j)^v}{n} = \sum_{j=0}^{n-1} \frac{(\beta_u)^j ((\gamma_1)^v)^j}{n} = \sum_{j=0}^{n-1} \frac{(\beta_u \gamma_v)^j}{n}.$$

[①]Not to mention famous, beautiful and well known – in short a celebrity matrix.

Clearly, if $u = v$ then

$$C_{u,u} = \frac{1}{n}\sum_{j=0}^{n-1}(\beta_u\gamma_u)^j = \frac{1}{n}\sum_{j=0}^{n-1}(1)^j = \frac{n}{n} = 1.$$

If $u \neq v$ then,

$$\beta_u\gamma_v = (\gamma_u)^{-1}\gamma_v = (\gamma_1)^{-u}\gamma_1^v = (\gamma_1)^{v-u} = \gamma_{v-u}.$$

And

$$C_{u,v} = \frac{1}{n}\sum_{j=0}^{n-1}(\gamma_{v-u})^j = \frac{1}{n}\cdot\frac{\gamma_{v-u}^n - 1}{\gamma_{v-u} - 1} = \frac{1}{n}\cdot\frac{1-1}{\gamma_{v-u}-1} = 0,$$

this follows by the formula for the sum of a geometric series, and as γ_{v-u} is an nth root of unity, and as such if we raise it to power n we get 1.

We just proved that the matrix C have ones on the diagonal and zero everywhere else. Namely, it is the identity matrix, establishing our claim that the given matrix is indeed the inverse matrix to the Vandermonde matrix. ■

Let us recap, given n point-value pairs $\{(\gamma_0, y_0), \ldots, (\gamma_{n-1}, y_{n-1})\}$ of a polynomial $p(x) = \sum_{i=0}^{n-1} a_i x^i$ over the set of nth roots of unity, then we can recover the coefficients of the polynomial by multiplying the vector $[y_0, y_1, \ldots, y_n]$ by the matrix V^{-1}. Namely,

$$\begin{pmatrix} a_0 \\ a_1 \\ a_2 \\ \vdots \\ a_{n-1} \end{pmatrix} = \underbrace{\frac{1}{n}\begin{pmatrix} 1 & \beta_0 & \beta_0^2 & \beta_0^3 & \cdots & \beta_0^{n-1} \\ 1 & \beta_1 & \beta_1^2 & \beta_1^3 & \cdots & \beta_1^{n-1} \\ 1 & \beta_2 & \beta_2^2 & \beta_2^3 & \cdots & \beta_2^{n-1} \\ 1 & \beta_3 & \beta_3^2 & \beta_3^3 & \cdots & \beta_3^{n-1} \\ \vdots & \vdots & \vdots & \vdots & \cdots & \vdots \\ 1 & \beta_{n-1} & \beta_{n-1}^2 & \beta_{n-1}^3 & \cdots & \beta_{n-1}^{n-1} \end{pmatrix}}_{V^{-1}} \begin{pmatrix} y_0 \\ y_1 \\ y_2 \\ y_3 \\ \vdots \\ y_{n-1} \end{pmatrix}.$$

Let us write a polynomial $W(x) = \sum_{i=0}^{n-1}(y_i/n)x^i$. It is clear that $a_i = W(\beta_i)$. That is to recover the coefficients of $p(\cdot)$, we have to compute a polynomial $W(\cdot)$ on n values: $\beta_0, \ldots, \beta_{n-1}$.

The final stroke, is to observe that $\{\beta_0, \ldots, \beta_{n-1}\} = \{\gamma_0, \ldots, \gamma_{n-1}\}$; indeed $\beta_i^n = (\gamma_i^{-1})^n = (\gamma_i^n)^{-1} = 1^{-1} = 1$. Namely, we can apply the **FFTAlg** algorithm on $W(x)$ to compute $a_0, \ldots, a_{n-1}$.

We conclude:

Theorem 18.3.2. *Given n point-value pairs of a polynomial $p(x)$ of degree $n-1$ over the set of n powers of the nth roots of unity, we can recover the polynomial $p(x)$ in $O(n \log n)$ time.*

Theorem 18.3.3. *Given two polynomials of degree n, they can be multiplied in $O(n \log n)$ time.*

18.4 The Convolution Theorem

Given two vectors: $A = [a_0, a_1, \ldots, a_n]$ and $B = [b_0, \ldots, b_n]$, their dot product is the quantity

$$A \cdot B = \langle A, B \rangle = \sum_{i=0}^{n} a_i b_i.$$

Let A_r denote the shifting of A by $n - r$ locations to the left (we pad it with zeros; namely, $a_j = 0$ for $j \notin \{0, \ldots, n\}$).

$$A_r = \Big[a_{n-r}, a_{n+1-r}, a_{n+2-r}, \ldots, a_{2n-r}\Big]$$

where $a_j = 0$ if $j \notin \Big[0, \ldots, n\Big]$.

Observation 18.4.1. $A_n = A$.

Example 18.4.2. For $A = [3, 7, 9, 15]$, $n = 3$
$A_2 = [7, 9, 15, 0]$,
$A_5 = [0, 0, 3, 7]$.

Definition 18.4.3. Let $c_i = A_i \cdot B = \sum_{j=n-i}^{2n-i} a_j b_{j-n+i}$, for $i = 0, \ldots, 2n$. The vector $[c_0, \ldots, c_{2n}]$ is the *convolution* of A and B.

Question 18.4.4. *How to compute the convolution of two vectors of length n?*

Definition 18.4.5. The resulting vector $[c_0, \ldots, c_{2n}]$ is the ***convolution*** of A and B.

Let $p(x) = \sum_{i=0}^{n} \alpha_i x^i$, and $q(x) = \sum_{i=0}^{n} \beta_i x^i$. The coefficient of x^i in $r(x) = p(x)q(x)$ is

$$d_i = \sum_{j=0}^{i} \alpha_j \beta_{i-j}.$$

On the other hand, we would like to compute $c_i = A_i \cdot B = \sum_{j=n-i}^{2n-i} a_j b_{j-n+i}$, which seems to be a very similar expression. Indeed, setting $\alpha_i = a_i$ and $\beta_l = b_{n-l-1}$ we get what we want.

To understand whats going on, observe that the coefficient of x^2 in the product of the two respective polynomials $p(x) = a_0 + a_1 x + a_2 x^2 + a_3 x^3$ and $q(x) = b_0 + b_1 x + b_2 x^2 + b_3 x^3$ is the sum of the entries on the anti diagonal in the following matrix, where the entry in the ith row and jth column is $a_i b_j$.

	a_0+	$a_1 x$	$+a_2 x^2$	$+a_3 x^3$
b_0			$a_2 b_0 x^2$	
$+b_1 x$		$a_1 b_1 x^2$		
$+b_2 x^2$	$a_0 b_2 x^2$			
$+b_3 x^3$				

Theorem 18.4.6. *Given two vectors $A = [a_0, a_1, \ldots, a_n]$, $B = [b_0, \ldots, b_n]$ one can compute their convolution in $O(n \log n)$ time.*

Proof: Let $p(x) = \sum_{i=0}^{n} a_{n-i} x^i$ and let $q(x) = \sum_{i=0}^{n} b_i x^i$. Compute $r(x) = p(x)q(x)$ in $O(n \log n)$ time using the convolution theorem. Let $c_0, \ldots, c_{2n}$ be the coefficients of $r(x)$. It is easy to verify, as described above, that $[c_0, \ldots, c_{2n}]$ is the convolution of A and B. ■

18.4.1 Complex numbers – a quick reminder

A complex number is a pair of real numbers x and y, written as $\tau = x + \mathbf{i}y$, where x is the ***real*** part and y is the ***imaginary*** part. Here $\mathbf{i}$ is of course the root of -1. In ***polar form***, we can write $\tau = r\cos\phi + \mathbf{i}r\sin\phi = r(\cos\phi + \mathbf{i}\sin\phi) = re^{i\phi}$, where $r = \sqrt{x^2 + y^2}$ and $\phi = \arcsin(y/x)$. To see the last part, define the following functions by their Taylor expansion

$$\begin{aligned} \sin x &= x - \frac{x^3}{3!} + \frac{x^5}{5!} - \frac{x^7}{7!} + \cdots, \\ \cos x &= 1 - \frac{x^2}{2!} + \frac{x^4}{4!} - \frac{x^6}{6!} + \cdots, \\ \text{and} \qquad e^x &= 1 + \frac{x}{1!} + \frac{x^2}{2!} + \frac{x^3}{3!} + \cdots. \end{aligned}$$

Since $\mathbf{i}^2 = -1$, we have that

$$e^{\mathbf{i}x} = 1 + \mathbf{i}\frac{x}{1!} - \frac{x^2}{2!} - \mathbf{i}\frac{x^3}{3!} + \frac{x^4}{4!} + \mathbf{i}\frac{x^5}{5!} - \frac{x^6}{6!} \cdots = \cos x + \mathbf{i}\sin x.$$

The nice thing about polar form, is that given two complex numbers $\tau = re^{\mathbf{i}\phi}$ and $\tau' = r'e^{\mathbf{i}\phi'}$, multiplying them is now straightforward. Indeed, $\tau \cdot \tau' = re^{\mathbf{i}\phi} \cdot r'e^{\mathbf{i}\phi'} = rr'e^{\mathbf{i}(\phi+\phi')}$. Observe that the function $e^{\mathbf{i}\phi}$ is 2π periodic (i.e., $e^{\mathbf{i}\phi} = e^{\mathbf{i}(\phi+2\pi)}$), and $1 = e^{\mathbf{i}0}$. As such, an nth root of 1, is a complex number $\tau = re^{\mathbf{i}\phi}$ such that $\tau^n = r^n e^{\mathbf{i}n\phi} = e^{\mathbf{i}0}$. Clearly, this implies that $r = 1$, and there must be an integer j, such that

$$n\phi = 0 + 2\pi j \implies \phi = j(2\pi/n).$$

These are all distinct values for $j = 0, \ldots, n-1$, which are the n distinct roots of unity.

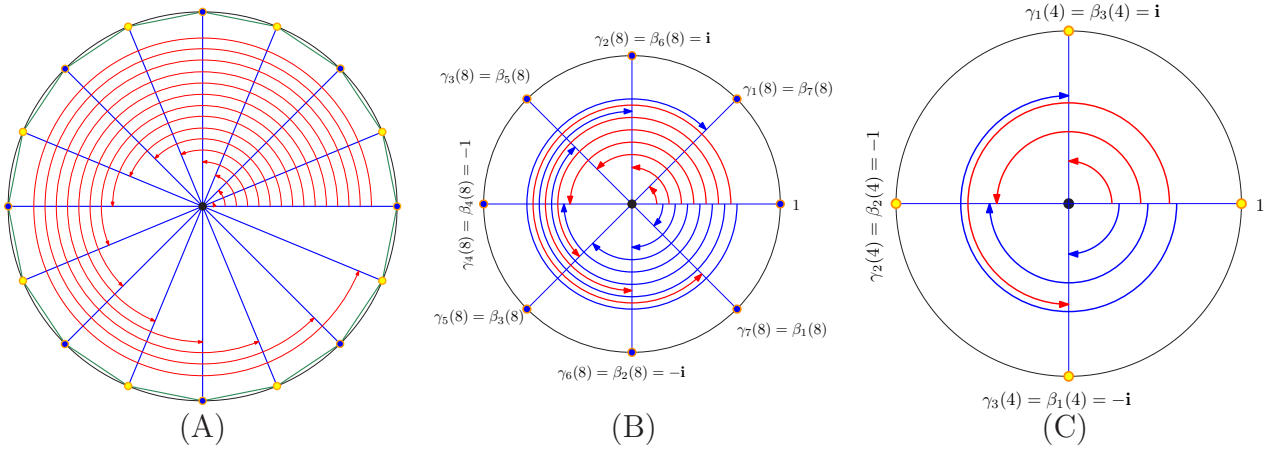


Figure 18.2: (A) The 16 roots of unity. (B) The 8 roots of unity. (C) The 4 roots of unity.

Part VII

Sorting Networks

Chapter 19

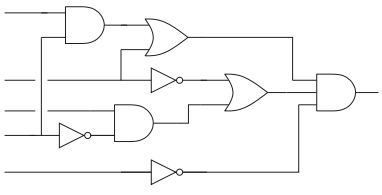
Sorting Networks

19.1 Model of Computation

It is natural to ask if one can perform a computational task considerably faster by using a different architecture (i.e., a different computational model).

The answer to this question is a resounding yes. A cute example is the ***Macaroni sort*** algorithm. We are given a set $S = \{s_1, \ldots, s_n\}$ of n real numbers in the range (say) $[1, 2]$. We get a lot of Macaroni (this are longish and very narrow tubes of pasta), and cut the ith piece to be of length s_i, for $i = 1, \ldots, n$. Next, take all these pieces of pasta in your hand, make them stand up vertically, with their bottom end lying on a horizontal surface. Next, lower your handle till it hit the first (i.e., tallest) piece of pasta. Take it out, measure it height, write down its number, and continue in this fashion till you have extracted all the pieces of pasta. Clearly, this is a sorting algorithm that works in linear time. But we know that sorting takes $\Omega(n \log n)$ time. Thus, this algorithm is much faster than the standard sorting algorithms.

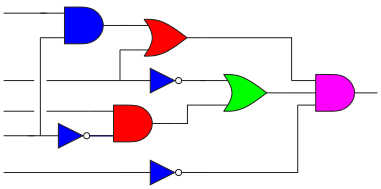
This faster algorithm was achieved by changing the computation model. We allowed new "strange" operations (cutting a piece of pasta into a certain length, picking the longest one in constant time, and measuring the length of a pasta piece in constant time). Using these operations we can sort in linear time.



If this was all we can do with this approach, that would have only been a curiosity. However, interestingly enough, there are natural computation models which are considerably stronger than the standard model of computation. Indeed, consider the task of computing the output of the circuit on the right (here, the input is boolean values on the input wires on the left, and the output is the single output on the right).

Clearly, this can be solved by ordering the gates in the "right" order (this can be done by topological sorting), and then computing the value of the gates one by one in this order, in such a way that a gate being computed knows the values arriving on its input wires. For the circuit above, this would require 8 units of time, since there are 8 gates.

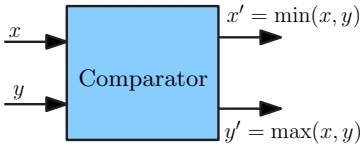
However, if you consider this circuit more carefully, one realized that we can compute this circuit in 4 time units. By using the fact that several gates are independent of each other, and we can compute them in parallel, as depicted on the right. Furthermore, circuits are inherently parallel and we should be able to take advantage of this fact.

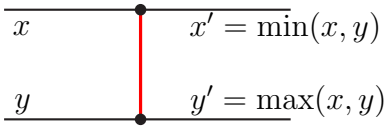


So, let us consider the classical problem of sorting n numbers. The question is whether we can sort them in *sublinear* time by allowing parallel comparisons. To this end, we need to precisely define our computation model.

19.2 Sorting with a circuit – a naive solution

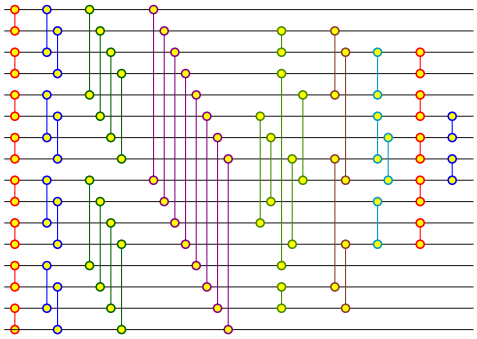
We are going to design a circuit, where the inputs are the numbers and we compare two numbers using a comparator gate. Such a gate has two inputs and two outputs, and it is depicted on the right.





We usually depict such a gate as a vertical segment connecting two wires, as depicted on the right. This would make drawing and arguing about sorting networks easier.

Our circuits would be depicted by horizontal lines, with vertical segments (i.e., gates) connecting between them. For example, see complete sorting network depicted on the right.



The inputs come on the wires on the left, and are output on the wires on the right. The largest number is output on the bottom line. Somewhat surprisingly, one can generate circuits from known sorting algorithms.

19.2.1 Definitions

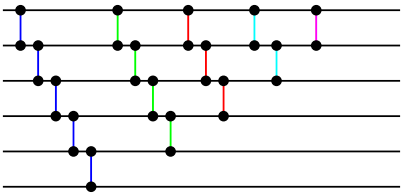
Definition 19.2.1. A ***comparison network*** is a DAG (directed acyclic graph), with n inputs and n outputs, where each gate (i.e., done) has two inputs and two outputs (i.e., two incoming edges, and two outgoing edges).

Definition 19.2.2. The ***depth*** of a wire is 0 at the input. For a gate with two inputs of depth d_1 and d_2 the depth on the output wire is $1 + \max(d_1, d_2)$. The ***depth*** of a comparison network is the maximum depth of an output wire.

Definition 19.2.3. A ***sorting network*** is a comparison network such that for any input, the output is monotonically sorted. The ***size*** of a sorting network is the number of gates in the sorting network. The ***running time*** of a sorting network is just its depth.

19.2.2 Sorting network based on insertion sort

Consider the sorting circuit on the left. Clearly, this is just the inner loop of the standard insertion sort. As such, if we repeat this loop, we get the sorting network on the right. It is easy to argue that this circuit sorts correctly all inputs (we removed some unnecessary gates).



An alternative way of drawing this sorting network is depicted in Figure 19.1 (ii). The next natural question, is how much time does it take for this circuit to sort the n numbers. Observe, that the running time of the algorithm is how many different time ticks we have to wait till the result stabilizes in all the gates. In our example, the alternative drawing immediately tell us how to schedule the computation of the gates. See Figure 19.1 (ii).

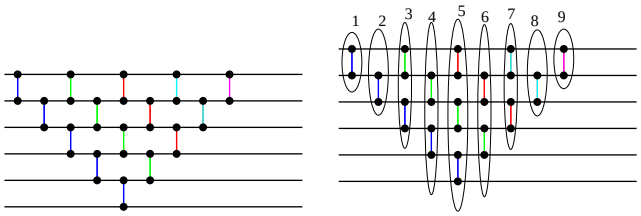


Figure 19.1: The sorting network inspired by insertion sort.

In particular, the above discussion implies the following result.

Lemma 19.2.4. *The sorting network based on insertion sort has $O(n^2)$ gates, and requires $2n-1$ time units to sort n numbers.*

19.3 The Zero-One Principle

The ***zero-one principle*** states that if a comparison network sort correctly all binary inputs (i.e., every number is either 0 or 1) then it sorts correctly all inputs. We (of course) need to prove that the zero-one principle is true.

Lemma 19.3.1. *If a comparison network transforms the input sequence $a = \langle a_1, a_2, \ldots, a_n \rangle$ into the output sequence $b = \langle b_1, b_2, \ldots, b_n \rangle$, then for any monotonically increasing function f, the network transforms the input sequence $f(a) = \langle f(a_1), \ldots, f(a_n) \rangle$ into the sequence $f(b) = \langle f(b_1), \ldots, f(b_n) \rangle$.*

Proof: Consider a single comparator with inputs x and y, and outputs $x' = \min(x, y)$ and $y' = \max(x, y)$. If $f(x) = f(y)$ then the claim trivially holds for this comparator. If $f(x) < f(y)$ then clearly

$$\begin{aligned} \max(f(x), f(y)) &= f(\max(x, y)) \text{ and} \\ \min(f(x), f(y)) &= f(\min(x, y)), \end{aligned}$$

since $f(\cdot)$ is monotonically increasing. As such, for the input $\langle x, y\rangle$, for $x < y$, we have output $\langle x, y\rangle$. Thus, for the input $\langle f(x), f(y)\rangle$ the output is $\langle f(x), f(y)\rangle$. Similarly, if $x > y$, the output is $\langle y, x\rangle$. In this case, for the input $\langle f(x), f(y)\rangle$ the output is $\langle f(y), f(x)\rangle$. This establish the claim for a single comparator.

Now, we claim by induction that if a wire carry a value a_i, when the sorting network get input $a_1, \ldots, a_n$, then for the input $f(a_1), \ldots, f(a_n)$ this wire would carry the value $f(a_i)$.

This is proven by induction on the depth on the wire at each point. If the point has depth 0, then its an input and the claim trivially hold. So, assume it holds for all points in our circuits of depth at most i, and consider a point p on a wire of depth $i + 1$. Let G be the gate which this wire is an output of. By induction, we know the claim holds for the inputs of G (which have depth at most i). Now, we the claim holds for the gate G itself, which implies the claim apply the above claim to the gate G, which implies the claim holds at p. ■

Theorem 19.3.2. *If a comparison network with n inputs sorts all 2^n binary strings of length n correctly, then it sorts all sequences correctly.*

Proof: Assume for the sake of contradiction, that it sorts incorrectly the sequence $a_1, \ldots, a_n$. Let $b_1, \ldots b_n$ be the output sequence for this input.

Let $a_i < a_k$ be the two numbers that are output in incorrect order (i.e. a_k appears before a_i in the output). Let

$$f(x) = \begin{cases} 0 & x \leq a_i \\ 1 & x > a_i. \end{cases}$$

Clearly, by the above lemma (Lemma 19.3.1), for the input

$$\langle f(a_1), \ldots, f(a_n)\rangle,$$

which is a binary sequence, the circuit would output $\langle f(b_1), \ldots, f(b_n)\rangle$. But then, this sequence looks like

$$000..0????f(a_k)????f(a_i)??1111$$

but $f(a_i) = 0$ and $f(a_j) = 1$. Namely, the output is a sequence of the form ????1????0????, which is not sorted.

Namely, we have a binary input (i.e., $\langle f(b_1), \ldots, f(b_n)\rangle$) for which the comparison network does not sort it correctly. A contradiction to our assumption. ■

19.4 A bitonic sorting network

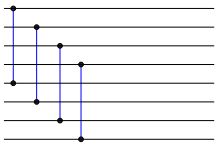
Definition 19.4.1. A ***bitonic sequence*** is a sequence which is first increasing and then decreasing, or can be circularly shifted to become so.

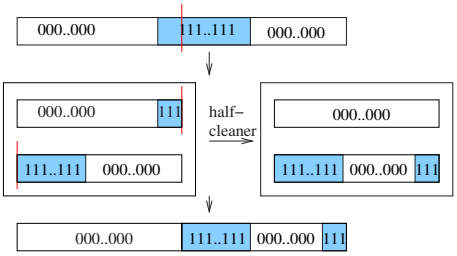
Example 19.4.2. The sequences $(1, 2, 3, \pi, 4, 5, 4, 3, 2, 1)$ and $(4, 5, 4, 3, 2, 1, 1, 2, 3)$ are bitonic, while the sequence $(1, 2, 1, 2)$ is not bitonic.

Observation 19.4.3. *A binary bitonic sequence (i.e., bitonic sequence made out only of zeroes and ones) is either of the form $0^i1^j0^k$ or of the form $1^i0^j1^k$, where 0^i (resp, 1^i) denote a sequence of i zeros (resp., ones).*

Definition 19.4.4. A ***bitonic sorter*** is a comparison network that sorts all bitonic sequences correctly.

Definition 19.4.5. A ***half-cleaner*** is a comparison network, connecting line i with line $i+n/2$. In particular, let **Half-Cleaner**$[n]$ denote the half-cleaner with n inputs. Note, that the depth of a **Half-Cleaner**$[n]$ is one, see figure on the right.





It is beneficial to consider what a half-cleaner do to an input which is a (binary) bitonic sequence. Clearly, in the specific example, depicted on the left, we have that the left half size is clean and all equal to 0. Similarly, the right size of the output is bitonic.

Specifically, one can prove by simple (but tedious) case analysis that the following lemma holds.

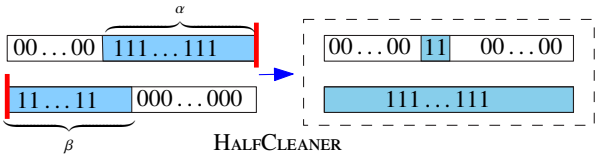
Lemma 19.4.6. *If the input to a half-cleaner (of size n) is a binary bitonic sequence then for the output sequence we have that*

(i) the elements in the top half are smaller than the elements in bottom half, and

(ii) one of the halves is clean, and the other is bitonic.

Proof: If the sequence is of the form $0^i1^j0^k$ and the block of ones is completely on the left side (i.e., its part of the first $n/2$ bits) or the right side, the claim trivially holds. So, assume that the block of ones starts at position $n/2-\beta$ and ends at $n/2+\alpha$.

If $n/2-\alpha \geq \beta$ then this is exactly the case depicted above and claim holds. If $n/2-\alpha < \beta$ then the second half is going to be all ones, as depicted on the right. Implying the claim for this case.



A similar analysis holds if the sequence is of the form $1^i0^j1^k$. ■

This suggests a simple recursive construction of **BitonicSorter**$[n]$, see Figure 19.2, and we have the following lemma.

Lemma 19.4.7. **BitonicSorter**$[n]$ *sorts bitonic sequences of length $n = 2^k$, it uses $(n/2)k = (n/2)\lg n$ gates, and it is of depth $k = \lg n$.*

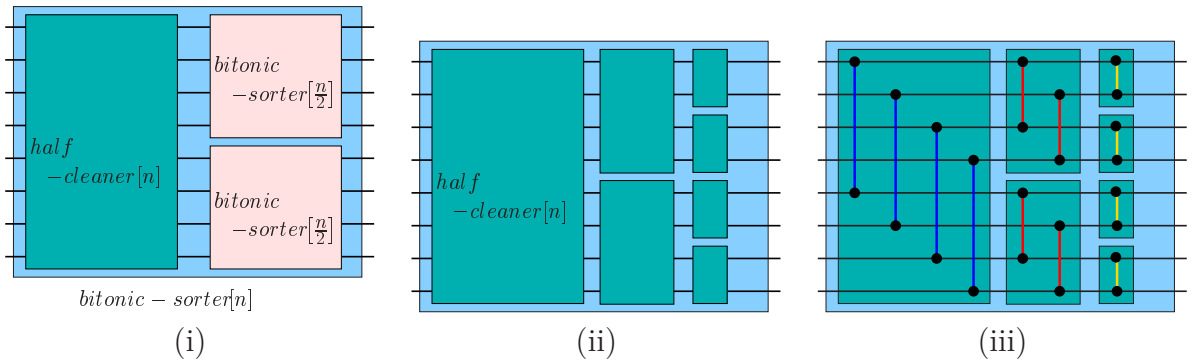


Figure 19.2: Depicted are the (i) recursive construction of **BitonicSorter**$[n]$, (ii) opening up the recursive construction, and (iii) the resulting comparison network.

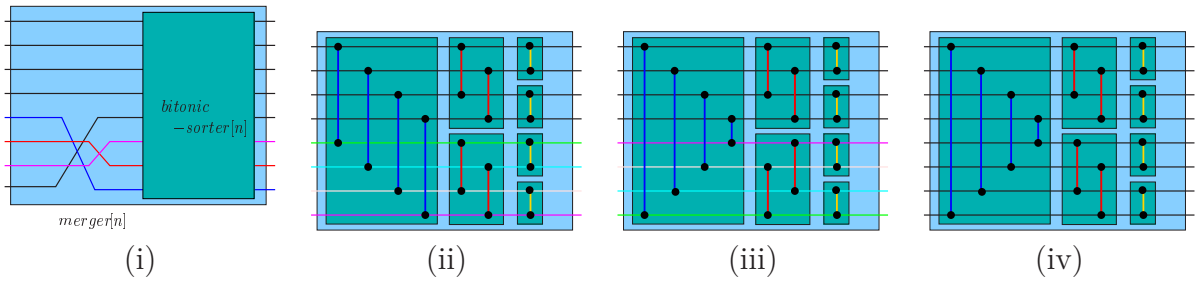


Figure 19.3: (i) **Merger** via flipping the lines of bitonic sorter. (ii) A **BitonicSorter**. (ii) The **Merger** after we "physically" flip the lines, and (iv) An equivalent drawing of the resulting **Merger**.

19.4.1 Merging sequence

Next, we deal with the following merging question. Given two *sorted* sequences of length $n/2$, how do we merge them into a single sorted sequence?

The idea here is concatenate the two sequences, where the second sequence is being flipped (i.e., reversed). It is easy to verify that the resulting sequence is bitonic, and as such we can sort it using the **BitonicSorter**$[n]$.

Specifically, given two sorted sequences $a_1 \leq a_2 \leq \ldots \leq a_n$ and $b_1 \leq b_2 \leq \ldots \leq b_n$, observe that the sequence $a_1, a_2, \ldots, a_n, b_n, b_{n-1}, b_{n-2}, \ldots, b_2, b_1$ is bitonic.

Thus, to merge two sorted sequences of length $n/2$, just flip one of them, and use **BitonicSorter**$[n]$, see Figure 19.3. This is of course illegal, and as such we take **BitonicSorter**$[n]$ and physically flip the last $n/2$ entries. The process is depicted in Figure 19.3. The resulting circuit **Merger** takes two sorted sequences of length $n/2$, and return a sorted sequence of length n.

It is somewhat more convenient to describe the **Merger** using a **FlipCleaner** component. See Figure 19.4

Lemma 19.4.8. *The circuit* **Merger**$[n]$ *gets as input two sorted sequences of length* $n/2 = 2^{k-1}$*, it uses* $(n/2)k = (n/2)\lg n$ *gates, and it is of depth* $k = \lg n$*, and it outputs a sorted*

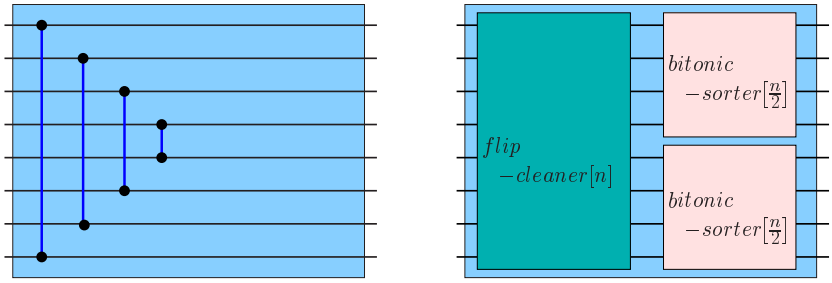
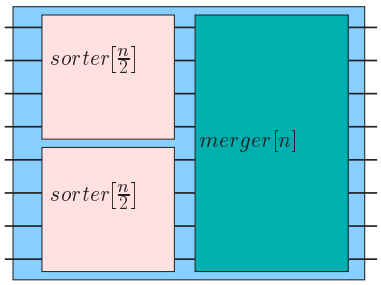


Figure 19.4: (i) **FlipCleaner**$[n]$, and (ii) **Merger**$[n]$ described using **FlipCleaner**.

sequence.

19.5 Sorting Network



We are now in the stage, where we can build a sorting network. To this end, we just implement ***merge sort*** using the **Merger**$[n]$ component. The resulting component **Sorter**$[n]$ is depicted on the right using a recursive construction.

Lemma 19.5.1. *The circuit* **Sorter**$[n]$ *is a sorting network (i.e., it sorts any* n *numbers) using* $G(n) = O(n \log^2 n)$ *gates. It has depth* $O(\log^2 n)$*. Namely,* **Sorter**$[n]$ *sorts* n *numbers in* $O(\log^2 n)$ *time.*

Proof: The number of gates is

$$G(n) = 2G(n/2) + Gates(\textbf{Merger}[n]).$$

Which is $G(n) = 2G(n/2) + O(n \log n) = O(n \log^2 n)$.

As for the depth, we have that $D(n) = D(n/2) + \text{Depth}(\textbf{Merger}[n]) = D(n/2) + O(\log(n))$, and thus $D(n) = O(\log^2 n)$, as claimed. ■

19.6 Faster sorting networks

One can build a sorting network of logarithmic depth (see [AKS83]). The construction however is very complicated. A simpler parallel algorithm would be discussed sometime in the next lectures. BTW, the AKS construction [AKS83] mentioned above, is better than bitonic sort for n larger than 2^{8046}.

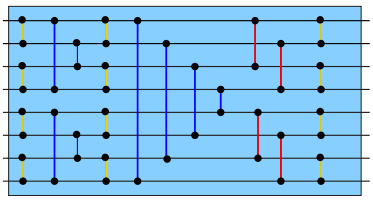


Figure 19.5: **Sorter**[8].

Part VIII

Linear Programming

Chapter 20

Linear Programming

20.1 Introduction and Motivation

In the VCR/guns/nuclear-bombs/napkins/star-wars/professors/butter/mice problem, the benevolent dictator, Biga Piguinus, of Penguina (a country in south Antarctica having 24 million penguins under its control) has to decide how to allocate her empire resources to the maximal benefit of her penguins. In particular, she has to decide how to allocate the money for the next year budget. For example, buying a nuclear bomb has a tremendous positive effect on security (the ability to destruct yourself completely together with your enemy induces a peaceful serenity feeling in most people). Guns, on the other hand, have a weaker effect. Penguina (the state) has to supply a certain level of security. Thus, the allocation should be such that:

$$x_{gun} + 1000 * x_{nuclear-bomb} \geq 1000,$$

where x_{guns} is the number of guns constructed, and $x_{nuclear-bomb}$ is the number of nuclear-bombs constructed. On the other hand,

$$100 * x_{gun} + 1000000 * x_{nuclear-bomb} \leq x_{security}$$

where $x_{security}$ is the total Penguina is willing to spend on security, and 100 is the price of producing a single gun, and $1,000,000$ is the price of manufacturing one nuclear bomb. There are a lot of other constrains of this type, and Biga Piguinus would like to solve them, while minimizing the total money allocated for such spending (the less spent on budget, the larger the tax cut).

More formally, we have a (potentially large) number of variables: $x_1, \ldots, x_n$ and a (potentially large) system of linear inequalities. We will refer to such an inequality as a ***constraint***. We would like to decide if there is an assignment of values to $x_1, \ldots, x_n$ where all these inequalities are satisfied. Since there might be infinite number of such solutions, we want the solution that maximizes some linear quantity. See the instance on the right.

$$\begin{array}{l} a_{11}x_1 + \ldots + a_{1n}x_n \leq b_1 \\ a_{21}x_1 + \ldots + a_{2n}x_n \leq b_2 \\ \qquad\qquad \vdots \\ a_{m1}x_1 + \ldots + a_{mn}x_n \leq b_m \\ \max \quad c_1x_1 + \ldots + c_nx_n. \end{array}$$

The linear target function we are trying to maximize is known as the ***objective function*** of the linear program. Such a problem is an instance of ***linear programming***. We refer to linear programming as LP.

20.1.1 History

Linear programming can be traced back to the early 19th century. It started in earnest in 1939 when L. V. Kantorovich noticed the importance of certain type of Linear Programming problems. Unfortunately, for several years, Kantorovich work was unknown in the west and unnoticed in the east.

Dantzig, in 1947, invented the simplex method for solving LP problems for the US Air force planning problems.

T. C. Koopmans, in 1947, showed that LP provide the right model for the analysis of classical economic theories.

In 1975, both Koopmans and Kantorovich got the Nobel prize of economics. Dantzig probably did not get it because his work was too mathematical. That is how it goes. Kantorovich was the only the Russian economist that got the Nobel prize[①].

20.1.2 Network flow via linear programming

To see the impressive expressive power of linear programming, we next show that network flow can be solved using linear programming. Thus, we are given an instance of max flow; namely, a network flow $\mathsf{G} = (\mathsf{V}, \mathsf{E})$ with source s and sink t, and capacities $\mathsf{c}(\cdot)$ on the edges. We would like to compute the maximum flow in G.

To this end, for an edge $(u \to v) \in E$, let $x_{u\to v}$ be a variable which is the amount of flow assign to $(u \to v)$ in the maximum flow. We demand that $0 \le x_{u\to v}$ and $x_{u\to v} \le \mathsf{c}(u \to v)$ (flow is non negative on edges, and it comply with the capacity constraints). Next, for any vertex v which is not the source or the sink, we require that $\sum_{(u\to v)\in E} x_{u\to v} = \sum_{(v\to w)\in E} x_{v\to w}$ (this is conservation of flow). Note, that an equality constraint $a = b$ can be rewritten as two inequality constraints $a \le b$ and $b \le a$. Finally, under all these constraints, we are interest in the maximum flow. Namely, we would like to maximize the quantity $\sum_{(\mathsf{s}\to u)\in E} x_{\mathsf{s}\to u}$. Clearly, putting all these constraints together, we get the linear program depicted on the right.

$\forall (u \to v) \in E$	$0 \le x_{u\to v}$
	$x_{u\to v} \le \mathsf{c}(u \to v)$
$\forall v \in V \setminus \{\mathsf{s}, \mathsf{t}\}$	$\sum_{(u\to v)\in E} x_{u\to v} - \sum_{(v\to w)\in E} x_{v\to w} \le 0$
	$\sum_{(u\to v)\in E} x_{u\to v} - \sum_{(v\to w)\in E} x_{v\to w} \ge 0$
maximizing	$\sum_{(\mathsf{s}\to u)\in E} x_{\mathsf{s}\to u}$

It is not too hard to write down min-cost network flow using linear programming.

[①] There were other economists that were born in Russia, but lived in the west that got the Nobel prize – Leonid Hurwicz for example.

20.2 The Simplex Algorithm

20.2.1 Linear program where all the variables are positive

$$\begin{aligned} \max \quad & \sum_{j=1}^{n} c_j x_j \\ \text{subject to} \quad & \sum_{j=1}^{n} a_{ij} x_j \leq b_i \\ & \text{for} \quad i = 1, 2, \ldots, m. \end{aligned}$$

We are given a LP, depicted on the left, where a variable can have any real value. As a first step to solving it, we would like to rewrite it, such that every variable is non-negative. This is easy to do, by replacing a variable x_i by two new variables x'_i and x''_i, where $x_i = x'_i - x''_i$, $x'_i \geq 0$ and $x''_i \geq 0$. For example, the (trivial) linear program containing the single constraint $2x + y \geq 5$ would be replaced by the following LP: $2x' - 2x'' + y' - y'' \geq 5$, $x' \geq 0$, $y' \geq 0$, $x'' \geq 0$ and $y'' \geq 0$.

Lemma 20.2.1. *Given an instance I of LP, one can rewrite it into an equivalent LP, such that all the variables must be non-negative. This takes linear time in the size of I.*

20.2.2 Standard form

Using Lemma 20.2.1, we can now require a LP to be specified using only positive variables. This is known as ***standard form***.

A linear program in standard form.

$$\begin{aligned} \max \quad & \sum_{j=1}^{n} c_j x_j \\ \text{subject to} \quad & \sum_{j=1}^{n} a_{ij} x_j \leq b_i \quad \text{for } i = 1, 2, \ldots, m \\ & x_j \geq 0 \qquad\qquad \text{for} \quad j = 1, \ldots, n. \end{aligned}$$

A linear program in standard form.
(Matrix notation.)

$$\begin{aligned} \max \quad & c^T x \\ \text{subject to} \quad & Ax \leq b. \\ & x \geq 0. \end{aligned}$$

Here the matrix notation rises, by setting

$$c = \begin{pmatrix} c_1 \\ \vdots \\ c_n \end{pmatrix}, b = \begin{pmatrix} b_1 \\ \vdots \\ b_m \end{pmatrix}, A = \begin{pmatrix} a_{11} & a_{12} & \ldots & a_{1(n-1)} & a_{1n} \\ a_{21} & a_{22} & \ldots & a_{2(n-1)} & a_{2n} \\ \vdots & \ldots & \ldots & \ldots & \vdots \\ a_{(m-1)1} & a_{(m-1)2} & \ldots & a_{(m-1)(n-1)} & a_{(m-1)n} \\ a_{m1} & a_{m2} & \ldots & a_{m(n-1)} & a_{mn} \end{pmatrix},$$

$$\text{and } x = \begin{pmatrix} x_1 \\ x_2 \\ \vdots \\ x_{n-1} \\ x_n \end{pmatrix}.$$

Note, that c, b and A are prespecified, and x is the vector of unknowns that we have to solve the LP for.

In the following in order to solve the LP, we are going to do a long sequence of rewritings till we reach the optimal solution.

20.2.3 Slack Form

We next rewrite the LP into ***slack form***. It is a more convenient[2] form for describing the **Simplex** algorithm for solving LP.

Specifically, one can rewrite a LP, so that every inequality becomes equality, and all variables must be positive; namely, the new LP will have a form depicted on the right (using matrix notation). To this end, we introduce new variables (***slack variables***) rewriting the inequality

max	$c^T x$
subject to	$Ax = b.$
	$x \geq 0.$

$$\sum_{i=1}^{n} a_i x_i \leq b$$

as

$$\begin{aligned} x_{n+1} &= b - \sum_{i=1}^{n} a_i x_i \\ x_{n+1} &\geq 0. \end{aligned}$$

Intuitively, the value of the slack variable x_{n+1} encodes how far is the original inequality for holding with equality.

Now, we have a special variable for each inequality in the LP (this is x_{n+1} in the above example). This variables are special, and would be called ***basic variables***. All the other variables on the right side are ***nonbasic variables*** (original isn't it?). A LP in this form is in ***slack form***.

The slack form is defined by a tuple (N, B, A, b, c, v).

Linear program in slack form.

$$\begin{aligned} \max \quad & z = v + \sum_{j \in N} c_j x_j, \\ \text{s.t.} \quad & x_i = b_i - \sum_{j \in N} a_{ij} x_j \quad for \quad i \in B, \\ & x_i \geq 0, \quad \forall i = 1, \ldots, n+m. \end{aligned}$$

B - Set of indices of basic variables
N - Set of indices of nonbasic variables
$n = |N|$ - number of original variables
b, c - two vectors of constants
$m = |B|$ - number of basic variables
(i.e., number of inequalities)
$A = \{a_{ij}\}$ - The matrix of coefficients
$N \cup B = \{1, \ldots, n+m\}$
v - objective function constant.

Exercise 20.2.2. Show that any linear program can be transformed into equivalent slack form.

[2] The word *convenience* is used here in the most liberal interpretation possible.

Example 20.2.3. Consider the following LP which is in slack form, and its translation into the tuple (N, B, A, b, c, v).

$$\begin{aligned}
\max \quad & z = 29 - \frac{1}{9}x_3 - \frac{1}{9}x_5 - \frac{2}{9}x_6 \\
& x_1 = 8 + \frac{1}{6}x_3 + \frac{1}{6}x_5 - \frac{1}{3}x_6 \\
& x_2 = 4 - \frac{8}{3}x_3 - \frac{2}{3}x_5 + \frac{1}{3}x_6 \\
& x_4 = 18 - \frac{1}{2}x_3 + \frac{1}{2}x_5
\end{aligned}$$

$$B = \{1, 2, 4\}, N = \{3, 5, 6\}$$

$$A = \begin{pmatrix} a_{13} & a_{15} & a_{16} \\ a_{23} & a_{25} & a_{26} \\ a_{43} & a_{45} & a_{46} \end{pmatrix} = \begin{pmatrix} -1/6 & -1/6 & 1/3 \\ 8/3 & 2/3 & -1/3 \\ 1/2 & -1/2 & 0 \end{pmatrix}$$

$$b = \begin{pmatrix} b_1 \\ b_2 \\ b_4 \end{pmatrix} = \begin{pmatrix} 8 \\ 4 \\ 18 \end{pmatrix} \qquad c = \begin{pmatrix} c_3 \\ c_5 \\ c_6 \end{pmatrix} = \begin{pmatrix} -1/9 \\ -1/9 \\ -2/9 \end{pmatrix}$$

$$v = 29.$$

Note that indices depend on the sets N and B, and also that the entries in A are negation of what they appear in the slack form.

20.2.4 The Simplex algorithm by example

Before describing the **Simplex** algorithm in detail, it would be beneficial to derive it on an example. So, consider the following LP.

$$\begin{aligned}
\max \quad & 5x_1 + 4x_2 + 3x_3 \\
s.t. \quad & 2x_1 + 3x_2 + x_3 \leq 5 \\
& 4x_1 + x_2 + 2x_3 \leq 11 \\
& 3x_1 + 4x_2 + 2x_3 \leq 8 \\
& x_1, x_2, x_3 \geq 0
\end{aligned}$$

Next, we introduce slack variables, for example, rewriting $2x_1 + 3x_2 + x_3 \leq 5$ as the constraints: $w_1 \geq 0$ and $w_1 = 5 - 2x_1 - 3x_2 - x_3$. The resulting LP in slack form is

$$\begin{aligned}
\max \quad z = \quad & 5x_1 + 4x_2 + 3x_3 \\
s.t. \quad w_1 = \quad & 5 - 2x_1 - 3x_2 - x_3 \\
w_2 = \quad & 11 - 4x_1 - x_2 - 2x_3 \\
w_3 = \quad & 8 - 3x_1 - 4x_2 - 2x_3 \\
& x_1, x_2, x_3, w_1, w_2, w_3 \geq 0
\end{aligned}$$

Here w_1, w_2, w_3 are the slack variables. Note also that they are currently also the basic variables. Consider the slack representation trivial solution, where all the non-basic variables are assigned zero; namely, $x_1 = x_2 = x_3 = 0$. We then have that $w_1 = 5$, $w_2 = 11$ and $w_3 = 8$. Fortunately for us, this is a feasible solution, and the associated objective value is $z = 0$.

We are interested in further improving the value of the objective function (i.e., z), while still having a feasible solution. Inspecting carefully the above LP, we realize that all the basic variables $w_1 = 5$, $w_2 = 11$ and $w_3 = 8$ have values which are strictly larger than zero. Clearly, if we change the value of one non-basic variable a bit, all the basic variables would

remain positive (we are thinking about the above system as being function of the nonbasic variables x_1, x_2 and x_3). So, consider the objective function $z = 5x_1 + 4x_2 + 3x_3$. Clearly, if we increase the value of x_1, from its current zero value, then the value of the objective function would go up, since the coefficient of x_1 for z is a positive number (5 in our example).

Deciding how much to increase the value of x_1 is non-trivial. Indeed, as we increase the value of x_1, the the solution might stop being feasible (although the objective function values goes up, which is a good thing). So, let us increase x_1 as much as possible without violating any constraint. In particular, for $x_2 = x_3 = 0$ we have that

$$\begin{aligned} w_1 &= 5 - 2x_1 - 3x_2 - x_3 &&= 5 - 2x_1 \\ w_2 &= 11 - 4x_1 - x_2 - 2x_3 &&= 11 - 4x_1 \\ w_3 &= 8 - 3x_1 - 4x_2 - 2x_3 &&= 8 - 3x_1. \end{aligned}$$

We want to increase x_1 as much as possible, as long as w_1, w_2, w_3 are non-negative. Formally, the constraints are that

$$\begin{aligned} w_1 = 5 - 2x_1 \geq 0, \\ w_2 = 11 - 4x_1 \geq 0, \\ \text{and } w_3 = 8 - 3x_1 \geq 0. \end{aligned}$$

This implies that whatever value we pick for x_1 it must comply with the inequalities $x_1 \leq 2.5$, $x_1 \leq 11/4 = 2.75$ and $x_1 \leq 8/3 = 2.66$. We select as the value of x_1 the largest value that still comply with all these conditions. Namely, $x_1 = 2.5$. Putting it into the system, we now have a solution which is

$$x_1 = 2.5,\ x_2 = 0,\ x_3 = 0,\ w_1 = 0,\ w_2 = 1,\ w_3 = 0.5 \qquad \Rightarrow \qquad z = 5x_1 + 4x_2 + 3x_3 = 12.5.$$

As such, all the variables are non-negative and this solution is feasible. Furthermore, this is a better solution than the previous one, since the old solution had (the objective function) value $z = 0$.

What really happened? One zero nonbasic variable (i.e., x_1) became non-zero, and one basic variable became zero (i.e., w_1). It is natural now to want to exchange between the nonbasic variable x_1 (since it is no longer zero) and the basic variable w_1. This way, we will preserve the invariant, that the current solution we maintain is the one where all the nonbasic variables are assigned zero.

So, consider the equality in the LP that involves w_1, that is $w_1 = 5 - 2x_1 - 3x_2 - x_3$. We can rewrite this equation, so that x_1 is on the left side:

$$x_1 = 2.5 - 0.5w_1 - 1.5x_2 - 0.5\ x_3. \tag{20.1}$$

The problem is that x_1 still appears in the right size of the equations for w_2 and w_3 in the LP. We observe, however, that any appearance of x_1 can be replaced by substituting it by

the expression on the right side of Eq. (20.1). Collecting similar terms, we get the following equivalent LP:

$$\begin{aligned}
\max \quad z &= 12.5 - 2.5w_1 - 3.5x_2 + 0.5x_3 \\
x_1 &= 2.5 - 0.5w_1 - 1.5x_2 - 0.5x_3 \\
w_2 &= 1 + 2w_1 + 5x_2 \\
w_3 &= 0.5 + 1.5w_1 + 0.5x_2 - 0.5x_3.
\end{aligned}$$

Note, that the nonbasic variables are now $\{w_1, x_2, x_3\}$ and the basic variables are $\{x_1, w_2, w_3\}$. In particular, the trivial solution, of assigning zero to all the nonbasic variables is still feasible; namely we set $w_1 = x_2 = x_3 = 0$. Furthermore, the value of this solution is 12.5.

This rewriting step, we just did, is called ***pivoting***. And the variable we pivoted on is x_1, as x_1 was transfered from being a nonbasic variable into a basic variable.

We would like to continue pivoting till we reach an optimal solution. We observe, that we can not pivot on w_1, since if we increase the value of w_1 then the objective function value goes down, since the coefficient of w_1 is -2.5. Similarly, we can not pivot on x_2 since its coefficient in the objective function is -3.5. Thus, we can only pivot on x_3 since its coefficient in the objective function is 0.5, which is a positive number.

Checking carefully, it follows that the maximum we can increase x_3 is to 1, since then w_3 becomes zero. Thus, rewriting the equality for w_3 in the LP; that is,

$$w_3 = 0.5 + 1.5w_1 + 0.5x_2 - 0.5x_3,$$

for x_3, we have

$$x_3 = 1 + 3w_1 + x_2 - 2w_3,$$

Substituting this into the LP, we get the following LP.

$$\begin{aligned}
\max \quad z &= 13 - w_1 - 3x_2 - w_3 \\
s.t. \quad x_1 &= 2 - 2w_1 - 2x_2 + w_3 \\
w_2 &= 1 + 2w_1 + 5x_2 \\
x_3 &= 1 + 3w_1 + x_2 - 2w_3
\end{aligned}$$

Can we further improve the current (trivial) solution that assigns zero to all the nonbasic variables? (Here the nonbasic variables are $\{w_1, x_2, w_3\}$.)

The resounding answer is no. We had reached the optimal solution. Indeed, all the coefficients in the objective function are negative (or zero). As such, the trivial solution (all nonbasic variables get zero) is maximal, as they must all be non-negative, and increasing their value decreases the value of the objective function. So we better stop.

Intuition. The crucial observation underlining our reasoning is that at each stage we had replace the LP by a completely equivalent LP. In particular, any feasible solution to the original LP would be feasible for the final LP (and vice versa). Furthermore, they would

have exactly the same objective function value. However, in the final LP, we get an objective function that can not be improved for any feasible point, an we stopped. Thus, we found the optimal solution to the linear program.

This gives a somewhat informal description of the simplex algorithm. At each step we pivot on a nonbasic variable that improves our objective function till we reach the optimal solution. There is a problem with our description, as we assumed that the starting (trivial) solution of assigning zero to the nonbasic variables is feasible. This is of course might be false. Before providing a formal (and somewhat tedious) description of the above algorithm, we show how to resolve this problem.

20.2.4.1 Starting somewhere

$$\begin{aligned} \max \quad & z = v + \sum_{j \in N} c_j x_j, \\ \text{s.t.} \quad & x_i = b_i - \sum_{j \in N} a_{ij} x_j \qquad \text{for } i \in B, \\ & x_i \geq 0, \qquad \forall i = 1, \ldots, n+m. \end{aligned}$$

We had transformed a linear programming problem into slack form. Intuitively, what the **Simplex** algorithm is going to do, is to start from a feasible solution and start walking around in the feasible region till it reaches the best possible point as far as the objective function is concerned. But *maybe* the linear program L is not feasible at all (i.e., no solution exists.). Let L be a linear program (in slack form depicted on the left. Clearly, if we set all $x_i = 0$ if $i \in N$ then this determines the values of the basic variables. If they are all positive, we are done, as we found a feasible solution. The problem is that they might be negative.

We generate a new LP problem L' from L. This LP $L' =$ **Feasible**(L) is depicted on the right. Clearly, if we pick $x_j = 0$ for all $j \in N$ (all the nonbasic variables), and a value large enough for x_0 then all the basic variables would be non-negatives, and as such, we have found a feasible solution for L'. Let **LPStartSolution**(L') denote this easily computable feasible solution.

$$\begin{aligned} \min \quad & x_0 \\ \text{s.t.} \quad & x_i = x_0 + b_i - \sum_{j \in N} a_{ij} x_j \qquad \text{for } i \in B, \\ & x_i \geq 0, \quad \forall i = 1, \ldots, n+m. \end{aligned}$$

We can now use the **Simplex** algorithm we described to find this optimal solution to L' (because we have a feasible solution to start from!).

Lemma 20.2.4. *The LP L is feasible if and only if the optimal objective value of LP L' is zero.*

Proof: A feasible solution to L is immediately an optimal solution to L' with $x_0 = 0$, and vice versa. Namely, given a solution to L' with $x_0 = 0$ we can transform it to a feasible solution to L by removing x_0. ∎

One technicality that is ignored above, is that the starting solution we have for L', generated by **LPStartSolution**(L) is not legal as far as the slack form is concerned, because the non-basic variable x_0 is assigned a non-zero value. However, this can be easily resolved by immediately pivoting on x_0 when we run the **Simplex** algorithm. Namely, we first try to decrease x_0 as much as possible.

Chapter 21

Linear Programming II

21.1 The Simplex Algorithm in Detail

The **Simplex** algorithm is presented on the right. We assume that we are given **SimplexInner**, a black box that solves a LP if the trivial solution of assigning zero to all the nonbasic variables is feasible. We remind the reader that $L' =$ **Feasible**(L) returns a new LP for which we have an easy feasible solution. This is done by introducing a new variable x_0 into the LP, where the original LP $\widehat{L}$ is feasible if and only if the new LP L has a feasible solution with $x_0 = 0$. As such, we set the target function in L to be minimizing x_0.

Simplex($\widehat{L}$ a LP)
    Transform $\widehat{L}$ into slack form.
    Let L be the resulting slack form.
    $L' \leftarrow$ **Feasible**(L)
    $x \leftarrow$ **LPStartSolution**(L')
    $x' \leftarrow$ **SimplexInner**(L', x) (*)
    $z \leftarrow$ objective function value of x'
    if $z > 0$ **then**
        return "No solution"
    $x'' \leftarrow$ **SimplexInner**(L, x')
    return x''

Figure 21.1: The **Simplex** algorithm.

We now apply **SimplexInner** to L' and the easy solution computed for L' by **LPStartSolution**(L'). If $x_0 > 0$ in the optimal solution for L' then there is no feasible solution for L, and we exit. Otherwise, we found a feasible solution to L, and we use it as the starting point for **SimplexInner** when it is applied to L.

Thus, in the following, we have to describe **SimplexInner** - a procedure to solve an LP in slack form, when we start from a feasible solution defined by the nonbasic variables assigned value zero.

One technicality that is ignored above, is that the starting solution we have for L', generated by **LPStartSolution**(L) is not legal as far as the slack form is concerned, because the non-basic variable x_0 is assigned a non-zero value. However, this can be easily resolve by immediately pivot on x_0 when we execute (*) in Figure 21.1. Namely, we first try to decrease x_0 as much as possible.

21.2 The SimplexInner Algorithm

We next describe the **SimplexInner** algorithm.

We remind the reader that the LP is given to us in slack form, see Figure **??**. Furthermore, we assume that the trivial solution $x = \tau$, which is assigning all nonbasic variables zero, is feasible. In particualr, we immediately get the objective value for this solution from the notation which is v.

Assume, that we have a nonbasic variable x_e that appears in the objective function, and furthermore its coefficient c_e is positive in (the objective function), which is $z = v + \sum_{j \in N} c_j x_j$. Formally, we pick e to be one of the indices of

$$\left\{ j \;\middle|\; c_j > 0, j \in N \right\}.$$

The variable x_e is the ***entering variable*** variable (since it is going to join the set of basic variables).

Clearly, if we increase the value of x_e (from the current value of 0 in τ) then one of the basic variables is going to vanish (i.e., become zero). Let x_l be this basic variable. We increase the value of x_e (the ***entering*** variable) till x_l (the ***leaving*** variable) becomes zero.

Setting all nonbasic variables to zero, and letting x_e grow, implies that $x_i = b_i - a_{ie} x_e$, for all $i \in B$.

All those variables must be non-negative, and thus we require that $\forall i \in B$ it holds $x_i = b_i - a_{ie} x_e \geq 0$. Namely, $x_e \leq (b_i / a_{ie})$ or alternatively, $\dfrac{1}{x_e} \geq \dfrac{a_{ie}}{b_i}$. Namely, $\dfrac{1}{x_e} \geq \max\limits_{i \in B} \dfrac{a_{ie}}{b_i}$ and, the largest value of x_e which is still feasible is

$$U = \left(\max_{i \in B} \frac{a_{ie}}{b_i} \right)^{-1}.$$

We pick l (the index of the leaving variable) from the set all basic variables that vanish to zero when $x_e = U$. Namely, l is from the set

$$\left\{ j \;\middle|\; \frac{a_{je}}{b_j} = U \quad \text{where } j \in B \right\}.$$

Now, we know x_e and x_l. We rewrite the equation for x_l in the LP so that it has x_e on the left size. Formally, we do

$$x_l = b_l - \sum_{j \in N} a_{lj} x_j \qquad \Rightarrow \qquad x_e = \frac{b_l}{a_{le}} - \sum_{j \in N \cup \{l\}} \frac{a_{lj}}{a_{le}} x_j, \qquad \text{where } a_{ll} = 1.$$

We need to remove all the appearances on the right side of the LP of x_e. This can be done by substituting x_e into the other equalities, using the above equality. Alternatively, we do beforehand Gaussian elimination, to remove any appearance of x_e on the right side of the equalities in the LP (and also from the objective function) replaced by appearances of x_l on the left side, which we then transfer to the right side.

In the end of this process, we have a new *equivalent* LP where the basic variables are $B' = (B \setminus \{l\}) \cup \{e\}$ and the non-basic variables are $N' = (N \setminus \{e\}) \cup \{l\}$.

In end of this ***pivoting*** stage the LP objective function value had increased, and as such, we made progress. Note, that the linear system is completely defined by which variables are basic, and which are non-basic. Furthermore, pivoting never returns to a combination (of basic/non-basic variable) that was already visited. Indeed, we improve the value of the objective function in each pivoting stage. Thus, we can do at most

$$\binom{n+m}{n} \leq \left(\frac{n+m}{n} \cdot e\right)^n$$

pivoting steps. And this is close to tight in the worst case (there are examples where 2^n pivoting steps are needed).

Each pivoting step takes polynomial time in n and m. Thus, the overall running time of **Simplex** is exponential in the worst case. However, in practice, **Simplex** is extremely fast.

21.2.1 Degeneracies

If you inspect carefully the **Simplex** algorithm, you would notice that it might get stuck if one of the b_is is zero. This corresponds to a case where $> m$ hyperplanes passes through the same point. This might cause the effect that you might not be able to make any progress at all in pivoting.

There are several solutions, the simplest one is to add tiny random noise to each coefficient. You can even do this symbolically. Intuitively, the degeneracy, being a local phenomena on the polytope disappears with high probability.

The larger danger, is that you would get into cycling; namely, a sequence of pivoting operations that do not improve the objective function, and the bases you get are cyclic (i.e., infinite loop).

There is a simple scheme based on using the symbolic perturbation, that avoids cycling, by carefully choosing what is the leaving variable. This is described in detail in Section 21.6.

There is an alternative approach, called ***Bland's rule***, which always choose the lowest index variable for entering and leaving out of the possible candidates. We will not prove the correctness of this approach here.

21.2.2 Correctness of linear programming

Definition 21.2.1. A solution to an LP is a ***basic solution*** if it the result of setting all the nonbasic variables to zero.

Note that the **Simplex** algorithm deals only with basic solutions. In particular we get the following.

Theorem 21.2.2 (Fundamental theorem of Linear Programming.). *For an arbitrary linear program, the following statements are true:*

(A) If there is no optimal solution, the problem is either infeasible or unbounded.
(B) If a feasible solution exists, then a basic feasible solution exists.
(C) If an optimal solution exists, then a basic optimal solution exists.

Proof: Proof is constructive by running the simplex algorithm. ■

21.2.3 On the ellipsoid method and interior point methods

The **Simplex** algorithm has exponential running time in the worst case.

The ellipsoid method is *weakly* polynomial (namely, it is polynomial in the number of bits of the input). Khachian in 1979 came up with it. It turned out to be completely useless in practice.

In 1984, Karmakar came up with a different method, called the *interior-point method* which is also weakly polynomial. However, it turned out to be quite useful in practice, resulting in an arm race between the interior-point method and the simplex method.

The question of whether there is a *strongly* polynomial time algorithm for linear programming, is one of the major open questions in computer science.

21.3 Duality and Linear Programming

Every linear program L has a ***dual linear program*** L'. Solving the dual problem is essentially equivalent to solving the ***primal linear program*** (i.e., the original) LP.

21.3.1 Duality by Example

Consider the linear program L depicted on the right (Figure 21.2). Note, that any feasible solution, gives us a lower bound on the maximal value of the target function, denoted by η. In particular, the solution $x_1 = 1, x_2 = x_3 = 0$ is feasible, and implies $z = 4$ and thus $\eta \geq 4$.

$$\begin{array}{ll} \max & z = 4x_1 + x_2 + 3x_3 \\ \text{s.t.} & x_1 + 4x_2 \leq 1 \\ & 3x_1 - x_2 + x_3 \leq 3 \\ & x_1, x_2, x_3 \geq 0 \end{array}$$

Figure 21.2: The linear program L.

Similarly, $x_1 = x_2 = 0$, $x_3 = 3$ is feasible and implies that $\eta \geq z = 9$.

We might be wondering how close is this solution to the optimal solution? In particular, if this solution is very close to the optimal solution, we might be willing to stop and be satisfied with it.

Let us add the first inequality (multiplied by 2) to the second inequality (multiplied by 3). Namely, we add the two inequalities:

$$\begin{aligned} 2(\ x_1 + 4x_2 \qquad) &\leq 2(1) \\ +3(3x_1 - x_2 + x_3) &\leq 3(3). \end{aligned}$$

The resulting inequality is

$$11x_1 + 5x_2 + 3x_3 \leq 11. \tag{21.1}$$

Note, that this inequality must hold for any feasible solution of L. Now, the objective function is $z = 4x_1 + x_2 + 3x_3$ and x_1,x_2 and x_3 are all non-negative, and the inequality of Eq. (21.1) has larger coefficients that all the coefficients of the target function, for the corresponding variables. It thus follows, that for any feasible solution, we have

$$z = 4x_1 + x_2 + 3x_3 \leq 11x_1 + 5x_2 + 3x_3 \leq 11,$$

since all the variables are non-negative. As such, the optimal value of the LP L is somewhere between 9 and 11.

We can extend this argument. Let us multiply the first inequality by y_1 and second inequality by y_2 and add them up. We get:

$$\begin{array}{|ccccccc|} \hline y_1(x_1 & + & 4x_2 & & &) \leq & y_1(1) \\ + \, y_2(3x_1 & - & x_2 & + & x_3 &) \leq & y_2(3) \\ \hline (y_1 + 3y_2)x_1 & + & (4y_1 - y_2)x_2 & + & y_2x_3 & \leq & y_1 + 3y_2. \\ \hline \end{array} \tag{21.2}$$

Compare this to the target function $z = 4x_1 + x_2 + 3x_3$. If this expression is bigger than the target function in each variable, namely

$$\begin{aligned} 4 &\leq y_1 + 3y_2 \\ 1 &\leq 4y_1 - y_2 \\ 3 &\leq y_2, \end{aligned}$$

then, $z = 4x_1 + x_2 + 3x_3 \leq (y_1 + 3y_2)x_1 + (4y_1 - y_2)x_2 + y_2x_3 \leq y_1 + 3y_2$, the last step follows by Eq. (21.2).

Thus, if we want the best upper bound on η (the maximal value of z) then we want to solve the LP $\widehat{L}$ depicted in Figure 21.3. This is the dual program to L and its optimal solution is an upper bound to the optimal solution for L.

$$\begin{array}{ll} \min & y_1 + 3y_2 \\ \text{s.t.} & y_1 + 3y_2 \geq 4 \\ & 4y_1 - y_2 \geq 1 \\ & y_2 \geq 3 \\ & y_1, y_2 \geq 0. \end{array}$$

Figure 21.3: The dual LP $\widehat{L}$. The primal LP is depicted in Figure 21.2.

21.3.2 The Dual Problem

Given a linear programming problem (i.e., ***primal problem***, seen in Figure 21.4 (a), its associated ***dual linear program*** is in Figure 21.4 (b). The standard form of the dual LP is depicted in Figure 21.4 (c). Interestingly, you can just compute the dual LP to the given dual LP. What you get back is the original LP. This is demonstrated in Figure 21.5.

We just proved the following result.

Lemma 21.3.1. *Let L be an LP, and let L' be its dual. Let L'' be the dual to L'. Then L and L'' are the same LP.*

$$\begin{aligned} \max \quad & \sum_{j=1}^{n} c_j x_j \\ \text{s.t.} \quad & \sum_{j=1}^{n} a_{ij} x_j \leq b_i, \\ & \qquad \text{for } i = 1, \ldots, m, \\ & x_j \geq 0, \\ & \qquad \text{for } j = 1, \ldots, n. \end{aligned}$$

(a) ***primal program***

$$\begin{aligned} \min \quad & \sum_{i=1}^{m} b_i y_i \\ \text{s.t.} \quad & \sum_{i=1}^{m} a_{ij} y_i \geq c_j, \\ & \qquad \text{for } j = 1, \ldots, n, \\ & y_i \geq 0, \\ & \qquad \text{for } i = 1, \ldots, m. \end{aligned}$$

(b) ***dual program***

$$\begin{aligned} \max \quad & \sum_{i=1}^{m} (-b_i) y_i \\ \text{s.t.} \quad & \sum_{i=1}^{m} (-a_{ij}) y_i \leq -c_j, \\ & \qquad \text{for } j = 1, \ldots, n, \\ & y_i \geq 0, \\ & \qquad \text{for} i = 1, \ldots, m. \end{aligned}$$

(c) dual program in standard form

Figure 21.4: Dual linear programs.

$$\begin{aligned} \max \quad & \sum_{i=1}^{m} (-b_i) y_i \\ \text{s.t.} \quad & \sum_{i=1}^{m} (-a_{ij}) y_i \leq -c_j, \\ & \qquad \text{for } j = 1, \ldots, n, \\ & y_i \geq 0, \\ & \qquad \text{for } i = 1, \ldots, m. \end{aligned}$$

(a) dual program

$$\begin{aligned} \min \quad & \sum_{j=1}^{n} -c_j x_j \\ \text{s.t.} \quad & \sum_{j=1}^{n} (-a_{ij}) x_j \geq -b_i, \\ & \qquad \text{for } i = 1, \ldots, m, \\ & x_j \geq 0, \\ & \qquad \text{for } j = 1, \ldots, n. \end{aligned}$$

(b) the dual program to the dual program

$$\begin{aligned} \max \quad & \sum_{j=1}^{n} c_j x_j \\ \text{s.t.} \quad & \sum_{j=1}^{n} a_{ij} x_j \leq b_i, \\ & \qquad \text{for } i = 1, \ldots, m, \\ & x_j \geq 0, \\ & \qquad \text{for } j = 1, \ldots, n. \end{aligned}$$

(c) ... which is the original LP.

Figure 21.5: The dual to the dual linear program. Computing the dual of (a) can be done mechanically by following Figure 21.4 (a) and (b). Note, that (c) is just a rewriting of (b).

21.3.3 The Weak Duality Theorem

Theorem 21.3.2. *If $(x_1, x_2, \ldots, x_n)$ is feasible for the primal LP and $(y_1, y_2, \ldots, y_m)$ is feasible for the dual LP, then*

$$\sum_j c_j x_j \leq \sum_i b_i y_i.$$

Namely, all the feasible solutions of the dual bound all the feasible solutions of the primal.

Proof: By substitution from the dual form, and since the two solutions are feasible, we know that

$$\sum_j c_j x_j \leq \sum_j \left(\sum_{i=1}^{m} y_i a_{ij} \right) x_j \leq \sum_i \left(\sum_j a_{ij} x_j \right) y_i \leq \sum_i b_i y_i \ .$$

■

Interestingly, if we apply the weak duality theorem on the dual program (namely, Figure 21.5 (a) and (b)), we get the inequality $\sum_{i=1}^{m}(-b_i)y_i \leq \sum_{j=1}^{n} -c_j x_j$, which is the original inequality in the weak duality theorem. Thus, the weak duality theorem does not imply the strong duality theorem which will be discussed next.

21.4 The strong duality theorem

The ***strong duality theorem*** states the following.

Theorem 21.4.1. *If the primal LP problem has an optimal solution $x^* = (x_1^*, \ldots, x_n^*)$ then the dual also has an optimal solution, $y^* = (y_1^*, \ldots, y_m^*)$, such that*

$$\sum_j c_j x_j^* = \sum_i b_i y_i^*.$$

Its proof is somewhat tedious and not very insightful, the basic idea to prove this theorem is to run the simplex algorithm simultaneously on both the primal and the dual LP making steps in sync. When the two stop, they must be equal of they are feasible. We omit the tedious proof.

21.5 Some duality examples

21.5.1 Shortest path

You are given a graph $\mathsf{G} = (\mathsf{V}, \mathsf{E})$, with source s and target t. We have weights $\omega(u, v)$ on each edge $(u \to v) \in \mathsf{E}$, and we are interested in the shortest path in this graph from s to t. To simplify the exposition assume that there are no incoming edges in s and no edges leave t. To this end, let d_x be a variable that is the distance between s and x, for any $x \in \mathsf{V}$. Clearly, we must have for any edge $(u \to v) \in \mathsf{E}$, that $d_u + \omega(u, v) \geq d_v$. We also know that $d_{\mathsf{s}} = 0$. Clearly, a trivial solution to this constraints is to set all the variables to zero. So, we are trying to find the assignment that maximizes d_{t}, such that all the constraints are filled. As such, the LP for computing the shortest path from s to t is the following LP.

$$\begin{aligned}
\max \quad & d_{\mathsf{t}} && \\
\text{s.t.} \quad & d_{\mathsf{s}} \leq 0 && \\
& d_u + \omega(u, v) \geq d_v && \forall\, (u \to v) \in \mathsf{E}, \\
& d_x \geq 0 && \forall x \in \mathsf{V}.
\end{aligned}$$

Equivalently, we get

$$\begin{aligned}
\max \quad & d_{\mathsf{t}} \\
\text{s.t.} \quad & d_{\mathsf{s}} \leq 0 \\
& d_v - d_u \leq \omega(u,v) && \forall\, (u \to v) \in \mathsf{E}, \\
& d_x \geq 0 && \forall x \in \mathsf{V}.
\end{aligned}$$

Let use compute the dual. To this end, let y_{uv} be the dual variable for the edge $(u \to v)$, and let y_{s} be the dual variable for the $d_{\mathsf{s}} \leq 0$ inequality. We get the following dual LP.

$$\begin{aligned}
\min \quad & \sum_{(u\to v)\in \mathsf{E}} y_{uv}\omega(u,v) \\
\text{s.t.} \quad & y_{\mathsf{s}} - \sum_{(\mathsf{s}\to u)\in \mathsf{E}} y_{\mathsf{s}u} \geq 0 && && (*) \\
& \sum_{(u\to x)\in \mathsf{E}} y_{ux} - \sum_{(x\to v)\in \mathsf{E}} y_{xv} \geq 0 && \forall x \in \mathsf{V}\setminus\{\mathsf{s},\mathsf{t}\} && (**) \\
& \sum_{(u\to \mathsf{t})\in \mathsf{E}} y_{u\mathsf{t}} \geq 1 && && (***) \\
& y_{uv} \geq 0 && \forall\, (u \to v) \in \mathsf{E}, \\
& y_{\mathsf{s}} \geq 0.
\end{aligned}$$

Look carefully at this LP. The trick is to think about the y_{uv} as a flow on the edge y_{uv}. (Also, we assume here that the weights are positive.) Then, this LP is the min cost flow of sending one unit of flow from the source s to t. Indeed, if the weights are positive, then (**) can be assumed to be hold with equality in the optimal solution, and this is conservation of flow. Equation (***) implies that one unit of flow arrives to the sink t. Finally, (*) implies that at least y_{s} units of flow leaves the source. The remaining of the LP implies that $y_{\mathsf{s}} \geq 1$. Of course, this min-cost flow version, is without capacities on the edges.

21.5.2 Set Cover and Packing

Consider an instance of Set Cover with $(\mathsf{S}, \mathcal{F})$, where $\mathsf{S} = \{u_1, \ldots, u_n\}$ and $\mathcal{F} = \{F_1, \ldots, F_m\}$, where $F_i \subseteq \mathsf{S}$. The natural LP to solve this problem is

$$\begin{aligned}
\min \quad & \sum_{F_j \in \mathcal{F}} x_j \\
\text{s.t.} \quad & \sum_{\substack{F_j\in\mathcal{F},\\ u_i\in F_j}} x_j \geq 1 && \forall u_i \in \mathsf{S}, \\
& x_j \geq 0 && \forall F_j \in \mathcal{F}.
\end{aligned}$$

The dual LP is

$$\begin{aligned}
\max \quad & \sum_{u_i \in \mathsf{S}} y_i & & \\
\text{s.t.} \quad & \sum_{u_i \in F_j} y_i \leq 1 & & \forall F_j \in \mathcal{F}, \\
& y_i \geq 0 & & \forall u_i \in \mathsf{S}.
\end{aligned}$$

This is a ***packing*** LP. We are trying to pick as many vertices as possible, such that no set has more than one vertex we pick. If the sets in $\mathcal{F}$ are pairs (i.e., the set system is a graph), then the problem is known as ***edge cover***, and the dual problem is the familiar ***independent set*** problem. Of course, these are all the fractional versions – getting an integral solution for these problems is completely non-trivial, and in all these cases is impossible in polynomial time since the problems are NP-Complete.

As an exercise, write the LP for Set Cover for the case where every set has a price associated with it, and you are trying to minimize the total cost of the cover.

21.5.3 Network flow

(We do the following in excruciating details – hopefully its make the presentation clearer.)

Let assume we are given an instance of network flow G, with source s, and sink t. As usual, let us assume there are no incoming edges into the source, no outgoing edges from the sink, and the two are not connected by an edge. The LP for this network flow is the following.

$$\begin{aligned}
\max \quad & \sum_{(\mathsf{s} \to v) \in \mathsf{E}} x_{\mathsf{s} \to v} & & \\
& x_{u \to v} \leq \mathsf{c}(u \to v) & & \forall\, (u \to v) \in \mathsf{E} \\
& \sum_{(u \to v) \in \mathsf{E}} x_{u \to v} - \sum_{(v \to w) \in \mathsf{E}} x_{v \to w} \leq 0 & & \forall v \in \mathsf{V} \setminus \{\mathsf{s}, \mathsf{t}\} \\
& - \sum_{(u \to v) \in \mathsf{E}} x_{u \to v} + \sum_{(v \to w) \in \mathsf{E}} x_{v \to w} \leq 0 & & \forall v \in \mathsf{V} \setminus \{\mathsf{s}, \mathsf{t}\} \\
& 0 \leq x_{u \to v} & & \forall\, (u \to v) \in \mathsf{E}.
\end{aligned}$$

To perform the duality transform, we define a dual variable for each inequality. We get the

following dual LP:

$$\begin{array}{lll}
\max & \sum_{(\mathsf{s}\to v)\in\mathsf{E}} x_{\mathsf{s}\to v} & \\
& x_{u\to v} \le \mathsf{c}(u\to v) & * \quad y_{u\to v} \qquad \forall\,(u\to v)\in\mathsf{E} \\
& \sum_{(u\to v)\in\mathsf{E}} x_{u\to v} - \sum_{(v\to w)\in\mathsf{E}} x_{v\to w} \le 0 & * \quad y_v \qquad \forall v\in\mathsf{V}\setminus\{\mathsf{s},\mathsf{t}\} \\
& -\sum_{(u\to v)\in\mathsf{E}} x_{u\to v} + \sum_{(v\to w)\in\mathsf{E}} x_{v\to w} \le 0 & * \quad y'_v \qquad \forall v\in\mathsf{V}\setminus\{\mathsf{s},\mathsf{t}\} \\
& 0 \le x_{u\to v} & \forall\,(u\to v)\in\mathsf{E}.
\end{array}$$

Now, we generate the inequalities on the coefficients of the variables of the target functions. We need to carefully account for the edges, and we observe that there are three kinds of edges: source edges, regular edges, and sink edges. Doing the duality transformation carefully, we get the following:

$$\begin{array}{lr}
\min \sum_{(u\to v)\in\mathsf{E}} \mathsf{c}(u\to v)\, y_{u\to v} & \\
1 \le y_{\mathsf{s}\to v} + y_v - y'_v & \forall\,(\mathsf{s}\to v)\in\mathsf{E} \\
0 \le y_{u\to v} + y_v - y'_v - y_u + y'_u & \forall\,(u\to v)\in\mathsf{E}(\mathsf{G}\setminus\{\mathsf{s},\mathsf{t}\}) \\
0 \le y_{v\to\mathsf{t}} - y_v + y'_v & \forall\,(v\to\mathsf{t})\in\mathsf{E} \\
y_{u\to v} \ge 0 & \forall\,(u\to v)\in\mathsf{E} \\
y_v \ge 0 & \forall v\in\mathsf{V} \\
y'_v \ge 0 & \forall v\in\mathsf{V}
\end{array}$$

To understand what is going on, let us rewrite the LP, introducing the variable $d_v = y_v - y'_v$, for each $v \in V$[①]. We get the following modified LP:

$$\begin{array}{lr}
\min \sum_{(u\to v)\in\mathsf{E}} \mathsf{c}(u\to v)\, y_{u\to v} & \\
1 \le y_{\mathsf{s}\to v} + d_v & \forall\,(\mathsf{s}\to v)\in\mathsf{E} \\
0 \le y_{u\to v} + d_v - d_u & \forall\,(u\to v)\in\mathsf{E}(\mathsf{G}\setminus\{\mathsf{s},\mathsf{t}\}) \\
0 \le y_{v\to\mathsf{t}} - d_v & \forall\,(v\to\mathsf{t})\in\mathsf{E} \\
y_{u\to v} \ge 0 & \forall\,(u\to v)\in\mathsf{E}
\end{array}$$

Adding the two variables for t and s, and setting their values as follows $d_{\mathsf{t}} = 0$ and $d_{\mathsf{s}} = 1$, we get the following LP:

[①]We could have done this directly, treating the two inequalities as equality, and multiplying it by a single variable that can be both positive and negative – however, it is useful to see why this is correct at least once.

$$\begin{aligned}
\min \sum_{(u\to v)\in \mathsf{E}} & \mathsf{c}(u\to v)\, y_{u\to v} & \\
& 0 \le y_{\mathsf{s}\to v} + d_v - d_{\mathsf{s}} && \forall\, (\mathsf{s}\to v) \in \mathsf{E} \\
& 0 \le y_{u\to v} + d_v - d_u && \forall\, (u\to v) \in \mathsf{E}(\mathsf{G}\setminus\{\mathsf{s},\mathsf{t}\}) \\
& 0 \le y_{v\to \mathsf{t}} + d_{\mathsf{t}} - d_v && \forall\, (v\to \mathsf{t}) \in \mathsf{E} \\
& y_{u\to v} \ge 0 && \forall\, (u\to v) \in \mathsf{E} \\
& d_{\mathsf{s}} = 1, \qquad d_{\mathsf{t}} = 0 &&
\end{aligned}$$

Which simplifies to the following LP:

$$\begin{aligned}
\min \sum_{(u\to v)\in \mathsf{E}} & \mathsf{c}(u\to v)\, y_{u\to v} & \\
& d_u - d_v \le y_{u\to v} && \forall\, (u\to v) \in \mathsf{E} \\
& y_{u\to v} \ge 0 && \forall\, (u\to v) \in \mathsf{E} \\
& d_{\mathsf{s}} = 1, \qquad d_{\mathsf{t}} = 0. &&
\end{aligned}$$

The above LP can be interpreted as follows: We are assigning weights to the edges (i.e., $y_{(u\to v)}$). Given such an assignment, it is easy to verify that setting d_u (for all u) to be the shortest path distance under this weighting to the sink t, complies with all inequalities, the assignment $d_{\mathsf{s}} = 1$ implies that we require that the shortest path distance from the source to the sink has length exactly one.

We are next going to argue that the optimal solution to this LP is a min-cut. Lets us first start with the other direction, given a cut (S, T) with $\mathsf{s} \in S$ and $\mathsf{t} \in T$, observe that setting

$$\begin{aligned}
d_u &= 1 && \forall u \in S \\
d_u &= 0 && \forall u \in T \\
y_{u\to v} &= 1 && \forall\, (u\to v) \in (S, T) \\
y_{u\to v} &= 0 && \forall\, (u\to v) \in \mathsf{E} \setminus (S, T)
\end{aligned}$$

is a valid solution for the LP.

As for the other direction, consider the optimal solution for the LP, and let its target function value be

$$\alpha^* = \sum_{(u\to v)\in \mathsf{E}} \mathsf{c}(u\to v)\, y^*_{u\to v}$$

(we use (*) notation to the denote the values of the variables in the optimal LP solution). Consider generating a cut as follows, we pick a random value uniformly in $z \in [0, 1]$, and we set $S = \left\{ u \;\middle|\; d^*_u \ge z \right\}$ and $T = \left\{ u \;\middle|\; d^*_u < z \right\}$. This is a valid cut, as $\mathsf{s} \in S$ (as $d^*_{\mathsf{s}} = 1$) and

$\mathsf{t} \in T$ (as $d^*_{\mathsf{t}} = 0$). Furthermore, an edge $(u \to v)$ is in the cut, only if $d^*_u > d^*_v$ (otherwise, it is not possible to cut this edge using this approach).

In particular, the probability of $u \in S$ and $v \in T$, is exactly $d^*_u - d^*_v$! Indeed, it is the probability that z falls inside the interval $[d^*_v, d^*_u]$. As such, $(u \to v)$ is in the cut with probability $d^*_u - d^*_v$ (again, only if $d^*_u > d^*_v$), which is bounded by $y^*_{(u\to v)}$ (by the inequality $d_u - d_v \leq y_{u\to v}$ in the LP).

So, let $X_{u\to v}$ be an indicator variable which is one if the edge is in the generated cu. We just argued that $\mathbf{E}[X_{u\to v}] = \mathbf{Pr}[X_{u\to v} = 1] \leq y^*_{(u\to v)}$. We thus have that the expected cost of this random cut is

$$\mathbf{E}\left[\sum_{(u\to v)\in \mathsf{E}} X_{u\to v}\, \mathsf{c}(u \to v)\right] = \sum_{(u\to v)\in \mathsf{E}} \mathsf{c}(u \to v)\, \mathbf{E}[X_{u\to v}\,] \leq \sum_{(u\to v)\in \mathsf{E}} \mathsf{c}(u \to v) y^*_{u\to v} = \alpha^*.$$

That is, the expected cost of a random cut here is at most the value of the LP optimal solution. In particular, there must be a cut that has cost at most α^*, see Remark 21.5.2 below. However, we argued that α^* is no larger than the cost of any cut. We conclude that α^* is the cost of the min cut.

We are now ready for the kill, the optimal value of the original max-flow LP; that is, the max-flow (which is a finite number because all the capacities are bounded numbers), is equal by the strong duality theorem, to the optimal value of the dual LP (i.e., α^*). We just argued that α^* is the cost of the min cut in the given network. As such, we proved the following.

Lemma 21.5.1. *The Min-Cut Max-Flow Theorem follows from the strong duality Theorem for Linear Programming.*

Remark 21.5.2. In the above, we used the following "trivial" but powerful argument. Assume you have a random variable Z, and consider its expectation $\mu = \mathbf{E}[Z]$. The expectation μ is the weighted average value of the values the random variable Z might have, and in particular, there must be a value z that might be assigned to Z (with non-zero probability), such that $z \leq \mu$. Putting it differently, the weighted average of a set of numbers is bigger (formally, no smaller) than some number in this set.

This argument is one of the standard tools in the ***probabilistic method*** – a technique to prove the existence of entities by considering expectations and probabilities.

21.6 Solving LPs without ever getting into a loop - symbolic perturbations

21.6.1 The problem and the basic idea

Consider the following LP:

$$\begin{aligned} \max \quad & z = v + \sum_{j \in N} c_j x_j, \\ \text{s.t.} \quad & x_i = b_i - \sum_{j \in N} a_{ij} x_j \qquad \text{for } i = 1, \ldots, n, \\ & x_i \geq 0, \qquad \forall i = 1, \ldots, n+m. \end{aligned}$$

(Here $B = \{1, \ldots, n\}$ and $N = \{n+1, \ldots, n+m\}$.) The **Simplex** algorithm might get stuck in a loop of pivoting steps, if one of the constants b_i becomes zero during the algorithm execution. To avoid this, we are going to add tiny infinitesimals to all the equations. Specifically, let $\varepsilon > 0$ be an arbitrarily small constant, and let $\varepsilon_i = \varepsilon^i$. The quantities $\varepsilon_1, \ldots, \varepsilon_n$ are infinitesimals of different scales. We slightly perturb the above LP by adding them to each equation. We get the following modified LP:

$$\begin{aligned} \max \quad & z = v + \sum_{j \in N} c_j x_j, \\ \text{s.t.} \quad & x_i = \varepsilon_i + b_i - \sum_{j \in N} a_{ij} x_j \qquad \text{for } i = 1, \ldots, n, \\ & x_i \geq 0, \qquad \forall i = 1, \ldots, n+m. \end{aligned}$$

Importantly, any feasible solution to the original LP translates into a valid solution of this LP (we made things better by adding these symbolic constants).

The rule of the game is now that we treat $\varepsilon_1, \ldots, \varepsilon_n$ as symbolic constants. Of course, when we do pivoting, we need to be able to compare two numbers and decide which one is bigger. Formally, given two numbers

$$\alpha = \alpha_0 + \alpha_1 \varepsilon_1 + \cdots + \alpha_n \varepsilon_n \qquad \text{and} \qquad \beta = \beta_0 + \beta_1 \varepsilon_1 + \cdots + \beta_n \varepsilon_n, \tag{21.3}$$

then $\alpha > \beta$ if and only if there is an index i such that $\alpha_0 = \beta_0, \alpha_1 = \beta_1, \ldots, \alpha_{i-1} = \beta_{i-1}$ and $\alpha_i > \beta_i$. That is, $\alpha > \beta$ if the vector $(\alpha_0, \alpha_1, \ldots, \alpha_n)$ is ***lexicographically*** larger than $(\beta_0, \beta_1, \ldots, \beta_n)$.

Significantly, but not obviously at this stage, the simplex algorithm would never divide an ε_i by an ε_j, so we are good to go – we can perform all the needed arithmetic operations of the **Simplex** using these symbolic constants, and we claim that now the constant term (which is a number of the form of Eq. (21.3)) is now never zero. This implies immediately that the **Simplex** algorithm always makes progress, and it does terminates. We still need to address the two issues:

(A) How are the symbolic perturbations are updated at each iteration?

(B) Why the constants can never be zero?

21.6.2 Pivoting as a Gauss elimination step

Consider the LP equations

$$x_i + \sum_{j \in N} a_{ij} x_j = b_i, \quad \text{for } i \in B,$$

where $B = \{1, \ldots, n\}$ and $N = \{n+1, \ldots, n+m\}$. We can write these equations down in matrix form

x_1	x_2	$\ldots$	x_n	x_{n+1}	x_{n+2}	$\ldots$	x_j	$\ldots$	x_{n+m}	const
1	0	$\ldots$	0	$a_{1,n+1}$	$a_{1,n+2}$	$\ldots$	$a_{1,j}$	$\ldots$	$a_{1,n+m}$	b_1
0	1	$\ldots$	0	$a_{2,n+1}$	$a_{2,n+2}$	$\ldots$	$a_{2,j}$	$\ldots$	$a_{2,n+m}$	b_2
$\vdots$	$\vdots$	$\vdots$	$\vdots$	$\vdots$	$\vdots$	$\vdots$	$\vdots$	$\vdots$	$\vdots$	$\vdots$
0	$0 \ldots 0$	1	$0 \ldots 0$	$a_{k,n+1}$	$a_{k,n+2}$	$\ldots$	$a_{k,j}$	$\vdots$	$a_{k,n+m}$	b_k
$\vdots$	$\vdots$	$\vdots$	$\vdots$	$\vdots$	$\vdots$	$\vdots$	$\vdots$	$\vdots$	$\vdots$	$\vdots$
0	$\ldots$	0	1	$a_{n,n+1}$	$a_{n,n+2}$	$\ldots$	$a_{n,j}$	$\ldots$	$a_{n,n+m}$	b_n

Assume that now we do a pivoting step with x_j entering the basic variables, and x_k leaving. To this end, let us multiply the kth row (i.e., the kth equation) by $1/a_{k,j}$, this result in the kth row having 1 instead of $a_{k,j}$. Let this resulting row be denoted by $\mathbf{r}$. Now, add $a_{i,j}\mathbf{r}$ to the ith row of the matrix, for all i. Clearly, in the resulting row/equation, the coefficient of x_j is going to be zero, in all rows except the kth one, where it is 1. Note, that on the matrix on the left side, all the columns are the same, except for the kth column, which might now have various numbers in this column. The final step is to exchange the kth column on the left, with the jth column on the right. And that is one pivoting step, when working on the LP using a matrix. It is very similar to one step of the Gauss elimination in matrices, if you are familiar with that.

21.6.2.1 Back to the perturbation scheme

We now add a new matrix to the above representations on the right side, that keeps track of the εs. This looks initially as follows.

x_1	x_2	$\ldots$	x_n	x_{n+1}	$\ldots$	x_j	$\ldots$	x_{n+m}	const	ε_1	ε_2	$\ldots$	ε_n
1	0	$\ldots$	0	$a_{1,n+1}$	$\ldots$	$a_{1,j}$	$\ldots$	$a_{1,n+m}$	b_1	1	0	$\ldots$	0
0	1	$\ldots$	0	$a_{2,n+1}$	$\ldots$	$a_{2,j}$	$\ldots$	$a_{2,n+m}$	b_2	0	1	$\ldots$	0
$\vdots$	$\vdots$	$\vdots$	$\vdots$	$\vdots$	$\vdots$	$\vdots$	$\vdots$	$\vdots$	$\vdots$	$\vdots$	$\vdots$	$\vdots$	$\vdots$
0	$0 \ldots 0$	1	$0 \ldots 0$	$a_{k,n+1}$	$\ldots$	$a_{k,j}$	$\vdots$	$a_{k,n+m}$	$\vdots$	$\vdots$	$\vdots$	$\vdots$	$\vdots$
$\vdots$	$\vdots$	$\vdots$	$\vdots$	$\vdots$	$\vdots$	$\vdots$	$\vdots$	$\vdots$	$\vdots$	$\vdots$	$\vdots$	$\vdots$	$\vdots$
0	$\ldots$	0	1	$a_{n,n+1}$	$\ldots$	$a_{n,j}$	$\ldots$	$a_{n,n+m}$	b_n	0	0	$\ldots$	1

Now, we run the algorithm as described above, using the εs to resolve which variables are entering and leaving. The critical observation is that throughout the algorithm execution we

are adding rows, and multiplying them by non-zero constants. The matrix on the right has initially full rank, and throughout the execution of the algorithm its rank remains the same (because the linear operation we do on the rows can not change the rank of the matrix). In particular, it is impossible that a row on the right side of the matrix is all zero, or equal to another row, or equal to another row if multiplied by a constant. Namely, the symbolic constant encoded by the εs as we run the **Simplex** algorithm can never be zero. And furthermore, these constants are never equal for two different equations. We conclude that the **Simplex** algorithm now always make progress in each pivoting step.

21.6.2.2 The overall algorithm

We run the **Simplex** algorithm with the above described symbolic perturbation. The final stroke is that each basic variable x_i in the computed solution now equal to a number of the form $x_i = \alpha_0 + \sum_i \alpha_i \varepsilon_i$. We interpret this as $x_i = \alpha_0$, by setting all the εs to be zero.

Chapter 22

Approximation Algorithms using Linear Programming

22.1 Weighted vertex cover

Consider the Weighted Vertex Cover problem. Here, we have a graph $\mathsf{G} = (\mathsf{V}, \mathsf{E})$, and each vertex $\mathsf{v} \in \mathsf{V}$ has an associated cost c_v. We would like to compute a vertex cover of minimum cost – a subset of the vertices of G with minimum total cost so that each edge has at least one of its endpoints in the cover. This problem is (of course) NP-Hard, since the decision problem where all the weights are 1, is the Vertex Cover problem, which we had shown to be NPC.

Let us first state this optimization problem is an integer programming. Indeed, for any $\mathsf{v} \in \mathsf{V}$, let define a variable x_v which is 1 if we decide to pick v to the vertex cover, and zero otherwise. The restriction that x_v is either 0 or 1, is written formally as $x_\mathsf{v} \in \{0, 1\}$. Next, its required that every edge $\mathsf{vu} \in \mathsf{E}$ is covered. Namely, we require that $x_\mathsf{v} \vee x_\mathsf{u}$ to be TRUE. For reasons that would be come clearer shortly, we prefer to write this condition as a linear inequality; namely, we require that $x_\mathsf{v} + x_\mathsf{u} \geq 1$. Finally, we would like to minimize the total cost of the vertices we pick for the cover; namely, we would like to minimize $\sum_{\mathsf{v} \in \mathsf{V}} x_\mathsf{v} \mathsf{c}_\mathsf{v}$. Putting it together, we get the following integer programming instance:

$$\begin{aligned} \min \quad & \sum_{\mathsf{v} \in \mathsf{V}} \mathsf{c}_\mathsf{v} x_\mathsf{v}, & & \\ \text{such that} \quad & x_\mathsf{v} \in \{0, 1\} & \forall \mathsf{v} \in \mathsf{V} & \qquad (22.1) \\ & x_\mathsf{v} + x_\mathsf{u} \geq 1 & \forall \mathsf{vu} \in \mathsf{E}. & \end{aligned}$$

Naturally, solving this integer programming efficiently is NP-Hard, so instead let us try to relax this optimization problem to be a LP (which we can solve efficiently, at least in practice[①]). To do this, we need to relax the integer program. We will do it by allowing the variables x_v to get real values between 0 and 1. This is done by replacing the condition that

[①] And also in theory if the costs are integers, using more advanced algorithms than the **Simplex** algorithm.

$x_{\mathsf{v}} \in \{0,1\}$ by the constraint $0 \leq x_{\mathsf{v}} \leq 1$. The resulting LP is

$$\begin{aligned} \min \quad & \sum_{\mathsf{v} \in \mathsf{V}} \mathsf{c}_{\mathsf{v}} x_{\mathsf{v}}, & & \\ \text{such that} \quad & 0 \leq x_{\mathsf{v}} & \forall \mathsf{v} \in \mathsf{V}, & \\ & x_{\mathsf{v}} \leq 1 & \forall \mathsf{v} \in \mathsf{V}, & (22.2) \\ & x_{\mathsf{v}} + x_{\mathsf{u}} \geq 1 & \forall \mathsf{vu} \in \mathsf{E}. & \end{aligned}$$

So, consider the optimal solution to this LP, assigning value $\widehat{x_{\mathsf{v}}}$ to the variable X_{v}, for all $\mathsf{v} \in \mathsf{V}$. As such, the optimal value of the LP solution is

$$\widehat{\alpha} = \sum_{\mathsf{v} \in \mathsf{V}} \mathsf{c}_{\mathsf{v}} \widehat{x_{\mathsf{v}}}.$$

Similarly, let the optimal integer solution to integer program (IP) Eq. (22.1) denoted by x^I_{v}, for all $\mathsf{v} \in \mathsf{V}$ and α^I, respectively. Note, that any feasible solution for the IP of Eq. (22.1), is a feasible solution for the LP of Eq. (22.2). As such, we must have that

$$\widehat{\alpha} \leq \alpha^I,$$

where α^I is the value of the optimal solution.

So, what happened? We solved the relaxed optimization problem, and got a fractional solution (i.e., values of $\widehat{x_{\mathsf{v}}}$ can be fractions). On the other hand, the cost of this fractional solution is better than the optimal cost. So, the natural question is how to turn this fractional solution into a (valid!) integer solution. This process is known as ***rounding***.

To this end, it is beneficial to consider a vertex v and its fractional value $\widehat{x_{\mathsf{v}}}$. If $\widehat{x_{\mathsf{v}}} = 1$ then we definitely want to put it into our solution. If $\widehat{x_{\mathsf{v}}} = 0$ then the LP consider this vertex to be useless, and we really do not want to use it. Similarly, if $\widehat{x_{\mathsf{v}}} = 0.9$, then the LP considers this vertex to be very useful (0.9 useful to be precise, whatever this "means"). Intuitively, since the LP puts its money where its belief is (i.e., $\widehat{\alpha}$ value is a function of this "belief" generated by the LP), we should trust the LP values as a guidance to which vertices are useful and which are not. Which brings to forefront the following idea: Lets pick all the vertices that are about certain threshold of usefulness according to the LP solution. Formally, let

$$S = \left\{ \mathsf{v} \;\middle|\; \widehat{x_{\mathsf{v}}} \geq 1/2 \right\}.$$

We claim that S is a valid vertex cover, and its cost is low.

Indeed, let us verify that the solution is valid. We know that for any edge vu, it holds

$$\widehat{x_{\mathsf{v}}} + \widehat{x_{\mathsf{u}}} \geq 1.$$

Since $0 \leq \widehat{x_{\mathsf{v}}} \leq 1$ and $0 \leq \widehat{x_{\mathsf{u}}} \leq 1$, it must be either $\widehat{x_{\mathsf{v}}} \geq 1/2$ or $\widehat{x_{\mathsf{u}}} \geq 1/2$. Namely, either $\mathsf{v} \in S$ or $\mathsf{u} \in A$, or both of them are in S, implying that indeed S covers all the edges of G.

As for the cost of S, we have

$$\mathsf{c}_S = \sum_{\mathsf{v}\in S} \mathsf{c}_\mathsf{v} = \sum_{\mathsf{v}\in S} 1\cdot \mathsf{c}_\mathsf{v} \leq \sum_{\mathsf{v}\in S} 2\widehat{x_\mathsf{v}}\cdot \mathsf{c}_\mathsf{v} \leq 2\sum_{\mathsf{v}\in \mathsf{V}} \widehat{x_\mathsf{v}}\mathsf{c}_\mathsf{v} = 2\widehat{\alpha} \leq 2\alpha^I,$$

since $\widehat{x_\mathsf{v}} \geq 1/2$ as $\mathsf{v} \in S$.

Since α^I is the cost of the optimal solution, we got the following result.

Theorem 22.1.1. *The Weighted Vertex Cover problem can be 2-approximated by solving a single LP. Assuming computing the LP takes polynomial time, the resulting approximation algorithm takes polynomial time.*

What lessons can we take from this example? First, this example might be simple, but the resulting approximation algorithm is non-trivial. In particular, I am not aware of any other 2-approximation algorithm for the weighted problem that does not use LP. Secondly, the ***relaxation*** of an optimization problem into a LP provides us with a way to get some insight into the problem in hand. It also hints that in interpreting the values returned by the LP, and how to use them to do the rounding, we have to be creative.

22.2 Revisiting Set Cover

In this section, we are going to revisit the Set Cover problem, and provide an approximation algorithm for this problem. This approximation algorithm would not be better than the greedy algorithm we already saw, but it would expose us to a new technique that we would use shortly for a different problem.

Set Cover

Instance: $(S, \mathcal{F})$
S - a set of n elements
$\mathcal{F}$ - a family of subsets of S, s.t. $\bigcup_{X\in\mathcal{F}} X = S$.
Question: The set $\mathcal{X} \subseteq F$ such that $\mathcal{X}$ contains as few sets as possible, and $\mathcal{X}$ covers S.

As before, we will first define an IP for this problem. In the following IP, the second condition just states that any $s \in s$, must be covered by some set.

$$\begin{aligned}
\min \quad & \alpha = \sum_{U\in\mathcal{F}} x_U, & \\
\text{s.t.} \quad & x_U \in \{0,1\} & \forall U \in \mathcal{F}, \\
& \sum_{U\in\mathcal{F}, s\in U} x_U \geq 1 & \forall s \in S.
\end{aligned}$$

Next, we relax this IP into the following LP.

$$\begin{aligned}
\min \quad & \alpha = \sum_{U \in \mathcal{F}} x_U, & \\
& 0 \leq x_U \leq 1 & \forall U \in \mathcal{F}, \\
& \sum_{U \in \mathcal{F}, s \in U} x_U \geq 1 & \forall s \in S.
\end{aligned}$$

As before, consider the optimal solution to the LP: $\forall U \in \mathcal{F}$, $\widehat{x_U}$, and $\widehat{\alpha}$. Similarly, let the optimal solution to the IP (and thus for the problem) be: $\forall U \in \mathcal{F}$, x_U^I, and α^I. As before, we would try to use the LP solution to guide us in the rounding process. As before, if $\widehat{x_U}$ is close to 1 then we should pick U to the cover and if $\widehat{x_U}$ is close to 0 we should not. As such, its natural to pick $U \in \mathcal{F}$ into the cover by randomly choosing it into the cover with ***probability*** $\widehat{x_U}$. Consider the resulting family of sets $\mathcal{G}$. Let Z_S be an indicator variable which is one if $S \in \mathcal{G}$. We have that the cost of $\mathcal{G}$ is $\sum_{S \in \mathcal{F}} Z_S$, and the expected cost is

$$\mathbf{E}\Big[\text{cost of } \mathcal{G}\Big] = \mathbf{E}\left[\sum_{S \in \mathcal{F}} Z_S\right] = \sum_{S \in \mathcal{F}} \mathbf{E}\Big[Z_S\Big] = \sum_{S \in \mathcal{F}} \mathbf{Pr}\Big[S \in \mathcal{G}\Big] = \sum_{S \in \mathcal{F}} \widehat{x_S} = \widehat{\alpha} \leq \alpha^I. \tag{22.3}$$

As such, in expectation, $\mathcal{G}$ is not too expensive. The problem, of course, is that $\mathcal{G}$ might fail to cover some element $s \in S$. To this end, we repeat this algorithm

$$m = 10 \lceil \lg n \rceil = O(\log n)$$

times, where $n = |S|$. Let $\mathcal{G}_i$ be the random cover computed in the ith iteration, and let $\mathcal{H} = \cup_i \mathcal{G}_i$. We return $\mathcal{H}$ as the required cover.

The solution $\mathcal{H}$ covers S. For an element $s \in S$, we have that

$$\sum_{U \in \mathcal{F}, s \in U} \widehat{x_U} \geq 1, . \tag{22.4}$$

and consider the probability that s is not covered by $\mathcal{G}_i$, where $\mathcal{G}_i$ is the family computed in the ith iteration of the algorithm. Since deciding if the include each set U into $\mathcal{G}_i$ is done independently for each set, we have that the probability that s is not covered is

$$\begin{aligned}
\mathbf{Pr}\Big[s \text{ not covered by } \mathcal{G}_i\Big] &= \mathbf{Pr}\Big[\text{ none of } U \in \mathcal{F}, \text{ such that } s \in U \text{ were picked into } \mathcal{G}_i\Big] \\
&= \prod_{U \in \mathcal{F}, s \in U} \mathbf{Pr}\Big[U \text{ was not picked into } \mathcal{G}_i\Big] \\
&= \prod_{U \in \mathcal{F}, s \in U} (1 - \widehat{x_U}) \\
&\leq \prod_{U \in \mathcal{F}, s \in U} \exp(-\widehat{x_U}) = \exp\left(-\sum_{U \in \mathcal{F}, s \in U} \widehat{x_U}\right) \\
&\leq \exp(-1) \leq \frac{1}{2},
\end{aligned}$$

by Eq. (22.4). As such, the probability that s is not covered in all m iterations is at most

$$\left(\frac{1}{2}\right)^m < \frac{1}{n^{10}},$$

since $m = O(\log n)$. In particular, the probability that one of the n elements of S is not covered by $\mathcal{H}$ is at most $n(1/n^{10}) = 1/n^9$.

Cost. By Eq. (22.3), in each iteration the expected cost of the cover computed is at most the cost of the optimal solution (i.e., α^I). As such the expected cost of the solution computed is

$$\mathsf{c}_{\mathcal{H}} \leq \sum_i \mathsf{c}_{B_i} \leq m\alpha^I = O\Big(\alpha^I \log n\Big).$$

. Putting everything together, we get the following result.

Theorem 22.2.1. *By solving an LP one can get an $O(\log n)$-approximation to set cover by a randomized algorithm. The algorithm succeeds with high probability.*

22.3 Minimizing congestion

Let G be a graph with n vertices, and let π_i and σ_i be two paths with the same endpoints $\mathsf{v}_i, \mathsf{u}_i \in \mathsf{V}(\mathsf{G})$, for $i = 1, \ldots, t$. Imagine that we need to send one unit of flow from v_i to u_i, and we need to choose whether to use the path π_i or σ_i. We would like to do it in such a way that not edge in the graph is being used too much.

Definition 22.3.1. Given a set X of paths in a graph G, the ***congestion*** of X is the maximum number of paths in X that use the same edge.

Consider the following linear program:

$$\begin{aligned}
\min \quad & w && \\
\text{s.t.} \quad & x_i \geq 0 && i = 1, \ldots, t, \\
& x_i \leq 1 && i = 1, \ldots, t, \\
& \sum_{\mathsf{e} \in \pi_i} x_i + \sum_{\mathsf{e} \in \sigma_i} (1 - x_i) \leq w && \forall \mathsf{e} \in E.
\end{aligned}$$

Let $\widehat{x_i}$ be the value of x_i in the optimal solution of this LP, and let $\widehat{w}$ be the value of w in this solution. Clearly, the optimal congestion must be bigger than $\widehat{w}$.

Let X_i be a random variable which is one with probability $\widehat{x_i}$, and zero otherwise. If $X_i = 1$ then we use π to route from v_i to u_i, otherwise we use σ_i. Clearly, the congestion of e is

$$Y_{\mathsf{e}} = \sum_{\mathsf{e} \in \pi_i} X_i + \sum_{\mathsf{e} \in \sigma_i} (1 - X_i).$$

And in expectation

$$\begin{aligned}\alpha_{\mathsf{e}} = \mathbf{E}\Big[Y_{\mathsf{e}}\Big] &= \mathbf{E}\Big[\sum_{\mathsf{e}\in\pi_i} X_i + \sum_{\mathsf{e}\in\sigma_i}(1-X_i)\Big] = \sum_{\mathsf{e}\in\pi_i}\mathbf{E}\Big[X_i\Big] + \sum_{\mathsf{e}\in\sigma_i}\mathbf{E}\Big[(1-X_i)\Big] \\ &= \sum_{\mathsf{e}\in\pi_i}\widehat{x_i} + \sum_{\mathsf{e}\in\sigma_i}(1-\widehat{x_i}) \leq \widehat{w}.\end{aligned}$$

Using the Chernoff inequality, we have that

$$\mathbf{Pr}\Big[Y_{\mathsf{e}} \geq (1+\delta)\alpha_{\mathsf{e}}\Big] \leq \exp\left(-\frac{\alpha_{\mathsf{e}}\delta^2}{4}\right) \leq \exp\left(-\frac{\widehat{w}\delta^2}{4}\right).$$

(Note, that this works only if $\delta < 2e-1$, see Theorem 10.2.7). Let $\delta = \sqrt{\frac{400}{\widehat{w}}\ln t}$. We have that

$$\mathbf{Pr}\Big[Y_{\mathsf{e}} \geq (1+\delta)\alpha_{\mathsf{e}}\Big] \leq \exp\left(-\frac{\delta^2\widehat{w}}{4}\right) \leq \frac{1}{t^{100}},$$

which is very small. In particular, if $t \geq n^{1/50}$ then all the edges in the graph do not have congestion larger than $(1+\delta)\widehat{w}$.

To see what this result means, let us play with the numbers. Let assume that $t = n$, and $\widehat{w} \geq \sqrt{n}$. Then, the solution has congestion larger than the optimal solution by a factor of

$$1+\delta = 1+\sqrt{\frac{20}{\widehat{w}}\ln t} \leq 1+\frac{\sqrt{20\ln n}}{n^{1/4}},$$

which is of course extremely close to 1, if n is sufficiently larger.

Theorem 22.3.2. *Given a graph with n vertices, and t pairs of vertices, such that for every pair (s_i, t_i) there are two possible paths to connect s_i to t_i. Then one can choose for each pair which path to use, such that the most congested edge, would have at most $(1+\delta)\mathrm{opt}$, where opt is the congestion of the optimal solution, and $\delta = \sqrt{\frac{20}{\widehat{w}}\ln t}$.*

When the congestion is low. Assume that $\widehat{w}$ is a constant. In this case, we can get a better bound by using the Chernoff inequality in its more general form, see Theorem 10.2.7. Indeed, set $\delta = c\ln t/\ln\ln t$, where c is a constant. For $\mu = \alpha_{\mathsf{e}}$, we have that

$$\begin{aligned}\mathbf{Pr}\Big[Y_{\mathsf{e}} \geq (1+\delta)\mu\Big] &\leq \left(\frac{e^{\delta}}{(1+\delta)^{1+\delta}}\right)^{\mu} = \exp\left(\mu\Big(\delta - (1+\delta)\ln(1+\delta)\Big)\right) = \exp\left(-\mu c'\ln t\right) \\ &\leq \frac{1}{t^{O(1)}},\end{aligned}$$

where c' is a constant that depends on c and grows if c grows. We thus proved that if the optimal congestion is $O(1)$, then the algorithm outputs a solution with congestion $O(\log t/\log\log t)$, and this holds with high probability.

Part IX

Approximate Max Cut

Chapter 23

Approximate Max Cut

We had encountered in the previous lecture examples of using rounding techniques for approximating discrete optimization problems. So far, we had seen such techniques when the relaxed optimization problem is a linear program. Interestingly, it is currently known how to solve optimization problems that are considerably more general than linear programs. Specifically, one can solve ***convex programming***. Here the feasible region is convex. How to solve such an optimization problems is outside the scope of this course. It is however natural to ask what can be done if one assumes that one can solve such general continuous optimization problems exactly.

In the following, we show that (optimization problem) max cut can be relaxed into a weird continuous optimization problem. Furthermore, this semi-definite program can be solved exactly efficiently. Maybe more surprisingly, we can round this continuous solution and get an improved approximation.

23.1 Problem Statement

Given an undirected graph $\mathsf{G} = (\mathsf{V}, \mathsf{E})$ and nonnegative weights ω_{ij}, for all $ij \in \mathsf{E}$, the ***maximum cut problem*** (MAX CUT) is that of finding the set of vertices S that maximizes the weight of the edges in the cut $\left(S, \overline{S}\right)$; that is, the weight of the edges with one endpoint in S and the other in $\overline{S}$. For simplicity, we usually set $\omega_{ij} = O$ for $ij \notin \mathsf{E}$ and denote the weight of a cut $\left(S, \overline{S}\right)$ by $w\left(S, \overline{S}\right) = \sum_{i \in S, j \in \overline{S}} \omega_{ij}$.

This problem is NP-Complete, and hard to approximate within a certain constant.

Given a graph with vertex set $\mathsf{V} = \{1, \ldots, n\}$ and nonnegative weights ω_{ij}, the weight of the maximum cut $w(S, \overline{S})$ is given by the following integer quadratic program:

$$\begin{array}{lllll} \text{(Q)} & & \max & \frac{1}{2}\sum_{i<j} \omega_{ij}(1 - y_i y_j) & \\ & & \text{subject to:} & y_i \in \{-1, 1\} & \forall i \in \mathsf{V}. \end{array}$$

Indeed, set $S = \left\{ i \;\middle|\; y_i = 1 \right\}$. Clearly, $\omega\left(S, \overline{S}\right) = \frac{1}{2}\sum_{i<j} \omega_{ij}(1 - y_i y_j)$.

Solving quadratic integer programming is of course NP-HARD. Thus, we will relax it, by thinking about the numbers y_i as unit vectors in higher dimensional space. If so, the multiplication of the two vectors, is now replaced by dot product. We have:

$$\begin{aligned} \text{(P)} \qquad \max \quad & \gamma = \frac{1}{2}\sum_{i<j} \omega_{ij}\,(1 - \langle v_i, v_j\rangle) \\ \text{subject to:} \quad & v_i \in \mathbb{S}^{(n)} \qquad \forall i \in V, \end{aligned}$$

where $\mathbb{S}^{(n)}$ is the n dimensional unit sphere in $\mathbb{R}^{n+1}$. This is an instance of semi-definite programming, which is a special case of convex programming, which can be solved in polynomial time (solved here means approximated within a factor of $(1+\varepsilon)$ of optimal, for any arbitrarily small $\varepsilon > 0$, in polynomial time). Namely, the solver finds a feasible solution with a the target function being arbitrarily close to the optimal solution. Observe that (P) is a relaxation of (Q), and as such the optimal solution of (P) has value larger than the optimal value of (Q).

The intuition is that vectors that correspond to vertices that should be on one side of the cut, and vertices on the other sides, would have vectors which are faraway from each other in (P). Thus, we compute the optimal solution for (P), and we uniformly generate a random vector $\vec{r}$ on the unit sphere $\mathbb{S}^{(n)}$. This induces a hyperplane h which passes through the origin and is orthogonal to $\vec{r}$. We next assign all the vectors that are on one side of h to S, and the rest to $\overline{S}$.

Summarizing, the algorithm is as follows: First, we solve (P), next, we pick a random vector $\vec{r}$ uniformly on the unit sphere $\mathbb{S}^{(n)}$. Finally, we set

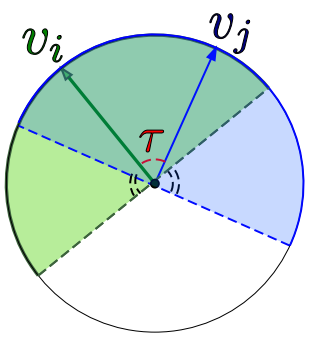
$$S = \left\{ v_i \;\middle|\; \langle v_i, \vec{r}\rangle \geq 0 \right\}.$$

23.1.1 Analysis

The intuition of the above rounding procedure, is that with good probability, vectors in the solution of (P) that have large angle between them would be separated by this cut.

Lemma 23.1.1. *We have* $\mathbf{Pr}\Big[\operatorname{sign}\Big(\langle v_i, \vec{r}\,\rangle\Big) \neq \operatorname{sign}(\langle v_j, \vec{r}\,\rangle)\Big] = \dfrac{1}{\pi}\arccos(\langle v_i, v_j\rangle)$.

Proof: Let us think about the vectors v_i, v_j and $\vec{r}$ as being in the plane. To see why this is a reasonable assumption, consider the plane g spanned by v_i and v_j, and observe that for the random events we consider, only the direction of $\vec{r}$ matter, which can be decided by projecting $\vec{r}$ on g, and normalizing it to have length 1. Now, the sphere is symmetric, and as such, sampling $\vec{r}$ randomly from $\mathbb{S}^{(n)}$, projecting it down to g, and then normalizing it, is equivalent to just choosing uniformly a vector from the unit circle.



Now, $\operatorname{sign}(\langle v_i, \vec{r}\,\rangle) \neq \operatorname{sign}(\langle v_j, \vec{r}\,\rangle)$ happens only if $\vec{r}$ falls in the double wedge formed by the lines perpendicular to v_i and v_j. The angle of this double wedge is exactly the angle between v_i and v_j. Now, since v_i and v_j are unit vectors, we have $\langle v_i, v_j\rangle = \cos(\tau)$, where $\tau = \angle v_i v_j$. Thus,

$$\mathbf{Pr}[\operatorname{sign}(\langle v_i, \vec{r}\,\rangle) \neq \operatorname{sign}(\langle v_j, \vec{r}\,\rangle)] = \frac{2\tau}{2\pi} = \frac{1}{\pi} \cdot \arccos(\langle v_i, v_j\rangle)\,,$$

as claimed. ∎

Theorem 23.1.2. *Let W be the random variable which is the weight of the cut generated by the algorithm. We have*

$$\mathbf{E}[W] = \frac{1}{\pi} \sum_{i<j} \omega_{ij} \arccos(\langle v_i, v_j\rangle)\,.$$

Proof: Let X_{ij} be an indicator variable which is 1 if and only if the edge ij is in the cut. We have

$$\mathbf{E}[X_{ij}] = \mathbf{Pr}[\operatorname{sign}(\langle v_i, \vec{r}\,\rangle) \neq \operatorname{sign}(\langle v_j, \vec{r}\,\rangle)] = \frac{1}{\pi} \arccos(\langle v_i, v_j\rangle)\,,$$

by Lemma 23.1.1. Clearly, $W = \sum_{i<j} \omega_{ij} X_{ij}$, and by linearity of expectation, we have

$$\mathbf{E}[W] = \sum_{i<j} \omega_{ij}\, \mathbf{E}[X_{ij}] = \frac{1}{\pi} \sum_{i<j} \omega_{ij} \arccos(\langle v_i, v_j\rangle)\,.$$

∎

Lemma 23.1.3. *For $-1 \leq y \leq 1$, we have $\dfrac{\arccos(y)}{\pi} \geq \alpha \cdot \dfrac{1}{2}(1-y)$, where $\alpha = \min\limits_{0 \leq \psi \leq \pi} \dfrac{2}{\pi} \dfrac{\psi}{1-\cos(\psi)}$.*

Proof: Set $y = \cos(\psi)$. The inequality now becomes $\frac{\psi}{\pi} \geq \alpha \frac{1}{2}(1-\cos\psi)$. Reorganizing, the inequality becomes $\frac{2}{\pi}\frac{\psi}{1-\cos\psi} \geq \alpha$, which trivially holds by the definition of α. ∎

Lemma 23.1.4. $\alpha > 0.87856$.

Proof: Using simple calculus, one can see that α achieves its value for $\psi = 2.331122...$, the nonzero root of $\cos\psi + \psi \sin\psi = 1$. ∎

Theorem 23.1.5. *The above algorithm computes in expectation a cut with total weight* $\alpha \cdot \text{Opt} \geq 0.87856\text{Opt}$*, where* Opt *is the weight of the maximal cut.*

Proof: Consider the optimal solution to (P), and lets its value be $\gamma \geq \text{Opt}$. We have

$$\mathbf{E}[W] = \frac{1}{\pi}\sum_{i<j}\omega_{ij}\arccos(\langle v_i, v_j\rangle) \geq \sum_{i<j}\omega_{ij}\alpha\frac{1}{2}(1-\langle v_i, v_j\rangle) = \alpha\gamma \geq \alpha\cdot\text{Opt},$$

by Lemma 23.1.3. ■

23.2 Semi-definite programming

Let us define a variable $x_{ij} = \langle v_i, v_j\rangle$, and consider the n by n matrix M formed by those variables, where $x_{ii} = 1$ for $i = 1, \ldots, n$. Let V be the matrix having $v_1, \ldots, v_n$ as its columns. Clearly, $M = V^TV$. In particular, this implies that for any non-zero vector $v \in \mathbb{R}^n$, we have $v^TMv = v^TA^TAv = (Av)^T(Av) \geq 0$. A matrix that has this property, is called ***positive semidefinite***. Interestingly, any positive semidefinite matrix P can be represented as a product of a matrix with its transpose; namely, $P = B^TB$. Furthermore, given such a matrix P of size $n \times n$, we can compute B such that $P = B^TB$ in $O(n^3)$ time. This is know as ***Cholesky decomposition***.

Observe, that if a semidefinite matrix $P = B^TB$ has a diagonal where all the entries are one, then B has columns which are unit vectors. Thus, if we solve (P) and get back a semi-definite matrix, then we can recover the vectors realizing the solution, and use them for the rounding.

In particular, (P) can now be restated as

$$\begin{aligned}
(SD) \qquad \max \quad & \frac{1}{2}\sum_{i<j}\omega_{ij}(1-x_{ij}) \\
\text{subject to:} \quad & x_{ii} = 1 && \text{for } i = 1, \ldots, n \\
& \left(x_{ij}\right)_{i=1,\ldots,n,j=1,\ldots,n} && \text{is a positive semi-definite matrix.}
\end{aligned}$$

We are trying to find the optimal value of a linear function over a set which is the intersection of linear constraints and the set of positive semi-definite matrices.

Lemma 23.2.1. *Let* $\mathcal{U}$ *be the set of* $n \times n$ *positive semidefinite matrices. The set* $\mathcal{U}$ *is convex.*

Proof: Consider $A, B \in \mathcal{U}$, and observe that for any $t \in [0,1]$, and vector $v \in \mathbb{R}^n$, we have:

$$v^T\Big(tA + (1-t)B\Big)v = v^T\Big(tAv + (1-t)Bv\Big) = tv^TAv + (1-t)v^TBv \geq 0 + 0 \geq 0,$$

since A and B are positive semidefinite. ■

Positive semidefinite matrices corresponds to ellipsoids. Indeed, consider the set $x^T Ax = 1$: the set of vectors that solve this equation is an ellipsoid. Also, the eigenvalues of a positive semidefinite matrix are all non-negative real numbers. Thus, given a matrix, we can in polynomial time decide if it is positive semidefinite or not (by computing the eigenvalues of the matrix).

Thus, we are trying to optimize a linear function over a convex domain. There is by now machinery to approximately solve those problems to within any additive error in polynomial time. This is done by using the interior point method, or the ellipsoid method. See [BV04, GLS93] for more details. The key ingredient that is required to make these methods work, is the ability to decide in polynomial time, given a solution, whether its feasible or not. As demonstrated above, this can be done in polynomial time.

23.3 Bibliographical Notes

The approximation algorithm presented is from the work of Goemans and Williamson [GW95]. Håstad [Hås01b] showed that MAX CUT can not be approximated within a factor of $16/17 \approx 0.941176$. Recently, Khot *et al.* [KKMO04] showed a hardness result that matches the constant of Goemans and Williamson (i.e., one can not approximate it better than α, unless P = NP). However, this relies on two conjectures, the first one is the "Unique Games Conjecture", and the other one is "Majority is Stablest". The "Majority is Stablest" conjecture was recently proved by Mossel *et al.* [MOO05]. However, it is not clear if the "Unique Games Conjecture" is true, see the discussion in [KKMO04].

The work of Goemans and Williamson was very influential and spurred wide research on using SDP for approximation algorithms. For an extension of the MAX CUT problem where negative weights are allowed and relevant references, see the work by Alon and Naor [AN04].

Part X

Learning and Linear Separability

Chapter 24

The Perceptron Algorithm

24.1 The perceptron algorithm

Assume, that we are given examples, say a database of cars, and you would like to determine which cars are sport cars, and which are regular cars. Each car record, can be interpreted as a point in high dimensions. For example, a sport car with 4 doors, manufactured in 1997, by Quaky (with manufacturer ID 6) will be represented by the point $(4, 1997, 6)$, marked as a sport car. A tractor made by General Mess (manufacturer ID 3) in 1998, would be stored as $(0, 1997, 3)$ and would be labeled as not a sport car.

Naturally, in a real database there might be hundreds of attributes in each record, for engine size, weight, price, maximum speed, cruising speed, etc, etc, etc.

We would like to automate this ***classification*** process, so that tagging the records whether they correspond to race cars be done automatically without a specialist being involved. We would like to have a learning algorithm, such that given several classified examples, develop its own conjecture about what is the rule of the classification, and we can use it for classifying new data.

That is, there are two stages for ***learning***: ***training*** and ***classifying***. More formally, we are trying to learn a function

$$f : \mathbb{R}^d \to \{-1, 1\} .$$

The challenge is, of course, that f might have infinite complexity – informally, think about a label assigned to items where the label is completely random – there is nothing to learn except knowing the label for all possible items.

This situation is extremely rare is the real world, and we would limit ourselves to a set of functions that can be easily described. For example, consider a set of red and blue points that are linearly separable, as demonstrated in Figure 24.1. That is, we are trying to learn a line ℓ that separates the red points from the blue points.

The natural question is now, given the red and blue points, how to compute the line ℓ? Well, a line or more generally a plane (or even a hyperplane) is the zero set of a linear

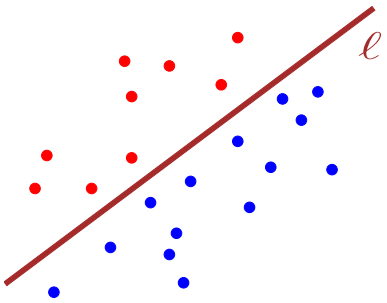


Figure 24.1: Linear separable red and blue point sets.

function, that has the form

$$\forall x \in \mathbb{R}^d \qquad f(x) = \langle a, x \rangle + b,$$

where $a = (a_1, \ldots, a_d), b = (b_1, \ldots, b_d) \in \mathbb{R}^2$, and $\langle a, x \rangle = \sum_i a_i x_i$ is the ***dot product*** of a and x. The classification itself, would be done by computing the sign of $f(x)$; that is $sign(f(x))$. Specifically, if $sign(f(x))$ is negative, it outside the class, if it is positive it is inside.

A set of training examples is a set of pairs

$$S = \Big\{ (x_1, y_1), \ldots, (x_n, y_n) \Big\},$$

where $x_i \in \mathbb{R}^d$ and $y_i \in \{-1,1\}$, for $i = 1, \ldots, n$.

A ***linear classifier*** h is a pair (w, b) where $w \in \mathbb{R}^d$ and $b \in \mathbb{R}$. The classification of $x \in \mathbb{R}^d$ is $sign(\langle w, x \rangle + b)$. For a ***labeled*** example (x, y), h classifies (x, y) correctly if $\text{sign}(\langle w, x \rangle + b) = y$.

Assume that the underlying label we are trying to learn has a linear classifier (this is a problematic assumption – more on this later), and you are given "enough" examples (i.e., n). How to compute the linear classifier for these examples?

One natural option is to use linear programming. Indeed, we are looking for $(\mathbf{w}, b)$, such that for an $(\mathbf{x}_i, y_i)$ we have $\text{sign}(\langle \mathbf{w}, \mathbf{x}_i \rangle + b) = y_i$, which is

$$\begin{aligned} & \langle \mathbf{w}, \mathbf{x}_i \rangle + b \geq 0 && \text{if } y_i = 1, \\ \text{and} \qquad & \langle \mathbf{w}, \mathbf{x}_i \rangle + b \leq 0 && \text{if } y_i = -1. \end{aligned}$$

Or equivalently, let $\mathbf{x}_i = \left(x_i^1, \ldots, x_i^d\right) \in \mathbb{R}^d$, for $i = 1, \ldots, m$, and let $\mathbf{w} = \left(w^1, \ldots, w^d\right)$, then we get the linear constraint

$$\begin{aligned} & \sum_{k=1}^{d} w^k x_i^k + b \geq 0 && \text{if } y_i = 1, \\ \text{and} \qquad & \sum_{k=1}^{d} w^k x_i^k + b \leq 0 && \text{if } y_i = -1. \end{aligned}$$

```
Algorithm perceptron(S: a set of l examples)
  w_0 ← 0, k ← 0
  R = max_{(x,y)∈S} ||x||.
  repeat
    for (x, y) ∈ S do
      if sign(⟨w_k, x⟩) ≠ y then
        w_{k+1} ← w_k + y * x
        k ← k + 1
  until no mistakes are made in the classification
  return w_k and k
```

Figure 24.2: The **perceptron** algorithm.

Thus, we get a set of linear constraints, one for each training example, and we need to solve the resulting linear program.

The main stumbling block is that linear programming is very sensitive to noise. Namely, if we have points that are misclassified, we would not find a solution, because no solution satisfying all of the constraints exists. Instead, we are going to use an iterative algorithm that converges to the optimal solution if it exists, see Figure 24.2.

Why does the **perceptron** algorithm converges to the right solution? Well, assume that we made a mistake on a sample $(\mathbf{x}, y)$ and $y = 1$. Then, $\langle \mathbf{w}_k, \mathbf{x} \rangle < 0$, and

$$\langle \mathbf{w}_{k+1}, \mathbf{x} \rangle = \langle \mathbf{w}_k + y * \mathbf{x}, x \rangle = \langle \mathbf{w}_k, \mathbf{x} \rangle + y \langle \mathbf{x}, \mathbf{x} \rangle = \langle \mathbf{w}_k, \mathbf{x} \rangle + y \|\mathbf{x}\| > \langle \mathbf{w}_k, \mathbf{x} \rangle .$$

Namely, we are "walking" in the right direction, in the sense that the new value assigned to $\mathbf{x}$ by $\mathbf{w}_{k+1}$ is larger ("more positive") than the old value assigned to $\mathbf{x}$ by $\mathbf{w}_k$.

Theorem 24.1.1. *Let S be a training set of examples, and let $R = \max_{(x,y)\in S} \|x\|$. Suppose that there exists a vector w_{opt} such that $\|w_{opt}\| = 1$, and a number $\gamma > 0$, such that*

$$y \langle w_{opt}, x \rangle \geq \gamma \qquad \forall (x, y) \in S.$$

Then, the number of mistakes made by the online **perceptron** *algorithm on S is at most*

$$\left(\frac{R}{\gamma}\right)^2 .$$

Proof: Intuitively, the **perceptron** algorithm weight vector converges to w_{opt}, To see that, let us define the distance between w_{opt} and the weight vector in the kth update:

$$\alpha_k = \left\| w_k - \frac{R^2}{\gamma} w_{opt} \right\|^2 .$$

We next quantify the change between α_k and α_{k+1} (the example being misclassified is (x, y)):

$$\begin{aligned}
\alpha_{k+1} &= \left\| w_{k+1} - \frac{R^2}{\gamma} w_{opt} \right\|^2 \\
&= \left\| w_k + y\mathbf{x} - \frac{R^2}{\gamma} w_{opt} \right\|^2 \\
&= \left\| \left(w_k - \frac{R^2}{\gamma} w_{opt} \right) + y\mathbf{x} \right\|^2 \\
&= \left\langle \left(w_k - \frac{R^2}{\gamma} w_{opt} \right) + y\mathbf{x},\ \left(w_k - \frac{R^2}{\gamma} w_{opt} \right) + y\mathbf{x} \right\rangle .
\end{aligned}$$

Expanding this we get:

$$\begin{aligned}
\alpha_{k+1} &= \left\langle \left(w_k - \frac{R^2}{\gamma} w_{opt} \right), \left(w_k - \frac{R^2}{\gamma} w_{opt} \right) \right\rangle + 2y \left\langle \left(w_k - \frac{R^2}{\gamma} w_{opt} \right), \mathbf{x} \right\rangle + \langle \mathbf{x}, \mathbf{x} \rangle \\
&= \alpha_k + 2y \left\langle \left(w_k - \frac{R^2}{\gamma} w_{opt} \right), x \right\rangle + \left\| x \right\|^2 .
\end{aligned}$$

As $(\mathbf{x}, y)$ is misclassified, it must be that $\text{sign}(\langle w_k, \mathbf{x} \rangle) \neq y$, which implies that $\text{sign}(y \langle \mathbf{w}_k, \mathbf{x} \rangle) = -1$; that is $y \langle \mathbf{w}_k, \mathbf{x} \rangle < 0$. Now, since $\left\| x \right\| \leq R$, we have

$$\begin{aligned}
\alpha_{k+1} &\leq \alpha_k + R^2 + 2y \langle w_k, \mathbf{x} \rangle - 2y \left\langle \frac{R^2}{\gamma} w_{opt}, \mathbf{x} \right\rangle \\
&\leq \alpha_k + R^2 + \qquad\qquad -2\frac{R^2}{\gamma} y \langle w_{opt}, x \rangle .
\end{aligned}$$

Next, since $y \langle w_{opt}\ ,\ x \rangle \geq \gamma$ for $\forall (x, y) \in S$, we have that

$$\begin{aligned}
\alpha_{k+1} &\leq \alpha_k + R^2 - 2\frac{R^2}{\gamma}\gamma \\
&\leq \alpha_k + R^2 - 2R^2 \\
&\leq \alpha_k - R^2 .
\end{aligned}$$

We have: $\alpha_{k+1} \leq \alpha_k - R^2$, and

$$\alpha_0 \ = \ \left\| 0 - \frac{R^2}{\gamma} w_{opt} \right\|^2 = \frac{R^4}{\gamma^2} \left\| w_{opt} \right\|^2 = \frac{R^4}{\gamma^2} \ .$$

Finally, observe that $\alpha_i \geq 0$ for all i. Thus, what is the maximum number of classification errors the algorithm can make?

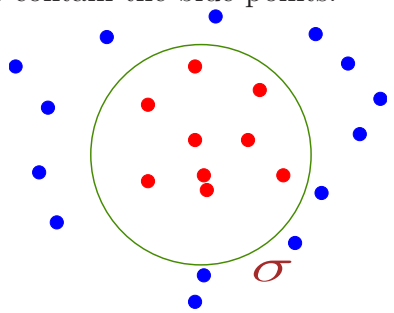
$$\left(\frac{R^2}{\gamma^2} \right) .$$

■

It is important to observe that any linear program can be written as the problem of separating red points from blue points. As such, the **perceptron** algorithm can be used to solve linear programs.

24.2 Learning A Circle

Given a set of red points, and blue points in the plane, we want to learn a circle that contains all the red points, and does not contain the blue points.



How to compute the circle σ ?

It turns out we need a simple but very clever trick. For every point $(x, y) \in P$ map it to the point $(x,\ y,\ x^2 + y^2)$. Let $z(P) = \left\{(x,\ y,\ x^2 + y^2) \;\middle|\; (x, y) \in P \right\}$ be the resulting point set.

Theorem 24.2.1. *Two sets of points R and B are separable by a circle in two dimensions, if and only if $z(R)$ and $z(B)$ are separable by a plane in three dimensions.*

Proof: Let $\sigma \equiv (x - a)^2 + (y - b)^2 = r^2$ be the circle containing all the points of R and having all the points of B outside. Clearly, $(x - a)^2 + (y - b)^2 \leq r^2$ for all the points of R. Equivalently

$$-2ax - 2by + \left(x^2 + y^2\right) \leq r^2 - a^2 - b^2.$$

Setting $z = x^2 + y^2$ we get that

$$h \equiv -2ax - 2by + z - r^2 + a^2 + b^2 \leq 0.$$

Namely, $p \in \sigma$ if and only if $h(z(p)) \leq 0$. We just proved that if the point set is separable by a circle, then the lifted point set $z(R)$ and $z(B)$ are separable by a plane.

As for the other direction, assume that $z(R)$ and $z(B)$ are separable in $3d$ and let

$$h \equiv ax + by + cz + d = 0$$

be the separating plane, such that all the point of $z(R)$ evaluate to a negative number by h. Namely, for $(x, y, x^2 + y^2) \in z(R)$ we have $ax + by + c(x^2 + y^2) + d \leq 0$

and similarly, for $(x, y, x^2 + y^2) \in B$ we have $ax + by + c(x^2 + y^2) + d \geq 0$.

Let $U(h) = \Big\{(x,y) \;\Big|\; h((x,y,x^2+y^2)) \leq 0\Big\}$. Clearly, if $U(h)$ is a circle, then this implies that $R \subset U(h)$ and $B \cap U(h) = \emptyset$, as required.

So, $U(h)$ are all the points in the plane, such that

$$ax + by + c\Big(x^2 + y^2\Big) \leq -d.$$

Equivalently

$$\left(x^2 + \frac{a}{c}x\right) + \left(y^2 + \frac{b}{c}y\right) \leq -\frac{d}{c}$$

$$\left(x + \frac{a}{2c}\right)^2 + \left(y + \frac{b}{2c}\right)^2 \leq \frac{a^2+b^2}{4c^2} - \frac{d}{c}$$

but this defines the interior of a circle in the plane, as claimed. ■

This example show that linear separability is a powerful technique that can be used to learn complicated concepts that are considerably more complicated than just hyperplane separation. This lifting technique showed above is the ***kernel technique*** or ***linearization***.

24.3 A Little Bit On VC Dimension

As we mentioned, inherent to the learning algorithms, is the concept of how complex is the function we are trying to learn. VC-dimension is one of the most natural ways of capturing this notion. (VC = Vapnik, Chervonenkis,1971).

A matter of expressivity. What is harder to learn:

1. A rectangle in the plane.
2. A halfplane.
3. A convex polygon with k sides.

Let $X = \{p_1, p_2, \ldots, p_m\}$ be a set of m points in the plane, and let R be the set of all halfplanes.

A half-plane r defines a binary vector

$$r(X) = (b_1, \ldots, b_m)$$

where $b_i = 1$ if and only if p_i is inside r.

Let

$$U(X, R) = \{r(X) \mid r \in R\}.$$

A set X of m elements is *shattered* by R if

$$|U(X, R)| = 2^m.$$

What does this mean?

The VC-dimension of a set of ranges R is the size of the largest set that it can shatter.

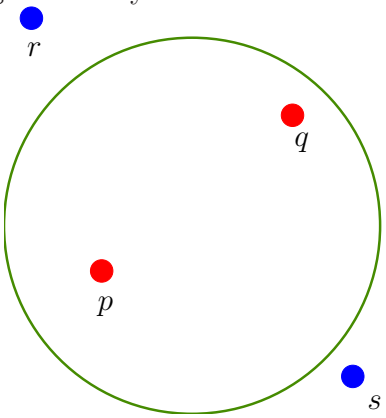
24.3.1 Examples

What is the VC dimensions of circles in the plane?

Namely, X is set of n points in the plane, and R is a set of all circles.

$X = \{p, q, r, s\}$

What subsets of X can we generate by circle?



$\{\}, \{r\}, \{p\}, \{q\}, \{s\}, \{p, s\}, \{p, q\}, \{p, r\}, \{r, q\}\{q, s\}$ and $\{r, p, q\}$, $\{p, r, s\}\{p, s, q\}, \{s, q, r\}$ and $\{r, p, q, s\}$

We got only 15 sets. There is one set which is not there. Which one?

The VC dimension of circles in the plane is 3.

Lemma 24.3.1 (Sauer Lemma). *If R has VC dimension d then $|U(X, R)| = O\left(m^d\right)$, where m is the size of X.*

Part XI

Compression, Information and Entropy

Chapter 25

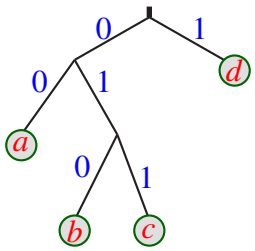
Huffman Coding

25.1 Huffman coding

(This portion of the class notes is based on Jeff Erickson class notes.)

A ***binary code*** assigns a string of 0s and 1s to each character in the alphabet. A code assigns for each symbol in the input a codeword over some other alphabet. Such a coding is necessary, for example, for transmitting messages over a wire, were you can send only 0 or 1 on the wire (i.e., for example, consider the good old telegraph and Morse code). The receiver gets a binary stream of bits and needs to decode the message sent. A prefix code, is a code where one can decipher the message, a character by character, by reading a prefix of the input binary string, matching it to a code word (i.e., string), and continuing to decipher the rest of the stream. Such a code is a ***prefix code***.

A binary code (or a prefix code) is ***prefix-free*** if no code is a prefix of any other. ASCII and Unicode's UTF-8 are both prefix-free binary codes. Morse code is a binary code (and also a prefix code), but it is not prefix-free; for example, the code for `S` ($\cdot\cdot\cdot$) includes the code for `E` ($\cdot$) as a prefix. (Hopefully the receiver knows that when it gets $\cdot\cdot\cdot$ that it is extremely unlikely that this should be interpreted as `EEE`, but rather `S`.



Any prefix-free binary code can be visualized as a binary tree with the encoded characters stored at the leaves. The code word for any symbol is given by the path from the root to the corresponding leaf; 0 for left, 1 for right. The length of a codeword for a symbol is the depth of the corresponding leaf. Such trees are usually referred to as ***prefix trees*** or ***code trees***.

The beauty of prefix trees (and thus of prefix odes) is that decoding is easy. As a concrete example, consider the tree on the right. Given a string '010100', we can traverse down the tree from the root, going left if get a '0' and right if we get '1'. Whenever we get to a leaf, we output the character output in the leaf, and we jump back to the root for the next character we are about to read. For the example '010100', after reading '010' our traversal in the tree leads us to the leaf marked with 'b', we jump back to the root and read the next input digit, which is '1', and this leads us to the leaf marked with 'd',

newline	16, 492
space	130,376
‘!’	955
‘"’	5,681
‘$’	2
‘%’	1
‘”’	1,174
‘(’	151
‘)’	151
‘*’	70
‘,’	13,276
‘–’	2,430
‘.’	6,769

‘0’	20
‘1’	61
‘2’	10
‘3’	12
‘4’	10
‘5’	14
‘6’	11
‘7’	13
‘8’	13
‘9’	14
‘:’	267
‘;’	1,108
‘?’	913

‘A’	48,165
‘B’	8,414
‘C’	13,896
‘D’	28,041
‘E’	74,809
‘F’	13,559
‘G’	12,530
‘H’	38,961
‘I’	41,005
‘J’	710
‘K’	4,782
‘L’	22,030
‘M’	15,298

‘N’	42,380
‘O’	46,499
‘P’	9,957
‘Q’	667
‘R’	37,187
‘S’	37,575
‘T’	54,024
‘U’	16,726
‘V’	5,199
‘W’	14,113
‘X’	724
‘Y’	12,177
‘Z’	215

‘_’	182
‘‘’	93
‘@’	2
‘/’	26

Figure 25.1: Frequency of characters in the book “A tale of two cities” by Dickens. For the sake of brevity, small letters were counted together with capital letters.

which we output, and jump back to the root. Finally, ’00’ leads us to the leaf marked by ’a’, which the algorithm output. Thus, the binary string ’010100’ encodes the string `"bda"`.

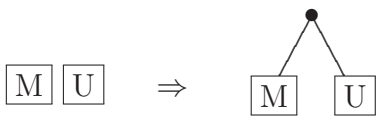
Suppose we want to encode messages in an n-character alphabet so that the encoded message is as short as possible. Specifically, given an array frequency counts $f[1 \ldots n]$, we want to compute a prefix-free binary code that minimizes the total encoded length of the message. That is we would like to compute a tree $\mathcal{T}$ that minimizes

$$\text{cost}(\mathcal{T}) = \sum_{i=1}^{n} f[i] * \text{len}(\text{code}(i)) \,, \tag{25.1}$$

where $\text{code}(i)$ is the binary string encoding the ith character and $\text{len}(s)$ is the length (in bits) of the binary string s.

As a concrete example, consider Figure 25.1, which shows the frequency of characters in the book “A tale of two cities”, which we would like to encode. Consider the characters ‘E’ and ‘Q’. The first appears $> 74,000$ times in the text, and other appears only 667 times in the text. Clearly, it would be logical to give ‘E’, the most frequent letter in English, a very short prefix code, and a very long (as far as number of bits) code to ‘Q’.

A nice property of this problem is that given two trees for some parts of the alphabet, we can easily put them together into a larger tree by just creating a new node and hanging the trees from this common node. For example, putting two characters together, we have the following.



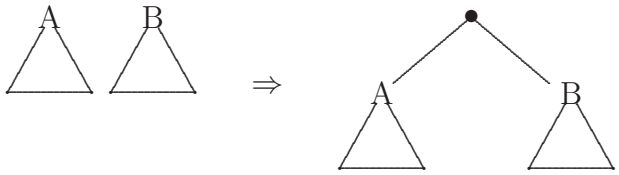
Similarly, we can put together two subtrees.

char	frequency	code
'A'	48165	1110
'B'	8414	101000
'C'	13896	00100
'D'	28041	0011
'E'	74809	011
'F'	13559	111111
'G'	12530	111110
'H'	38961	1001

char	frequency	code
'I'	41005	1011
'J'	710	1111011010
'K'	4782	11110111
'L'	22030	10101
'M'	15298	01000
'N'	42380	1100
'O'	46499	1101
'P'	9957	101001
'Q'	667	1111011001

char	frequency	code
'R'	37187	0101
'S'	37575	1000
'T'	54024	000
'U'	16726	01001
'V'	5199	1111010
'W'	14113	00101
'X'	724	1111011011
'Y'	12177	111100
'Z'	215	1111011000

Figure 25.2: The resulting prefix code for the frequencies of Figure 25.1. Here, for the sake of simplicity of exposition, the code was constructed only for the A—Z characters.



25.1.1 The algorithm to build Hoffman's code

This suggests a simple algorithm that takes the two least frequent characters in the current frequency table, merge them into a tree, and put the merged tree back into the table (instead of the two old trees). The algorithm stops when there is a single tree. The intuition is that infrequent characters would participate in a large number of merges, and as such would be low in the tree – they would be assigned a long code word.

This algorithm is due to David Huffman, who developed it in 1952. Shockingly, this code is the best one can do. Namely, the resulting code is *asymptotically* gives the best possible compression of the data (of course, one can do better compression in practice using additional properties of the data and careful hacking). This ***Huffman coding*** is used widely and is the basic building block used by numerous other compression algorithms.

To see how such a resulting tree (and the associated code) looks like, see Figure 25.2 and Figure 25.3.

25.1.2 Analysis

Lemma 25.1.1. *Let $\mathcal{T}$ be an optimal code tree. Then $\mathcal{T}$ is a full binary tree (i.e., every node of $\mathcal{T}$ has either 0 or 2 children).*

In particular, if the height of $\mathcal{T}$ is d, then there are leafs nodes of height d that are sibling.

Proof: If there is an internal node in $\mathcal{T}$ that has one child, we can remove this node from $\mathcal{T}$, by connecting its only child directly with its parent. The resulting code tree is clearly a better compressor, in the sense of Eq. (25.1).

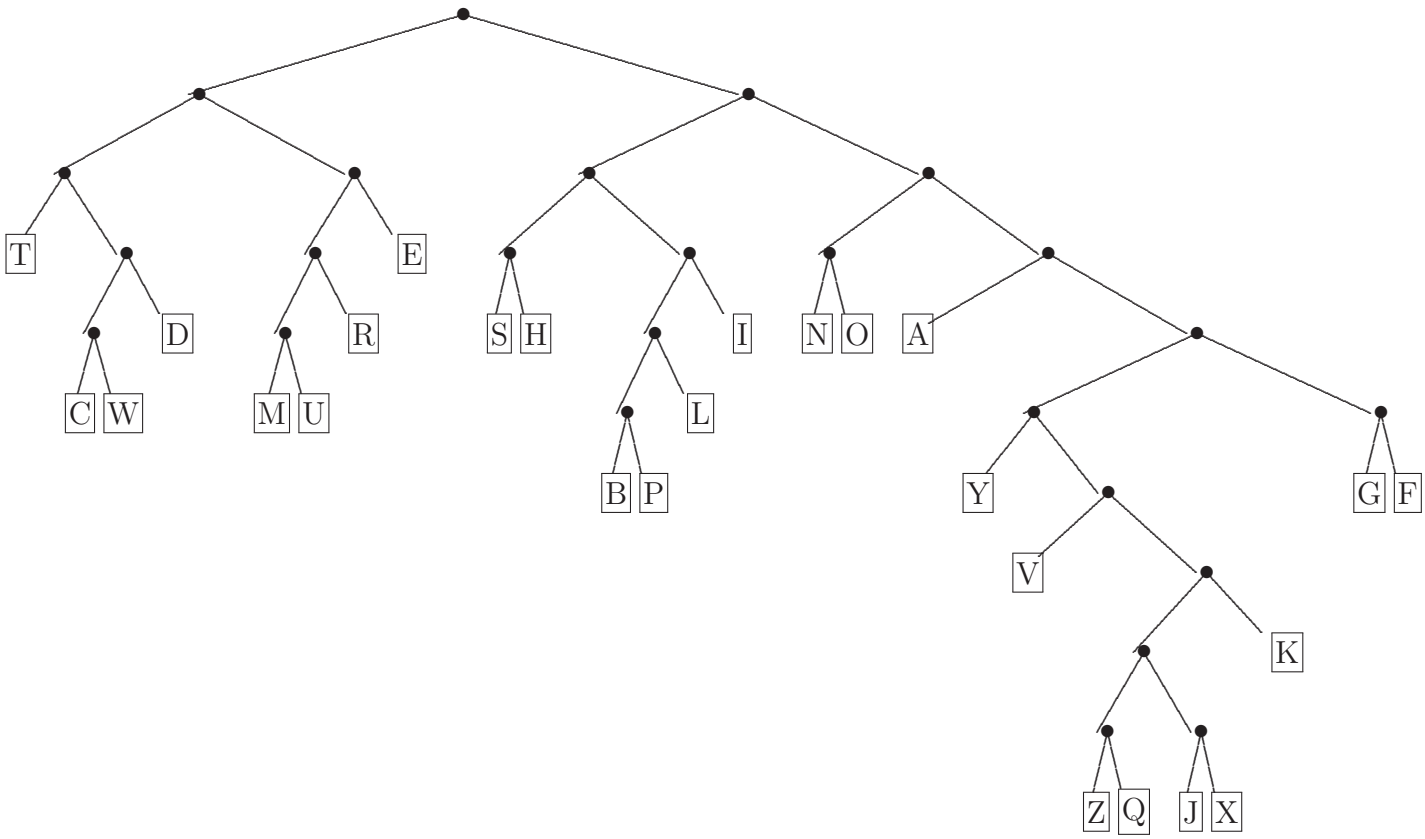


Figure 25.3: The Huffman tree generating the code of Figure 25.2.

As for the second claim, consider a leaf u with maximum depth d in $\mathcal{T}$, and consider it parent $v = \overline{\mathrm{p}}(u)$. The node v has two children, and they are both leafs (otherwise u would not be the deepest node in the tree), as claimed. ■

Lemma 25.1.2. *Let x and y be the two least frequent characters (breaking ties between equally frequent characters arbitrarily). There is an optimal code tree in which x and y are siblings.*

Proof: More precisely, there is an optimal code in which x and y are siblings and have the largest depth of any leaf. Indeed, let $\mathcal{T}$ be an optimal code tree with depth d. The tree $\mathcal{T}$ has at least two leaves at depth d that are siblings, by Lemma 25.1.1.

Now, suppose those two leaves are not x and y, but some other characters α and β. Let $\mathcal{T}'$ be the code tree obtained by swapping x and α. The depth of x increases by some amount Δ, and the depth of α decreases by the same amount. Thus,

$$\mathrm{cost}(\mathcal{T}') = \mathrm{cost}(\mathcal{T}) - \Big(f[\alpha] - f[x]\Big)\,\Delta.$$

By assumption, x is one of the two least frequent characters, but α is not, which implies that $f[\alpha] > f[x]$. Thus, swapping x and α does not increase the total cost of the code. Since $\mathcal{T}$

was an optimal code tree, swapping x and α does not decrease the cost, either. Thus, $\mathcal{T}'$ is also an optimal code tree (and incidentally, $f[\alpha]$ actually equals $f[x]$). Similarly, swapping y and b must give yet another optimal code tree. In this final optimal code tree, x and y as maximum-depth siblings, as required. ∎

Theorem 25.1.3. *Huffman codes are optimal prefix-free binary codes.*

Proof: If the message has only one or two different characters, the theorem is trivial. Otherwise, let $f[1 \ldots n]$ be the original input frequencies, where without loss of generality, $f[1]$ and $f[2]$ are the two smallest. To keep things simple, let $f[n+1] = f[1] + f[2]$. By the previous lemma, we know that some optimal code for $f[1..n]$ has characters 1 and 2 as siblings. Let $\mathcal{T}_{\text{opt}}$ be this optimal tree, and consider the tree formed by it by removing 1 and 2 as it leaves. We remain with a tree $\mathcal{T}'_{\text{opt}}$ that has as leafs the characters $3, \ldots, n$ and a "special" character $n+1$ (which is the parent of 1 and 2 in $\mathcal{T}_{\text{opt}}$) that has frequency $f[n+1]$. Now, since $f[n+1] = f[1] + f[2]$, we have

$$\begin{aligned}
\text{cost}(\mathcal{T}_{\text{opt}}) &= \sum_{i=1}^{n} f[i]\text{depth}_{\mathcal{T}_{\text{opt}}}(i) \\
&= \sum_{i=3}^{n+1} f[i]\text{depth}_{\mathcal{T}_{\text{opt}}}(i) + f[1]\text{depth}_{\mathcal{T}_{\text{opt}}}(1) + f[2]\text{depth}_{\mathcal{T}_{\text{opt}}}(2) - f[n+1]\text{depth}_{\mathcal{T}_{\text{opt}}}(n+1) \\
&= \text{cost}\Big(\mathcal{T}'_{\text{opt}}\Big) + \Big(f[1] + f[2]\Big)\,\text{depth}(\mathcal{T}_{\text{opt}}) - \Big(f[1] + f[2]\Big)(\text{depth}(\mathcal{T}_{\text{opt}}) - 1) \\
&= \text{cost}\Big(\mathcal{T}'_{\text{opt}}\Big) + f[1] + f[2]. \qquad (25.2)
\end{aligned}$$

This implies that minimizing the cost of $\mathcal{T}_{\text{opt}}$ is equivalent to minimizing the cost of $\mathcal{T}'_{\text{opt}}$. In particular, $\mathcal{T}'_{\text{opt}}$ must be an optimal coding tree for $f[3 \ldots n+1]$. Now, consider the Huffman tree $\mathcal{T}'_H$ constructed for $f[3, \ldots, n+1]$ and the overall Huffman tree $\mathcal{T}_H$ constructed for $f[1, \ldots, n]$. By the way the construction algorithm works, we have that $\mathcal{T}'_H$ is formed by removing the leafs of 1 and 2 from $\mathcal{T}$. Now, by induction, we know that the Huffman tree generated for $f[3, \ldots, n+1]$ is optimal; namely, $\text{cost}\Big(\mathcal{T}'_{\text{opt}}\Big) = \text{cost}(\mathcal{T}'_H)$. As such, arguing as above, we have

$$\text{cost}(\mathcal{T}_H) = \text{cost}(\mathcal{T}'_H) + f[1] + f[2] = \text{cost}\Big(\mathcal{T}'_{\text{opt}}\Big) + f[1] + f[2] = \text{cost}(\mathcal{T}_{\text{opt}})\,,$$

by Eq. (25.2). Namely, the Huffman tree has the same cost as the optimal tree. ∎

25.1.3 What do we get

For the book "A tale of two cities" which is made out of 779,940 bytes, and using the above Huffman compression results in a compression to a file of size 439,688 bytes. A far cry from what `gzip` can do (301,295 bytes) or `bzip2` can do (220,156 bytes!), but still very impressive when you consider that the Huffman encoder can be easily written in a few hours of work.

(This numbers ignore the space required to store the code with the file. This is pretty small, and would not change the compression numbers stated above significantly.

25.1.4 A formula for the average size of a code word

Assume that our input is made out of n characters, where the ith character is p_i fraction of the input (one can think about p_i as the probability of seeing the ith character, if we were to pick a random character from the input).

Now, we can use these probabilities instead of frequencies to build a Huffman tree. The natural question is what is the length of the codewords assigned to characters as a function of their probabilities?

In general this question does not have a trivial answer, but there is a simple elegant answer, if all the probabilities are power of 2.

Lemma 25.1.4. *Let $1, \ldots, n$ be n symbols, such that the probability for the ith symbol is p_i, and furthermore, there is an integer $l_i \geq 0$, such that $p_i = 1/2^{l_i}$. Then, in the Huffman coding for this input, the code for i is of length l_i.*

Proof: The proof is by easy induction of the Huffman algorithm. Indeed, for $n = 2$ the claim trivially holds since there are only two characters with probability $1/2$. Otherwise, let i and j be the two characters with lowest probability. It must hold that $p_i = p_j$ (otherwise, $\sum_k p_k$ can not be equal to one). As such, Huffman's merges this two letters, into a single "character" that have probability $2p_i$, which would have encoding of length $l_i - 1$, by induction (on the remaining $n - 1$ symbols). Now, the resulting tree encodes i and j by code words of length $(l_i - 1) + 1 = l_i$, as claimed. ∎

In particular, we have that $l_i = \lg 1/p_i$. This implies that the average length of a code word is

$$\sum_i p_i \lg \frac{1}{p_i}.$$

If we consider X to be a random variable that takes a value i with probability p_i, then this formula is

$$\mathbb{H}(X) = \sum_i \mathbf{Pr}[X = i] \lg \frac{1}{\mathbf{Pr}[X = i]},$$

which is the ***entropy*** of X.

Chapter 26

Entropy, Randomness, and Information

"If only once - only once - no matter where, no matter before what audience - I could better the record of the great Rastelli and juggle with thirteen balls, instead of my usual twelve, I would feel that I had truly accomplished something for my country. But I am not getting any younger, and although I am still at the peak of my powers there are moments - why deny it? - when I begin to doubt - and there is a time limit on all of us."

– –Romain Gary, The talent scout..

26.1 Entropy

Definition 26.1.1. The ***entropy*** in bits of a discrete random variable X is given by

$$\mathbb{H}(X) = -\sum_x \mathbf{Pr}\Big[X = x\Big] \lg \mathbf{Pr}\Big[X = x\Big].$$

Equivalently, $\mathbb{H}(X) = \mathbf{E}\Big[\lg \frac{1}{\mathbf{Pr}[X]}\Big]$.

The ***binary entropy*** function $\mathbb{H}(p)$ for a random binary variable that is 1 with probability p, is $\mathbb{H}(p) = -p \lg p - (1-p)\lg(1-p)$. We define $\mathbb{H}(0) = \mathbb{H}(1) = 0$. See Figure 26.1.

The function $\mathbb{H}(p)$ is a concave symmetric around $1/2$ on the interval $[0, 1]$ and achieves its maximum at $1/2$. For a concrete example, consider $\mathbb{H}(3/4) \approx 0.8113$ and $\mathbb{H}(7/8) \approx 0.5436$. Namely, a coin that has $3/4$ probably to be heads have higher amount of "randomness" in it than a coin that has probability $7/8$ for heads.

We have $\mathbb{H}'(p) = -\lg p + \lg(1-p) = \lg \frac{1-p}{p}$ and $\mathbb{H}''(p) = \frac{p}{1-p} \cdot \Big(-\frac{1}{p^2}\Big) = -\frac{1}{p(1-p)}$. Thus, $\mathbb{H}''(p) \leq 0$, for all $p \in (0, 1)$, and the $\mathbb{H}(\cdot)$ is concave in this range. Also, $\mathbb{H}'(1/2) = 0$, which implies that $\mathbb{H}(1/2) = 1$ is a maximum of the binary entropy. Namely, a balanced coin has the largest amount of randomness in it.

Example 26.1.2. A random variable X that has probability $1/n$ to be i, for $i = 1, \ldots, n$, has entropy $\mathbb{H}(X) = -\sum_{i=1}^n \frac{1}{n} \lg \frac{1}{n} = \lg n$.

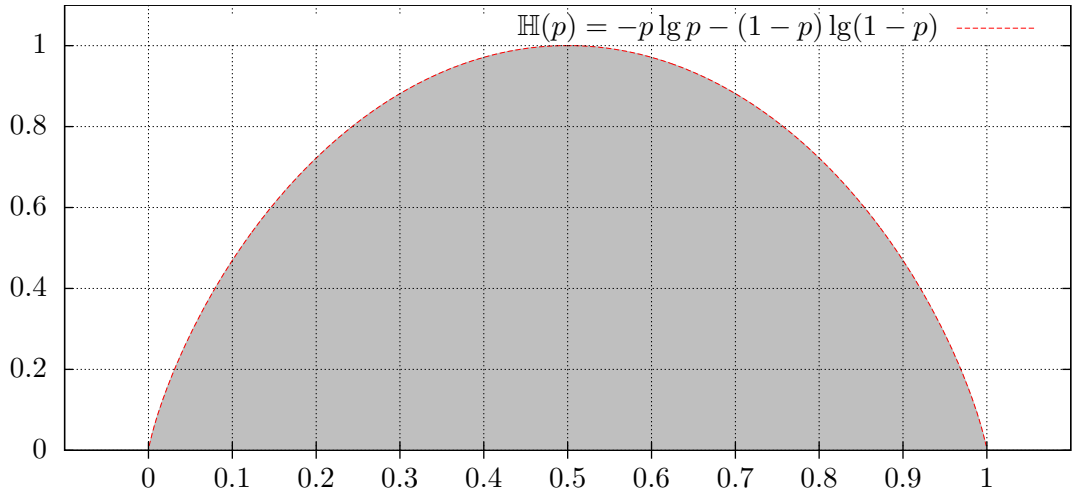


Figure 26.1: The binary entropy function.

Note, that the entropy is oblivious to the exact values that the random variable can have, and it is sensitive only to the probability distribution. Thus, a random variables that accepts $-1, +1$ with equal probability has the same entropy (i.e., 1) as a fair coin.

Lemma 26.1.3. *Let X and Y be two independent random variables, and let Z be the random variable (X, Y). Then $\mathbb{H}(Z) = \mathbb{H}(X) + \mathbb{H}(Y)$.*

Proof: In the following, summation are over all possible values that the variables can have. By the independence of X and Y we have

$$\begin{aligned}
\mathbb{H}(Z) &= \sum_{x,y} \mathbf{Pr}\Big[(X,Y) = (x,y)\Big] \lg \frac{1}{\mathbf{Pr}[(X,Y) = (x,y)]} \\
&= \sum_{x,y} \mathbf{Pr}\Big[X = x\Big] \mathbf{Pr}\Big[Y = y\Big] \lg \frac{1}{\mathbf{Pr}[X = x]\,\mathbf{Pr}[Y = y]} \\
&= \sum_{x} \sum_{y} \mathbf{Pr}[X = x]\,\mathbf{Pr}[Y = y] \lg \frac{1}{\mathbf{Pr}[X = x]} \\
&\qquad + \sum_{y} \sum_{x} \mathbf{Pr}[X = x]\,\mathbf{Pr}[Y = y] \lg \frac{1}{\mathbf{Pr}[Y = y]} \\
&= \sum_{x} \mathbf{Pr}[X = x] \lg \frac{1}{\mathbf{Pr}[X = x]} + \sum_{y} \mathbf{Pr}[Y = y] \lg \frac{1}{\mathbf{Pr}[Y = y]} \\
&= \mathbb{H}(X) + \mathbb{H}(Y)\,.
\end{aligned}$$

■

Lemma 26.1.4. *Suppose that nq is integer in the range $[0, n]$. Then $\dfrac{2^{n\mathbb{H}(q)}}{n+1} \leq \binom{n}{nq} \leq 2^{n\mathbb{H}(q)}$.*

Proof: This trivially holds if $q = 0$ or $q = 1$, so assume $0 < q < 1$. We know that

$$\binom{n}{nq} q^{nq}(1-q)^{n-nq} \leq (q + (1-q))^n = 1.$$

As such, since $q^{-nq}(1-q)^{-(1-q)n} = 2^{n(-q \lg q - (1-q)\lg(1-q))} = 2^{n\mathbb{H}(q)}$, we have

$$\binom{n}{nq} \leq q^{-nq}(1-q)^{-(1-q)n} = 2^{n\mathbb{H}(q)}.$$

As for the other direction, let $\mu(k) = \binom{n}{k} q^k (1-q)^{n-k}$. We claim that $\mu(nq) = \binom{n}{nq} q^{nq}(1-q)^{n-nq}$ is the largest term in $\sum_{k=0}^{n} \mu(k) = 1$. Indeed,

$$\Delta_k = \mu(k) - \mu(k+1) = \binom{n}{k} q^k (1-q)^{n-k} \left(1 - \frac{n-k}{k+1}\frac{q}{1-q}\right),$$

and the sign of this quantity is the sign of the last term, which is

$$\operatorname{sign}(\Delta_k) = \operatorname{sign}\left(1 - \frac{(n-k)q}{(k+1)(1-q)}\right) = \operatorname{sign}\left(\frac{(k+1)(1-q) - (n-k)q}{(k+1)(1-q)}\right).$$

Now,

$$(k+1)(1-q) - (n-k)q = k + 1 - kq - q - nq + kq = 1 + k - q - nq.$$

Namely, $\Delta_k \geq 0$ when $k \geq nq + q - 1$, and $\Delta_k < 0$ otherwise. Namely, $\mu(k) < \mu(k+1)$, for $k < nq$, and $\mu(k) \geq \mu(k+1)$ for $k \geq nq$. Namely, $\mu(nq)$ is the largest term in $\sum_{k=0}^{n} \mu(k) = 1$, and as such it is larger than the average. We have $\mu(nq) = \binom{n}{nq} q^{nq}(1-q)^{n-nq} \geq \frac{1}{n+1}$, which implies

$$\binom{n}{nq} \geq \frac{1}{n+1} q^{-nq}(1-q)^{-(n-nq)} = \frac{1}{n+1} 2^{n\mathbb{H}(q)}.$$

■

Lemma 26.1.4 can be extended to handle non-integer values of q. This is straightforward, and we omit the easy but tedious details.

Corollary 26.1.5. *We have:*
(i) $q \in [0, 1/2] \Rightarrow \binom{n}{\lfloor nq \rfloor} \leq 2^{n\mathbb{H}(q)}$. *(ii)* $q \in [1/2, 1]$ $\binom{n}{\lceil nq \rceil} \leq 2^{n\mathbb{H}(q)}$.

(iii) $q \in [1/2, 1] \Rightarrow \frac{2^{n\mathbb{H}(q)}}{n+1} \leq \binom{n}{\lfloor nq \rfloor}$. *(iv)* $q \in [0, 1/2] \Rightarrow \frac{2^{n\mathbb{H}(q)}}{n+1} \leq \binom{n}{\lceil nq \rceil}$.

The bounds of Lemma 26.1.4 and Corollary 26.1.5 are loose but sufficient for our purposes. As a sanity check, consider the case when we generate a sequence of n bits using a coin with probability q for head, then by the Chernoff inequality, we will get roughly nq heads in this sequence. As such, the generated sequence Y belongs to $\binom{n}{nq} \approx 2^{n\mathbb{H}(q)}$ possible sequences that have similar probability. As such, $\mathbb{H}(Y) \approx \lg \binom{n}{nq} = n\mathbb{H}(q)$, by Example 26.1.2, a fact that we already know from Lemma 26.1.3.

26.1.1 Extracting randomness

Entropy can be interpreted as the amount of unbiased random coin flips can be extracted from a random variable.

Definition 26.1.6. An extraction function $\mathbf{Ext}$ takes as input the value of a random variable X and outputs a sequence of bits y, such that $\mathbf{Pr}\Big[\mathbf{Ext}(X) = y \;\Big|\; |y| = k\Big] = \frac{1}{2^k}$, whenever $\mathbf{Pr}[|y| = k] > 0$, where $|y|$ denotes the length of y.

As a concrete (easy) example, consider X to be a uniform random integer variable out of $0, \ldots, 7$. All that $\mathbf{Ext}(X)$ has to do in this case, is to compute the binary representation of x. However, note that Definition 26.1.6 is somewhat more subtle, as it requires that all extracted sequence of the same length would have the same probability.

Thus, for X a uniform random integer variable in the range $0, \ldots, 11$, the function $\mathbf{Ext}(x)$ can output the binary representation for x if $0 \le x \le 7$. However, what do we do if x is between 8 and 11? The idea is to output the binary representation of $x - 8$ as a two bit number. Clearly, Definition 26.1.6 holds for this extraction function, since $\mathbf{Pr}\Big[\mathbf{Ext}(X) = 00 \;\Big|\; |\mathbf{Ext}(X)| = 2\Big] = \frac{1}{4}$, as required. This scheme can be of course extracted for any range.

The following is obvious, but we provide a proof anyway.

Lemma 26.1.7. *Let x/y be a faction, such that $x/y < 1$. Then, for any i, we have $x/y < (x+i)/(y+i)$.*

Proof: We need to prove that $x(y+i) - (x+i)y < 0$. The left size is equal to $i(x-y)$, but since $y > x$ (as $x/y < 1$), this quantity is negative, as required. ∎

Theorem 26.1.8. *Suppose that the value of a random variable X is chosen uniformly at random from the integers $\{0, \ldots, m-1\}$. Then there is an extraction function for X that outputs on average at least $\lfloor \lg m \rfloor - 1 = \lfloor \mathbb{H}(X) \rfloor - 1$ independent and unbiased bits.*

Proof: We represent m as a sum of unique powers of 2, namely $m = \sum_i a_i 2^i$, where $a_i \in \{0, 1\}$. Thus, we decomposed $\{0, \ldots, m-1\}$ into a disjoint union of blocks that have sizes which are distinct powers of 2. If a number falls inside such a block, we output its relative location in the block, using binary representation of the appropriate length (i.e., k if the block is of size 2^k). One can verify that this is an extraction function, fulfilling Definition 26.1.6.

Now, observe that the claim holds trivially if m is a power of two. Thus, consider the case that m is not a power of 2. If X falls inside a block of size 2^k then the entropy is k. Thus, for the inductive proof, assume that are looking at the largest block in the decomposition, that is $m < 2^{k+1}$, and let $u = \Big\lfloor \lg(m - 2^k) \Big\rfloor < k$. There must be a block of size u in the decomposition of m. Namely, we have two blocks that we known in the decomposition of m, of sizes 2^k and 2^u. Note, that these two blocks are the largest blocks in the decomposition of m. In particular, $2^k + 2 * 2^u > m$, implying that $2^{u+1} + 2^k - m > 0$.

Let Y be the random variable which is the number of bits output by the extractor algorithm.

By Lemma 26.1.7, since $\frac{m-2^k}{m} < 1$, we have

$$\frac{m-2^k}{m} \leq \frac{m-2^k+\left(2^{u+1}+2^k-m\right)}{m \qquad +(2^{u+1}+2^k-m)} = \frac{2^{u+1}}{2^{u+1}+2^k}.$$

Thus, by induction (we assume the claim holds for all integers smaller than m), we have

$$\begin{aligned}
\mathbf{E}[Y] &\geq \frac{2^k}{m}k + \frac{m-2^k}{m}\left(\underbrace{\left\lfloor \lg(m-2^k)\right\rfloor}_{u} - 1\right) = \frac{2^k}{m}k + \frac{m-2^k}{m}(\underbrace{k-k}_{=0}+u-1) \\
&= k + \frac{m-2^k}{m}(u-k-1) \\
&\geq k + \frac{2^{u+1}}{2^{u+1}+2^k}\left(u-k-1\right) = k - \frac{2^{u+1}}{2^{u+1}+2^k}(1+k-u)\,,
\end{aligned}$$

since $u-k-1 \leq 0$ as $k > u$. If $u = k-1$, then $\mathbf{E}[Y] \geq k - \frac{1}{2}\cdot 2 = k-1$, as required. If $u = k-2$ then $\mathbf{E}[Y] \geq k - \frac{1}{3}\cdot 3 = k-1$. Finally, if $u < k-2$ then

$$\mathbf{E}[Y] \geq k - \frac{2^{u+1}}{2^k}(1+k-u) = k - \frac{k-u+1}{2^{k-u-1}} = k - \frac{2+(k-u-1)}{2^{k-u-1}} \geq k-1,$$

since $(2+i)/2^i \leq 1$ for $i \geq 2$. ■

Chapter 27

Even more on Entropy, Randomness, and Information

> "It had been that way even before, when for years at a time he had not seen blue sky, and each second of those years could have been his last. But it did not benefit an Assualtman to think about death. Though on the other hand you had to think a lot about possible defeats. Gorbovsky had once said that death is worse than any defeat, even the most shattering. Defeat was always really only an accident, a setback which you could surmount. You had to surmount it. Only the dead couldn't fight on."
>
> – – Defeat, Arkady and Boris Strugatsky.

27.1 Extracting randomness

27.1.1 Enumerating binary strings with j ones

Consider a binary string of length n with j ones. $S(n, j)$ denote the set of all such binary strings. There are $\binom{n}{j}$ such strings. For the following, we need an algorithm that given a string U of n bits with j ones, maps it into a number in the range $0, \ldots, \binom{n}{j} - 1$.

To this end, consider the full binary tree $\mathcal{T}$ of height n. Each leaf, encodes a string of length n, and mark each leaf that encodes a string of $S(n, j)$. Consider a node v in the tree, that is of height k; namely, the path π_v from the root of $\mathcal{T}$ to v is of length k. Furthermore, assume there are m ones written on the path π_v. Clearly, any leaf in the subtree of v that is in $S(n, j)$ is created by selecting $j - m$ ones in the remaining $n - k$ positions. The number of possibilities to do so is $\binom{n-k}{j-m}$. Namely, given a node v in this tree $\mathcal{T}$, we can quickly compute the number of elements of $S(n, j)$ stored in this subtree.

As such, let traverse $\mathcal{T}$ using a standard **DFS** algorithm, which would always first visit the '0' child before the '1' child, and use it to enumerate the marked leaves. Now, given a string x of S_j, we would like to compute what number would be assigned to by the above **DFS** procedure. The key observation is that calls made by the **DFS** on nodes that are not on the path, can be skipped by just computing directly how many marked leaves are there

in the subtrees on this nodes (and this we can do using the above formula). As such, we can compute the number assigned to x in linear time.

The cool thing about this procedure, is that we do not need $\mathcal{T}$ to carry it out. We can think about $\mathcal{T}$ as being a virtual tree.

Formally, given a string x made out of n bits, with j ones, we can in $O(n)$ time map it to an integer in the range $0, \ldots, \binom{n}{j} - 1$, and this mapping is one-to-one. Let **EnumBinomCoeffAlg** denote this procedure.

27.1.2 Extracting randomness

Theorem 27.1.1. *Consider a coin that comes up heads with probability $p > 1/2$. For any constant $\delta > 0$ and for n sufficiently large:*

(A) One can extract, from an input of a sequence of n flips, an output sequence of $(1-\delta)n\mathbb{H}(p)$ (unbiased) independent random bits.

(B) One can not extract more than $n\mathbb{H}(p)$ bits from such a sequence.

Proof: There are $\binom{n}{j}$ input sequences with exactly j heads, and each has probability $p^j(1-p)^{n-j}$. We map this sequence to the corresponding number in the set $S_j = \left\{0, \ldots, \binom{n}{j} - 1\right\}$. Note, that this, conditional distribution on j, is uniform on this set, and we can apply the extraction algorithm of Theorem 26.1.8 to S_j. Let Z be the random variables which is the number of heads in the input, and let B be the number of random bits extracted. We have

$$\mathbf{E}[B] = \sum_{k=0}^{n} \mathbf{Pr}[Z = k]\, \mathbf{E}\Big[B \;\Big|\; Z = k\Big],$$

and by Theorem 26.1.8, we have $\mathbf{E}\Big[B \;\Big|\; Z = k\Big] \geq \left\lfloor \lg \binom{n}{k} \right\rfloor - 1$. Let $\varepsilon < p - 1/2$ be a constant to be determined shortly. For $n(p-\varepsilon) \leq k \leq n(p+\varepsilon)$, we have

$$\binom{n}{k} \geq \binom{n}{\lfloor n(p+\varepsilon) \rfloor} \geq \frac{2^{n\mathbb{H}(p+\varepsilon)}}{n+1},$$

by Corollary 26.1.5 (iii). We have

$$\begin{aligned}
\mathbf{E}\Big[B\Big] &\geq \sum_{k=\lfloor n(p-\varepsilon)\rfloor}^{\lceil n(p+\varepsilon)\rceil} \mathbf{Pr}\Big[Z = k\Big]\, \mathbf{E}\Big[B \;\Big|\; Z = k\Big] \geq \sum_{k=\lfloor n(p-\varepsilon)\rfloor}^{\lceil n(p+\varepsilon)\rceil} \mathbf{Pr}\Big[Z = k\Big] \left(\left\lfloor \lg \binom{n}{k} \right\rfloor - 1 \right) \\
&\geq \sum_{k=\lfloor n(p-\varepsilon)\rfloor}^{\lceil n(p+\varepsilon)\rceil} \mathbf{Pr}\Big[Z = k\Big] \left(\lg \frac{2^{n\mathbb{H}(p+\varepsilon)}}{n+1} - 2 \right) \\
&= \Big(n\mathbb{H}(p+\varepsilon) - \lg(n+1) - 2\Big)\, \mathbf{Pr}[|Z - np| \leq \varepsilon n] \\
&\geq \Big(n\mathbb{H}(p+\varepsilon) - \lg(n+1) - 2\Big) \left(1 - 2\exp\left(-\frac{n\varepsilon^2}{4p}\right)\right),
\end{aligned}$$

since $\mu = \mathbf{E}[Z] = np$ and $\mathbf{Pr}\Big[|Z - np| \geq \frac{\varepsilon}{p} pn\Big] \leq 2\exp\Big(-\frac{np}{4}\Big(\frac{\varepsilon}{p}\Big)^2\Big) = 2\exp\Big(-\frac{n\varepsilon^2}{4p}\Big)$, by the Chernoff inequality. In particular, fix $\varepsilon > 0$, such that $\mathbb{H}(p+\varepsilon) > (1-\delta/4)\mathbb{H}(p)$, and since p is fixed $n\mathbb{H}(p) = \Omega(n)$, in particular, for n sufficiently large, we have $-\lg(n+1) \geq -\frac{\delta}{10} n\mathbb{H}(p)$. Also, for n sufficiently large, we have $2\exp\Big(-\frac{n\varepsilon^2}{4p}\Big) \leq \frac{\delta}{10}$. Putting it together, we have that for n large enough, we have

$$\mathbf{E}[B] \;\geq\; \left(1 - \frac{\delta}{4} - \frac{\delta}{10}\right) n\mathbb{H}(p) \left(1 - \frac{\delta}{10}\right) \geq (1-\delta)\, n\mathbb{H}(p)\,,$$

as claimed.

As for the upper bound, observe that if an input sequence x has probability $\mathbf{Pr}[X = x]$, then the output sequence $y = \mathbf{Ext}(x)$ has probability to be generated which is at least $\mathbf{Pr}[X = x]$. Now, all sequences of length $|y|$ have equal probability to be generated. Thus, we have the following (trivial) inequality

$$2^{|\mathbf{Ext}(x)|}\, \mathbf{Pr}[X = x] \leq 2^{|\mathbf{Ext}(x)|}\, \mathbf{Pr}[y = \mathbf{Ext}(x)] \leq 1,$$

implying that $|\mathbf{Ext}(x)| \leq \lg(1/\mathbf{Pr}[X = x])$. Thus,

$$\mathbf{E}\Big[B\Big] = \sum_x \mathbf{Pr}\Big[X = x\Big]\, |\mathbf{Ext}(x)| \leq \sum_x \mathbf{Pr}\Big[X = x\Big] \lg \frac{1}{\mathbf{Pr}[X = x]} = \mathbb{H}(X)\,.$$

■

27.2 Bibliographical Notes

The presentation here follows [MU05, Sec. 9.1-Sec 9.3].

Chapter 28

Shannon's theorem

> "This has been a novel about some people who were punished entirely too much for what they did. They wanted to have a good time, but they were like children playing in the street; they could see one after another of them being killed - run over, maimed, destroyed - but they continued to play anyhow. We really all were very happy for a while, sitting around not toiling but just bullshitting and playing, but it was for such a terrible brief time, and then the punishment was beyond belief; even when we could see it, we could not believe it."
>
> – – A Scanner Darkly, Philip K. Dick.

28.1 Coding: Shannon's Theorem

We are interested in the problem sending messages over a noisy channel. We will assume that the channel noise is behave "nicely".

Definition 28.1.1. The input to a ***binary symmetric channel*** with parameter p is a sequence of bits $x_1, x_2, \ldots$, and the output is a sequence of bits $y_1, y_2, \ldots$, such that $\mathbf{Pr}[x_i = y_i] = 1 - p$ independently for each i.

Translation: Every bit transmitted have the same probability to be flipped by the channel. The question is how much information can we send on the channel with this level of noise. Naturally, a channel would have some capacity constraints (say, at most 4,000 bits per second can be sent on the channel), and the question is how to send the largest amount of information, so that the receiver can recover the original information sent.

Now, its important to realize that handling noise is unavoidable in the real world. Furthermore, there are tradeoffs between channel capacity and noise levels (i.e., we might be able to send considerably more bits on the channel but the probability of flipping [i.e., p] might be much larger). In designing a communication protocol over this channel, we need to figure out where is the optimal choice as far as the amount of information sent.

Definition 28.1.2. A (k, n) ***encoding function*** $\texttt{Enc} : \{0,1\}^k \to \{0,1\}^n$ takes as input a sequence of k bits and outputs a sequence of n bits. A (k, n) ***decoding function*** $\texttt{Dec} : \{0,1\}^n \to \{0,1\}^k$ takes as input a sequence of n bits and outputs a sequence of k bits.

Thus, the sender would use the encoding function to send its message, and the receiver would use the transmitted string (with the noise in it), to recover the original message. Thus, the sender starts with a message with k bits, it blow it up to n bits, using the encoding function (to get some robustness to noise), it send it over the (noisy) channel to the receiver. The receiver takes the given (noisy) message with n bits, and use the decoding function to recover the original k bits of the message.

Naturally, we would like k to be as large as possible (for a fixed n), so that we can send as much information as possible on the channel.

The following celebrated result of Shannon[①] in 1948 states exactly how much information can be sent on such a channel.

Theorem 28.1.3 (Shannon's theorem). *For a binary symmetric channel with parameter $p < 1/2$ and for any constants $\delta, \gamma > 0$, where n is sufficiently large, the following holds:*

(i) For an $k \leq n(1 - \mathbb{H}(p) - \delta)$ there exists (k, n) encoding and decoding functions such that the probability the receiver fails to obtain the correct message is at most γ for every possible k-bit input messages.

(ii) There are no (k, n) encoding and decoding functions with $k \geq n(1 - \mathbb{H}(p) + \delta)$ such that the probability of decoding correctly is at least γ for a k-bit input message chosen uniformly at random.

28.1.0.1 Intuition behind Shanon's theorem

Let assume the senders has sent a string $S = s_1 s_2 \ldots s_n$. The receiver got a string $T = t_1 t_2 \ldots t_n$, where $p = \mathbf{Pr}[t_i \neq s_i]$, for all i. In particular, let U be the Hamming distance between S and T; that is, $U = \sum_i \Big[s_i \neq t_i \Big]$. Under our assumptions $\mathbf{E}[U] = pn$, and U is a binomial variable. By Chernoff inequality, we know that $U \in \Big[(1-\delta)np, (1+\delta)np\Big]$ with high probability, where δ is some tiny constant. So lets assume this indeed happens. This means that T is in a ring R centered at S, with inner radius $(1-\delta)np$ and outer radius $(1+\delta)np$. This ring has

$$\sum_{i=(1-\delta)np}^{(1+\delta)np} \binom{n}{i} \leq 2 \binom{n}{(1+\delta)np} \leq \alpha = 2 \cdot 2^{n\mathbb{H}((1+\delta)p)}.$$

Let us pick as many rings as possible in the hypercube so that they are disjoint: $R_1, \ldots, R_\kappa$. If somehow magically, every word in the hypercube would be covered, then we could use all the possible 2^n codewords, then the number of rings κ we would pick would be at least

$$\kappa \geq \frac{2^n}{|R|} \geq \frac{2^n}{2 \cdot 2^{n\mathbb{H}((1+\delta)p)}} \approx 2^{n(1-\mathbb{H}((1+\delta)p))}.$$

In particular, consider all possible strings of length k such that $2^k \leq \kappa$. We map the ith string in $\{0,1\}^k$ to the center C_i of the ith ring R_i. Assuming that when we send C_i, the

[①] Claude Elwood Shannon (April 30, 1916 - February 24, 2001), an American electrical engineer and mathematician, has been called "the father of information theory".

receiver gets a string in R_i, then the decoding is easy - find the ring R_i containing the received string, take its center string C_i, and output the original string it was mapped to. Now, observe that

$$k = \lfloor \log \kappa \rfloor = n\Big(1 - \mathbb{H}\left((1+\delta)p\right)\Big) \approx n(1 - \mathbb{H}(p)),$$

as desired.

28.1.0.2 What is wrong with the above?

The problem is that we can not find such a large set of disjoint rings. The reason is that when you pack rings (or balls) you are going to have wasted spaces around. To overcome this, we would allow rings to overlap somewhat. That makes things considerably more involved. The details follow.

28.2 Proof of Shannon's theorem

The proof is not hard, but requires some care, and we will break it into parts.

28.2.1 How to encode and decode efficiently

28.2.1.1 The scheme

Our scheme would be simple. Pick $k \leq n(1-\mathbb{H}(p)-\delta)$. For any number $i = 0, \ldots, \widehat{K} = 2^{k+1}-1$, randomly generate a binary string Y_i made out of n bits, each one chosen independently and uniformly. Let $Y_0, \ldots, Y_{\widehat{K}}$ denote these code words. Here, we have

$$\widehat{K} = 2^{n(1-\mathbb{H}(p)-\delta)}.$$

For each of these codewords we will compute the probability that if we send this codeword, the receiver would fail. Let $X_0, \ldots, X_K$, where $K = 2^k - 1$, be the K codewords with the lowest probability to fail. We assign these words to the 2^k messages we need to encode in an arbitrary fashion.

The decoding of a message w is done by going over all the codewords, and finding all the codewords that are in (Hamming) distance in the range $[p(1-\varepsilon)n, p(1+\varepsilon)n]$ from w. If there is only a single word X_i with this property, we return i as the decoded word. Otherwise, if there are no such words or there is more than one word, the decoder stops and report an error.

28.2.1.2 The proof

Intuition. Let S_i be all the binary strings (of length n) such that if the receiver gets this word, it would decipher it to be i (here are still using the extended codeword $Y_0, \ldots, Y_{\widehat{K}}$). Note, that if we remove some codewords from consideration, the set S_i just increases in size.

Let W_i be the probability that X_i was sent, but it was not deciphered correctly. Formally, let r denote the received word. We have that

$$W_i = \sum_{r \notin S_i} \mathbf{Pr}\Big[r \text{ received when } X_i \text{ was sent}\Big].$$

To bound this quantity, let $\Delta(x, y)$ denote the Hamming distance between the binary strings x and y. Clearly, if x was sent the probability that y was received is

$$w(x, y) = p^{\Delta(x,y)}(1-p)^{n-\Delta(x,y)}.$$

As such, we have

$$\mathbf{Pr}[r \text{ received when } X_i \text{ was sent}] = w(X_i, r).$$

Let $\overline{S_{i,r}}$ be an indicator variable which is 1 if $r \notin S_i$. We have that

$$W_i = \sum_{r \notin S_i} \mathbf{Pr}[r \text{ received when } X_i \text{ was sent}] = \sum_{r \notin S_i} w(X_i, r) = \sum_r \overline{S_{i,r}} w(X_i, r).$$

The value of W_i is a random variable of our choice of $Y_0, \ldots, Y_{\widehat{K}}$. As such, its natural to ask what is the expected value of W_i.

Consider the ring

$$R(r) = \Big\{ x \;\Big|\; (1-\varepsilon)np \le \Delta(x, r) \le (1+\varepsilon)np \Big\},$$

where $\varepsilon > 0$ is a small enough constant. Suppose, that the code word Y_i was sent, and r was received. The decoder return i if Y_i is the only codeword that falls inside $R(r)$.

Lemma 28.2.1. *Given that Y_i was sent, and r was received and furthermore $r \in R(Y_i)$, then the probability of the decoder failing, is*

$$\tau = \mathbf{Pr}\Big[r \notin S_i \;\Big|\; r \in R(Y_i)\Big] \le \frac{\gamma}{8},$$

where γ is the parameter of Theorem 28.1.3.

Proof: The decoder fails here, only if $R(r)$ contains some other codeword Y_j ($j \neq i$) in it. As such,

$$\tau = \mathbf{Pr}\Big[r \notin S_i \;\Big|\; r \in R(Y_i)\Big] \le \mathbf{Pr}[Y_j \in R(r), \text{ for any } j \neq i] \le \sum_{j \neq i} \mathbf{Pr}[Y_j \in R(r)].$$

Now, we remind the reader that the Y_js are generated by picking each bit randomly and independently, with probability $1/2$. As such, we have

$$\mathbf{Pr}[Y_j \in R(r)] = \sum_{m=(1-\varepsilon)np}^{(1+\varepsilon)np} \frac{\binom{n}{m}}{2^n} \le \frac{n}{2^n} \binom{n}{\lfloor (1+\varepsilon)np \rfloor},$$

since $(1+\varepsilon)p < 1/2$ (for ε sufficiently small), and as such the last binomial coefficient in this summation is the largest. By Corollary 26.1.5 (i), we have

$$\mathbf{Pr}\Big[Y_j \in R(r)\Big] \leq \frac{n}{2^n}\binom{n}{\lfloor (1+\varepsilon)np \rfloor} \leq \frac{n}{2^n} 2^{n\mathbb{H}((1+\varepsilon)p)} = n2^{n(\mathbb{H}((1+\varepsilon)p)-1)}.$$

As such, we have

$$\begin{aligned}
\tau = \mathbf{Pr}\Big[r \notin S_i \;\Big|\; r \in R(Y_i)\Big] &\leq \sum_{j\neq i} \mathbf{Pr}\Big[Y_j \in R(r)\Big] \\
&\leq \widehat{K}\, \mathbf{Pr}\Big[Y_1 \in R(r)\Big] \leq 2^{k+1} n 2^{n(\mathbb{H}((1+\varepsilon)p)-1)} \\
&\leq n2^{n(1-\mathbb{H}(p)-\delta)+1} n 2^{n(\mathbb{H}((1+\varepsilon)p)-1)} \leq n2^{n\Big(\mathbb{H}((1+\varepsilon)p)-\mathbb{H}(p)-\delta\Big)+1}
\end{aligned}$$

since $k \leq n(1-\mathbb{H}(p)-\delta)$. Now, we choose ε to be a small enough constant, so that the quantity $\mathbb{H}\left((1+\varepsilon)p\right) - \mathbb{H}(p) - \delta$ is equal to some (absolute) negative (constant), say $-\beta$, where $\beta > 0$. Then, $\tau \leq n2^{-\beta n+1}$, and choosing n large enough, we can make τ smaller than $\gamma/2$, as desired. As such, we just proved that

$$\tau = \mathbf{Pr}\Big[r \notin S_i \;\Big|\; r \in R(Y_i)\Big] \leq \frac{\gamma}{2}.$$

■

Lemma 28.2.2. *We have, that $\sum_{r\notin R(Y_i)} w(Y_i, r) \leq \gamma/8$, where γ is the parameter of Theorem 28.1.3.*

Proof: This quantity, is the probability of sending Y_i when every bit is flipped with probability p, and receiving a string r such that more than εpn bits where flipped. But this quantity can be bounded using the Chernoff inequality. Let $Z = \Delta(Y_i, r)$, and observe that $\mathbf{E}[Z] = pn$, and it is the sum of n independent indicator variables. As such

$$\sum_{r\notin R(Y_i)} w(Y_i, r) = \mathbf{Pr}[|Z - \mathbf{E}[Z]| > \varepsilon pn] \leq 2\exp\left(-\frac{\varepsilon^2}{4} pn\right) < \frac{\gamma}{4},$$

since ε is a constant, and for n sufficiently large. ■

Lemma 28.2.3. *For any i, we have $\mu = \mathbf{E}[W_i] \leq \gamma/4$, where γ is the parameter of Theorem 28.1.3.*

Proof: By linearity of expectations, we have

$$\begin{aligned}
\mu = \mathbf{E}[W_i] = \mathbf{E}\left[\sum_r \overline{S_{i,r}} w(Y_i, r)\right] &= \sum_r \mathbf{E}\Big[\overline{S_{i,r}} w(Y_i, r)\Big] \\
= \sum_r \mathbf{E}\Big[\overline{S_{i,r}}\Big]\, w(Y_i, r) &= \sum_r \mathbf{Pr}[x \notin S_i]\, w(Y_i, r),
\end{aligned}$$

since $\overline{S_{i,r}}$ is an indicator variable. Setting, $\tau = \mathbf{Pr}\Big[r \notin S_i \;\Big|\; r \in R(Y_i)\Big]$ and since $\sum_r w(Y_i, r) = 1$, we get

$$\begin{aligned}
\mu &= \sum_{r \in R(Y_i)} \mathbf{Pr}[x \notin S_i]\, w(Y_i, r) + \sum_{r \notin R(Y_i)} \mathbf{Pr}[x \notin S_i]\, w(Y_i, r) \\
&= \sum_{r \in R(Y_i)} \mathbf{Pr}\Big[x \notin S_i \;\Big|\; r \in R(Y_i)\Big]\, w(Y_i, r) + \sum_{r \notin R(Y_i)} \mathbf{Pr}[x \notin S_i]\, w(Y_i, r) \\
&\leq \sum_{r \in R(Y_i)} \tau \cdot w(Y_i, r) + \sum_{r \notin R(Y_i)} w(Y_i, r) \leq \tau + \sum_{r \notin R(Y_i)} w(Y_i, r) \leq \frac{\gamma}{4} + \frac{\gamma}{4} = \frac{\gamma}{2}.
\end{aligned}$$

Now, the receiver got r (when we sent Y_i), and it would miss encode it only if (i) r is outside of $R(Y_i)$, or $R(r)$ contains some other codeword Y_j ($j \neq i$) in it. As such,

$$\tau = \mathbf{Pr}\Big[r \notin S_i \;\Big|\; r \in R(Y_i)\Big] \leq \mathbf{Pr}[Y_j \in R(r), \text{ for any } j \neq i] \leq \sum_{j \neq i} \mathbf{Pr}[Y_j \in R(r)]\,.$$

Now, we remind the reader that the Y_js are generated by picking each bit randomly and independently, with probability $1/2$. As such, we have

$$\mathbf{Pr}[Y_j \in R(r)] = \sum_{m=(1-\varepsilon)np}^{(1+\varepsilon)np} \frac{\binom{n}{m}}{2^n} \leq \frac{n}{2^n} \binom{n}{\lfloor (1+\varepsilon)np \rfloor},$$

since $(1+\varepsilon)p < 1/2$ (for ε sufficiently small), and as such the last binomial coefficient in this summation is the largest. By Corollary 26.1.5 (i), we have

$$\mathbf{Pr}[Y_j \in R(r)] \leq \frac{n}{2^n} \binom{n}{\lfloor (1+\varepsilon)np \rfloor} \leq \frac{n}{2^n} 2^{n\mathbb{H}((1+\varepsilon)p)} = n 2^{n(\mathbb{H}((1+\varepsilon)p)-1)}.$$

As such, we have

$$\begin{aligned}
\tau = \mathbf{Pr}\Big[r \notin S_i \;\Big|\; r \in R(Y_i)\Big] &\leq \sum_{j \neq i} \mathbf{Pr}[Y_j \in R(r)]\,. \leq \widehat{K}\, \mathbf{Pr}[Y_1 \in R(r)] \leq 2^{k+1} n 2^{n(\mathbb{H}((1+\varepsilon)p)-1)} \\
&\leq n 2^{n(1-\mathbb{H}(p)-\delta)+1+n(\mathbb{H}((1+\varepsilon)p)-1)} \leq n 2^{n\Big(\mathbb{H}((1+\varepsilon)p)-\mathbb{H}(p)-\delta\Big)+1}
\end{aligned}$$

since $k \leq n(1 - \mathbb{H}(p) - \delta)$. Now, we choose ε to be a small enough constant, so that the quantity $\mathbb{H}\left((1+\varepsilon)p\right) - \mathbb{H}(p) - \delta$ is negative (constant). Then, choosing n large enough, we can make τ smaller than $\gamma/2$, as desired. As such, we just proved that

$$\tau = \mathbf{Pr}\Big[r \notin S_i \;\Big|\; r \in R(Y_i)\Big] \leq \frac{\gamma}{2}.$$

■

In the following, we need the following trivial (but surprisingly deep) observation.

Observation 28.2.4. *For a random variable X, if $\mathbf{E}[X] \leq \psi$, then there exists an event in the probability space, that assigns X a value $\leq \mu$.*

This holds, since $\mathbf{E}[X]$ is just the average of X over the probability space. As such, there must be an event in the universe where the value of X does not exceed its average value.

The above observation is one of the main tools in a powerful technique to proving various claims in mathematics, known as the ***probabilistic method***.

Lemma 28.2.5. *For the codewords $X_0, \ldots, X_K$, the probability of failure in recovering them when sending them over the noisy channel is at most γ.*

Proof: We just proved that when using $Y_0, \ldots, Y_{\widehat{K}}$, the expected probability of failure when sending Y_i, is $\mathbf{E}[W_i] \leq \gamma_2$, where $\widehat{K} = 2^{k+1} - 1$. As such, the expected total probability of failure is

$$\mathbf{E}\left[\sum_{i=0}^{\widehat{K}} W_i\right] = \sum_{i=0}^{\widehat{K}} \mathbf{E}[W_i] \leq \frac{\gamma}{2} 2^{k+1} = \gamma 2^k,$$

by Lemma 28.2.3 (here we are using the facts that all the random variables we have are symmetric and behave in the same way). As such, by Observation 28.2.4, there exist a choice of Y_is, such that

$$\sum_{i=0}^{\widehat{K}} W_i \leq 2^k \gamma.$$

Now, we use a similar argument used in proving Markov's inequality. Indeed, the W_i are always positive, and it can not be that 2^k of them have value larger than γ, because in the summation, we will get that

$$\sum_{i=0}^{\widehat{K}} W_i > 2^k \gamma.$$

Which is a contradiction. As such, there are 2^k codewords with failure probability smaller than γ. We set our 2^k codeword to be these words. Since we picked only a subset of the codewords for our code, the probability of failure for each codeword shrinks, and is at most γ. ■

Lemma 28.2.5 concludes the proof of the constructive part of Shannon's theorem.

28.2.2 Lower bound on the message size

We omit the proof of this part.

28.3 Bibliographical Notes

The presentation here follows [MU05, Sec. 9.1-Sec 9.3].

Part XII

Matchings

Chapter 29

Matchings

29.1 Definitions

Definition 29.1.1. For a graph $G = (V, E)$ a set $M \subseteq E$ of edges is a matching if no pair of edges of M has a common vertex.

A matching is perfect if it covers all the vertices of G. For a weight function w, which assigns real weight to the edges of G, a matching M is a maximal weight matching, if M is a matching and $w(M) = \sum_{e \in M} w(e)$ is maximal.

Definition 29.1.2. If there is no weight on the edges, we consider the weight of every edge to be one, and in this case, we are trying to compute a maximum size matching.

Problem 29.1.3. Given a graph G and a weight function on the edges, compute the maximum weight matching in G.

29.2 Unweighted matching in a bipartite graph

We remind the reader that there is a simple way to do a matching in a bipartite graph using network flow. Since this was already covered, we will not repeat it here.

29.3 Matchings and Alternating Paths

Consider a matching M. An edge $e \in M$ is a ***matching edge***. Naturally, Any edge $e' \in E(G) \setminus M$ is ***free***. In particular, a vertex $v \in V(G)$ is *matched* if it is adjacent to an edge in M. Naturally, a vertex v' which is not matched is ***free***.

An ***alternating path*** is a simple path that its edges are alternately matched and free. An ***alternating cycle*** is defined similarly. The *length* of a path/cycle is the number of edges in it.

For an alternating path/cycle π, its ***weight*** is

$$\gamma(\pi, M) = \sum_{e \in \pi \setminus M} w(e) - \sum_{e \in \pi \cap M} w(e). \tag{29.1}$$

Namely, it is the total weight of the free edges in π minus the weight of the matched edges. This is a natural concept because of the following lemma.

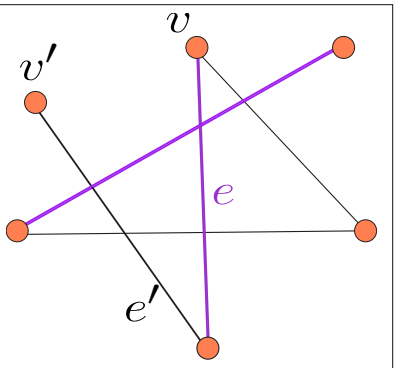


Figure 29.1: The edge e is in the matching, while e' is free.

Lemma 29.3.1. *Let M be a matching, and let π be an alternating path/cycle with positive weight such that*

$$M' = M \oplus \pi = (M \setminus \pi) \cup (\pi \setminus M)$$

is a matching, then $w(M')$ is bigger; namely, $w(M') > w(M)$.

Proof: Just observe that $w(M') = w(M) + \gamma(\pi, M)$. ■

Definition 29.3.2. An alternating path is ***augmenting*** if it starts and ends in a free vertex.

Observation 29.3.3. *If M has an augmenting path π then M is not of maximum size matching (this is for the unweighted case), since $M \oplus \pi$ is a larger matching.*

Theorem 29.3.4. *Let M be a matching, and T be a maximum size matching, and $k = |T| - |M|$. Then M has k vertex disjoint augmenting paths. At least one of length $\leq n/k - 1$.*

Proof: Let $E' = M \oplus T$, and let $H = (V, E')$. Clearly, every vertex in H has at most degree 2 because every vertex is adjacent to at most one edge of M and one edge of T. Thus, H is a collection of paths and (even length) cycles. The cycles are of even length since the edges of the cycle are alternating between two matchings (i.e., you can think about the cycle edges as being 2-colorable).

Now, there are k more edges of T in $M \oplus T$ than of M. Every cycle have the same number of edges of M and T. Thus, a path in H can have at most one more edge of T than of M. In such a case, this path is an augmenting path for M. It follows that there are at least k augmenting paths for M in H.

As for the claim on the length of the shortest augmenting path, observe that if all these (vertex disjoint) augmenting paths were of length $\geq n/k$ then the total number of vertices in H would be at least $(n/k + 1)k > n$. A contradiction. ■

Theorem 29.3.5. *Let M be a matching of maximum weight among matchings of size $|M|$. Let π be an augmenting path for M of maximum weight, and let T be the matching formed by augmenting M using π. Then T is of maximum weight among matchings of size $|M|+1$.*

Proof: Let S be a matching of maximum weight among all matchings with $|M|+1$ edges. And consider $H = (V, M \oplus S)$.

Consider a cycle σ in H. The weight $\gamma(\sigma, M)$ (see Eq. (29.1)) must be zero. Indeed, if $\gamma(\sigma, M) > 0$ then $M \oplus \sigma$ is a matching of the same size as M which is heavier than M. A contradiction to the definition of M as the maximum weight such matching.

Similarly, if $\gamma(\sigma, M) < 0$ than $\gamma(\sigma, S) = -\gamma(\sigma, M)$ and as such $S \oplus \sigma$ is heavier than S. A contradiction.

By the same argumentation, if σ is a path of even length in the graph H then $\gamma(\sigma, M) = 0$ by the same argumentation.

Let U_S be all the odd length paths in H that have one edge more in S than in M, and similarly, let U_M be the odd length paths in H that have one edge more of M than an edge of S.

We know that $|U_S| - |U_M| = 1$ since S has one more edge than M. Now, consider a path $\pi \in U_S$ and a path $\pi' \in U_M$. It must be that $\gamma(\pi, M) + \gamma(\pi', M) = 0$. Indeed, if $\gamma(\pi, M) + \gamma(\pi', M) > 0$ then $M \oplus \pi \oplus \pi'$ would have bigger weight than M while having the same number of edges. Similarly, if $\gamma(\pi, M) + \gamma(\pi', M) < 0$ (compared to M) then $S \oplus \pi \oplus \pi'$ would have the same number of edges as S while being a heavier matching. A contradiction.

Thus, $\gamma(\pi, M) + \gamma(\pi', M) = 0$. Thus, we can pair up the paths in U_S to paths in U_M, and the total weight of such a pair is zero, by the above argumentation. There is only one path μ in U_S which is not paired, and it must be that $\gamma(\mu, M) = w(S) - w(M)$ (since everything else in H has zero weight as we apply it to M to get S).

This establishes the claim that we can augment M with a single path to get a maximum weight matching of cardinality $|M|+1$. Clearly, this path must be the heaviest augmenting path that exists for M. Otherwise, there would be a heavier augmenting path σ' for M such that $w(M \oplus \sigma') > w(S)$. A contradiction to the maximality of S. ∎

The above theorem imply that if we always augment along the maximum weight augmenting path, than we would get the maximum weight matching in the end.

29.4 Maximum Weight Matchings in A Bipartite Graph

Let $G = (L \cup R, E)$ be the given bipartite graph, with $w : E \to \mathbb{R}$ be the non-negative weight function. Given a matching M we define the graph G_M to be the directed graph, where if $rl \in M$, $l \in L$ and $r \in R$ then we add $(r \to l)$ to $E(G_M)$ with weight $\alpha((r \to l)) = w(rl)$ Similarly, if $rl \in E \setminus M$ then add the edge $(l \to r) \in E(G_M)$ to G_M and set $\alpha((l \to r)) = -w(rl)$

Namely, we direct all the matching edges from right to left, and assign them their weight, and we direct all other edges from left to right, with their negated weight. Let G_M denote the resulting graph.

An augmenting path π in G must have an odd number of edges. Since G is bipartite, π must have one endpoint on the left side and one endpoint on the right side. Observe, that a path π in G_M has weight $\alpha(\pi) = -\gamma(\pi, M)$.

Let U_L be all the unmatched vertices in L and let U_R be all the unmatched vertices in R.

Thus, what we are looking for is a path π in G_M starting U_L going to U_R with maximum weight $\gamma(\pi)$, namely with minimum weight $\alpha(\pi)$.

Lemma 29.4.1. *If M is a maximum weight matching with k edges in G, than there is no negative cycle in G_M where $\alpha(\cdot)$ is the associated weight function.*

Proof: Assume for the sake of contradiction that there is a cycle C, and observe that $\gamma(C) = -\alpha(C) > 0$. Namely, $M \oplus C$ is a new matching with bigger weight and the same number of edges. A contradiction to the maximality of M. ■

The algorithm. So, we now can find a maximum weight in the bipartite graph G as follows: Find a maximum weight matching M with k edges, compute the maximum weight augmenting path for M, apply it, and repeat till M is maximal.

Thus, we need to find a minimum weight path in G_M between U_L and U_R (because we flip weights). This is just computing a shortest path in the graph G_M which does not have negative cycles, and this can just be done by using the **Bellman-Ford** algorithm. Indeed, collapse all the vertices of U_L into a single vertex, and all the uncovered vertices of U_R into a single vertex. Let H_M be the resulting graph. Clearly, we are looking for the shortest path between the two vertices corresponding to U_L and U_R in H_M and since this graph has no negative cycles, this can be done using the **Bellman-Ford** algorithm, which takes $O(nm)$ time. We conclude:

Lemma 29.4.2. *Given a bipartite graph G and a maximum weight matching M of size k one can find a maximum weight augmenting path for G in $O(nm)$ time, where n is the number of vertices of G and m is the number of edges.*

We need to apply this algorithm $n/2$ times at most, as such, we get:

Theorem 29.4.3. *Given a weight bipartite graph G, with n vertices and m edges, one can compute a maximum weight matching in G in $O(n^2 m)$ time.*

29.4.1 Faster Algorithm

It turns out, in fact, that the graph here is very special, and one can use the Dijkstra algorithm. We omit any further details, and just state the result. The interested student can figure out the details (warning: this is not easy).

Theorem 29.4.4. *Given a weight bipartite graph G, with n vertices and m edges, one can compute a maximum weight matching in G in $O(n(n \log n + m))$ time.*

29.5 The Bellman-Ford Algorithm - A Quick Reminder

The **Bellman-Ford** algorithm computes the shortest path from a single source s in a graph G that has no negative cycles to all the vertices in the graph. Here G has n vertices and m edges. The algorithm works by initializing all distances to the source to be ∞ (formally, for all $u \in V(G)$, we set $d[u] \leftarrow \infty$ and $d[s] \leftarrow 0$). Then, it n times scans all the edges, and for every edge $(u \to v) \in E(G)$ it performs a **Relax**(u, v) operation. The relax operation checks if $x = d[u] + w\Big((u \to v)\Big) < d[v]$, and if so, it updates $d[v]$ to x, where $d[u]$ denotes the current distance from s to u. Since **Relax**(u, v) operation can be performed in constant time, and we scan all the edges n times, it follows that the overall running time is $O(mn)$.

We claim that in the end of the execution of the algorithm the shortest path length from s to u is $d[u]$, for all $u \in V(G)$. Indeed, every time we scan the edges, we set at least one vertex distance to its final value (which is its shortest path length). More formally, all vertices that their shortest path to s have i edges, are being set to their shortest path length in the ith iteration of the algorithm, as can be easily proved by induction. This implies the claim.

Notice, that if we want to detect negative cycles, we can ran **Bellman-Ford** for an additional iteration. If the distances changes, we know that there is a negative cycle somewhere in the graph.

Chapter 30

Matchings II

30.1 Maximum Size Matching in a Non-Bipartite Graph

The results from the previous lecture suggests a natural algorithm for computing a maximum size (i.e., matching with maximum number of edges in it) matching in a general (i.e., not necessarily bipartite) graph. Start from an empty matching M and repeatedly find an augmenting path from an unmatched vertex to an unmatched vertex. Here we are discussing the unweighted case.

Notations. Let $\mathcal{T}$ be a given tree. For two vertices $x, y \in V(\mathcal{T})$, let τ_{xy} denote the path in $\mathcal{T}$ between x and y. For two paths π and π' that share an endpoint, let $\pi \mid\mid \pi'$ denotes the path resulting from concatenating π to π'. For a path π, let $|\pi|$ denote the number of edges in π.

30.1.1 Finding an augmenting path

We are given a graph G and a matching M, and we would to compute a bigger matching in G. We will do it by computing an augmenting path for M.

We first observe that if G has any edge with both endpoints being free, we can just add it to the current matching. Thus, in the following, we assume that for all edges, at least one of their endpoint is covered by the current matching M. Our task is to find an augmenting path in M.

We start by collapsing the unmatched vertices to a single vertex s, and let H be the resulting graph. Next, we compute an ***alternating BFS*** of H starting from s. Formally, we perform a BFS on H starting from s such that for the even levels of the tree the algorithm is allowed to traverse only edges in the matching M, and in odd levels the algorithm traverses the unmatched edges. Let $\mathcal{T}$ denote the resulting tree.

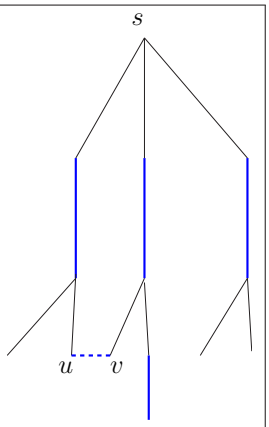


Figure 30.1: A cycle in the alternating BFS tree.

An augmenting path in G corresponds to an odd cycle in H with passing through the vertex s.

Definition 30.1.1. An edge $uv \in E(G)$ is a ***bridge*** if the the following conditions are met: (i) u and v have the same depth in $\mathcal{T}$, (ii) if the depth of u in $\mathcal{T}$ is even then uv is free (i.e., $uv \notin M$, and (iii) if the depth of u in $\mathcal{T}$ is odd then $uv \in M$.

Note, that given an edge uv we can check if it is a bridge in constant time after linear time preprocessing of $\mathcal{T}$ and G.

The following is an easy technical lemma.

Lemma 30.1.2. *Let v be a vertex of G, M a matching in G, and let π be the shortest alternating path between s and v in G. Furthermore, assume that for any vertex w of π the shortest alternating path between w and s is the path along π.*

Then, the depth $d_{\mathcal{T}}(v)$ of v in $\mathcal{T}$ is $|\pi|$.

Proof: By induction on $|\pi|$. For $|\pi| = 1$ the proof trivially holds, since then v is a neighbor of s in G, and as such it is a child of s in $\mathcal{T}$.

For $|\pi| = k$, consider the vertex just before v on π, and let us denote it by u. By induction, the depth of u in $\mathcal{T}$ is $k-1$. Thus, when the algorithm computing the alternating BFS visited u, it tried to hang v from it in the next iteration. The only possibility for failure is if the algorithm already hanged v in earlier iteration of the algorithm. But then, there exists a shorter alternating path from s to v, which is a contradiction. ■

Lemma 30.1.3. *If there is an augmenting path in G for a matching M, then there exists an edge $uv \in E(G)$ which is a bridge in $\mathcal{T}$.*

Proof: Let π be an augmenting path in G. The path π corresponds to a an odd length alternating cycle in H. Let σ be the shortest odd length alternating cycle in G (note that both edges in σ that are adjacent to s are unmatched).

For a vertex x of σ, let $d(x)$ be the length of the shortest alternating path between x and s in H. Similarly, let $d'(x)$ be the length of the shortest alternating path between s and x along σ. Clearly, $d(x) \leq d'(x)$, but we claim that in fact $d(x) = d'(x)$, for all $x \in \sigma$. Indeed, assume for the sake of contradiction that $d(x) < d'(x)$, and let π_1, π_2 be the two paths from x to s formed by σ. Let η be the shortest alternating path between s and x. We know that $|\eta| < |\pi_1|$ and $|\eta| < |\pi_2|$. It is now easy to verify that either $\pi_1 \mid\mid \eta$ or $\pi_2 \mid\mid \eta$ is an alternating cycle shorter than σ involving s, which is a contradiction.

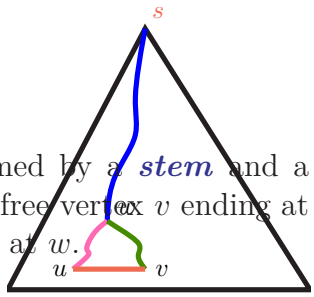
But then, take the two vertices of σ furthest away from s. Clearly, both of them have the same depth in $\mathcal{T}$, since $d(u) = d'(u) = d'(v) = d(v)$. By Lemma 30.1.2, we now have that $d_{\mathcal{T}}(u) = d(u) = d(v) = d_{\mathcal{T}}(v)$. Establishing the first part of the claim. See Figure 30.1.

As for the second claim, observe that it easily follows as σ is created from an alternating path. ■

Thus, we can do the following: Compute the alternating BFS $\mathcal{T}$ for H, and find a bridge uv in it. If M is not a maximal matching, then there exists an augmenting path for G and by Lemma 30.1.3 there exists a bridge. Computing the bridge uv takes $O(m)$ time.

Extract the paths from s to u and from s to v in $\mathcal{T}$, and glue them together with the edge uv to form an odd cycle μ in H; namely, $\mu = \tau_{su} \mid\mid uv \mid\mid \tau_{vs}$. If μ corresponds to an alternating path in G then we are done, since we found an alternating path, and we can apply it and find a bigger matching.

But μ, in fact, might have common edges. In particular, let π_{su} and π_{sv} be the two paths from s to u and v, respectively. Let w be the lowest vertex in $\mathcal{T}$ that is common to both π_{su} and π_{sv}.



Definition 30.1.4. Given a matching M, a ***flower*** for M is formed by a ***stem*** and a ***blossom***. The stem is an even length alternating path starting at a free vertex s ending at vertex w, and the blossom is an odd length (alternating) cycle based at w.

Lemma 30.1.5. *Consider a bridge edge $uv \in G$, and let w be the least common ancestor (LCA) of u and v in $\mathcal{T}$. Consider the path π_{sw} together with the cycle $C = \pi_{wu} \mid\mid uv \mid\mid \pi_{vw}$. Then π_{sw} and C together form a flower.*

Proof: Since only the even depth nodes in $\mathcal{T}$ have more than one child, w must be of even depth, and as such π_{sw} is of even length. As for the second claim, observe that $\alpha = |\pi_{wu}| = |\pi_{wv}|$ since the two nodes have the same depth in $\mathcal{T}$. In particular, $|C| = |\pi_{wu}| + |\pi_{wv}| + 1 = 2\alpha + 1$, which is an odd number. ■

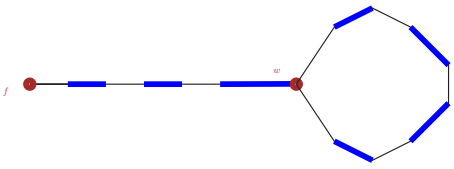
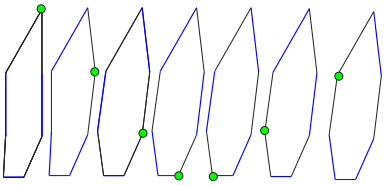


Figure 30.2: A blossom is made out of a ***stem*** (the path fw), and an odd length cycle which is the blossom. Together they form a flower.

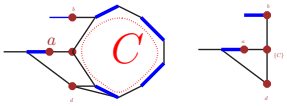
Let us translate this blossom of H back to the original graph G. The path s to w corresponds to an alternating path starting at a free vertex f (of G) and ending at w, where the last edge is in the stem is in the matching, the cycle $w \ldots u \ldots v \ldots w$ is an alternating odd length cycle in G where the two edges adjacent to w are unmatched.

We can not apply a blossom to a matching in the hope of getting better matching. In fact, this is illegal and yield something which is not a matching. On the positive side, we discovered an odd alternating cycle in the graph G. Summarizing the above algorithm, we have:

Lemma 30.1.6. *Given a graph G with n vertices and m edges, and a matching M, one can find in $O(n+m)$ time, either a blossom in G or an augmenting path in G.*



To see what to do next, we have to realize how a matching in G interact with an odd length cycle which is computed by our algorithm (i.e., blossom). In particular, assume that the free vertex in the cycle is unmatched. To get a maximum number of edges of the matching in the cycle, we must at most $(n-1)/2$ edges in the cycle, but then we can rotate the matching edges in the cycle, such that any vertex on the cycle can be free. See figure on the right.



Let G/C denote the graph resulting from collapsing such an odd cycle C into single vertex. The new vertex is marked by $\{C\}$.

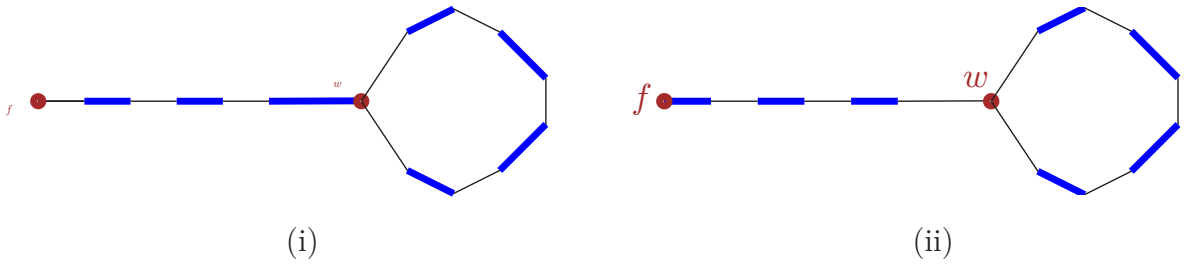
Lemma 30.1.7. *Given a graph G, a matching M, and a flower B, one can find a matching M' with the same cardinality, such that the blossom of B contains a free (i.e., unmatched) vertex in M'.*

Proof: If the stem of B is empty and B is just formed by a blossom, and then we are done. Otherwise, B was as stem π which is an even length alternating path starting from from a free vertex v. Observe that the matching $M' = M \oplus \pi$ is of the same cardinality, and the cycle in B now becomes an alternating odd cycle, with a free vertex.

Intuitively, what we did is to apply the stem to the matching M. See Figure 30.3. ■

Theorem 30.1.8. *Let M be a matching, and let C be a blossom for M with an unmatched vertex v. Then, M is a maximum matching in G if and only if $M/C = M \setminus C$ is a maximum matching in G/C.*

Proof: Let G/C be the collapsed graph, with $\{C\}$ denoting the vertex that correspond to the cycle C.



(i) (ii)

Figure 30.3: (i) the flower, and (ii) the invert stem.

Note, that the collapsed vertex $\{C\}$ in G/C is free. Thus, an augmenting path π in G/C either avoids the collapsed vertex $\{C\}$ altogether, or it starts or ends there. In any case, we can rotate the matching around C such that π would be an augmenting path in G. Thus, if M/C is not a maximum matching in G/C then there exists an augmenting path in G/C, which in turn is an augmenting path in G, and as such M is not a maximum matching in G.

Similarly, if π is an augmenting path in G and it avoids C then it is also an augmenting path in G/C, and then M/C is not a maximum matching in G/C.

Otherwise, since π starts and ends in two different free vertices and C has only one free vertex, it follows that π has an endpoint outside C. Let v be this endpoint of π and let u be the first vertex of π that belongs to C. Let σ be the path $\pi[v, u]$.

Let f be the free vertex of C. Note that f is unmatched. Now, if $u = f$ we are done, since then π is an augmenting path also in G/C. Note that if u is matched in C, as such, it must be that the last edge e in π is unmatched. Thus, rotate the matching M around C such that u becomes free. Clearly, then σ is now an augmenting path in G (for the rotated matching) and also an augmenting path in G/C. ■

Corollary 30.1.9. *Let M be a matching, and let C be an alternating odd length cycle with the unmatched vertex being free. Then, there is an augmenting path in G if and only if there is an augmenting path in G/C.*

30.1.2 The algorithm

Start from the empty matching M in the graph G.

Now, repeatedly, try to enlarge the matching. First, check if you can find an edge with both endpoints being free, and if so add it to the matching. Otherwise, compute the graph H (this is the graph where all the free vertices are collapsed into a single vertex), and compute an alternating BFS tree in H. From the alternating BFS, we can extract the shortest alternating cycle based in the root (by finding the highest bridge). If this alternating cycle corresponds to an alternating path in G then we are done, as we can just apply this alternating path to the matching M getting a bigger matching.

If this is a flower, with a stem ρ and a blossom C then apply the stem to M (i.e., compute the matching $M \oplus \rho$). Now, C is an odd cycle with the free vertex being unmatched. Compute

recursively an augmenting path π in G/C. By the above discussing, we can easily transform this into an augmenting path in G. Apply this augmenting path to M.

Thus, we succeeded in computing a matching with one edge more in it. Repeat till the process get stuck. Clearly, what we have is a maximum size matching.

30.1.2.1 Running time analysis

Every shrink cost us $O(m+n)$ time. We need to perform $O(n)$ recursive shrink operations till we find an augmenting path, if such a path exists. Thus, finding an augmenting path takes $O(n(m+n))$ time. Finally, we have to repeat this $O(n)$ times. Thus, overall, the running time of our algorithm is $O(n^2(m+n)) = O(n^4)$.

Theorem 30.1.10. *Given a graph G with n vertices and m edges, computing a maximum size matching in G can be done in $O(n^2m)$ time.*

30.2 Maximum Weight Matching in A Non-Bipartite Graph

This the hardest case and it is non-trivial to handle. There are known polynomial time algorithms, but I feel that they are too involved, and somewhat cryptic, and as such should not be presented in class. For the interested student, a nice description of such an algorithm is presented in

Combinatorial Optimization - Polyhedral and efficiency by Alexander Schrijver Vol. A, 453–459.

The description above also follows loosely the same book.

Part XIII

Union Find

Chapter 31

Union Find

31.1 Union-Find

31.1.1 Requirements from the data-structure

We want to maintain a collection of sets, under the following operations.

(i) **makeSet**(x) - creates a set that contains the single element x.

(ii) **find**(x) - returns the set that contains x.

(iii) **union**(A,B) - returns the set which is the union of A and B. Namely $A \cup B$. Namely, this operation merges the two sets A and B and return the merged set.

> **Scene:** It's a fine sunny day in the forest, and a rabbit is sitting outside his burrow, tippy-tapping on his typewriter.
> Along comes a fox, out for a walk.
> *Fox*: "What are you working on?"
> *Rabbit*: "My thesis."
> *Fox*: "Hmmm. What's it about?"
> *Rabbit*: "Oh, I'm writing about how rabbits eat foxes."
> *Fox*: (incredulous pause) "That's ridiculous! Any fool knows that rabbits don't eat foxes."
> *Rabbit*: "Sure they do, and I can prove it. Come with me."
> They both disappear into the rabbit's burrow. After a few minutes, the rabbit returns, alone, to his typewriter and resumes typing.
> **Scene inside the rabbit's burrow**: In one corner, there is a pile of fox bones. In another corner, a pile of wolf bones. On the other side of the room, a huge lion is belching and picking his teeth.
> (The End)
> **Moral**: It doesn't matter what you choose for a thesis subject.
> It doesn't matter what you use for data.
> What does matter is who you have for a thesis advisor.
>
> – – Anonymous

31.1.2 Amortized analysis

We use a data-structure as a black-box inside an algorithm (for example Union-Find in Kruskal algorithm for computing minimum spanning tee). So far, when we design a data-structure we cared about worst case time for operation. Note however, that this is not necessarily the right measure. Indeed, we care about the *overall* running time spend on

doing operations in the data-structure, and less about its running time for a single operation.

Formally, the ***amortized running-time*** of an operation is the average time it takes to perform an operation on the data-structure. Formally, the amortized time of an operation is $\frac{\text{overall running time}}{\text{number of operations}}$.

31.1.3 The data-structure

To implement this operations, we are going to use Reversed Trees. In a reversed tree, every element is stored in its own node. A node has one pointer to its parent. A set is uniquely identified with the element stored in the root of such a reversed tree. See Figure 31.1 for an example of how such a reversed tree looks like.

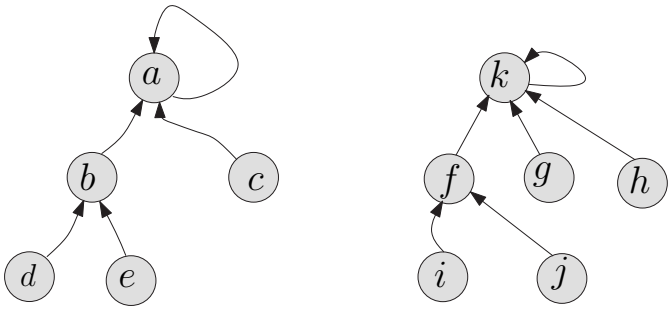
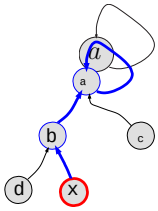


Figure 31.1: The Union-Find representation of the sets $A = \{a, b, c, d, e\}$ and $B = \{f, g, h, i, j, k\}$. The set A is uniquely identified by a pointer to the root of A, which is the node containing a.

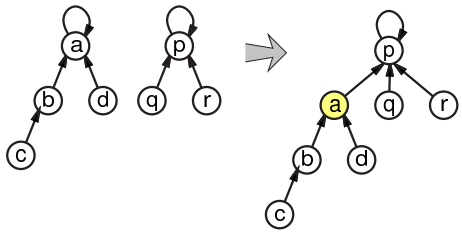
We implement the operations of the Union-Find data structure as follows:

(A) **makeSet**: Create a singleton pointing to itself:

(B) **find**(x): We start from the node that contains x, and we start traversing up the tree, following the parent pointer of the current node, till we get to the root, which is just a node with its parent pointer pointing to itself.
Thus, doing a **find**(x) operation in the reversed tree shown on the right, involve going up the tree from $x \to b \to a$, and returning a as the set.

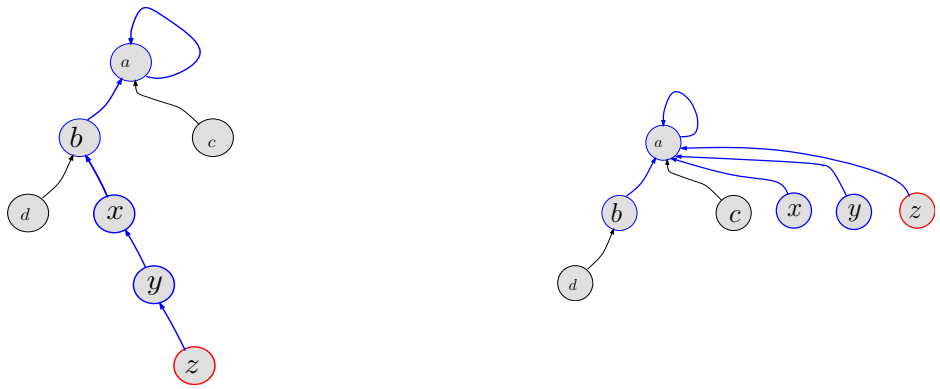


(C) **union**(a, p): We merge two sets, by hanging the root of one tree, on the root of the other. Note, that this is a destructive operation, and the two original sets no longer exist. Example of how the new tree representing the new set is depicted on the right.



Note, that in the worst case, depth of tree can be linear in n (the number of elements stored in the tree), so the **find** operation might require $\Omega(n)$ time. To see that this worst case is realizable perform the following sequence of operations: create n sets of size 1, and repeatedly merge the current set with a singleton. If we always merge (i.e., do **union**) the current set with a singleton by hanging the current set on the singleton, the end result would be a reversed tree which looks like a linked list of length n. Doing a **find** on the deepest element will take linear time.

So, the question is how to further improve the performance of this data-structure. We are going to do this, by using two "hacks":



(a) (b)

Figure 31.2: (a) The tree before performing **find**(z), and (b) The reversed tree after performing **find**(z) that uses path compression.

```
makeSet(x)
  p̄(x) ← x
  rank(x) ← 0
```

```
find(x)
  if x ≠ p̄(x) then
    p̄(x) ← find(p̄(x))
  return p̄(x)
```

```
union(x, y )
  A ← find(x)
  B ← find(y)
  if rank(A) > rank(B) then
    p̄(B) ← A
  else
    p̄(A) ← B
    if rank(A) = rank(B) then
      rank(B) ← rank(B) + 1
```

Figure 31.3: The pseudo-code for the Union-Find data-structure that uses both path-compression and union by rank. For element x, we denote the parent pointer of x by $\overline{\mathrm{p}}(x)$.

(i) ***Union by rank***: Maintain for every tree, in the root, a bound on its depth (called ***rank***). Always hang the smaller tree on the larger tree.

(ii) ***Path compression***: Since, anyway, we travel the path to the root during a **find** operation, we might as well hang all the nodes on the path directly on the root.

An example of the effects of path compression are depicted in Figure 31.2. For the pseudo-code of the **makeSet**, **union** and **find** using path compression and union by rank, see Figure 31.3.

We maintain a rank which is associated with each element in the data-structure. When a singleton is being created, its associated rank is set to zero. Whenever two sets are being merged, we update the rank of the new root of the merged trees. If the two trees have different root ranks, then the rank of the root does not change. If they are equal then we set the rank of the new root to be larger by one.

31.2 Analyzing the Union-Find Data-Structure

Definition 31.2.1. A node in the union-find data-structure is a ***leader*** if it is the root of a (reversed) tree.

Lemma 31.2.2. *Once a node stop being a leader (i.e., the node in top of a tree), it can never become a leader again.*

Proof: Note, that an element x can stop being a leader only because of a **union** operation that hanged x on an element y. From this point on, the only operation that might change x parent pointer, is a **find** operation that traverses through x. Since path-compression can only change the parent pointer of x to point to some other element y, it follows that x parent pointer will never become equal to x again. Namely, once x stop being a leader, it can never be a leader again. ■

Lemma 31.2.3. *Once a node stop being a leader then its rank is fixed.*

Proof: The rank of an element changes only by the **union** operation. However, the **union** operation changes the rank, only for elements that are leader after the operation is done. As such, if an element is no longer a leader, than its rank is fixed. ■

Lemma 31.2.4. *Ranks are monotonically increasing in the reversed trees, as we travel from a node to the root of the tree.*

Proof: It is enough to prove, that for every edge $u \rightarrow v$ in the data-structure, we have $\mathrm{rank}(u) < \mathrm{rank}(v)$. The proof is by induction. Indeed, in the beginning of time, all sets are singletons, with rank zero, and the claim trivially holds.

Next, assume that the claim holds at time t, just before we perform an operation. Clearly, if this operation is **union** (A, B), and assume that we hanged $\mathrm{root}(A)$ on $\mathrm{root}(B)$. In this case, it must be that $\mathrm{rank}(\mathrm{root}(B))$ is now larger than $\mathrm{rank}(\mathrm{root}(A))$, as can be easily verified. As such, if the claim held before the **union** operation, then it is also true after it was performed.

If the operation is **find**, and we traverse the path π, then all the nodes of π are made to point to the last node v of π. However, by induction, $\mathrm{rank}(v)$ is larger than the rank of all the other nodes of π. In particular, all the nodes that get compressed, the rank of their new parent, is larger than their own rank. ■

Lemma 31.2.5. *When a node gets rank k than there are at least $\geq 2^k$ elements in its subtree.*

Proof: The proof is by induction. For $k = 0$ it is obvious since a singleton has a rank zero, and a single element in the set. Next observe that a node gets rank k only if the merged two roots has rank $k - 1$. By induction, they have 2^{k-1} nodes (each one of them), and thus the merged tree has $\geq 2^{k-1} + 2^{k-1} = 2^k$ nodes. ■

Lemma 31.2.6. *The number of nodes that get assigned rank k throughout the execution of the Union-Find data-structure is at most $n/2^k$.*

Proof: Again, by induction. For $k = 0$ it is obvious. We charge a node v of rank k to the two elements u and v of rank $k-1$ that were leaders that were used to create the new larger set. After the merge v is of rank k and u is of rank $k-1$ and it is no longer a leader (it can not participate in a union as a leader any more). Thus, we can charge this event to the two (no longer active) nodes of degree $k-1$. Namely, u and v.

By induction, we have that the algorithm created at most $n/2^{k-1}$ nodes of rank $k-1$, and thus the number of nodes of rank k created by the algorithm is at most $\leq \left(n/2^{k-1}\right)/2 = n/2^k$. ■

Lemma 31.2.7. *The time to perform a single* **find** *operation when we perform union by rank and path compression is $O(\log n)$ time.*

Proof: The rank of the leader v of a reversed tree T, bounds the depth of a tree T in the Union-Find data-structure. By the above lemma, if we have n elements, the maximum rank is $\lg n$ and thus the depth of a tree is at most $O(\log n)$. ■

Surprisingly, we can do much better.

Theorem 31.2.8. *If we perform a sequence of m operations over n elements, the overall running time of the Union-Find data-structure is $O((n+m)\log^* n)$.*

We remind the reader that $\log^*(n)$ is the number one has to take lg of a number to get a number smaller than two (there are other definitions, but they are all equivalent, up to adding a small constant). Thus, $\log^* 2 = 1$ and $\log^* 2^2 = 2$. Similarly, $\log^* 2^{2^2} = 1 + \log^*(2^2) = 2 + \log^* 2 = 3$. Similarly, $\log^* 2^{2^{2^2}} = \log^*(65536) = 4$. Things get really exciting, when one considers

$$\log^* 2^{2^{2^{2^2}}} = log^* 2^{65536} = 5.$$

However, $\log^*$ is a monotone increasing function. And $\beta = 2^{2^{2^{2^2}}} = 2^{65536}$ is a huge number (considerably larger than the number of atoms in the universe). Thus, for all practical purposes, $\log^*$ returns a value which is smaller than 5. Intuitively, Theorem 31.2.8 states (in the amortized sense), that the Union-Find data-structure takes constant time per operation (unless n is larger than β which is unlikely).

It would be useful to look on the inverse function to $\log^*$.

Definition 31.2.9. Let $\text{Tower}(b) = 2^{\text{Tower}(b-1)}$ and $\text{Tower}(0) = 1$.

So, $\text{Tower}(i)$ is just a tower of $2^{2^{2^{\cdot^{\cdot^{\cdot^2}}}}}$ of height i. Observe that $\log^*(\text{Tower}(i)) = i$.

Definition 31.2.10. For $i \geq 0$, let $\text{Block}(i) = [\text{Tower}(i-1)+1, \text{Tower}(i)]$; that is

$$\text{Block}(i) = \left[z, 2^{z-1}\right] \qquad \text{for} \qquad z = \text{Tower}(i-1)+1.$$

For technical reasons, we define $\text{Block}(0) = [0,1]$. As such,

$$\begin{aligned}
\text{Block}(0) &= \left[0,1\right] \\
\text{Block}(1) &= \left[2,2\right] \\
\text{Block}(2) &= \left[3,4\right] \\
\text{Block}(3) &= \left[5,16\right] \\
\text{Block}(4) &= \left[17,65536\right] \\
\text{Block}(5) &= \left[65537, 2^{65536}\right] \\
&\vdots
\end{aligned}$$

The running time of **find**(x) is proportional to the length of the path from x to the root of the tree that contains x. Indeed, we start from x and we visit the sequence:

$$x_1 = x, x_2 = \overline{\text{p}}(x) = \overline{\text{p}}(x_1), ..., x_i = \overline{\text{p}}(x_{i-1}), \ldots, x_m = \text{root of tree.}$$

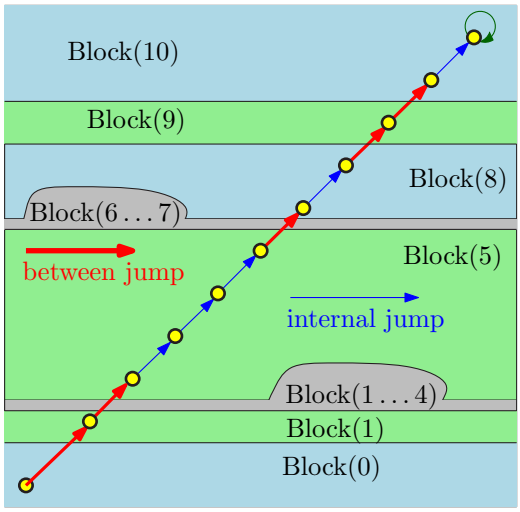
Clearly, we have for this sequence: $\text{rank}(x_1) < \text{rank}(x_2) < \text{rank}(x_3) < \ldots < \text{rank}(x_m)$, and the time it takes to perform **find**(x) is proportional to m, the length of the path from x to the root of the tree containing x.

Definition 31.2.11. A node x is in the ith block if $\text{rank}(x) \in \text{Block}(i)$.

We are now looking for ways to pay for the **find** operation, since the other two operations take constant time.

Observe, that the maximum rank of a node v is $O(\log n)$, and the number of blocks is $O(\log^* n)$, since $O(\log n)$ is in the block $\text{Block}(c \log^* n)$, for c a constant sufficiently large.

In particular, consider a **find** (x) operation, and let π be the path visited. Next, consider the ranks of the elements of π, and imagine partitioning π into which blocks each element rank belongs to. An example of such a path is depicted on the right. The price of the **find** operation is the length of π.



Formally, for a node x, $\nu = \text{index}_{\text{B}}(x)$ is the index of the block that contains $\text{rank}(x)$. Namely, $\text{rank}(x) \in \text{Block}\big(\text{index}_{\text{B}}(x)\big)$. As such, $\text{index}_{\text{B}}(x)$ is the ***block of*** x.

Now, during a **find** operation, since the ranks of the nodes we visit are monotone increasing, once we pass through from a node v in the ith block into a node in the $(i+1)$th block, we can never go back to the ith block (i.e., visit elements with rank in the ith block). As such, we can charge the visit to nodes in π that are next to a element in a different block, to the number of blocks (which is $O(\log^* n)$).

Definition 31.2.12. Consider a path π traversed by a **find** operation. Along the path π, an element x, such that $\overline{\text{p}}(x)$ is in a different block, is a ***jump between blocks***.

On the other hand, a jump during a **find** operation inside a block is called an ***internal jump***; that is, x and $\overline{\text{p}}(x)$ are in the same block.

Lemma 31.2.13. *During a single* **find**(x) *operation, the number of jumps between blocks along the search path is* $O(\log^* n)$.

Proof: Consider the search path $\pi = x_1, \ldots, x_m$, and consider the list of numbers $0 \leq \text{index}_{\text{B}}(x_1) \leq \text{index}_{\text{B}}(x_2) \leq \ldots \leq \text{index}_{\text{B}}(x_m)$. We have that $\text{index}_{\text{B}}(x_m) = O(\log^* n)$. As such, the number of elements x in π such that $\text{index}_{\text{B}}(x) \neq \text{index}_{\text{B}}(\overline{\text{p}}(x))$ is at most $O(\log^* n)$. ■

Consider the case that x and $\overline{\text{p}}(x)$ are both the same block (i.e., $\text{index}_{\text{B}}(x) = \text{index}_{\text{B}}(\overline{\text{p}}(x))$ and we perform a find operation that passes through x. Let $r_{\text{bef}} = \text{rank}(\overline{\text{p}}(x))$ before the **find** operation, and let r_{aft} be $\text{rank}(\overline{\text{p}}(x))$ after the **find** operation. Observe, that because of path compression, we have $r_{\text{aft}} > r_{\text{bef}}$. Namely, when we jump inside a block, we do some work: we make the parent pointer of x jump forward and the new parent has higher rank. We will charge such internal block jumps to this "progress".

Lemma 31.2.14. *At most* $|\text{Block}(i)| \leq \text{Tower}(i)$ **find** *operations can pass through an element* x*, which is in the* i*th block (i.e.,* $\text{index}_{\text{B}}(x) = i$*) before* $\overline{\text{p}}(x)$ *is* no longer *in the* i*th block. That is* $\text{index}_{\text{B}}(\overline{\text{p}}(x)) > i$.

Proof: Indeed, by the above discussion, the parent of x increases its rank every-time an internal jump goes through x. Since there at most $|\mathrm{Block}(i)|$ different values in the ith block, the claim follows. The inequality $|\mathrm{Block}(i)| \leq \mathrm{Tower}(i)$ holds by definition, see Definition 31.2.10. ∎

Lemma 31.2.15. *There are at most $n/\mathrm{Tower}(i)$ nodes that have ranks in the ith block throughout the algorithm execution.*

Proof: By Lemma 31.2.6, we have that the number of elements with rank in the ith block is at most

$$\sum_{k \in \mathrm{Block}(i)} \frac{n}{2^k} = \sum_{k=\mathrm{Tower}(i-1)+1}^{\mathrm{Tower}(i)} \frac{n}{2^k} = n \cdot \sum_{k=\mathrm{Tower}(i-1)+1}^{\mathrm{Tower}(i)} \frac{1}{2^k} \leq \frac{n}{2^{\mathrm{Tower}(i-1)}} = \frac{n}{\mathrm{Tower}(i)}.$$

∎

Lemma 31.2.16. *The number of internal jumps performed, inside the ith block, during the lifetime of the union-find data-structure is $O(n)$.*

Proof: An element x in the ith block, can have at most $|\mathrm{Block}(i)|$ internal jumps, before all jumps through x are jumps between blocks, by Lemma 31.2.14. There are at most $n/\mathrm{Tower}(i)$ elements with ranks in the ith block, throughout the algorithm execution, by Lemma 31.2.15. Thus, the total number of internal jumps is

$$|\mathrm{Block}(i)| \cdot \frac{n}{\mathrm{Tower}(i)} \leq \mathrm{Tower}(i) \cdot \frac{n}{\mathrm{Tower}(i)} = n.$$

∎

We are now ready for the last step.

Lemma 31.2.17. *The number of internal jumps performed by the Union-Find data-structure overall is $O(n \log^* n)$.*

Proof: Every internal jump can be associated with the block it is being performed in. Every block contributes $O(n)$ internal jumps throughout the execution of the union-find data-structures, by Lemma 31.2.16. There are $O(\log^* n)$ blocks. As such there are at most $O(n \log^* n)$ internal jumps. ∎

Lemma 31.2.18. *The overall time spent on m* **find** *operations, throughout the lifetime of a union-find data-structure defined over n elements, is $O((m+n)\log^* n)$.*

Theorem 31.2.8 now follows readily from the above discussion.

Bibliography

[ACG+99] G. Ausiello, P. Crescenzi, G. Gambosi, V. Kann, A. Marchetti-Spaccamela, and M. Protasi. *Complexity and approximation.* Springer-Verlag, Berlin, 1999.

[AKS83] M. Ajtai, J. Komlós, and E. Szemerédi. An $O(n \log n)$ sorting network. In *Proc. 15th Annu. ACM Sympos. Theory Comput.*, pages 1–9, 1983.

[AN04] N. Alon and A. Naor. Approximating the cut-norm via grothendieck's inequality. In *Proc. 36th Annu. ACM Sympos. Theory Comput.*, pages 72–80, 2004.

[ASS96] H. Abelson, G. J. Sussman, and J. Sussman. *Structure and interpretation of computer programs.* MIT Press, 1996.

[BV04] S. Boyd and L. Vandenberghe. *Convex Optimization.* Cambridge, 2004.

[Chr76] N. Christofides. Worst-case analysis of a new heuristic for the travelling salesman problem. Technical Report Report 388, Graduate School of Industrial Administration, Carnegie Mellon University, 1976.

[CKX10] J. Chen, I. A. Kanj, and G. Xia. Improved upper bounds for vertex cover. *Theor. Comput. Sci.*, 411(40-42):3736–3756, 2010.

[CS00] S. Cho and S. Sahni. A new weight balanced binary search tree. *Int. J. Found. Comput. Sci.*, 11(3):485–513, 2000.

[FG88] T. Feder and D. H. Greene. Optimal algorithms for approximate clustering. In *Proc. 20th Annu. ACM Sympos. Theory Comput.*, pages 434–444, 1988.

[FG95] U. Feige and M. Goemans. Approximating the value of two power proof systems, with applications to max 2sat and max dicut. In *ISTCS '95: Proceedings of the 3rd Israel Symposium on the Theory of Computing Systems (ISTCS'95)*, page 182, Washington, DC, USA, 1995. IEEE Computer Society.

[GJ90] M. R. Garey and D. S. Johnson. *Computers and Intractability; A Guide to the Theory of NP-Completeness.* W. H. Freeman & Co., 1990.

[GLS93] M. Grötschel, L. Lovász, and A. Schrijver. *Geometric Algorithms and Combinatorial Optimization*, volume 2 of *Algorithms and Combinatorics.* Springer-Verlag, Berlin Heidelberg, 2nd edition, 1993.

[GT89] A. V. Goldberg and R. E. Tarjan. Finding minimum-cost circulations by canceling negative cycles. *J. Assoc. Comput. Mach.*, 36(4):873–886, 1989.

[GW95] M. X. Goemans and D. P. Williamson. Improved approximation algorithms for maximum cut and satisfiability problems using semidefinite programming. *J. Assoc. Comput. Mach.*, 42(6):1115–1145, November 1995.

[Har04] S. Har-Peled. Clustering motion. *Discrete Comput. Geom.*, 31(4):545–565, 2004.

[Hås01a] J. Håstad. Some optimal inapproximability results. *J. Assoc. Comput. Mach.*, 48(4):798–859, July 2001.

[Hås01b] J. Håstad. Some optimal inapproximability results. *J. Assoc. Comput. Mach.*, 48(4):798–859, 2001.

[Kar78] R. M. Karp. A characterization of the minimum cycle mean in a digraph. *Discrete Math.*, 23:309–311, 1978.

[KKMO04] S. Khot, G. Kindler, E. Mossel, and R. O'Donnell. Optimal inapproximability results for max cut and other 2-variable csps. In *Proc. 45th Annu. IEEE Sympos. Found. Comput. Sci.*, pages 146–154, 2004. To appear in SICOMP.

[MOO05] E. Mossel, R. O'Donnell, and K. Oleszkiewicz. Noise stability of functions with low influences invariance and optimality. In *Proc. 46th Annu. IEEE Sympos. Found. Comput. Sci.*, pages 21–30, 2005.

[MR95] R. Motwani and P. Raghavan. *Randomized Algorithms*. Cambridge University Press, 1995.

[MU05] M. Mitzenmacher and U. Upfal. *Probability and Computing – randomized algorithms and probabilistic analysis*. Cambridge, 2005.

[SA96] R. Seidel and C. R. Aragon. Randomized search trees. *Algorithmica*, 16:464–497, 1996.

[Sch04] A. Schrijver. *Combinatorial Optimization : Polyhedra and Efficiency (Algorithms and Combinatorics)*. Springer, July 2004.

[Tar85] É. Tardos. A strongly polynomial minimum cost circulation algorithm. *Combinatorica*, 5(3):247–255, 1985.

[WG75] H. W. Watson and F. Galton. On the probability of the extinction of families. *J. Anthrop. Inst. Great Britain*, 4:138–144, 1875.

Index