LABORATORY FOR
COMPUTER SCIENCE

MASSACHUSETTS
INSTITUTE OF
TECHNOLOGY

MIT/LCS/RSS-27

Advanced Algorithms

Michel X. Goemans

December, 1994

545 TECHNOLOGY SQUARE; CAMBRIDGE, MASSACHUSETTS 02139 (617) 253-5851

Preface

This document consists of the lecture notes for *6.854J/18.415J Advanced Algorithms* as it was taught by Michel Goemans in the Fall 1994. The notes are for the most part based on notes scribed by students during the last four years, with help from the teaching assistants, Marios Papaefthymiou (Fall 1990), Michael Klugerman (Fall 1991 and 1992), Esther Jesurum (Fall 1993) and Matthew Andrews (Fall 1994). The notes are not meant to be in polished form, and they probably contain several errors and omissions. Rather, they are intended to be an aid in teaching and/or studying a few advanced topics in the design and analysis of algorithms. Part of these notes have not been covered in class, but have been included for completeness. Also, some topics which have been covered in the past but not this year (such as basis reduction for lattices) have been omitted. The topics covered this year are online algorithms, randomized algorithms, linear programming, network flow, and approximation algorithms. A few new results have been covered for the first time this year, such as the result of Koutsoupias and Papadimitriou on the competitiveness of the work function algorithm for the k-server problem, and the approximation algorithm of Linial, London and Rabinovich and Aumann and Rabani for the multicommodity cut problem using embeddings of metrics.

Help from the following (certainly incomplete) list of scribes is acknowledged: M. Andrews, R. Blumofe. C. Celebiler, A. Chou, L. Feeney, F. Garcia, L. Girod, D. Gupta, J. Huang, J. Kleinberg, M. Klugerman, M. Kiwi, J. Kučan, J. Leo, X. Luo, Y. Ma, P. McCorquodale, R. Ostrovsky, S. Raghavan, K. Randall, A. Russell, I. Saias, R. Sidney, E. Soylemez, R. Sundaram, S. Toledo, L. Tucker-Kellogg D. Wald, D. Williamson, D. Wilson, E. Wolf and Y. Yin.

Contents

Lectures

Linear Programming

Network Flow

Approximation Algorithms

18.415/6.854 Advanced Algorithms September 1994

On-Line Algorithms

Lecturer: Michel X. Goemans

1 Introduction

The first topic we will cover in this course is *on-line algorithms*. The concept of on-line algorithms has been around for quite some time (it was for example present in the seventies in the bin packing literature, see [13, 20]), but it has been a fairly hot research area since the publication of the landmark paper by Sleator and Tarjan [19] in 1985.

Typically when we solve problems and design algorithms we assume that we know all the data *a priori*. However in many practical situations this may not be true and rather than have the input in advance it may be presented to us as we proceed. We give below some examples to motivate the topic of on-line algorithms.

1.1 Investment Problem

Consider an investor with a given sum of money which he wishes to invest so as to maximize his gain at the end of a specified period of time. He has a variety of options: he may keep funds in a money market account, he may buy certificates of deposit or he may invest in the stock market. He can keep a mixed portfolio and he can reallocate his assets during the course of the investment period. Now in the offline case he has full information about the behaviour of the various financial and capital markets and so can compute an optimal strategy to maximize his profit. An on-line algorithm, however, is a strategy which at each point in time decides what portfolio to maintain based only on past information and with no knowledge whatsoever about the future. The profit generated is a determinant of the quality of the on-line strategy. We will see later how exactly to quantify this quality by introducing the notion of competitive analysis.

1.2 Ski Rental Problem

Consider the following scenario: you are a skier and, each day, need to either rent skis for \$1 or buy a pair of skis for \$$T$ which will last for the rest of the season. Unfortunately, you do not know when the ski season will end. If you knew a priori the length of the season (this is the offline case) say L then it is obvious that you should rent if $L < T$ and buy if $L \geq T$. On the other hand, an on-line strategy for this problem would first fix an integer k, and then rent for k days and buy on day $k+1$ (if it has the opportunity).

At this point we must pause and decide how to evaluate the performance of on-line algorithms? One way to analyze such algorithms is to assume that the input is given according to a certain probability distribution, and compute the expected behavior of the algorithm based on this distribution. However we wish to avoid making assumptions about input distributions. The analytic machinery we present will not demand that the inputs come from some known distribution, but instead will compare the performance of the on-line algorithm with that of the best offline algorithm. This notion of comparison is called *competitive analysis* and was introduced by Sleator and Tarjan [19].

Definition 1 *An on-line algorithm* A *is α-competitive if for all input sequences σ, $C_{\mathrm{A}}(\sigma) \leq \alpha C_{\mathrm{MIN}}(\sigma)$, where $C_{\mathrm{A}}(\sigma)$ is the cost of the on-line strategy* A *for σ and $C_{\mathrm{MIN}}(\sigma)$ is the cost of the optimal offline algorithm for σ.*

Let us now reconsider the ski rental problem in the light of the above definition. One strategy is to buy on the first day. This strategy is T-competitive and the worst input sequence is obtained when $L = 1$. Another on-line strategy is to rent for the first $T-1$ days and then buy. If $L < T$ the cost of the on-line and optimal offline strategy is the same. If $L \geq T$ the cost of the on-line strategy is $2T-1$ and the cost of the optimal offline strategy is T. Hence this algorithm is $(2-1/T)$-competitive and it is fairly easy to show that it is the optimal on-line algorithm.

In many cases, the definition of competitiveness is slightly relaxed by imposing that $C_{\mathrm{A}}(\sigma) \leq \alpha C_{\mathrm{MIN}}(\sigma) + c$ for some constant c independent of σ.

2 Paging Problem

This is a problem which arises in system software. We have a two level memory divided into pages of fixed size. There are k pages in fast memory while the remaining pages are in slow memory. The input sequence σ in this problem are requests to pages $< \sigma_1, \sigma_2, \sigma_3 \cdots >$. If the request for a page σ_i is in fast memory then we need not do anything, otherwise we have a page fault and are faced with the problem of deciding which page in the fast memory to swap with σ_i. Our cost in this case is the number of page faults. An on-line algorithm for this problem decides in the case of a page fault which page to remove from fast memory without any information of the subsequent page requests. For example, here are some possible on-line strategies:

LIFO (Last-In-First-Out): remove the page most recently placed in fast memory.

FIFO (First-In-First-Out): remove the page that has been in memory longest.

LRU (Least-Recently-Used): replace the page whose most recent access is earliest.

LFU (Least-Frequently-Used): replace the page that has been accessed the least.

FWF (Flush-When-Full): remove any unmarked page and mark the requested page. Whenever all pages in fast memory are marked, unmark them all.

Let us consider the LIFO strategy for the paging problem. Consider an input sequence of two pages p and q repeatedly requested one after the other. It is easy to see that LIFO is not α-competitive for any α. We say that LIFO is *not competitive.* A similar comment can be made about LFU.

In the theorem to follow we will show that there exist k-competitive strategies (where k is the size of the fast memory). Later, we shall show that this is the best possible and we shall also characterize MIN - the optimal offline algorithm.

Theorem 1 LRU *is k-competitive.*

Proof:

Note that by definition of our problem, both LRU and MIN have the same initial configuration in fast memory (i.e. the same k pages in fast memory.) We must show that for any k initial pages in fast memory, and for all σ, $C_{\text{LRU}}(\sigma) \leq k \cdot C_{\text{MIN}}(\sigma)$.

Let us examine actions of LRU for any particular fixed σ. Let us divide the input sequence into phases:

$$\sigma = \sigma_1, \ldots, \underbrace{\sigma_i}_{\text{first page fault}}, \underbrace{[\sigma_{(i+1)}, \ldots, \sigma_j]}_{\text{phase 1}}, \underbrace{[\sigma_{(j+1)}, \ldots, \sigma_l]}_{\text{phase 2}}, \ldots$$

where each phase contains exactly k page faults, ending with a page fault. For example, the first phase ends with σ_j where

$$j = min\{t \;\; : \;\; \text{LRU has } k \text{ page faults in } \sigma_{(i+1)}, \ldots, \sigma_t\}$$

Let us consider a cost of a single phase for both LRU and MIN. By definition, LRU makes k faults. We argue that MIN must make at least one page fault on each phase. Let us consider two different cases.

Case 1: during the same phase, LRU faults twice on some page p. Thus, the phase looks likes this:

$$\ldots, [\sigma_{(i+1)}, \ldots, \overbrace{\underbrace{\sigma_{p1} = p}_{\text{first page fault } p}, \ldots, \underbrace{\sigma_{p2} = p}_{\text{the second page fault of } p},}^{\text{all } k \text{ pages in fast memory have been requested before second request to } p} \ldots, \sigma_j], \ldots$$

Notice that after the first page fault to page p, p is brought into fast memory and is later replaced (by some other page) only if p becomes the least recently used page in fast memory (by definition of LRU.) Hence, if there is another page fault to p, it means that all other k pages that are resident in fast memory just before σ_{p_2} came after the first page fault to p but before the second request to p. Thus, in this case, $k+1$ different pages have been requested in a phase. This, however, forces MIN to make at least one page fault in every phase , i.e. $C_{\text{MIN}} \geq 1$ for every phase.

Case 2: LRU faults on k different pages. Here, we consider two sub-cases, depending on the last page-fault (say on p) which occurred before the beginning of current phase:

$$\underbrace{\sigma_i = p}_{\text{last page fault before current phase}}, \underbrace{[\sigma_{(i+1)}, \cdots, \sigma_j]}_{\text{current phase}},$$

Case 2a: In a current phase, there is a page-fault to page p.

$$\underbrace{\sigma_i = p}_{\text{last page fault before current phase}}, \underbrace{[\sigma_{(i+1)}, \cdots, p, \cdots, \sigma_j]}_{\text{current phase}},$$

This case is very similar to case 1. In particular, before the second page fault to p occurs, there must have been k requests to pages other then p, to force p to be swapped out. Thus, during the current phase there are a total of $k+1$ different page requests. Thus $C_{\text{MIN}}(phase) \geq 1$.

Case 2b: In a current phase, there is no page-fault to page p.

$$\sigma_i = p, \quad \underbrace{[\sigma_{(i+1)}, \cdots, \sigma_j]}_{\text{current phase with no page fault on } p}$$

Let us consider MIN's behavior. Before the phase starts it must have p in its fast memory, and other $k-1$ locations which we don't care about. Notice, however, that during current phase, k different requests arrive, none of them are p. Thus, to accommodate all of them, MIN must get rid of p, hence $C_{\text{MIN}}(phase) \geq 1$ as well.

Thus, for every phase, MIN must pay at least one page fault, wile LRU pays k faults. However, what about the very first page fault, before the beginning of the first phase? Observe that both LRU and MIN start in the same configuration. Hence, the first time there is a request to page which in not in memory, it is in fact not in memory for both LRU and MIN. Hence they both must do a page fault. □

Remark 1 *Notice that essentially the same argument goes for* FIFO *in order to show that* FIFO *is also k-competitive.*

The above theorem was proved by Sleator and Tarjan [19]. Notice that in the above proof we have not used explicitly what is the optimal offline strategy MIN. We have only related the cost of the on-line algorithm under consideration to a *lower bound* on the cost of any on-line algorithm. This observation might seem elementary but comparing to a lower bound is a very typical technique in the analysis of algorithms (not just on-line algorithms but also, as we shall see later in the class, approximation algorithms).

The moral of the above proof-technique could be summarized as follows: we can view competitiveness as a game between A's strategy and an adversary who comes up with very bad inputs to A, knowing ahead of time A's strategy. That is, we can view it as a "game" between a player A, and an adversary. The adversary make the first move by selecting σ_1 and then the player has to process this request. Then the adversary selects (as his second move) σ_2 and the player processes σ_2 and so on. Of course, the adversary knowing A's strategy in advance can simulate the game and therefore can find (ahead of time) and play the most nasty sequence for A.

3 An Optimal Off-Line Algorithm for Paging

In this section, we determine an optimal off-line paging algorithm. We consider an off-line algorithm which we call *longest forward distance* or LFD. When LFD must service a page fault it removes the page, among the pages currently in memory, whose next request comes last.

Theorem 2 (Belady [1]) LFD *is an optimal off-line algorithm for paging.*

Proof:

By contradiction: assume that there is an algorithm MIN superior to LFD. Let $\sigma = \sigma_1\sigma_2\ldots\sigma_n$ be a sequence so that $C_{\mathrm{LFD}}(\sigma) > C_{\mathrm{MIN}}(\sigma)$. At some point in the sequence the two algorithms diverge. Let σ_i be the page request which initiates this divergence. Before processing σ_i the two algorithms have the same pages in memory so that σ_i must be a fault for both. Since they diverge at σ_i, they discard different pages to service σ_i. Let q be the page that LFD discards and p the page that MIN discards. Let t be the first time after i that MIN discards q. We will alter MIN's behavior between σ_i and σ_t to create a new algorithm MIN*. MIN* will service σ_i the same way that LFD does (by removal of q) and will satisfy $C_{\mathrm{MIN}*}(\sigma) \leq C_{\mathrm{MIN}}(\sigma)$.

By the definition of LFD, the next request to p, say σ_a, comes before the next request to q, say σ_b, so that

$$\sigma = \sigma_1\ldots\sigma_i\ldots\sigma_a = p\ldots\sigma_b = q\ldots\sigma_n$$

MIN* services σ_i by discarding q (as does LFD). After processing σ_i, then, MIN* and MIN share exactly $k-1$ pages of memory. As we describe the behavior of MIN* we will argue that, until MIN and MIN* converge, they share exactly $k-1$ pages of memory. MIN* behaves as follows:

CASE **1:** $i < t < b$. We describe how MIN* services a page request σ_l for $i < l < t$. By induction on l, we will show that MIN and MIN* share exactly $k-1$ pages of memory. This implies that there is a unique page e which MIN* has in fast memory but MIN does not. Initially, i.e. for $l = i+1$, this page e is p.

$\sigma_l \neq e$: When $\sigma_l \neq e$, MIN faults exactly when MIN* faults. When there is a fault, MIN* discards the same page that MIN does (recall that MIN does not discard q until time t so that MIN* can always do this). If MIN and MIN* shared $k-1$ pages before the request of σ_l then they share $k-1$ pages afterwards.

$\sigma_l = e$: When $\sigma_l = e$, MIN faults but MIN* does not. In this case, MIN replaces some page with e. Again, if MIN and MIN* shared $k-1$ pages before the request of σ_l then they share $k-1$ pages afterwards.

Finally, to service σ_t, which is a fault for both, MIN* discards e. Since MIN (by definition) discards q, MIN and MIN* again have converged. From the above descriptions, it is clear that $C_{\text{MIN*}}(\sigma) \leq C_{\text{MIN}}(\sigma)$.

CASE **2:** $t \geq b$. In this case, MIN* follows the same strategy as above for σ_l where $i < l < b$. Notice that the same analysis is applicable and, upon arrival at $\sigma_b = q$, MIN and MIN* have exactly $k-1$ pages in common. Further notice that, since p is requested before q, the events described in the ($\sigma_l = e$) portion of the above discussion have taken place at least once so that $C_{\text{MIN*}}(\sigma_1 \ldots \sigma_{b-1}) < C_{\text{MIN}}(\sigma_1 \ldots \sigma_{b-1})$. By definition, MIN does not fault on $\sigma_b = q$. MIN* does and replaces e (the unique page which it has in memory that MIN does not) with q. MIN* and MIN then have the same pages in memory after servicing σ_b. MIN* then behaves as MIN does for the remainder of σ. Since $C_{\text{MIN*}}(\sigma_1 \ldots \sigma_{b-1}) < C_{\text{MIN}}(\sigma_1 \ldots \sigma_{b-1})$, $C_{\text{MIN*}}(\sigma) \leq C_{\text{MIN}}(\sigma)$, as desired.

We conclude that $C_{\text{LFD}}(\sigma_1 \ldots \sigma_i) \leq C_{\text{MIN*}}(\sigma_1 \ldots \sigma_i)$, an optimal strategy, and by induction that $C_{\text{LFD}}(\sigma) \leq C_{\text{MIN}}(\sigma)$. □

4 A Lower Bound on the (Deterministic) Competitive Ratio for the Paging Problem

We show a lower bound of k for the competitive ratio of any deterministic on-line algorithm for the paging problem. We conclude that LRU is an optimal deterministic on-line algorithm for this problem.

Theorem 3 *For all on-line algorithms* A *there is a sequence of requests* $\sigma^{\text{A}} = \sigma_1^{\text{A}} \ldots \sigma_n^{\text{A}}$ *so that* $C_{\text{A}}(\sigma^{\text{A}}) \geq k \cdot C_{\text{LFD}}(\sigma^{\text{A}})$. *This sequence* σ^{A} *can in fact be chosen from a universe of only* $k+1$ *pages.*

Proof:

Given A, let σ_i^{A} be the (unique) page excluded from the k-page memory of A after servicing $\sigma_1^{A} \ldots \sigma_{i-1}^{\text{A}}$. With this sequence A faults on every request. Then, assuming that the memory of A begins in some full initial state, we have that for all prefixes σ of σ^{A}, $C_{\text{A}}(\sigma) = |\sigma|$. For LFD, however, we have:

Lemma 4 *For all finite sequences σ chosen from a universe of $k+1$ pages $C_{\mathrm{LFD}}(\sigma) \leq \frac{|\sigma|}{k}$.*

Proof of Lemma:

Suppose that σ_i induces a page fault for LFD. We show that LFD incurs no page faults on any of $\sigma_{i+1}, \ldots, \sigma_{i+k-1}$. Let p be the page which LFD discards to service σ_i. Since there are exactly $k+1$ pages, the next fault of LFD must be made on a request to p. Notice that every other page in memory (before the service of σ_i) must be requested *before* p is next requested (p is, by definition, requested last among these pages). Hence at least $k-1$ requests separate σ_i from the next fault, as desired. □

Let σ be a prefix of σ^{A} of length kl. Then $C_{\mathrm{A}}(\sigma) = kl \geq k \cdot C_{\mathrm{LFD}}(\sigma)$ so that A is, at best, k-competitive. □

Remark 2 FIFO *is also a k-competitive on-line algorithm for the paging problem and so is also optimal.*

5 Randomized On-Line Algorithms

The on-line algorithms we have discussed up to this point have been deterministic. One shortcoming of such algorithms is that an adversary can always exactly determine the behavior of the algorithm for an input σ. This leads to diabolical inputs like that crafted in the previous theorem. This motivates the introduction of the class of randomized on-line algorithms which will have better behavior in this respect. Informally, a randomized on-line algorithm is simply an on-line algorithm that has access to a random coin. Formally,

Definition 2 *A* randomized on-line algorithm A *is a probability distribution* $\{\mathrm{A}_x\}$ *on a space of deterministic on-line algorithms.*

Notice that when we talk about randomized on-line algorithms, we must decide what information the adversary is allowed to have. If we wish to say that the adversary does not see any coin-flips of the algorithm (or, equivalently, that the adversary must select his "nasty" sequence in advance, when he has no knowledge of the actual pages removed from memory) then we define

Definition 3 *An oblivious adversary knows the distribution on the deterministic on-line algorithms induced by* A*, but has no access to its coin-tosses.*

The above definition also implies that the adversary can not inspect which pages A holds in his fast memory. Equivalently, this means that the oblivious adversary must construct his nasty sequence σ before the game between him and randomized A starts. Thus, notice that against an oblivious adversary, randomization is useful in order to hide the status of the on-line algorithm. The notion of competitiveness is now defined as follows:

Definition 4 *A randomized on-line algorithm* A *distributed over deterministic on-line algorithms* $\{A_x\}$ *has a* competitive ratio *of* α *against any oblivious adversary if*

$$\exists c, \forall \sigma, \operatorname*{Exp}_x[C_{A_x}(\sigma)] \leq \alpha C_{\mathrm{MIN}}(\sigma) + c.$$

The additional constant c is introduced to account for possible differences in the initial configuration of the on-line algorithm and the adversary.

We introduce two randomized on-line paging algorithms:

RANDOM: RANDOM services page faults by discarding a random page, i.e. a page selected uniformly from the pages in fast memory.

MARKING: Initially, all pages are *marked.* When a page p is requested,

1. If p is not in memory then
   - If all pages in memory are marked then unmark all pages
   - Swap p with a uniformly selected unmarked page in memory.
2. Mark page p in memory.

We will first focus on MARKING. We will show that

- By using *randomization*, MARKING achieves a competitive ratio of $2H_k$ against an oblivious adversary, where $H_k = 1 + \frac{1}{2} + \ldots + \frac{1}{k}$ is the k-th harmonic number.
- MARKING's competitive ratio is nearly optimal in that no randomized algorithm has competitive ratio better than H_k. In proving this, we will introduce a general method of proving lower bounds for randomized competitive algorithms against an oblivious adversary.

Both results are due to Fiat et al. [9].

6 Analysis of the MARKING Algorithm

Theorem 5 MARKING *is a* $2H_k$*-competitive paging algorithm against any oblivious adversary, where* $H_k = \sum_{i=1}^{k} \frac{1}{i}$ *is the kth harmonic number.*

In other words, we need to show that

$$\forall \sigma, \mathrm{Exp}[C_{\mathrm{M}}(\sigma)] \leq 2H_k C_{\mathrm{MIN}}(\sigma),$$

where M for conciseness denotes MARKING. This implicitly assumes that M and MIN have initially the same pages in fast memory (otherwise, we would have an additional constant c). To prove this competitiveness, we need to prove an upper bound on the LHS and a lower bound on the RHS, and show that the inequality still holds.

We begin by fixing the input sequence σ and by then dividing it into phases as follows.

- The first phase begins on the first page fault.
- The $i+1$-st phase starts on the request following the last request of phase i.
- If phase p starts on σ_{i_p} then it ends on $\sigma_{i_{p+1}-1}$ where $|\{\sigma_{i_p}, \ldots, \sigma_{i_{p+1}}\}| = k+1$ but $|\{\sigma_{i_p}, \ldots, \sigma_{i_{p+1}-1}\}| = k$.

Thus, in a phase, exactly k distinct pages are requested.

Notice that the initial portion before the first phase costs nothing. Hence the cost of M on σ is the sum of the costs of M on all phases of σ.

Before we go any further let us understand the dynamics of a phase by the following remarks. Let us denote by S_i the set of pages that were in memory just before phase i begins.

Remark 1: Once a page has been accessed in a phase it is marked and hence will remain in memory till the end of the phase. Thus we may pay a price of 1 for a page q only if this is is the first time q is accessed during this phase and if q is not in memory.

Remark 2: We claim that at the end (or the beginning) of every phase the k pages in fast memory are all marked and correspond to the k distinct pages requested during that phase. This can easily be shown by induction. At the beginning of the first phase all pages in fast memory are marked (since no page fault has yet occurred). By induction, one can thus assume that the first request in a phase will cause a page fault (this is true for phase 1 and, for later phases, this follows by induction from the claim applied to the previous phase). Then, during the phase, the k distinct pages requested will be marked and will remain in fast memory until the end of the phase, proving the inductive statement.

Remark 3: Obviously, the definition of a phase does not depend on the coin-tosses of M but only on the input sequence. In other words, the same input sequence will always be divided in exactly the same way into phases regardless of the coin-tosses of M. The coin-tosses only affect the dynamics of M's behavior within a phase. Also (see remark 2), S_i does not depend on the coin-tosses – S_i is simply the k distinct requested pages in phase $i-1$.

By these remarks we can focus on the k distinct pages accessed in phase i; for each such page, we only concentrate on the first time this page is accessed since only such first-time requests can cost us anything. We divide such first time requests into two categories.

- **Clean Requests**: These are requests to pages that did not belong to S_i. We have to pay a price of 1 for each such request regardless of the coin-tosses.
- **Stale Requests**: These are requests to pages that belonged to set S_i at the beginning of the phase. Unfortunately, such a page may have been ejected to make room for a clean page. If it is still around, we pay 0 or else we pay 1. Thus our first major task is to find the expected cost of a stale request.

Since a phase consists of requests to exactly k distinct pages, if there are l clean requests then there must be $k-l$ stale requests.

6.1 The expected cost of a stale request within a phase

Consider a typical stale request σ_j to page p within phase i. Assume there have been s stale requests and c clean requests so far in the phase. We wish to find the expected cost of this request in terms of c and s. To do so, we need the probability that page p is still in memory after s stale and c clean requests. Notice that since this is the first time we are accessing p, p is unmarked.

Now what do we know? We know that the pages requested by the s stale requests are in S_i, by definition of a stale request. Furthermore, since requested pages are marked, the stale pages will not be removed until the end of the phase, therefore they are in fast memory at time $j-1$. This means that both fast memory at time $j-1$ and S_i contain all of these s stale pages. Therefore, the c clean requests now occupy c uniformly distributed slots from the remaining $k-s$ slots. Now we can easily compute the expected cost of a stale request. It is the probability that the requested page is one of the c pages which is no longer in fast memory due to the clean requests. There are $k-s$ candidates for these pages. Hence:

$$\mathrm{Exp}[C_{\mathrm{M}}(\sigma_j)] = \frac{c}{k-s}.$$

6.2 Bounding the total cost of M on a phase

We summarize the last section as follows:

Lemma 6 *The expected cost of the $s+1$-st stale request in a phase is equal to $\frac{c}{k-s}$ where c is the number of preceding clean requests in this phase.*

Suppose we know that there are l_i clean requests within phase i. Recall that l_i is defined purely in terms of the input sequence (see Remark 3) and does not depend on the coin-tosses.

Then there must be $k-l_i$ stale requests within this phase. Clearly the number of clean requests preceding any stale request is no more than l_i. Using this and the last lemma and summing over all $k-l_i$ stale requests the expected cost of all the stale requests on the phase is no more than D where

$$D = \frac{l_i}{k} + \frac{l_i}{k-1} + \ldots + \frac{l_i}{k-(k-l_i-1)}.$$

Also the cost of all all clean requests is exactly l_i since each clean request costs 1. Hence the total expected cost of M on phase i is no more than $D+l_i$. Now

$$D + l_i = l_i\left(1 + \frac{1}{l_i+1} + \frac{1}{l_i+2} + \ldots + \frac{1}{k}\right) \le l_i H_k.$$

In summary:

Lemma 7 *The expected cost of* M *on phase i of σ is no more than $l_i H_k$ where $H_k = 1 + \frac{1}{2} + \ldots + \frac{1}{k}$ is the $k-th$ Harmonic number.*

6.3 Bounding the total cost of any offline algorithm M on a phase

Consider any offline algorithm A. Let us simulate A and M on the same input sequence σ. As before we can divide σ into phases defined by the execution of M. Let us define a potential function Φ_i which is the number of pages in A's memory that are not in M's memory just before Phase i begins. Notice that this potential function is well-defined because the pages in M's memory at the start of Phase i is determined by σ and not by the coin-tosses of M. (see Remark 3 again!).

We know that M receives l_i clean requests in phase i. By the definition of clean requests these were not in M's memory at the start of Phase i. Hence at least $l_i - \Phi_i$ of these pages are also not in A's memory at the start of Phase i. Thus if we let $C_i(\mathrm{A})$ be the cost of A during phase i we have

Lemma 8 $C_i(A) \geq l_i - \Phi_i$.

Next, by definition A has Φ_{i+1} pages at the end of Phase i that are not in M's memory. Thus M has a set P_i of Φ_{i+1} pages at the end of Phase i that are not in A's memory. But each page in M's memory at the end of Phase i was accessed in Phase i. Thus all pages in P_i must have been in $A's$ memory at some time during Phase i but they have been ejected by the end of the phase. Each ejection costs 1 and thus $C_i(\mathrm{A})$ must be at least $|P_i| = \Phi_{i+1}$. Thus:

Lemma 9 $C_i(\mathrm{A}) \geq \Phi_{i+1}$.

We can easily combine Lemmas 8 and 9 to get the more useful bound of $C_i(\mathrm{A}) \geq \frac{1}{2}(l_i - \Phi_i + \Phi_{i+1})$. If we now amortize this bound over the first n phases, we see that

$$\begin{aligned} C(A) &\geq \frac{1}{2}(l_1 - \Phi_1 + \Phi_2 + l_2 - \Phi_2 + \Phi_3 \ldots + l_n - \Phi_n + \Phi_{n+1}) \\ &= \frac{1}{2}(\sum_{i=1}^{n} l_i - \Phi_1 + \Phi_{n+1}). \end{aligned}$$

Since $\Phi_{n+1} \geq 0$ and we assume that $\Phi_1 = 0$, we derive:

Lemma 10 $C(\mathrm{A}) \geq \frac{1}{2}\sum_{i=1}^{n} l_i$

6.4 Establishing the Competitive Factor of M

From Lemma 7 the total expected cost of M on input σ is at most $H_k \sum_i l_i$. Also from Lemma 10 the total cost of any offline algorithm A on input σ is at least $0.5 \sum_i l_i$. Thus M is $2H_k$ competitive.

7 Lower Bound for any Randomized Algorithm

7.1 A useful technique

How do we prove a lower bound on the competitive factor α (even if we allow an additional constant) of a randomized algorithm against an oblivious adversary? We do so by picking a distribution D of input sequences and evaluating the *expected cost* of the best on-line algorithm and the *expected cost* of MIN (i.e. the best offline algorithm) on distribution D. We now develop this method.

Any randomized algorithm can be considered as a distribution A over all possible deterministic on-line algorithms A_x. Let $C_{\mathrm{H}}(\sigma)$ be the cost of deterministic algorithm H on input σ. Thus when we say that A is α-competitive we mean that for all σ,

$$\operatorname*{Exp}_x[C_{\mathrm{A}_x}(\sigma)] \le \alpha C_{\mathrm{MIN}}(\sigma) + c.$$

Let us use σ^j to denote the first j elements in sequence σ. Assume we have a distribution D over the input sequences σ. Fix a j. We can take expectation (over input sequences of length j using distribution D) over both sides of the last equation to get

$$\operatorname*{Exp}_y[\operatorname*{Exp}_x[C_{\mathrm{A}_x}(\sigma_y^j)]] \le \alpha \operatorname*{Exp}_y[C_{\mathrm{MIN}}(\sigma_y^j)] + c.$$

We can use Fubini's theorem to exchange the two expectation quantifiers on the LHS and derive:

$$\operatorname*{Exp}_x[\operatorname*{Exp}_y[C_{\mathrm{A}_x}(\sigma_y^j)]] \le \alpha \operatorname*{Exp}_y[C_{\mathrm{MIN}}(\sigma_y^j)] + c.$$

Let $m_j = Min_{\mathrm{H}}(\mathrm{Exp}_y[C_{\mathrm{H}}(\sigma_y^j)])$ where the minimum is taken over deterministic on-line algorithms H. In other words m_j is the expected cost of the best on-line deterministic algorithm on input sequences distributed according to D. Using this definition in the last equation we have:

$$m_j \le \alpha \operatorname*{Exp}_y[C_{\mathrm{MIN}}(\sigma_y^j)] + c$$

and thus

$$\frac{m_j}{\mathrm{Exp}_y[C_{\mathrm{MIN}}(\sigma_y^j)]} \le \alpha + \frac{c}{\mathrm{Exp}_y[C_{\mathrm{MIN}}(\sigma_y^j)]}.$$

Suppose D is chosen so that $\lim_{j\to\infty} \mathrm{Exp}_y[C_{\mathrm{MIN}}(\sigma_y^j)]$ is ∞. Then:

$$\lim_{j\to\infty} \frac{m_j}{\mathrm{Exp}_y[C_{\mathrm{MIN}}(\sigma_y^j)]} \le \alpha.$$

What is this equation saying? It says that the competitive ratio of any randomized algorithm against an oblivious adversary is at least as big as the ratio of the expected cost of the best deterministic on-line algorithm to the expected cost of the best offline algorithm when evaluated using distribution D over "large enough" input sequences. Notice that we are free to choose D to maximize this ratio.

7.2 Applying the general method to the Paging problem

We will prove the following result using our general method.

Theorem 11 *For the paging problem, if* A *is a randomized on-line* α*-competitive algorithm against an oblivious adversary then* $\alpha \geq H_k$*, even if there are only a total of* $k+1$ *pages that can be requested.*

Proof:

In order to have the inequality $m_j \leq \mathrm{Exp}_x[\mathrm{Exp}_y[C_{\mathrm{A}_x}(\sigma_y^j)]]$ as tight as possible, we choose D such that any deterministic on-line algorithm performs equally badly (in expected cost). In our case this can be obtained by choosing σ_i uniformly among all $k+1$ pages and independent of all previous requests. Since the fast memory only contains k pages, any deterministic algorithm will pay an expected cost of $\frac{1}{k+1}$ for serving σ_i resulting in an expected cost of $\frac{j}{k+1}$. Thus $m_j = \frac{j}{k+1}$.

Our lower bound technique therefore implies that

$$\alpha \geq \lim_{j\to\infty} \frac{j}{(k+1)\,\mathrm{Exp}_y[C_{\mathrm{MIN}}(\sigma_y^j)]}.$$

Using this, we can restate the claim of our theorem as

$$(1) \qquad \lim_{j\to\infty} \frac{j}{\mathrm{Exp}_y[C_{\mathrm{MIN}}(\sigma_y^j)]} = (k+1)H_k.$$

To prove this claim, we need to study the behavior of MIN. We divide σ into stochastic phases. Phase $i, i \geq 0$ consists of the requests indexed in $[X_i, X_i + 1, \ldots, X_{i+1} - 1]$ where $X_0 = 1$ and $X_{i+1} = min\{t : \{\sigma_{X_i}, \sigma_{X_i+1}, \ldots, \sigma_t\} = \{1, \ldots, k+1\}\}$.

Notice that the X_i are random variables since σ is distributed according to D (which in our case is the uniform distribution.) A phase contains requests to only k pages and hence if MIN has a page fault in some phase, the next page fault cannot occur before the next phase. Thus the number of page faults incurred by MIN on σ is at most the number of phases that occur up to time j. And hence the expected cost incurred by MIN is at most the expected number of phases that can occur up to time j. Hence

$$\mathop{\mathrm{Exp}}_y[C_{\mathrm{MIN}}(\sigma_y^j)] \leq 1 + \mathrm{Exp}[\max\{p : X_p \leq j\}].$$

To compute the LHS of (1) we use some results from the theory of stochastic processes. Since the random variables $\{Y_i\} = \{X_{i+1} - X_i : i \geq 0\}$ are independent and identically distributed, they form a so-called "renewal process" and by the elementary renewal theorem we have that

$$\begin{aligned}
\lim_{j\to\infty} \frac{j}{1+\mathrm{Exp}[\max\{p : X_p \le j\}]} &= \lim_{j\to\infty} \frac{j}{\mathrm{Exp}[\max\{p : X_p \le j\}]} \\
&= \mathrm{Exp}[\text{length of phase}] \\
&= \mathrm{Exp}[X_1 - 1] = \mathrm{Exp}[X_1] - 1.
\end{aligned}$$

Intuitively, this says that the expected number of phases is asymptotically equal to the length of the sequence divided by the expected length of a phase. Removing the "expected" this makes perfect sense.

We have therefore shown that

$$\alpha \ge \frac{\mathrm{Exp}[X_1] - 1}{k+1}.$$

We now need to compute $\mathrm{Exp}[X_1]$. By definition, $X_1 = min\{t : \{\sigma_1, \ldots, \sigma_t\} = \{1, \ldots, k+1\}\}$. Each σ_i is uniformly distributed among the $k+1$ pages and is independent of σ_j for any $j \ne i$. Computing $\mathrm{Exp}[X_1]$ under these circumstances is called the "coupon collector problem": Given a collection of $k+1$ distinct coupons (say baseball cards), $\mathrm{Exp}[X_1]$ represents the expected number of independently and uniformly selected coupons a collector has to acquire before obtaining the complete set of distinct coupons. In order to solve this problem, we introduce the random variables Z_i for $1 \le i \le k+1$, where $Z_i = min\{t : |\{\sigma_1, \ldots, \sigma_t\}| = i\}$ for $i = 1, \ldots, k+1$. Thus $Z_1 = 1$ and $Z_{k+1} = X_1$. Note that between Z_i and Z_{i+1}, the probability of getting a new page is $\frac{k+1-i}{k+1}$ at each step. So waiting for a new page between Z_i and Z_{i+1} is a Bernoulli process and the expected waiting time is $\frac{k+1}{k+1-i}$. Hence $\mathrm{Exp}[Z_{i+1} - Z_i] = \frac{k+1}{k+1-i}$. Thus

$$\begin{aligned}
\mathrm{Exp}[X_1] &= \mathrm{Exp}[Z_{k+1}] = \Sigma_{i=1}^{k}(\mathrm{Exp}[Z_{i+1}] - \mathrm{Exp}[Z_i]) + \mathrm{Exp}[Z_1] \\
&= (\sum_{i=1}^{k} \mathrm{Exp}[Z_{i+1} - Z_i]) + 1 \\
&= (k+1)(1 + \frac{1}{2} + \frac{1}{3} + \ldots + \frac{1}{k}) + 1 \\
&= (k+1)H_k + 1.
\end{aligned}$$

So $\alpha \ge \frac{[(k+1)H_k+1]-1}{k+1} = H_k$. This completes the proof of Theorem 11. □

From Theorem 11, we see that MARKING is optimal up to a factor of 2. An optimal (i.e. with competitive ration H_k) randomized on-line algorithm against any oblivious adversary has been obtained by McGeoch and Sleator [17].

8 Types of Adversaries

In this section, we discuss various types of adversaries for on-line algorithms and their relative power. The definitions and results in this and the following section are due to Ben-David, Borodin, Karp, Tardos, and Wigderson [2].

So far, we have analyzed the performance of a randomized on-line algorithm A against an *oblivious* adversary Q. Q knows the distribution over deterministic algorithms that A is using, but must generate a request sequence σ without knowledge of the results of A's coin tosses. We have analyzed A's performance in terms of a "game" between A and Q. Before the beginning of the game, Q selects a number n to be the number of requests $\sigma_1, \sigma_2, \ldots, \sigma_n$ which it will make. Then Q begins to make requests and A begins to process them on-line with "answers" $a_1, a_2, \ldots, a_n$. If A is a randomized algorithm, then we think of A as making coin tosses in order to choose each a_i. But Q must make each request σ_i with *no* knowledge of the previous answers $a_1, \ldots, a_{i-1}$. Thus, the "game" is not really very interesting — Q may as well choose all the σ_i right at the beginning. In order to make the game genuinely interactive, we consider stronger types of adversaries, which have access to the coin tosses of A.

Definition 5 *An* **adaptive adversary**, Q, *against a randomized algorithm,* A, *is an adversary which has access to* A*'s previous coin tosses. That is, when* Q *selects request* σ_i *it has knowledge of* A*'s responses* $a_1, a_2, \ldots, a_{i-1}$.

For an *adaptive* adversary, Q, however, we give Q the task of processing the *same* requests $\sigma_1, \sigma_2, \ldots, \sigma_n$ given the same resources and initial conditions as A. Thus Q incurs a cost, $C_{\mathrm{Q}}(\mathrm{A})$, which is the sum of the costs of Q's responses $q_1, q_2, \ldots, q_n$, even as A's cost, which we now call $C_{\mathrm{A}}(\mathrm{Q})$, is the sum of the cost of A's responses $a_1, a_2, \ldots, a_n$. We wish to define two distinct types of adaptive adversaries based on the way in which they process these requests and the resulting difference in *their* incurred cost, $C_{\mathrm{Q}}(\mathrm{A})$.

Definition 6 *An* **adaptive offline adversary**, Q, *against an algorithm,* A, *is an adversary which makes a request and then gets back* A*'s response before making its next request.* **After** A *has responded to all the requests* $\sigma_1, \sigma_2, \ldots, \sigma_n$*, then* Q *will process these same requests using the optimal* **offline** *strategy,* MIN*. Thus* $C_{\mathrm{Q}}(\mathrm{A}) = C_{\mathrm{MIN}}(\sigma)$*, where* σ *is the sequence of requests which* Q *makes.*

Definition 7 *An* **adaptive on-line adversary**, Q, *against an algorithm,* A, *is an adversary for which* both A and Q *must respond to each request before* Q *decides what the next request will be. The adversary,* Q, *learns of* A*'s response,* a_i*, to each request,* σ_i*, only* after *it has made its own response,* q_i*, but before it has to decide which request* σ_{i+1} *to make next.*

Remark 1: MIN is at least as good as any *on-line* strategy, so for any adaptive *on-line* adversary, there is an adaptive *offline* adversary which does *at least* as well. Thus the concept of an adaptive offline adversary is stronger than that of an adaptive on-line adversary.

Remark 2: The concept of an adaptive on-line adversary is stronger than that of an oblivious adversary.

Consider the case in which A is a randomized algorithm. Then $a_1, a_2, \ldots, a_n$ are random variables to be determined by A's coin tosses. Since $q_2, q_3, \ldots, q_n$ depend on the a_i's which precede them, they too are random variables. Similarly, the costs associated with each of these responses $a_1, a_2, \ldots, a_n, q_2, q_3, \ldots, q_n$ are random variables. Finally, the costs $C_{\mathrm{A}}(\mathrm{Q})$ and $C_{\mathrm{Q}}(\mathrm{A})$ are random variables, since they are formed by summing the costs of the a_i's and q_i's (including q_1), respectively. Thus in considering how effective A is we want to consider the expectations of these quantities, $E[C_{\mathrm{A}}(\mathrm{Q})]$ and $E[C_{\mathrm{Q}}(\mathrm{A})]$.

Definition 8 *A randomized algorithm,* A*, is α-competitive against an adaptive adversary,* Q*, if*

$$E[C_{\mathrm{A}}(\mathrm{Q})] \leq \alpha E[C_{\mathrm{Q}}(\mathrm{A})].$$

Remark 3: If A is deterministic, there is no difference between oblivious, adaptive offline, and adaptive on-line adversaries.

9 Some Results about Competitiveness against Various Types of Adversaries

An adaptive offline adversary is so strong that an algorithm which must confront an adaptive offline adversary is *not helped* by making use of randomization.

Theorem 12 *If there is a randomized on-line algorithm* A *which is α-competitive against any adaptive offline adversary, then there is an α-competitive deterministic on-line algorithm* G.

Proof:

Let A be distributed over deterministic on-line algorithms A_x. So for any adaptive off-line adversary Q,

$$E_x[C_{\mathrm{A}_x}(\mathrm{Q})] - \alpha E_x[C_{\mathrm{Q}}(\mathrm{A}_x)] \leq 0.$$

Again, note the use of expectation in the second term. Q's cost is a random variable as well, based on A's coin tosses.

We now show how to construct a deterministic on-line algorithm G so that $C_{\mathrm{G}}(\sigma) \leq \alpha C_{\mathrm{MIN}}(\sigma)$, for any σ. Assume that the first request to A is σ_1; we can restrict our attention to adversaries that start with σ_1. Maximizing over such adversaries, we obtain:

$$Max_{\mathrm{Q}}\{E_x[C_{\mathrm{A}_x}(\mathrm{Q})] - \alpha E_x[C_{\mathrm{Q}}(\mathrm{A}_x)]\} \leq 0.$$

We can now condition values on the answer a_1:

$$E_{a_1}[Max_{\mathrm{Q}}\{E_x[C_{\mathrm{A}_x}(\mathrm{Q}) \mid a_1] - \alpha E_x[C_{\mathrm{Q}}(\mathrm{A}_x) \mid a_1]\}] \leq 0.$$

Since this expectation is non-positive, there must exist some specific a_1^* for which the value of the expression inside the brackets is non-positive. That is,

$$Max_{\mathrm{Q}}\{E_x[C_{\mathrm{A}_x}(\mathrm{Q}) \mid a_1^*] - \alpha E_x[C_{\mathrm{Q}}(\mathrm{A}_x) \mid a_1^*]\} \leq 0.$$

So our deterministic algorithm G will be one that plays a_1^* on request σ_1. Now Q plays σ_2; again, we restrict our attention to adversaries that play this way. Conditioning now on the choice of a_2,

$$E_{a_2}[Max_{\mathrm{Q}}\{E_x[C_{\mathrm{A}_x}(\mathrm{Q}) \mid a_1^*, a_2] - \alpha E_x[C_{\mathrm{Q}}(\mathrm{A}_x) \mid a_1^*, a_2]\}] \leq 0.$$

So there must be some answer a_2^* for which

$$Max_{\mathrm{Q}}\{E_x[C_{\mathrm{A}_x}(\mathrm{Q}) \mid a_1^* a_2^*] - \alpha E_x[C_{\mathrm{Q}}(\mathrm{A}_x) \mid a_1^* a_2^*]\} \leq 0,$$

the maximum being taken over the adversaries playing σ_1 first and playing σ_2 when the answer to σ_1 is a_1^*. The algorithm G will play a_2^*.

Proceeding in this way, we obtain a sequence of answers $a_1^*, \ldots, a_i^*$ for which

$$Max_{\mathrm{Q}}\{E_x[C_{\mathrm{A}_x}(\mathrm{Q}) \mid a_1^* \cdots a_i^*] - \alpha E_x[C_{\mathrm{Q}}(\mathrm{A}_x) \mid a_1^* \cdots a_i^*]\} \leq 0$$

until request σ_{i+1} is the special "STOP" request. At this point, the first term is the cost of our deterministic algorithm G on $\sigma = \sigma_1 \cdots \sigma_{i+1}$, and the second term is $C_{\mathrm{MIN}}(\sigma)$, since Q can answer the request sequence off-line.

Thus, by always playing an answer which makes the expectation non-positive, G will be α-competitive. □

Remark 4: This is merely a proof that G exists. It does not tell us how we might construct G, but the proof can be made somewhat constructive. We won't go into the details here.

Theorem 13 *If* G *is an α-competitive randomized on-line algorithm against any adaptive* on-line *adversary and there is a β-competitive randomized on-line algorithm, H, against any oblivious adversary, then* G *is an $\alpha\beta$-competitive randomized on-line algorithm against any adaptive* offline *adversary.*

Proof:

Let Q be any adaptive off-line adversary, and G_x, H_y denote the two randomized on-line algorithms with their associated probability distributions. We will also use the notation Q_x to emphasize that the behavior of Q depends on the coin tosses of G. We show a pair of inequalities which together will prove the theorem.

Consider first that Q is playing G_x. The algorithm H_y watches from a distance and plays along. Then Q is simply an oblivious adversary with respect to H_y, so for all x,

$$E_y[C_{\mathrm{H}_y}(\mathrm{Q}_x)] \leq \beta C_{Q_x}(\mathrm{G}_x).$$

Thus we have the first of our inequalities:

$$E_x E_y[C_{\mathrm{H}_y}(\mathrm{Q}_x)] \leq \beta E_x[C_{Q_x}(\mathrm{G}_x)]. \tag{2}$$

Now consider the case in which Q and H join forces to play against G. That is, we form an adaptive *on-line* adversary Q' which generates requests according to the strategy of Q and uses the answers $h_1, h_2, \ldots$ of H. The cost of Q' against G_x is $C_{\mathrm{H}_y}(\mathrm{Q}_x)$, since H_y is being used to provide the answers; meanwhile, the cost of G_x against Q' is simply $C_{\mathrm{G}_x}(\mathrm{Q}_x)$, since Q' uses Q to generate the requests. G is α-competitive against any adaptive on-line adversary, so for all y,

$$E_x[C_{\mathrm{G}_x}(\mathrm{Q}_x)] \leq \alpha E_x[C_{\mathrm{H}_y}(\mathrm{Q}_x)]$$

$$E_y E_x[C_{\mathrm{G}_x}(\mathrm{Q}_x)] \leq \alpha E_y E_x[C_{\mathrm{H}_y}(\mathrm{Q}_x)].$$

The left-hand side is independent of y, and we can swap the order of the expectations on this right-hand side, obtaining the second inequality:

$$E_x[C_{\mathrm{G}_x}(\mathrm{Q}_x)] \leq \alpha E_x E_y[C_{\mathrm{H}_y}(\mathrm{Q}_x)]. \tag{3}$$

Combining these two inequalities,

$$E_x[C_{\mathrm{G}_x}(\mathrm{Q}_x)] \leq \alpha\beta E_x[C_{Q_x}(\mathrm{G}_x)],$$

completing the proof that G is $\alpha\beta$-competitive against Q. □

Corollary 14 *Under the same assumptions, there is a* deterministic *$\alpha\beta$-competitive on-line algorithm.*

Example:

For paging, there is a lower bound of k on the competitive ratio of a *deterministic* on-line algorithm, but there is also an H_k-competitive randomized on-line algorithm against any oblivious adversary. So by the contrapositive of the corollary, there is a lower bound of $\frac{k}{H_k}$ on the competitive ratio of a randomized on-line algorithm against any adaptive on-line adversary.

Corollary 15 *If* G *is an α-competitive randomized on-line algorithm against any adaptive* on-line *adversary, then there is an α^2-competitive deterministic on-line algorithm.*

The corollary is true because if G is α-competitive against any adaptive on-line adversary, then G is α-competitive against any oblivious adversary, and the corollary then follows from the theorem.

This last corollary is quite striking; it says that to show the existence of a competitive deterministic algorithm, we need only construct a randomized algorithm which is competitive against an adaptive on-line adversary.

10 Analysis of Random

In previous sections, we've seen Marking used as a competitive algorithm against an oblivious adversary. An even simpler randomized algorithm is Random, which, when needed, removes a uniformly selected page from the main algorithm's fast memory. We'll show

Theorem 16 (Raghavan and Snir [18]) Random *(*R*) is k-competitive against any adaptive on-line adversary* Q*.*

The proof of the theorem uses a *potential function* argument, which is a fundamental technique in establishing competitive ratios.

Proof:

The proof is based on the idea of a potential function Φ which measures the similarity between the k pages in Q's fast memory and the k pages in R's fast memory.

More precisely, let Q_i be the set of pages in Q's memory just after servicing request σ_i, and similarly for R_i. Then let $\Phi_i = |R_i \cap Q_i|$, the size of the intersection between Q's memory and R's.

Now, let $X_i = C_{\mathrm{R}}(\sigma_i) - kC_{\mathrm{Q}}(\sigma_i) - k(\Phi_i - \Phi_{i-1})$. The first thing to notice is that, for a sequence of requests $\sigma_1, \ldots, \sigma_j$,

$$\sum_{i=1}^{j} X_i = C_{\mathrm{R}}(\mathrm{Q}) - kC_{\mathrm{Q}}(\mathrm{R}) - k(\Phi_j - \Phi_0).$$

If we assume that all algorithms start with the same set of pages in fast memory, then $\Phi_0 = k$. Since no Φ_j can be greater than k, we have

$$\sum_{i=1}^{j} X_i \geq C_{\mathrm{R}}(\mathrm{Q}) - kC_{\mathrm{Q}}(\mathrm{R}),$$

from which, by the linearity of expectations,

$$\sum_{i=1}^{j} E[X_i] \geq E[C_{\mathrm{R}}(\mathrm{Q})] - kE[C_{\mathrm{Q}}(\mathrm{R})].$$

$$\cdots \quad \underbrace{\sigma_b \quad \cdots \quad \rlap{$\overbrace{}^{R_{i-1}}$}\sigma_a \quad \cdots}_{Q_{i-1}} \quad \sigma_c \quad \cdots \quad \sigma_d \quad \cdots$$

Figure 1: Pages of memory.

Thus, if we can establish that $\sum_{i=1}^{j} E[X_i] \leq 0$ then we will have shown that R is k-competitive.

Claim 17 *$E[X_i] \leq 0$ for all X_i.*

It is sufficient to show that the claim holds for any given pair of states Q_{i-1} and R_{i-1}:

$$E[X_i \mid Q_{i-1}, R_{i-1}] \leq 0 \qquad \text{for all states } Q_{i-1} \text{ and } R_{i-1}.$$

We will establish this by cases, considering where σ_i lies relative to Q_{i-1}, R_{i-1} and their intersection. For illustration, see Figure 1.

1. $\sigma_i \in (Q_{i-1} \cap R_{i-1})$
   The case is that of σ_a in Figure 1. Here there is no cost for either Q or R, since the page is already in both of their memories, so $C_{\mathrm{R}}(\sigma_i) = C_{\mathrm{Q}}(\sigma_i) = 0$. Since neither algorithm alters its memory, $\Phi_i = \Phi_{i-1}$, so $X_i = 0$, and thus
   $$E[X_i \mid Q_{i-1}, R_{i-1}] = 0.$$

2. $\sigma_i \in (Q_{i-1} \setminus R_{i-1})$
   This is like σ_b in Figure 1. In this case only R needs to do any work, so $C_{\mathrm{Q}}(\sigma_i) = 0$ and $C_{\mathrm{R}}(\sigma_i) = 1$. When R chooses the page to eliminate from fast memory it chooses it uniformly, choosing a page in Q_{i-1} with probability Φ_{i-1}/k. If the chosen page is in Q_{i-1} then one page will be eliminated from $Q_{i-1} \cap R_{i-1}$ and one added, so $\Phi_i = \Phi_{i-1}$. If not, then $\Phi_i = \Phi_{i-1} + 1$, since no page has been eliminated from the intersection. Thus
   $$E[\Phi_i - \Phi_{i-1} \mid Q_{i-1}, R_{i-1}] = 1 - \frac{\Phi_{i-1}}{k},$$
   so
   $$E[X_i \mid Q_{i-1}, R_{i-1}] = 1 - 0 - k\left(1 - \frac{\Phi_{i-1}}{k}\right) = 1 - k - \Phi_{i-1}.$$
   Since there was at least one page (σ_i) not in $Q_{i-1} \cap R_{i-1}$ we know that $\Phi_{i-1} < k$, so
   $$E[X_i \mid Q_{i-1}, R_{i-1}] \leq 0.$$

3. $\sigma_i \in (R_{i-1} \setminus Q_{i-1})$
   This is like σ_c in Figure 1. Since only Q needs to do any work, $C_{\mathrm{Q}}(\sigma_i) = 1$ and $C_{\mathrm{R}}(\sigma_i) = 0$. Now, whatever Q does in this case, the potential Φ_i will not decrease; as in the previous case, it will either stay the same or increase by one, so $\Phi_i - \Phi_{i-1} \geq 0$. Thus $X_i \leq 0 - k - k(0) < 0$, so
   $$E[X_i \mid Q_{i-1}, R_{i-1}] < 0.$$

4. $\sigma_i \notin (R_{i-1} \cup Q_{i-1})$
   This is the case of σ_d in Figure 1. Since both Q and R must pull σ_i into fast memory, $C_{\mathrm{Q}}(\sigma_i) = C_{\mathrm{R}}(\sigma_i) = 1$. Unlike the previous cases, Φ_i may here either stay the same, go up by one, or go down by one. Now, let's calculate the probability that Φ_i decreases by one. We have
   $$\Pr[\Phi_i - \Phi_{i-1} \geq 0 \mid Q_{i-1}, R_{i-1}] \geq \frac{1}{k}.$$
   To see this, fix the page that Q chooses to discard. If the page is not in $(Q_{i-1} \cap R_{i-1})$ then, with probability 1, Φ_i does not decrease. If the page is in $(Q_{i-1} \cap R_{i-1})$ then R will choose to discard the same page with probability $1/k$. Thus the probability in both cases is at least $1/k$. This gives us
   $$\Pr[\Phi_i - \Phi_{i-1} < 0 \mid Q_{i-1}, R_{i-1}] \leq 1 - \frac{1}{k},$$
   so
   $$E[\Phi_i - \Phi_{i-1} \mid Q_{i-1}, R_{i-1}] \geq (-1)(1 - \frac{1}{k}).$$
   This is enough to give us
   $$E[X_i \mid Q_{i-1}, R_{i-1}] \leq 1 - k(1) - k(-1)\left(1 - \frac{1}{k}\right) = 0.$$

$\diamond$

Since $E[X_i \mid Q_{i-1}, R_{i-1}] \leq 0$ for all Q_{i-1} and R_{i-1}, $E[X_i] \leq 0$, so
$$0 \geq \sum_{i=1}^{j} E[X_i] \geq C_{\mathrm{R}}(\mathrm{Q}) - kC_{\mathrm{Q}}(\mathrm{R})$$
which proves the theorem. $\square$

Note that we have assumed for this analysis that the adversary doesn't change its memory when it doesn't have to. Though we don't prove it here, it can be shown that this type of *lazy* adversary is at least as strong as any other, so the assumption is harmless.

11 The k–Server Problem

So far, we have been dealing with various concepts related to on-line algorithms, as well as the paging problem. In this and subsequent sections, we consider another famous problem for its apparent simplicity, the k-server problem. However, as it will appear, much less is known for the k-server problem than for the paging problem, although it has been extensively studied in the last few years.

The following is a practical analogy for the k–server problem. Suppose we have a city with a set of k police cars. When an emergency occurs somewhere in the city, one police car is chosen and dispatched to the site. We ignore the length of time for the car to travel to the site, ignore the length of time for the emergency to be handled, and count only the distance the police car must travel. Another emergency occurs and some car, maybe the same one, is chosen to travel to the next site. The emergencies must be attended to in order, and the dispatcher does not know in advance how many emergencies will occur where. Again, the cost function (to be minimized) is simply the sum of distances that the cars must travel, over all the emergencies. To formally analyze the k–server problem, we need to define a *metric space.*

Definition 9 *A* metric space *is a set of points V along with a distance function $d : (V \times V) \rightarrow \mathbb{R}$ such that*

1. $d(u, v) \geq 0, \ \ \forall u, v \in V.$
2. $d(u, v) = 0$ *iff* $u = v$
3. $d(u, v) = d(v, u), \ \ \forall u, v \in V$
4. $d(u, v) + d(v, w) \geq d(u, w), \ \ \forall u, v, w \in V.$

In other words, d is nonnegative, strictly positive between different points, satisfies the triangle inequality, and is symmetric. Often for purposes of this problem V is finite, but this is not obligatory.

Definition 10 (The k-server problem) *The input is a metric space V, a set of k "servers" located at points in V, and a stream of requests $\sigma_1, \sigma_2, \ldots,$ each of which is a point in V. For each request, one at a time, you must move some server from its present location to the requested point. The goal is to minimize the total distance travelled by all servers over the course of the stream of requests.*

In general, the optimal solution to a k–server problem can be computed off-line by dynamic programming. (Dynamic programming can be done in polynomial time.) Another possible approach is to construct a graph such that the n requests and the k servers correspond to $n + k$ distinct vertices. For those familiar with the theory of network flows, the off-line problem can be solved as a minimum-cost flow problem on n copies of the graph. One must construct appropriate weighted edges. The class is invited to think about this as an exercise.

Claim 18 *For any stream of requests, on-line or off-line, only one server needs to be moved at each request.*

To show a contradiction, assume otherwise. In response to some request, σ_i, in your stream, you move server j to point σ_i and, in order to minimize the overall cost, you also move server k to some other location, perhaps to "cover ground" because of j's move. If server k is never again used, then the extra move is a waste, so assume server k is used for some subsequent request σ_m. However, by the triangle inequality (see the definition of a metric space), server k could have gone directly from its original location to the point σ_m at no more cost than stopping at the intermediate position after request σ_i.

We might, in general, consider algorithms which move more than one server at a time. While we will sometimes do this for purposes of analysis, we can in general ignore such algorithms thanks to the triangle inequality.

11.1 Special Cases of the k–Server Problem

1. Paging.
   The paging problem is a special case of the k-server problem, in which the k servers are the k slots of fast memory, V is the set of pages and $d(u, v) = 1$ for $u \neq v$. In other words, paging is just the k–server problem but with a uniform distance metric.

2. Two-headed Disk.
   You have a disk with concentric tracks. Two disk-heads can be moved linearly from track to track. The two heads are never moved to the same location and

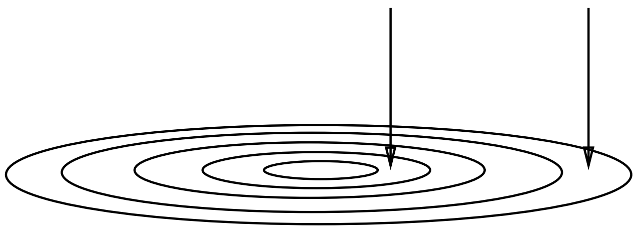


Figure 2: Two-headed disk.

   need never cross. The metric is the sum of the linear distances the two heads have to move to service all the disk's I/O requests. Note that the two heads move exclusively on the line that is half the circumference and the disk spins to give access to the full area.

11.2 Summary of known results

We already have results about paging that can be used for the k–server problem. Positive results about paging can't be generalized, but since paging is a case of the k–server problem, the lower bound for paging also applies in general.

Theorem 19 (Manasse-McGeoch-Sleator [16]) *For no metric spaces is there a deterministic α-competitive algorithm for the k-server problem such that $\alpha < k$.*

The general proof is similar to the previous proof with a uniform distance metric. However, in the case of paging we actually have a k-competitive deterministic on-line algorithm.

Open Problem: Whether or not there exists a k–competitive deterministic algorithm for the general k–server problem. In fact, for some time it was unknown whether there exists a $f(k)$-competitive algorithm, where $f(k)$ is a function of k.

Theorem 20 *Fiat, Rabani and Ravid [10] proved by induction on k that there is an $f(k)$-competitive algorithm for the general k–server problem, where $f(k) = e^{O(k \log k)}$.*

Very recently, it was shown by Koutsoupias and Papadimitriou [15] that the so-called *work-function algorithm* of Chrobak and Larmore [7] is $2k - 1$-competitive. It is an open problem whether the same algorithm is in fact k-competitive.

11.2.1 Results for specific k

$k = 1$ This case is trivial, since there is never any choice about which server to send. Thus any reasonable algorithm is 1-competitive.

$k = 2$ There is a 2-competitive deterministic algorithm for this case, due to Manasse, McGeoch and Sleator [16]. Even with this small k, the algorithm is nontrivial and would make a challenging exercise. The work function algorithm is also known to be 2-competitive [7]. There has also been work on developing competitive algorithms for 2 servers which are "fast" in the sense that they perform only a constant amount of work per request [12, 4, 14].

$k = |V| - 1$. Then the algorithm Balance, described later, is k-competitive [16]. A variation of Balance is also known to be 10-competitive for the case $k = 2$ (Irani and Rubinfeld [12]).

11.2.2 Results for specific metric spaces

paging For paging, we've already seen a k-competitive algorithm.

line, tree For these configurations, there is a k-competitive deterministic algorithm [5].

circle For points on the circle, there is a deterministic k^3-competitive algorithm.

11.2.3 Greedy

The most obvious on-line algorithm for the k-server problem is GREEDY, in which a given request is serviced by whichever server is closest at the time. The following example, however, shows the major flaw in this algorithm:

Consider two servers 1 and 2 and two additional points A and b, positioned as follows (assume something like a Euclidean metric):

$$\bullet^1 \qquad\qquad\qquad \bullet^2 \quad \circ^a \quad \circ^b$$

Now take a sequence of requests $ababab\ldots$. GREEDY will attempt to service all requests with server 2, since 2 will always be closest to both A and b, whereas an algorithm which moves 1 to A and 2 to b, or vice versa, will suffer no cost beyond that initial movement. Thus GREEDY can't be α-competitive for any α.

11.2.4 The BALANCE Algorithm

We now describe the BALANCE algorithm mentioned above. At all times, we keep track of the total distance travelled so far by each server, D_{server}, and try to "even out" the workload among the servers. When request i arrives, it is serviced by whichever server, x, minimizes the quantity $D_x + d(x,i)$, where D_x is the distance travelled so far by server x, and $d(x,i)$ is the distance x would have to travel to service request i. BALANCE is k-competitive when $|V| = k+1$. However, it is not even competitive for $k = 2$. Consider indeed the following instance suggested by Jon Kleinberg. The metric space corresponds to a rectangle $abcd$ where $d(a,b) = d(c,d) = \alpha$ is much smaller than $d(b,c) = d(a,d) = \beta$. If the sequence of requests is $abcdabcd\ldots$, the cost of BALANCE is β per request, while the cost of MIN is α per request.

A slight variation of BALANCE in which one minimizes $D_x + 2d(x,i)$ can be shown to be 10-competitive for $k = 2$ [12].

12 The Randomized Algorithm, HARMONIC

While GREEDY doesn't work very well on its own, the intuition of sending the closest server can be useful if we randomize it slightly. Instead of sending the closest server every time, we can send a given server with probability inversely proportional to its distance from the request.

Thus for a request A we can try sending a server at x with probability $1/(Nd(x,a))$ for some N. Since, if On is the set of on-line servers we want

$$1 = \sum_{x \in \mathrm{On}} \frac{1}{Nd(x,a)}$$

we set

$$N = \sum_{x \in \mathrm{On}} \frac{1}{d(x,a)}$$

This algorithm is known as the Harmonic algorithm (H). Note, once again, that in the special case of paging this is identical to Random.

Harmonic is competitive as stated below.

Theorem 21 (Grove [11]) Harmonic *is* $(\frac{5}{4}k2^k - 2k)$*-competitive against an adaptive on-line adversary.*

Before this fairly recent result, Harmonic was only known to be 3^{17000}-competitive for $k = 3$ [3] and 3-competitive for 2 servers [6].

There is a known lower bound for this algorithm. Specifically, for some adaptive on-line adversary Q, we know that $C_H(\mathrm{Q}) = \binom{k+1}{2} C_{\mathrm{Q}}(\mathrm{H})$, (where H denotes Harmonic) so we can't hope to do better than $\binom{k+1}{2}$-competitiveness. It is open whether the competitive factor for Harmonic is indeed equal to this lower bound.

Finally, recall that we can get an α^2-competitive deterministic on-line algorithm from an α-competitive randomized algorithm. So this theorem will also give us a $(\frac{5}{4}k2^k - 2k)^2$-competitive deterministic algorithm, although the algorithm that we get this way is far from efficient.

12.1 Analysis of Harmonic

We prove here a slightly weaker result than Grove's result.

Theorem 22 Harmonic *is* $k^2 2^{k-1}$*-competitive against an adaptive online adversary.*

Proof:

In the following, let OFF denote both the servers of the online adaptive adversary which Harmonic plays against, as well as their locations in the metric space.

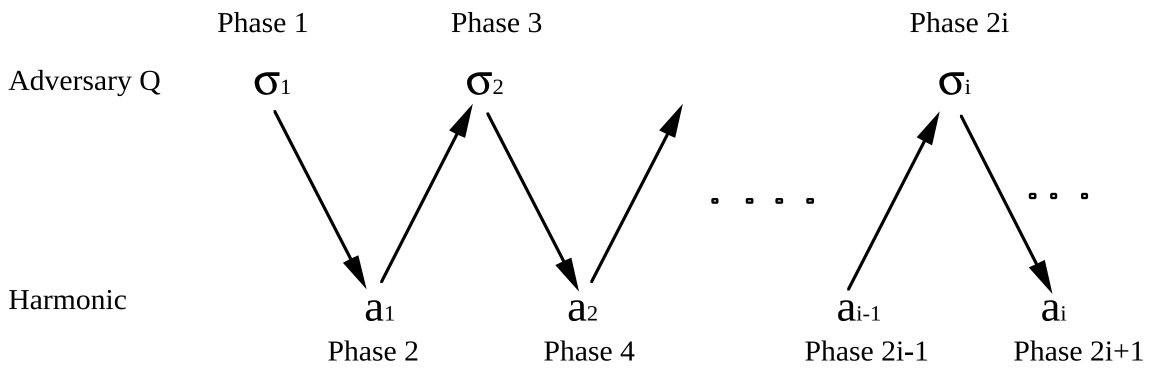


Figure 3: Odd and even phases of OFF vs. ON.

We define phases (Figure 3) by:

- In the odd phase $2i-1$, the online adversary moves a server to vertex A at cost d_i. The adversary then makes a request σ_i at vertex A. (Note that OFF can move its server before making the request because it is an online adversary, so the order makes no difference).

- In the even phase $2i$, the player moves an online server $a_i \in$ ON to A and incurs a cost of l_i. l_i is in fact a random variable since HARMONIC chooses the server to move probabilistically.

The actions of ON and OFF taken during one pair of odd/even phases are illustrated in Figure 4.

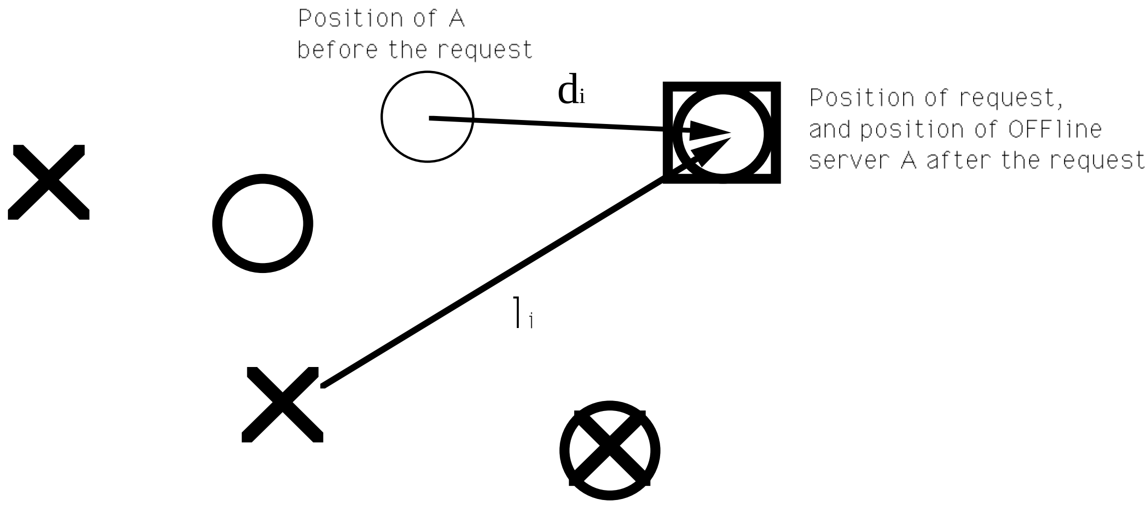


Figure 4: The movement of servers. X's denote the servers of ON while O's denote the servers of OFF.

12.1.1 Analysis by Means of a Potential Function

To aid in the analysis of HARMONIC, we introduce a potential function Φ. Let $\Phi(j)$ denote the value of Φ at the end of phase j. The potential function will be required to satisfy the following properties.

1. $\Phi(0) = 0$, $\Phi(j) \geq 0$.

2. $\Phi(2i-1) - \Phi(2i-2) \leq k^2 2^{k-1} d_i$.

3. $E[l_i + \Phi(2i) - \Phi(2i-1)] \leq 0$.

These properties allow us to prove the competitive ratio of $k^2 2^{k-1}$ since, by summing 2. and 3., we obtain:

$$\begin{aligned}
E[\textstyle\sum_{i=1}^{j} l_i] &= \sum_{i=1}^{j} E[l_i] \\
&= \sum_{i=1}^{j}(E[l_i + E[\Phi(2i) - \Phi(2i-1)]])g + \sum_{i=1}^{j} E[\Phi(2i-1) - \Phi(2i-2)] \\
&\quad -E[\Phi(2j)] + E[\Phi(0)] \\
&\leq k^2 2^{k-1} \sum_{i=1}^{j} E[d_i] \\
&= k^2 2^{k-1} E[\sum_{i=1}^{j} d_i].
\end{aligned}$$

12.1.2 Derivation of the Potential Function Φ

We now obtain a potential function satisfying the above constraints. Φ will depend on three factors:

1. The vertices in ON.
2. The vertices in OFF.
3. The past history of moves by ON and OFF.

Φ is obtained by means of a matching M mapping the vertices in ON bijectively to the vertices in OFF. If $x \in$ ON then we denote by $M(x) \in$ OFF the server in OFF matched to x.

Initially we impose that $x = M(x)$ for all servers x in ON (since ON and OFF start with the same initial configuration of servers). This implies that $\Phi(0) = 0$.

In an odd phase (i.e. when a server from OFF is moved, the matching is kept unchanged.

When (during an even phase) a server in ON is moved from vertex b to the position of the OFFline server A (where a request occurs and with $M(a) = A$), the matching is updated by the following rule: If ON moves the server b which was originally matched to server B from OFF, then M is updated according to $M(a) = B$ and $M(b) = A$.

We can now define Φ by:

$$\Phi = \sum_{x \in \mathrm{ON}} \lambda(x)$$

where for $\Gamma \subseteq$ OFF we define the radius by:

$$R(x, \Gamma) = \max_{Y \in \Gamma} d(x, Y)$$

and

$$\lambda(x) = k2^k \min_{\Gamma \subseteq \mathrm{OFF}} \left\{ \frac{R(x,\Gamma)}{2^{|\Gamma|}} : M(x) \in \Gamma \right\}.$$

Also define Γ^*_x to be the argmin of the term in the definition of $\lambda(x)$.

Note that $\lambda(x) \geq kd(x, M(x))$ for all x since $2^{|\Gamma|} \leq 2^k$ and $R(x,\Gamma) \geq d(x, M(x))$ for any $\Gamma \ni M(x)$.

Two final definitions are:

$$\rho(x) = R(x, \Gamma^*_x) \text{ and } n(x) = |\Gamma^*|.$$

So $\lambda(x) = k2^k \frac{\rho(x)}{2^{n(x)}}$.

12.1.3 Why we Argue on Matchings

A small example now suffices to provide an intuitive feeling for why matchings provide a tool for measuring the potential difference between ON and OFF. Suppose that there are seven servers positioned as shown in Figure 5. If the adversary consistently makes requests to the poorly matched server on the right, then the adversary will pay 0 cost, while Harmonic will keep paying until it moves a server from the left side of the network. In the matching, one edge will be large, and so both $R(x, \Gamma^*_x)$ and $\lambda(x)$ will be large until such a move is made.

Figure 5: An example of a configuration of OFF's servers (O's) and ON's servers (X's). If the adversary requests at the unmatched OFFline server on the right, then Harmonic eventually pays a high cost while the adversary pays 0.

12.1.4 Proof of Properties about Φ

Property 1

Initially $\Gamma^*_x = \{x\}$ so $\rho(x) = 0$ which means $\lambda(x) = 0$ and $\Phi(0) = 0$. It is clear that $\Phi(j) \geq 0$.

Property 2

In an odd phase, the adversary moves a server in OFF by a distance d_i. Let $\lambda'(x)$ be the value of λ after the move. Then

$$\lambda'(x) \leq k2^k \frac{R'(x, \Gamma^*_x)}{2^{\Gamma^*_x}}$$

where Γ_x^* is the value from before the move and:

$$\begin{aligned} R'(x,\Gamma_x^*) &= \max_{Y\in\Gamma_x^*} d'(x,Y) \\ &\leq \max_{Y\in\Gamma_x^*}\{d(x,Y)\} + d_i \\ &= R(x,\Gamma_x^*) + d_i. \end{aligned}$$

Note that the distance function is primed to indicate that Y may have moved during the phase. However, the matching is not altered during an odd phase, so we were able to use the same Γ_x^* (since $M(x)$ still belongs to Γ_x^*).

Since $|\Gamma_x^*|$ is at least one, we now can calculate that:

$$\begin{aligned} \lambda'(x) &\leq k2^k \frac{R'(x,\Gamma_x^*)}{2^{|\Gamma_x^*|}} \\ &\leq \lambda(x) + k2^k \frac{d_i}{2^{|\Gamma_x^*|}} \\ &\leq \lambda(x) + k2^{k-1} d_i. \end{aligned}$$

Therefore,

$$\begin{aligned} \Phi(2i-1) - \Phi(2i-2) &= \textstyle\sum_{x\in \mathrm{ON}}(\lambda'(x) - \lambda(x)) \\ &\leq k(k2^{k-1}d_i) \\ &\leq k^2 2^{k-1} d_i. \end{aligned}$$

Property 3

We now need to show that $E[l_i + \Phi(2i) - \Phi(2i-1)] \leq 0$. In the rest of this proof, we shall be conditioning on P, the current position of the servers in ON and OFF. Clearly, if the inequality is true while conditioning on P, then it will remain true when we take the expectation over P. First consider what $E[l_i|P]$ represents. It is the expected distance that the server moved by Harmonic will have to travel. By the definition of Harmonic, we know that, given P, $x \in$ ON is moved with probability $\frac{1}{Nd(x,A)}$.

Therefore,

$$\begin{aligned} E[l_i|P] &= \textstyle\sum_{x\in \mathrm{ON}} \frac{1}{Nd(x,A)} d(x,A) \\ &= \frac{k}{N} \end{aligned}$$

since ON contains k servers.

So for Φ to satisfy property 3, we must show that

$$E[\Phi(2i) - \Phi(2i-1)|P] \leq -E[l_i|P] = -\frac{k}{N}.$$

Because we are in an even phase, OFF already has a server at A. We will examine the changes in the matching depending on which server ON moves in order to approach this problem. Each of the cases has a probability attached, and so the results of each case can be used to determine an expected value for $\Phi(2i) - \Phi(2i-1)$. Note that in the following analysis the servers of OFF are fixed, since the adversary has already moved its server during the odd phase.

Case 1 (occuring with probability $\frac{1}{Nd(a,A)}$):

ON moves server a to the request at A (where $M(a) = A$ before the phase).

Then after the move, $\lambda'(a) = 0$ and $\lambda'(c) = \lambda(c)$ for all $c \neq a$. Therefore,

$$\Phi(2i) - \Phi(2i-1) = -\lambda(a) = -k2^k \frac{\rho(a)}{2^{n(a)}}.$$

Case 2:

ON moves a server $b \neq a$ where $M(b) = B$ before the move.

Case i: $d(a, B) \leq \rho(a)$.

Then $\lambda'(c) = \lambda(c)$ for $c \neq a, c \neq b$. Moreover, $\lambda'(b) = 0$. And lastly, $\lambda'(a) \leq \lambda(a)$ since the fact that $d(a, B) \leq \rho(a)$ implies that B (the new OFFline server matched to a, belongs to Γ_a^*) (see Figure 6). Therefore, $\Phi(2i) - \Phi(2i-1) \leq 0$.

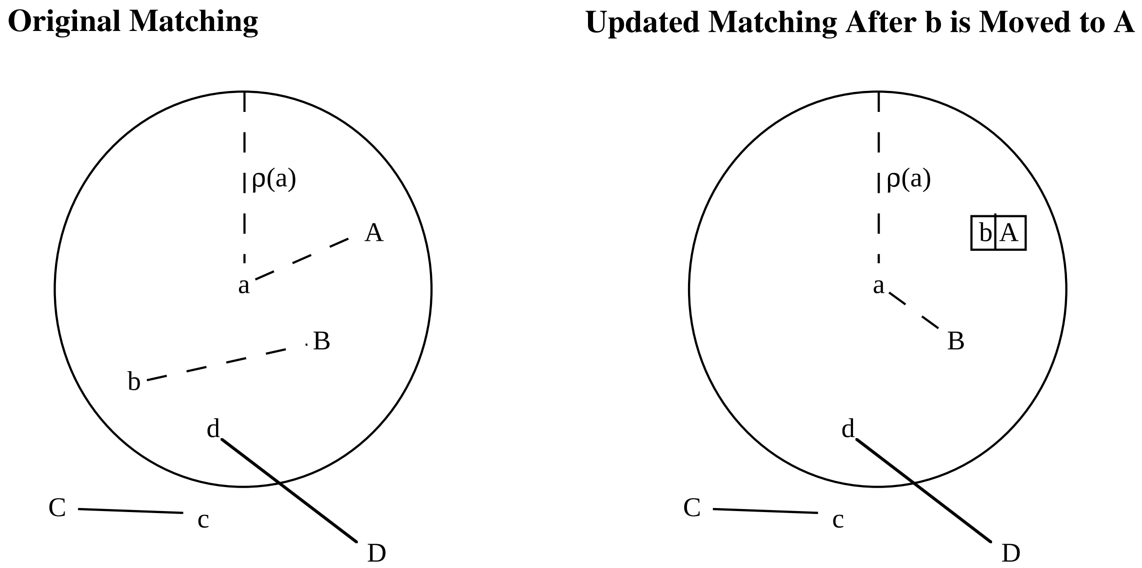


Figure 6: The configuration of the matching when $d(a, B) \leq \rho(a)$, and when the matching is updated.

Case ii: $d(a, B) > \rho(a)$.

Now define l so that B is the lth closest OFF server to a from outside of the disk of radius $\rho(a)$ centered at a. Consider a disk centered at a of radius $\rho(a) + d(A, b) + \rho(b)$. (See Figure 7). Clearly B is in that disk, since

$$d(a, B) \leq d(a, A) + d(A, b) + d(b, B) \leq \rho(a) + d(A, b) + \rho(b).$$

In fact, this big disk contains at least $\max(n(a)+l, n(b))$ servers from OFF. (i.e. $n(a)+l$ is all OFF servers in the disk centered at a with radius $d(a,B)$ and $n(b)$ is all servers in the disk of radius $\rho(b)$ centered at b).

Therefore,

$$\begin{aligned}
\lambda'(a) &\leq k2^k \tfrac{R(a,\Gamma)}{2^{|\Gamma|}} \\
&\leq k2^k \frac{\rho(a)+d(A,b)+\rho(b)}{2^{\max(n(a)+l,n(b))}} \\
&\leq k2^k [\frac{\rho(a)}{2^{n(a)}} + \frac{d(A,b)}{2^{n(a)+l}} + \frac{\rho(b)}{2^{n(b)}}] \\
&= \lambda(a) + k2^k \frac{d(A,b)}{2^{n(a)+l}} + \lambda(b).
\end{aligned}$$

Furthermore, $\lambda'(b) = 0$ since b is moved to the position A of the request and $M'(b) = A$. Lastly, $\lambda'(c) = \lambda(c)$ for $c \neq a, c \neq b$. This means that in this case when b is the server moved to A:

$$\Phi(2i) - \Phi(2i-1) \leq \frac{k2^k d(A,b)}{2^{n(a)+l}}.$$

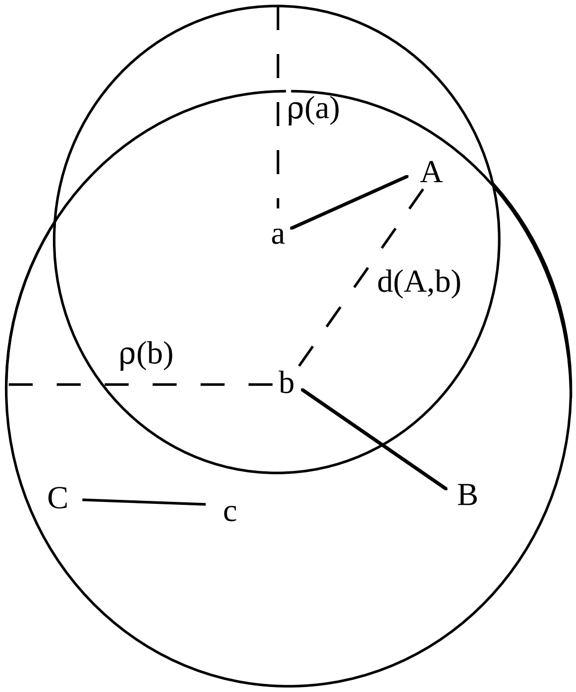


Figure 7: The configuration of the matching when $d(a,B) > \rho(a)$.

12.1.5 The Expected Value for $\Phi(2i) - \Phi(2i-1)$ Over All Cases

Weighting the three cases presented above by the probability of their occurences, we have that:

$$\begin{aligned}
E[\Phi(2i) - \Phi(2i-1)] &\leq \frac{1}{Nd(a,A)}\left(-\frac{k2^k\rho(a)}{2^{n(a)}}\right) + \sum_{l=1}^{k-n(a)} \frac{1}{Nd(A,b_l)}\left(\frac{k2^k d(A,b_l)}{2^{n(a)+l}}\right) \\
&\leq \frac{k2^k}{N2^{n(a)}}\left[-1 + \sum_{l=1}^{k-n(a)} \frac{1}{2^l}\right] \\
&= -\tfrac{k}{N} \text{ (geometric series)}
\end{aligned}$$

where b_l is the server which before the move has $M(b) = B$, where B is the lth closest OFF server to a from outside the disk of radius $\rho(a)$ centered at a. Note that the sum is to $k - n(a)$ which is the number of OFF servers outside of the disk of radius $\rho(a)$ centered at a.

This is the result we wanted.

□

12.1.6 The Intuition

Recall that $\lambda(x) = k2^k \min_{\Gamma} \dfrac{R(x,\Gamma)}{2^{|\Gamma|}} = k2^k \dfrac{\rho(A)}{2^{n(a)}}$.

Here is some intuition behind the use of the use of the exponential factor $2^{|\Gamma|}$. If $n(a)$ is large then there are not many OFF servers outside of $\rho(a)$ so the $2^{n(a)}$ term is large, making $\lambda(a)$ smaller. On the other hand, if $n(a)$ is small, then there are many OFF servers outside of the circle, and these contribute to a big potential function by making $\frac{2^k}{2^{n(a)}}$ big. These two cases correspond to the cost we could expect. If $n(a)$ is large, then HARMONIC will have a reasonably high probability of moving an ONline server whose matched OFFline server is in the disk centered at a of radius $\rho(a)$ to cover a request at A. This would result in a "small" expected cost. However, by a similar argument, we would expect a larger expected cost if $n(a)$ is small.

13 A k-competitive, deterministic algorithm for trees

The results of this section are due to Chrobak and Larmore [5]. Let (V, E) be a tree (an undirected graph without cycles) with a positive distance function on each edge. We view each edge, e, as an actual segment of length $d(e)$. Let W denote the infinite set of all points on all edges in E, and let requests be any points in W. (The cost of travelling from an endpoint of e to a non-vertex point in edge e will simply be the fraction of $d(e)$ proportional to the location of the point from the end.) The algorithm presented will of course also apply to the discrete case where all requests are vertices.

Definition 11 *At any time, there is some request σ which we are trying to service. We say that a server is* active *if there is no server located between it and σ. If there are several servers located at the same point, with no servers located between them and σ, we pick exactly one of them to be active. We can do this deterministically according to some ordering of the servers if we like, calling the highest-priority server the active one. Note that this definition makes sense because the (acyclic) tree structure means that there is exactly 1 path between any two points.*

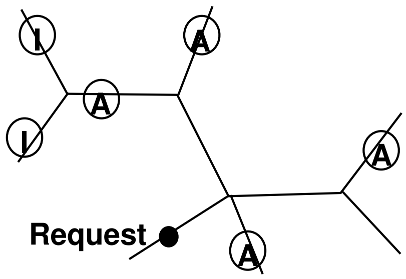


Figure 8: Active servers are marked with an A, inactive ones with an I. Note that if all servers move with a constant speed, the closest one (at the bottom) will reach the request and the others will become inactive somewhere on the way to the request.

Our algorithm A is simple to describe: to service a request i, all active servers move towards i at constant speed, each one stopping when one of the following is true

1. the server reaches the destination, i.
2. the server is eclipsed by another server and becomes inactive;

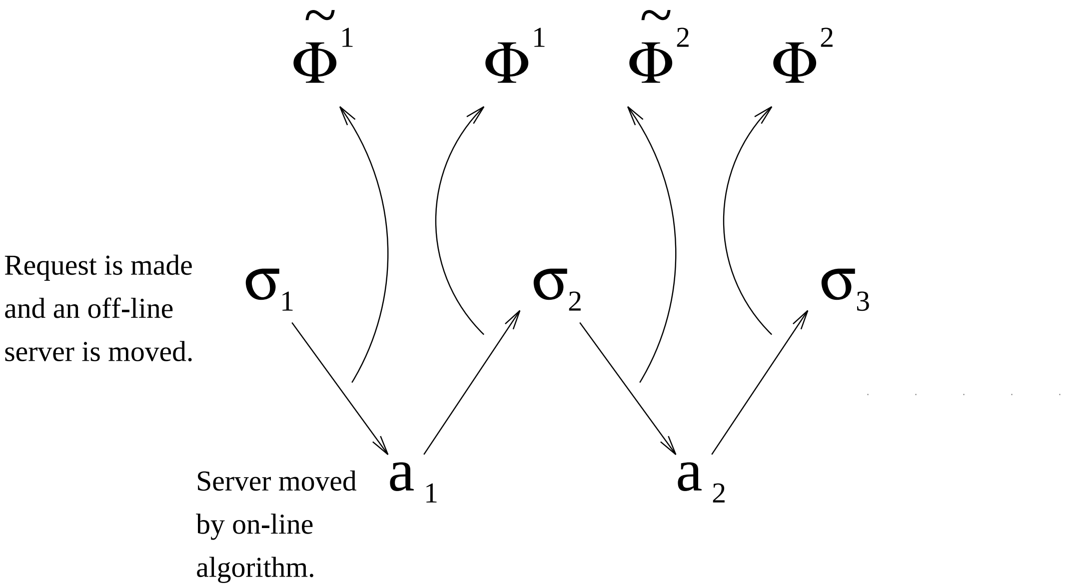
According to the second condition, a server that is active at the beginning of a request might not be active later on in the request. As soon as a server sees another server between it and i, it becomes inactive and stops moving.

13.1 Proof of k-competitiveness

We will use a potential function to show that, for any sequence $\sigma_1, \sigma_2, \ldots$ of requests, $C_A(\sigma) \leq kC_{\text{MIN}}(\sigma)$. Let Φ^t be the value of the potential function after the on-line algorithm has processed σ_t. Let $\tilde{\Phi}^t$ be the value of the potential function before the on-line algorithm has processed σ_t but after the off-line algorithm has processed it. Suppose that the on-line servers are at postions $s_1, s_2, \ldots, s_k$ and the off-line servers are at positions $a_1, a_2, \ldots, a_k$. We define the potential funtion by

$$\Phi = \sum_{i<j} d(s_i, s_j) + kM(s, a).$$

where $M(s, a)$ is the minimum cost matching between the on-line servers and the off-line servers. i.e. $M(s, a) = \min_\tau \sum_{i=1}^{k} d(a_i, s_{\tau(i)})$ where the minimum is taken over all permutations τ. Note that this is a potential function which often arises in the analysis of algorithms for the k-server problem.



Claim 23

$$\begin{aligned} \tilde{\Phi}^t - \Phi^{t-1} &\leq kC_{\text{MIN}}(\sigma_t), \\ \Phi^t - \tilde{\Phi}^t &\leq -C_A(\sigma_t). \end{aligned}$$

where A is the on-line algorithm. From this claim we can derive that $\Phi^t - \Phi^{t-1} \leq kC_{\text{MIN}}(\sigma_t) - C_A(\sigma_t)$. If we take the sum over t then the left hand side telescopes to give us $\Phi^{\text{final}} - \Phi^0 \leq kC_{\text{MIN}}(\sigma) - C_A(\sigma)$. We know that $0 \leq \Phi^{\text{final}}$ and so $C_A(\sigma) \leq kC_{\text{MIN}}(\sigma) + \Phi^0$. We assume that all of the on-line servers and all of the off-line servers are initially at the same single point. This implies that $\Phi^0 = 0$ and so $C_A(\sigma) \leq kC_{\text{MIN}}(\sigma)$. From this inequality we can see that the algorithm is k-competitive. It only remains for us to prove the claim.

Proof of claim We first consider what happens when the off-line algorithm moves a server to the request. Choose τ so that $M(s,a) = \sum_{i=1}^k d(a_i, s_{\tau(i)})$. Suppose that the off-line algorithm chooses to move a_l to the request. Since none of the on-line servers are moving $\sum_{i<j} d(s_i, s_j)$ does not change. Hence

$$\begin{aligned} \tilde{\Phi}^t - \Phi^{t-1} = k\Delta M(s,a) &\leq k(\text{distance traveled by } a_l \text{ to the request}) \\ &= kC_{\text{MIN}}(\sigma_t). \end{aligned}$$

We now consider how Φ changes when the on-line algorithm moves its servers. Number the on-line servers so that $s_1, s_2, \ldots, s_q$ are active and $s_{q+1}, \ldots, s_k$ are inactive.

The active servers are all moving at the same speed towards the request. Suppose that they all move a small distance ϵ. It is easy to see that without loss of generality a_l is matched to an active server in the minimum cost matching. (Recall that a_l is the off-line server which is already at the request.) Hence the distance between these two servers decreases by ϵ. Also, the distance between other pairs of matching servers increases by at most ϵ. Therefore,

$$\Delta M \leq (q-1)\epsilon - \epsilon = (q-2)\epsilon.$$

We also have to consider the change in $\sum_{i<j} d(s_i, s_j)$. Let

$$\begin{aligned} S_I &= \sum_{q<i<j} d(s_i, s_j), \\ S_{IA} &= \sum_{i \leq q < j} d(s_i, s_j), \\ S_A &= \sum_{i<j\leq q} d(s_i, s_j). \end{aligned}$$

Then,

$$\begin{aligned} \Delta S_A &= -2\epsilon \left(\begin{array}{c} q \\ 2 \end{array} \right) = 2\epsilon \frac{q(q-1)}{2} = -\epsilon q(q-1), \\ \Delta S_{IA} &= (k-q)(\epsilon - (q-1)\epsilon) = (k-q)\epsilon(2-q), \\ \Delta S_I &= 0. \end{aligned}$$

The first equation comes from the fact that each pair of active servers move towards each other by a distance -2ϵ. The second is true because there are $k-q$ inactive servers, each of which has one active server moving away from it and $q-1$ moving towards it. From the above equations we derive that $\sum_{i<j} d(s_i, s_j)$ increases by $\epsilon\,[(k-q)(2-q) - q(q-1)]$. Hence

$$\begin{aligned} \Delta\Phi &\leq \epsilon\,[(k-q)(2-q) - q(q-1) + k(q-2)] \\ &= -q\epsilon. \end{aligned}$$

Now $q\epsilon$ is the cost incurred by the on-line algorithm when it moves its q active servers a distance ϵ. Hence $\Phi^t - \tilde{\Phi}^t \leq -C_A(\sigma_t)$ which is the inequality that we were trying to prove.

13.2 Paging as a case of k-server on a tree

Note that a special case of the k-server problem on a tree is paging. Create M tree-vertices corresponding to the pages of main (slow) memory, create one dummy tree-vertex, v, and connect v to all the other M vertices using edges of length $\frac{1}{2}$. Note that the cost of moving a server from one page to another is $\frac{1}{2} + \frac{1}{2}$, the cost of

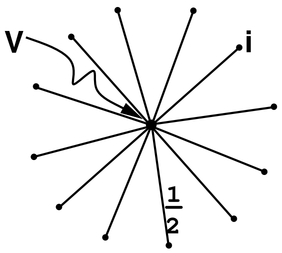


Figure 9: Paging as a special case of the k-server problem on a tree.

swapping. (More generally, we could let the length of the edge from v to page i be any positive function $f(i)$, obtaining the *Generalized Paging* problem, where the cost of swapping pages i and j is $f(i) + f(j)$.)

Let us consider the behavior of the above algorithm A on this special case. The resulting algorithm for paging is known as FLUSH-WHEN-FULL. The interpretation is simple if one keeps track of "marked" pages. When a server is at a vertex corresponding to a page p, this page is considered marked. As soon as the server leaves that vertex to go towards v, the page will be unmarked.

Initially there are k servers on k pages. These pages are thus marked. Suppose request σ_i causes the first page fault. Algorithm A will then move all servers towards v, resulting in the unmarking of all pages in fast memory. All these servers move at constant speed towards v and thus will reach the middle vertex v at the same time. One (arbitrarily selected) server at v will continue to the requested page σ_i and the other $k-1$ will become inactive. The page σ_i will then be marked. Later, if there is a page fault on say σ_j and there is at least one server at vertex v, one of these active servers will be moved to σ_j. In terms of paging, this is interpreted as swappping σ_j with an arbitrarily selected *unmarked* page of fast memory and then marking σ_j. The claim of the previous section implies that FLUSH-WHEN-FULL is k-competitive.

Of course we don't have to move more than one server per step. We could "pretend" to moves servers simultaneously but actually just keep track of where the servers should be (by keeping track of which pages of fast memory are marked) and move one server to the request-destination per step. The cost per step would be the the total distance travelled by that server since the last time it reached a request-destination. This way the cost per page-fault is always exactly one.

FLUSH-WHEN-FULL is much like MARKING, except that MARKING uses randomization to select a server at a tie. FLUSH-WHEN-FULL is k-competitive while MARKING is H_k-competitive against an oblivious adversary. This shows how useful a simple randomization step can be. FLUSH-WHEN-FULL applies to *Generalized Paging* and is the only known k-competitive algorithm for that problem.

Question: Could LRU be an example of FLUSH-WHEN-FULL? Yes, it is. In other words, LRU would never get rid of a marked page and thus, by carefully selecting which unmarked page to remove from fast memory, FLUSH-WHEN-FULL reduces to

LRU. To see that a marked page is never removed from fast memory by LRU, notice that each marked page has been requested after the last request to any unmarked page.

14 Electric Network Theory

We will use electric network theory for a randomized k-server algorithm due to Coppersmith, Doyle, Raghavan, and Snir [8]. Their algorithm will be k-competitive against an adaptive on-line adversary for a subclass of metrics.

An *electric network* is a graph $G = (V, E)$ such that each edge has weight $\sigma(e) = \frac{1}{R(e)}$, where $R(e) \in \mathbb{R}^+$ is called the *resistance* and $\sigma(e) \in \mathbb{R}^+$ is called the *conductance* of edge e. We can then ask what the *effective* resistance (also called the *equivalent* resistance) between any two vertices is, i.e. the resistance which is felt when applying a difference of voltage between any two vertices. The effective conductance is the inverse of the effective resistance.

For resistances in series, the effective resistance between the endpoints is equal to the sum of the resistances. For resistances in parallel, the effective conductance is equal to the sum of the conductances. See Figures 10 and 11. In general, though, these rules are not enough to determine the effective resistance between any two vertices in an electric network (consider the case when the underlying graph is the complete graph K_4 on 4 vertices). In full generality, one has to use Kirchoff's first law and the relation $V = RI$. Simply stated, the first law says that the sum of the currents entering a junction is equal to the sum of the currents leaving that junction.

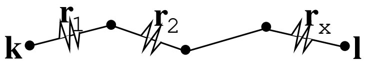


Figure 10: Resistance in series. The effective resistance between k and l is $r_1 + r_2 + \ldots + r_x$.

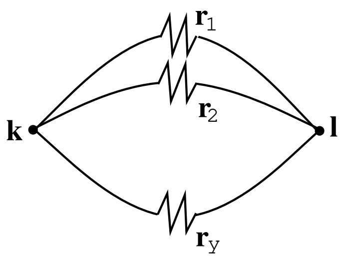


Figure 11: Resistances in parallel. The effective conductance between k and l is $\frac{1}{r_1} + \frac{1}{r_2} + \ldots + \frac{1}{r_y}$.

Definition 12 *A distance matrix $D = [d_{ij}]$ is* resistive *if it is the effective resistance matrix of some electric network G, $G = (V, E)$.*

Now let us use the effective resistance as a metric. This is justified by the following proposition.

Proposition 24 *If D is resistive then D is symmetric and D satisfies the triangle inequality $d_{ij} + d_{jk} \geq d_{ik}$ $\forall i, j, k$.*

The converse of this proposition is not necessarily true. A metric does not necessarily induce a resistive distance matrix. In fact, there are metric spaces on four points that aren't resistive. Satisfying the triangle inequality isn't enough.

Metrics which correspond to resistive distance matrices will be referred to as *resistive.*

What are resistive matrices or resistive metrics? Here are two simple examples.

1. if D is 3×3, symmetric, and satisfies the triangle inequality, then it is always resistive. Hence, the above proposition is true in the converse for 3×3 matrices. As an example, consider a triangular network with effective resistances 3, 4, and 5. See figure 12. We claim that the edge-resistances are $\frac{3}{11}$, $\frac{2}{11}$ and $\frac{1}{11}$ (see figure). We verify it for the vertices 1 and 3. Consider the two paths between these vertices. We need the effective resistance to be 4, and thus the effective conductance to be $\frac{1}{4}$. Verify that

$$\frac{2}{11} + \left(\frac{1}{\frac{11}{3} + \frac{11}{1}} \right) = \frac{11}{44} = \frac{1}{4}$$

2. a tree metric is resistive. To see this, make a tree of resistances with $R_{ij} = d(i, j)$ along the edges of the tree. Because there are no cycles, every pair of points is connected by a series of resistances. The effective resistance is the sum of the edge-resistances.

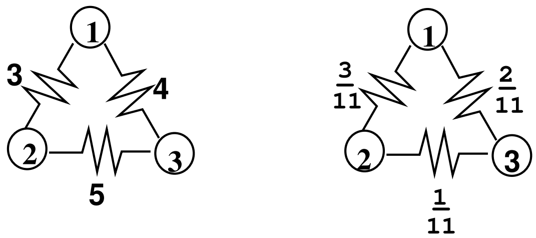


Figure 12: The given matrix of effective resistances (*left*) is relabelled (*right*) as the corresponding electrical network with resistances.

Another property of resistive metrics is given by the following lemma.

Lemma 25 *If D is resistive then any induced submatrix D' is resistive.*

How can we compute an array of conductances σ from D? We will not go through the proof here, but the outline of the algorithm is as follows:

- Assume D is $n \times n$. Construct an $(n-1) \times (n-1)$ matrix $\bar{D}$ such that $\bar{d}_{ij} = (d_{1i} + d_{1j} - d_{ij})/2 \geq 0$ for $2 \leq i, j \leq n$.
- Let $\bar{\sigma} = \bar{D}^{-1}$.
- Let $\sigma_{ij} = -\bar{\sigma}_{ij}$ for $i \neq j$, $2 \leq i, j \leq n$.
- Let $\sum_{j \neq i} \sigma_{ij} = \bar{\sigma}_{ii}$ for $2 \leq i \leq n$. This allows us to determine σ_{i1} and σ_{1i}.
- Let $\sigma_{ii} = 0$ for $1 \leq i \leq n$.

If D is resistive, the above procedure yield a matrix σ of non-negative conductances.

14.1 The Algorithm: RWALK

Consider the following algorithm for the k-server problem. We have servers at $a_1 \ldots a_k$, and we get a request at a_{k+1}. Consider the distance matrix D' among these $k+1$ elements. Assuming D' is resistive, calculate σ' and then move the server at a_i to the request with probability

$$Pr(a_i \rightarrow a_{k+1}) = \frac{\sigma'_{i,k+1}}{\sum_{1 \leq j \leq k} \sigma'_{j,k+1}}$$

Notice that a shorter distance corresponds to a higher conductance which in turn corresponds to a higher probability. Thus, this algorithm is intuitively correct because we are more likely to move a server "close" to the request.

Theorem 26 *If every induced subgraph on $k+1$ points is resistive, then* RWALK *is k-competitive against an adaptive on-line adversary.*

Two important cases covered by this theorem are:

- k-server with $k = 2$. RWALK can be used and is thus 2-competitive.
- k-server on a tree. We know that D' will always be resistive (by Lemma 25) and hence RWALK can be used. The theorem shows that RWALK is k-competitive in this case.

Proof:

We need to show

$$E[C_{\mathrm{RWALK}}(\mathrm{Q})] \leq kE[C_{\mathrm{Q}}(\mathrm{RWALK})]$$

where Q is an adaptive on-line adversary. We can rewrite this as

$$E[C_{\mathrm{RWALK}} - kC_{\mathrm{Q}}] \leq 0.$$

We will in fact show that

$$E[\Phi + C_{\mathrm{RWALK}} - kC_{\mathrm{Q}}] \leq 0$$

where Φ is a potential function that we will define.

We will show that at every step,

$$E[\Delta\Phi + C_{\mathrm{RWALK}}(step) - kC_{\mathrm{Q}}(step)] \leq 0.$$

In words, this means that, in any single step, the cost of RWALK is at most k times the cost of the adversary, once the costs are amortized according to the potential function Φ. Summing over all steps, we obtain that

$$E[\Phi_n - \Phi_0 + C_{\mathrm{RWALK}}(\mathrm{Q}) - kC_{\mathrm{Q}}(\mathrm{RWALK})] \leq 0,$$

for a sequence of n requests. Since $\Phi_n \geq 0$ and we shall assume that $\Phi_0 = 0$, we derive the competitive factor of k. ($\Phi_0 = 0$ corresponds to the case in which all servers start initially from the same location (see below), and $\Phi_0 \neq 0$ would just result in a weak competitive factor of k.)

We want Φ to measure the difference between the locations of RWALK's servers and Q's servers. Let $a = a_1, ..., a_k$ be the set of locations of RWALK's servers, and let $b = b_1, ..., b_k$ be the set of locations of Q's servers. Then define a potential function

$$\Phi(a,b) = \sum_{1 \leq i < j \leq k} d(a_i, a_j) + kM(a,b)$$

where $M(a,b)$ is the cost of the minimum cost matching between A and b (in other words, $M(a,b) = \min_\alpha \sum_{i=1}^{k} d(a_i, b_{\alpha(i)})$ where the minimum is taken over all permutations α). Intuitively, the sum is the amount of "separation" of the elements of A, and the matching term is the "difference" between the algorithm and the adversary. Some intuitive argument justifying the $M(a,b)$ term goes as follows. If the algorithm has to pay much more than the adversary, it means that (some of) the algorithm's servers are far away from the adversary's servers, implying that $M(a,b)$ is large. The reduction in $M(a,b)$ can therefore be used to pay for the move.

Let us consider the request σ_i. We decompose the processing of σ_i into two steps.

- step 1: adversary moves a server to σ_i.

- step 2: on-line algorithm RWALK moves a server to σ_i.

Step 1. The adversary moves from b_j to b'_j. Then $\Delta\Phi \leq kd(b_j, b'_j)$ because given the minimum matching for the old A and b, we can get a matching for the new A and b by using the same matching. So if b_j was matched to a_i, then

$$\Delta\Phi \leq k(d(a_i, b'_j) - d(a_i, b_j)) \leq kd(b_j, b'_j).$$

We also have

$$C_{\mathrm{RWALK}}(step) = 0,$$

and

$$C_{\mathrm{Q}}(step) = d(b_j, b'_j).$$

Therefore,

$$E[\Delta\Phi - kC_{\mathrm{Q}}(step)] \leq 0.$$

Case 2. The on-line algorithm moves. Since the adversary has already moved a server to the requested node, assume WLOG that the request is at b_1 (we can always renumber). Let the minimum matching M before the move be $(a_1, b_1) \ldots (a_k, b_k)$ (again we can renumber to make this the case).

Assume that RWALK moves a_j to the request b_1. We claim that

$$\Delta M(a, b) \leq d(a_1, b_j) - d(a_1, b_1) - d(a_j, b_j)$$

since a possible matching can be defined from the minimum cost matching between the old A's and b's by simply assigning a_1 to b_j and a_j to b_1:

$a_1 - b_j$
$a_2 - b_2$
$\vdots$
$a_j - b_1$
$\vdots$
$a_k - b_k$

From the triangle inequality, we have $d(a_1, b_j) - d(a_j, b_j) \leq d(a_1, a_j)$ and thus $\Delta M(a, b) \leq d(a_1, a_j) - d(a_1, b_1)$. Therefore,

$$\Delta\Phi \leq \sum_{i \neq j}[d(a_i, b_1) - d(a_i, a_j)] + k[d(a_1, a_j) - d(a_1, b_1)].$$

We also have $C_{\mathrm{RWALK}}(step) = d(a_j, b_1)$ and hence

$$\begin{aligned}
\Delta\Phi + C_{\mathrm{RWALK}}(step) &\leq \sum_{i \neq j}[d(a_i, b_1) - d(a_i, a_j)] + k[d(a_1, a_j) - d(a_1, b_1)] + d(a_j, b_1) \\
&\leq \sum_{i}[d(a_i, b_1) - d(a_i, a_j) + d(a_1, a_j) - d(a_1, b_1)].
\end{aligned}$$

We will show that $E[\Delta\Phi + C_{\text{RWALK}}(step)] \leq 0$ by showing that

$$\sum_{j=1}^{k} \sigma'_{j,k+1} \left(\sum_{i=1}^{k} e(a_1, a_i, a_j, b_1) \right) = 0$$

where

$$e(a_1, a_i, a_j, b_1) = d(a_i, b_1) - d(a_i, a_j) + d(a_1, a_j) - d(a_1, b_1).$$

For convenience, define $a_{k+1} = b_1$. Now we have to recall a few facts about electric networks to get a handle on the $\sigma'_{j,k+1}$. Recall that

$$\sigma'_{j,k+1} = -\bar{\sigma}_{j,k+1} \quad \text{for } j = 2, ..., k$$

and

$$\sum_{j=1}^{k} \sigma'_{j,k+1} = \bar{\sigma}_{k+1,k+1}.$$

We know that $\bar{D} \cdot \bar{\sigma} = I$, so

$$\sum_{j=2}^{k+1} \bar{d}(a_i, a_j)\ \bar{\sigma}_{j,k+1} = 0 \quad \text{if } i = 2, ..., k.$$

Summing over i, we get

$$\sum_{i=2}^{k} \sum_{j=2}^{k+1} \bar{d}(a_i, a_j)\ \bar{\sigma}_{j,k+1} = 0$$

or, using the definition of $\bar{\sigma}$,

$$\sum_{i=2}^{k} \left(-\sum_{j=2}^{k} \bar{d}(a_i, a_j)\sigma'_{j,k+1} + \sum_{j=1}^{k} \sigma'_{j,k+1}\bar{d}(a_i, a_{k+1}) \right) = 0.$$

Since $\bar{d}(a_i, a_1) = (d(a_i, a_1) - d(a_1, a_1) - d(a_i, a_1))/2 = 0$, we can extend the first summation over j from 1 to k:

$$\sum_{i=2}^{k} \sum_{j=1}^{k} \left(\bar{d}(a_i, b_1) - \bar{d}(a_i, a_j) \right) \sigma'_{j,k+1} = 0.$$

The term in parenthesis is

$$\begin{aligned} \bar{d}(a_i, b_1) - \bar{d}(a_i, a_j) &= \frac{1}{2}[d(a_1, a_i) + d(a_1, b_1) - d(a_i, b_1) - d(a_1, a_i) - d(a_1, a_j) + d(a_i, a_j)] \\ &= -\frac{1}{2} e(a_1, a_i, a_j, b_1) \end{aligned}$$

so we finally get

$$\sum_{j=1}^{k} \sigma'_{j,k+1} \left(\sum_{i=1}^{k} e(a_1, a_i, a_j, b_1) \right) = 0$$

since we can do the sum of i from 1 because $e(a_1, a_1, a_j, b_1) = 0$.

Therefore, we have that the expected amortized cost is negative and thus we have a k-competitive algorithm for any resistive metric.

□

15 The work function algorithm

In this section, we show that the work function algorithm for the k-server problem is $2k-1$-competitive. The algorithm was proposed by Chrobak and Larmore [7] but they were only able to show that it is 2-competitive when $k = 2$. The following proof is due to Koutsoupias and Papadimitriou [15]. Before this result was obtained, no algorithm was known to have a competitive ratio which was polynomial in k for an arbitrary metric space. It is believed that the work function algorithm is in fact k-competitive.

15.1 Definition of the work function

Let $A_0 = \{a_{0,1}, a_{0,2}, \ldots, a_{0,k}\}$ be the initial configuration of the servers and let $r_1, r_2, \ldots, r_t, \ldots$ be the sequence of requests. If X is a multiset of k points in the metric space we define the *work function* at time t by,

$$\begin{aligned} w_t(X) = &\text{the optimal cost of servicing the first } t \text{ requests} \\ &\text{and ending up with the servers in configuration } X. \end{aligned}$$

From this definition we can see that the initial work function w_0 can be expressed as,

$$\begin{aligned} w_0(X) &= \text{minimum cost matching between } A_0 \text{ and } X \\ &= \min_{\text{perm } \sigma} \sum_{i=1}^{k} d(a_{0,i}, x_{\sigma(i)}) \\ &= d(A_0, X). \end{aligned}$$

Before servicing r_t the servers will be in some configuration Y. Hence,

$$w_t(X) = \min_{Y : r_t \in Y} [w_{t-1}(Y) + d(Y, X)].$$

We shall need a number of properties of the work function.

Lemma 27

$$\forall t, X, Z : w_t(Z) \leq w_t(X) + d(X, Z).$$

Proof:

$$\begin{aligned} w_t(X) + d(X, Z) &= \min_{Y : r_t \in Y} \{w_{t-1}(Y) + d(Y, X) + d(X, Z)\} \\ &\geq \min_{Y : r_t \in Y} \{w_{t-1}(Y) + d(Y, Z)\} \\ &= w_t(Z). \end{aligned}$$

□

Lemma 28

$$\forall t, X : w_t(X) = \min_{x \in X} \{w_{t-1}(X - x + r_t) + d(r_t, x)\}.$$

Proof:

- ($\leq$) This is obvious since if we take $Y = X - x + r_t$ then $d(Y, X) = d(r_t, x)$.
- ($\geq$) Take the optimum Y in the definition of $w_t(X)$. Let x be the element of X which is matched to r_t in the minimum cost matching between X and Y. Let $Y' = X - x + r_t$. Then,

$$\begin{aligned} w_{t-1}(Y) + d(Y, X) &= w_{t-1}(Y) + d(Y, Y') + d(x, r_t) \\ &\geq w_{t-1}(Y') + d(x, r_t) && \text{by lemma 27} \\ &\geq \min_{x \in X} \left(w_{t-1}(X - x + r_t) + d(x, r_t)\right). \end{aligned}$$

□

Lemma 29

$$\forall t, X : w_t(X) \geq w_{t-1}(X).$$

Proof:

$$\begin{aligned} w_t(X) &= w_{t-1}(Y) + d(Y, X) && \text{for some } Y \\ &\geq w_{t-1}(X) && \text{by lemma 27.} \end{aligned}$$

□

Lemma 30

$$\forall X : r_t \in X \Rightarrow w_t(X) = w_{t-1}(X).$$

Proof:

From lemma 28 with $x = r_t$ we get $w_t(X) \leq w_{t-1}(X)$. Together with lemma 29 this implies the result. □

Lemma 31

$$\forall t, X : w_t(X) = \min_{x \in X} \{w_t(X - x + r_t) + d(r_t, x)\}.$$

Proof:

This is immediate from lemmas 28 and 30. □

15.2 Definition of the work function algorithm

Let A_{t-1} be the configuration of the on-line servers before r_t is requested. When the request comes in the work function algorithm moves $a \in A_{t-1}$ which minimizes

$$w_{t-1}(A_{t-1} - a + r_t) + d(a, r_t).$$

i.e. $A_t \leftarrow A_{t-1} - a + r_t$ where

$$a := \operatorname{argmin}_{a \in A_{t-1}} \left\{ w_{t-1}(A_{t-1} - a + r_t) + d(a, r_t) \right\}.$$

Note that $w_{t-1}(A_{t-1} - a + r_t) + d(a, r_t) = w_t(A_{t-1})$. Hence if we were to move the servers to A_{t-1} after serving request r_t then we would move the server at r_t to a. The work function algorithm does this move backwards.

For analysis purposes we assume that the on-line servers and the off-line servers finish at the same position. (If this is not the case we can repeatedly request points where there is an off-line server but no on-line one. This does not increase the off-line cost.) It is clear from the above definition that the on-line cost of serving r_t is $d(a, r_t)$. If r_f is the final request then the total off-line cost of serving all the requests is,

$$w_f(A_f) = \sum_{t=1}^{f} \left[w_t(A_t) - w_{t-1}(A_{t-1}) \right].$$

We define the off-line pseudocost of serving request r_t to be $w_t(A_t) - w_{t-1}(A_{t-1})$. We then define the extended cost of serving request r_t to be the sum of the off-line pseudocost and the on-line cost. i.e.

$$\text{extended cost } = w_t(A_t) - w_{t-1}(A_{t-1}) + d(a, r_t).$$

But, by definition of the work function algorithm, $w_t(A_{t-1}) = w_{t-1}(A_t) + d(a, r_t)$. Hence,

$$\begin{aligned} \text{extended cost } &= w_t(A_{t-1}) - w_{t-1}(A_{t-1}) \\ &\leq \max_X \left\{ w_t(X) - w_{t-1}(X) \right\}. \end{aligned}$$

Note that the above equation gives an upper bound for the extended cost which is *independent of the algorithm* or the location of the servers. The definition of extended cost means that if

$$\text{extended cost } \leq (c+1) \text{ off-line cost}$$

then the work-function algorithm is c-competitive. We shall show that

$$\text{extended cost } \leq 2k \,(\text{ off-line cost}),$$

proving the desired competitive ratio.

Our next goal is to define a potential function $\Phi(w)$. We shall then show that, at every request, the increase of the potential function is lower bounded by the extended cost and that the total increase of the potential function is upper bounded by $2k$ times the off-line cost. More formally, we will show that

$$\begin{array}{lcl} \Phi(w_t) - \Phi(w_{t-1}) & \geq & \max_X \{w_t(X) - w_{t-1}(X)\}, \\ \Phi(w_f) & \leq & 2kw_f(A_f) + c. \end{array}$$

For this purpose, we shall need a better understanding of $\max_X \{w_t(X) - w_{t-1}(X)\}$. The following lemma is useful. Koutsoupias and Papadimitriou show a generalization of this lemma which they call "quasiconvexity".

Lemma 32

$$\forall t, \forall A, B, \forall a \in A, \exists b \in B : w_t(A) + w_t(B) \geq w_t(A - a + b) + w_t(B - b + a).$$

Proof:

By induction on t. When $t = 0$, $w_t(X) = w_0(X) = d(A_0, X)$ for all configurations X. Let a_0 be the element of A_0 which is matched with a in the minimum cost matching between A and A_0. Let b be the element of B which is matched with a_0 in the minimum cost matching between A_0 and B. Then,

$$\begin{array}{rcl} w_0(A) + w_0(B) & = & d(A_0, A) + d(A_0, B) \\ & = & d(A_0 - a_0, A - a) + d(A_0 - a_0, B - b) + d(a_0, a) + d(a_0, b) \\ & \geq & d(A_0, A - a + b) + d(A_0, B - b + a) \\ & = & w_0(A - a + b) + w_0(B - b + a). \end{array}$$

Now assume that the lemma is true for $t - 1$. Then by lemma 28,

$$(4) \qquad \left\{ \begin{array}{lcl} \exists x \in A : w_t(A) & = & w_{t-1}(A - x + r_t) + d(x, r_t), \\ \exists y \in B : w_t(B) & = & w_{t-1}(B - y + r_t) + d(y, r_t). \end{array} \right.$$

If $a = x$ then let $b = y$. By definition of w_t,

$$\begin{array}{rcl} w_t(A - x + y) & \leq & w_{t-1}(A - x + r_t) + d(y, r_t), \\ w_t(B - y + x) & \leq & w_{t-1}(B - y + r_t) + d(x, r_t). \end{array}$$

Hence,

$$w_t(A - x + y) + w_t(B - y + x) \leq w_t(A) + w_t(B).$$

If $a \neq x$ then $a \in A - x + r_t$. Using the inductive hypothesis on $A - x + r_t$ and $B - y + r_t$, $\exists b \in B - y + r_t$:

(5)

$$w_{t-1}(A - x + r_t) + w_{t-1}(B - y + r_t) \geq w_{t-1}(A - x + r_t - a + b) + w_{t-1}(B - y + r_t - b + a)$$

But,

$$(6) \qquad \left\{ \begin{array}{lcl} w_t(A - a + b) & \leq & w_{t-1}(A - a + b - x + r_t) + d(x, r_t), \\ w_t(B - b + a) & \leq & w_{t-1}(B - b + a - y + r_t) + d(y, r_t). \end{array} \right.$$

Finally, (4), (5) and (6) imply the lemma. □

Definition 13 *A is called a minimizer of the point a with respect to w_t if A minimizes $(w_t(A) - \sum_{x \in A} d(a, x))$.*

We can redefine $\max_X \{w_t(X) - w_{t-1}(X)\}$ as

$$\max_X \left\{ (w_t(X) - \sum_{x \in X} d(x, r_t)) - (w_{t-1}(X) - \sum_{x \in X} d(x, r_t)) \right\}.$$

A minimizer of r_t with respect to w_{t-1} would thus maximize the second term. The next lemma shows somewhat surprisingly that such a minimizer would in fact maximize the entire expression.

Lemma 33 *Assume that A is a minimizer of r_t with respect to w_{t-1}. Then $w_t(A) - w_{t-1}(A) = \max_X \{w_t(X) - w_{t-1}(X)\}$.*

Proof:

We need to show that,

$$\forall B : w_t(A) + w_{t-1}(B) \geq w_{t-1}(A) + w_t(B).$$

This is equivalent to,

$$\forall a \in A : w_{t-1}(A - a + r_t) + d(a, r_t) + w_{t-1}(B) \geq \min_{b \in B} \{w_{t-1}(B - b + r_t) + d(b, r_t) + w_{t-1}(A)\}.$$

By assumption,

$$w_{t-1}(A) - \sum_{x \in A} d(x, r_t) \leq w_{t-1}(A - a + b) - \sum_{y \in A - a + b} d(y, r_t),$$

or,

$$w_{t-1}(A) + d(b, r_t) \leq w_{t-1}(A - a + b) + d(a, r_t).$$

Hence we only need to show that,

$$w_{t-1}(A - a + r_t) + w_{t-1}(B) \geq \min_{b \in B} \{w_{t-1}(B - b + r_t) + w_{t-1}(A - a + b)\}.$$

But by lemma 32, $r_t \in A - a + r_t \Rightarrow \exists b \in B :$

$$w_{t-1}(A - a + r_t) + w_{t-1}(B) \geq w_{t-1}(A - a + b) + w_{t-1}(B - b + r_t).$$

□

Lemma 34 *Assume that A is a minimizer with respect to w_{t-1}. Then A is also a minimizer with respect to w_t.*

Proof:

We need to show that,

$$\forall B : w_t(A) - \sum_{a \in A} d(a, r_t) \geq w_t(B) - \sum_{b \in B} d(b, r_t),$$

or, for all $y \in B$,

$$\min_{x \in A} \left\{ w_{t-1}(A - x + r_t) + d(x, r_t) - \sum_{a \in A} d(a, r_t) \right\} \leq w_{t-1}(B - y + r_t) + d(y, r_t) - \sum_{b \in B} d(b, r_t).$$

Since A is a minimizer of r_t with respect to w_{t-1},

$$w_{t-1}(A) - \sum_{a \in A} d(a, r_t) \leq w_{t-1}(B - x + y) - \sum_{b \in B + x - y} d(b, r_t),$$

or,

$$w_{t-1}(A) + d(x, r_t) - \sum_{a \in A} d(a, r_t) \leq w_{t-1}(B + x - y) - \sum_{b \in B} d(b, r_t) + d(y, r_t).$$

Now $r_t \in B - y + r_t$, hence by lemma 32,

$$\exists x \in A : w_{t-1}(A - x + r_t) + w_{t-1}(B - y + x) \leq w_{t-1}(A) + w_{t-1}(B - y + r_t),$$

which implies that

$$w_{t-1}(A - x + r_t) + d(x, r_t) - \sum_{a \in A} d(a, r_t) \leq w_{t-1}(B - y + r_t) + d(y, r_t) - \sum_{b \in B} d(b, r_t).$$

$\square$

15.3 Definition of the potential function

We define the potential function as follows.

$$\Phi(w_t) = \min_U \left\{ k w_t(U) + \sum_{i=1}^{k} \min_{B_i} \left(w_t(B_i) - \sum_{j=1}^{k} d(u_j, b_{ij}) \right) \right\}.$$

Thus B_i is a minimizer of u_i with respect to w_t.

Lemma 35 *We can assume that $r_t \in U$ when trying to minimize the above expression.*

Proof:

By lemma 31,

$$w_t(U) = w_t(U - u_l + r_t) + d(r_t, u_l) \text{ for some } u_l.$$

By the triangle equality,

$$d(r_t, u_l) - d(u_l, b_{ij}) \geq -d(r_t, b_{ij}). \tag{7}$$

Let $U' = U - u_l + r_t$.

$$\Phi(w_t) = \min_{U'} \begin{cases} kw_t(U') & + \ \sum_{i=1, i\neq l}^{k} \min_{B_i} \left(w_t(B_i) - \sum_{j=1}^{k} d(u_i, b_{ij})\right) \\ & + \ \min_{B_l} \left(w_t(B_l) + \sum_{j=1}^{k} \left(-d(u_l, b_{lj}) + d(r_t, u_l)\right)\right). \end{cases} \tag{8}$$

But by (7),

$$-\sum_{j=1}^{k} \left(d(u_l, b_{lj}) + d(r_t, u_l)\right) \geq -\sum_{j=1}^{k} d(r_t, b_{lj}).$$

Substituting this inequality into (8) gives us the required result. $\square$

Let $U, B_1, B_2, \ldots, B_k$ attain the minimum in the definition of $\Phi(w_t)$. The above lemma means that we can assume $r_t = u_l$. Hence we can assume that B_l is a minimizer of r_t with respect to w_t. Therefore by lemma 34 we can take B_l to be a minimizer of r_t with respect to w_{t-1}. Let $A = B_l$. By defeinition of Φ, we derive that

$$\Phi(w_{t-1}) \leq kw_{t-1}(A) + \sum_{i=1}^{k} \left(w_{t-1}(B_i) - \sum_{j=1}^{k} d(u_i, b_{ij})\right).$$

Therefore, by Lemmas 29 and 32, we obtain

$$\Phi(w_t) - \Phi(w_{t-1}) \geq w_t(A) - w_{t-1}(A) = \max_X \left\{w_t(X) - w_{t-1}(X)\right\}.$$

Summing over all requests, we obtain that the total extended cost is at most $\Phi(w_f) - \Phi(w_0)$.

We only need to upper bound this quantity in terms of the off-line cost. Let $U = A_f, B_i = A_f$ (where A_f is the final location of the off-line or on-line servers). This gives us,

$$\begin{aligned} \Phi(w_f) &\leq kw_f(A_f) + kw_f(A_f) - \sum_{a\in A_f, b\in A_f} d(a, b) \\ &\leq 2kw_f(A_f). \end{aligned}$$

Hence if we let $c = \Phi(w_0)$ (constant dependent only on the initial configuration, not the request sequence) then,

$$\text{total extended cost } \leq 2kw_f(A_f) + c.$$

By the remarks following the definition of extended cost this means that the work function algorithm is $(2k-1)$-competitive.

References

[1] L. Belady. A study of replacement algorithms for virtual storage computers. *IBM Syst. J.*, 5:78–101, 1966.

[2] S. Ben-David, A. Borodin, R. Karp, G. Tardos, and A. Wigderson. On the power of randomization in on-line algorithms. *Algorithmica*, 11:2–14, 1990.

[3] P. Berman, H. Karloff, and G. Tardos. A competitive three-server algorithm. In *Proceedings of the 1st ACM-SIAM Symposium on Discrete Algorithms*, 1990.

[4] M. Chrobak and L. Larmore. On fast algorithms for two servers. *Journal of Algorithms*, 12:607–614, 1991.

[5] M. Chrobak and L. Larmore. An optimal on-line algorithm for k-servers on trees. *SIAM Journal on Computing*, 20(1):144–148, February 1991.

[6] M. Chrobak and L. Larmore. HARMONIC is 3-competitive for 2 servers. *Theoretical Computer Science*, 98:339–346, 1992.

[7] M. Chrobak and L. Larmore. The server problem and on-line games. In D. Sleator and L. McGeoch, editors, *On-line algorithms*, volume 7 of *DIMACS Series in Discrete Mathematics and Theoretical Computer Science*, pages 11–64. AMS, 1992.

[8] D. Coppersmith, P. Doyle, P. Raghavan, and M. Snir. Random walks on weighted graphs, and application to on-line algorithms. *Journal of the ACM*, 40:421–453, 1993. A preliminary version appeared in the Proceedings of the 22nd STOC, 369–378, 1990.

[9] A. Fiat, R. M. Karp, M. Luby, L. A. McGeoch, D. D. Sleator, and N. E. Young. Competitive paging algorithms. *Journal of Algorithms*, 12:685–699, 1991.

[10] A. Fiat, Y. Rabani, and Y. Ravid. Competitive k-server algorithms. In *Proceedings of the 31st Annual Symposium on Foundations of Computer Science*, 1990.

[11] E. Grove. The harmonic online k-server algorithm is competitive. In *Proceedings of the 23rd Annual ACM Symposium on Theory of Computing*, pages 260–266, May 1991.

[12] S. Irani and R. Rubinfeld. A competitive 2-server algorithm. *Information Processing Letters*, 39:85–91, 1991.

[13] D. Johnson. Fast algorithms for bin packing. *Journal of Computer and System Sciences*, 8:272–314, 1974.

[14] J. M. Kleinberg. On-line algorithms for robot navigation and server problems. Master's thesis, MIT, Cambridge, MA, 1994.

[15] E. Koutsoupias and C. Papadimitriou. On the k-server conjecture. In *Proceedings of the 26th Annual ACM Symposium on Theory of Computing*, pages 507–511, 1994.

[16] M. Manasse, L. McGeoch, and D. Sleator. Competitive algorithms for server problems. *Journal of Algorithms*, 11:208–230, 1990.

[17] L. McGeoch and D. Sleator. A strongly competitive randomized paging algorithm. *Algorithmica*, 6:816–825, 1991.

[18] P. Raghavan and M. Snir. Memory vs. randomization in on-line algorithms. In *Proceedings of the 1989 ICALP Conference*, 1989.

[19] D. Sleator and R. Tarjan. Amortized efficiency of list update and paging rules. *Communications of the ACM*, 28(2):202–208, 1985.

[20] A. Yao. New algorithms for bin packing. *Journal of the ACM*, 27:207–227, 1980.

18.415/6.854 Advanced Algorithms	September 1994

Randomized Algorithms

Lecturer: Michel X. Goemans

1 Introduction

We have already seen some uses of randomization in the design of on-line algorithms. In these notes, we shall describe other important illustrations of randomized algorithms in other areas of the theory of algorithms. For those interested in learning more about randomized algorithms, we strongly recommend the forthcoming book by Motwani and Raghavan. [9]. First we shall describe some basic principles which typically underly the construction of randomized algorithms. The description follows a set of lectures given by R.M. Karp [7].

1. **Foiling the adversary**. This does not need much explanation since this was the main use of randomization in the context of on-line algorithms. It applies to problems which can be viewed as a game between the algorithm designer and the adversary. The adversary has some payoff function (running time of the algorithm, cost of solution produced by algorithm, competitive ratio, ...) and the algorithm designer tries to minimize the adversary's payoff. In this framework randomization helps in confusing the adversary. The adversary cannot predict the algorithm's moves.

2. **Abundance of witnesses**. Often problems are of the type "does input have property p?" (for example. "Is n composite?"). Typically the property of interest can be established by providing an object, called a "witness". In some cases, finding witnesses deterministically may be difficult. In such cases randomization may help. For example if there exists a probability space where witnesses are abundant then a randomized algorithm is likely to find one by repeated sampling. If repeated sampling yields a witness, we have a mathematical proof that the input has the property. Otherwise we have strong evidence that the input doesn't have the property, but no proof. A randomized algorithm which may return an incorrect answer is called a *Monte-Carlo* algorithm. This is in contrast with a *Las Vegas* algorithm which is guaranteed to return the correct answer.

3. **Checking an identity**
For example, given a function of several variables $f(x_1, \dots, x_n)$, is $f(x_1, \dots, x_n) \equiv 0$? One way to test this is to generate a random vector $a_1, \dots, a_n$ and evaluate $f(a_1, \dots, a_n)$. If its value is not 0, then clearly $f \not\equiv 0$.

Suppose we can generate random vectors $a_1, \ldots, a_n$ under some probability distribution so that

$$P = Pr\left[f(a_1, \ldots, a_n) = 0 | f \not\equiv 0\right] < \frac{1}{2},$$

or any other constant bounded away from 1. Then we can determine whether or not $f \equiv 0$ with high probability. Notice that this is a special case of category 2, since in this probability space, vectors a for which $f(a_1, \ldots, a_n) \neq 0$ constitute "witnesses".

4. **Random ordering of input**
The performance of an algorithm may depend upon the ordering of input data; using randomization this dependence is removed. The classic example is Quicksort, which takes $O(n^2)$ time in the worst case but when randomized takes $O(n \lg n)$ expected time, and the running time depends only on the coin tosses, not on the input. This can be viewed as a special case of category 1. Notice that randomized quicksort is a Las Vegas algorithm; the output is always correctly sorted.

5. **Fingerprinting**
This is a technique for representing a large object by a small fingerprint. Under appropriate circumstances, if two objects have the same fingerprint, then there is strong evidence that they are identical.

An example is the randomized algorithm for pattern matching by Karp and Rabin [8]. Suppose we are given a string of length n such as

$$randomizearandomlyrandomrandomizedrandom$$

and a pattern of size m such as *random*. The task is to find all the places the pattern appears in the long string.

Let us first describe our model of computation. We assume a simplified version of the unit-cost RAM model in which the standard operations $+, -, *, /, <, =$ take one unit of time provided they are performed over a field whose size is polynomial in the input size. In our case, the input size is $O(n + m) = O(n)$ and thus operations on numbers with $O(\log n)$ bits take only one unit of time.

A naive approach is to try starting at each location and compare the pattern to the m characters starting at that location; this takes $O(nm)$ time (in our model of computation we cannot compare two strings of m characters in $O(1)$ time unless $m = O(\log n)$). The best deterministic algorithm takes $O(n + m)$ time, but it is complicated. There is, however, a fairly simple $O(n + m)$ time randomized algorithm.

Say that the pattern X is a string of bits $x_1, \ldots, x_m$ and similarly $Y = y_1, \ldots, y_n$. We want to compare X, viewed as a number to $Y_i = y_i, \ldots, y_{i+m-1}$. This would normally take $O(m)$ time, but it can be done much more quickly by computing fingerprints and comparing those. To compute fingerprints, choose a prime p. Then the fingerprint of X is $h(X) = X \bmod p$ and similarly $h(Y_i) = Y_i \bmod p$. Clearly $h(X) \neq h(Y_i) \Rightarrow X \neq Y_i$. The converse is not necessarily true. Say that we have a *false match* if $h(X) = h(Y_i)$ but $X \neq Y_i$. A false match occurs iff p divides $|X - Y_i|$.

We show that if p is selected uniformly among all primes less than some threshold Q then the probability of a small match is small. First how many primes p divide $|X - Y_i|$? Well, since every prime is at least 2 and $|X - Y_i| \leq 2^m$, we must have at most m primes dividing $|X - Y_i|$. As a result, if p is chosen uniformly at random among $\{q : q \text{ prime and } q \leq Q\}$ then $Pr[h(X) = h(Y_i)|X \neq Y_i] \leq \frac{m}{\pi(Q)}$, where $\pi(n)$ denotes the number of primes less or equal to n. Thus, the probability that there is a false match for some i is upper bounded by n times $\frac{m}{\pi(Q)}$. Since $\pi(n)$ is asymptotically equal to $n/\ln n$, we derive that this probability is $O(\frac{\ln n}{n})$ if $Q = n^2 m$. This result can be refined by using the following lemma and the fact that there is a false match for some i if p divides $\prod_i |X - Y_i| \leq 2^{nm}$.

Lemma 1 *The number of primes dividing $a \leq 2^n$ is at most $\pi(n) + O(1)$.*

The refined version is the following:

Theorem 2 *If p is chosen uniformly at random among $\{q : q \text{ prime and } q \leq n^2 m\}$, then the probability of a false match for some i is upper bounded by $\frac{2+O(1)}{n}$.*

The fingerprint has only $\lg(n^2 m)$ bits, much smaller than m. Operations on the fingerprints can be done in $O(1)$ time in our computational model.

The advantage of this approach is that it is easy to compute $h(Y_{i+1})$ from $h(Y_i)$ in $O(1)$ time:

$$\begin{aligned} Y_{i+1} &= 2Y_i + y_{i+m} - 2^m y_i \\ h(Y_{i+1}) &= 2h(Y_i) + y_{i+m} - 2^m y_i \pmod{p}. \end{aligned}$$

One then checks if the fingerprints are equal. If they are, the algorithm claims that a match has been found and continues. To reduce the probability of failure, one can repeat the algorithm with another prime (or several other primes) and ouput only those who were matches for all primes tried. This is thus a Monte Carlo algorithm whose running time is $O(n + m)$.

This algorithm can easily be transformed into a Las Vegas algorithm. Whenever there is a potential match (i.e. the fingerprints are equal), we compare X and Y_i directly at a cost of $O(m)$. The *expected* running time is now $O((n + m) + km + nm\frac{2}{n}) = O(km + n)$, where k denotes the number of real matches.

6. **Symmetry breaking**
   This is useful in distributed algorithms, but we won't have much to say about it in this class. In that context, it is often necessary for several processors to collectively decide on an action among several (seemingly indistinguishable) actions, and randomization helps in this case.

7. **Rapidly mixing Markov chains**
   These are useful for counting problems, such as counting the number of cycles in a graph, or the number of trees, or matchings, or whatever. First, the counting problem is transformed into a sampling problem. Markov chains can be used to generate points of a given space at random, but we need them to converge rapidly — such Markov chains are called rapidly mixing. This area is covered in details in these notes.

2 Randomized Algorithm for Bipartite Matching

We now look at a randomized algorithm by Mulmuley, Vazirani and Vazirani [10] for bipartite matching. This algorithm uses randomness to check an identity.

Call an undirected graph $G = (V, E)$ bipartite if (1) $V = A \cup B$ and $A \cap B = \emptyset$, and (2) for all $(u, v) \in E$, either $u \in A$ and $v \in B$, or $u \in B$ and $v \in A$. An example of a bipartite graph is shown in Figure 1.

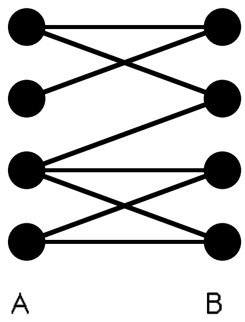


Figure 1: Sample bipartite graph.

A *matching* on G is a collection of vertex-disjoint edges. A *perfect matching* is a matching that covers every vertex. Notice that we must have $|A| = |B|$.

We can now pose two problems:

1. Does G have a perfect matching?

2. Find a perfect matching or argue that none exists.

Both of these problems can be solved in polynomial time. In this lecture we show how to solve the first problem in randomized polynomial time, and next lecture we'll cover the second problem. These algorithms are simpler than the deterministic ones, and lead to parallel algorithms which show the problems are in the class RNC. RNC is Randomized NC, and NC is the complexity class of problems that can be solved in polylogarithmic time on a number of processes polynomial in the size of the input. No NC algorithm for either of these problems is known.

The Mulmuley, Vazirani and Vazirani randomized algorithm works as follows. Consider the adjacency matrix A on graph $G = (V, E)$ whose entries a_{ij} are defined as follows:

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

where the indices i and j correspond to vertices of the vertex sets A and B respectively.

There exists a perfect matching in the graph G if and only if the adjacency matrix contains a set of n 1's, no two of which are in the same column or row. In other words, if all other entries were 0 a permutation matrix would result.

Consider the function called the *permanent* of A, defined as follows:

$$perm(A) = \sum_{permutations\ \sigma} \left(\prod_{i=1}^{n} a_{i\sigma_i} \right). \tag{2}$$

This gives the number of perfect matchings of the graph A represents. Unfortunately the best known algorithm for computing the permanent has running time O($n2^n$). However, note the similarity of the formula for computing the permanent to that for the determinant of A:

$$det(A) = \sum_{permutations\ \sigma} sign(\sigma) \left(\prod_{i=1}^{n} a_{i\sigma_i} \right).$$

The determinant can be computed in $O(n^3)$ time by using Gaussian elimination (and in $O(\log^2(n))$ time on $O(n^{3.5})$ processors). Note also that:

$$det(A) \neq 0 \Rightarrow perm(A) \neq 0 \Leftrightarrow \exists \text{ perfect matching.}$$

Unfortunately the converse is not true.

To handle the converse, we replace each entry a_{ij} of matrix A with $(a_{ij}x_{ij})$, where x_{ij} is a variable. Now both $det(A)$ and $perm(A)$ are polynomials in x_{ij}. It follows

$$det(A) \equiv 0 \Leftrightarrow perm(A) \equiv 0 \Leftrightarrow \nexists \text{ perfect matching.}$$

A polynomial in 1 variable of degree n will be identically equal to 0 if and only if it is equal to 0 at $n+1$ points. So, if there was only one variable, we could compute this determinant for $n+1$ values and check whether it is identically zero. Unfortunately, we are dealing here with polynomials in several variables.

So to test whether $det(A) \equiv 0$, we will generate values for the x_{ij} and check if the resulting matrix has $det(A) = 0$ using Gaussian elimination. If it is not 0, we know the determinant is not equivalent to 0, so G has a perfect matching.

Theorem 3 *Let the values of the* x_{ij} *be independently and uniformly distributed in* $[1, 2, \ldots, 2n]$*, where* $n = |A| = |B|$*. Let* A' *be the resulting matrix. Then*

$$Pr[det(A') = 0 | det(A) \not\equiv 0] \leq \frac{1}{2}.$$

It follows from the theorem that if G has a perfect matching, we'll find a witness in k trials with probability at least $1 - 1/2^k$.

In fact, this theorem is just a statement about polynomials. We can restate it as follows:

Theorem 4 *Let* $f(x_1, \ldots, x_n)$ *be a multivariate polynomial of degree* d*. Let* x_i *be independently and uniformly distributed in* $\{1, 2, \ldots, 2d\}$*. Then*

$$Pr[f(x_1, \ldots, x_n) \neq 0 | f \not\equiv 0] \geq \frac{1}{2}.$$

This theorem can be used for other problems as well.

Instead of proving this theorem we'll prove a stronger version which can be used for the second problem, that of finding the perfect matching.

Consider assigning costs c_{ij} ($c_{ij} \in \mathbb{N}$) to the edges $(i, j) \in E$. Define the cost of a matching M as:

$$c(M) = \sum_{(i,j) \in M} c_{ij}.$$

Now, consider the matrix A with entries $a_{ij} w_{ij}$, where $w_{ij} = 2^{c_{ij}}$. Then:

$$perm(A) = \sum_M 2^{c(M)}$$

and

$$det(A) = \sum_M sign(M) 2^{c(M)}.$$

If a unique minimum cost matching with cost c^* exists then $\det(A)$ will be nonzero and, in fact, it will be an odd multiple of 2^{c^*}.

We will prove that if we select the costs according to a suitable probability distribution then, with high probability, there exists a unique minimum cost matching. Let c_{ij} be independent, identically distributed random variables with distribution uniform in the interval $[1, \ldots, 2m]$, where $m = |E|$. The algorithm computes $\det(A)$ and claims that there is a perfect matching if and only if $\det(A)$ is nonzero. The only situation in which this algorithm can err is when there is a perfect matching, but $\det(A) = 0$. This is thus a Monte-Carlo algorithm. The next theorem upper-bounds the probability of making a mistake.

Theorem 5 *Assume that there exists a perfect matching in* G*. Then the probability that we will err with our algorithm is at most* $\frac{1}{2}$*.*

If a higher reliability is desired then it can be attained by running multiple passes, and only concluding that there is no perfect matching if no pass can find one.

Proof:

We need to compute $Pr\left[det(A)=0\right]$. Though this quantity is difficult to compute, we can fairly easily find an upper bound for it. As we have seen previously,

$$\begin{aligned} Pr\left[det(A)=0\right] &= 1 - Pr\left[det(A) \neq 0\right] \\ &\leq 1 - P \end{aligned}$$

where

$$P = Pr\left[\exists \text{ unique minimum cost matching}\right].$$

Indeed, if there is a unique minimum cost matching of cost say c^* then $det(A)$ is an odd multiple of 2^{c^*} and, hence, non-zero. The following claim completes the proof.

Claim 6 $P \geq \frac{1}{2}$

Given a vector c, define d_{ij} to be the maximum value for c_{ij} such that (i,j) is part of some minimum cost matching.

We can then draw the following inferences:

$$\begin{cases} c_{ij} > d_{ij} \Rightarrow (i,j) \; \textit{is not part of ANY minimum cost matching} \\ c_{ij} = d_{ij} \Rightarrow (i,j) \; \textit{is part of SOME minimum cost matching} \\ c_{ij} < d_{ij} \Rightarrow (i,j) \; \textit{is part of EVERY minimum cost matching} \end{cases}$$

Thus, if $c_{ij} \neq d_{ij}$ for all $(i,j) \Rightarrow \exists$ *a unique minimum cost matching* M. Moreover, this matching is given by $M = \{(i,j) \mid c_{ij} < d_{ij}\}$.

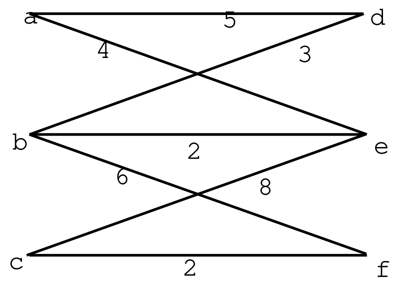


Figure 2: Example graph for d_{ij} computations.

Figure 3 shows an example of a bipartite graph with the values of c_{ij} assigned. Notice that in this graph $c^* = 9$. Consider the edge (c,f). The cheapest perfect matching not containing (c,f) has a cost of $6+8+5=19$. The other edges in the perfect matching containing (c,f) have a total cost of 7, so $d_{cf} = 12$. Thus, it is in every perfect matching. $d_{ad} = 5 = 4+3+2-2-2 = c_{ad}$. Thus it is in some minimum cost matching. Finally, $d_{ce} = -2 = 9-6-5 < c_{ce}$, so (c, e) is not in any minimum cost matching.

Therefore,

$$\begin{aligned}
Pr[unique\ minimum\ cost\ matching] &\geq Pr\left[c_{ij} \neq d_{ij} \text{ for all } (i,j)\right] \\
&= 1 - Pr\left[c_{ij} = d_{ij}\ for\ some\ (i,j)\right] \\
&\geq 1 - \sum_{(i,j)\in E} Pr\left[c_{ij} = d_{ij}\right] \\
&\geq 1 - m \cdot \frac{1}{2m} \\
&= \frac{1}{2}.
\end{aligned}$$

The equation in the next to last line is justified by our selection of $m = |E|$ and the fact that d_{ij} is independent of c_{ij}, so that the probability of c_{ij} being equal to the particular value d_{ij} is either $\frac{1}{2m}$ iff d_{ij} is in the range $[1, \ldots, 2m]$ or 0 otherwise. $\diamond$

$\square$

Notice that if we repeat the algorithm with new random c_{ij}'s, then the second trial will be independent of the first run. Thus, we can arbitrarily reduce the error probability of our algorithm, since the probability of error after t iterations is at most $\left(\frac{1}{2}\right)^t$.

Also, note that, in the proof, we do not make use of the assumption that we are working with matchings. Thus, this proof technique is applicable to a wide class of problems.

2.1 Constructing a Perfect Matching

In order to construct a perfect matching, assume that there exists a unique minimum cost matching (which we have shown to be true with probability at least $\frac{1}{2}$) with cost c^*. The determinant of A will then be an odd multiple of 2^{c^*}. By expanding the determinant along row i, it can be computed by the following formula:

$$\sum_j \pm 2^{c_{ij}} a_{ij} det(A_{ij}) \tag{3}$$

where A_{ij} is the matrix created by removing column i and row j from the matrix A (See figure 3), and the sign depends on the parity of $i+j$. The term in the summation above will be an odd multiple of 2^{c^*} if $(i,j) \in M$ and an even multiple otherwise. So, we can reconstruct a perfect matching M by letting:

$$M = \left\{(i,j) \mid 2^{c_{ij}} det(A_{ij})\ is\ an\ odd\ multiple\ of\ 2^{c^*}\right\}. \tag{4}$$

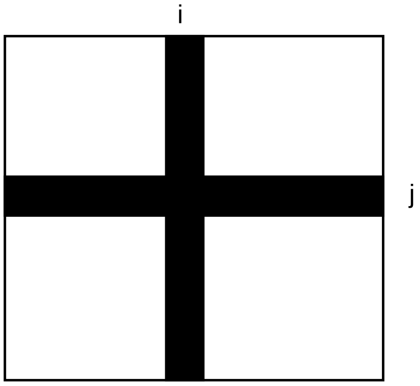


Figure 3: The Matrix A_{ij}. The matrix A_{ij} is formed by removing the ith column and the jth row from the matrix A.

Note that c^* can be obtained since 2^{c^*} is the largest power of 2 in $det(A)$.

The algorithm we have presented can be seen to be in RNC since a determinant can be computed in RNC.

We can apply a similar algorithm for solving the following related matching problem:

> Given:
>
> - A bipartite graph G,
> - A coloring for each edge in G of either *red* or *blue*,
> - an integer k,
>
> find a perfect matching with exactly k red edges.

However, in contrast to the problem of finding any perfect matching, it is not known whether this problem is in P or even NP-complete. For this problem, define the entries a_{ij} of the matrix A as follows:

$$a_{ij} = \begin{cases} 0 & \text{if } (i,j) \notin E \\ w_{ij} & \text{if } (i,j)\ is\ blue \\ w_{ij}x & \text{if } (i,j)\ is\ red \end{cases} \tag{5}$$

where x is a variable. Both the permanent of A and the determinant of A are now polynomials in one variable, x, and we wish to know the coefficients c_k of x^k. If all w_{ij} were 1, c_k would represent the number of perfect matchings with exactly k red edges. If there does exist a perfect matching with k red eges, then $\Pr(c_k = 0) \leq \frac{1}{2}$ by the same argument we derived the probability that $\det(A) = 0$ when a perfect matching exists, since we can always decompose the determinant into a sum of products of matrices with x^k.

We can now compute all the c_k by computing the determinant of A in $n+1$ different points and interpolating from that data to compute the coefficients.

3 Markov Chains

A lot of recent randomized algorithms are based on the idea of rapidly mixing Markov chains. A Markov chain is a stochastic process, i.e. a random process that evolves with time. It is defined by:

- A set of states (that we shall assume to be finite) $1, \ldots, N$.
- A transition matrix P where the entry p_{ij} represents the probability of moving to state j when at state i, i.e. $p_{ij} = Pr\left[X_{t+1} = j \mid X_t = i\right]$, where X_t is a random variable denoting the state we are in at time t.

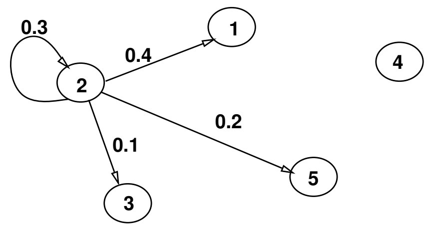


Figure 4: A Markov Chain.

Figure 4 partially illustrates a set of states having the following transition matrix:

$$P = \begin{pmatrix} 0 & 0 & 1 & 0 & 0 \\ 0.4 & 0.3 & 0.1 & 0 & 0.2 \\ 0 & 0.5 & 0 & 0 & 0.5 \\ 0.2 & 0.8 & 0 & 0 & 0 \\ 0.1 & 0.1 & 0.1 & 0.1 & 0.6 \end{pmatrix} \tag{6}$$

The transition matrix P satisfies the following two conditions and any such matrix is called "stochastic":

- $P \geq 0$
- $\sum_j p_{ij} = 1$ for all i.

Suppose we are given an initial probability distribution $\pi^{(0)}$ where we denote $\Pr(X_0 = i)$ by $\pi_i^{(0)}$. Then,

$$\begin{aligned}
\pi_j^{(1)} &= Pr(X_1 = j) \\
&= \sum_i Pr(X_1 = j \mid X_0 = i) Pr(X_0 = i) \\
&= \sum_{i=1}^{n} p_{ij} \cdot \pi_i^0,
\end{aligned}$$

i.e.

$$\pi^{(1)} = \pi^{(0)} \cdot P.$$

Repeating this argument, we obtain

$$\pi^{(s)} = \pi^{(0)} \cdot P^s.$$

Here, $\pi^{(s)}$ represents the distribution after s steps. Therefore, the matrix P^s is the so-called "s-step transition matrix". Its entries are $p_{ij}^s = Pr\left[X_{t+s} = j \mid X_t = i\right]$.

Definition 1 *A Markov Chain is said to be "ergodic" if* $\lim_{s\to\infty} p_{ij}^s = \pi_j > 0$ *for all* j *and is independent of* i.

In this case,

$$\begin{aligned}
P^{\infty} &= \lim_{s\to\infty} [P^s] \\
&= \begin{bmatrix} \pi_1 & \dots & \pi_j & \dots & \pi_n \\ \vdots & & \vdots & & \vdots \\ \pi_1 & \dots & \pi_j & \dots & \pi_n \end{bmatrix}
\end{aligned}$$

Hence, π is independent of the starting distribution $\pi^{(0)}$:

$$\pi = \pi^{(0)} \cdot P^{\infty}.$$

Any vector π which satisfies $\pi P = \pi$ and $\sum_i \pi_i = 1$ is called a *stationary distribution.*

Proposition 7 *For an ergodic MC, π is a stationary distribution, and moreover it is the unique stationary distribution.*

Proof:

We have already shown that $\pi^{(0)} P = \pi^{(1)}$ which implies

$$P^{\infty} = \lim_{s\to\infty} P^{s+1} = (\lim_{s\to\infty} P^s) P = P^{\infty}\ P$$

Since $\pi^{(0)}P^\infty = \pi$ for any probability distribution $\pi^{(0)}$, we have $\pi^{(0)}P^\infty = \pi^{(0)}P^\infty P$ which implies $\pi P = \pi$. Since $P \cdot 1 = 1$ where 1 is the vector of 1's, we derive that $P^\infty \cdot 1 = 1$, which says that $\sum_i \pi_i = 1$.

The reason why there is a unique stationary distribution is simply because by starting from another stationary distribution, say $\tilde{\pi}$, we always remain in this distribution implying that $\tilde{\pi} = \pi$. □

Proposition 7 gives an "easy" way to calculate the limiting distribution of an ergodic Markov chain from the transition matrix P. We just solve the linear system of equations $\pi P = \pi$, $\sum_i \pi_i = 1$.

4 Ergodicity and time reversibility

Theorem 8 *An MC is ergodic if and only if both of the following are true:*

1. *it is* irreducible. *That is, the underlying graph (consisting of states and transitions with positive probabilities on them) is strongly connected. Formally, for all i and j there is s such that $p_{ij}^{(s)} > 0$.*
2. *the chain is* aperiodic. *That is, you cannot divide states into subgroups so that you must go from one to another in succession. Formally,*

$$\gcd\{s : p_{ij}^{(s)} > 0\} = 1$$

for all i and j.

Definition 2 *An ergodic MC is called* **time reversible** *(TR) if the chain remains a Markov chain when you "go backwards". More formally, if π is the stationary distribution, then*

$$\pi_i p_{ij} = \pi_j p_{ji}$$

for all pairs of states i and j or, in words, the expected (or ergodic) flow from i to j equals the expected flow from j to i.

Proposition 9 *Consider an ergodic MC. Suppose there exists γ such that the* balance *conditions are satisfied: $\gamma_i p_{ij} = \gamma_j p_{ji}, \forall i, j$ and also, $\sum_i \gamma_i = 1$. Then γ is the stationary distribution, and clearly, the MC is TR.*

Clearly the MC in Figure 5 is ergodic (strong connectivity (i.e., irreducibility) and aperiodicity are obvious). It is clear that there exists a stationary distribution, and we can easily guess one. Consider $\pi_1 = \frac{1}{3}$ and $\pi_2 = \frac{2}{3}$. Since one can easily verify that π satisfies the balance conditions, π must be the stationary distribution (and the MC is time-reversible).

Consider an ergodic MC which is also symmetric ($p_{ij} = p_{ji}$) as in Figure 6. Then the stationary distribution is $\pi_i = \frac{1}{N}$, where N is the number of states.

In these notes, we shall consider MC's that are ergodic and symmetric, and therefore, have a uniform stationary distribution over states.

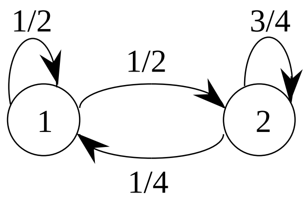


Figure 5: An example of an MC with a stationary distribution.

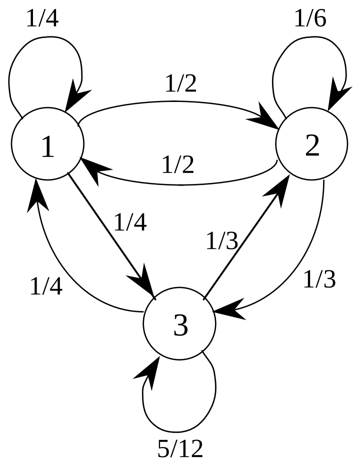


Figure 6: A symmetric ergodic MC.

5 Counting problems

Suppose you would like to count the number of blue-eyed people on a planet. One approach is to take some sort of census, checking everyone on the planet for blue eyes. Usually this is not feasible. If you know the total number of people on the planet, then you could take a sample in such a way that every person has the same probability of being selected. This would be a uniformly selected sample of size n out of a universe of size N. Let Y be the random variable representing the number of individuals in this sample that have the property (blue eyes, in our example). Then we can infer that the total number of individuals with this property is approximately $\frac{YN}{n}$.

If the actual number of individuals with the property is pN, then we have a Bernoulli process with n trials and success probability p. The random variable Y has a Binomial distribution with parameters (n, p). We are interested in finding out how close $\frac{Y}{n}$ is to p, and to do this we can use *Chernoff bounds*, which are exponentially decreasing on the tail of the distribution. Since Chernoff bounds are quite useful, we will digress for a while and derive them in some generality.

Lemma 10 *Let X_i be independent Bernoulli random variables with probability of success p_i. Then, for all $\alpha > 0$ and all $t > 0$, we have*

$$Pr\left[\sum_{i=1}^{n} X_i > t\right] \leq e^{-\alpha t} \prod_{i=1}^{n} E\left[e^{\alpha X_i}\right] = e^{-\alpha t} \prod_{i=1}^{n} \left(p_i e^{\alpha} + (1 - p_i)\right).$$

Proof:

$$Pr\left[\sum_{i=1}^{n} X_i > t\right] = Pr\left[e^{\alpha \sum_{i=1}^{n} X_i} > e^{\alpha t}\right]$$

for any $\alpha > 0$. Moreover, this can be written as $Pr[Y > a]$ with $Y \geq 0$. From Markov's inequality we have

$$Pr[Y > a] \leq \frac{E[Y]}{a}$$

for any nonnegative random variable. Thus,

$$\begin{aligned} Pr\left[\sum_{i=1}^{n} X_i > t\right] &\leq e^{-\alpha t} E\left[e^{\alpha \sum_i X_i}\right] \\ &= e^{-\alpha t} \prod_{i=1}^{n} E\left[e^{\alpha X_i}\right] \text{ because of independence.} \end{aligned}$$

The equality then follows from the definition of expectation. □

Setting $t = (1+\epsilon)E[\sum_i X_i]$ for some $\epsilon > 0$ and $\alpha = \ln(1+\epsilon)$, we obtain:

Corollary 11 *Let X_i be independent Bernoulli random variables with probability of success p_i, and let $np = E[\sum_{i=1}^{n} X_i] = \sum_{i=1}^{n} p_i$. Then, for all $\epsilon > 0$, we have*

$$Pr\left[\sum_{i=1}^{n} X_i > (1+\epsilon)np\right] \leq (1+\epsilon)^{-(1+\epsilon)np} \prod_{i=1}^{n} E\left[(1+\epsilon)^{X_i}\right] \leq \left[\frac{e^{\epsilon}}{(1+\epsilon)^{(1+\epsilon)}}\right]^{np}.$$

The second inequality of the corollary follows from the fact that

$$E\left[(1+\epsilon)^{X_i}\right] = p_i(1+\epsilon) + (1-p_i) = 1 + \epsilon p_i \leq e^{\epsilon p_i}.$$

For ϵ in the range $(0,1)$, we can simplify the above expression and write the following more classical form of the Chernoff bound:

Theorem 12 (Chernoff bound) *Let X_i be independent Bernoulli random variables with probability of success p_i, let $Y = \sum_{i=1}^n X_i$, and let $np = \sum_{i=1}^n p_i$. Then, for $1 > \epsilon > 0$, we have*

$$Pr\left[Y - np > \epsilon np\right] \leq e^{-\epsilon^2 np/3}.$$

(For other ranges of ϵ, we simply have to change the constant $1/3$ appropriately in the exponent.)

Similarly, we can write a Chernoff bound for the probability that Y is below the mean.

Theorem 13 (Chernoff bound) *Let X_i be independent Bernoulli random variables with probability of success p_i, let $Y = \sum_{i=1}^n X_i$, and let $np = \sum_{i=1}^n p_i$. Then, for $1 > \epsilon > 0$, we have*

$$Pr\left[Y - np < \epsilon np\right] \leq \left[\frac{e^{\epsilon}}{(1-\epsilon)^{(1-\epsilon)}}\right]^{np} \leq e^{-\epsilon^2 np/2}.$$

The last upper bound of $e^{-\epsilon^2/2}$ can be derived by a series expansion.

Let us go back to our counting problem. We can use Theorems 12 and 13 to see what sample size n we need to ensure that the relative error in our estimate of p is arbitrarily small. Suppose we wish to impose the bound $\Pr\left\{|\frac{Y}{n} - p| > \epsilon p\right\} \leq \delta$. Imposing $e^{-\epsilon^2 np/3} \leq \frac{\delta}{2}$, we derive that we can let the number of samples to be

$$n = \frac{3}{\epsilon^2 p} \log \frac{2}{\delta}.$$

Notice that n is polynomial in $\frac{1}{\epsilon}$, $\log \frac{1}{\delta}$, and $\frac{1}{p}$. If p is exponentially small, then this may be a bad approach. For example, if we were trying to count the number of American citizens who have dodged the draft, have become President of the country and who are being sued for sexual harassment, we would need an exponential number of trials.

These notions can be formalized as follows. Suppose we would like to compute an integral number $f(x)$ (x represents the input).

Definition 3 *An fpras (fully polynomial randomized approximation scheme) for $f(x)$ is a randomized algorithm which given x and ϵ outputs an integer $g(x)$ such that*

$$Pr\left[\left|\frac{g(x) - f(x)}{f(x)}\right| \leq \epsilon\right] \geq \frac{3}{4}$$

and runs in time polynomial in the size of the input x and in $\frac{1}{\epsilon}$.

Thus repeated sampling can be used to obtain an fpras if we can view $f(x)$ as the number of elements with some property in a universe which is only polynomially bigger, and if we can sample uniformly from this universe. Notice that the running time is assumed to be polynomial in $\frac{1}{\epsilon}$. If we were to impose the stronger condition that the running time be polynomial in $\ln \frac{1}{\epsilon}$ then we would be able to compute $f(x)$ exactly in randomized polynomial time whenever the size of the universe is exponential in the input size (simply run the fpras with ϵ equal to the inverse of the size of the universe).

Going back to our original counting problem and assuming that p is not too small, the question now is how to draw a uniformly selected individual on the planet (or more generally a uniformly generated element in the universe under consideration). One possible approach is to use a Markov chain where there is one state for each individual. Assuming that each individual has at most 1000 friends, and that "friendship" is symmetric, we set the transition probability from an individual to each of his friends to be $\frac{1}{2000}$. Then if an individual has k friends, the transition probability to himself will be $1 - \frac{k}{2000} \geq \frac{1}{2}$, implying that the chain is aperiodic.

If the graph of friendship is strongly connected (everyone knows everyone else through some sequence of friends of friends) then this MC is ergodic, and the stationary distribution is the uniform distribution on states.

Recall that

$$\lim_{s \to \infty} P^s = P^\infty = \begin{pmatrix} \pi_1 & \cdots & \pi_j & \cdots & \pi_n \\ \pi_1 & \cdots & \pi_j & \cdots & \pi_n \\ \vdots & \ddots & \vdots & \ddots & \vdots \\ \pi_1 & \cdots & \pi_j & \cdots & \pi_n \end{pmatrix}$$

If this limit converges quickly, we can simulate the MC for a finite number of steps and get "close" to the stationary distribution. Therefore, it would be useful for us to know the rate of convergence to the stationary distribution if we want to use Markov chains to approximately sample for a distribution.

It turns out that the rate of convergence is related to the eigenvalues of the transition matrix, P. Given a stochastic matrix P (recall that P is stochastic if it is nonnegative and all row sums are 1) with eigenvalues $\lambda_1, \ldots, \lambda_N$, we have the following.

1. All $|\lambda_i| \leq 1$. Indeed, if $Pe_i = \lambda e_i$ then $P^s e_i = \lambda^s e_i$, and the fact that the LHS is bounded implies that the RHS must also be.

2. Since P is stochastic, $\lambda_1 = 1$ ($P1 = 1$).

3. The MC is ergodic iff $|\lambda_i| < 1$ for all $i \neq 1$.

4. If P is symmetric then all eigenvalues are real.

5. if P is symmetric and stochastic then if $p_{ii} \geq \frac{1}{2}$ for all i then $\lambda_i \geq 0$ for all i. Indeed, if we let $Q = 2P - I$, then Q is stochastic. Hence, the ith eigenvalue for Q, $\lambda_i^Q = 2\lambda_i - 1$, but $|2\lambda_i - 1| \leq 1$ implies that $0 \leq \lambda_i \leq 1$.

6. **Speed of convergence**
The speed of convergence is dictated by the second largest eigenvalue. For simplicity, suppose P has N linearly independent eigenvectors. Then P can be expressed as $A^{-1}DA$ where D is the diagonal matrix of eigenvalues ($D_{ii} = \lambda_i$). And
$$P^2 = A^{-1}DAA^{-1}DA = A^{-1}D^2A,$$
or, in general,
$$\begin{aligned} P^s &= A^{-1}D^sA \\ &= \sum_{i=1}^{N} \lambda_i^s M_i \\ &= M_1 + \sum_{i=2}^{N} \lambda_i^s M_i \end{aligned}$$
where M_i is the matrix formed by regrouping corresponding terms from the matrix multiplication. If the MC is ergodic, then for $i \neq 1$, $\lambda_i < 1$, so $\lim_{s\to\infty} \lambda_i^s = 0$ implying $M_1 = P^\infty$. Then $P^s - P^\infty = \sum_{i=2}^{N} \lambda_i^s M_i$ is dominated by the term corresponding to $\lambda_{max} = \max_{i\neq 1} |\lambda_i|$. More generally,

Theorem 14 *Consider an ergodic time-reversible MC with stationary distribution π. Then the relative error after t steps is*
$$\Delta = \max_{i,j} \frac{|p_{ij}^{(t)} - \pi_j|}{\pi_j} \leq \frac{\lambda_{max}^t}{\min_j \pi_j}.$$
In particular, for an ergodic symmetric chain with $p_{ii} \geq \frac{1}{2}$, we have $\lambda_1 > \lambda_2 \geq \ldots \geq \lambda_N \geq 0$, so $\lambda_{max} = \lambda_2$.

Corollary 15 *For an ergodic symmetric MC with $p_{ii} \geq \frac{1}{2}$, the relative error $\Delta \leq \epsilon$ if $t \geq \frac{\log(N/\epsilon)}{\log(1/\lambda_2)}$.*

Returning to our example: In order to calculate how many iterations are needed until we are arbitrarily close to the uniform distribution π, we need to evaluate the second eigenvalue. For this purpose, Jerrum and Sinclair [11] have derived a relationship between the so-called *conductance* of the MC and λ_{max}. Their result can be viewed as an isoperimetric inequality, and its derivation is analogous to an isoperimetric inequality of Cheeger [4] for Riemannian manifolds, or results on expanders by Alon [1] and Alon and Milman [2].

6 Conductance of Markov chains (Jerrum-Sinclair)

Given a set S of states, let C_S denote the capacity of S, which is the probability of being in some state in S when steady-state is reached. Specifically,

$$C_S = \sum_{i \in S} \pi_i.$$

Define F_S, the ergodic flow out of S (expected flow) so that

$$F_S = \sum_{i \in S, j \notin S} \pi_i p_{ij},$$

(summing over transitions from S to the complement of S, $\overline{S}$). Clearly $F_S \leq C_S$.

Define $\Phi_S = F_S / C_S$, which is the probability of leaving S given that we are already in S. Define the *conductance* of the MC

$$\Phi := \min_{S: C_S \leq \frac{1}{2}} \Phi_S.$$

Intuitively, if we have an MC with small conductance, then once we are in a set S, we are "stuck" in S for a long time. This implies that it will take a long time to reach the stationary distribution, so the rate of convergence will be small. We might therefore expect that λ_2 will be close to 1 if the conductance is small.

Theorem 16 (Jerrum-Sinclair[11]) *For an ergodic MC that is TR, we have*

$$\lambda_2 < 1 - \Phi^2/2.$$

Remark 1 *There exist corresponding lower bounds expressing that $\Delta \geq \lambda_2^t$ and $\lambda_2 \geq 1 - 2\Phi$. This therefore shows that the conductance is an appropriate measure of the speed of convergence.*

7 Evaluation of Conductance of Markov Chains

Given a markov chain, the task will be to evaluate the conductance Φ. In order to generate Markov chains with a uniform steady state distribution and rapidly mixing property, we restrict our attention to the following MC: symmetric and equal transition probability p between states having a transition (i.e. for all $i \neq j$, either $p_{ij} = p_{ji} = 0$ or $p_{ij} = p_{ji} = p$). Therefore, it will have a uniform steady state distribution $\pi_i = 1/N$, where N denotes the number of states. Instead of looking at the MC, we can look at the underlying graph $G = (V, E)$, $E = \{(i,j) : p_{ij} = p_{ji} = p\}$. For a set S of states, let $\delta(S) = \{(i,j) \in E : i \in S, j \notin S\}$, then,

$$C_S = \sum_{i \in S} \pi_i = \frac{1}{N} |S|,$$

$$F_S = \sum_{i \in S, j \notin S} \pi_i p_{ij} = p \cdot \frac{1}{N} |\delta(S)|,$$

$$\Phi_S = \frac{F_S}{C_S} = p \frac{|\delta(S)|}{|S|}.$$

Definition 4 *The magnification factor of G is*

$$\gamma(G) = \min_{0 < |S| \leq \frac{|V|}{2}} \frac{|\delta(S)|}{|S|}.$$

Therefore,

$$\Phi = \min_S \Phi_S = p \cdot \gamma(G).$$

In the rest of this section, we study the conductance of a simple MC, which will be useful in the problem of counting matchings. Take a MC on states that are all binary numbers with d bits so that the underlying graph is a d-cube (with 2^d states). Two nodes are adjacent only if they differ by exactly one bit. Refer to Figure 7 for a 3-cube.

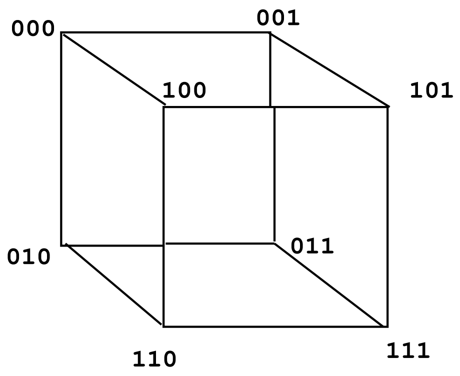


Figure 7: A 3-cube Markov chain.

If we put probabilities of $\frac{1}{2d}$ on all out-transitions then the self loops get probabilities $\frac{1}{2}$. The MC is symmetric and ergodic, so $\pi_i = \frac{1}{2^d}$. If we simulate a random walk on this d-cube, we obtain a random d-bit number.

Claim 17 *The magnification factor of the d-cube $\gamma(G) = 1$.*

Proof:

1. $\gamma(G) \leq 1$.
   Let S_1 be a vertex set of "half cube" (e.g. all vertices with state number starting with 0). Then clearly, every vertex in S_1 will have exactly one edge incident to it leaving S_1. Therefore, $|\delta(S_1)| = |S_1| = \frac{|V|}{2}$, and

$$\gamma(G) = \min_{0<|S|\leq\frac{|V|}{2}} \frac{|\delta(S)|}{|S|} \leq \frac{|\delta(S_1)|}{|S_1|} = 1.$$

2. $\gamma(G) \geq 1$.
   For all x_1, x_2, define a random path between x_1 and x_2 selected uniformly among all shortest paths between x_1 and x_2. For example, for

$$\begin{array}{rcccccccc} x_1 & = & 1 & 0 & 1 & 1 & 0 & 1 & 0 \\ x_2 & = & 1 & 0 & 0 & 1 & 1 & 1 & 1 \end{array}$$

we first look at the bits in which they differ (3rd, 5th and 7th). The order in which you change these bits determines a (or all) shortest path in the d-cube.

Given $e \in E$, by symmetry,

$$E[\# \text{ of paths through e}] = \frac{1}{|E|} \cdot T,$$

where T is the total length of all shortest paths. We can compute this by first choosing x_1 (2^d choices), sum up all the paths from x_1(from length 1 to d). For any path p, we count it twice: $x_1 \to x_2$ and $x_2 \to x_1$. Thus,

$$T = \frac{1}{2} \cdot 2^d \sum_{k=0}^{d} \binom{d}{k} k.$$

This can also be written as $T = \frac{1}{2} \cdot 2^d \sum_{k=0}^{d} \binom{d}{k}(d-k)$. Taking the average of these two expressions, we deduce that

$$T = \frac{1}{2} \cdot d2^d \sum_{k=0}^{d} \binom{d}{k} = d2^{2d-2}.$$

On the other hand, $|E| = \frac{1}{2}2^d \cdot d$. Hence,

$$E[\# \text{ of paths through } e] = \frac{T}{|E|} = \frac{d2^{2d-2}}{d2^{d-1}} = 2^{d-1}.$$

Consider any set S. By linearity of expectations,

$$E[\# \text{ of paths intersecting } \delta(S)] \leq \sum_{e\in\delta(S)} E[\# \text{ of paths through } e] = 2^{d-1}|\delta(S)|.$$

However, the number of paths crossing $\delta(S)$ should be at least $|S| \cdot |\overline{S}|$. This implies that

$$|S| \cdot |\overline{S}| \leq E[\# \text{ of paths through } \delta(S)] = 2^{d-1} \cdot |\delta(S)|.$$

Since $|S| \leq \frac{|V|}{2}$, $|\overline{S}| \geq \frac{|V|}{2} = 2^{d-1}$. Therefore, for any set S,

$$\frac{|\delta(S)|}{|S|} \geq \frac{|\overline{S}|}{2^{d-1}} \geq 1.$$

So, $\gamma(G) \geq 1$.

□

This gives us the conductance of the corresponding MC: $\Phi = p \cdot \gamma(G) = \frac{1}{2d}$. Then $\lambda_2 \leq 1 - \frac{\Phi^2}{2} = 1 - \frac{1}{8d^2}$. The steady-state distribution is $\pi_j = \frac{1}{2^d}$ for all j. Thus, the relative error after t steps is

$$\Delta = \max_{i,j} \frac{p_{ij}^t - \pi_j}{\pi_j} \leq \frac{\lambda_2^t}{\min_j \pi_j} \leq 2^d \cdot \left(1 - \frac{1}{8d^2}\right)^t.$$

If we want $\Delta \leq \epsilon$, we shall choose t such that:

$$\begin{aligned} t &\geq \frac{d \ln 2 - \ln \epsilon}{-\ln(1 - \frac{1}{8d^2})} \\ &\geq 8d^2 \cdot (d \ln 2 - \ln \epsilon). \end{aligned}$$

In this case, although the MC has an exponential number of states (2^d), we only need $O(d^3)$ steps to generate an almost uniform state (with ϵ say constant or even as small as $e^{-O(d)}$).

In general, let M be an ergodic, time reversible Markov chain with $e^{q(n)}$ states, where $q(n)$ is a polynomial in n (n represents the size of the input). If its conductance $\Phi \geq \frac{1}{p(n)}$, where $p(n)$ is a polynomial in n, we will say that it has the *rapidly mixing property.* The relative error after t steps is

$$\Delta_t \leq \frac{\lambda_2^t}{\min_j \pi_j} \leq \frac{(1 - \frac{\Phi^2}{2})^t}{e^{q(n)}} \leq \epsilon,$$

To get $\Delta_t \leq \epsilon$, we only need to take $t = 2p^2(n)\left(q(n) + \ln \frac{1}{\epsilon}\right)$ steps, a polynomial number in n and $\ln \frac{1}{\epsilon}$.

Thus a rapidly-mixing MC with uniform stationary distribution with state space M can be used as an ϵ-sampling scheme on M:

Definition 5 *A* fully polynomial ϵ-sampling scheme *(also called an* ϵ-generator*) for a set M is an algorithm that runs in time poly(size of input, $\ln \frac{1}{\epsilon}$), and outputs an element $x \in M$ with probability $\pi(x)$ such that*

$$\max_{x \in M} \left| \pi(x) - \frac{1}{|M|} \right| \leq \frac{\epsilon}{|M|}.$$

(M is typically given implicitly by a relation (i.e. on input x, M is the set of strings y satisfying some relation $< x, y >$).)

8 Approximation by sampling

We now sketch how an ϵ-sampling scheme can be used to develop a randomized approximation algorithm for the counting problem we discussed before. To evaluate $|M|$, first we immerse M into a larger set $V \supseteq M$. Then we sample from V, and approximate $\frac{|M|}{|V|}$ by

$$\frac{\text{size of } (M \cap \text{ sample})}{\text{size of sample}}.$$

This scheme works well if $|M|$ is polynomially comparable to $|V|$. But if $|M| \ll |V|$ (i.e. $|M|$ exponentially smaller than $|V|$), this scheme will have trouble, since in order to obtain a small relative error in the approximation, the number of samples will need to be so large (i.e. exponential) as to make this approach infeasible. (See our previous discussion for the problem of counting individuals, and our study of Chernoff bounds.)

Example: Suppose we wish to approximate π. If we take a square with side-length 2 and we inscribe within it a circle of radius 1, then the ratio of the area of the circle to the area of the square is $\pi/4$. Thus the probability that a uniformly generated point in the square belongs to the circle is precisely $\pi/4$.

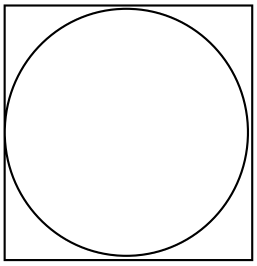


Figure 8: How (not) to calculate π.

By generating points within the square at random according to a uniform distribution, we can approximate π as simply 4 times the fraction of points that lie within the circle. The accuracy of such an approximation depends on how closely we can approximate the uniform distribution and on the number of samples. However, we will run into trouble if we want to estimate $vol(B_n)$ by using the same method, where $B_n = \{x \in R^n : ||x|| \leq 1\}$, since the volume is exponentially smaller than the volume of the corresponding cube (2^n). Nevertheless, a very nice application of rapidly mixing markov chains is precisely in the computation of the volume of a convex body or region [6]. To avoid the problem just mentioned, what is done is to immerse the body whose volume V_0 needs to be computed in a sequence of bodies of volumes V_1,

$V_2, \ldots,$ such that V_i/V_{i+1} is is polynomially bounded. Then one can evaluate V_0 by the formula:

$$\frac{V_0}{V_n} = \frac{V_0}{V_1} \cdot \frac{V_1}{V_2} \cdot \frac{V_2}{V_3} \cdots \frac{V_{n-1}}{V_n}.$$

We now show how this technique can be used to develop a fully polynomial randomized approximation scheme for computing the permanent of a class of 0-1 matrices.

9 Approximating the permanent

Recall that for an $n \times n$ 0-1 matrix A, the *permanent* of A, $\mathrm{perm}(A)$, is the number of perfect matchings in the bipartite graph G whose incidence matrix is A. It is known that computing $\mathrm{perm}(A)$ is #P-complete.

In order to develop an approximation scheme for the permanent, we use the technique of approximation by sampling. As a naive adoption of this technique, we can generate edge sets at random and count the fraction that are perfect matchings. Unfortunately, this scheme may resemble searching for a needle in a haystack. If the fraction of edge sets that are perfect matchings is very small, then in order to obtain a small relative error in the approximation, the relative error in the sampling may need to be so small and the number of samples may need to be so large as to make this approach infeasible.

Instead of trying to directly approximate the fraction of edge sets that are perfect matchings, we can try to approximate a different ratio from which the permanent can be computed. Specifically, for $k = 1, 2, \ldots, n$, let $\mathcal{M}_k$ denote the set of matchings with size k, and let $m_k = |\mathcal{M}_k|$ denote the number of matchings with size k. The permanent of A is then given by $\mathrm{perm}(A) = m_n$, and we can express $\mathrm{perm}(A)$ as the product of ratios:

$$\mathrm{perm}(A) = \frac{m_n}{m_{n-1}} \frac{m_{n-1}}{m_{n-2}} \cdots \frac{m_2}{m_1} m_1 \tag{7}$$

($m_1 = |E|$). Thus, we can approximate the permanent of A by approximating the ratios m_k/m_{k-1} for $k = 2, 3, \ldots, n$. We write m_k/m_{k-1} as

$$\frac{m_k}{m_{k-1}} = \frac{u_k}{m_{k-1}} - 1 \tag{8}$$

where $u_k = |\mathcal{U}_k|$ and $\mathcal{U}_k = \mathcal{M}_k \cup \mathcal{M}_{k-1}$ (see Figure 9), and then we use an ϵ-sampling scheme for $\mathcal{U}_k$ to approximate the fraction m_{k-1}/u_k. To summarize our approach, for each $k = 2, 3, \ldots, n$, we take random samples over a uniform distribution on the set of matchings of size k and $k-1$, and we count the fraction that are matchings of size $k-1$; this gives us m_{k-1}/u_k, and we use Equation 8 to get m_k/m_{k-1}. Equation 7 then gets us $\mathrm{perm}(A)$.

The following two claims establish the connection between ϵ-sampling of $\mathcal{U}_k$ and approximation of the permanent of A.

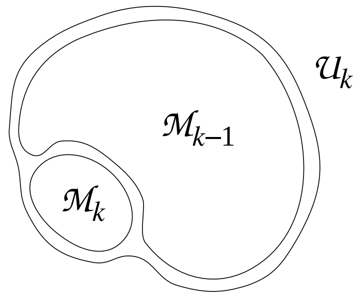


Figure 9: Each matching in $\mathcal{U}_k$ has size either k or $k-1$.

Claim 18 (Broder) *If there is a fully polynomial ϵ-sampling scheme for $\mathcal{U}_k = \mathcal{M}_k \cup \mathcal{M}_{k-1}$, then there exists a randomized approximation scheme for m_k/m_{k-1} that runs in time that is polynomial in the values $1/\epsilon$, n, and u_k/m_k.* ∎

Claim 19 *If for each $k = 2, 3, \ldots, n$, there is a fully polynomial ϵ-sampling scheme for $\mathcal{U}_k$, then there exists a randomized approximation scheme for the permanent of A that runs in time that is polynomial in the values $1/\epsilon$, n, and $\max_k(u_k/m_k)$.* ∎

For those graphs with $\max_k(u_k/m_k)$ bounded by a polynomial in n, Claim 19 gives us a fully polynomial randomized approximation scheme for the permanent — provided, of course, that we can produce an ϵ-sampling scheme for $\mathcal{U}_k$. In fact, it turns out that

$$\max_k \left\{ \frac{u_k}{m_k} \right\} = \frac{u_n}{m_n}.$$

This is because $\{m_k\}$ is log-concave (i.e. $m_k m_{k+2} \leq m_{k+1}^2$). Thus, if we can develop an ϵ-sampling scheme for the matchings $\mathcal{U}_k$ for each $k = 2, 3, \ldots, n$, then for the class of graphs with u_n/m_n bounded by a polynomial in n, we have an fpras for the permanent. After developing an ϵ-sampling scheme, we will look at such a class of graphs.

An ϵ-sampling scheme for matchings

We now turn our attention to developing an ϵ-sampling scheme for $\mathcal{U}_k = \mathcal{M}_k \cup \mathcal{M}_{k-1}$, and it should come as no surprise that we will use the technique of rapidly mixing Markov chains.

We now define a Markov chain whose states are matchings in $\mathcal{U}_k$. Consider any pair M_i, M_j of states (matchings) and create a transition between them according to the following rules:

- If M_i and M_j differ by the addition or removal of a single edge, that is, $M_i \triangle M_j = \{e\}$ for some edge e, then there is a transition from M_i to M_j and a transition from M_j to M_i. Both transitions have probability $p_{ij} = p_{ji} = 1/2m$ where m denotes the number of edges in the graph. $\big(M_i \triangle M_j = (M_i - M_j) \cup (M_j - M_i)\big)$

- If M_i and M_j are both matchings of size $k-1$ and they differ by removing one edge and adding another edge that has an endpoint in common with the removed edge, that is, $M_i, M_j \in \mathcal{M}_{k-1}$ and $M_i \triangle M_j = \{(u,v),(v,w)\}$ for some pair of edges (u,v) and (v,w), then there is a transition from M_i to M_j and a transition from M_j to M_i. Both transitions have probability $p_{ij} = p_{ji} = 1/2m$.

To complete the Markov chain, for each state M_i, we add a loop transition from M_i to itself with probability p_{ii} set to ensure that $\sum_j p_{ij} = 1$.

It is easy to see that this Markov chain is ergodic since the self-loops imply aperiodicity, and irreducibility can be seen from the construction. Irreducibility implicitly assumes the existence of some matching of size k; otherwise the chain might not be irreducible. The proof of irreducibility indeed stems on the fact that any matching of size k can reach any matching of size $k-1$, implying that one can reach any matching from any other matching provided a matching of size k exists. This Markov chain is time reversible and has the desired uniform steady state probability distribution, $\pi_i = 1/u_k$, since it is clearly symmetric. Furthermore, $p_{ii} \geq 1/2$ for each state M_i, and therefore, $\lambda_{\max} = \lambda_2$ which means that we can bound the relative error after t steps by:

$$\Delta_t \leq u_k \left(1 - \frac{\Phi^2}{2}\right)^t.$$

Finally, this Markov chain also has the property that $p_{ij} = p = 1/2m$ for every transition with $i \neq j$, and this property allows us to compute Φ by:

$$\Phi = \frac{1}{2m}\gamma(H)$$

where we recall that $\gamma(H)$ is the magnification of the underlying graph H (not the graph G on which we are sampling matchings).

If we could now lower bound $\gamma(H)$ by $\gamma(H) \geq 1/p(n)$ where $p(n)$ is a polynomial in n, then since $m \leq n^2$, we would have $\Phi \geq 1/p'(n)$ ($p'(n)$ is also a polynomial in n), and so we would have a fully polynomial ϵ-sampling scheme for $\mathcal{U}_k$. We cannot actually show such a lower bound, but the following theorem gives us a lower bound of $\gamma(H) \geq m_k/cu_k$ (c is a constant), and this, by Claim 19, is sufficient to give us a randomized approximation scheme that is polynomial in $1/\epsilon$, n, and u_n/m_n.

Theorem 20 (Dagum, Luby, Mihail and Vazirani [5]) *There exists a constant c such that*

$$\gamma(H) \geq \frac{1}{c}\frac{m_k}{u_k}.$$

Corollary 21 *There exists a fully polynomial ϵ-sampling scheme for $\mathcal{U}_k$ provided that $\frac{u_k}{m_k} = O(n^{c'})$ for some c'.*

A preliminary lemma is required.

Lemma 22 *Let M_1, M_2 be matchings in a bipartite graph. Then $M_1 \triangle M_2$ is a union of vertex disjoint paths and cycles which alternate between the edges of M_1 and M_2.*

Sketch of the proof of main result: For each pair of states M_1, M_2 with $M_1 \in \mathcal{M}_{k-1}$ and $M_2 \in \mathcal{M}_k$, we pick a random path from M_1 to M_2 in H as follows. By lemma 22 we can write the symmetric difference of the matchings M_1 and M_2 as

$$M_1 \triangle M_2 = C \cup D \cup E$$

where each element of C denotes a cycle or path in G with the same number of edges from M_1 as from M_2, each element of D denotes a path in G with one more edge from M_2 than from M_1, and each element of E denotes a path in G with one more edge from M_1 than from M_2. Notice that there must be exactly one more element in D than in E. We order the paths in each set at random so $[C_1, C_2, \ldots, C_r]$ is a random permutation of the r paths (or cycles) in C, $[D_1, D_2, \ldots, D_q]$ is a random permutation of the q paths in D, and $[E_1, E_2, \ldots, E_{q-1}]$ is a random permutation of the $q-1$ paths in E. A path from M_1 to M_2 in H is then given by:

$$\begin{tikzcd}
M_1 \arrow[dr, "D_1"] & \\
& M' = M_1 \triangle D_1 \arrow[dl, "E_1"] \\
M'' = M' \triangle E_1 & \\
\vdots \arrow[dr, "D_q"] & \\
& M''' \arrow[d, "C_1"] \\
& \vdots \arrow[d, "C_r"] \\
& M_2
\end{tikzcd}$$

Of course, $M_1 \to (M_1 \triangle D_1)$ may not actually be a transition of the chain, but $M_1 \to (M_1 \triangle D_1)$ does define a path of transitions if we use the edges of D_1 two at a time.

The crucial part of this proof is showing that there exists a constant c such that for any edge e of H, the expected number of paths that go through e is upper bounded by

$$\mathrm{E}\,[\text{number of paths through } e] \leq c m_{k-1}.$$

We will not do this part of the proof. Now if we consider any set $S \subseteq V$ of vertices from H, by linearity of expectation, the expected number of paths that cross from S over to $\overline{S}$ is upper bounded by

$$\mathrm{E}\left[\text{number of paths crossing } S\right] \leq cm_{k-1}\left|\delta(S)\right|$$

where we recall that $\delta(S)$ denotes the coboundary of S. Therefore, there exists some choice for the paths such that not more than $cm_{k-1}\left|\delta(S)\right|$ of them cross the boundary of S.

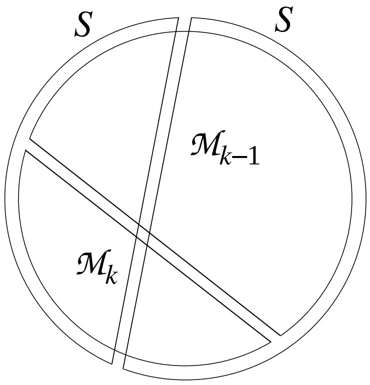


Figure 10: Partitioning $\mathcal{U}_k$.

Each vertex of S is either a matching of $\mathcal{M}_k$ or a matching of $\mathcal{M}_{k-1}$ and likewise for $\overline{S}$, so we can partition $\mathcal{U}_k$ as shown in Figure 10. We assume, without loss of generality, that

$$\frac{|S \cap \mathcal{M}_k|}{|S|} \geq \frac{m_k}{u_k}, \tag{9}$$

and therefore,

$$\frac{\left|\overline{S} \cap \mathcal{M}_{k-1}\right|}{\left|\overline{S}\right|} \geq \frac{m_{k-1}}{u_k} \tag{10}$$

(otherwise, we exchange S and $\overline{S}$). The number of paths that cross S must be at least $|S \cap \mathcal{M}_k||\overline{S} \cap \mathcal{M}_{k-1}|$ since for any $M_1 \in \overline{S} \cap \mathcal{M}_{k-1}$, $M_2 \in S \cap \mathcal{M}_k$ there must be a path from M_1 to M_2 which crosses from S to $\overline{S}$. By multiplying together Inequalities 9 and 10, we have

$$|S \cap \mathcal{M}_k|\left|\overline{S} \cap \mathcal{M}_{k-1}\right| \geq \frac{m_k m_{k-1}\left|S\right|\left|\overline{S}\right|}{u_k^2}.$$

But we have already seen that we can choose paths so that the number of paths crossing the boundary of S is not more than $cm_{k-1}\,|\delta(S)|$. Therefore, it must be the case that

$$cm_{k-1}\,|\delta(S)| \geq \frac{m_k m_{k-1}\,|S|\,\left|\overline{S}\right|}{u_k^2}.$$

Notice that this statement is unchanged if we replace S by $\bar{S}$. So, without loss of generality $|S| \leq \frac{u_k}{2}$ which implies that $\left|\overline{S}\right| \geq \frac{u_k}{2}$. Hence

$$\frac{m_k}{u_k^2}\,|S|\,\left|\overline{S}\right| \geq \frac{m_k}{u_k}\frac{|S|}{2}$$

which implies that

$$\frac{|\delta(S)|}{|S|} \geq \frac{1}{2c}\frac{m_k}{u_k}$$

$$\gamma(H) \geq \frac{1}{2c}\frac{m_k}{u_k}.$$

■

A class of graphs with u_n/m_n polynomially bounded

We finish this discussion by considering a class of graphs for which u_n/m_n is bounded by a polynomial in n. Specifically, we consider the class of *dense* bipartite graphs. A dense bipartite graph is a bipartite graph in which every vertex has degree at least $n/2$ (recall that n is number of vertices on each side of the bipartition).

We now show that for dense bipartite graphs, $m_{n-1}/m_n \leq n^2$. Since $u_n/m_n = 1+m_{n-1}/m_n$, this bound gives us the desired result. Consider a matching $M \in \mathcal{M}_{n-1}$ with edges $\{(u_1,v_1),(u_2,v_2),\ldots,(u_{n-1},v_{n-1})\}$ so that u_n and v_n are the two exposed vertices. Since both u_n and v_n have degree at least $n/2$, there are the following two possibilities.

- (u_n,v_n) is an edge which implies that $M' := M \cup \{(u_n,v_n)\} \in \mathcal{M}_n$.
- There exists an i for $1 \leq i \leq n-1$ such that both (u_n,v_i) and (u_i,v_n) are edges (this follows from the pigeonhole principle). In this case $M' := M - \{(u_i,v_i)\} \cup \{(u_n,v_i),(u_i,v_n)\} \in \mathcal{M}_n$.

Thus we can define a function $f : \mathcal{M}_{n-1} \to \mathcal{M}_n$ by letting $f(M) = M'$.

Now consider a matching $M' \in \mathcal{M}_n$, and let $f^{-1}(M')$ denote the set of matchings $M \in \mathcal{M}_{n-1}$ such that $f(M) = M'$. For each $M \in \mathcal{M}_{n-1}$ such that $f(M) = M'$, M can be obtained from M' in one of two different ways.

- Some pair of edges $(u_i,v_i),(u_j,v_j)$ are removed from M' and replaced with a single edge that must be either (u_i,v_j) or (u_j,v_i). We can choose the pair of edges to remove in $\binom{n}{2}$ ways and we can choose the replacement edge in at most 2 ways.

- An edge (u_i, v_i) is removed from M'. This edge can be chosen in n ways.

Thus, there are at most

$$2\binom{n}{2} + n = n(n-1) + n = n^2$$

matchings in $\mathcal{M}_{n-1}$ that could possibly map to M'. This means that $|f^{-1}(M')| \leq n^2$ for every matching $M' \in \mathcal{M}_n$, and therefore, $m_{n-1}/m_n \leq n^2$.

Thus we have shown that for dense bipartite graphs, there exists a fully polynomial randomized approximation scheme for the permanent. This result is tight in the sense that graphs can be constructed such that $\frac{m_{n-1}}{m_n}$ is exponential and whose vertices have degree $\geq \frac{n}{2} - \epsilon$. There is also a theorem of Broder which says that

Theorem 23 *Counting perfect matchings on dense bipartite graphs is #P-complete.*

Other applications of Markov Chains

There are other uses for Markov Chains in the design of algorithms. Of these the most interesting is for computing the volume of convex bodies. It can be shown that a fully polynomial randomized approximation scheme may be constructed for this problem [6]. This is interesting because it can also be shown that one cannot even approximate the volume to within an exponential factor of n^{cn} for $c < 0.5$ in polynomial time, where n is the dimension [3].

References

[1] N. Alon. Eigenvalues and expanders. *Combinatorica*, 6:83–96, 1986.

[2] N. Alon and V. Milman. λ_1, isoperimetric inequalities for graphs and superconcentrators. *Journal of Combinatorial Theory B*, 38:73–88, 1985.

[3] I. Bárány and Z. Füredi. Computing the volume is difficult. In *Proceedings of the 18th Annual ACM Symposium on Theory of Computing*, pages 442–447, 1986.

[4] J. Cheeger. A lower bound for the smallest value of the Laplacian. In R. Gunning, editor, *Problems in analysis*, pages 195–199. Princeton University Press, 1970.

[5] P. Dagum, M. Mihail, M. Luby, and U. Vazirani. Polytopes, permanents and graphs with large factors. In *Proceedings of the 29th Annual Symposium on Foundations of Computer Science*, pages 412–422, 1988.

[6] M. Dyer, A. Frieze, and R. Kannan. A random polynomial algorithm for approximating the volume of convex bodies. *Journal of the ACM*, pages 1–17, 1991.

[7] R. Karp. An introduction to randomized algorithms. *Discrete Applied Mathematics*, 34:165–201, 1991.

[8] R. M. Karp and M. O. Rabin. Efficient randomized pattern-matching algorithms. *IBM Journal of Research and Development*, 31:249–260, 1987.

[9] R. Motwani and P. Raghavan. *Randomized Algorithms*. 1994.

[10] K. Mulmuley, U. Vazirani, and V. Vazirani. Matching is as easy as matrix inversion. *Combinatorica*, 7(1):105–113, 1987.

[11] A. Sinclair and M. Jerrum. Approximate counting, uniform generation and rapidly mixing markov chains. *Information and Computation*, 82:93–133, 1989.

18.415/6.854 Advanced Algorithms	October 1994

Linear Programming

Lecturer: Michel X. Goemans

1 An Introduction to Linear Programming

Linear programming is a very important class of problems, both algorithmically and combinatorially. Linear programming has many applications. From an algorithmic point-of-view, the simplex was proposed in the forties (soon after the war, and was motivated by military applications) and, although it has performed very well in practice, is known to run in exponential time in the worst-case. On the other hand, since the early seventies when the classes P and NP were defined, it was observed that linear programming is in NP$\cap$ co-NP although no polynomial-time algorithm was known at that time. The first polynomial-time algorithm, the ellipsoid algorithm, was only discovered at the end of the seventies. Karmarkar's algorithm in the mid-eighties lead to very active research in the area of interior-point methods for linear programming. We shall present one of the numerous variations of interior-point methods in class. From a combinatorial perspective, systems of linear inequalities were already studied at the end of the last century by Farkas and Minkovsky. Linear programming, and especially the notion of duality, is very important as a proof technique. We shall illustrate its power when discussing approximation algorithms. We shall also talk about network flow algorithms where linear programming plays a crucial role both algorithmically and combinatorially. For a more in-depth coverage of linear programming, we refer the reader to [1, 4, 7, 8, 5].

A linear program is the problem of optimizing a linear objective function in the decision variables, $x_1 \ldots x_n$, subject to linear equality or inequality constraints on the x_i's. In standard form, it is expressed as:

$$
\begin{array}{lll}
\text{Min} & \displaystyle\sum_{j=1}^{n} c_j x_j & \text{(objective function)} \\
\text{subject to:} & & \\
& \displaystyle\sum_{j=1}^{n} a_{ij} x_j = b_i, \quad i = 1 \ldots m & \text{(constraints)} \\
& x_j \geq 0, \qquad\qquad j = 1 \ldots n & \text{(non-negativity constraints)}
\end{array}
$$

where $\{a_{ij}, b_i, c_j\}$ are given.

A linear program is expressed more conveniently using matrices:

$$
\min c^T x \;\; \text{subject to} \;\; \left\{ \begin{array}{rcl} Ax & = & b \\ x & \geq & 0 \end{array} \right.
$$

where

$$
\begin{aligned}
x &= \begin{pmatrix} x_1 \\ \vdots \\ x_n \end{pmatrix} && \in \mathbb{R}^{n\times 1} \\
b &= \begin{pmatrix} b_1 \\ \vdots \\ b_m \end{pmatrix} && \in \mathbb{R}^{m\times 1} \\
c &= \begin{pmatrix} c_1 \\ \vdots \\ c_n \end{pmatrix} && \in \mathbb{R}^{n\times 1} \\
A &= \begin{pmatrix} a_{11} & & \\ & \ddots & \\ & & a_{mn} \end{pmatrix} && \in \mathbb{R}^{m\times n}
\end{aligned}
$$

2 Basic Terminology

Definition 1 *If x satisfies $Ax = b, x \geq 0$, then x is* **feasible**.

Definition 2 *A linear program (LP) is feasible if there exists a feasible solution, otherwise it is said to be* **infeasible**.

Definition 3 *An* **optimal solution** *x^* is a feasible solution s.t. $c^T x^* = \min\{c^T x : Ax = b, x \geq 0\}$.*

Definition 4 *LP is unbounded (from below) if $\forall \lambda \in \mathbb{R}$, $\exists$ a feasible x^* s.t. $c^T x^* \leq \lambda$.*

3 Equivalent Forms

A linear program can take on several forms. We might be maximizing instead of minimizing. We might have a combination of equality and inequality contraints. Some variables may be restricted to be non-positive instead of non-negative, or be unrestricted in sign. Two forms are said to be equivalent if they have the same set of optimal solutions or are both infeasible or both unbounded.

1. A maximization problem can be expressed as a minimization problem.

$$\max c^T x \Leftrightarrow \min -c^T x$$

2. An equality can be represented as a pair of inequalities.

$$
a_i^T x = b_i \quad
\begin{aligned}
&\Leftrightarrow \left\{ \begin{array}{rcr} a_i^T x & \leq & b_i \\ a_i^T x & \geq & b_i \end{array} \right. \\
&\Leftrightarrow \left\{ \begin{array}{rcr} a_i^T x & \leq & b_i \\ -a_i^T x & \leq & -b_i \end{array} \right.
\end{aligned}
$$

3. By adding a slack variable, an inequality can be represented as a combination of equality and non-negativity constraints.

$$a_i^T x \leq b_i \Leftrightarrow a_i^T x + s_i = b_i,\ s_i \geq 0.$$

4. Non-positivity constraints can be expressed as non-negativity constraints.

   To express $x_j \leq 0$, replace x_j everywhere with $-y_j$ and impose the condition $y_j \geq 0$.

5. x may be unrestricted in sign.

   If x is unrestricted in sign, i.e. non-positive or non-negative, everywhre replace x_j by $x_j^+ - x_j^-$, adding the constraints $x_j^+,\ x_j^- \geq 0$.

In general, an inequality can be represented using a combination of equality and non-negativity constraints, and *vice versa*.

Using these rules, $\min\left\{c^T x \text{ s.t. } Ax \geq b\right\}$ can be transformed into $\min\left\{c^T x^+ - c^T x^- \text{ s.t. } Ax^+ - Ax^- - Is = b,\ x^+, x^-, s \geq 0\right\}$. The former LP is said to be in **canonical** form, the latter in **standard** form.

Conversely, an LP in standard form may be written in canonical form. $\min\left\{c^T x \text{ s.t. } Ax = b,\ x \geq 0\right\}$ is equivalent to $\min\left\{c^T x \text{ s.t. } Ax \geq b,\ -Ax \geq -b,\ Ix \geq 0\right\}$. This may be rewritten as $A'x \geq b'$, where $A' = \begin{pmatrix} \text{A} \\ \text{-A} \\ \text{I} \end{pmatrix}$ and $b' = \begin{pmatrix} \text{b} \\ \text{-b} \\ \text{0} \end{pmatrix}$.

4 Example

Consider the following linear program:

$$\min x_2 \text{ subject to } \left\{ \begin{array}{rcrcl} x_1 & & & \geq & 2 \\ 3x_1 & - & x_2 & \geq & 0 \\ x_1 & + & x_2 & \geq & 6 \\ -x_1 & + & 2x_2 & \geq & 0 \end{array} \right.$$

The optimal solution is $(4, 2)$ of cost 2 (see Figure 1). If we were maximizing x_2 instead of minimizing under the same feasible region, the resulting linear program would be unbounded since x_2 can increase arbitrarily. From this picture, the reader should be convinced that, for any objective function for which the linear program is bounded, there exists an optimal solution which is a "corner" of the feasible region. We shall formalize this notion in the next section.

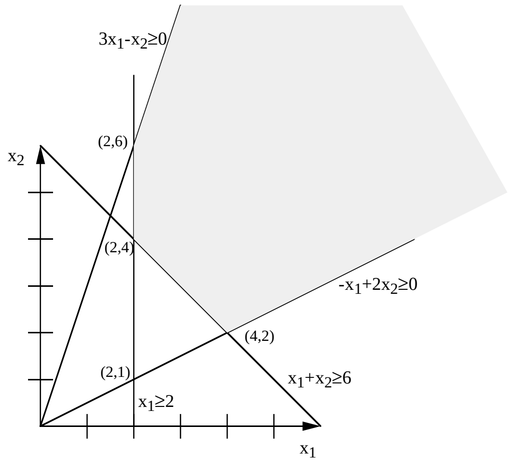


Figure 1: Graph representing primal in example.

An example of an infeasible linear program can be obtained by reversing some of the inequalities of the above LP:

$$\begin{array}{rcrcl} x_1 & & & \leq & 2 \\ 3x_1 & - & x_2 & \geq & 0 \\ x_1 & + & x_2 & \geq & 6 \\ -x_1 & + & 2x_2 & \leq & 0. \end{array}$$

5 The Geometry of LP

Let $P = \{x : Ax = b,\ x \geq 0\} \subseteq \mathbb{R}^n$.

Definition 5 *x is a* **vertex** *of P if $\nexists y \neq 0$ s.t. $x + y,\ x - y \in P$.*

Theorem 1 *Assume $\min\{c^T x : x \in P\}$ is finite, then $\forall x \in P, \exists$ a vertex x' such that $c^T x' \leq c^T x$.*

Proof:

If x is a vertex, then take $x' = x$.

If x is not a vertex, then, by definition, $\exists y \neq 0$ s.t. $x + y,\ x - y \in P$. Since $A(x + y) = b$ and $A(x - y) = b$, $Ay = 0$.

WLOG, assume $c^T y \leq 0$ (take either y or $-y$). If $c^T y = 0$, choose y such that $\exists j$ s.t. $y_j < 0$. Since $y \neq 0$ and $c^T y = c^T(-y) = 0$, this must be true for either y or $-y$.

Consider $x + \lambda y,\ \lambda > 0$. $c^T(x + \lambda y) = c^T x + \lambda c^T y \leq c^T x$, since $c^T y$ is assumed non-positive.

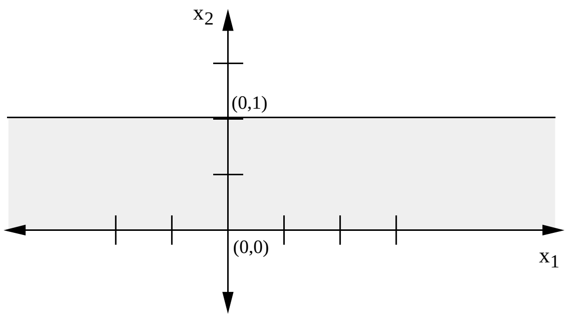


Figure 2: A polyhedron with no vertex.

Case 1 $\exists j$ such that $y_j < 0$

As λ increases, component j decreases until $x + \lambda y$ is no longer feasible.

Choose $\lambda = \min_{\{j: y_j < 0\}}\{x_j / -y_j\} = x_k / -y_k$. This is the largest λ such that $x + \lambda y \geq 0$. Since $Ay = 0$, $A(x + \lambda y) = Ax + \lambda Ay = Ax = b$. So $x + \lambda y \in P$, and moreover $x + \lambda y$ has one more zero component, $(x + \lambda y)_k$, than x.

Replace x by $x + \lambda y$.

Case 2 $y_j \geq 0\ \forall j$

By assumption, $c^T y < 0$ and $x + \lambda y$ is feasible for all $\lambda \geq 0$, since $A(x + \lambda y) = Ax + \lambda Ay = Ax = b$, and $x + \lambda y \geq x \geq 0$. But $c^T(x + \lambda y) = c^T x + \lambda c^T y \to -\infty$ as $\lambda \to \infty$, implying LP is unbounded, a contradiction.

Case 1 can happen at most n times, since x has n components. By induction on the number of non-zero components of x, we obtain a vertex x'.

□

Remark: The theorem was described in terms of the polyhedral set $P = \{x : Ax = b : x \geq 0\}$. Strictly speaking, the theorem is not true for $P = \{x : Ax \geq b\}$. Indeed, such a set P might not have any vertex. For example, consider $P = \{(x_1, x_2) : 0 \leq x_2 \leq 1\}$ (see Figure 2). This polyhedron has no vertex, since for any $x \in P$, we have $x + y$, $x - y \in P$, where $y = (1, 0)$. It can be shown that P has a vertex iff $Rank(A) = n$. Note that, if we transform a program in canonical form into standard form, the non-negativity constraints imply that the resulting matrix A has full column rank, since

$$Rank \begin{bmatrix} A \\ -A \\ I \end{bmatrix} = n.$$

Corollary 2 *If* $\min\{c^T x : Ax = b, x \geq 0\}$ *is finite, There exists an optimal solution,* x^**, which is a vertex.*

Proof:

Suppose not. Take an optimal solution. By Theorem 1 there exists a vertex costing no more and this vertex must be optimal as well. $\square$

Corollary 3 *If* $P = \{x : Ax = b, x \geq 0\} \neq \emptyset$*, then* P *has a vertex.*

Theorem 4 *Let* $P = \{x : Ax = b, x \geq 0\}$*. For* $x \in P$*, let* A_x *be a submatrix of* A *corresponding to* j *s.t.* $x_j > 0$*. Then* x *is a vertex iff* A_x *has linearly independent columns. (i.e.* A_x *has full column rank.)*

Example $A = \begin{bmatrix} 2 & 1 & 3 & 0 \\ 7 & 3 & 2 & 1 \\ 0 & 0 & 0 & 5 \end{bmatrix}$ $x = \begin{bmatrix} 2 \\ 0 \\ 1 \\ 0 \end{bmatrix}$ $A_x = \begin{bmatrix} 2 & 3 \\ 7 & 2 \\ 0 & 0 \end{bmatrix}$, and x is a vertex.

Proof:

Show $\neg i \rightarrow \neg ii$.

Assume x is not a vertex. Then, by definition, $\exists y \neq 0$ s.t. $x + y,\ x - y \in P$. Let A_y be submatrix corresponding to non-zero components of y.

As in the proof of Theorem 1,

$$\left.\begin{array}{ccccc} Ax & + & Ay & = & b \\ Ax & - & Ay & = & b \end{array}\right\} \Rightarrow Ay = 0.$$

Therefore, A_y has dependent columns since $y \neq 0$.

Moreover,

$$\left.\begin{array}{ccccc} x & + & y & \geq & 0 \\ x & - & y & \geq & 0 \end{array}\right\} \Rightarrow y_j = 0 \text{ whenever } x_j = 0.$$

Therefore A_y is a submatrix of A_x. Since A_y is a submatrix of A_x, A_x has linearly dependent columns.

Show $\neg ii \rightarrow \neg i$.

Suppose A_x has linearly dependent columns. Then $\exists y$ s.t. $A_x y = 0,\ y \neq 0$. Extend y to $\mathbb{R}^n$ by adding 0 components. Then $\exists y \in \mathbb{R}^n$ s.t. $Ay = 0,\ y \neq 0$ and $y_j = 0$ wherever $x_j = 0$.

Consider $y' = \lambda y$ for small $\lambda \geq 0$. Claim that $x + y',\ x - y' \in P$, by argument analogous to that in Case 1 of the proof of Theorem 1, above. Hence, x is not a vertex.

$\square$

6 Bases

Let x be a vertex of $P = \{x : Ax = b, x \geq 0\}$. Suppose first that $|\{j : x_j > 0\}| = m$ (where A is $m \times n$). In this case we denote $B = \{j : x_j > 0\}$. Also let $A_B = A_x$; we use this notation not only for A and B, but also for x and for other sets of indices. Then A_B is a square matrix whose columns are linearly independent (by Theorem 4), so it is non-singular. Therefore we can express x as $x_j = 0$ if $j \notin B$, and since $A_B x_B = b$, it follows that $x_B = A_B^{-1} b$. The variables corresponding to B will be called *basic*. The others will be referred to as *nonbasic*. The set of indices corresponding to nonbasic variables is denoted by $N = \{1, \ldots, n\} - B$. Thus, we can write the above as $x_B = A_B^{-1} b$ and $x_N = 0$.

Without loss of generality we will assume that A has full row rank, $\text{rank}(A) = m$. Otherwise either there is a redundant constraint in the system $Ax = b$ (and we can remove it), or the system has no solution at all.

If $|\{j : x_j > 0\}| < m$, we can augment A_x with additional linearly independent columns, until it is an $m \times m$ submatrix of A of full rank, which we will denote A_B. In other words, although there may be less than m positive components in x, it is convenient to always have a *basis* B such that $|B| = m$ and A_B is non-singular. This enables us to always express x as we did before, $x_N = 0$, $x_B = A_B^{-1} b$.

Summary x is a vertex of P iff there is $B \subseteq \{1, \ldots, n\}$ such that $|B| = m$ and

1. $x_N = 0$ for $N = \{1, \ldots, n\} - B$
2. A_B is non-singular
3. $x_B = A_B^{-1} b \geq 0$

In this case we say that x is a *basic feasible solution*. Note that a vertex can have several basic feasible solution corresponding to it (by augmenting $\{j : x_j > 0\}$ in different ways). A basis might not lead to any basic feasible solution since $A_B^{-1} b$ is not necessarily nonnegative.

Example:

$$
\begin{aligned}
x_1 + x_2 + x_3 &= 5 \\
2x_1 - x_2 + 2x_3 &= 1 \\
x_1, x_2, x_3 &\geq 0
\end{aligned}
$$

We can select as a basis $B = \{1, 2\}$. Thus, $N = \{3\}$ and

$$\begin{aligned}
A_B &= \begin{pmatrix} 1 & 1 \\ 2 & -1 \end{pmatrix} \\
A_B^{-1} &= \begin{pmatrix} \frac{1}{3} & \frac{1}{3} \\ \frac{2}{3} & \frac{-1}{3} \end{pmatrix} \\
A_B^{-1}b &= \begin{pmatrix} 2 \\ 3 \end{pmatrix} \\
x &= (2, 3, 0)
\end{aligned}$$

Remark. A crude upper bound on the number of vertices of P is $\binom{n}{m}$. This number is exponential (it is upper bounded by n^m). We can come up with a tighter approximation of $\binom{n-\frac{m}{2}}{\frac{m}{2}}$, though this is still exponential. The reason why the number is much smaller is that most basic solutions to the system $Ax = b$ (which we counted) are not feasible, that is, they do not satisfy $x \geq 0$.

7 The Simplex Method

The Simplex algorithm [Dantzig,1947] [2] solves linear programming problems by focusing on basic feasible solutions. The basic idea is to start from some vertex v and look at the adjacent vertices. If an improvement in cost is possible by moving to one of the adjacent vertices, then we do so. Thus, we will start with a bfs corresponding to a basis B and, at each iteration, try to improve the cost of the solution by removing one variable from the basis and replacing it by another.

We begin the Simplex algorithm by first rewriting our LP in the form:

$$\begin{array}{ll}
\min & c_B x_B + c_N x_N \\
\text{s.t.} & A_B x_B + A_N x_N = b \\
& x_B, x_N \geq 0
\end{array}$$

Here B is the basis corresponding to the bfs we are starting from. Note that, for any solution x, $x_B = A_B^{-1}b - A_B^{-1}A_N x_N$ and that its total cost, $c^T x$ can be specified as follows:

$$\begin{aligned}
c^T x &= c_B x_B + c_N x_N \\
&= c_B(A_B^{-1}b - A_B^{-1}A_N x_N) + c_N x_N \\
&= c_B A_B^{-1}b + (c_N - c_B A_B^{-1}A_N)x_N
\end{aligned}$$

We denote the *reduced* cost of the non-basic variables by $\tilde{c}_N$, $\tilde{c}_N = c_N - c_B A_B^{-1}A_N$, i.e. the quantity which is the coefficient of x_N above. If there is a $j \in N$ such that

$\tilde{c}_j < 0$, then by increasing x_j (up from zero) we will decrease the cost (the value of the objective function). Of course x_B depends on x_N, and we can increase x_j only as long as all the components of x_B remain positive.

So in a step of the Simplex method, we find a $j \in N$ such that $\tilde{c}_j < 0$, and increase it as much as possible while keeping $x_B \geq 0$. It is not possible any more to increase x_j, when one of the components of x_B is zero. What happened is that a non-basic variable is now positive and we include it in the basis, and one variable which was basic is now zero, so we remove it from the basis.

If, on the other hand, there is no $j \in N$ such that $\tilde{c}_j < 0$, then we stop, and the current basic feasible solution is an optimal solution. This follows from the new expression for $c^T x$ since x_N is nonnegative.

Remarks:

1. Note that some of the basic variables may be zero to begin with, and in this case it is possible that we cannot increase x_j at all. In this case we can replace say j by k in the basis, but without moving from the vertex corresponding to the basis. In the next step we might replace k by j, and be stuck in a loop. Thus, we need to specify a "pivoting rule" to determine which index should enter the basis, and which index should be removed from the basis.

2. While many pivoting rules (including those that are used in practice) can lead to infinite loops, there is a pivoting rule which will not (known as the minimal index rule - choose the minimal j and k possible [Bland, 1977]). This fact was discovered by Bland in 1977. There are other methods of "breaking ties" which eliminate infinite loops.

3. There is no known pivoting rule for which the number of pivots in the worst case is better than exponential.

4. The question of the complexity of the Simplex algorithm and the last remark leads to the question of what is the length of the shortest path between two vertices of a convex polyhedron, where the path is along edges, and the length of the path in measured in terms of the number of vertices visited.

   Hirsch Conjecture: For m hyperplanes in d dimensions the length of the shortest path between any two vertices of the arrangement is at most $m - d$.

   This is a very open question — there is not even a polynomial bound proven on this length.

   On the other hand, one should note that even if the Hirsch Conjecture is true, it doesn't say much about the Simplex Algorithm, because Simplex generates paths which are monotone with respect to the objective function, whereas the shortest path need not be monotone.

Recently, Kalai (and others) has considered a randomized pivoting rule. The idea is to randomly permute the index columns of A and to apply the Simplex method, always choosing the smallest j possible. In this way, it is possible to show a subexponential bound on the expected number of pivots. This leads to a subexponential bound for the diameter of any convex polytope defined by m hyperplanes in a d dimension space.

The question of the existence of a polynomial pivoting scheme is still open though. We will see later a completely different algorithm which *is* polynomial, although not strongly polynomial (the existence of a strongly polynomial algorithm for linear programming is also open). That algorithm will not move from one vertex of the feasible domain to another like the Simplex, but will confine its interest to points in the interior of the feasible domain.

A visualization of the geometry of the Simplex algorithm can be obtained from considering the algorithm in 3 dimensions (see Figure 3). For a problem in the form $\min\{c^T x : Ax \leq b\}$ the feasible domain is a polyhedron in $\mathbb{R}^3$, and the algorithm moves from vertex to vertex in each step (or does not move at all).

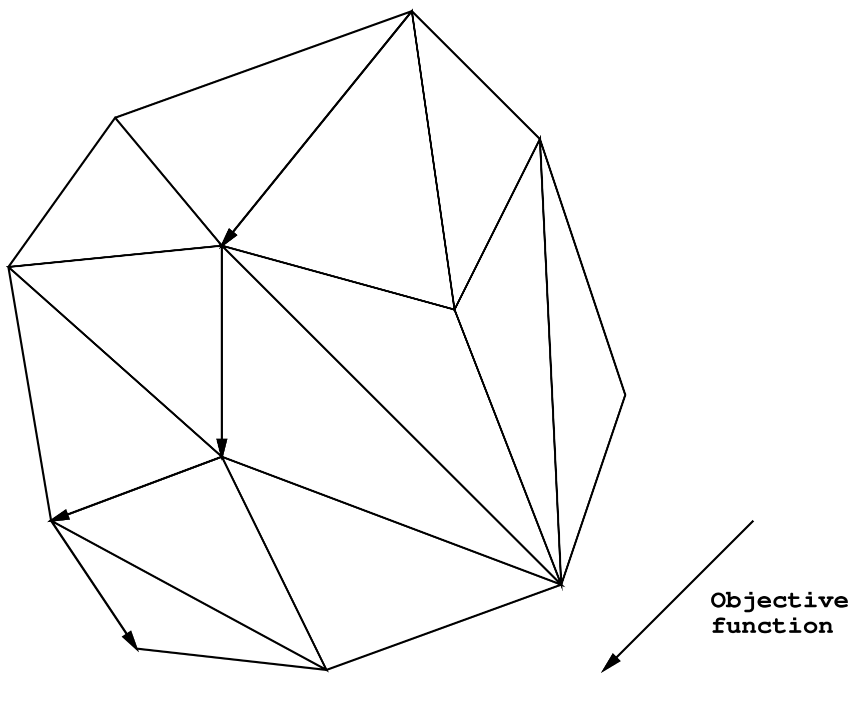


Figure 3: Traversing the vertices of a convex body (here a polyhedron in $\mathbb{R}^3$).

8 When is a Linear Program Feasible ?

We now turn to another question which will lead us to important properties of linear programming. Let us begin with some examples.

We consider linear programs of the form $Ax = b$, $x \geq 0$. As the objective function has no effect on the feasibility of the program, we ignore it.

We first restrict our attention to systems of equations (i.e. we neglect the non-negativity constraints).

Example: Consider the system of equations:

$$\begin{array}{rcrcrcl} x_1 & + & x_2 & + & x_3 & = & 6 \\ 2x_1 & + & 3x_2 & + & x_3 & = & 8 \\ 2x_1 & + & x_2 & + & 3x_3 & = & 0 \end{array}$$

and the linear combination

$$\begin{array}{rcrcrcrcl} -4 & \times & x_1 & + & x_2 & + & x_3 & = & 6 \\ 1 & \times & 2x_1 & + & 3x_2 & + & x_3 & = & 8 \\ 1 & \times & 2x_1 & + & x_2 & + & 3x_3 & = & 0 \end{array}$$

The linear combination results in the equation

$$0x_1 + 0x_2 + 0x_3 = -16$$

which means of course that the system of equations has no feasible solution.

In fact, an elementary theorem of linear algebra says that if a system has no solution, there is always a vector y such as in our example ($y = (-4, 1, 1)$) which proves that the system has no solution.

Theorem 5 *Exactly one of the following is true for the system $Ax = b$:*

1. *There is x such that $Ax = b$.*
2. *There is y such that $A^T y = 0$ but $y^T b = 1$.*

This is not quite enough for our purposes, because a system can be feasible, but still have no non-negative solutions $x \geq 0$. Fortunately, the following lemma establishes the equivalent results for our system $Ax = b, x \geq 0$.

Theorem 6 (Farkas' Lemma) *Exactly one of the following is true for the system $Ax = b$, $x \geq 0$:*

1. *There is x such that $Ax = b$, $x \geq 0$.*
2. *There is y such that $A^T y \geq 0$ but $b^T y < 0$.*

Proof:

We will first show that the two conditions cannot happen together, and then than at least one of them must happen.

Suppose we do have both x and y as in the statement of the theorem.

$$Ax = b \Longrightarrow y^T Ax = y^T b \Longrightarrow x^T A^T y = y^T b$$

but this is a contradiction, because $y^T b < 0$, and since $x \geq 0$ and $A^T y \geq 0$, so $x^T A^T y \geq 0$.

The other direction is less trivial, and usually shown using properties of the Simplex algorithm, mainly duality. We will use another tool, and later use Farkas' Lemma to prove properties about duality in linear programming. The tool we shall use is the Projection theorem, which we state without proof:

Theorem 7 (Projection Theorem) *Let K be a closed convex (see Figure 4) non-empty set in $\mathbb{R}^n$, and let b be any point in $\mathbb{R}^n$. The projection of b onto K is a point $p \in K$ that minimizes the Euclidean distance $\|b - p\|$. Then p has the property that for all $z \in K$, $(z - p)^T(b - p) \leq 0$ (see Figure 5) non-empty set.*

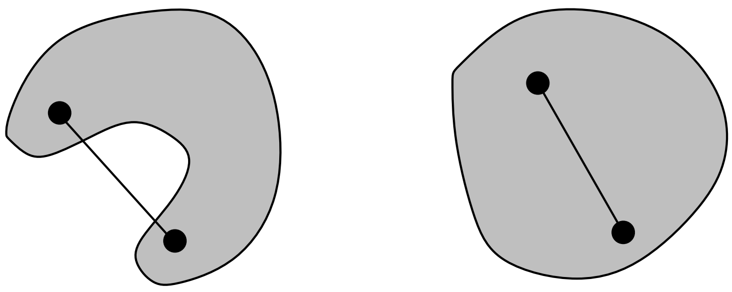


Figure 4: Convex and non-convex sets in $\mathbb{R}^2$.

We are now ready to prove the other direction of Farkas' Lemma. Assume that there is no x such that $Ax = b$, $x \geq 0$; we will show that there is y such that $A^T y \geq 0$ but $y^T b < 0$.

Let $K = \{Ax : x \geq 0\} \subseteq \mathbb{R}^m$ (A is an $m \times n$ matrix). K is a cone in $\mathbb{R}^m$ and it is convex, non-empty and closed. According to our assumption, $Ax = b$, $x \geq 0$ has no solution, so b does not belong to K. Let p be the projection of b onto K.

Since $p \in K$, there is a $w \geq 0$ such that $Aw = p$. According to the Projection Theorem, for all $z \in K$, $(z-p)^T(b-p) \leq 0$ That is, for all $x \geq 0$ $(Ax-p)^T(b-p) \leq 0$

We define $y = p - b$, which implies $(Ax-p)^T y \geq 0$. Since $Aw = p$, $(Ax - Aw)^T y \geq 0$. $(x - w)^T(A^T y) \geq 0$ for all $x \geq 0$ (remember that w was fixed by choosing b).

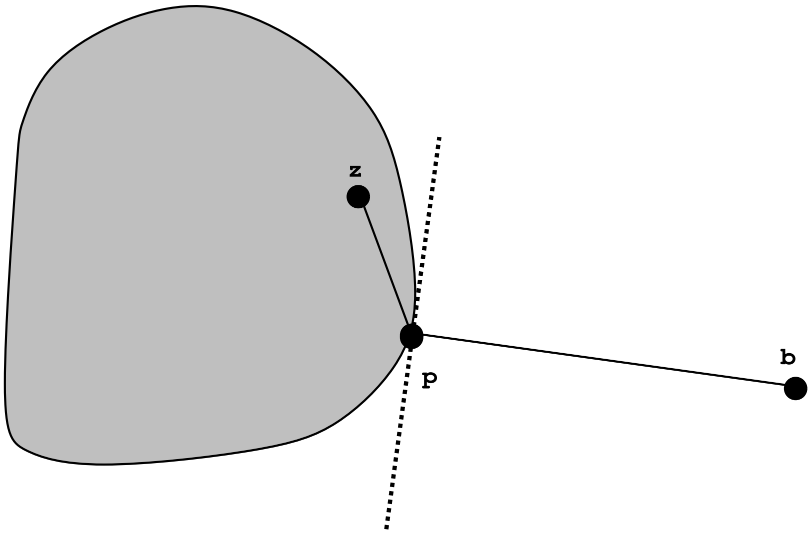


Figure 5: The Projection Theorem.

Set $x = w + \begin{pmatrix} 0 \\ 0 \\ \vdots \\ 1 \\ \vdots \\ 0 \end{pmatrix}$ (w plus a unit vector with a 1 in the i-th row). Note that x is non-negative, because $w \geq 0$.

This will extract the i-th column of A, so we conclude that the i-th component of $A^T y$ is non-negative $(A^T y)_i \geq 0$, and since this is true for all i, $A^T y \geq 0$.

Now it only remains to show that $y^T b < 0$.

$y^t b = (p-y)^T y = p^T y - y^T y$ Since $(Ax-p)^T y \geq 0$ for all $x \geq 0$, taking x to be zero shows that $p^T y \leq 0$. Since $b \notin K$, $y = p - b \neq 0$, so $y^T y > 0$. So $y^T b = p^T y - y^T y < 0$. $\square$

Using a very similar proof one can show the same for the canonical form:

Theorem 8 *Exactly one of the following is true for the system $Ax \leq b$:*

1. *There is x such that $Ax \leq b$.*

2. *There is $y \geq 0$ such that $A^T y = 0$ but $y^T b < 0$.*

The intuition behind the precise form for 2. in the previous theorem lies in the proof that both cannot happen. The contradiction $0 = 0x = (y^T A)x = y^T(Ax) = y^T b < 0$ is obtained if $A^T y = 0$ and $y^T b < 0$.

9 Duality

Duality is the most important concept in linear programming. Duality allows to provide a *proof* of optimality. This is not only important algorithmically but also it leads to beautiful combinatorial statements. For example, consider the statement

> In a graph, the smallest number of edges in a path between two specified vertices s and t is equal to the maximum number of $s-t$ cuts (i.e. subsets of edges whose removal disconnects s and t).

This result is a direct consequence of duality for linear programming.

Duality can be motivated by the problem of trying to find lower bounds on the value of the optimal solution to a linear programming problem (if the problem is a maximization problem, then we would like to find upper bounds). We consider problems in standard form:

$$\begin{array}{ll} \min & c^T x \\ \text{s.t.} & Ax = b \\ & x \geq 0 \end{array}$$

Suppose we wanted to obtain the best possible upper bound on the cost function. By multiplying each equation $A_m x = b_m$ by some number y_m and summing up the resulting equations, we obtain that $y^T A x = b^T y$. if we impose that the coefficient of x_j in the resulting inequality is less or equal to c_j then $b^T y$ must be a lower bound on the optimal value since x_j is constrained to be nonnegative. To get the best possible lower bound, we want to solve the following problem:

$$\begin{array}{ll} \max & b^T y \\ \text{s.t.} & A^T y \leq c \end{array}$$

This is another linear program. We call this one the *dual* of the original one, called the *primal*. As we just argued, solving this dual LP will give us a lower bound on the optimum value of the primal problem. *Weak duality* says precisely this: if we denote the optimum value of the primal by z, $z = \min c^T x$, and the optimum value of the dual by w, then $w \leq z$. We will use Farkas' lemma to prove *strong duality* which says that these quantities are in fact equal. We will also see that, in general, the dual of the dual is the problem.

Example:

$$\begin{array}{rcccccccc} z = \min & x_1 & + & 2x_2 & + & 4x_3 & & \\ & x_1 & + & x_2 & + & 2x_3 & = & 5 \\ & 2x_1 & + & x_2 & + & 3x_3 & = & 8 \end{array}$$

The first equality gives a lower bound of 5 on the optimum value z, since $x_1 + 2x_2 + 4x_3 \geq x_1 + x_2 + 2x_3 = 5$ because of nonnegativity of the x_i. We can get an even better lower bound by taking 3 times the first equality minus the second one. This gives $x_1 + 2x_2 + 3x_3 = 7 \leq x_1 + 2x_2 + 4x_3$, implying a lower bound of 7 on z. For $x = \begin{pmatrix} 3 \\ 2 \\ 0 \end{pmatrix}$, the objective function is precisely 7, implying optimality. The mechanism of generating lower bounds is formalized by the dual linear program:

$$\begin{array}{lrcrcl} \max & 5y_1 & + & 8y_2 & & \\ & y_1 & + & 2y_2 & \leq & 1 \\ & y_1 & + & y_2 & \leq & 2 \\ & 2y_1 & + & 3y_2 & \leq & 4 \end{array}$$

y_1 represents the multiplier for the first constraint and y_2 the multiplier for the second constraint, This LP's objective function also achieves a maximum value of 7 at $y = \begin{pmatrix} 3 \\ -1 \end{pmatrix}$.

We now formalize the notion of duality. Let P and D be the following pair of dual linear programs:

$$\begin{array}{ll} (P) & z = \min\{c^T x : Ax = b, x \geq 0\} \\ (D) & w = \max\{b^T y : A^T y \leq c\}. \end{array}$$

(P) is called the *primal* linear program and (D) the *dual* linear program.

In the proof below, we show that the dual of the dual is the primal. In other words, if one formulates (D) as a linear program in standard form (i.e. in the same form as (P)), its dual $D(D)$ can be seen to be equivalent to the original primal (P). In any statement, we may thus replace the roles of primal and dual without affecting the statement.

Proof:

The dual problem D is equivalent to $\min\{-b^T y : A^T y + Is = c, s \geq 0\}$. Changing forms we get $\min\{-b^T y^+ + b^T y^- : A^T y^+ - A^T y^- + Is = c, \text{ and } y^+, y^-, s \geq 0\}$. Taking the dual of this we obtain: $\max\{-c^T x : A(-x) \leq -b, -A(-x) \leq b, I(-x) \leq 0\}$. But this is the same as $\min\{c^T x : Ax = b, x \geq 0\}$ and we are done. □

We have the following results relating w and z.

Lemma 9 *(Weak Duality)* $z \geq w$.

Proof:

Suppose x is primal feasible and y is dual feasible. Then, $c^T x \geq y^T Ax = y^T b$, thus $z = \min\{c^T x : Ax = b, x \geq 0\} \geq \max\{b^T y : A^T y \leq c\} = w$. □

From the preceding lemma we conclude that the following cases are *not* possible (these are dual statements):

1. P is feasible and unbounded and D feasible.

2. P is feasible and D is feasible and unbounded.

We should point out however that both the primal and the dual might be infeasible.

To prove a stronger version of the *weak duality lemma,* let's recall the following corollary of *Farkas' Lemma* (Theorem 8):

Corollary 10 *Exactly one of the following is true:*

1. $\exists x' : A'x' \leq b'$.
2. $\exists y' \geq 0 : (A')^T y' = 0$ *and* $(b')^T y' < 0$.

Theorem 11 *(Strong Duality) If P or D is feasible then $z = w$.*

Proof:

We only need to show that $z \leq w$. Assume without loss of generality (by duality) that P is feasible. If P is unbounded, then by *Weak Duality*, we have that $z = w = -\infty$. Suppose P is bounded, and let x^* be an optimal solution, *i.e.* $Ax^* = b$, $x^* \geq 0$ and $c^T x^* = z$. We claim that $\exists y$ s.t. $A^T y \leq c$ and $b^T y \geq z$. If so we are done. Suppose no such y exists. Then, by the preceding corollary, with $A' = \begin{pmatrix} A^T \\ -b^T \end{pmatrix}$, $b' = \begin{pmatrix} c \\ -z \end{pmatrix}$, $x' = y$, $y' = \begin{pmatrix} x \\ \lambda \end{pmatrix}$, $\exists x \geq 0$, $\lambda \geq 0$ such that

$$\begin{aligned} & Ax = \lambda b \\ \text{and} \quad & c^T x < \lambda z. \end{aligned}$$

We have two cases

- **Case 1:** $\lambda \neq 0$. Since we can normalize by λ we can assume that $\lambda = 1$. This means that $\exists x \geq 0$ such that $Ax = b$ and $c^T x < z$. But this is a contradiction with the optimality of x^*.
- **Case 2:** $\lambda = 0$. This means that $\exists x \geq 0$ such that $Ax = 0$ and $c^T x < 0$. If this is the case then $\forall \mu \geq 0$, $x^* + \mu x$ is feasible for P and its cost is $c^T(x^* + \mu x) = c^T x^* + \mu(c^T x) < z$, which is a contradiction.

□

9.1 Rules for Taking Dual Problems

If P is a minimization problem then D is a maximization problem. If P is a maximization problem then D is a minimization problem. In general, using the rules for transforming a linear program into standard form, we have that the dual of (P):

$z = \min\ c_1^T x_1 + c_2^T x_2 + c_3^T x_3$

s.t.

$$\begin{aligned}
A_{11}x_1 + A_{12}x_2 + A_{13}x_3 &= b_1 \\
A_{21}x_1 + A_{22}x_2 + A_{23}x_3 &\geq b_2 \\
A_{31}x_1 + A_{32}x_2 + A_{33}x_3 &\leq b_3 \\
x_1 \geq 0\ ,\ x_2 \leq 0\ ,\ x_3 \text{ UIS} &
\end{aligned}$$

(where UIS means "unrestricted in sign" to emphasize that no constraint is on the variable) is (D)

$w = \max\ b_1^T y_1 + b_2^T y_2 + b_3^T y_3$

s.t.

$$\begin{aligned}
A_{11}^T y_1 + A_{21}^T y_2 + A_{31}^T y_3 &\leq c_1 \\
A_{12}^T y_1 + A_{22}^T y_2 + A_{32}^T y_3 &\geq c_2 \\
A_{13}^T y_1 + A_{23}^T y_2 + A_{33}^T y_3 &= c_3 \\
y_1 \text{ UIS}\ ,\ y_2 \geq 0\ ,\ y_3 \leq 0 &
\end{aligned}$$

10 Complementary Slackness

Let P and D be

$$\begin{aligned}
(P) \quad & z = \min\{c^T x : Ax = b, x \geq 0\} \\
(D) \quad & w = \max\{b^T y : A^T y \leq c\},
\end{aligned}$$

and let x be feasible in P, and y be fesible in D. Then, by weak duality, we know that $c^T x \geq b^T y$. We call the difference $c^T x - b^T y$ the *duality gap*. Then we have that the duality gap is zero iff x is optimal in P, and y is optimal in D. That is, the duality gap can serve as a good measure of how close a feasible x and y are to the optimal solutions for P and D. The duality gap will be used in the description of the interior point method to monitor the progress towards optimality.

It is convenient to write the dual of a linear program as

$$(D) \quad w = \max\{b^T y : A^T y + s = c \text{ for some } s \geq 0\}$$

Then we can write the duality gap as follows:

$$\begin{aligned}
c^T x - b^T y &= c^T x - x^T A^T y \\
&= x^T (c - A^T y) \\
&= x^T s
\end{aligned}$$

since $A^T y + s = c$.

The following theorem allows to check optimality of a primal and/or a dual solution.

Theorem 12 *(Complementary Slackness)*
Let x^*, (y^*, s^*) be feasible for (P), (D) respectively. The following are equivalent:

1. *x^* is an optimal solution to (P) and (y^*, s^*) is an optimal solution to (D).*
2. *$(s^*)^T x^* = 0$.*
3. *$x_j^* s_j^* = 0$, $\forall\, j = 1, \dots, n$.*
4. *If $s_j^* > 0$ then $x_j^* = 0$.*

Proof:
Suppose (1) holds, then, by strong duality, $c^T x^* = b^T y^*$. Since $c = A^T y^* + s^*$ and $Ax^* = b$, we get that $(y^*)^T A x^* + (s^*)^T x^* = (x^*)^T A^T y^*$, and thus, $(s^*)^T x^* = 0$ (i.e (2) holds). It follows, since x_j^*, $s_j^* \geq 0$, that $x_j^* s_j^* = 0$, $\forall\, j = 1, \dots, n$ (i.e. (3) holds). Hence, if $s_j^* > 0$ then $x_j^* = 0$, $\forall\, j = 1, \dots, n$ (i.e. (4) holds). The converse also holds, and thus the proof is complete. $\square$

In the example of section 9, the complementary slackness equations corresponding to the primal solution $x = (3, 2, 0)^T$ would be:

$$\begin{aligned} y_1 + 2y_2 &= 1 \\ y_1 + y_2 &= 2 \end{aligned}$$

Note that this implies that $y_1 = 3$ and $y_2 = -1$. Since this solution satisfies the other constraint of the dual, y is dual feasible, proving that x is an optimum solution to the primal (and therefore y is an optimum solution to the dual).

11 Size of a Linear Program

11.1 Size of the Input

If we want to solve a Linear Program in polynomial time, we need to know what would that mean, i.e. what would the size of the input be. To this end we introduce two notions of the size of the input with respect to which the algorithm we present will run in polynomial time. The first measure of the input size will be the *size* of a LP, but we will introduce a new measure L of a LP that will be easier to work with. Moreover, we have that $L \leq \text{size}(LP)$, so that any algorithm running in time polynomial in L will also run in time polynomial in *size(LP)*.

Let's consider the linear program of the form:

$$\begin{aligned} & \min c^T x \\ s.t.\ & \\ & Ax = b \\ & x \geq 0 \end{aligned}$$

where we are given as inputs the coefficients of A (an $m \times n$ matrix), b (an $m \times 1$ vector), and c (an $n \times 1$ vector), whith rational entries.

We can further assume, without loss of generality, that the given coefficients are all integers, since any LP with rational coefficients can be easily transformed into an equivalent one with integer coefficients (just multiply everything by l.c.d.). In the rest of these notes, we assume that A, b, c have integer coefficients.

For any integer n, we define its size as follows:

$$size(n) \triangleq 1 + \lceil \log_2(|n| + 1) \rceil$$

where the first 1 stands for the fact that we need one bit to store the sign of n, $size(n)$ represents the number of bits needed to encode n in binary. Analogously, we define the size of a $p \times 1$ vector d, and of a $p \times l$ matrix M as follows:

$$size(v) \triangleq \textstyle\sum_{i=1}^{p} size(v_i)$$
$$size(M) \triangleq \textstyle\sum_{i=1}^{p} \sum_{j=1}^{l} size(m_{ij})$$

We are then ready to talk about the size of a *LP*.

Definition 6 (Size of a linear program)

$$\text{size(LP)} \triangleq \text{size}(A) + \text{size}(b) + \text{size}(c).$$

A more convenient definition of the size of a linear program is given next.

Definition 7

$$L \triangleq \text{size}(det_{max}) + \text{size}(b_{max}) + \text{size}(c_{max}) + m + n$$

where

$$\begin{aligned} det_{max} &\triangleq \max_{A'}(|\det(A')|) \\ b_{max} &\triangleq \max_{i}(|b_i|) \\ c_{max} &\triangleq \max_{j}(|c_j|) \end{aligned}$$

and A' is any square submatrix of A.

Proposition 13 $L < \text{size(LP)}, \; \forall A, b, c$.

Before proving this result, we first need the following lemma:

Lemma 14 *1. If $n \in \mathbb{Z}$ then $|n| \leq 2^{size(n)-1} - 1$.*

2. If $v \in \mathbb{Z}^n$ then $\|v\| \leq \|v\|_1 \leq 2^{size(v)-n} - 1$.

3. If $A \in \mathbb{Z}^{n\times n}$ then $|det(A)| \leq 2^{size(A)-n^2} - 1$.

Proof:

1. By definition.

2. $1+\|v\| \leq 1+\|v\|_1 = 1+\sum_{i=1}^{n}|v_i| \leq \prod_{i=1}^{n}(1+|v_i|) \leq \prod_{i=1}^{n} 2^{size(v_i)-1} = 2^{size(v)-n}$ where we have used 1.

3. Let $a_1, \ldots, a_n$ be the columns of A. Since $|det(A)|$ represents the volume of the parallelepiped spanned by $a_1, \ldots, a_n$, we have

$$|det(A)| \leq \prod_{i=1}^{n}\|a_i\|.$$

Hence, by 2,

$$1+|det(A)| \leq 1+\prod_{i=1}^{n}\|a_i\| \leq \prod_{i=1}^{n}(1+\|a_i\|) \leq \prod_{i=1}^{n} 2^{size(a_i)-n} = 2^{size(A)-n^2}.$$

□

We now prove Proposition 13.

Proof:

If B is a square submatrix of A then, by definition, $size(B) \leq size(A)$. Moreover, by lemma 14, $1+|det(B)| \leq 2^{size(B)-1}$. Hence,

$$\lceil \log(1+|det(B)|)\rceil \leq size(B)-1 < size(B) \leq size(A). \tag{1}$$

Let $v \in \mathbb{Z}^p$. Then $size(v) \geq size(\max_j |v_j|)+p-1 = \lceil \log(1+\max_j |v_j|)\rceil + p$. Hence,

$$size(b)+size(c) \geq \lceil \log(1+\max_j |c_j|)\rceil + \lceil \log(1+\max_i |b_i|)\rceil + m+n. \tag{2}$$

Combining equations (1) and (2), we obtain the desired result. □

Remark 1 *$det_{max} * b_{max} * c_{max} * 2^{m+n} < 2^L$, since for any integer n, $2^{\text{size}(n)} > |n|$.*

In what follows we will work with L as the size of the input to our algorithm.

11.2 Size of the Output

In order to even hope to solve a linear program in polynomial time, we better make sure that the solution is representable in size polynomial in L. We know already that if the LP is feasible, there is at least one vertex which is an optimal solution. Thus, when finding an optimal solution to the LP, it makes sense to restrict our attention to vertices only. The following theorem makes sure that vertices have a compact representation.

Theorem 15 *Let x be a vertex of the polyhedron defined by $Ax = b, x \geq 0$. Then,*

$$x^T = \left(\frac{p_1}{q}\ \frac{p_2}{q} \cdots \frac{p_n}{q} \right),$$
$$\text{where } p_i\ (i = 1, \ldots, n),\ q \in \mathbb{N},$$

and

$$0 \leq p_i < 2^L$$
$$1 \leq q < 2^L.$$

Proof:

Since x is a basic feasible solution, $\exists$ a basis B such that $x_B = A_B^{-1}b$ and $x_N = 0$. Thus, we can set $p_j = 0$, $\forall\ j \in N$, and focus our attention on the x_j's such that $j \in B$. We know by linear algebra that

$$x_B = A_B^{-1}b = \frac{1}{det(A_B)} cof(A_B)b$$

where $cof(A_B)$ is the cofactor matrix of A_B. Every entry of A_B consists of a determinant of some submatrix of A. Let $q = |det(A_B)|$, then q is an integer since A_B has integer components, $q \geq 1$ since A_B is invertible, and $q \leq det_{\max} < 2^L$. Finally, note that $p_B = qx_B = |cof(A_B)b|$, thus $p_i \leq \sum_{j=1}^{m} |cof(A_B)_{ij}||b_j| \leq m\ det_{\max}\ b_{\max} < 2^L$.

$\square$

12 Complexity of linear programming

In this section, we show that linear programming is in NP$\cap$ co-NP. This will follow from duality and the estimates on the size of any vertex given in the previous section. Let us define the following decision problem:

Definition 8 ($\mathcal{LP}$)

Input: *Integral A, b, c,* and a rational number λ,

Question: Is $\min\{c^T x : Ax = b,\ x \geq 0\} \leq \lambda$?

Theorem 16 $\mathcal{LP} \in \text{NP} \cap \text{co-NP}$

Proof:

First, we prove that $\mathcal{LP} \in \text{NP}$.

If the linear program is feasible and bounded, the "certificate" for verification of instances for which $\min\{c^T x : Ax = b, x \geq 0\} \leq \lambda$ is a vertex x' of $\{Ax = b, x \geq 0\}$ s.t. $c^T x' \leq \lambda$. This vertex x' always exists since by assumption the minimum is finite. Given x', it is easy to check in polynomial time whether $Ax' = b$ and $x' \geq 0$. We also need to show that the size of such a certificate is polynomially bounded by the size of the input. This was shown in section 11.2.

If the linear program is feasible and unbounded, then, by strong duality, the dual is infeasible. Using Farkas' lemma on the dual, we obtain the existence of $\tilde{x}$: $A\tilde{x} = 0$, $\tilde{x} \geq 0$ and $c^T\tilde{x} = -1 < 0$. Our certificate in this case consists of both a vertex of $\{Ax = b,\ x \geq 0\}$ (to show feasiblity) and a vertex of $\{Ax = 0,\ x \geq 0,\ c^T x = -1\}$ (to show unboundedness if feasible). By choosing a vertex x' of $\{Ax = 0,\ x \geq 0,\ c^T x = -1\}$, we insure that x' has polynomial size (again, see Section 11.2).

This proves that $\mathcal{LP} \in \text{NP}$. (Notice that when the linear program is infeasible, the answer to $\mathcal{LP}$ is "no", but we are not responsible to offer such an answer in order to show $\mathcal{LP} \in \text{NP}$).

Secondly, we show that $\mathcal{LP} \in \text{co-NP}$, i.e. $\overline{\mathcal{LP}} \in \text{NP}$, where $\overline{\mathcal{LP}}$ is defined as:

Input: A, b, c, and a rational number λ,

Question: Is $\min\{c^T x : Ax = b,\ x \geq 0\} > \lambda$?

If $\{x : Ax = b,\ x \geq 0\}$ is nonempty, we can use strong duality to show that $\overline{\mathcal{LP}}$ is indeed equivalent to:

Input: A, b, c, and a rational number λ,

Question: Is $\max\{b^T y : A^T y \leq c\} > \lambda$?

which is also in NP, for the same reason as $\mathcal{LP}$ is.

If the primal is infeasible, by Farkas' lemma we know the existence of a y s.t. $A^T y \geq 0$ and $b^T y = -1 < 0$. This completes the proof of the theorem. $\square$

13 Solving a Liner Program in Polynomial Time

The first polynomial-time algorithm for linear programming is the so-called *ellipsoid algorithm* which was proposed by Khachian in 1979 [6]. The ellipsoid algorithm was in fact first developed for convex programming (of which linear programming is a special case) in a series of papers by the russian mathematicians A.Ju. Levin and, D.B. Judin and A.S. Nemirovskii, and is related to work of N.Z. Shor. Though of polynomial running time, the algorithm is impractical for linear programming. Nevertheless it has extensive theoretical applications in combinatorial optimization. For example, the stable set problem on the so-called perfect graphs can be solved in polynomial time using the ellipsoid algorithm. This is however a non-trivial non-combinatorial algorithm.

In 1984, Karmarkar presented another polynomial-time algorithm for linear programming. His algorithm avoids the combinatorial complexity (inherent in the simplex algorithm) of the vertices, edges and faces of the polyhedron by staying well inside the polyhedron (see Figure 13). His algorithm lead to many other algorithms for linear programming based on similar ideas. These algorithms are known as *interior point methods*.

Figure 6: Exploring the interior of a convex body.

It still remains an open question whether there exists a strongly polynomial algorithm for linear programming, i.e. an algorithm whose running time depends on m and n and not on the size of any of the entries of A, b or c.

In the rest of these notes, we discuss an interior-point method for linear programming and show its polynomiality.

High-level description of an interior-point algorithm:

1. If x (current solution) is close to the boundary, then map the polyhedron onto another one s.t. x is well in the interior of the new polyhedron (see Figure 7).

2. Make a step in the transformed space.

3. Repeat (a) and(b) until we are close enough to an optimal solution.

Before we give description of the algorithm we give a theorem, the corollary of which will be a key tool used in determinig when we have reached an optimal solution.

Theorem 17 *Let x_1, x_2 be vertices of* $\begin{array}{l} Ax = b, \\ x \geq 0. \end{array}$

If $c^T x_1 \neq c^T x_2$ *then* $|c^T x_1 - c^T x_2| > 2^{-2L}$.

Proof:

By Theorem 15, $\exists\ q_i,\ q_2$, such that $1 \leq q_1, q_2 < 2^L$, and $q_1 x_1, q_2 x_2 \in \mathbb{N}^n$. Furthermore,

$$\begin{aligned}
|c^T x_1 - c^T x_2| &= \left| \frac{q_1 c^T x_1}{q_1} - \frac{q_2 c^T x_2}{q_2} \right| \\
&= \left| \frac{q_1 q_2 (c^T x_1 - c^T x_2)}{q_1 q_2} \right| \\
&\geq \frac{1}{q_1 q_2} \qquad \text{since } c^T x_1 - c^T x_2 \neq 0,\ q_1, q_2 \geq 1 \\
&> \frac{1}{2^L 2^L} = 2^{-2L} \quad \text{since } q_1, q_2 < 2^L.
\end{aligned}$$

□

Corollary 18 *Assume* $z = \min\{c^T x : \underbrace{Ax = b, x \geq 0}_{\textit{polyhedron } P}\}$.

Assume x is feasible to P, and such that $c^T x \leq z + 2^{-2L}$.

Then, any vertex x' such that $c^T x' \leq c^T x$ is an optimal solution of the LP.

Proof:

Suppose x' is not optimal. Then, $\exists x^*$, an optimal vertex, such that $c^T x^* = z$.
Since x' is not optimal, $c^T x' \neq c^T x^*$, and by Theorem 17

$$\begin{array}{rrcll}
\Rightarrow & c^T x' - c^T x^* & > & 2^{-2L} & \\
\Rightarrow & c^T x' & > & c^T x^* + 2^{-2L} & \\
& & = & Z + 2^{-2L} & \\
& & \geq & c^T x & \text{by definition of } x \\
& & \geq & c^T x' & \text{by definition of } x' \\
\Rightarrow & c^T x' & > & c^T x', &
\end{array}$$

a contradiction. □

What this corollary tells us is that we do not need to be very precise when choosing an optimal vertex. More precisely we only need to compute the objective function with error less than 2^{-2L}. If we find a vertex that is within that margin of error, then it will be optimal.

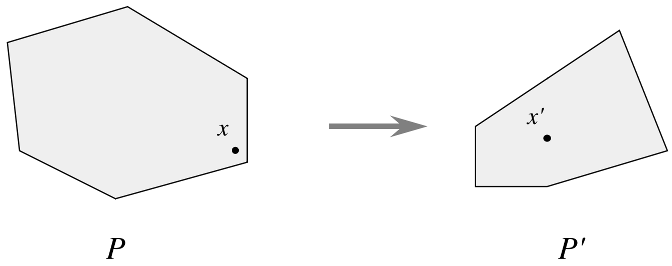


Figure 7: A centering mapping. If x is close to the boundary, we map the polyhedron P onto another one P', s.t. the image x' of x is closer to the center of P'.

13.1 Ye's Interior Point Algorithm

In the rest of these notes we present Ye's [9] interior point algorithm for linear programming. Ye's algorithm (among several others) achieves the best known asymptotic running time in the literature, and our presentation incorporates some simplifications made by Freund [3].

We are going to consider the following linear programming problem:

$$(P)\begin{cases} \text{minimize} & Z = c^T x \\ \text{subject to} & Ax = b, \\ & x \geq 0 \end{cases}$$

and its dual

$$(D)\begin{cases} \text{maximize} & W = b^T y \\ \text{subject to} & A^T y + s = c, \\ & s \geq 0. \end{cases}$$

The algorithm is primal-dual, meaning that it simultaneously solves both the primal and dual problems. It keeps track of a primal solution $\overline{x}$ and a vector of dual slacks $\overline{s}$ (i.e. $\exists\ \overline{y} : A^T\overline{y} = c - \overline{s}$) such that $\overline{x} > 0$ and $\overline{s} > 0$. The basic idea of this algorithm is to stay away from the boundaries of the polyhedron (the hyperplanes $x_j \geq 0$ and $s_j \geq 0$, $j = 1, 2, \ldots, n$) while approaching optimality. In other words, we want to make the duality gap

$$c^T\overline{x} - b^T\overline{y} = \overline{x}^T\overline{s} > 0$$

very small but stay away from the boundaries. Two tools will be used to achieve this goal in polynomial time.

Tool 1: Scaling (see Figure 7)

Scaling is a crucial ingredient in interior point methods. The two types of scaling commonly used are *projective scaling* (the one used by Karmarkar) and *affine scaling* (the one we are going to use).

Suppose the current iterate is $\overline{x} > 0$ and $\overline{s} > 0$, where $\overline{x} = (\overline{x}_1, \overline{x}_2, \dots, \overline{x}_n)^T$, then the affine scaling maps x to x' as follows.

$$x = \begin{pmatrix} x_1 \\ x_2 \\ \cdot \\ \cdot \\ \cdot \\ x_n \end{pmatrix} \longrightarrow x' = \begin{pmatrix} \frac{x_1}{\overline{x}_1} \\ \frac{x_2}{\overline{x}_2} \\ \cdot \\ \cdot \\ \cdot \\ \frac{x_n}{\overline{x}_n} \end{pmatrix}.$$

Notice this transformation maps $\overline{x}$ to $e = (1, \dots, 1)^T$.

We can express the scaling transformation in matrix form as $x' = \overline{X}^{-1}x$ or $x = \overline{X}x'$, where

$$\overline{X} = \begin{pmatrix} \overline{x}_1 & 0 & 0 & \dots & 0 \\ 0 & \overline{x}_2 & 0 & \dots & 0 \\ \vdots & & \vdots & & \vdots \\ 0 & 0 & \dots & \overline{x}_{n-1} & 0 \\ 0 & 0 & \dots & 0 & \overline{x}_n \end{pmatrix}.$$

Using matrix notation we can rewrite the linear program (P) in terms of the transformed variables as:

$$\begin{aligned} \text{minimize} \quad & Z = c^T \overline{X} x' \\ \text{subject to} \quad & A\overline{X}x' = b, \\ & x' \geq 0. \end{aligned}$$

If we define $\overline{c} = \overline{X}c$ (note that $\overline{X} = \overline{X}^T$) and $\overline{A} = A\overline{X}$ we can get a linear program in the original form as follows.

$$\begin{aligned} \text{minimize} \quad & Z = \overline{c}^T x' \\ \text{subject to} \quad & \overline{A}x' = b, \\ & x' \geq 0. \end{aligned}$$

We can also write the dual problem (D) as:

$$\begin{aligned} \text{maximize} \quad & W = b^T y \\ \text{subject to} \quad & (A\overline{X})^T y + \overline{X}s = \overline{c}, \\ & \overline{X}s \geq 0 \end{aligned}$$

or, equivalently,

$$\begin{aligned} \text{maximize} \quad & W = b^T y \\ \text{subject to} \quad & \overline{A}^T y + s' = \overline{c}, \\ & s' \geq 0 \end{aligned}$$

where $s' = \overline{X}s$, i.e.

$$s' = \begin{pmatrix} s_1\overline{x}_1 \\ s_2\overline{x}_2 \\ \cdot \\ \cdot \\ s_n\overline{x}_n \end{pmatrix}.$$

One can easily see that

$$x_j s_j = x'_j s'_j \qquad \forall j \in \{1, \ldots, n\} \tag{3}$$

and, therefore, the duality gap $x^T s = \sum_j x_j s_j$ remains unchanged under affine scaling. As a consequence, we will see later that one can always work equivalently in the transformed space.

Tool 2: Potential Function

Our potential function is designed to measure how small the duality gap is and how far the current iterate is away from the boundaries. In fact we are going to use the following "logarithmic barrier function".

Definition 9 (Potential Function, $G(x, s)$)

$$G(x, s) \triangleq q \ln(x^T s) - \sum_{j=1}^{n} \ln(x_j s_j), \qquad \text{for some q,}$$

where q is a parameter that must be chosen appropriately.

Note that the first term goes to $-\infty$ as the duality gap tends to 0, and the second term goes to $+\infty$ as $x_i \to 0$ or $s_i \to 0$ for some i. Two questions arise immediately concerning this potential function.

Question 1: How do we choose q?

Lemma 19 *Let $x, s > 0$ be vectors in $\mathbb{R}^{n\times 1}$. Then*

$$n \ln x^T s - \sum_{j=1}^{n} \ln x_j s_j \geq n \ln n.$$

Proof:

Given any n positive numbers $t_1, \ldots, t_n$, we know that their geometric mean does not exceed their arithmetic mean, i.e.

$$\left(\prod_{j=1}^{n} t_j\right)^{1/n} \leq \frac{1}{n}\left(\sum_{j=1}^{n} t_j\right).$$

Taking the logarithms of both sides we have

$$\frac{1}{n}\left(\sum_{j=1}^{n}\ln t_j\right) \le \ln\left(\sum_{j=1}^{n} t_j\right) - \ln n.$$

Rearranging this inequality we get

$$n\ln\left(\sum_{j=1}^{n} t_j\right) - \left(\sum_{j=1}^{n}\ln t_j\right) \ge n\ln n.$$

(In fact the last inequality can be derived directly from the concavity of the logarithmic function). The lemma follows if we set $t_j = x_j s_j$. □

Since our objective is that $G \to -\infty$ as $x^T s \to 0$ (since our primary goal is to get close to optimality), according to Lemma 19, we should choose some $q > n$ (notice that $\ln x^T s \to -\infty$ as $x^T s \to 0$) . In particular, if we choose $q = n+1$, the algorithm will terminate after $O(nL)$ iterations. In fact we are going to set $q = n + \sqrt{n}$, which gives us the smallest number — $O(\sqrt{n}L)$ — of iterations by this method.

Question 2: When can we stop?

Suppose that $x^T s \le 2^{-2L}$, then $c^T x - Z \le c^T x - b^T y = x^T s \le 2^{-2L}$, where Z is the optimum value to the primal problem. From Corollary 18, the following claim follows immediately.

Claim 20 *If $x^T s \le 2^{-2L}$, then any vertex x^* satisfying $c^T x^* \le c^T x$ is optimal.*

In order to find x^* from x, two methods can be used. One is based on purely algebraic techniques (but is a bit cumbersome to describe), while the other (the cleanest one in literature) is based upon basis reduction for lattices. We shall not elaborate on this topic, although we'll get back to this issue when discussing basis reduction in lattices.

Lemma 21 *Let x, s be feasible primal-dual vectors such that $G(x,s) \le -k\sqrt{n}L$ for some constant k. Then*

$$x^T s < e^{-kL}.$$

Proof:

By the definition of $G(x,s)$ and the previous theorem we have:

$$\begin{aligned}
-k\sqrt{n}L &\ge G(x,s) \\
&= (n+\sqrt{n})\ln x^T s - \sum_{j=1}^{n}\ln x_j s_j \\
&\ge \sqrt{n}\ln x^T s + n\ln n.
\end{aligned}$$

Rearranging we obtain

$$\begin{aligned}\ln x^T s &\leq -kL - \sqrt{n}\ln n \\ &< -kL.\end{aligned}$$

Therefore

$$x^T s < e^{-kL}. \qquad \square$$

The previous lemma and claim tell us that we can stop whenever $G(x,s) \leq -2\sqrt{n}L$. In practice, the algorithm can terminate even earlier, so it is a good idea to check from time to time if we can get the optimal solution right away.

Please notice that according to Equation (3) the affine transformation does not change the value of the potential function. Hence we can work either in the original space or in the transformed space when we talk about the potential function.

14 Description of Ye's Interior Point Algorithm

Initialization:

Set $i = 0$.

Choose $x^0 > 0$, $s^0 > 0$, and y^0 such that $Ax^0 = b$, $A^T y^0 + s^0 = c$ and $G(x^0, s^0) = O(\sqrt{n}L)$. (Details are not covered in class but can be found in the appendix. The general idea is as follows. By augmenting the linear program with additional variables, it is easy to obtain a feasible solution. Moreover, by carefully choosing the augmented linear program, it is possible to have feasible primal and dual solutions x and s such that all x_j's and s_j's are large (say 2^L). This can be seen to result in a potential of $O(\sqrt{n}L)$.)

Iteration:

$$\textbf{while } G(x^i, s^i) > -2\sqrt{n}L$$
$$\textbf{do}\left\{\begin{array}{l}\left.\begin{array}{l}\text{either a primal step (changing x}^{\text{i}}\text{ only)}\\ \text{or a dual step (changing s}^{\text{i}}\text{ only)}\end{array}\right\}\text{to get } (x^{i+1}, s^{i+1})\\ i := i+1\end{array}\right.$$

The iterative step is as follows. Affine scaling maps (x^i, s^i) to (e, s'). In this transformed space, the point is far away from the boundaries. Either a dual or primal step occurs, giving $(\tilde{x}, \tilde{s})$ and reducing the potential function. The point is then mapped back to the original space, resulting in (x^{i+1}, s^{i+1}).

Next, we are going to describe precisely how the primal or dual step is made such that

$$G(x^{i+1}, s^{i+1}) - G(x^i, s^i) \leq -\frac{7}{120} < 0$$

holds for either a primal or dual step, yielding an $O(\sqrt{n}L)$ total number of iterations.

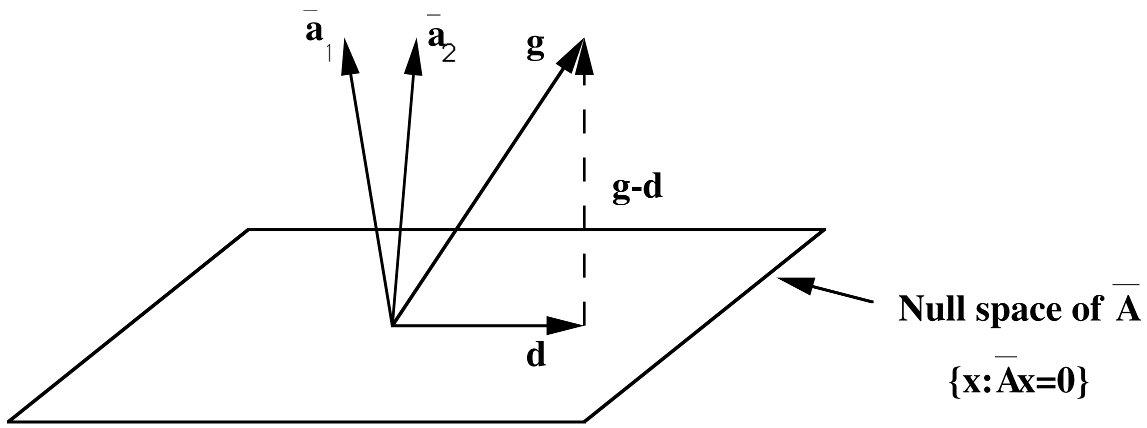


Figure 8: Null space of $\overline{A}$ and gradient direction g.

In order to find the new point $(\tilde{x}, \tilde{s})$ given the current iterate (e, s') (remember we are working in the transformed space), we compute the gradient of the potential function. This is the direction along which the value of the potential function changes at the highest rate. Let g denote the gradient. Recall that (e, s') is the map of the current iterate, we obtain

$$\begin{aligned} g &= \nabla_x G(x,s)|_{(e,s')} \\ &= \frac{q}{x^T s} s - \left.\begin{pmatrix} 1/x_1 \\ \vdots \\ 1/x_n \end{pmatrix}\right|_{(e,s')} \\ &= \frac{q}{e^T s'} s' - e \qquad (4) \end{aligned}$$

We would like to maximize the change in G, so we would like to move in the direction of $-g$. However, we must insure the new point is still feasible (i.e. $\overline{A}\tilde{x} = b$). Let d be the projection of g onto the null space $\{x : \overline{A}x = 0\}$ of $\overline{A}$. Thus, we will move in the direction of $-d$.

Claim 22 $d = (I - \overline{A}(\overline{A}\,\overline{A}^T)^{-1}\overline{A})g$.

Proof:
Since $g - d$ is orthogonal to the null space of $\overline{A}$, it must be the combination of some row vectors of $\overline{A}$. Hence we have

$$\begin{cases} \overline{A}d = 0 \\ \exists w, \text{ s.t. } \overline{A}^T w = g - d. \end{cases}$$

This implies

$$\begin{cases} \overline{A}^T w = g - d \\ (\overline{A}\,\overline{A}^T)w = \overline{A}g \end{cases} \qquad \text{(normal equations).}$$

Solving the normal equations, we get

$$w = (\overline{A}\,\overline{A}^T)^{-1}\overline{A}g$$

and

$$d = g - \overline{A}^T(\overline{A}\,\overline{A}^T)^{-1}\overline{A}g = (I - \overline{A}^T(\overline{A}\,\overline{A}^T)^{-1}\overline{A})g.$$

□

A potential problem arises if g is nearly perpendicular to the null space of $\overline{A}$. In this case, $||d||$ will be very small, and each primal step will not reduce the potential greatly. Instead, we will perform a dual step.

In particular, if $||d|| = ||d||_2 = \sqrt{d^T d} \geq 0.4$, we make a primal step as follows.

$$\begin{aligned} \tilde{x} &= e - \frac{1}{4||d||}d \\ \tilde{s} &= s'. \end{aligned}$$

Claim 23 $\tilde{x} > 0$.

Proof:

$\tilde{x}_j = 1 - \frac{1}{4}\frac{d_j}{||d||} \geq \frac{3}{4} > 0$. □

This claim insures that the new iterate is still an interior point. For the similar reason, we will see that $\tilde{s} > 0$ when we make a dual step.

Proposition 24 *When a primal step is made,* $G(\tilde{x}, \tilde{s}) - G(e, s') \leq -\frac{7}{120}$.

If $||d|| < 0.4$, we make a dual step. Again, we calculate the gradient

$$\begin{aligned} h &= \nabla_s G(x,s)|_{(e,s')} \\ &= \frac{q}{e^T s'}e - \begin{pmatrix} 1/s'_1 \\ \vdots \\ 1/s'_n \end{pmatrix} \end{aligned} \tag{5}$$

Notice that $h_j = g_j / s_j$, thus h and g can be seen to be approximately in the same direction.

Suppose the current dual feasible solution is y', s' such that

$$\overline{A}^T y' + s' = \overline{c}.$$

Again, we restrict the solution to be feasible, so

$$\begin{aligned} \overline{A}^T y + \tilde{s} &= \overline{c} \\ \tilde{s} - s' &= \overline{A}^T (y' - y) \end{aligned}$$

Thus, in the dual space, we move perpendicular to the null space and in the direction of $-(g-d)$.

Thus, we have

$$\tilde{s} = s' - (g-d)\mu$$

For any μ, $\exists y \;\; \overline{A}^T y + \tilde{s} = c$

So, we can choose $\mu = \frac{e^T s'}{q}$ and get $\overline{A}^T(y' + \mu w) + \tilde{s} = c$.

Therefore,

$$\begin{aligned} \tilde{s} &= s' - \frac{e^T s'}{q}(g-d) \\ &= s' - \frac{e^T s'}{q}(q\frac{s'}{e^T s'} - e - d) \\ &= \frac{e^T s'}{q}(d+e) \\ \tilde{x} &= x' = e. \end{aligned}$$

One can show that $\tilde{s} > 0$ as we did in Claim 23. So such move is legal.

Proposition 25 *When a dual step is made,* $G(\tilde{x}, \tilde{s}) - G(e, s') \leq -\frac{1}{6}$

According to these two propositions, the potential function decreases by a constant amount at each step. So if we start from an initial interior point (x^0, s^0) with $G(x^0, s^0) = O(\sqrt{n}L)$, then after $O(\sqrt{n}L)$ iterations we will obtain another interior point (x^j, s^j) with $G(x^j, s^j) \leq -k\sqrt{n}L$. From Lemma 21, we know that the duality gap $(x^j)^T s^j$ satisfies

$$(x^j)^T s^j \leq 2^{-kL},$$

and the algorithm terminates by that time. Moreover, each iteration requires $O(n^3)$ operations. Indeed, in each iteration, the only non-trivial task is the computation of the projected gradient d. This can be done by solving the linear system $(\bar{A}\bar{A}^T)w = \bar{A}g$ in $O(n^3)$ time using Gaussian elimination. Therefore, the overall time complexity of this algorithm is $O(n^{3.5}L)$. By using approximate solutions to the linear systems, we can obtain $O(n^{2.5})$ time per iteration, and total time $O(n^3 L)$.

15 Analysis of the Potential Function

In this section, we prove the two propositions of the previous section, which concludes the analysis of Ye's algorithm.

Proof of Proposition 24:

$$
\begin{aligned}
G(\tilde{x},\tilde{s}) - G(e,s') &= G(e - \frac{1}{4||d||}d, \tilde{s}) - G(e,s') \\
&= q\ln\left(e^T s' - \frac{d^T s'}{4||d||}\right) - \sum_{j=1}^{n}\ln\left(1 - \frac{d_j}{4||d||}\right) - \sum_{j=1}^{n}\ln s'_j - \\
&\quad -q\ln\left(e^T s'\right) + \sum_{j=1}^{n}\ln 1 + \sum_{j=1}^{n}\ln s'_j \\
&= q\ln\left(1 - \frac{d^T s'}{4||d||e^T s'}\right) - \sum_{j=1}^{n}\ln\left(1 - \frac{d_j}{4||d||}\right).
\end{aligned}
$$

Using the relation

$$
-x - \frac{x^2}{2(1-a)} \le \ln(1-x) \le -x \tag{6}
$$

which holds for $|x| \le a < 1$, we get:

$$
\begin{aligned}
G(\tilde{x},\tilde{s}) - G(e,s') &\le -\frac{q\,d^T s'}{4||d||e^T s'} + \sum_{j=1}^{n}\frac{d_j}{4||d||} + \sum_{j=1}^{n}\frac{d_j^2}{16||d||^2\,2(3/4)} \qquad \text{for } a = 1/4 \\
&= -\frac{q\,d^T s'}{4||d||e^T s'} + \frac{e^T d}{4||d||} + \frac{1}{24} \\
&= \frac{1}{4||d||}(e - \frac{q}{e^T s'}s')^T d + \frac{1}{24} \\
&= \frac{1}{4||d||}(-g)^T d + \frac{1}{24} \\
&= -\frac{||d||^2}{4||d||} + \frac{1}{24} \\
&= -\frac{||d||}{4} + \frac{1}{24} \\
&\le -\frac{1}{10} + \frac{1}{24} \\
&= -\frac{7}{120}.
\end{aligned}
$$

Note that $g^T d = ||d||^2$, since d is the projection of g. (This is where we use the fact that d is the projected gradient!) □

Before proving Proposition 25, we need the following lemma.

Lemma 26
$$\sum_{j=1}^{n} \ln(\tilde{s}_j) - n\ln(\frac{e^T\tilde{s}}{n}) \geq \frac{-2}{15}.$$

Proof:

Using the equality $\tilde{s} = \frac{\Delta}{q}(e+d)$ and Equation 6, which holds for $|x| \leq a < 1$, we see that

$$\begin{aligned}
\textstyle\sum_{j=1}^{n} \ln(\tilde{s}_j) - n\ln(\frac{e^T\tilde{s}}{n}) &= \textstyle\sum_{j=1}^{n} \ln(\frac{\Delta}{q}(1+d_j)) - n\ln(\frac{\Delta}{q}(1+\frac{e^Td}{n})) \\
&\geq \textstyle\sum_{j=1}^{n}(d_j - \frac{{d_j}^2}{2(3/5)}) - n\frac{e^Td}{n} \\
&\geq -\frac{||d||^2}{6/5} \\
&\geq \frac{-2}{15}
\end{aligned}$$

□

Proof of Proposition 25:

Using Lemma 26 and the inequality
$$\sum_{j=1}^{n} \ln(s_j) \leq n\ln(\frac{e^Ts}{n}),$$
which follows from the concavity of the logarithm function, we have

$$\begin{aligned}
G(e,\tilde{s}) - G(e,s') &= q\ln(\tfrac{e^T\tilde{s}}{e^Ts'}) - \textstyle\sum_{j=1}^{n}\ln(\tilde{s}_j) + \sum_{j=1}^{n}\ln(s'_j) \\
&\leq q\ln(\tfrac{e^T\tilde{s}}{e^Ts'}) + \tfrac{2}{15} - n\ln(\tfrac{e^T\tilde{s}}{n}) + n\ln(\tfrac{e^Ts'}{n}) \\
&= \tfrac{2}{15} + \sqrt{n}\ln(\tfrac{e^T\tilde{s}}{e^Ts'})
\end{aligned}$$

On the other hand,
$$e^T\tilde{s} = \frac{\Delta}{q}(n + e^Td)$$
and recall that $\Delta = e^Ts'$,
$$\frac{e^T\tilde{s}}{e^Ts'} = \frac{1}{q}(n+e^Td) \leq \frac{1}{n+\sqrt{n}}(n+0.4\sqrt{n}),$$
since, by Cauchy-Schwartz inequality, $|e^Td| \leq ||e||\,||d|| = \sqrt{n}||d||$. Combining the above inequalities yields

$$\begin{aligned}
G(e,\tilde{s}) - G(e,s') &\leq \tfrac{2}{15} + \sqrt{n}\ln(1 - \tfrac{0.6\sqrt{n}}{n+\sqrt{n}}) \\
&\leq \tfrac{2}{15} - \tfrac{0.6n}{n+\sqrt{n}} \\
&\leq \tfrac{2}{15} - \tfrac{3}{10} = -\tfrac{1}{6}
\end{aligned}$$

since $n + \sqrt{n} \leq 2n$. $\square$

This completes the analysis of Ye's algorithm.

16 Bit Complexity

Throughout the presentation of the algorithm, we assumed that all operations can be performed exactly. This is a fairly unrealistic assumption. For example, notice that $\|d\|$ might be irrational since it involves a square root. However, none of the thresholds we set were crucial. We could for example test whether $\|d\| \geq 0.4$ or $\|d\| \leq 0.399$. To test this, we need to compute only a few bits of $\|d\|$. Also, if we perform a primal step (i.e. $\|d\| \geq 0.4$) and compute the first few bits of $\|d\|$ so that the resulting approximation $\|d\|_{ap}$ satisfies $(4/5)\|d\| \leq \|d\|_{ap} \leq \|d\|$ then if we go through the analysis of the primal step performed in Proposition 1, we obtain that the reduction in the potential function is at least $19/352$ instead of the previous $7/120$. Hence, by rounding $\|d\|$ we can still maintain a constant decrease in the potential function.

Another potential problem is when using Gaussian elimination to compute the projected gradient. We mentioned that Gaussian elimination requires $O(n^3)$ arithmetic operations but we need to show that, during the computation, the numbers involved have polynomial size. For that purpose, consider the use of Gaussian elimination to solve a system $Ax = b$ where

$$A = A^{(1)} = \begin{pmatrix} a_{11}^{(1)} & a_{12}^{(1)} & \dots & a_{1n}^{(1)} \\ a_{21}^{(1)} & a_{22}^{(1)} & \dots & a_{2n}^{(1)} \\ \vdots & \vdots & \ddots & \vdots \\ a_{m1}^{(1)} & a_{m2}^{(1)} & \dots & a_{mn}^{(1)} \end{pmatrix}.$$

Assume that $a_{11} \neq 0$ (otherwise, we can permute rows or columns). In the first iteration, we substract $a_{i1}^{(1)}/a_{11}^{(1)}$ times the first row from row i where $i = 2, \dots, m$, resulting in the following matrix:

$$A^{(2)} = \begin{pmatrix} a_{11}^{(2)} & a_{12}^{(2)} & \dots & a_{1n}^{(2)} \\ 0 & a_{22}^{(2)} & \dots & a_{2n}^{(2)} \\ \vdots & \vdots & \ddots & \vdots \\ 0 & a_{m2}^{(2)} & \dots & a_{mn}^{(2)} \end{pmatrix}.$$

In general, $A^{(i+1)}$ is obtained by subtracting $a_{ji}^{(i)}/a_{ii}^{(i)}$ times row i from row j of $A^{(i)}$ for $j = i+1, \dots, m$.

Theorem 27 *For all $i \leq j, k$, $a_{jk}^{(i)}$ can be written in the form $\det(B)/\det(C)$ where B and C are some submatrices of A.*

Proof:

Let B_i denote the $i \times i$ submatrix of $A^{(i)}$ consisting of the first i entries of the first i rows. Let $B^{(i)}_{jk}$ denote the $i \times i$ submatrix of $A^{(i)}$ consisting of the first $i-1$ rows and row j, and the first $i-1$ columns and column k. Since B_i and $B^{(i)}_{jk}$ are upper triangular matrices, their determinants are the products of the entries along the main diagonal and, as a result, we have:

$$a^{(i)}_{ii} = \frac{\det(B_i)}{\det(B_{i-1})}$$

and

$$a^{(i)}_{jk} = \frac{\det(B^{(i)}_{jk})}{\det(B_{i-1})}.$$

Moreover, remember that row operations do not affect the determinants and, hence, the determinants of $B^{(i)}_{jk}$ and B_{i-1} are also determinants of submatrices of the original matrix A. $\square$

Using the fact that the size of the determinant of any submatrix of A is at most the size of the matrix A, we obtain that all numbers occuring during Gaussian elimination require only $O(L)$ bits.

Finally, we need to round the current iterates x, y and s to $O(L)$ bits. Otherwise, these vectors would require a constantly increasing number of bits as we iterate. By rounding up x and s, we insure that these vectors are still strictly positive. It is fairly easy to check that this rounding does not change the potential function by a significant amount and so the analysis of the algorithm is still valid. Notice that now the primal and dual constraints might be slightly violated but this can be taken care of in the rounding step.

A Transformation for the Interior Point Algorithm

In this appendix, we show how a pair of dual linear programs

$$(P) \quad \begin{array}{ll} Min & c^T x \\ s.t. & Ax = b \\ & x \geq 0 \end{array} \qquad\qquad (D) \quad \begin{array}{llcccc} Max & b^T y & & & & \\ s.t. & A^T y & + & s & = & c \\ & & & s & \geq & 0 \end{array}$$

can be transformed so that we know a strictly feasible primal solution x_0 and a strictly feasible vector of dual slacks s_0 such that $G(x_0; s_0) = O(\sqrt{n}L)$ where

$$G(x; s) = q \ln(x^T s) - \sum_{j=1}^{n} \ln(x_j s_j)$$

and $q = n + \sqrt{n}$.

Consider the pair of dual linear programs:

$$
\begin{array}{llrclclcl}
 & Min & c^T x & + & k_c x_{n+1} & & & & \\
(P') & s.t. & Ax & + & (b - 2^{2L}Ae)x_{n+1} & & & = & b \\
 & & (2^{4L}e - c)^T x & & & + & 2^{4L}x_{n+2} & = & k_b \\
 & & x \geq 0 & & x_{n+1} \geq 0 & & x_{n+2} \geq 0 & &
\end{array}
$$

and

$$
\begin{array}{llrclclclcl}
 & Min & b^T y & + & k_b y_{m+1} & & & & & & \\
(D') & s.t. & A^T y & + & (2^{4L}e - c)y_{m+1} & + & s & & & = & c \\
 & & (b - 2^{2L}Ae)^T y & & & & & + \; s_{n+1} & & = & k_c \\
 & & & & 2^{4L}y_{m+1} & & & & + \; s_{n+2} & = & 0 \\
 & & & & & & s, & s_{n+1}, & s_{n+2} & \geq & 0
\end{array}
$$

where $k_b = 2^{6L}(n+1) - 2^{2L}c^T e$ is chosen in such a way that $x' = (x, x_{n+1}, x_{n+2}) = (2^{2L}e, 1, 2^{2L})$ is a (strict) feasible solution to (P') and $k_c = 2^{6L}$. Notice that $(y', s') = (y, y_{m+1}, s, s_{n+1}, s_{n+2}) = (0, -1, 2^{4L}e, k_c, 2^{4L})$ is a feasible solution to (D') with $s' > 0$. x' and (y', s') serve as our initial feasible solutions.

We have to show:

1. $G(x'; s') = O(\sqrt{n'}L)$ where $n' = n + 2$,
2. the pair $(P') - (D')$ is equivalent to $(P) - (D)$,
3. the input size L' for (P') as defined in the lecture notes does not increase too much.

The proofs of these statements are simple but heavily use the definition of L and the fact that vertices have all components bounded by 2^L.

We first show 1. Notice first that $x'_j s'_j = 2^{6L}$ for all j, implying that

$$
\begin{aligned}
G(x'; s') &= (n' + \sqrt{n'}) \ln(x'^T s') - \sum_{j=1}^{n'} \ln(x'_j s'_j) \\
&= (n' + \sqrt{n'}) \ln(2^{6L} n') - n' \ln(2^{6L}) \\
&= \sqrt{n'} \ln(2^{6L}) + (n' + \sqrt{n'}) \ln(n') \\
&= O(\sqrt{n'}L)
\end{aligned}
$$

In order to show that $(P') - (D')$ are equivalent to $(P) - (D)$, we consider an optimal solution x^* to (P) and an optimal solution (y^*, s^*) to (D) (the case where (P) or (D) is infeasible is considered in the problem set). Without loss of generality, we can assume that x^* and (y^*, s^*) are vertices of the corresponding polyhedra. In particular, this means that $x^*_j, |y^*_j|, s^*_j < 2^L$.

Proposition 28 *Let* $x' = (x^*, 0, (k_b - (2^{4L}e - c)^T x^*)/2^{4L})$ *and let* $(y', s') = (y^*, 0, , s^*, k_c - (b - 2^{2L}Ae)^T y^*, 0)$. *Then*

1. x' *is a feasible solution to* (P') *with* $x'_{n+2} > 0$,
2. (y', s') *is a feasible solution to* (D') *with* $s'_{n+1} > 0$,
3. x' *and* (y', s') *satisfy complementary slackness, i.e. they constitute a pair of optimal solutions for* $(P') - (D')$.

Proof:

To show that x' is a feasible solution to (P') with $x'_{n+2} > 0$, we only need to show that $k_b - (2^{4L}e - c)^T x^* > 0$ (the reader can easily verify that x' satisfy all the equalities defining the feasible region of (P')). This follows from the fact that

$$(2^{4L}e - c)^T x^* \leq n(2^{4L} + 2^L)2^L = n(2^{5L} + 2^{2L}) < n2^{6L}$$

and

$$k_b = 2^{6L}(n+1) - 2^{2L}c^T e \geq 2^{6L}(n+1) - 2^{2L}n \max_j |c_j| \geq 2^{6L}n + 2^{6L} - 2^{3L} > n2^{6L}$$

where we have used the definition of L and the fact that vertices have all their entries bounded by 2^L.

To show that (y', s') is a feasible solution to (D') with $s'_{n+1} > 0$, we only need to show that $k_c - (b - 2^{2L}Ae)^T y^* > 0$. This is true since

$$\begin{aligned}(b - 2^{2L}Ae)^T y^* &\leq b^T y^* - 2^{2L}e^T A^T y^* \\ &\leq m \max_i |b_i| 2^L + 2^{2L} nm \max_{i,j} |a_{ij}| 2^L \\ &= 2^{2L} + 2^{4L} < 2^{6L} = k_c.\end{aligned}$$

x' and (y', s') satisfy complementary slackness since

- $x^{*T}s^* = 0$ by optimality of x^* and (y^*, s^*) for (P) and (D)
- $x'_{n+1}s'_{n+1} = 0$ and
- $x'_{n+2}s'_{n+2} = 0$.

□

This proposition shows that, from an optimal solution to $(P) - (D)$, we can easily construct an optimal solution to $(P') - (D')$ of the same cost. Since this solution has $s'_{n+1} > 0$, any optimal solution $\hat{x}$ to (P') must have $\hat{x}_{n+1} = 0$. Moreover, since $x'_{n+2} > 0$, any optimal solution $(\hat{y}, \hat{s})$ to (D') must satisfy $\hat{s}_{n+2} = 0$ and, as a result, $\hat{y}_{m+1} = 0$. Hence, from any optimal solution to $(P') - (D')$, we can easily deduce an optimal solution to $(P) - (D)$. This shows the equivalence between $(P) - (D)$ and $(P') - (D')$.

By some tedious but straightforward calculations, it is possible to show that L' (corresponding to $(P')-(D')$) is at most $24L$. In other words, $(P)-(D)$ and $(P')-(D')$ have equivalent sizes.

References

[1] V. Chvatal. *Linear Programming.* W.H. Freeman and Company, 1983.

[2] G. Dantzig. Maximization of a linear function of variables subject to linear inequalities. In T. Koopmans, editor, *Activity Analysis of Production and Allocation*, pages 339–347. John Wiley & Sons, Inc., 1951.

[3] R. M. Freund. Polynomial-time algorithms for linear programming based only on primal scaling and project gradients of a potential function. *Mathematical Programming*, 51:203–222, 1991.

[4] D. Goldfarb and M. Todd. Linear programming. In *Handbook in Operations Research and Management Science*, volume 1, pages 73–170. Elsevier Science Publishers B.V., 1989.

[5] C. C. Gonzaga. Path-following methods for linear programming. *SIAM Review*, 34:167–224, 1992.

[6] L. Khachian. A polynomial algorithm for linear programming. *Doklady Akad. Nauk USSR*, 244(5):1093–1096, 1979.

[7] K. Murty. *Linear Programming.* John Wiley & Sons, 1983.

[8] A. Schrijver. *Theory of Linear and Integer Programming.* John Wiley & Sons, 1986.

[9] Y. Ye. An $O(n^3L)$ potential reduction algorithm for linear programming. *Mathematical Programming*, 50:239–258, 1991.

18.415/6.854 Advanced Algorithms November 1994

Network flows

Lecturer: Michel X. Goemans

In these notes, we study some problems in "Network Flows". For a more comprehensive treatment, the reader is referred to the surveys [12, 1], or to the recent book [2]. Network flow problems arise in a variety of settings; the underlying networks might be transportation networks, communication networks, hydraulic networks, computer chips, or some abstract network. The field was born from applications in the 40's and 50's and has since developed into a strong methodological core with numerous algorithmic issues. The first polynomial time algorithms for network flow problems have been developed in the 70's, and constant progress towards faster and faster algorithms has been made in the 80's. Network flow problems can be formulated as linear programs and, as a result, all the methodology of linear programming can be applied. Duality plays a crucial role, and the simplex algorithm can take advantage of the structure of network flow problems (bases can be nicely characterized).

Some of the basic problems in this area include the single source shortest path problem, the maximum flow problem, and the minimum cost flow problem. First, we shall briefly review each of them and then we shall describe a polynomial time algorithm due to Goldberg and Tarjan [14] for the minimum cost flow problem.

1 Single Source Shortest Path Problem

We are interested in the following problem:
Given

- a directed graph $G = (V, E)$ where V is the set of vertices and E is the set of edges,
- and a length function $l : E \rightarrow \mathbb{Z}$,
- a distinguished vertex $s \in V$ (the source vertex),

Find for all $v \in V$ the length $\delta(v)$ of the shortest path from s to v.

This is NP-hard if we allow negative length cycles (i.e. cycles for which the sum of the lengths of its edges is negative). However, if all lengths are nonnegative ($l(u, v) \geq 0$ for all edges $(u, v) \in E$) then a standard algorithm that solves this problem is Dijkstra's algorithm [6] (see also [4]). The implementation of Dijkstra's algorithm is based on the implementation of a priority queue and various implementations of this priority queue lead to different worst-case running times. Using a Fibonacci heap implementation [10] of the priority queue, it can be shown that the algorithm has a total running time of $O(m + n \log n)$ where $m = |E|$ and $n = |V|$. This is the best known strongly polynomial algorithm for the single-source shortest path problem. An algorithm is said to be **strongly polynomial** if

1. It performs a polynomially bounded number of operations in the number of input data (in this case m and n). We allow the operations +,-,*,< and rational division.

2. The sizes of the numbers occurring during the algorithm are polynomially bounded in the size of the input.

There are also single-source shortest path algorithms which may not be strongly polynomial, i.e. algorithms whose running time depends on $L = \max l(u, v) + 1$. These algorithms may achieve a better running time than Dijkstra's algorithm, provided L is not too large. Listed below are four such algorithms:

Dial [5]	$O(m + nL)$
Johnson [16]	$O(m \log \log L)$
Gabow [11]	$O(m \log_d L)$ where $d = \max(2, \lceil m/n \rceil)$
Ahuja, Mehlhorn, Orlin, Tarjan [3]	$O(m + n\sqrt{\log L})$

Observe that all these algorithms except Dial's algorithm are polynomial since the size of the input is at least $\log L$.

If negative lengths are allowed then the problem can still be solved in polynomial time provided that no negative length cycle exists. The algorithm of Bellman-Ford solves this problem in $O(nm)$ time.

We would like to point out that these problems are defined on a directed graph. An undirected shortest path problem can easily be reduced to a directed instance by replacing every edge by bidirected edges. This reduction is fine if all lengths are nonnegative (in which case Dijkstra's algorithm can be used), but does not work if there are edges of negative length. In this case, indeed, a negative length cycle would be created. However, the undirected shortest path problem on an undirected graph with possibly negative edge lengths but no negative length cycle can still be solved in polynomial time. The algorithm is, however, fairly complicated and is based on a reduction of the problem to nonbipartite matching.

2 The Maximum Flow Problem

Given

- a directed graph $G = (V, E)$ where V is the set of vertices and E is the set of edges,

- capacity $u(v, w) \geq 0$ for $(v, w) \in E$,

- source $s \in V$,

- sink $t \in V$,

a flow f is an assignment of values to the edges which satisfies $f(v,w) \le u(v,w)$ for all edges (v,w) and which satisfies the flow conservation constraints

$$\sum_{w:(v,w)\in E} f(v,w) = \sum_{w:(w,v)\in E} f(w,v)$$

for all vertices v except s and t. The goal is to find a flow such that the net flow $\sum_w f(s,w)$ out of s ($\sum_{v:(s,v)\in E} f(s,v)$) is maximum. One can easily derive that the net flow out of s is equal to the net flow into t, and thus we could maximize this latter quantity as well.

All these constraints are linear constraints and the objective function is linear, so the maximum flow problem (MAX FLOW) is a linear program. We could therefore exploit the structure of the linear program to tailor the simplex algorithm. This has been done and, in fact, although no version of the simplex algorithm is known to run in polynomial time for general linear programs, it is known that it can be made to run in polynomial time (or even in strongly polynomial time) for the maximum flow problem. Goldfarb and Hao [15] have developed a version of the simplex which makes at most nm pivots and run in $O(n^2m)$ time. However, there are some more efficient combinatorial algorithms which exploit the structure of the problem. Most known algorithms are based on the concept of "augmenting paths", introduced by Ford and Fulkerson [8]. There are a variety of algorithms with different running times. The best strongly polynomial running time of a max flow algorithm is $O(nm \log n^2/m)$ (due to Goldberg and Tarjan [13]).

3 Minimum Cost Circulation Problem

In the minimum cost circulation problem, we are given a directed graph $G = (V, E)$. For each arc $(v,w) \in E$, we are given a cost $c(v,w)$, a lower bound $l(v,w)$ and an upper bound $u(v,w)$. Throughout, we assume that $l(.,.), u(.,.)$ and $c(.,.)$ are integral unless mentioned otherwise. We will associate a flow f with each arc of the graph. This flow will be required to satisfy $l(v,w) \le f(v,w) \le u(v,w)$ and the cost of the flow on (v,w) is defined to be $c(v,w)f(v,w)$. This is the classical notation. However, in our lectures, we adopt Goldberg and Tarjan's notation [14] in which every directed arc (v,w) is represented by arcs (v,w) and (w,v) (see Figure 1).

This will simplify the proofs later on. In this notation, the flow $f(w,v)$ on (w,v) is assumed to be equal to $-f(v,w)$, i.e. the flow is antisymmetric. Using this antisymmetry assumption, the lower bound on the flow $f(v,w)$ is equivalent to an upper bound of $-l(v,w)$ on $f(w,v)$. Also, the cost $c(w,v)$ on the arc (w,v) is defined to be $-c(v,w)$. This ensures that, if we push some flow on (v,w) and then decide to push it back from w to v, we get a full refund of the cost incurred (i.e. $c(v,w)f(v,w)$). Notice that the total cost of the flow on the arcs (v,w) and (w,v) is equal to

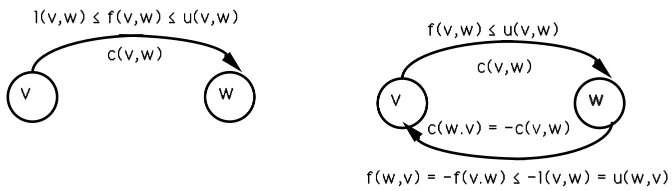


Figure 1: Standard notation vs. Goldberg-Tarjan notation.

$c(v,w)f(v,w) + c(w,v)f(w,v) = c(v,w)f(v,w) - c(v,w)f(w,v) = 2c(v,w)f(v,w) = 2c(w,v)f(w,v)$.

To recapitulate, we are given a bidirected[1] graph $G = (V, E)$, a capacity function $u : E \to \mathbb{Z}$ and a cost function $c : E \to \mathbb{Z}$. The cost function is assumed to be antisymmetric:

$$c(v,w) = -c(w,v) \qquad\qquad \forall (v,w) \in E.$$

A flow is a function $f : E \longrightarrow \mathbb{R}$, which is assumed

1. to be antisymmetric, i.e. $f(v,w) = -f(w,v)$, and

2. to satisfy the capacity constraints: $f(v,w) \leq u(v,w)$ for all $(v,w) \in E$.

The cost of a flow is defined as:

$$c \cdot f = \sum_{(v,w) \in E} c(v,w)u(v,w).$$

A flow is said to be a *circulation* if

$$\sum_w f(v,w) = 0$$

for all $v \in V$. Using the antisymmetry constraints, this is equivalent to saying that the flow out of vertex v minus the flow into v is equal to 0 for all $v \in V$. These conditions are thus the flow conservation constraints. The *minimum cost circulation problem* is the problem of finding a circulation of minimum cost.

A closely related problem to the minimum cost circulation problem is the *minimum cost flow problem.* In this problem, we are also given a supply function $b : V \to \mathbb{Z}$ satisfying $\sum_{v \in V} b(v) = 0$ and the flow is required to satisfy

$$\sum_w f(v,w) = b(v) \tag{1}$$

for all $v \in V$. The goal is to find a flow satisfying (1) of minimum cost. The minimum cost circulation problem is clearly a special case of the minimum cost flow problem (simply take $b(v) = 0$ for all $v \in V$). However, the converse is also true. Consider

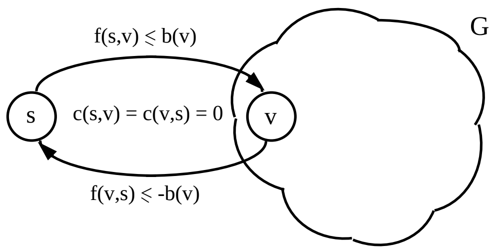


Figure 2: How to convert a minimum cost flow problem into a minimum cost circulation problem.

the following transformation that converts any instance of the minimum cost flow problem into an instance of the minimum cost circulation problem (see Figure 3).

Let $G' = (V', E')$ be the graph obtained by extending G with one extra vertex, say s, linked to all other vertices, i.e. $V' = V \cup \{s\}$ and $E' = E \cup \{(s, v) : v \in V\} \cup \{(v, s) : v \in V\}$. For these new edges, let $c(s, v) = c(v, s) = 0$ and $u(s, v) = b(v) = -u(v, s)$, the other costs and capacities remaining unchanged. The capacities on the bidirected edges incident to s have been chosen in such a way that, for any flow f on this extended graph, we have $f(s, v) = b(v)$. Therefore, any circulation f on G' induces a flow on G satisfying (1) and vice versa. Since this circulation on G' and this flow on G have the same cost, we can solve the minimum cost flow problem by solving a minimum cost circulation problem.

In these notes, we develop a purely combinatorial algorithm to solve the minimum cost circulation problem and we will also show that this problem can be solved in strongly polynomial time. (We'd like to point out that for the minimum cost flow or circulation problem, it is not known whether the simplex method can be adapted to run in strongly polynomial time (contrary to the case for the maximum flow problem).)

In many situations, the circulation is required to be integral. This additional restriction is not restrictive as indicated in the following Theorem — sometimes referred to as the *integrality theorem*.

Theorem 1 *If $u(v, w) \in \mathbb{Z}$ for all $(v, w) \in E$ then there exists an optimal circulation (or flow) with $f(v, w) \in \mathbb{Z}$.*

Although there are several ways to prove this result, we will deduce it later in the notes from a simple algorithm for the minimum cost circulation problem. More precisely, we will show that, at every iteration of the algorithm, the current circulation is integral and, hence, it is also integral when the algorithm terminates.

The minimum cost circulation problem has some interesting special cases as described in the next sections. Our strongly polynomial time algorithm for the minimum cost circulation problem will thus lead to strongly polynomial time algorithms

[1] $(v, w) \in E$ implies $(w, v) \in E$.

for these special cases (although more efficient algorithms can be designed for these special cases).

3.1 The Maximum Flow Problem

The maximum flow problem is a special case of the minimum cost circulation problem. Indeed, given an instance of the maximum flow problem, add an edge between s and t (see Figure 3.1) and define $u(t, s) = \infty$, $u(s, t) = 0$, $c(t, s) = -1 = -c(s, t)$ and $c(v, w) = 0$ for all $(v, w) \neq (s, t)$.

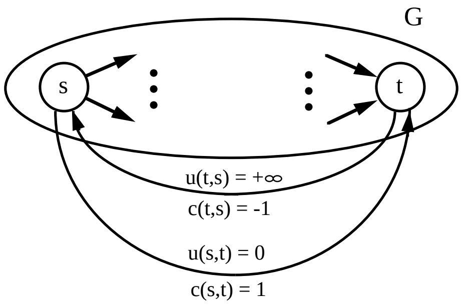


Figure 3: How to transform a maximum flow problem into a minimum cost circulation problem.

The capacities on the bidirected edge (s, t) is such that $f(t, s) \geq 0$, implying that the flow goes from t to s. There is a one-to-one correspondence between circulations in this extended graph and flows in the original graph satisfying all flow conservation constraints in $V \setminus \{s, t\}$. Moreover, the cost of any circulation in this extended graph is exactly equal to minus the net flow out of s (or into t) in the original graph. As a result, the maximum flow problem in G is equivalent to the minimum cost circulation problem in the extended graph.

Using the integrality theorem (Theorem 1), we obtain that the flow of maximum value can be assumed to be integral whenever the capacities are integral.

3.2 Bipartite Matching

The maximum cardinality matching problem on a bipartite graph $G = (A, B, E)$ (A and B denotes the bipartition of the vertex set) is the problem of finding the largest number of disjoint edges. This problem is a special case of the maximum flow problem and, hence, of the minimum cost circulation problem. To transform the maximum cardinality bipartite matching problem into a maximum flow problem (see Figure 3.2), we

1. direct all the edges from A to B,

2. add a source vertex s, a sink vertex t,

3. add the edges (s, a) for all vertices $a \in A$ and the edges (b, t) for all vertices $b \in B$ and

4. define the capacity of all existing edges to be 1 and the capacity of their reverse edges to be 0 (in other words, the flow on the existing edges have a lower bound of 0).

By the integrality theorem, we know that the flow on any existing edge can be assumed to be either 0 or 1. Therefore, to any flow f, there corresponds a matching $M = \{(v, w) \in E : f(v, w) = 1\}$ whose cardinality is precisely equal to the net amount of flow out of vertex s.

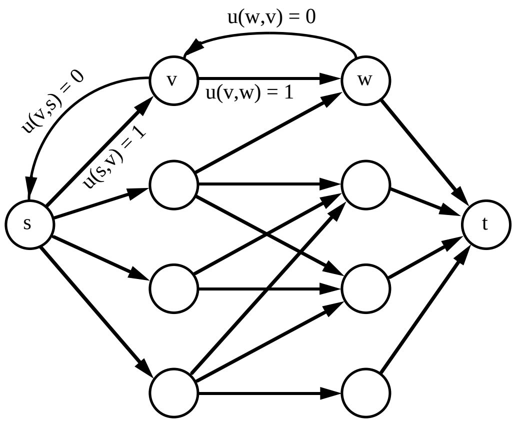


Figure 4: Maximum cardinality bipartite matching is a special case of maximum-flow.

It is also easy to construct from a matching M a flow of value $|M|$. As a result, any integral flow of maximum value will correspond to a matching of maximum cardinality.

In fact, the minimum weighted bipartite matching problem is also a special case of the minimum cost circulation problem. We can modify the above transformation in the following way. Define the cost of any edge of the original graph to be its original cost and the cost of any new edge to be 0. Now, we can model three versions of the minimum weighted bipartite matching problem by appropriately defining the capacities on the edges (t, s) and (s, t):

1. If $u(t, s) = n$ and $u(s, t) = -n$ where $n = |A| = |B|$, we get the minimum weighted perfect (a *perfect* matching is a matching that covers all the vertices) matching.

2. If $u(t, s) = n$ and $u(s, t) = 0$, we obtain the minimum weighted matching.

3. If $u(t, s) = k$ and $u(s, t) = -k$, we obtain the minimum weighted matching of size k.

3.3 Shortest paths

The single source shortest path problem is also a special case of the minimum cost flow problem. Indeed, by setting $l(v,w) = 0$ and $u(v,w) = 1$ for every edge (and letting their cost be the original cost), and introducing an edge from t to s with $u(t,s) = l(t,s) = 1$ and $c(t,s) = 0$, we obtain an equivalent instance of the minimum cost circulation problem.

4 Some Important Notions

We now go back to the minimum cost circulation problem, and before describing a polynomial time algorithm for it, we present some useful tools.

4.1 Residual Graph

Given a minimum cost circulation problem and a circulation f, we define *the residual graph* $G_f = (V, E_f)$ with respect to f by $E_f = \{(v,w) : f(v,w) < u(v,w)\}$. For example, if $u(v,w) = 5$, $u(w,v) = -1$ and $f(v,w) = 3$ (hence, $f(w,v) = -3$ by antisymmetry) then both (v,w) and (w,v) will be present in E_f. However, if $f(v,w) = 1$ (i.e. $f(w,v) = -1$), only (v,w) will be in E_f. With respect to f, we define *the residual capacity* of the edge (v,w) by

$$u_f(v,w) = u(v,w) - f(v,w).$$

Notice that the edges of the residual graph have a positive residual capacity.

4.2 Potentials

We associate with each vertex v a vertex potential $p(v)$. The potential of a vertex can be interpreted as the dual variable corresponding to the flow conservation constraints in the linear programming formulation of the problem. The *reduced cost* of the edge (v,w) is then defined as $c_p(v,w) := c(v,w) + p(v) - p(w)$. Note that the reduced costs are still antisymmetric i.e. $c_p(w,v) = c(w,v) + p(w) - p(v) = -c(v,w) - p(v) + p(w) = -c_p(v,w)$. Note also that the *cost*

$$c(\Gamma) := \sum_{(v,w)\in\Gamma} c(v,w)$$

of a directed cycle Γ is equal to its reduced cost

$$c_p(\Gamma) = \sum_{(v,w)\in\Gamma} c_p(v,w)$$

since the vertex potential of any vertex v on the cycle is added and subtracted exactly once. More generally, we have the following result.

Theorem 2 *For any $p : V \to \mathbb{Z}$ and any circulation f, we have $c \cdot f = c_p \cdot f$.*

Proof: By definition,

$$\begin{aligned} c_p \cdot f = \sum_{(v,w)\in E} c_p(v,w) f(v,w) &= \sum_{(v,w)\in E} c(v,w) f(v,w) + \sum_{(v,w)\in E} p(v) f(v,w) \\ &\quad - \sum_{(v,w)\in E} p(w) f(v,w) \\ &= c \cdot f + \sum_{v\in V} p(v) \sum_{w:(v,w)\in E} f(v,w) \\ &\quad - \sum_{w\in V} p(w) \sum_{v:(v,w)\in E} f(v,w) \\ &= c \cdot f + 0 - 0 = c \cdot f, \end{aligned}$$

since by definition of a circulation $\sum_{w:(v,w)\in E} f(v,w) = 0$. □

5 When is a circulation Optimal?

The next theorem characterizes a circulation of minimum cost.

Theorem 3 *For a circulation f, the following are equivalent:*

1. *f is of minimum cost,*
2. *there are no negative (reduced) cost cycles in the residual graph,*
3. *there exist potentials p such that $c_p(v,w) \geq 0$ for $(v,w) \in E_f$.*

This is essentially strong duality, but we will not refer to linear programming in the proof. **Proof:**

- $(\neg 2) \Rightarrow (\neg 1)$.
  Let Γ be a negative cost cycle in E_f. Let

  $$\delta = \min_{(v,w)\in\Gamma} u_f(v,w) > 0.$$

  By *pushing* δ units of flow along Γ, we mean replacing f by $\tilde{f}$ where

  $$\tilde{f}(v,w) = \begin{cases} f(v,w) + \delta & (v,w) \in \Gamma \\ f(v,w) - \delta & (w,v) \in \Gamma \\ f(v,w) & \text{otherwise} \end{cases}.$$

  Notice that, as defined, $\tilde{f}$ also satisfies the antisymmetry constraints and is a circulation. Moreover, $c \cdot \tilde{f} = c \cdot f + \delta \cdot c(\Gamma) < c \cdot f$. This implies that f is not of minimum cost.

- $2 \Rightarrow 3$.
  Let G' be obtained from the residual graph G_f by adding a vertex s linked to all other vertices by edges of cost 0 (the costs of these edges do not matter). Let $p(v)$ be the length of the shortest path from s to v in G' with respect to the costs $c(.,.)$.

  This quantities are well-defined since G_f does not contain any negative cost directed cycle. By definition of the shortest paths, we have $p(w) \leq p(v)+c(v,w)$ for all edges $(v,w) \in E_f$. This implies that $c_p(v,w) \geq 0$ whenever $(v,w) \in E_f$.

- $3 \Rightarrow 1$.
  The proof is by contradiction. Let f^* be a circulation such that $c \cdot f^* < c \cdot f$. Consider $f'(v,w) = f^*(v,w) - f(v,w)$. By definition of the residual capacities, f' is a feasible circulation with respect to $u_f(.,.)$. Its cost is

$$\begin{aligned} c \cdot f' = c_p \cdot f' &= \sum_{(v,w)\in E} c_p(v,w) f'(v,w) \\ &= 2 \sum_{(v,w)\in E: f'(v,w)>0} c_p(v,w) f'(v,w) \\ &\geq 0, \end{aligned}$$

  since $f'(v,w) > 0$ implies that $(v,w) \in E_f$ and, hence, $c_p(v,w) \geq 0$. This contradicts the fact that $c \cdot f' = c \cdot f^* - c \cdot f < 0$.

□

A problem is *well characterized* if its decision version belongs to $NP \cap co{-}NP$. The above theorem gives a good characterization for the minimum cost circulation problem (to be precise, we would also need to show that the potentials can be compactly encoded). It also naturally leads to a simple algorithm, first discovered by Klein [17] also known as the 'Cycle Cancelling Algorithm'.

6 Klein's Cycle Canceling Algorithm

Cycle canceling algorithm (Klein):

1. Let f be any circulation.

2. While G_f contains a negative cycle Γ do
   push $\delta = \min_{(v,w)\in\Gamma} u_f(v,w)$ along Γ.

Recall that in the previous code "push" means that we increase the flow by δ along the well oriented edges of Γ and decrease it by δ along the other edges of Γ.

In Step 1, we will assume that $f = 0$ satisfies the capacity constraints (i.e. $f = 0$ is a circulation). If this is not the case then a circulation can be obtained by solving one maximum flow problem or by modifying the instance so that a circulation can easily be found.

The cycle canceling algorithm can be used to prove Theorem 1. The proof is by induction. Assume that the initial circulation is chosen to be integral. Now, if at iteration k the circulation is integral, then the residual capacities as well as δ are also integral. Therefore, the circulation remains integral throughout the algorithm.

For the maximum flow problem as discussed in Section 3.1, any negative cost directed cycle must consist of a directed path from s to t along with the arc (t, s) since the only negative cost arc is (t, s). Therefore, in this special case, Klein's algorithm reduces to the well-known Ford-Fulkerson's augmenting path algorithm [8].

Ford-Fulkerson's augmenting path algorithm:

1. Start with the zero flow: $f = 0$.
2. While G_f contains a directed path P from s to t do
   push $\delta = \min_{(v,w)\in\Gamma} u_f(v, w)$ along P.

In the next Theorem, we show that the cycle canceling algorithm is correct if the costs and capacities are integral.

Theorem 4 *If $c : E \to \mathbb{Z}$ and $u : E \to \mathbb{Z}$ then Klein's algorithm terminates after $O(mCU)$ iterations where $m = |E|$, C is an upper bound on the absolute value of any cost and U is an upper bound on the absolute value of any capacity. Moreover, the resulting circulation is optimal.*

Proof: Since the costs are integral, any cycle of negative cost has a cost of at most -1. Moreover, if $(v, w) \in G_f$ then $u_f(v, w) \geq 1$ which implies that $\delta \geq 1$. Therefore, at each iteration, the cost of the current circulation decreases by at least 1 unit. On the other hand, since $|c(v, w)| \leq C$ and $|f(v, w)| \leq U$, the absolute value of the cost of the optimal circulation is at most mCU. Therefore, the algorithm terminates after $O(mCU)$ iterations. At that point, the residual graph does not contain any negative cycle and, hence, by Theorem 3, the circulation is optimal. $\square$

The bound given in Theorem 4 is however not polynomial. In fact, if the negative cycles (or the directed paths in Ford and Fulkerson's algorithm) are not appropriately chosen, the worst-case running time of the algorithm is exponential. In Figure 6, we have an instance of the maximum flow problem in which if we augment alternatively along the paths $s - 1 - 2 - t$ and $s - 2 - 1 - t$, the number of iterations will be $2C$ since at each iteration we push only one additional unit of flow from s to t. If $C = 2^n$, this gives an exponential bound.

Even more surprisingly, the cycle canceling algorithm and the augmenting path algorithm without any specification of which negative directed cycle or which directed st-path to select are not correct if the capacities are irrational. In [9], it is shown that

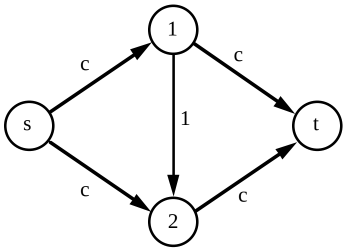


Figure 5: Numbers on the arcs represent the capacities. The reverse arcs have zero capacities.

the augmenting path algorithm can converge to a suboptimal flow if the capacities are irrational and the directed paths are selected in an unfortunate way.

To obtain a *polynomial* running time, we therefore need to specify which negative directed cycle to cancel. If the negative cycle resulting in the maximum cost improvement is selected, the number of iterations becomes polynomial. Unfortunately, finding this cycle is NP-hard. For the maximum flow problem, however, this selection rule reduces to finding the st-path with maximum residual capacity. Such a path can be found in polynomial time (for example, by adapting Dijkstra's algorithm). The resulting algorithm, due to Edmonds and Karp [7], requires $O(m \log U)$ iterations. The time per iteration is $O(m)$ (amortized). Hence we can implement the algorithm with a total running time of $O(m^2 \log U)$ (Tarjan [20]).

For a long time the question of finding a strongly polynomial algorithm (and even its existence) for the minimum cost circulation problem was kept open. In 1985, Éva Tardos [19] devised the first such algorithm. In 1987, Goldberg and Tarjan [14] produced an improved version that we will now present.

7 The Goldberg-Tarjan Algorithm

Define the mean cost of a cycle Γ to be

$$\frac{c(\Gamma)}{|\Gamma|} = \frac{\sum_{(v,w)\in\Gamma} c(v,w)}{|\Gamma|}$$

where $|\Gamma|$ represents the number of arcs in Γ. The minimum mean cost cycle of a graph can be found in strongly polynomial time, namely in $O(nm)$ time, by adapting the Bellman-Ford algorithm for the all pairs shortest path problem. Let

$$\mu(f) = \min_{\text{cycles } \Gamma \text{ in } E_f} \frac{c(\Gamma)}{|\Gamma|}$$

denote the minimum mean cost of all cycles in G_f.

Goldberg-Tarjan algorithm [14]:

1. Let $f = 0$.
2. While $\mu(f) < 0$ do
   push $\delta = \min_{(v,w)\in\Gamma} u_f(v,w)$ along a minimum mean cost cycle Γ of G_f.

The Goldberg-Tarjan algorithm is a cycle canceling algorithm since G_f has a negative directed cycle iff $\mu(f) < 0$.

For the maximum flow problem, this algorithm reduces to the Edmonds-Karp shortest augmenting path algorithm [7] in which the st-path with the fewest number of arcs is selected. The Edmonds-Karp shortest augmenting path algorithm requires $O(m)$ time per augmentation and the number of augmentations is $O(nm)$. This results in a running time of $O(nm^2)$.

8 Analysis of the Goldberg-Tarjan Algorithm

Before analyzing the Goldberg-Tarjan cycle canceling algorithm, we need some definitions.

Definition 1 *A circulation f is ϵ-optimal if there exists p such that $c_p(v,w) \geq -\epsilon$ for all $(v,w) \in E_f$.*

For $\epsilon = 0$, this definition reduces to the condition 3 of Theorem 3, and, therefore, a 0-optimal circulation is a minimum cost circulation.

Definition 2 *$\epsilon(f) =$ minimum ϵ such that f is ϵ-optimal.*

We now show that approximate optimality is sufficient when the costs are integral.

Theorem 5 *If f is a circulation with $\epsilon(f) < \frac{1}{n}$ then f is optimal.*

Proof: $\epsilon(f) < \frac{1}{n}$ implies that, for some potentials p, $c_p(v,w) > -\frac{1}{n}$ for all $(v,w) \in E_f$. Therefore, any cycle Γ of G_f has reduced cost greater than $-|\Gamma|\frac{1}{n} \geq -1$. Since the costs are integral and the cost of any cycle is equal to its reduced cost, we obtain that any directed cycle of G_f has nonnegative cost. Theorem 3 implies that f is optimal. □

The following Theorem shows that the minimum mean cost $\mu(f)$ of all cycles in G_f represents how close the circulation f is from optimality.

Theorem 6 *For any circulation f, $\mu(f) = -\epsilon(f)$.*

Proof:

- $\mu(f) \geq -\epsilon(f)$.
  By definition, there exists p such that $c_p(v,w) \geq -\epsilon(f)$ for all $(v,w) \in E_f$. This implies that $c_p(\Gamma) \geq -\epsilon(f)|\Gamma|$ for any directed cycle Γ of G_f. But, for any directed cycle Γ, $c(\Gamma) = c_p(\Gamma)$. Therefore, dividing by $|\Gamma|$, we obtain that the mean cost of any directed cycle of G_f is at least $-\epsilon(f)$. Hence, $\mu(f) \geq -\epsilon(f)$.

- $-\mu(f) \geq \epsilon(f)$.
  To show that $-\mu(f) \geq \epsilon(f)$, we want to construct a function p such that $c_p(v,w) \geq \mu(f)$ for all $(v,w) \in E_f$. Let $\tilde{c}(v,w) = c(v,w) + (-\mu(f))$ for all $(v,w) \in E_f$. Notice that G_f has no negative cost cycle with respect to $\tilde{c}(.,.)$ since the mean cost of any directed cycle of G_f is increased by $-\mu(f)$. Next, add a new node s to G_f and also arcs from s to v for all $v \in V$. Let $\tilde{c}(s,v)$ be any value, say 0. Let $p(v)$ be the cost with respect to $\tilde{c}(.,.)$ of the shortest path from s to v in this augmented graph. Hence, for all $(v,w) \in E_f$, we have $p(w) \leq p(v) + \tilde{c}(v,w) = p(v) + c(v,w) - \mu(f)$ implying that $c_p(v,w) \geq \mu(f)$.

$\square$

We are now ready to analyze the algorithm. First, we show that, using $\epsilon(f)$ as a measure of near-optimality, the algorithm produces circulations which are closer and closer to optimal.

Theorem 7 *Let f be a circulation and let f' be the circulation obtained by canceling the minimum mean cost cycle Γ in E_f. Then $\epsilon(f) \geq \epsilon(f')$.*

Proof: By definition, there exists p such that

$$c_p(v,w) \geq -\epsilon(f) \tag{2}$$

for all $(v,w) \in E_f$. Moreover, for all $(v,w) \in \Gamma$, we have $c_p(v,w) = -\epsilon(f)$ since, otherwise, its mean cost would not be $-\epsilon(f)$. We claim that, for the same p, (2) holds for all $(v,w) \in E_{f'}$. Indeed, if $(v,w) \in E_{f'} \cap E_f$, (2) certainly holds. If $(v,w) \in E_{f'} \setminus E_f$ then (w,v) certainly belongs to Γ. Hence, $c_p(v,w) = -c_p(w,v) = \epsilon(f) \geq 0$ and (2) is also satisfied. $\square$

Next, we show that $\epsilon(f)$ decreases after a certain number of iterations.

Theorem 8 *Let f be any circulation and let f' be the circulation obtained by performing m iterations of the Golberg-Tarjan algorithm. Then*

$$\epsilon(f') \leq (1 - \frac{1}{n})\epsilon(f).$$

Proof: Let p be such that $c_p(v,w) \geq -\epsilon(f)$ for all $(v,w) \in E_f$. Let Γ_i be the cycle canceled at the ith iteration. Let k be the smallest integer such that there exists $(v,w) \in \Gamma_{k+1}$ with $c_p(v,w) \geq 0$. We know that canceling a cycle removes at least one arc with negative reduced cost from the residual graph and creates only arcs with positive reduced cost. Therefore $k \leq m$. Let f' be the flow obtained after k iterations. By Theorem 6, $-\epsilon(f')$ is equal to the mean cost of Γ_{k+1} which is:

$$\begin{aligned} \frac{\sum_{(v,w)\in\Gamma_{k+1}} c_p(v,w)}{l} &\geq \frac{-(l-1)}{l}\epsilon(f) \\ &= -(1-\frac{1}{l})\epsilon(f) \geq -(1-\frac{1}{n})\epsilon(f), \end{aligned}$$

where $l = |\Gamma_{k+1}|$. Therefore, by Theorem 7, after m iterations, $\epsilon(f)$ decreases by a factor of $(1 - \frac{1}{n})$. $\square$

Theorem 9 *Let* $C = \max_{(v,w)\in E} |c(v,w)|$. *Then the Goldberg-Tarjan algorithm finds a minimum cost circulation after canceling* $nm\log(nC)$ *cycles (*$\log = \log_e$*).*

Proof: The initial circulation $f = 0$ is certainly C-optimal since, for $p = 0$, we have $c_p(v,w) \geq -C$. Therefore, by Theorem 8, the circulation obtained after $nm \log nC$ iterations is $\epsilon-$optimal where:

$$\epsilon \leq \left(1 - \frac{1}{n}\right)^{n\log(nC)} C < e^{-\log(nC)}C = \frac{C}{nC} = \frac{1}{n},$$

where we have used the fact that $(1-\frac{1}{n})^n < e^{-1}$ for all $n > 0$. The resulting circulation is therefore optimal by Theorem 5. □

The overall running time of the Goldberg-Tarjan algorithm is therefore $O(n^2m^2\log(nC))$ since the minimum mean cost cycle can be obtained in $O(nm)$ time.

9 A Faster Cycle-Canceling Algorithm

We can improve upon the algorithm presented in the previous sections by using a more flexible selection of cycles for canceling and explicitly maintaining potentials to help identify cycles for canceling. The idea is to use the potentials we get from the minimum mean cost cycle to compute the edge costs $c_p(v,w)$ and then push flow along all cycles with only negative cost edges. The algorithm Cancel and Tighten is described below.

Cancel and Tighten:

1. Cancel: As long as there exists a cycle Γ in G_f with $c_p(v,w) < 0, \forall(v,w) \in \Gamma$ push as much flow as possible along Γ.
2. Tighten: Compute a minimum mean cost cycle in G_f and update p.

We now show that the Cancel step results in canceling at most m cycles each iteration and the flow it gives is $(1-1/n)\epsilon(f)$ optimal.

Theorem 10 *Let f be a circulation and let f' be the circulation obtained by performing the Cancel step. Then we cancel at most m cycles to get f' and*

$$\epsilon(f') \leq (1 - \frac{1}{n})\epsilon(f).$$

Proof: Let p be such that $c_p(v,w) \geq -\epsilon(f)$ for all $(v,w) \in E_f$. Let Γ be any cycle in f' and let l be the length of Γ. We know that canceling a cycle removes at least one arc with negative reduced cost from the residual graph and creates only arcs with positive reduced cost. Therefore we can cancel at most m cycles. Now $G_{f'}$

has no negative cycles therefore every cycle in $G_{f'}$ contains an edge (v,w) such that $c_p(v,w) \geq 0$. Hence the mean cost of Γ is at least:

$$\begin{aligned}\frac{\sum_{(v,w)\in\Gamma} c_p(v,w)}{l} &\geq \frac{-(l-1)}{l}\epsilon(f)\\ &= -(1-\frac{1}{l})\epsilon(f) \geq -(1-\frac{1}{n})\epsilon(f),\end{aligned}$$

□

The above result implies that the Cancel and Tighten procedure finds a minimum cost circulation in at most $n\log(nC)$ iterations (by an analysis which is a replication of Theorem 9). It also takes us $O(n)$ time to find a cycle on the admissible graph. This implies that each Cancel step takes $O(nm)$ steps due to the fact that we cancel at most m cycles and thus a running time of $O(nm)$ for one iteration of the Cancel and Tighten Algorithm. Therefore the overall running time of Cancel and Tighten is $O(n^2 m\log(nC))$ (i.e. an amortized time of $O(n)$ per cycle canceled). We can further improve this by using dynamic trees [14] to get an amortized time of $O(\log n)$ per cycle canceled and this results in an $O(nm\log n\log(nC))$ algorithm.

10 Alternative Analysis: A Strongly Polynomial Bound

In this section, we give another analysis of the algorithm. This analysis has the advantage of showing that the number of iterations is strongly polynomial, i.e. that it is polynomial in n and m and does not depend on C. The first strongly polynomial algorithm for the minimum cost circulation problem is due to Tardos [19].

Definition 3 *An arc $(v,w)\in E$ is $\epsilon-$fixed if $f(v,w)$ is the same for all $\epsilon-$optimal circulations.*

There exists a simple criterion for deciding whether an arc is $\epsilon-$fixed.

Theorem 11 *Let $\epsilon > 0$. Let f be a circulation and p be node potentials such that f is ϵ-optimal with respect to p. If $|c_p(v,w)| \geq 2n\epsilon$ then $(v,w)$is $\epsilon-$fixed.*

Proof: The proof is by contradiction. Let f' be an ϵ-optimal circulation for which $f'(v,w) \neq f(v,w)$. Assume that $|c_p(v,w)| \geq 2n\epsilon$. Without loss of generality, we can assume by antisymmetry that $c_p(v,w) \leq -2n\epsilon$. Hence $(v,w) \notin E_f$, i.e. $f(v,w) = u(v,w)$. This implies that $f'(v,w) < f(v,w)$. Let $E_< = \{(x,y)\in E : f'(x,y) < f(x,y)\}$.

Claim 12 *There exists a cycle Γ in $(V,E_<)$ that contains (v,w).*

Proof: Since $(v,w) \in E_<$, it is sufficient to prove the existence of a directed path from w to v in $(V, E_<)$. Let $S \subseteq V$ be the nodes reachable from w in $(V, E_<)$. Assume $v \notin S$. By flow conservation, we have

$$\sum_{x \in S, y \notin S} (f(x,y) - f'(x,y)) = \sum_{x \in S} \sum_{y \in V} (f(x,y) - f'(x,y)) = 0.$$

However, $f(v,w) - f'(v,w) > 0$, i.e. $f(w,v) - f'(w,v) < 0$, and by assumption $w \in S$ and $v \notin S$. Therefore, there must exists $x \in S$ and $y \notin S$ such that $f(x,y) - f'(x,y) > 0$, implying that $(x,y) \in E_<$. This contradicts the fact that $y \notin S$. □

By definition of $E_<$, we have that $E_< \subseteq E_{f'}$. Hence, the mean cost of Γ is at least $\mu(f') = -\epsilon(f') = -\epsilon$. On the other hand, the mean cost of Γ is ($l = |\Gamma|$):

$$\begin{aligned} \frac{c(\Gamma)}{l} &= \frac{c_p(\Gamma)}{l} = \frac{1}{l}\left(c_p(v,w) + \sum_{(x,y) \in \Gamma \setminus \{(v,w)\}} c_p(x,y)\right) \\ &\leq \frac{1}{l}(-2n\epsilon + (l-1)\epsilon) < \frac{1}{l}(-l\epsilon) = -\epsilon, \end{aligned}$$

a contradiction. □

Theorem 13 *The Goldberg-Tarjan algorithm terminates after $O(m^2 n \log n)$ iterations.*

Proof: If an arc becomes fixed during the execution of the algorithm, then it will remain fixed since $\epsilon(f)$ does not increase. We claim that, as long as the algorithm has not terminated, one additional arc becomes fixed after $O(mn \log n)$ iterations. Let f be the current circulation and let Γ be the first cycle canceled. After $mn \log(2n)$ iterations, we obtain a circulation f' with

$$\epsilon(f') \leq \left(1 - \frac{1}{n}\right)^{n \log(2n)} \epsilon(f) < e^{-\log(2n)} \epsilon(f) = \frac{\epsilon(f)}{2n}$$

by Theorem 10. Let p' be potentials for which f' satisfies the $\epsilon(f')$-optimality constraints. By definition of Γ,

$$-\epsilon(f) = \frac{c_{p'}(\Gamma)}{|\Gamma|}.$$

Hence,

$$\frac{c_{p'}(\Gamma)}{|\Gamma|} < -2n\epsilon(f').$$

Therefore, there exists $(v,w) \in \Gamma$ such that $|c_{p'}(v,w)| > -2n\epsilon(f')$. By the previous Theorem, (v,w) is $\epsilon(f')-$fixed. Moreover, (v,w) is not $\epsilon(f)-$fixed since canceling Γ increased the flow on (v,w). This proves that, after $mn \log(2n)$ iterations, one additional arc becomes fixed and therefore the algorithm terminates in $m^2 n \log(2n)$ iterations. □

Using the $O(mn)$ algorithm for the minimum mean cost cycle problem, we obtain a $O(m^3n^2 \log n)$ algorithm for the minimum cost circulation problem. Using the Cancel and Tighten improvement we obtain a running time of $O(m^2n^2 \log n)$. And if we implement Cancel and Tighten with the dynamic trees data structure we get a running time of $O(m^2 n \log^2 n)$.

The best known strongly polynomial algorithm for the minimum cost circulation problem is due to Orlin [18] and runs in $O(m \log n(m + n \log n)) = O(m^2 \log n + mn \log^2 n)$ time.

References

[1] R. K. Ahuja, T. L. Magnanti, and J. B. Orlin. Some recent advances in network flows. *SIAM Review*, 33:175–219, 1991.

[2] R. K. Ahuja, T. L. Magnanti, and J. B. Orlin. *Network flows: Theory, algorithms, and applications*. Prentice Hall, 1993.

[3] R. K. Ahuja, K. Mehlhorn, J. B. Orlin, and R. E. Tarjan. Faster algorithms for the shortest path problem. *Journal of the ACM*, 37:213–223, 1990.

[4] T. Cormen, C. Leiserson, and R. Rivest. *Introduction to Algorithms*. MIT Press, Cambridge, MA, 1990.

[5] R. Dial. Shortest path forest with topological ordering. *Communications of the ACM*, 12:632–633, 1969.

[6] E. W. Dijkstra. A note on two problems in connection with graphs. *Numer. Math.*, 1:269–271, 1959.

[7] J. Edmonds and R. M. Karp. Theoretical improvements in algorithmic efficiency for network flow problems. *Journal of the ACM*, 19:248–264, 1972.

[8] L. R. Ford and D. R. Fulkerson. Maximal flow through a network. *Canad. J. Math.*, 8:399–404, 1956.

[9] L. R. Ford and D. R. Fulkerson. *Flows in Networks*. Princeton Univ. Press, Princeton, NJ, 1963.

[10] M. L. Fredman and R. E. Tarjan. Fibonacci heaps and their uses in improved network optimization problems. *Journal of the ACM*, 34:569–615, 1987.

[11] H. N. Gabow. Scaling algorithms for network problems. *Journal of Computer and System Sciences*, 31:148–168, 1985.

[12] A. V. Goldberg, E. Tardos, and R. E. Tarjan. Network flow algorithms. In B. Korte, L. Lovasz, H. J. Promel, and A. Schrijver, editors, *Paths, flows, and VLSI-layout*, volume 9 of *Algorithms and Combinatorics*, pages 101–164. Springer-Verlag, 1990.

[13] A. V. Goldberg and R. E. Tarjan. A new approach to the maximum flow problem. *Journal of the ACM*, 35:921–940, 1988. Preliminary version in Proc. 18th Symposium on Theory of Computing, pages 136–146, 1986.

[14] A. V. Goldberg and R. E. Tarjan. Finding minimum-cost circulations by canceling negative cycles. *Journal of the ACM*, 36:873–886, 1989. Preliminary version in Proceedings of the 20th Annual ACM Symposium on Theory of Computing, pages 388–397, 1987.

[15] D. Goldfarb and J. Hao. A primal simplex algorithm that solves the maximum flow problem in at most nm pivots and $o(n^2m)$ time. *Mathematical Programming*, 47:353–365, 1990.

[16] D. Johnson. A priority queue in which initialization and queue operations take $O(\log\log D)$. *Math. Systems Theory*, 15:295–309, 1982.

[17] M. Klein. A primal method for minimal cost flows with applications to the assignment and transportation problems. *Management Science*, 14:205–220, 1967.

[18] J. B. Orlin. A faster strongly polynomial minimum cost flow algorithm. In *Proceedings of the 20th Annual ACM Symposium on Theory of Computing*, pages 377–387, 1988.

[19] É. Tardos. A strongly polynomial minimum cost circulation algorithm. *Combinatorica*, 5:247–255, 1985.

[20] R. E. Tarjan. Algorithms for maximum network flow. *Mathematical Programming Study*, 26:1–11, 1986.

18.415/6.854 Advanced Algorithms	November 1994

Approximation Algorithms

Lecturer: Michel X. Goemans

1 Introduction

Many of the optimization problems we would like to solve are NP-hard. There are several ways of coping with this apparent hardness. For most problems, there are straightforward exhaustive search algorithms, and one could try to speed up such an algorithm. Techniques which can be used include divide-and-conquer (or the refined branch-and-bound which allows to eliminate part of the search tree by computing, at every node, bounds on the optimum value), dynamic programming (which sometimes leads to pseudo-polynomial algorithms), cutting plane algorithms (in which one tries to refine a linear programming relaxation to better match the convex hull of integer solutions), randomization, etc. Instead of trying to obtain an optimum solution, we could also settle for a suboptimal solution. The latter approach refers to *heuristic* or "rule of thumb" methods. The most widely used such methods involve some sort of local search of the problem space, yielding a locally optimal solution. In fact, heuristic methods can also be applied to polynomially solvable problems for which existing algorithms are not "efficient" enough. A $\Theta(n^{10})$ algorithm (or even a linear time algorithm with a constant of 10^{100}), although efficient from a complexity point of view, will probably never get implemented because of its inherent inefficiency.

The drawback with heuristic algorithms is that it is difficult to compare them. Which is better, which is worse? For this purpose, several kinds of analyses have been introduced.

1. **Empirical analysis**. Here the heuristic is tested on a bunch of (hopefully meaningful) instances, but there is no guarantee that the behavior of the heuristic on these instances will be "typical" (what does it mean to be typical?).

2. **Average-case analysis**, dealing with the average-case behavior of a heuristic over some distribution of instances. The difficulty with this approach is that it can be difficult to find a distribution that matches the real-life data an algorithm will face. Probabilistic analyses tend to be quite hard.

3. **Worst-case analysis**. Here, one tries to evaluate the performance of the heuristic on the worst possible instance. Although this may be overly pessimistic, it gives a stronger guarantee about an algorithm's behavior. This is the type of analysis we will be considering in these notes.

To this end, we introduce the following definition:

Definition 1 *The performance guarantee of a heuristic algorithm for a minimization (maximization) problem is α if the algorithm is guaranteed to deliver a solution whose value is at most (at least) α times the optimal value.*

Definition 2 *An α-approximation algorithm is a polynomial time algorithm with a performance guarantee of α.*

Before presenting techniques to design and analyze approximation algorithms as well as specific approximation algorithms, we should first consider which performance guarantees are unlikely to be achievable.

2 Negative Results

For some hard optimization problems, it is possible to show a limit on the performance guarantee achievable in polynomial-time (assuming $P \neq NP$). A standard method for proving results of this form is to show that the existence of an α-approximation algorithm would allow you to solve some NP-complete decision problem in polynomial time. Even though NP-complete problems have equivalent complexity when exact solutions are desired, the reductions don't necessarily preserve approximability. The class of NP-complete problems can be subdivided according to how well a problem can be approximated.

As a first example, for the traveling salesman problem (given nonnegative lengths on the edges of a complete graph, find a tour — a closed walk visiting every vertex exactly once — of minimum total length), there is no α-approximation for any α unless $P = NP$. Indeed such an algorithm could be used to decide whether a graph (V, E) has an Hamiltonian cycle (simply give every edge e a length of 1 and every non-edge a very high or infinite length).

As another example, consider the bin packing problem. You have an integer T and weights $x_1, \dots, x_n \in [0, T]$, and you want to partition them into as few sets ("bins") as possible such that the sum of the weights in each set is at most T. It is NP-complete to decide whether k bins are sufficient.

In fact, there is no α-approximation algorithm for this problem, for any $\alpha < 3/2$. To see this, consider the partition problem: given weights $x_1, \dots, x_n \in [0, S]$ whose total sum is $2S$, is there a *partition* of the weights into two sets such that the sum in each set is S? This is the same as asking: are two bins sufficient when each bin has capacity S? If we had an α-approximation algorithm ($\alpha < 3/2$), we could solve the partition problem[1]. In general, if the problem of deciding whether a value is at most k is NP-complete then there is no α-approximation algorithm with $\alpha < \frac{k+1}{k}$ for the problem of minimizing the value unless $P = NP$.

[1] But wait, you exclaim — isn't there a polynomial-time approximation scheme for the bin packing problem? In fact, very good approximation algorithms can be obtained for this problem if you allow additive as well as multiplicative constants in your performance guarantee. It is our more restrictive model that makes this negative result possible. See Section 6 for more details.

Until 1992, pedestrian arguments such as this provided essentially the only known examples of non-approximability results. Then came a string of papers culminating in the result of Arora, Lund, Motwani, Sudan, and Szegedy[1] (based on the work of Arora and Safra [2]). They introduced a new characterization of NP in terms of probabilistically checkable proofs (PCP). In the new characterization, for any language L in NP, given the input x and a "proof" y of polynomial length in x, the verifier will toss $O(\log n)$ coins (where n is the size of x) to determine $k = O(1)$ positions or bits in the string y to probe; based on the values of these k bits, the verifier will answer "yes" or "no". The new characterization shows the existence of such a verifier V and a proof y such that (i) if $x \in L$ then there exists a proof y such that V outputs "yes" independently of the random bits, (ii) if $x \notin L$ then for every proof y, V outputs "no" with probability at least 0.5.

From this characterization, they deduce the following negative result for MAX 3SAT: given a set of clauses with at most 3 literals per clause, find an assignment maximizing the number of satisfied clauses. They showed the following:

Theorem 1 *For some $\epsilon > 0$, there is no $1 - \epsilon$-approximation algorithm*[2] *for MAX 3SAT unless $P = NP$.*

The proof goes as follows. Take any NP-complete language L. Consider the verifier V given by the characterization of Arora et al. The number of possible output of the $O(\log n)$ toin cosses is $S = 2^{O(\log n)}$ which is polynomial in n. Consider any outcome of these coin tosses. This gives k bits, say $i_1, \ldots, i_k$ to examine in the proof y. Based on these k bits, V will decide whether to answer yes or no. The condition that it answers yes can be expressed as a boolean formula on these k bits (with the Boolean variables being the bits of y). This formula can be expressed as the disjunction ("or") of conjunctions ("and") of k literals, one for each satisfying assignment. Equivalently, it can be written as the conjunction of disjunction of k literals (one for each rejecting assignment). Since k is $O(1)$, this latter k-SAT formula with at most 2^k clauses can be expressed as a 3-SAT formula with a constant number of clauses and variables (depending exponentially on k). (More precisely, using the classical reduction from SAT to 3-SAT, we would get a 3-SAT formula with at most $k2^k$ clauses and variables.) Call this constant number of clauses $M \leq k2^k = O(1)$. If $x \in L$, we know that there exists a proof y such that all SM clauses obtained by concatenating the clauses for each random outcome is satisfiable. However, if $x \notin L$, for any y, the clauses corresponding to at least half the possible random outcomes cannot be all satisfied. This means that if $x \notin L$, at least $S/2$ clauses cannot be satisfied. Thus either all SM clauses can be satisfied or at most $SM - \frac{S}{2}$ clauses can be satisfied. If we had an approximation algorithm with performance guarantee better than $1 - \epsilon$ where $\epsilon = \frac{1}{2M}$ we could decide whether $x \in L$ or not, in polynomial

[2] In our definition of approximation algorithm, the performance guarantee is less than 1 for *maximization* problems.

time (since our construction can be carried out in polynomial time). This proves the theorem.

The above theorem implies a host of negative results, by considering the complexity class MAX-SNP. defined by Papadimitriou and Yannakakis [21].

Corollary 2 *For any MAX-SNP-complete problem, there is an absolute constant* $\epsilon > 0$ *such that there is no* $(1-\epsilon)$*-approximation algorithm unless* $P = NP$.

The class of MAX-SNP problems is defined in the next section and the corollary is derived there. We first give some examples of problems that are complete for MAX-SNP.

1. MAX 2-SAT: Given a set of clauses with one or two literals each, find an assignment that maximizes the number of satisfied clauses.

2. MAX k-SAT: Same as MAX 2-SAT, but each clause has up to k literals.

3. MAX CUT: Find a subset of the vertices of a graph that maximizes the number of edges crossing the associated cut.

4. The Travelling Salesman Problem with the triangle inequality is MAX-SNP-hard. (There is a technical snag here: MAX-SNP contains only maximization problems, whereas TSP is a minimization problem.)

2.1 MAX-SNP Complete Problems

Let's consider an alternative definition of NP due to Fagin [9]. NP, instead of being defined computationally, is defined as a set of predicates or functions on structures G:

$$\exists S \forall x \exists y \ \psi(x, y, G, S)$$

where ψ is a quantifier free expression. Here S corresponds to the witness or the proof.

Consider for example the problem SAT. We are given a set of clauses, where each clause is the disjunction of literals. (A literal is a variable or its negation.) We want to know if there is a way to set the variables true or false, such that every clause is true. Thus here G is the set of clauses, S is the set of literals to be set to true, x represents the clauses, y represents the literals, and

$$\psi(x, y, G, S) = (P(G, y, x) \wedge y \in S) \vee (N(G, y, x) \wedge y \notin S)$$

Where $P(G, y, x)$ is true iff y appears positively in clause x, and $N(G, y, x)$ is true iff y appears negated in clause x.

Strict NP is the set of problems in NP that can be defined without the the third quantifier:

$$\exists S \forall x \; \psi(x, G, S)$$

where ψ is quantifier free.

An example is 3-SAT, the version of SAT in which every clause has at most 3 literals. Here $x = (x_1, x_2, x_3)$ (all possible combinations of three variables) and G is the set of possible clauses; for example $(x_1 \vee x_2 \vee x_3)$, $(\overline{x_1} \vee x_2 \vee \overline{x_3})$, and so forth. Then ψ is a huge conjunction of statements of the form: If (x_1, x_2, x_3) appears as $(\overline{x_1} \vee x_2 \vee \overline{x_3})$, then $x_1 \notin S \vee x_2 \in S \vee x_3 \notin S$.

Instead of asking that for each x we get $\psi(x, G, S)$, we can ask that the number of x's for which $\psi(x, G, S)$ is true be maximized:

$$\max_S |\{x : \psi(x, G, S)\}|$$

In this way, we can derive an optimization problem from an SNP predicate. These maximization problems comprise the class MAX-SNP (MAXimization, Strict NP) defined by Papadimitriou and Yannakakis [21]. Thus, MAX 3SAT is in MAX-SNP.

Papadimitriou and Yannakakis then introduce an *L-reduction* (L for linear), which preserverses approximability. In particular, if P L-reduces to P', and there exists an α-approximation algorithm for P', then there exists a $\gamma\alpha$-approximation algorithm for P, where γ is some constant depending on the reduction.

Given L-reductions, we can define MAX-SNP complete problems to be those $P \in$ MAX-SNP for which $Q \leq_L P$ for all $Q \in$ MAX-SNP. Some examples of MAX-SNP complete problems are MAX 3SAT, MAX 2SAT (and in fact MAX kSAT for any fixed $k > 1$), and MAX-CUT. The fact that MAX 3SAT is MAX-SNP-complete and Theorem 1 implies the corollary mentioned previously.

For MAX 3SAT, ε in the statement of Theorem 1 can be chosen can be set to 1/74 (Bellare and Sudan [5]).

Minimization problems may not be able to be expressed so that they are in MAX-SNP, but they can still be MAX-SNP hard. Examples of such problems are:

- TSP with edge weights 1 and 2 (i.e., $d(i, j) \in \{1, 2\}$ for all i, j). In this case, there exists a 7/6-approximation algorithm due to Papadimitriou and Yannakakis.

- Steiner tree with edge weights 1 and 2.

- Minimum Vertex Cover. (Given a graph $G = (V, E)$, a vertex cover is a set $S \subseteq V$ such that $(u, v) \in E \Rightarrow u \in S$ or $v \in S$.)

3 The Design of Approximation Algorithms

We now look at key ideas in the design and analysis of approximation algorithms. We will concentrate on minimization problems, but the ideas apply equally well to maximization problems. Since we are interested in the minimization case, we know that an α-approximation algorithm H has cost $C_H \leq \alpha C_{OPT}$ where C_{OPT} is the cost of the optimal solution, and $\alpha \geq 1$.

Relating C_H to C_{OPT} directly can be difficult. One reason is that for NP-hard problems, the optimum solution is not well characterized. So instead we can relate the two in two steps:

1. $LB \leq C_{OPT}$
2. $C_H \leq \alpha LB$

Here LB is a lower bound on the optimal solution.

3.1 Relating to Optimum Directly

This is not always necessary, however. One algorithm whose solution is easy to relate directly to the optimal solution is Christofides' [6] algorithm for the TSP with the triangle inequality ($d(i,j) + d(j,k) \leq d(i,k)$ for all i,j,k). This is a $\frac{3}{2}$-approximation algorithm, and is the best known for this problem. The algorithm is as follows:

1. Compute the minimum spanning tree T of the graph $G = (V, E)$.
2. Let O be the odd degree vertices in T. One can prove that $|O|$ is even.
3. Compute a minimum cost perfect matching M on the graph induced by O.
4. Add the edges in M to E. Now the degree of every vertex of G is even. Therefore G has an Eulerian tour. Trace the tour, and take shortcuts when the same vertex is reached twice. This cannot increase the cost since the triangle inequality holds.

We claim that $Z_C \leq \frac{3}{2} Z_{TSP}$, where Z_C is the cost of the tour produced by Christofides' algorithm, and Z_{TSP} is the cost of the optimal solution. The proof is easy:

$$\frac{Z_C}{Z_{TSP}} \leq \frac{Z_T + Z_M}{Z_{TSP}} = \frac{Z_T}{Z_{TSP}} + \frac{Z_M}{Z_{TSP}} \leq 1 + \frac{1}{2} = \frac{3}{2}.$$

Here Z_T is the cost of the minimum spanning tree and Z_M is the cost of the matching. Clearly $Z_T \leq Z_{TSP}$, since if we delete an edge of the optimal tour a spanning tree results, and the cost of the minimum spanning tree is at most the cost of that tree. Therefore $\frac{Z_T}{Z_{TSP}} \leq 1$.

To show $\frac{Z_M}{Z_{TSP}} \leq \frac{1}{2}$, consider the optimal tour visiting only the vertices in O. Clearly by the triangle inequality this is of length no more than Z_{TSP}. There are an even number of vertices in this tour, and so also an even number of edges, and the tour defines two disjoint matchings on the graph induced by O. At least one of these has cost $\leq \frac{1}{2} Z_{TSP}$, and the cost of Z_M is no more than this.

3.2 Using Lower Bounds

Let

$$C_{OPT} = \min_{x \in S} f(x).$$

A lower bound on C_{OPT} can be obtained by a so-called *relaxation*. Consider a related optimization problem $LB = \min_{x \in R} g(x)$. Then LB is a lower bound on C_{OPT} (and the optimization problem is called a relaxation of the original problem) if the following conditions hold:

$$S \subseteq R \tag{1}$$

$$\text{g}(x) \leq f(x) \text{ for all } x \in S. \tag{2}$$

Indeed these conditions imply

$$LB = \min_{x \in R} g(x) \leq \min_{x \in S} f(x) = C_{OPT}.$$

Most classical relaxations are obtained by using linear programming. However, there are limitations as to how good an approximation LP can produce. We next show how to use a linear programming relaxation to get a 2-approximation algorithm for Vertex Cover, and show that this particular LP relaxation cannot give a better approximation algorithm.

3.3 An LP Relaxation for Minimum Weight Vertex Cover (VC)

A vertex cover U in a graph $G = (V, E)$ is a subset of vertices such that every edge is incident to at least one vertex in U. The vertex cover problem is defined as follows: Given a graph $G = (V, E)$ and weight $w(v) \geq 0$ for each vertex v, find a vertex cover $U \subseteq V$ minimizing $w(U) = \sum_{v \in U} w(v)$. (Note that the problem in which nonpositive weight vertices are allowed can be handled by including all such vertices in the cover, deleting them and the incident edges, and finding a minimum weight cover of the remaining graph. Although this reduction preserves optimality, it does not maintain approximability; consider, for example, the case in which the optimum vertex cover has 0 cost (or even negative cost).)

This can be expressed as an integer program as follows. Let $x(v) = 1$ if $v \in U$ and $x(v) = 0$ otherwise. Then

$$C_{OPT} = \min_{x \in S} \sum_{v \in V} w(v)x(v)$$

where

$$S = \left\{ x \in R^{|V|} : \begin{array}{ll} x(v) + x(w) \geq 1 & \forall (v,w) \in E \\ x(v) \in \{0,1\} & \forall v \in V \end{array} \right\}.$$

We now relax S, turning the problem into a linear program:

$$LB = \min_{x \in R} \sum_{v \in V} w(v)x(v)$$

$$R = \left\{ x \in R^{|V|} : \begin{array}{ll} x(v) + x(w) \geq 1 & \forall (v,w) \in E \\ x(v) \geq 0 & \forall v \in V \end{array} \right\}.$$

In order to show that R is a relaxation, we must show that it satisfies conditions 1 and 2. Condition 1 clearly holds, as $0, 1 \geq 0$. Furthermore, condition 2 also holds, since the objective function is unchanged. Thus, we can conclude that $LB \leq C_{OPT}$, and we can prove that an algorithm H is an α-approximation algorithm for VC by showing $C_H \leq \alpha LB$.

The limitation of this relaxation is that there are instances where $LB \sim \frac{1}{2}C_{OPT}$. This implies that it cannot be used to show any $\alpha < 2$, since if we could then H would give a better answer than the optimum. One such instance is K_n: the complete graph on n vertices, with all vertices weight 1. All the nodes but one must be in the cover (otherwise there will be an edge between two that are not, with neither in the cover set). Thus, $C_{OPT} = n - 1$. The relaxation, on the other hand, can have $x(v) = \frac{1}{2}, \forall v \in V$. Thus, $LB \leq \frac{n}{2}$, which means $LB \sim \frac{1}{2}C_{OPT}$.

3.4 How to use Relaxations

There are two main techniques to derive an approximately optimal solution from a solution to the relaxed problem.

1. **Rounding**
   Find an optimal solution x^* to the relaxation. Round $x^* \in R$ to an element $x' \in S$. Then prove $f(x') \leq \alpha g(x^*)$ which implies

   $$f(x') \leq \alpha LB \leq \alpha C_{OPT}$$

   Often randomization is helpful, as we shall see in later sections. In this case $x^* \in R$ is randomly rounded to some element $x' \in S$ so that $E[f(x')] \leq \alpha g(x^*)$. These algorithms can sometimes be derandomized, in which case one finds an x'' such that $f(x'') \leq E[f(x')]$.

2. **Primal-Dual**
   Consider some weak dual of the relaxation:
   $$\max\{h(y) : y \in D\} \leq \min\{g(x) : x \in R\}$$
   Construct $x \in S$ from $y \in D$ such that
   $$f(x) \leq \alpha h(y) \leq \alpha h(y_{\max}) \leq \alpha g(x_{\min}) \leq \alpha C_{OPT}.$$
   Notice that y can be any element of D, not necessarily an optimal solution to the dual.

We now illustrate these techniques on the minimum weight vertex cover problem.

3.4.1 Rounding applied to VC

This is due to Hochbaum [16]. Let x^* be the optimal solution of the LP relaxation. Let
$$U = \{v \in V : x^*(v) \geq \frac{1}{2}\}$$
We claim U is a 2-approximation of the minimum weight VC. Clearly U is a vertex cover, because for $(u,v) \in E$ we have $x^*(u) + x^*(v) \geq 1$, which implies $x^*(u) \geq 1/2$ or $x^*(v) \geq 1/2$. Also
$$\sum_{v \in U} w(v) \leq \sum_{v \in V} w(v) 2x^*(v) = 2LB$$
since $2x^*(v) \geq 1$ for all $v \in U$.

3.4.2 Primal-Dual applied to VC

This is due to Bar-Yehuda and Even [4]. First formulate the dual problem. Let $y \in R^{|E|}$; the elements of y are $y(e)$ for $e = (u,v) \in E$. The dual is:
$$\max \sum_{e \in E} y(e)$$
$$\sum_{u:e=(v,u) \in E} y(e) \leq w(v) \quad \forall v \in V \tag{3}$$
$$y(e) \geq 0 \quad \forall e \in E. \tag{4}$$

Initialize C (the vertex cover) to the empty set, $y = 0$ and $F = E$. The algorithm proceeds by repeating the following two steps while $F \neq \emptyset$:

1. Choose some $e = (u,v) \in F$. Increase $y(e)$ as much as possible, until inequality (3) becomes tight for u or v. Assume WLOG it is tight for u.

2. Add u to C and remove all edges incident to u from F.

Clearly C is a vertex cover. Furthermore
$$\sum_{v \in C} w(v) = \sum_{v \in C} \sum_{u:e=(v,u) \in E} y(e) = \sum_{e=(v,u) \in E} |C \cap \{v,u\}| y(e) \leq \sum_{e \in E} 2y(e) \leq 2LB.$$

4 The Min-Cost Perfect Matching Problem

In this section, we illustrate the power of the primal-dual technique to derive approximation algorithms. We consider the following problem.

Definition 3 *The* Minimum-Cost Perfect Matching Problem (MCPMP) *is as follows: Given a complete graph $G = (V, E)$ with $|V|$ even and a nonnegative cost function $c_e \geq 0$ on the edges $e \in E$, find a perfect matching M such that the cost $c(M)$ is minimized, where $c(M) = \sum_{e \in M} c_e$.*

The first polynomial time algorithm for this problem was given by Edmonds [8] and has a running time of $O(n^4)$ where $n = |V|$. To date, the fastest strongly polynomial time algorithm is due to Gabow [10] and has a running time of $O(n(m + n \lg n))$ where $m = |E|$. For dense graphs, $m = \Theta(n^2)$, this algorithm gives a running time of $O(n^3)$. The best weakly polynomial algorithm is due to Gabow and Tarjan [12] and runs in time $O(m\sqrt{n\alpha(m,n)\log n}\log nC)$ where C is a bound on the costs c_e. For dense graphs with $C = O(n)$, this bound gives an $O^*(n^{2.5})$ running time.

As you might suspect from these bounds, the algorithms involved are fairly complicated. Also, these algorithms are too slow for many of the instances of the problem that arise in practice. In this section, we discuss an approximation algorithm by Goemans and Williamson [13] that runs in time $O(n^2 \lg n)$. (This bound has recently been improved by Gabow, Goemans and Williamson [11] to $O(n(n + \sqrt{m \lg \lg n}))$.) Although MCPMP itself is in PTIME, this algorithm is sufficiently general to give approximations for many NP-hard problems as well.

The algorithm of Goemans and Williamson is a 2-approximation algorithm — it outputs a perfect matching with cost not more than a factor of 2 larger than the cost of a minimum-cost perfect matching. This algorithm requires that the costs c_e make up a metric, that is, c_e must respect the triangle inequality: $c_{ij} + c_{jk} \geq c_{ik}$ for all triples i, j, k of vertices.

4.1 A linear programming formulation

The basic idea used in the 2-approximation algorithm of Goemans and Williamson is linear programming and duality. The min-cost perfect matching problem can be formulated as a linear program. The algorithm does not directly solve the linear program, but during its operation, it can compute a feasible solution to the dual program. This dual feasible solution actually certifies the factor of 2 approximation. Before writing down the linear program, we start with an observation.

Consider a matching M and a set $S \subset V$ of vertices with $|S|$ odd. If M is a perfect matching, then since $|S|$ is odd, there must be some edge in the matching that has one endpoint inside S and the other outside. In other symbols, let $\delta(S)$ be the set of edges in E with exactly one endpoint in S; if M is a perfect matching and $|S|$ is odd, then $M \cap \delta(S) \neq \emptyset$.

With this observation, we can now formulate MCPMP as a linear program:

$$
\begin{aligned}
Z = \text{Min} \sum_{e \in E} c_e x_e & \\
\text{subject to:} \quad \sum_{e \in \delta(S)} x_e \geq 1 & \qquad \text{for all } S \subset V \text{ with } |S| \text{ odd} \\
x_e \geq 0 & \qquad \text{for all } e \in E.
\end{aligned}
$$

We can now see that the value Z of this linear program is a lower bound on the cost of any perfect matching. In particular, for any perfect matching M, we let

$$
x_e = \begin{cases} 1 & \text{if } e \in M; \\ 0 & \text{otherwise.} \end{cases}
$$

Clearly, this assignment is a feasible solution to the linear program, so we know that $Z \leq c(M)$. This bound also applies to a minimum-cost perfect matching M^*, so we have $Z \leq c(M^*)$.

Note that this is a huge linear program having one constraint for each $S \subset V$ of odd cardinality. Though it is too large to be solved in polynomial time by any of the linear programming algorithms we have seen, the ellipsoid method can actually solve this program in polynomial time. We do not consider this solution technique; rather we let the linear program and its dual serve as a tool for developing and analyzing the algorithm.

We now consider the dual linear program:

$$
\begin{aligned}
Z = \text{Max} \sum_{\substack{S \subset V, \\ |S| \text{ odd}}} y_S & \\
\text{subject to:} \quad \sum_{\substack{S \subset V, \\ e \in \delta(S)}} y_S \leq c_e & \qquad \text{for all } e \in E \\
y_S \geq 0 & \qquad \text{for all } S \subset V \text{ with } |S| \text{ odd.}
\end{aligned}
$$

Note that by strong duality, the value Z of this dual linear program is the same as the value Z of the primal program.

This dual linear program is used to verify that the perfect matching output by the algorithm is actually within a factor of 2 of optimal. The algorithm outputs two things:

1. a perfect matching M', and

2. a dual feasible solution y such that

$$
c(M') \leq 2 \sum_{\substack{S \subset V, \\ |S| \text{ odd}}} y_S.
$$

Since y is dual feasible, we know that

$$\sum_{\substack{S \subset V, \\ |S| \text{ odd}}} y_S \leq Z \leq c(M)$$

where M is any perfect matching. Thus we have

$$c(M') \leq 2Z \leq 2c(M^*)$$

where M^* is a min-cost perfect matching. The algorithm is therefore (given that it runs in polynomial time) a 2-approximation algorithm for MCPMP.

To be precise, the algorithm need not actually output the dual feasible solution y — it is only needed as an analysis tool to prove the factor of 2 approximation bound. In spite of the fact that there are an exponential number of y_S variables, the algorithm could actually compute and output the y_S values since it turns out that only a polynomial number of them are non-zero. When we finally get to exhibiting the algorithm, we will include the computation of the y_S values.

4.2 From forest to perfect matching

Rather than directly compute the perfect matching M', the algorithm first computes a forest F' from which M' can be derived. In the forest F', all components have even size, and furthermore, F' is edge-minimal in the sense that if any edge of F' is removed, then the resulting forest has an odd size component. Additionally, the cost of F' is bounded by twice the value of the dual feasible solution; that is,

$$c(F') \leq 2 \sum_{\substack{S \subset V, \\ |S| \text{ odd}}} y_S.$$

We now show how to convert F' into a perfect matching M' such that $c(M') \leq c(F')$. The idea is as follows. Starting from the forest F', consider any vertex v with degree at least 3. Take two edges (u, v) and (v, w); remove them and replace them with the single edge (u, w). Since the edge costs obey the triangle inequality, the resulting forest must have a cost not more than $c(F')$. Thus, if we can iterate on this operation until all vertices have degree 1, then we have our perfect matching M'.

The only thing that can get in the way of the operation just described is a vertex of degree 2. Fortunately, we can show that all vertices of F' have odd degree. Notice then that this property is preserved by the basic operation we are using. (As a direct consequence, the property that all components are even is also preserved.) Therefore, if all vertices of F' have odd degree, we can iteration the basic operation to produce a perfect matching M' such that $c(M') \leq c(F')$. Notice that M' is produced after $O(n)$ iterations.

Lemma 3 *All vertices of F' have odd degree.*

Proof: Suppose there is a vertex v with even degree, and let v be in component A of F'. Removing v and all its incident edges partitions A into an even number k of smaller components $A_1, A_2, \ldots, A_k$. If all k of these components have odd size, then it must be the case that A has odd size. But we know that A has even size — all components of F' have even size — so there must be a component A_i with even size. Let v_i denote the vertex in A_i such that (v, v_i) is an edge of F'. Now if we start from F' and remove the edge (v, v_i), we separate A into two even size components. This contradicts the edge-minimality of F'. □

4.3 The algorithm

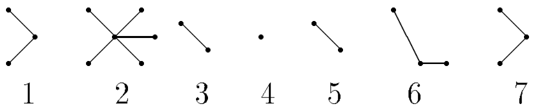
The algorithm must now output an edge-minimal forest F' with even size components and be able to compute a dual feasible solution y such that $c(F') \leq 2 \sum y_S$.

At the highest level, the algorithm is:

1. Start with $F = \emptyset$.
2. As long as there exists an odd size component of F, add an edge between two components (at least one of which has odd size).

Note that the set of components of F is initially just the set of vertices V.

The choice of edges is guided by the dual linear program shown earlier. We start with all the dual variables equal to zero; $y_S = 0$. Suppose at some point in the execution we have a forest F as shown below and a dual solution y. Look at the



components S of odd cardinality (components 1, 4, 6 and 7, in this case). For these components, increase y_S by some δ, leaving all other values of y_S unchanged. That is,

$$y_S \leftarrow \begin{cases} y_S + \delta & \text{if } S \text{ is an odd size component of } F \\ y_S & \text{otherwise.} \end{cases}$$

Make δ as large as possible while keeping y_S dual feasible. By doing this, we make the constraint on some edge e tight; for some e the constraint

$$\sum_{\substack{S \subset V, \\ e \in \delta(S)}} y_S \leq c_e$$

becomes

$$\sum_{\substack{S \subset V, \\ e \in \delta(S)}} y_S = c_e.$$

This is the edge e that we add to F. (If more than one edge constraint becomes tight simultaneously, then just arbitrarily pick one of the edges to add.)

We now state the algorithm to compute F'. The steps that compute the dual feasible solution y are commented out by placing the text in curly braces.

$F \leftarrow \emptyset$
$\mathcal{C} \leftarrow \{\{i\} \mid i \in V\}$ $\quad\{\textit{The components of } F\}$
$\{\textit{Let } y_S \leftarrow 0 \textit{ for all } S \textit{ with } |S| \textit{ odd.}\}$
$\forall i \in V$ do $d(i) \leftarrow 0$ $\quad\{d(i) = \sum_{S \ni i} y_S\}$
while $\exists C \in \mathcal{C}$ with $|C|$ odd do
  Find edge $e = (i, j)$ such that $i \in C_p$, $j \in C_q$, $p \neq q$
    which minimizes $\delta = \frac{c_e - d(i) - d(j)}{\lambda(C_p) + \lambda(C_q)}$
    where $\lambda(C) = \begin{cases} 1 & \text{if } |C| \text{ is odd} \\ 0 & \text{otherwise} \end{cases}$ (*i.e.*, the parity of C).
  $F \leftarrow F \cup \{e\}$
  $\forall C \in \mathcal{C}$ with $|C|$ odd do
    $\forall i \in C$ do $d(i) \leftarrow d(i) + \delta$
    $\{\textit{Let } y_C \leftarrow y_C + \delta.\}$
  $\mathcal{C} \leftarrow \mathcal{C} \setminus \{C_p, C_q\} \cup \{C_p \cup C_q\}$
$F' \leftarrow$ edge-minimal F

4.4 Analysis of the algorithm

Lemma 4 *The values of the variables y_S computed by the above algorithm constitute a dual feasible solution.*

Proof: We show this by induction on the *while* loop. Specifically, we show that at the start of each iteration, the values of the variables y_S are feasible for the dual linear program. We want to show that for each edge $e \in E$,

$$\sum_{\substack{S \subset V, \\ e \in \delta(S)}} y_S \leq c_e.$$

The base case is trivial since all variables y_S are initialized to zero and the cost function c_e is nonnegative. Now consider an edge $e' = (i', j')$ and an iteration. There are two cases to consider.

In the first case, suppose both i' and j' are in the same component at the start of the iteration. In this case, there is no component $C \in \mathcal{C}$ for which $e' \in \delta(C)$. Therefore, since the only way a variable y_S gets increased is when S is a component, none of the variables y_S with $e' \in \delta(S)$ get increased at this iteration. By the induction hypothesis, we assume the iteration starts with

$$\sum_{\substack{S \subset V, \\ e' \in \delta(S)}} y_S \leq c'_e,$$

and therefore, since the left-hand-side of this inequality does not change during the iteration, this inequality is also satisfied at the start of the next iteration.

In the second case, suppose i' and j' are in different components; $i' \in C_{p'}$ and $j' \in C_{q'}$ at the start of the iteration. In this case, we can write

$$\begin{aligned} \sum_{\substack{S \subset V, \\ e' \in \delta(S)}} y_S &= \sum_{\substack{S \subset V, \\ i' \in S}} y_S + \sum_{\substack{S \subset V, \\ j' \in S}} y_S \\ &= d(i') + d(j'), \end{aligned}$$

where $d(i)$ is defined by

$$d(i) = \sum_{\substack{S \subset V, \\ i \in S}} y_S.$$

The equality follows because since i' and j' are in different components, if S contains both i' and j', then S is not and never has been a component; hence, for such a set S, we have $y_S = 0$. We know that during this iteration $d(i')$ will be incremented by δ if and only if $y_{C_{p'}}$ is incremented by δ, and this occurs if and only if $\lambda(C_{p'}) = 1$. Let $d'(i')$ and $d'(j')$ be the new values of $d(i')$ and $d(j')$ after this iteration. Then we have

$$\begin{aligned} d'(i') &= d(i') + \delta\lambda(C_{p'}), \quad \text{and} \\ d'(j') &= d(j') + \delta\lambda(C_{q'}). \end{aligned}$$

Now, by the way δ is chosen, we know that

$$\delta \leq \frac{c_{e'} - d(i') - d(j')}{\lambda(C_{p'}) + \lambda(C_{q'})}.$$

Thus, at the beginning of the next iteration we have

$$\begin{aligned} \sum_{\substack{S \subset V, \\ e' \in \delta(S)}} y_S &= d'(i') + d'(j') \\ &= d(i') + \delta\lambda(C_{p'}) + d(j') + \delta\lambda(C_{q'}) \\ &= d(i') + d(j') + \delta[\lambda(C_{p'}) + \lambda(C_{q'})] \\ &\leq d(i') + d(j') + \frac{c_{e'} - d(i') - d(j')}{\lambda(C_{p'}) + \lambda(C_{q'})}[\lambda(C_{p'}) + \lambda(C_{q'})] \\ &= c_{e'}. \end{aligned}$$

Finally, for completeness sake, we note that the constraint $y_S \geq 0$ is satisfied because $y_S = 0$ initially and $\delta \geq 0$. $\square$

As a final observation, we note that when the algorithm selects an edge e', the corresponding constraint in the dual linear program becomes tight. This means that for all edges $e \in F$, we have

$$\sum_{\substack{S \subset V, \\ e \in \delta(S)}} y_S = c_e.$$

4.5 A simulated run of the algorithm

Since the algorithm as given can be difficult to visualize, here is an example of how it would execute. See Figure 1.

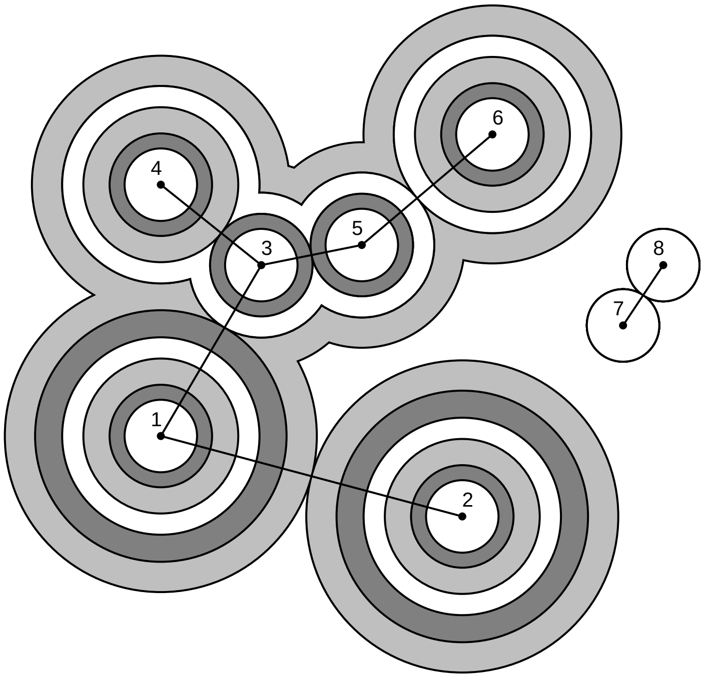


Figure 1: A sample run of the algorithm. The various values of $d(i)$ are indicated by the shaded regions around the components.

We'll assume a Euclidean distance metric to ease visualization. Now, initially, all points (1 through 8) are in separate components, and $d(i)$ is 0 for all i. Since the metric is Euclidean distance, the first edge to be found will be $(7, 8)$. Since both components are of odd size, δ will be half the distance between them $((c_e - 0 - 0)/(1 + 1))$. Since, in fact, *all* components are of odd size, every $d(i)$ will be increased by this amount, as indicated by the innermost white circle around each point. The set $\{7, 8\}$ now becomes a single component of even size.

In general, we can see the next edge to be chosen by finding the pair of components whose boundaries in the picture can be most easily made to touch. Thus, the next edge is $(3, 5)$, since the boundaries of their regions are closest, and the resulting values of $d(i)$ are represented by the dark gray bands around points 1 through 6. Note that the component $\{7, 8\}$ does not increase its $d(i)$ values since it is of even size.

We continue in this way, expanding the "moats" around odd-sized components until all components are of even size. Since there is an even number of vertices and we always expand odd-sized components, we are guaranteed to reach such a point.

4.6 Final Steps of Algorithm and Proof of Correctness

Let $F' = \{e \in F : F \setminus \{e\}$ has an odd sized component$\}$. We will show that the F' so obtained is a forest with components of even cardinality and that it is edge minimal with respect to this property. It is obvious that F' is a forest since $F' \subset F$ and F is a forest. We will also show that the cost of this forest (F'), is less than or equal to twice the dual solution. In section 4.2 we showed how to build a matching from this forest with the cost of the matching less than or equal to the cost of the forest. Thus, this gives us a 2-approximation algorithm for matching. As an example see the figure given below.

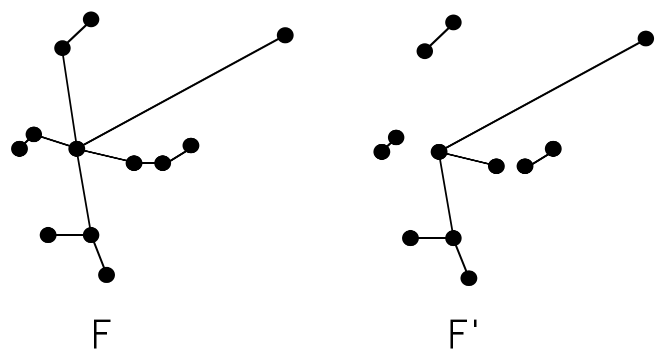


Figure 2: Example showing how to get F' from F.

Theorem 5 *Let $F' = \{e \in F : F \setminus \{e\}$ has an odd sized component$\}$. Then*

1. *every component of F' has an even number of vertices and F' is edge minimal with respect to this property..*
2. $\sum_{e \in F'} c_e \leq 2 \sum_S y_S$.

Proof:

Let us first show that every component of F' has an even number of vertices. Suppose not. Then consider the components of F. Every component of F has an even number of vertices by design of the algorithm. Consider a component of F' which has an odd number of vertices and let us denote it as T_i'. Let T_i be the component that T_i' belongs to in F. Let $N_1, \ldots, N_j$ be the components of F within T_i obtained by removing T_i' (see figure 3). T_i has an even number of vertices. N_k with $1 \leq k \leq j$ has an even number of vertices because, otherwise, the edge from N_k to T_i' would belong to F' by definition. But this implies that T_i' is even, which is a contradiction.

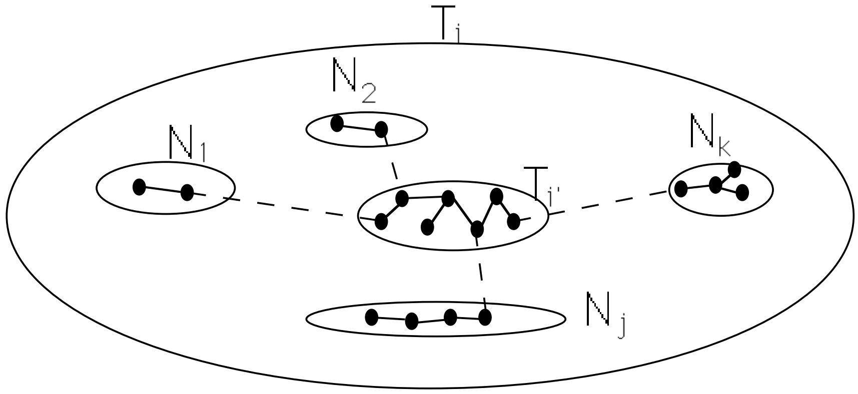


Figure 3: Every component of F' has an even # of vertices.

A simple proof by contradiction shows that F' is edge minimal. Suppose F' is not edge minimal. Then there is an edge or set of edges which can be removed which leave even sized components. Consider one such edge e. It falls into one of two categories:

1. Its removal divides a component into two even sized components. But this means that e was already removed by the definition of F'.

2. Its removal divides a component into two odd sized components. Despite the fact that other edges may be removed, as well, an two odd sized component will remain in the forest. Thus, e cannot be removed.

Now let us prove the second portion of the theorem. In what follows, though we do not explicitly notate it, when we refer to a set S of vertices, we mean a set S of vertices with $|S|$ odd. We observe that by the choice of the edges e in F, we have

$$c_e = \sum_{S: e \in \delta(S)} y_S$$

for all $e \in F$. Thus,

$$\begin{aligned} \sum_{e \in F'} c_e &= \sum_{e \in F'} \sum_{S: e \in \delta(S)} y_S \\ &= \sum_S y_S |F' \cap \delta(S)| \end{aligned}$$

Thus we need to show,

$$\sum_S y_S |F' \cap \delta(S)| \leq 2 \sum_S y_S$$

We will show this by induction. In what follows, bear in mind that F' is what we have, at the end of the algorithm. We will show the above relation holds at every iteration.

Initially, $y_S = 0$. Thus the LHS and RHS are both zero. Thus, this is true initially.

Let us suppose it is true at any intermediate stage in the algorithm. We will show that it will remain true in the next iteration. From one iteration to the next the only y_S that change are those with $C \in \mathcal{C}$ with $|C|$ odd. Thus if we show the increase in the LHS is less than the RHS we are done. i.e.

$$\delta \sum_{C \in \mathcal{C}, |C| \text{ odd}} |F' \cap \delta(C)| \leq 2\delta |\{C \in \mathcal{C}, |C| \text{ odd}\}|$$

or

$$\sum_{C \in \mathcal{C}, |C| \text{ odd}} |F' \cap \delta(C)| \leq 2 |\{C \in \mathcal{C}, |C| \text{ odd}\}|$$

Now, define a graph H with $C \in \mathcal{C}$ as vertices, with an edge between C_p and C_q if there exists an edge in $F' \cap \{\delta(C_p) \cap \delta(C_q)\}$. We can partition these vertices into two groups based on their cardinality. Those that have even cardinality and those that have odd cardinality. Remove from this graph all vertices that have even cardinality and are isolated (they have no edges incident to them). We will denote the resulting set of vertices of odd and even cardinality by Odd and $Even$ respectively.

Now $\sum_{C \in \mathcal{C}, |C| \text{ odd}} |F' \cap \delta(C)|$ corresponds to the sum of the degrees of vertices in Odd in the graph H. And, $|\{C \in \mathcal{C}, |C| \text{ odd}\}|$, corresponds to the number of vertices in odd. Thus we need to show:

$$\sum_{v \in Odd} d_H(v) \leq 2|Odd|$$

where $d_H(v)$ denotes the degree of node v in the graph H. Since F' is a forest, H is also a forest and we have:

Number of edges in $H \leq$ number of vertices in H. Or

$$\frac{\sum_{v \in Odd} d_H(v) + \sum_{v \in Even} d_H(v)}{2} \leq |Odd| + |Even|$$

or

$$\sum_{v \in Odd} d_H(v) \leq 2|Odd| + \sum_{v \in Even} (2 - d_H(v))$$

We now claim that if $v \in Even$ then v is not a leaf. If this is true then $(2 - d_H(v)) \leq 0$ for $v \in Even$ and so we are done.

Suppose there is a $v_i \in Even$ which is a leaf. Consider the component C in H that v_i is contained in. By the construction of H, each tree in F' is either contained solely in the vertices represented by C or it is strictly outside C. Since each tree in F' contains an even number of vertices C does (w.r.t. the original graph), as well. So v_i and $C - v_i$ each contains an even number of vertices. As a result, removing the edge between v_i and $C - v_i$ would leave even sized components, thus contradicting the minimality of F'.

$\square$

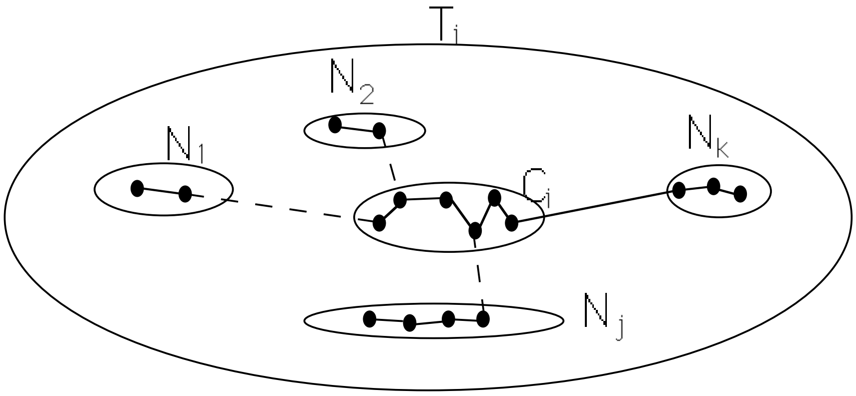


Figure 4: $d_H(v) \geq 2$ for $v \in Even$

4.7 Some implementation details

The algorithm can be implemented in $O(n^2 \log n)$. For this purpose, notice that the number of iterations is at most $n-1$ since F is a forest. The components of F can be maintained as a union-find structure and, therefore, all mergings of components take $O(n\alpha(n,n))$ where α is the inverse Ackermann function. In order to get the $O(n^2 \log n)$ bound, we shall show that every iteration can be implemented in $O(n \log n)$ time.

In order to find the edge e to add to F, we maintain a priority queue containing the edges between different components of F. This initialization of this priority queue takes $O(n^2 \log n)$ time. In order to describe the key of an edge, we need to introduce a notion of *time*. The time is initially set to zero and increases by δ in each iteration (the time can be seen to be the maximum of d_i over all vertices i). The key of an edge $e = (i,j)$ is equal to the time at which we would have $c_e = d_i + d_j$ if the parity of the components containing i and j don't change. The edge to be selected is therefore the edge with minimum key and can be obtained in $O(n \log n)$. When two components merge, we need to update the keys of edges incident to the resulting component (since the parity might have changed). By keeping track of only one edge between two components (the one with smallest key), we need to update the keys of $O(n)$ edges when two components merge. This can be done in $O(n \log n)$ ($O(\log n)$ per update).

To complete the discussion, we need to show how to go from F to F'. By performing a post-order traversal of the tree and computing the parity of the subtrees encountered (in a recursive manner), this step can be implemented in $O(n)$ time.

5 Approximating MAX-CUT

In this section, we illustrate the fact that improved approximation algorithms can be obtained by considering relaxations more sophisticated than linear ones. At the same time, we will also illustrate the fact that rounding a solution from the relaxation in a randomized fashion can be very useful. For this purpose, we consider approximation algorithms for the MAX-CUT problem. The unweighted version of this problem is as follows:

Given: A graph $G = (V, E)$.
Find: A partition $(S, \bar{S})$ such that $d(S) := |\delta(S)|$ is maximized.

It can be shown that this problem is NP-hard and MAX SNP-complete and so we cannot hope for an approximation algorithm with guarantee arbitrarily close to 1 unless $P = NP$. In the weighted version of the problem each edge has a weight w_{ij} and we define $d(S)$ by,

$$d(S) = \sum_{(i,j)\in E: i\in S, j\notin S} w_{ij}.$$

For simplicity we focus on the unweighted case. The results that we shall obtain will also apply to the weighted case.

Recall that an α-approximation algorithm for MAX-CUT is a polynomial time algorithm which delivers a cut $\delta(S)$ such that $d(S) \geq \alpha z_{MC}$ where z_{MC} is the value of the optimum cut. Until 1993 the best known α was 0.5 but now it is 0.878 due to an approximation algorithm of Goemans and Williamson [14]. We shall first of all look at three (almost identical) algorithms which have an approximation ratio of 0.5.

1. **Randomized construction.** We select S uniformly from all subsets of V. i.e. For each $i \in V$ we put $i \in S$ with probability $\frac{1}{2}$ (independently of $j \neq i$).

$$\begin{aligned} E\left[d(S)\right] &= \textstyle\sum_{(i,j)\in E} Pr\left[(i,j) \in \delta(S)\right] && \text{by linearity of expectations} \\ &= \textstyle\sum_{(i,j)\in E} Pr\left[i \in S, j \notin S \text{ or } i \notin S, j \in S\right] \\ &= \tfrac{1}{2}|E|. \end{aligned}$$

But clearly $z_{MC} \leq |E|$ and so we have $E\left[d(S)\right] \geq \frac{1}{2} z_{MC}$. Note that by comparing our cut to $|E|$, the best possible bound that we could obtain is $\frac{1}{2}$ since for K_n (the complete graph on n vertices) we have $|E| = \binom{n}{2}$ and $z_{MC} = \frac{n^2}{4}$.

2. **Greedy procedure.** Let $V = \{1, 2, \ldots, n\}$ and let $E_j = \{i : (i,j) \in E \text{ and } i < j\}$. It is clear that $\{E_j : j = 2, \ldots, n\}$ forms a partition of E. The algorithm is:

```
Set S = {1}
For j = 2 to n do
    if |S ∩ E_j| ≤ (1/2)|E_j|
        then S ← S ∪ {j}.
```

If we define $F_j = E_j \cap \delta(S)$ then we can see that $\{F_j : j = 2, \ldots, n\}$ is a partition of $\delta(S)$. By definition of the algorithm it is clear that $|F_j| \geq \frac{|E_j|}{2}$. By summing over j we get $d(S) \geq \frac{|E|}{2} \geq \frac{z_{MC}}{2}$. In fact, the greedy algorithm can be obtained from the randomized algorithm by using the method of conditional expectations.

3. **Local search.** Say that $\delta(S)$ is locally optimum if $\forall i \in S : d(S - \{i\}) \leq d(S)$ and $\forall i \notin S : d(S \cup \{i\}) \leq d(S)$.

Lemma 6 *If $\delta(S)$ is locally optimum then $d(S) \geq \frac{|E|}{2}$.*

Proof:

$$\begin{aligned}
d(S) &= \frac{1}{2}\sum_{i \in V}\{\text{number of edges in cut incident to } i\} \\
&= \frac{1}{2}\sum_{i \in V} |\delta(S) \cap \delta(i)| \\
&\geq \frac{1}{2}\sum_{i \in V} \frac{1}{2} d(i)| \\
&= \frac{2}{4}|E| \\
&= \frac{|E|}{2}.
\end{aligned}$$

The inequality is true because if $|\delta(S) \cap \delta(i)| < \frac{1}{2}|\delta(i)|$ for some i then we can move i to the other side of the cut and get an improvement. This contradicts local optimality. □

In local search we move one vertex at a time from one side of the cut to the other until we reach a local optimum. In the unweighted case this is a polynomial time algorithm since the number of different values that a cut can take is $O(n^2)$. In the weighted case the running time can be exponential. Haken and Luby [15] have shown that this can be true even for 4-regular graphs. For cubic graphs the running time is polynomial [22].

Over the last 15-20 years a number of small improvements were made in the approximation ratio obtainable for MAX-CUT. The ratio increased in the following manner:

$$\frac{1}{2} \rightarrow \frac{1}{2} + \frac{1}{2m} \rightarrow \frac{1}{2} + \frac{1}{2n} \rightarrow \frac{1}{2} + \frac{n-1}{4m}$$

where $m = |E|$ and $n = |V|$, but asymptotically this is still 0.5.

5.1 Randomized 0.878 Algorithm

The algorithm that we now present is randomized but it differs from our previous randomized algorithm in two important ways.

- The event $i \in S$ is not independent from the event $j \in S$.
- We compare the cut that we obtain to an upper bound which is better that $|E|$.

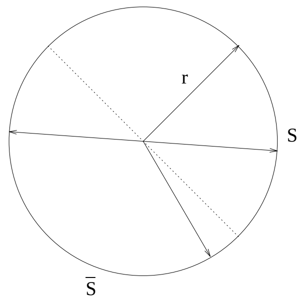


Figure 5: The sphere S_n.

Suppose that for each vertex $i \in V$ we have a vector $v_i \in \mathbb{R}^n$ (where $n = |V|$). Let S_n be the unit sphere $\{x \in \mathbb{R}^n : ||x|| = 1\}$. Take a point r uniformly distributed on S_n and let $S = \{i \in V : v_i \cdot r \geq 0\}$ (Figure 5). (Note that without loss of generality $||v_i|| = 1$.) Then by linearity of expectations:

$$E\left[d(S)\right] = \sum_{(i,j)\in E} Pr\left[\text{sign}(v_i \cdot r) \neq \text{sign}(v_j \cdot r)\right]. \tag{5}$$

Lemma 7

$$\begin{aligned} Pr\left[sign(v_i \cdot r) \neq sign(v_j \cdot r)\right] &= Pr\left[random\ hyperplane\ separates\ v_i\ and\ v_j\right] \\ &= \frac{\alpha}{\pi} \end{aligned}$$

where $\alpha = \arccos(v_i \cdot v_j)$ *(the angle between* v_i *and* v_j*).*

Proof: This result is easy to see but it is a little difficult to formalize. Let P be the 2-dimensional plane containing v_i and v_j. Then $P \cap S_n$ is a circle. With probability 1, $H = \{x : x \cdot r = 0\}$ intersects this circle in exactly two points s and t (which are diametrically opposed). See figure 6. By symmetry s and t are uniformly distributed on the circle. The vectors v_i and v_j are separated by the hyperplane H if and only if either s or t lies on the smaller arc between v_i and v_j. This happens with probability $\frac{2\alpha}{2\pi} = \frac{\alpha}{\pi}$. $\square$

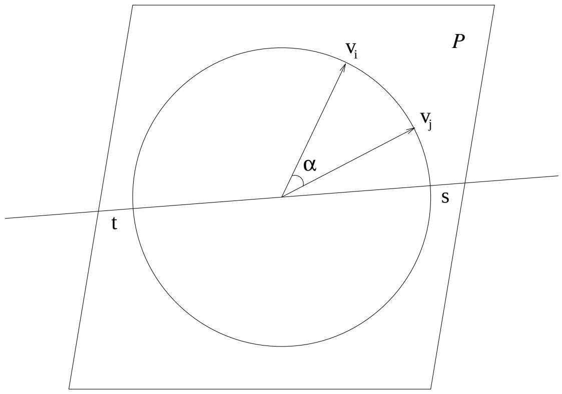


Figure 6: The plane P.

From equation 5 and lemma 7 we obtain:

$$E\left[d(S)\right] = \sum_{(i,j)\in E} \frac{\arccos(v_i \cdot v_j)}{\pi}.$$

Observe that $E\left[d(S)\right] \leq z_{MC}$ and so

$$\max_{v_i} E\left[d(S)\right] \leq z_{MC},$$

where we maximize over all choices for the v_i's. We actually have $\max_{v_i} E\left[d(S)\right] = z_{MC}$. Let $\delta(T)$ be a cut such that $d(T) = z_{MC}$ and let e be the unit vector whose first component is 1 and whose other components are 0. If we set

$$v_i = \begin{cases} e & \text{if } i \in T \\ -e & \text{otherwise.} \end{cases}$$

then $\delta(S) = \delta(T)$ with probability 1. This means that $E\left[d(S)\right] = z_{MC}$.

Corollary 8

$$z_{MC} = \max_{||v_i||=1} \sum_{(i,j)\in E} \frac{\arccos(v_i \cdot v_j)}{\pi}.$$

Unfortunately this is as difficult to solve as the original problem and so at first glance we have not made any progress.

5.2 Choosing a good set of vectors

Let $f : [-1,1] \rightarrow [0,1]$ be a function which satisfies $f(-1) = 1$, $f(1) = 0$. Consider the following program:

$$
\begin{array}{llll}
 & \text{Max} & \displaystyle\sum_{(i,j)\in E} f(v_i \cdot v_j) & \\
 & \text{subject to:} & & \\
(P) & & ||v_i|| = 1 & i \in V.
\end{array}
$$

If we denote the optimal value of this program by z_P then we have $z_{MC} \leq z_P$. This is because if we have a cut $\delta(T)$ then we can let,

$$
v_i = \begin{cases} e & \text{if } i \in T \\ -e & \text{otherwise.} \end{cases}
$$

Hence $\sum_{(i,j)\in E} f(v_i \cdot v_j) = d(T)$ and $z_{MC} \leq z_P$ follows immediately.

5.3 The Algorithm

The framework for the 0.878 approximation algorithm for MAX-CUT can now be presented.

1. Solve (P) to get a set of vectors $\{v_1^*, \ldots, v_n^*\}$.
2. Uniformly select $r \in S_n$.
3. Set $S = \{i : v_i^* \cdot r \geq 0\}$.

Theorem 9

$$
E\left[d(S)\right] \geq \alpha z_P \geq \alpha z_{MC}
$$

where,

$$
\alpha = \min_{-1\leq x\leq 1} \frac{\arccos(x)}{\pi f(x)}.
$$

Proof:

$$
\begin{aligned}
E\left[d(S)\right] &= \sum_{(i,j)\in E} \frac{\arccos(v_i^* \cdot v_j^*)}{\pi} \\
&\geq \alpha \sum_{(i,j)\in E} f(v_i^* \cdot v_j^*) \\
&= \alpha z_P \\
&\geq \alpha z_{MC}.
\end{aligned}
$$

□

We must now choose f such that (P) can be solved in polynomial time and α is as large as possible. We shall show that (P) can be solved in polynomial time whenever f is linear and so if we define,

$$f(x) = \frac{1-x}{2}$$

then our first criterion is satisfied. (Note that $f(-1) = 1$ and $f(1) = 0$.) With this choice of f,

$$\begin{aligned}\alpha &= \min_{-1 \le x \le 1} \frac{2\arccos(x)}{\pi(1-x)} \\ &= \frac{2\arccos(-0.689)}{\pi(1-(-0.689))} \\ &= 0.87856.\end{aligned}$$

(See figure 7.)

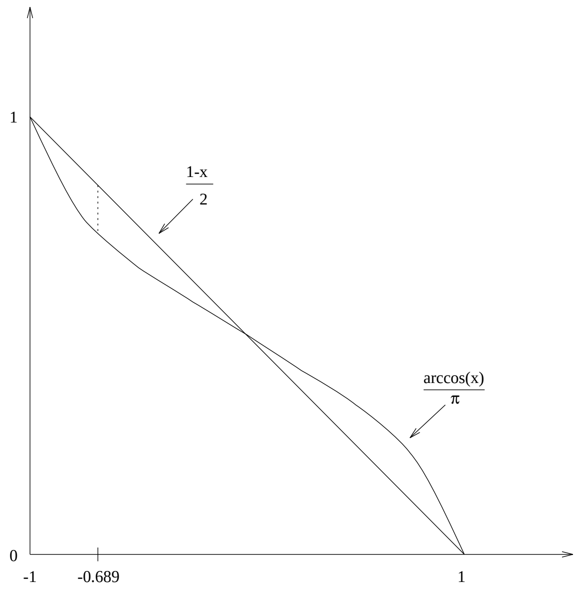


Figure 7: Calculating α.

5.4 Solving (P)

We now turn our attention to solving:

$$\text{Max} \sum_{(i,j)\in E} \frac{1}{2}(1 - v_i \cdot v_j)$$

subject to:

$$(P) \qquad ||v_i|| = 1 \qquad i \in V.$$

Let $Y = (y_{ij})$ where $y_{ij} = v_i \cdot v_j$. Then,

- $||v_i|| = 1 \Rightarrow y_{ii} = 1$ for all i.
- $y_{ij} = v_i \cdot v_j \Rightarrow Y \succeq 0$, where $Y \succeq 0$ means that Y is positive semi-definite: $\forall x : x^T Y x \geq 0$.) This is true because,

$$\begin{aligned} x^T Y x &= \sum_i \sum_j x_i x_j (v_i \cdot v_j) \\ &= \left\| \sum_i x_i v_i \right\|^2 \\ &\geq 0. \end{aligned}$$

Conversely if $Y \succeq 0$ and $y_{ii} = 1$ for all i then it can be shown that there exists a set of v_i's such that $y_{ij} = v_i \cdot v_j$. Hence (P) is equivalent to,

$$(P') \qquad \begin{aligned} \text{Max} \quad & \sum_{(i,j)\in E} \frac{1}{2}(1 - y_{ij}) \\ \text{subject to:} \quad & \\ & Y \succeq 0 \\ & y_{ii} = 1 \qquad\qquad i \in V. \end{aligned}$$

Note that $Q := \{Y : Y \succeq 0, y_{ii} = 1\}$ is convex. (If $A \succeq 0$ and $B \succeq 0$ then $A + B \succeq 0$ and also $\frac{A+B}{2} \succeq 0$.) It can be shown that maximizing a concave function over a convex set can be done in polynomial time. Hence we can solve (P') in polynomial time since linear functions are concave. This completes the analysis of the algorithm.

5.5 Remarks

1. The optimum Y could be irrational but in this case we can find a solution with an arbitrarily small error in polynomial time.
2. To solve (P') in polynomial time we could use a variation of the interior point method for linear programming.
3. Given Y, v_i can be obtained using a Cholesky factorization ($Y = VV^T$).
4. The algorithm can be derandomized using the method of conditional expectations. This is quite intricate.
5. The analysis is very nearly tight. For the 5-cycle we have z_{MC} and $z_P = \frac{5}{2}(1 + \cos\frac{\pi}{5}) = \frac{25+5\sqrt{5}}{8}$ which implies that $\frac{z_{MC}}{z_P} = 0.88445$.

6 Bin Packing and $P \parallel C_{max}$

One can push the notion of approximation algorithms a bit further than we have been doing and define the notion of approximation schemes:

Definition 4 *A* polynomial approximation scheme (pas) *is a family of algorithms* $A_\epsilon : \epsilon > 0$ *such that for each* $\epsilon > 0$*,* A_ϵ *is a* $(1+\epsilon)$*-approximation algorithm which runs in polynomial time in input size for fixed* ϵ*.*

Definition 5 *A* fully polynomial approximation scheme (fpas) *is a* pas *with running time which is polynomial both in input size and* $1/\epsilon$*.*

It is known that if π is a strongly NP-complete problem, then π has no *fpas* unless $P = NP$. From the result of Arora et al. described in Section 2, we also know that there is no *pas* for any $MAX - SNP$ hard problem unless $P = NP$.

We now consider two problems which have a very similar flavor; in fact, they correspond to the same NP-complete decision problem. However, they considerably differ in terms of approximability: one has a *pas*, the other does not.

- **Bin Packing**: Given item sizes $a_1, a_2, \ldots, a_n \geq 0$ and a bin size of T, find a partition of $I_1, \ldots, I_k$ of $1, \ldots, n$, such that $\sum_{i \in I_l} a_i \leq T$ and k is minimized (the items in I_l are assigned to bin l).
- $\boldsymbol{P \parallel C_{max}}$: Given n jobs with processing times $p_1, \ldots, p_n$ and m machines, find a partition $\{I_1, \ldots, I_m\}$ of $\{1, \ldots, n\}$, such that the makespan defined as $\max_i(\sum_{j \in I_i} p_j)$ is minimum. (The makespan represents the maximum completion time on any machine given that the jobs in I_i are assigned to machine i).

The decision versions of the two problems are identical and NP-complete. However when we consider approximation algorithms for the two problems we have completely different results. In the case of the bin packing problem there is no α-approximation algorithm with $\alpha < 3/2$, unless $P = NP$.

Proposition 10 *There is no* α*-approximation algorithm with* $\alpha < 3/2$*, for bin packing, unless* $P = NP$*, as seen in Section 2.*

However, we shall see, for $P \parallel C_{max}$ we have α-approximation algorithms for any α.

Definition 6 *An algorithm* **A** *has an asymptotic performance guarantee of* α *if*

$$\alpha \geq \limsup_{k \to \infty} \alpha_k$$

where

$$\alpha_k = \sup_{I:OPT(I)=k} \frac{A(I)}{OPT(I)}$$

and $OPT(I)$ denotes the value of instance I and $A(I)$ denotes the value returned by algorithm **A***.*

For $P \parallel C_{max}$, there is no difference between an asymptotic performance and a performance guarantee. This follows from the fact that $P \parallel C_{max}$ satisfies a scaling property : an instance with value $\beta OPT(I)$ can be constructed by multiplying every processing time p_j by β.

Using this definition we can analogously define a *polynomial asymptotic approximation scheme (paas)*. And a *fully polynomial asymptotic approximation scheme (fpaas)*.

Now we will state some results to illustrate the difference in the two problems when we consider approximation algorithms.

1. For bin packing, there does not exist an α-approximation algorithm with $\alpha < 3/2$, unless $P = NP$. Therefore there is no *pas* for bin packing unless $P = NP$.

2. For $P \parallel C_{max}$ there exists a *pas*. This is due to Hochbaum and Shmoys [17]. We will study this algorithm in more detail in today's lecture.

3. For bin packing there exists a *paas*. (Fernandez de la Vega and Lueker [7]).

4. For $P \parallel C_{max}$ there exists no *fpaas* unless $P = NP$. This is because the existence of a *fpaas* implies the existence of a *fpas* and the existence of a *fpas* is ruled out unless $P = NP$ because, $P \parallel C_{max}$ is strongly NP-complete.

5. For bin packing there even exists a *fpaas*. This was shown by Karmarkar and Karp [18].

6.1 Approximation algorithm for $P||C_{max}$

We will now present a polynomial approximation scheme for the $P||C_{max}$ scheduling problem.

We analyze a pas for $P||C_{max}$, discovered by Hochbaum and Shmoys [17]. The idea is to use a relation similar to the one between an optimization problem and its decision problem. That is, if we have a way to solve decision problem, we can use binary search to find the exact solution. Similarly, in order to obtain a $(1+\epsilon)$-approximation algorithm, we are going to use a so-called $(1+\epsilon)$-relaxed decision version of the problem and binary search.

Definition 7 *$(1+\epsilon)$-relaxed decision version of $P||C_{max}$ is a procedure that given ϵ and a deadline T, returns either:*

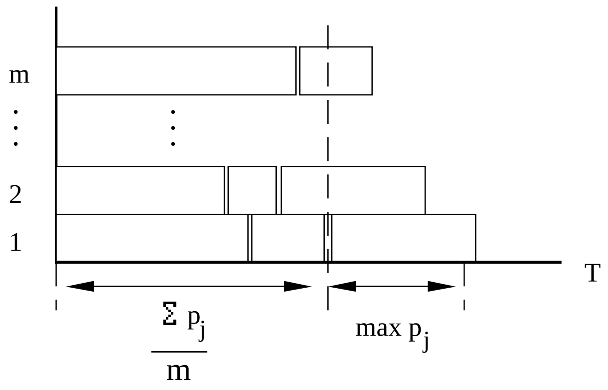


Figure 8: List scheduling.

- "NO" — *if there does not exist a schedule with makespan* $\leq T$*, or*
- "YES" — *if a schedule with makespan* $\leq (1+\epsilon)T$ *exists.*

In case of "yes", the actual schedule must also be provided.

Notice that in some cases both answers are valid. In such a case, we do not care if the procedure outputs "yes" or "no". Suppose we have such a procedure. Then we use binary search to find the solution. To begin our binary search, we must find an interval where optimal $C_{\max}$ is contained. Notice that $\left(\sum_j p_j\right)/m$ is an average load per machine and $\max_j p_j$ is the length of the longest job. We can put a bound on optimum $C_{\max}$ as follows:

Lemma 11 *Let*

$$L = \max\left(\max_j p_j, \frac{\left(\sum_j p_j\right)}{m}\right)$$

then $L \leq C_{\max} < 2L$*.*

Proof: Since the longest job must be completed, we have $\max_j p_j \leq C_{\max}$. Also, since $\left(\sum_j p_j\right)/m$ is the average load, we have $\left(\sum_j p_j\right)/m \leq C_{\max}$. Thus, $L \leq C_{\max}$.

The upper bound relies on the concept of list scheduling, which dictates that a job is never processed on some machine, if it can be processed earlier on another machine. That is, we require that if there is a job waiting, and an idle machine, we must use this machine to do the job. We claim that for such a schedule $C_{\max} < 2L$. Consider the job that finishes last, say job k. Notice that when it starts, *all* other machines are busy. Moreover, the time elapsed up to that point is no more than the average load of the machines (see Figure 8). Therefore,

$$\begin{aligned} C_{\max} &\leq \frac{\sum_{j \neq k} p_j}{m} + p_k \\ &< \frac{\sum_j p_j}{m} + \max_j p_j \\ &\leq 2 \max\left(\max_j p_j, \frac{\sum_j p_j}{m}\right) \\ &= 2L. \end{aligned}$$

□

Now we have an interval on which to do a (logarithmic) binary search for $C_{\max}$. By T_1 and T_2 we denote lower and upper bound pointers we are going to use in our binary search. Clearly, $T = \sqrt{T_1 T_2}$ is the midpoint in the logarithmic sense. Based on Lemma 11, we must search for the solution in the interval $[L, \ldots, 2L]$. Since we use logarithmic scale, we set $\log T_1 = \log_2 L$, $\log T_2 = \log_2 L + 1$ and $\log T = \frac{1}{2}(\log_2 T_1 + \log_2 T_2)$.

When do we stop? The idea is to use different value of ϵ. That is, the approximation algorithm proceeds as follows. Every time, the new interval is chosen depending on whether the procedure for the $(1+\epsilon/2)$-relaxed decision version returns a "no" or (in case of "yes") a schedule with makespan $\leq (1+\epsilon/2)T$, where $T = \sqrt{T_1 T_2}$ and $[T_1, \ldots, T_2]$ is the current interval. The binary search continues until the bounds T_1, T_2 satisfy the relation $\frac{T_2(1+\epsilon/2)}{T_1} \leq (1+\epsilon)$, or equivalently $\frac{T_2}{T_1} \leq \frac{1+\epsilon}{1+\epsilon/2}$. The number of iterations required to satisfy this relation is $O(\lg(1/\epsilon))$. Notice that this value is a *constant* for a *fixed* ϵ. At termination, the makespan of the schedule corresponding to T_2 will be within a factor of $(1+\epsilon)$ of the optimal makespan.

In order to complete the analysis of the algorithm, it remains to describe the procedure for the $(1+\epsilon/2)$-relaxed decision procedure for any ϵ. Intuitively, if we look at what can go wrong in list scheduling, we see that it is "governed" by "long" jobs, since small jobs can be easily accommodated. This is the approach we take, when designing procedure that solves the $(1+\epsilon/2)$-relaxed decision version of the problem. For the rest of our discussion we will denote $\epsilon/2$ by ϵ'.

Given $\{p_j\}$, ϵ' and T, the procedure operates as follows:

Step 1: *Remove all (small) jobs with* $p_j \leq \epsilon' T$.

Step 2: *Somehow (to be specified later) solve the* $(1+\epsilon')$*-relaxed decision version of the problem for the remaining (big) jobs.*

Step 3: *If answer in step 2 is "no", then return that there does not exist a schedule with makespan* $\leq T$.
If answer in step 2 is "yes", then with a deadline of $(1+\epsilon')T$ *put back all small jobs using list scheduling (i.e. greedy strategy), one at a time. If all jobs are*

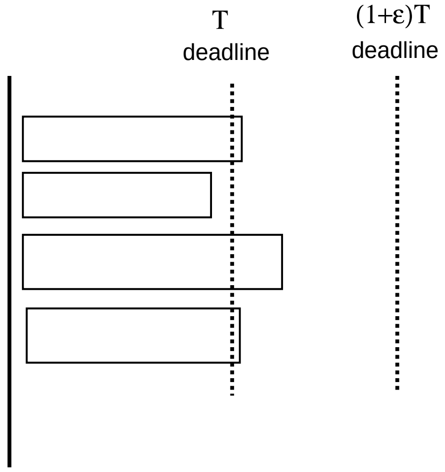


Figure 9: Scheduling "small" jobs.

accommodated then return that schedule, else return that there does not exist a schedule with makespan $\leq T$.

Step 3 of the algorithm gives the final answer of the procedure. In case of a "yes" it is clear that the answer is correct. In case of a "no" that was propagated from Step 2 it is also clear that the answer is correct. Finally, if we fail to put back all the small jobs we must also show that the algorithm is correct. Let us look at a list schedule in which some small jobs have been scheduled but others couldn't (see Figure 9).

If we cannot accomodate all small jobs with a deadline of $(1+\epsilon')T$, it means that all machines are busy at time T since the processing time of each small job is $\leq \epsilon' T$. Hence, the average load per processor exceeds T. Therefore, the answer "no" is correct.

Now, we describe Step 2 of the algorithm for $p_j > \epsilon' T$. Having eliminated the small jobs, we obtain a constant (when ϵ is fixed) upper bound on the number of jobs processed on one machine. Also, we would like to have only a small number of distinct processing times in order to be able to enumerate in polynomial time all possible schedules. For this purpose, the idea is to use rounding. Let q_j be the largest number of the form $\epsilon' T + k\epsilon'^2 T \leq p_j$ for some $k \in \mathbf{N}$. A refinement of Step 2 is the following.

2.1 Address the decision problem: Is there a schedule for $\{q_j\}$ with makespan $\leq T$?

2.2 If the answer is "no", then return that there does not exist a schedule with makespan $\leq T$.
If the answer is "yes", then return that schedule.

The Lemma that follows justifies the correctness of the refined Step 2.

Lemma 12 *Step 2 of the algorithm is correct.*

Proof: If Step 2.1 returns "no", then it is clear that the final answer of Step 2 should be "no", since $q_j \leq p_j$.

If Step 2.1 returns "yes", then the total increase of the makespan due to the replacement of q_j by p_j is no greater than $(1/\epsilon')\epsilon'^2 T = \epsilon' T$. This is true, because we have at most $T/(\epsilon' T) = 1/\epsilon'$ jobs per machine, and because $p_j \leq q_j + \epsilon'^2 T$ by definition. Thus, the total length of the schedule with respect to $\{p_j\}$ is at most $T + \epsilon' T = (1 + \epsilon')T$. □

It remains to show how to solve the decision problem of Step 2.1. We can achieve this in polynomial time using dynamic programming. Note that the input to this decision problem is "nice": We have at most $P = \lfloor 1/\epsilon' \rfloor$ jobs per machine, and at most $Q = 1 + \left\lfloor \frac{1-\epsilon'}{\epsilon'^2} \right\rfloor$ distinct processing times. Since ϵ' is considered to be fixed, we essentially have a constant number of jobs per machine and a constant number $q'_1, \ldots, q'_Q$ of processing times. Let $\vec{n} = \{n_1, \ldots, n_Q\}$, where n_i denotes the number of jobs whose processing time is q_i. We use the fact that the decision problems of $P||C_{max}$ and the bin packing problems are equivalent. Let $f(\vec{n})$ denote the minimum number of machines needed to process $\vec{n}$ by time T. Finally, let $R = \{\vec{r} = (r_1, \ldots, r_Q) : \sum_{i=1}^{Q} r_i q'_i \leq T, r_i \leq n_i, r_i \in \mathbb{N}$. R represents the sets of jobs that can be processed on a single machine with a deadline of T. The recurrence for the dynamic programming formulation of the problem is

$$f(\vec{n}) = 1 + \min_{\vec{r} \in R} f(\vec{n} - \vec{r}),$$

namely we need one machine to accomodate the jobs in $\vec{r} \in R$ and $f(\vec{n} - \vec{r})$ machines to accomodate the remaining jobs. In order to compute this recurrence we first have to compute the at most Q^P vectors in R. The upper bound on the size of R comes from the fact that we have at most P jobs per machine and each job can have one of at most Q processing times. Subsequently, for each one of the vectors in R we have to iterate for n^Q times, since $n_i \leq n$ and there are Q components in $\vec{n}$. Thus, the running time of Step 2.1 is $O(n^{1/\epsilon'^2}(1/\epsilon'^2)^{(1/\epsilon')})$.

From this point we can derive the overall running time of the pas in a straightforward manner. Since Step 2 iterates $O(\lg(1/\epsilon))$ times and since $\epsilon = 2\epsilon'$, the overall running time of the algorithm is $O(n^{1/\epsilon^2}(1/\epsilon^2)^{(1/\epsilon)}\lg(1/\epsilon))$.

7 Randomized Rounding for Multicommodity Flows

In this section, we look at using randomness to approximate a certain kind of multicommodity flow problem. The problem is as follows: given a directed graph $G = (V, E)$, with sources $s_i \in V$ and sinks $t_i \in V$ for $i = 1, \ldots, k$, we want to find a path P_i from s_i to t_i for $1 \leq i \leq k$ such that the "width" or "congestion" of any edge is as small as possible. The "width" of an edge is defined to be the number of paths using that edge. This multicommodity flow problem is NP-complete in general.

The randomized approximation algorithm that we discuss in these notes is due to Raghavan and Thompson [24].

7.1 Reformulating the problem

The multicommodity flow problem can be formulated as the following integer program:

$$
\begin{array}{lll}
 & \text{Min} \quad W & \\
\multicolumn{3}{l}{\text{subject to:}} \\
(6) & \sum_w x_i(v,w) - \sum_w x_i(w,v) = \begin{cases} 1 & \text{if } v = s_i \\ -1 & \text{if } v = t_i \\ 0 & \text{otherwise} \end{cases} & i = 1,\ldots,k, v \in V, \\
 & x_i(v,w) \in \{0,1\} & i = 1,\ldots,k, (v,w) \in E, \\
(7) & \sum_i x_i(v,w) \leq W & (v,w) \in E.
\end{array}
$$

Notice that constraint (6) forces the x_i to define a path (perhaps not simple) from s_i to t_i. Constraint (7) ensures that every edge has width no greater than W, and the overall integer program minimizes W.

We can consider the LP relaxation of this integer program by replacing the constraints $x_i(v,w) \in \{0,1\}$ with $x_i(v,w) \geq 0$. The resulting linear program can be solved in polynomial time by using interior point methods discussed earlier in the course. The resulting solution may not be integral. For example, consider the multicommodity flow problem with one source and sink, and suppose that there are exactly i edge-disjoint paths between the source and sink. If we weight the edges of each path by $\frac{1}{i}$ (i.e. set $x(v,w) = \frac{1}{i}$ for each edge of each path), then $W_{LP} = \frac{1}{i}$. The value W_{LP} can be no smaller: since there are i edge-disjoint paths, there is a cut in the graph with i edges. The average flow on these edges will be $\frac{1}{i}$, so that the width will be at least $\frac{1}{i}$.

The fractional solution can be decomposed into paths using *flow decomposition*, a standard technique from network flow theory. Let x be such that $x \geq 0$ and

$$\sum_w x(v,w) - \sum_w x(w,v) = \begin{cases} a & \text{if } v = s_i \\ -a & \text{if } v = t_i \\ 0 & \text{otherwise.} \end{cases}$$

Then we can find paths $P_1, \ldots, P_l$ from s_i to t_i such that

$$
\begin{aligned}
\alpha_1, \ldots, \alpha_l &\in \mathbb{R}^+ \\
\sum_i \alpha_i &= a \\
\sum_{j:(v,w)\in P_j} \alpha_j &\leq x(v,w).
\end{aligned}
$$

To see why we can do this, suppose we only have one source and one sink, s and t. Look at the "residual graph" of x: that is, all edges (v, w) such that $x(v, w) > 0$. Find some path P_1 from s to t in this graph. Let $\alpha_1 = \min_{(v,w)\in P_1} x(v, w)$. Set

$$x'(v,w) = \begin{cases} x(v,w) - \alpha_1 & (v,w) \in P_1 \\ x(v,w) & \text{otherwise.} \end{cases}$$

We can now solve the problem recursively with $a' = a - \alpha_1$.

7.2 The algorithm

We now present Raghavan and Thompson's randomized algorithm for this problem.

1. Solve the LP relaxation, yielding W_{LP}.

2. Decompose the fractional solution into paths, yielding paths P_{ij} for $i = 1, \ldots, k$ and $j = 1, \ldots, j_i$ (where P_{ij} is a path from s_i to t_i), and yielding $\alpha_{ij} > 0$ such that $\sum_j \alpha_{ij} = 1$ and
$$\sum_i \sum_{j:(v,w)\in P_{ij}} \alpha_{ij} \leq W_{LP}.$$

3. (Randomization step) For all i, cast a j_i-faced die with face probabilities α_{ij}. If the outcome is face f, select path P_{if} as the path P_i from s_i to t_i.

We will show, using a Chernoff bound, that with high probability we will get small congestion. Later we will show how to derandomize this algorithm. To carry out the derandomization it will be important to have a strong handle on the Chernoff bound and its derivation.

7.3 Chernoff bound

For completeness, we include the derivation of a Chernoff bound, although it already appears in the randomized algorithms chapter.

Lemma 13 *Let X_i be independent Bernoulli random variables with probability of success p_i. Then, for all $\alpha > 0$ and all $t > 0$, we have*

$$Pr\left[\sum_{i=1}^{k} X_i > t\right] \leq e^{-\alpha t} \prod_{i=1}^{k} E\left[e^{\alpha X_i}\right] = e^{-\alpha t} \prod_{i=1}^{k} \left(p_i e^{\alpha} + (1 - p_i)\right).$$

Proof:

$$Pr\left[\sum_{i=1}^{k} X_i > t\right] = Pr\left[e^{\alpha \sum_{i=1}^{k} X_i} > e^{\alpha t}\right]$$

for any $\alpha > 0$. Moreover, this can be written as $Pr[Y > a]$ with $Y \geq 0$. From Markov's inequality we have

$$Pr[Y > a] \leq \frac{E[Y]}{a}$$

for any nonnegative random variable. Thus,

$$\begin{aligned} Pr\left[\sum_{i=1}^{k} X_i > t\right] &\leq e^{-\alpha t} E\left[e^{\alpha \sum_i X_i}\right] \\ &= e^{-\alpha t} \prod_{i=1}^{k} E\left[e^{\alpha X_i}\right] \text{ because of independence.} \end{aligned}$$

The equality then follows from the definition of expectation. □

Setting $t = (1+\beta)E[\sum_i X_i]$ for some $\beta > 0$ and $\alpha = \ln(1+\beta)$, we obtain:

Corollary 14 *Let X_i be independent Bernoulli random variables with probability of success p_i, and let $M = E[\sum_{i=1}^{k} X_1] = \sum_{i=1}^{k} p_i$. Then, for all $\beta > 0$, we have*

$$Pr\left[\sum_{i=1}^{k} X_i > (1+\beta)M\right] \leq (1+\beta)^{-(1+\beta)M} \prod_{i=1}^{k} E\left[(1+\beta)^{X_i}\right] \leq \left[\frac{e^{\beta}}{(1+\beta)^{(1+\beta)}}\right]^M .$$

The second inequality of the corollary follows from the fact that

$$E\left[(1+\beta)^{X_i}\right] = p_i(1+\beta) + (1-p_i) = 1 + \beta p_i \leq e^{\beta p_i}.$$

7.4 Analysis of the R-T algorithm

Raghavan and Thompson show the following theorem.

Theorem 15 *Given $\epsilon > 0$, if the optimal solution to the multicommodity flow problem W^* has value $W^* = \Omega(\log n)$ where $n = |V|$, then the algorithm produces a solution of width $W \leq W^* + c\sqrt{W^* \ln n}$ with probability $1 - \epsilon$ (where c and the constant in $\Omega(\log n)$ depends on ϵ, see the proof).*

Proof: Fix an edge $(v,w) \in E$. Edge (v,w) is used by commodity i with probability $p_i = \sum_{j:(v,w)\in P_{ij}} \alpha_{ij}$. Let X_i be a Bernoulli random variable denoting whether or not (v,w) is in path P_i. Then $W(v,w) = \sum_{i=1}^{k} X_i$, where $W(v,w)$ is the width of edge (v,w). Hence,

$$E[W(v,w)] = \sum_i p_i = \sum_i \sum_{j:(v,w)\in P_{ij}} \alpha_{ij} \leq W_{LP} \leq W^*.$$

Now using the Chernoff bound derived earlier,

$$Pr[W(v,w) \geq (1+\beta)W^*] \leq \left[\frac{e^{\beta}}{(1+\beta)^{(1+\beta)}}\right]^{W^*} = e^{[\beta-(1+\beta)\ln(1+\beta)]W^*}.$$

Assume that $\beta \leq 1$. Then, one can show that

$$\frac{e^{\beta}}{(1+\beta)^{(1+\beta)}} = e^{\beta-(1+\beta)\ln(1+\beta)} \leq e^{-\beta^2/3}.$$

Therefore, for

$$\beta = \sqrt{\frac{3\ln\frac{n^2}{\varepsilon}}{W^*}},$$

we have that

$$Pr[W(v,w) \geq (1+\beta)W^*] \leq \frac{\varepsilon}{n^2}.$$

Notice that our assumption that $\beta \leq 1$ is met if

$$W^* \geq 6\ln n - 3\ln\varepsilon.$$

For this choice of β, we derive that

$$(1+\beta)W^* = W^* + \sqrt{3W^*\ln\frac{n^2}{\varepsilon}}.$$

We consider now the maximum congestion. We have

$$Pr[\max_{(v,w)\in E} W(v,w) \geq (1+\beta)W^*] \leq \sum_{(v,w)\in E} Pr[W(v,w) \geq (1+\beta)W^*] \leq |E|\frac{\varepsilon}{n^2} \leq \varepsilon,$$

proving the result. $\square$

7.5 Derandomization

We will use the method of conditional probabilities. We will need to supplement this technique, however, with an additional trick to carry through the derandomization. This result is due to Raghavan [23].

We can represent the probability space using a decision tree. At the root of the tree we haven't made any decisions. As we descend the tree from the root we represent the choices first for commodity 1, then for commodity 2, etc. Hence the root has j_1 children representing the j_1 possible paths for commodity 1. Each of these nodes has j_2 children, one for each of the j_2 possible paths for commodity 2. We continue in the manner, until we have reached level k. Clearly the leaves of this tree represent all the possible choices of paths for the k commodities.

A node at level i (the root is at level 0) is labeled by the i choices of paths for commodities $1\ldots i : l_1 \ldots l_i$. Now we define:

$$g(l_1\ldots l_i) = Pr\left[\max_{(v,w)\in E} W(v,w) \geq (1+\beta)W^* \;\middle|\; \begin{array}{l} l_1 \text{ for commodity } 1 \\ l_2 \text{ for commodity } 2 \\ \vdots \\ l_i \text{ for commodity } i \end{array}\right].$$

By conditioning on the choice of the path for commodity i, we obtain that

$$
\begin{aligned}
g(l_1 \dots l_{i-1}) &= \sum_{j=1}^{j_i} \alpha_{ij} g(l_1, \dots, l_{i-1}, j) && (8) \\
&\geq \min_j g(l_1, \dots, l_{i-1}, j) && (9)
\end{aligned}
$$

If we could compute $g(l_1, l_2, \dots)$ efficiently, we could start from $g(\emptyset)$ and by selecting the minimum at each stage construct a sequence $g(\emptyset) \geq g(l_1) \geq g(l_1, l_2) \geq \dots \geq g(l_1, l_2, \dots, l_k)$. Unfortunately we don't know how to calculate these quantities. Therefore we need to use an additional trick.

Instead of using the exact value g, we shall use a *pessimistic estimator* for the probability of failure. From the derivation of the Chernoff bound and the analysis of the algorithm, we know that

$$
(10)\; Pr\left[\max_{(v,w)\in E} W(v,w) \geq (1+\beta)W^*\right] \leq (1+\beta)^{-(1+\beta)W^*} \sum_{(v,w)\in E} E\left[\prod_{i=1}^{k} e^{\beta X_i^{(v,w)}}\right],
$$

where the superscript on X_i denotes the dependence on the edge (v, w), i.e. $X_i^{(v,w)} = 1$ if (v, w) belongs to the path P_i. Letting $h(l_1, \dots, l_i)$ be the RHS of (10) when we condition on selecting path P_{jl_j} for commodity j, $j = 1, \dots, i$, we observe that:

1. $h(l_1, \dots, l_i)$ can be easily computed,
2. $g(l_1, \dots, l_i) \leq h(l_1, \dots, l_i)$ and
3. $h(l_1, \dots, l_i) \geq \min_j h(l_1, \dots, l_{i-1}, j)$.

Therefore, selecting the minimum in the last inequality at each stage, we construct a sequence such that $1 > \varepsilon \geq h(\emptyset) \geq h(l_1) \geq h(l_1, l_2) \geq \dots \geq h(l_1, l_2, \dots, l_k) \geq g(l_1, l_2, \dots, l_k)$. Since $g(l_1, l_2, \dots, l_k)$ is either 0 or 1 (there is no randomness involved), we must have that the choice of paths of this deterministic algorithm gives a maximum congestion less than $(1 + \beta)W^*$.

8 Multicommodity Flow

Consider an undirected graph $G = (V, E)$ with a capacity u_e on each edge. Suppose that we are given k commodities and a demand for f_i units of commodity i between two points s_i and t_i. In the area of multicommodity flow, one is interested in knowing whether all commodities can be shipped simultaneously. That is, can we find flows of value f_i between s_i and t_i such that the sum over all commodities of the flow on each edge (in either direction) is at most the capacity of the edge.

There are several variations of the problem. Here, we consider the concurrent flow problem: Find α^* where α^* is the maximum α such that for each commodity we can ship αf_i units from s_i to t_i. This problem can be solved by linear programming since all the constraints are linear. Indeed, one can have a flow variable for each edge and each commodity (in addition to the variable α), and the constraints consist of the flow conservation constraints for each commodity as well as a capacity constraint for every edge. An example is shown in figure 8. The demand for each commodity is 1 unit and the capacity on each edge is 1 unit. It can be shown that $\alpha^* = \frac{3}{4}$.

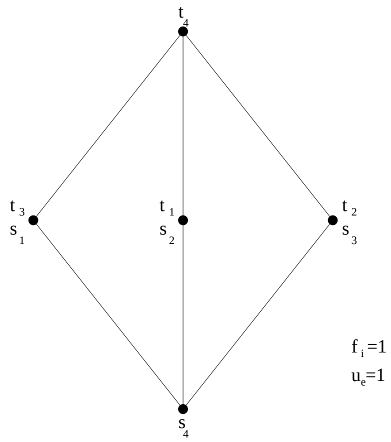


Figure 10: An example of the multi-commodity flow problem.

When there is only one commodity, we know that the maximum flow value is equal to the minimum cut value. Let us investigate whether there is such an analogue for multicommodity flow. Consider a cut $(S, \bar{S})$. As usual $\delta(S)$ is the set of edges with exactly one endpoint in S. Let,

$$f(S) = \sum_{i:|S \cap \{s_i, t_i\}|=1} f_i.$$

Since all flow between S and $\bar{S}$ must pass along one of the edges in $\delta(S)$ we must have,

$$\alpha^* \leq \frac{u(\delta(S))}{f(S)}$$

where $u(\delta(S)) = \sum_{e \in \delta(S)} u_e$. The multicommodity cut problem is to find a set S which minimizes $\frac{u(\delta(S))}{f(S)}$. We let β^* be the minimum value attainable and so we have $\alpha^* \leq \beta^*$. But, in general, we don't have equality. For example, in Figure 8, we have $\beta^* = 1$. In fact, it can be shown that the multicommodity cut problem is NP-hard. We shall consider the following two related questions.

1. In the worst case, how large can $\frac{\beta^*}{\alpha^*}$ be?

2. Can we obtain an approximation algorithm for the multicommodity cut problem?

In special cases, answers have been given to these questions by Leighton and Rao [19] and in subsequent work by many other authors. In this section, we describe a very recent, elegant and general answer due to Linial, London and Rabinovitch [20]. The technique they used is the embedding of metrics. The application to multicommodity flows was also independently obtained by Aumann and Rabani [3].

We first describe some background material on metrics and their embeddings.

Definition 8 *(X, d) is a metric space or d is a metric on a set X if*

1. $\forall x, y : d(x, y) \geq 0$.
2. $\forall x, y : d(x, y) = d(y, x)$.
3. $\forall x, y, z : d(x, y) + d(y, z) \geq d(x, z)$.

Strictly speaking we have defined a semi-metric since we do not have the condition $d(x, y) = 0 \Rightarrow x = y$. We will be dealing mostly with finite metric spaces, where X is finite. In $\mathbb{R}^n$ then the following are all metrics:

$$
\begin{array}{llllll}
d(x,y) & = & ||x-y||_2 & = & \sqrt{\sum (x_i - y_i)^2} & \ell_2 \text{ metric} \\
d(x,y) & = & ||x-y||_1 & = & \sum |x_i - y_i| & \ell_1 \text{ metric} \\
d(x,y) & = & ||x-y||_\infty & = & \max_i |x_i - y_i| & \ell_\infty \text{ metric} \\
d(x,y) & = & ||x-y||_p & = & (\sum |x_i - y_i|^p)^{\frac{1}{p}} & \ell_p \text{ metric}
\end{array}
$$

Definition 9 *(X, d) can be embedded into (Y, ℓ) if there exists a mapping $\varphi : X \rightarrow Y$ which satisfies $\forall x, y : \ell(\varphi(x), \varphi(y)) = d(x, y)$.*

Definition 10 *(X, d) can be embedded into (Y, ℓ) with distortion c if there exists a mapping $\varphi : X \rightarrow Y$ which satisfies $\forall x, y : d(x, y) \leq \ell(\varphi(x), \varphi(y)) \leq c d(x, y)$.*

The following are very natural and central questions regarding the embedding of metrics.

ISIT-ℓ_p: Given a finite metric space (X, d), can it be embedded into $(\mathbb{R}^n, \ell_p)$ for some n?

EMBED-ℓ_p: Given a finite metric space (X, d) and $c \geq 1$, find an embedding of (X, d) into $(\mathbb{R}^n, \ell_p)$ with distortion at most c for some n.

As we will see in the following theorems, the complexity of these questions depend critically on the metric themselves.

Theorem 16 *Any (X, d) can be embedded into $(\mathbb{R}^n, \ell_\infty)$ where $n = |X|$. (Thus, the answer to ISIT-ℓ_∞ is always "yes".)*

Proof: We define a coordinate for each point $z \in X$. Let $d(x, z)$ be the z coordinate of x. Then,

$$\ell_\infty(\varphi(x), \varphi(y)) = \max_{z \in X} |d(x, z) - d(y, z)| = d(x, y),$$

because of the triangle inequality. □

Theorem 17 *ISIT-$\ell_2 \in P$. (i.e., ISIT-ℓ_2 can be answered in polynomial time.)*

Proof: Assume that there exists an embedding of $\{1, 2, \ldots, n\}$ into $\{v_1 = 0, v_2, \ldots, v_n\}$. Consider one such embedding. Then,

$$d^2(i,j) = ||v_i - v_j||^2 = (v_i - v_j)(v_i - v_j) = v_i^2 - 2v_i \cdot v_j + v_j^2.$$

But $v_i^2 = d^2(1,i)$ and $v_j^2 = d^2(1,j)$ which means that,

$$v_i \cdot v_j = \frac{d^2(1,i) + d^2(1,j) - d^2(i,j)}{2}.$$

We now construct $M = (m_{ij})$ where,

$$m_{ij} = \frac{d^2(1,i) + d^2(1,j) - d^2(i,j)}{2}.$$

Hence if M is not positive semi-definite then there is no embedding into ℓ_2. If M is positive semidefinite then we carry out a Cholesky decomposition on M to express M as $M = VV^T$. From the rows of V we can obtain an embedding into $(\mathbb{R}^n, \ell_2)$. $\square$

Theorem 18 *ISIT-ℓ_1 is NP-complete.*

This theorem is given without proof. The reduction is from MAX CUT, since as we will see later there is a very close relationship between l_1-embeddable metrics and cuts. We also omit the proof of the following theorem.

Theorem 19 *Let $X \subseteq \mathbb{R}^n$. (X, ℓ_2) can be embedded into $(\mathbb{R}^m, \ell_1)$ for some m.*

The converse of this theorem is not true as can be seen from the metric space $(\{(0,0), (-1,0), (1,0), (0,1)\}, \ell_1)$.

8.1 Reducing multicommodity flow/cut questions to embedding questions

In this section, we relate α^* and β^* through the use of metrics.

Claim 20

$$\beta^* = \min_{\ell_1\text{-embeddable metrics } (V,\ell)} \frac{\sum_{(x,y)\in E} u_{xy}\ell(x,y)}{\sum_{i=1}^k f_i \ell(s_i,t_i)}.$$

Proof:

($\geq$) Given S, let

$$\varphi(x) = \begin{cases} 0 & \text{if } x \notin S \\ 1 & \text{if } x \in S. \end{cases}$$

Let ℓ be the ℓ_1 metric on the line, i.e. $\ell(a,b) = |a-b|$. Then,

$$\begin{aligned} u(\delta(S)) &= \sum_{(x,y)\in E} u_{xy}\ell(x,y) \\ f(S) &= \sum_{i=1}^{k} f_i \ell(s_i,t_i) \end{aligned}$$

since $\ell(x,y) = 1$ if and only if x is separated from y by S and 0 otherwise.

($\leq$) We can view any ℓ_1 embeddable metric ℓ as a combination of cuts. See figure 11 for the 2-dimensional case.

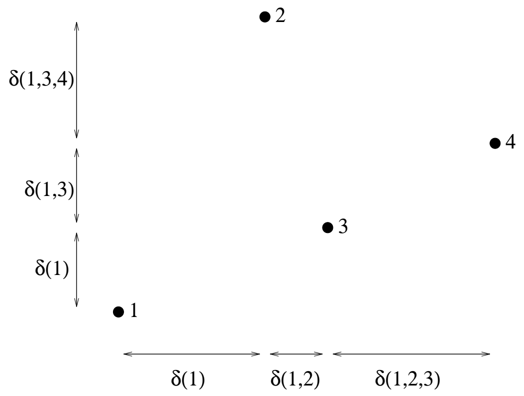


Figure 11: Viewing an ℓ_1-metric as a combination of cuts.

For any set S define a metric 1_S by,

$$1_S = \begin{cases} 1 & \text{if } x,y \text{ are separated by } S \\ 0 & \text{otherwise.} \end{cases}$$

Then we can write ℓ as,

$$\ell(x,y) = \sum_i \alpha_i 1_{S_i}(x,y),$$

where the α_i's are nonnegative. Hence,

$$\frac{\sum_{(x,y)\in E} u_{xy}\ell(x,y)}{\sum_{i=1}^{k} f_i \ell(s_i,t_i)} = \frac{\sum_i \alpha_i u(\delta(S_i))}{\sum_i \alpha_i f(S_i)} \geq \min_S \frac{u(\delta(S))}{f(S)}.$$

$\square$

Claim 21

$$\alpha^* = \min_{\ell_\infty\text{-}embeddable\ metrics\ (V,d)} \frac{\sum_{(x,y)\in E} u_{xy} d(x,y)}{\sum_i f_i d(s_i,t_i)}.$$

Note that by theorem 16 we actually minimize over all metrics.
Proof:

($\leq$) For any metric d let the volume of an edge (x,y) be $u_{xy}d(x,y)$. The total volume of the graph is $\sum_{(x,y)\in E} u_{xy}d(x,y)$. If we send a fraction α of the demand then the amount of volume that we use is at least $\alpha \sum_i f_i d(s_i,t_i)$. Hence $\alpha \sum_i f_i d(s_i,t_i) \leq \sum_{(x,y)\in E} u_{xy} d(x,y)$.

($\geq$) We use the strong duality of linear programming. α^* can be formulated as a linear program in several different ways. Here we use a formulation which works well for the purpose of this proof although it is quite impractical. Enumerate the paths from s_i to t_i, let P_{ij} be the jth such path and let x_{ij} be the flow on P_{ij}. The linear program corresponding to multicommodity flow is,

$$\begin{aligned}
\text{Max}\quad & \alpha \\
\text{subject to:} & \\
& \alpha f_i - \sum_j x_{ij} \leq 0 && i \in \{1,\ldots,k\} \\
& \sum_i \sum_{j:e\in P_{ij}} x_{ij} \leq u_e && e \in E \\
& \alpha \geq 0 \\
& x_{ij} \geq 0
\end{aligned}$$

The dual of this linear program is:

$$\begin{aligned}
\text{Min}\quad & \sum_{e\in E} u_e \ell_e \\
\text{subject to:} & \\
& \sum_{i=1}^{k} f_i h_i \geq 1 \\
& \sum_{e\in P_{ij}} \ell_e - h_i \geq 0 && \forall i,j \\
& h_i \geq 0 \\
& \ell_e \geq 0
\end{aligned}$$

The second constraint in the dual implies that h_i is at most the shortest path length between s_i and t_i with respect to ℓ_e. By strong duality if ℓ is an optimum solution to the dual then,

$$\begin{aligned}\alpha^* &= \sum_{e\in E} u_e \ell_e \\ &\geq \frac{\sum_{e\in E} u_e \ell_e}{\sum_{i=1}^k f_i h_i} \\ &\geq \frac{\sum_{(i,j)\in E} u_{ij} d(i,j)}{\sum_{i=1}^k f_i d(s_i,t_i)},\end{aligned}$$

where $d(a,b)$ represents the shortest path length with respect to ℓ_e. The first inequality holds because $\sum_{i=1}^k f_i h_i$ is constrained to be at least 1.

$\square$

Linial, London, and Rabinovitch and Aumann and Rabani use the following strategy to bound $\frac{\beta^*}{\alpha^*}$ and approximate the minimum multicommodity cut.

1. Using linear programming, find α^* and the corresponding metric d as given in claim 21.
2. Embed d into $(\mathbb{R}^m, \ell_1)$ with distortion c. Let ℓ be the resulting metric.

By claim 20 this shows that $\frac{\beta^*}{\alpha^*} \leq c$ since,

$$\beta^* \leq \frac{\sum_{(x,y)\in E} u_{xy} \ell(x,y)}{\sum_{i=1}^k f_i \ell(s_i,t_i)} \leq \frac{c\sum_{(x,y)\in E} u_{xy} d(x,y)}{\sum_{i=1}^k f_i d(s_i,t_i)} = c\alpha^*.$$

In order to approximate the minimum multicommodity cut, we can use the proof of claim 20 to decompose ℓ into cuts. If S is the best cut among them then,

$$\frac{u(\delta(S))}{f(S)} \leq \frac{\sum_{(x,y)\in E} u_{xy} \ell(x,y)}{\sum_{i=1}^k f_i \ell(s_i,t_i)}.$$

Our remaining two questions are:

- How do we get an embedding of d into ℓ? Equivalently, how can we embed ℓ_∞ into ℓ_1.
- What is c?

8.2 Embedding metrics into ℓ_1

The following theorem is due to Bourgain.

Theorem 22 *For all metrics d on n points, there exists an embedding of d into ℓ_1 which satisfies:*

$$d(x,y) \leq ||x-y||_1 \leq O(\log n) d(x,y).$$

Proof: Let k range over $\{1, 2, 4, 8, \ldots, 2^j, \ldots, 2^p\}$ where $p = \lfloor \log n \rfloor$. Hence we have $p + 1 = O(\log n)$ different values for k. Now choose n_k sets of size k. At first take all sets of size k, i.e., $n_k = \binom{n}{k}$. Introduce a coordinate for every such set. This implies that points are mapped into a space of dimension $\sum_{j=0}^{p} n_{2^j} < 2^n$. For a set A of size k the corresponding coordinate of a point x is,

$$\frac{\alpha d(x, A)}{n_k}$$

where $d(x, A) = \min_{z \in A} d(x, z)$ and α is a constant which we shall determine later. Suppose that $d(x, A) = d(x, s)$ and $d(y, A) = d(y, t)$, where s and t are in A. Then,

$$d(x, A) - d(y, A) = d(x, s) - d(y, t) \leq d(x, t) - d(y, t) \leq d(x, y).$$

Exchanging the roles of x and y, we deduce that $|d(x, A) - d(y, A)| \leq d(x, y)$. Hence,

$$\begin{aligned} ||x - y||_1 &= \sum_A \frac{\alpha}{n_{|A|}} |d(x, A) - d(y, A)| \\ &\leq \alpha \sum_A \frac{1}{n_{|A|}} d(x, y) \\ &= \alpha(p + 1) d(x, y) \\ &= O(\log n) d(x, y). \end{aligned}$$

We now want to prove that $||x - y||_1 \geq d(x, y)$. Fix two points x and y and define,

$$\begin{aligned} B(x, r) &= \{z : d(x, z) \leq r\}, \\ \bar{B}(x, r) &= \{z : d(x, z) < r\}, \\ \rho_0 &= 0, \\ \rho_t &= \inf\{r : |B(x, r)| \geq 2^t, |B(y, r)| \geq 2^t\}. \end{aligned}$$

Let ℓ be the least index such that $\rho_\ell \geq \frac{d(x,y)}{4}$. Redefine ρ_ℓ so that it is equal to $\frac{d(x,y)}{4}$. Observe that for all t either $|\bar{B}(x, \rho_t)| < 2^t$ or $|\bar{B}(y, \rho_t)| < 2^t$. Since $B(x, \rho_{\ell-1}) \cap B(y, \rho_{\ell-1}) = \emptyset$ we have $2^{\ell-1} + 2^{\ell-1} \leq n \Rightarrow \ell \leq p$. Now fix $k = 2^j$ where $p - 1 \geq j \geq p - \ell$ and let $t = p - j$ (thus, $1 \leq t \leq l$). By our observation we can assume without loss of generality that $|\bar{B}(x, \rho_t)| < 2^t$. Let A be a set of size k and consider the following two conditions.

1. $A \cap \bar{B}(x, \rho_t) = \emptyset$.
2. $A \cap B(y, \rho_{t-1}) \neq \emptyset$.

If 1. and 2. hold then $d(x, A) \geq \rho_t$ and $d(y, A) \leq \rho_{t-1}$ and so $|d(x, A) - d(y, A)| \geq \rho_t - \rho_{t-1}$. Let

$$R_k = \{A : |A| = k \text{ and } A \text{ satisfies conditions 1. and 2.}\}$$

Lemma 23 *For some constant $\beta > 1$ (independent of k), there are at least $\frac{n_k}{\beta}$ sets of size k which satisfy conditions 1. and 2, i.e. $|R_k| \geq \frac{n_k}{\beta}$.*

From this lemma we derive,

$$\begin{aligned}
||x-y||_1 &\geq \sum_{j=p-\ell,k=2^j}^{p-1} \sum_{R_k} \frac{\alpha}{n_k}|d(x,A)-d(y,A)| \\
&\geq \sum_{j=p-\ell,k=2^j}^{p-1} \frac{n_k}{\beta}\frac{\alpha}{n_k}(\rho_{p-j}-\rho_{p-j-1}) \\
&= \frac{\alpha}{\beta}\sum_{j=p-\ell}^{p-1}(\rho_{p-j}-\rho_{p-j-1}) \\
&= \frac{\alpha}{\beta}\rho_\ell \\
&= \frac{\alpha}{4\beta}d(x,y).
\end{aligned}$$

Hence if we choose $\alpha = 4\beta$ then we have $||x-y||_1 \geq d(x,y)$. We now have to prove lemma 23.

Proof of lemma 23: Since $|\bar{B}(x,\rho_t)| < 2^t$, $|B(y,\rho_{t-1})| \geq 2^{t-1}$ and we are considering all sets of size k the following is a restatement of the lemma: Given disjoint sets P and Q with $a = |P| < 2^t$ and $b = |Q| \geq 2^{t-1}$, if E is the event that a uniformly selected A misses P and intersects Q then $\Pr[E] \geq \frac{1}{\beta}$. We calculate this probability as follows:

$$\begin{aligned}
\Pr[E] &= \frac{\binom{n-a}{k} - \binom{n-a-b}{k}}{\binom{n}{k}} \\
&= \frac{(n-a)!(n-k)!}{(n-a-k)!n!} - \frac{(n-a-b)!(n-k)!}{(n-a-b-k)!n!} \\
&= \frac{n-a}{n}\frac{n-a-1}{n-1}\cdots\frac{n-a-k+1}{n-k+1} - \\
&\quad \frac{n-a-b}{n}\frac{n-a-b-1}{n-1}\cdots\frac{n-a-b-k+1}{n-k+1} \\
&= \left(1-\frac{a}{n}\right)\left(1-\frac{a}{n-1}\right)\cdots\left(1-\frac{a}{n-k+1}\right) - \\
&\quad \left(1-\frac{a+b}{n}\right)\left(1-\frac{a+b}{n-1}\right)\cdots\left(1-\frac{a+b}{n-k+1}\right).
\end{aligned}$$

As an approximation (this can be made formal), we replace $(1-\frac{a}{n-j})$ by $e^{-a/n}$, and $(1-\frac{a+b}{n-j})$ by $e^{-(a+b)/n}$. Thus,

$$\Pr[E] \approx e^{-\frac{ak}{n}} - e^{-\frac{(a+b)k}{n}} \geq e^{-\frac{ak}{n}}\left(1-e^{-\frac{bk}{n}}\right).$$

This for example shows that if a, b and k are all $\sqrt{n}$ then this probability is a constant, which may seem a bit paradoxical. Using our bounds on a and b, we get

$$\begin{aligned} \Pr[E] &\approx e^{-\frac{ak}{n}}\left(1-e^{-\frac{bk}{n}}\right) \geq e^{-\frac{2^t 2^j}{n}}\left(1-e^{-\frac{2^{t-1}2^j}{n}}\right) \\ &= e^{-\frac{2^p}{n}}\left(1-e^{-\frac{2^{p-1}}{n}}\right) \geq e^{-1}\left(1-e^{-\frac{1}{4}}\right). \end{aligned}$$

We now choose $\beta \approx \left(e^{-1}\left(1-e^{-\frac{1}{4}}\right)\right)^{-1}$ and the proof is complete. $\square$

Bourgain's proof is not quite algorithmic since the dimension is exponential. Linial, London and Rabinovitch just sample uniformly with $n_k = O(\log n)$ and show that with high probability the embedding has the required properties. This follows from a Chernoff bound.

We have thus shown that the distortion c can be chosen to be $O(\log n)$. We can do even better by proving the following variant to Bourguain's theorem.

Theorem 24 *Let d be a metric on a set V of n points. Suppose that $T \subseteq V$ and $|T| = k$. Then there exists an embedding of d into ℓ_1 which satisfies:*

$$\begin{array}{ll} ||x-y||_1 \leq O(\log k) d(x,y) & \forall x, y \in V. \\ ||x-y||_1 \geq d(x,y) & \forall x, y \in T. \end{array}$$

In order to prove this theorem we restrict the metric to T and then embed the restricted metric. If we look at the entire vertex set V then the first part of the original proof still works.

This new theorem is enough to show that $\frac{\beta^*}{\alpha^*} \leq O(\log k)$ and to approximate the multicommodity cut to within $O(\log k)$. This result is best possible in the sense that we can have $\frac{\beta^*}{\alpha^*} = \Theta(\log k)$.

References

[1] S. Arora, C. Lund, R. Motwani, M. Sudan, and M. Szegedy. Proof verification and hardness of approximation problems. In *Proceedings of the 33rd Annual Symposium on Foundations of Computer Science*, pages 14–23, 1992.

[2] S. Arora and S. Safra. Probabilistic checking of proofs. In *Proceedings of the 33rd Annual Symposium on Foundations of Computer Science*, pages 2–13, 1992.

[3] Y. Aumann and Y. Rabani. An $O(\log k)$ approximate min-cut max-flow theorem and approximation algorithm. Manuscript, 1994.

[4] R. Bar-Yehuda and S. Even. A linear time approximation algorithm for the weighted vertex cover problem. *Journal of Algorithms*, 2:198–203, 1981.

[5] M. Bellare and M. Sudan. Improved non-approximability results. In *Proceedings of the 26th Annual ACM Symposium on Theory of Computing*, pages 184–193, 1994.

[6] N. Christofides. Worst-case analysis of a new heuristic for the travelling salesman problem. Technical Report 388, Graduate School of Industrial Administration, Carnegie Mellon University, Pittsburgh, PA, 1976.

[7] W. F. de la Vega and G. S. Luecker. Bin packing can be solved within $(1+\epsilon)$ in linear time. *Combinatorica*, 1(4), 1981.

[8] J. Edmonds. Maximum matching and a polyhedron with 0,1-vertices. *Journal of Research of the National Bureau of Standards B*, 69B:125–130, 1965.

[9] R. Fagin. Generalized first-order spectra, and polynomial-time recognizable sets. In R. Karp, editor, *Complexity of Computations*. AMS, 1974.

[10] H. N. Gabow. Data structures for weighted matching and nearest common ancestors with linking. In *Proceedings of the 1st ACM-SIAM Symposium on Discrete Algorithms*, pages 434–443, 1990.

[11] H. N. Gabow, M. X. Goemans, and D. P. Williamson. An efficient approximation algorithm for the survivable network design problem. In *Proceedings of the Third MPS Conference on Integer Programming and Combinatorial Optimization*, pages 57–74, 1993.

[12] H. N. Gabow and R. E. Tarjan. Faster scaling algorithms for general graph matching problems. Technical Report CU-CS-432-89, University of Colorado, Boulder, 1989.

[13] M. X. Goemans and D. P. Williamson. A general approximation technique for constrained forest problems. In *Proceedings of the 3rd Annual ACM-SIAM Symposium on Discrete Algorithms*, pages 307–316, 1992.

[14] M. X. Goemans and D. P. Williamson. Improved approximation algorithms for maximum cut and satisfiability problems using semidefinite programming. In *Proceedings of the 26th Annual ACM Symposium on Theory of Computing*, pages 422–431, 1994.

[15] A. Haken and M. Luby. Steepest descent can take exponential time for symmetric connection networks. *Complex Systems*, 2:191–196, 1988.

[16] D. Hochbaum. Approximation algorithms for set covering and vertex cover problems. *SIAM Journal on Computing*, 11:555–556, 1982.

[17] D. Hochbaum and D. Shmoys. Using dual approximation algorithms for scheduling problems: theoretical and practical results. *Journal of the ACM*, 34(1), Jan. 1987.

[18] N. Karmarkar and R. Karp. An efficient approximation scheme for the one-dimensional bin-packing problem. In *Proceedings of the 23rd Annual Symposium on Foundations of Computer Science*, 1982.

[19] T. Leighton and S. Rao. An approximate max-flow min-cut theorem for uniform multicommodity flow problems with applications to approximation algorithms. In *Proceedings of the 29th Annual Symposium on Foundations of Computer Science*, pages 422–431, 1988.

[20] N. Linial, E. London, and Y. Rabinovich. The geometry of graphs and some of its algorithmic applications. In *Proceedings of the 35th Annual Symposium on Foundations of Computer Science*, 1994.

[21] C. H. Papadimitriou and M. Yannakakis. Optimization, approximation, and complexity classes. *Journal of Computer and System Sciences*, 43:425–440, 1991.

[22] S. Poljak. Integer linear programs and local search for max-cut. Preprint, 1993.

[23] P. Raghavan. Probabilistic construction of deterministic algorithms: approximating packing integer programs. *Journal of Computer and System Sciences*, 37:130–143, 1988.

[24] P. Raghavan and C. D. Thompson. Randomized rounding: a technique for provably good algorithms and algorithmic proofs. *Combinatorica*, 7:365 – 374, 1987.