Randomized Algorithms

An Introduction through Unique Sink Orientations

Lecture Notes

Bernd Gärtner, ETH Zürich

February 13, 2004

Preface

This document contains lecture notes for the course *Randomized Algorithms* I have taught at ETH Zürich in the Winter term of 2003/2004. Although this course has been offered at ETH in the past (and will be offered in the future), it was a somewhat special year.

In the three previous years, the course was designed and held as a block course by Emo Welzl, as part of the international Graduate Program *Combinatorics, Geometry and Computation* (CGC). In December 2002, ETH decided to stop local funding for CGC, with all the consequences: no more interesting predoc students from all over the world, no more schools and seminars, and no more block courses.

Starting from Winter 2004, Angelika Steger, the new full professor in the Institute of Theoretical Computer Science, will take over the course as part of the recently established bachelor and master program of the Department.

And in between, it was me. Being a one-shot has the advantage of being free: there was no need to design a course that lasts for years, fits together with other courses, or even covers the classic material. (There was also no need to write lecture notes, but this is a different story.)

I made use of this freedom and taught a very specialized course, dealing with *unique sink orientations* of cubes, a topic of current research by a small group of people, some of them (including me) at ETH. Part of the material has never been covered in class before, and some of it has been developed specifically for the course. Under these circumstances, the lecture notes are not only a service to the students, but also a document collecting what I have learned myself in preparing the course.

Fortunately, the randomized techniques to deal with unique sink orientations are not so different from the techniques being used for other, more common problems. For example, game theory and the theory of random walks are important general tools that also apply to unique sink orientations; consequently, they are also covered in the course. The point I want to make is that unique sink orientations are fairly special objects, but the techniques applied to them during the course are of more general interest and at least partially justify the label *Randomized Algorithms*.

I would like to thank Péter Csorba and Kaspar Fischer for inspiring discussions, careful proofreading, and for providing written solutions; the lecture notes would not nearly be the same without them! Many thanks also to Ingo Schurr for his suggestions and help (in particular, his implementations) concerning the Markov chain for generating and counting unique sink orientations. Bernhard von Stengel helped to improve the game theory chapter. Finally, the students of the course have influenced the lecture notes through interesting questions and remarks, which I have tried to incorporate; thanks to all of you for sharing this experiment!

Contents

1 Introduction **4**
- 1.1 The Emperor's Maze . . . 4
  - 1.1.1 Keep Left? . . . 6
  - 1.1.2 Roll the Dice! . . . 6
- 1.2 The Maze Game Revisited . . . 7
- 1.3 Adversary Models . . . 9
- 1.4 Sources of Randomness . . . 9
- 1.5 Preview of the Course . . . 10

2 Smallest Enclosing Balls **13**
- 2.1 Basics . . . 13
- 2.2 Algorithm `LeaveItThenTakeIt` . . . 16
  - 2.2.1 Correctness . . . 17
  - 2.2.2 Runtime . . . 19
- 2.3 More Basics . . . 22
- 2.4 Algorithm `TakeItOrLeaveIt` . . . 25
  - 2.4.1 Runtime . . . 25

3 Unique Sink Orientations of Cubes **30**
- 3.1 Definition and Examples . . . 30
- 3.2 Smallest Enclosing Balls Revisited . . . 32
- 3.3 The Algorithmic Model . . . 33
- 3.4 Basic Theory . . . 36
- 3.5 The 2-dimensional Case . . . 39

4 Zero-Sum Games **43**
- 4.1 Basics . . . 43
- 4.2 Solving the Game . . . 46
- 4.3 Game Trees . . . 50
- 4.4 The Sequence Form . . . 52
- 4.5 Solving the Game in Sequence Form . . . 53
- 4.6 Yao's Theorem . . . 59

5 Random Walks **62**
5.1 Two Warm-Ups 62
5.2 The `RandomEdge` Algorithm 65
5.2.1 Morris's USO 67
5.2.2 `RandomEdge` on Morris's USO 70
5.3 Random Unique Sink Orientations 75
5.3.1 Markov Chains 76
5.3.2 A Markov Chain for USOs 80
5.4 Counting USOs with the Markov Chain 85

6 Acyclic Unique Sink Orientations **90**
6.1 The RandomFacet Algorithm 90
6.2 Analysis of `RandomFacet` 93

7 Solutions to Exercises **99**
7.1 Solutions to Chapter 1 99
7.2 Solutions to Chapter 2 103
7.3 Solutions to Chapter 3 109
7.4 Solutions to Chapter 4 112
7.5 Solutions to Chapter 5 114
7.6 Solutions to Chapter 6 118

Chapter 1

Introduction

Randomized algorithms are algorithms that flip coins in order to take certain decisions. This concept extends the classical model of deterministic algorithms and has become very useful and popular within the last twenty years. In many cases, randomized algorithms are faster, simpler, or just more elegant than deterministic ones. A well-known such algorithm—taught in many introductory computer science courses—is randomized Quicksort: the expected runtime of the randomized version is $O(n \log n)$ for all sequences of n numbers, while the naive deterministic variants require quadratic runtime in the worst case.

In this introductory chapter, we discuss just one very simple randomized algorithm, but in doing so, some important points become clear already. We will see that a randomized algorithm can be *provably* better than any deterministic algorithm for the same task.[1] Our example will also indicate that *two-player games* are useful in designing and analyzing randomized algorithms. The author hopes that the example itself—dealing with the Roman Emperor Determinatus and his captive, the Gaul Randomix—will not insult the mind of the educated reader.

1.1 The Emperor's Maze

The malicious Roman Emperor Determinatus plays a game with his captives. If the captive wins, he or she will be freed, but if Determinatus wins, the captive will be thrown to the lions. However, the rules are such that Determinatus believes he can always win.

The game takes place in the Emperor's underground maze which consists of *chambers* and *tunnels* connecting the chambers. The maze has the structure of the complete graph K_n: any two among the n chambers are connected by a tunnel. The maze has two special chambers, the *entrance* and the *exit*. Moreover, the tunnels are one-way (which is enforced by a number of $\binom{n}{2}$ guards, blocking for each tunnel exactly one of its two doorways). This means that once a tunnel has been traversed, there's no way back.

The positions of the guards thus determine an *orientation* of the K_n where the orientation of each edge indicates the direction in which the corresponding tunnel can be traversed.

[1] Such statements can be proved for specific models of computation; in the standard model that formalizes deterministic algorithms via *Turing machines* and randomized algorithms via *probabilistic Turing machines*, it is not known whether randomization substantially helps.

The night before the actual game takes place, Determinatus explains the rules to his latest captive, a Gaul called Randomix: in order to win the game, Randomix has to traverse the maze from entrance to exit, *without* going through *all* the chambers. The Emperor also says that he is being fair to Randomix by guaranteeing that

(i) the maze has at least three chambers, so that Randomix will not trivially lose,

(ii) the entrance can be *left* through all its incident tunnels,

(iii) the exit can be *entered* through all its incident tunnels, and

(iv) there is no danger of getting caught in a cycle, i.e. once a chamber is left, there's no way of coming back to it later.

Determinatus even brags about being extremely fair, because his guarantees ensure that even the most stupid captive will reach the exit after passing through at most $n-1$ tunnels (as we know, actually using $n-1$ tunnels would be fatal for the captive, though). Moreover, it is possible to reach the exit via just one tunnel. The only problem here is that the tunnels are long and narrow, and the chamber at the other end of the tunnel becomes only visible once it has been reached.

To summarize, what we are dealing with is an *acyclic* orientation of K_n, with the source corresponding to the entrance chamber and the sink to the exit chamber, see Figure 1.1 for an example.

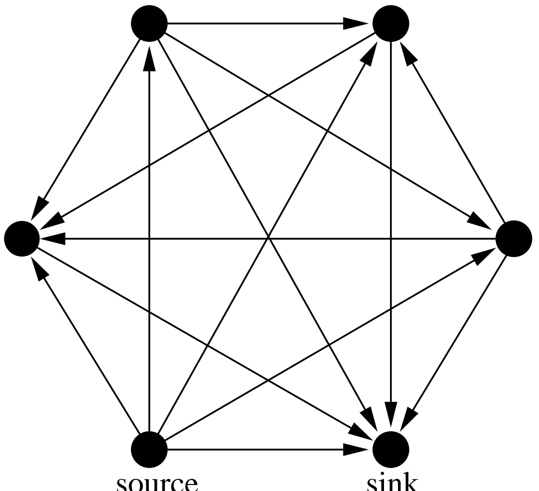


Figure 1.1: Acyclic orientation of K_6

After being informed about this setup, Randomix's spirits rise: wouldn't it be an extremely unlucky situation to go through all the chambers when in fact there is a direct tunnel to the exit from *any* of the chambers? But then Determinatus declares that there is one final rule: Randomix has to commit *now* to an algorithm for finding the exit, and the guards would make sure that he is actually following this algorithm the next day in the maze.

1.1.1 Keep Left?

Hastily, Randomix starts thinking about algorithms. One that immediately comes to his mind is the following: within a chamber, always take the leftmost one among the non-blocked tunnels.[2] In the orientation of Figure 1.1, this would indeed be a winning strategy, because it traverses only three out of the six chambers, see Figure 1.2 (left). But it quickly becomes clear to Randomix that there is also an orientation on which this algorithm would take him directly to the lions den, see Figure 1.2 (right). Even worse, he realizes that Determinatus, who gets to know the algorithm in advance, would of course confront him with exactly this bad orientation! Similarly, a Keep-Right strategy doesn't work: as Determinatus is free to place source and sink wherever he wants, he could simply interchange source and sink and flip all edge directions in Figure 1.2 (right) to arrive at an orientation on which Keep-Right runs through all the chambers.

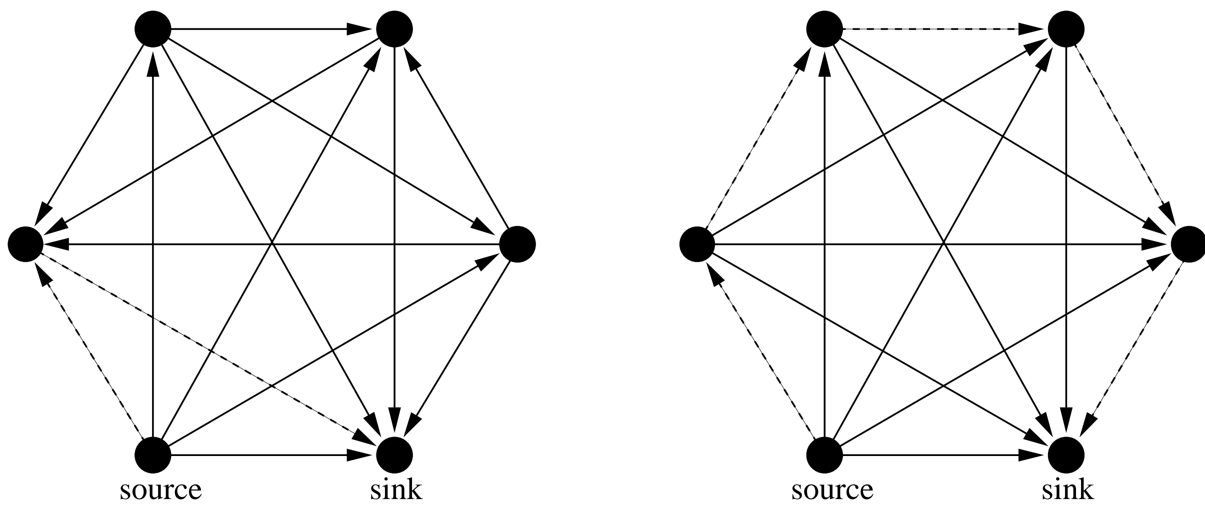


Figure 1.2: Algorithm Keep-Left

For a few seconds, Randomix tries to come up with more sophisticated algorithms (in every second chamber, keep right instead of left; always try to stay as close to the 'middle' tunnel as possible, . . .), but then he gives up, knowing that all this won't help (Exercise 1.1 asks you to prove this in a general setting).

1.1.2 Roll the Dice!

Just as Determinatus is getting impatient waiting for Randomix's algorithm, the Gaul realizes that the Emperor is just a silly Roman, and that he knows a way to survive with high probability. He takes the papyrus roll provided by the Emperor for this purpose and writes down his algorithm in words (pseudocode had not been invented at that time):

[2]What this algorithm does obviously depends on the geometric layout of the maze; let's assume that the Emperor is a fan of symmetry and that the layout really looks like in Figure 1.1.

I will take along an $(n-1)$-sided die with numbers 1 through $n-1$ on its faces. In every chamber I reach, I roll the dice until some number k appears that is no larger than the number of non-blocked tunnels. Then I take the k-th non-blocked tunnel from the left.

The Emperor is baffled: although he *knows* Randomix's algorithm, and he can also make his guards check that Randomix behaves according to it, he does *not* know which tunnels Randomix will choose. In contrast to what Randomix thinks, Determinatus is only malicious but not stupid, so he realizes that whatever orientation he will provide the next day, he will be forced to free Randomix with probability $1 - 1/(n-1)!$ (Exercise 1.2).

Historical note: Determinatus tried his best in not letting Randomix escape his fate, by downsizing the beautiful maze he was so proud of to the minimum of only three chambers that he had guaranteed to Randomix. Since $1/(3-1)! = 1/2$, it was, after all, a fair game. Still, as the first captive during the long regency of Determinatus, Randomix won, and the Emperor subsequently enrolled in a *randomized algorithms* class.

1.2 The Maze Game Revisited

An acyclic orientation of the K_n uniquely corresponds to a total order of the vertices. This means, we can number the vertices from 1 (the sink) to n (the source) so that there is an oriented edge (v, w) iff v's number is higher than w's (Exercise 1.3). Under this scheme, the number of a vertex is one larger than its outdegree, see Figure 1.3.

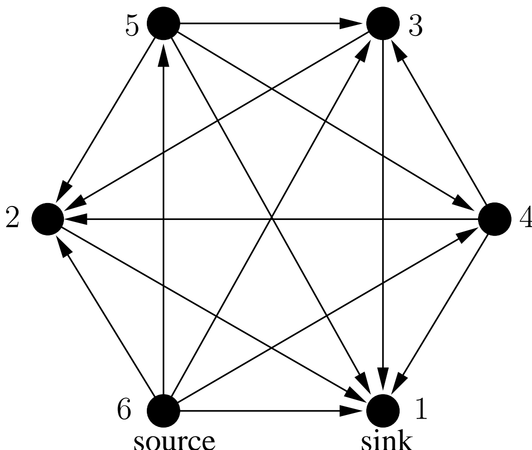


Figure 1.3: Acyclic orientation of K_6 with induced vertex numbers

Let us interpret Randomix's algorithm as a process on natural numbers. Given that he is currently in the chamber with number i, there are $i-1$ non-blocked tunnels, leading to the $i-1$ chambers with numbers $1, \ldots, i-1$; by rolling the dice, he will end up in any of these chambers with the same probability. From the chamber he actually reaches, the process continues in the same fashion.

In pseudocode, we can write this down as a recursive procedure `Maze`.

Algorithm 1.1

```
Maze(i):
  IF i > 1 THEN
    choose j ∈ {1, ..., i − 1} uniformly at random
    Maze(j)
  END
```

What is the *expected* number $f(n)$ of chambers Randomix has to go through in a maze with n chambers? This is the same as the expected number of comparisons of the form '$i > 1$' throughout a call of `Maze`(n), because exactly one such comparison is done for every chamber that is visited. We conclude that

$$\begin{aligned} f(1) &= 1, \\ f(2) &= 2, \\ f(i) &= 1 + \frac{1}{i-1}\sum_{j=1}^{i-1} f(j), \quad i > 1. \end{aligned}$$

For $i > 2$, this yields

$$(i-1)f(i) - (i-2)f(i-1) = 1 + f(i-1),$$

equivalently

$$f(i) = \frac{1}{i-1} + f(i-1) = \sum_{j=2}^{i-1}\frac{1}{j} + f(2) = H_{i-1} + 1,$$

with

$$H_n := \sum_{j=1}^{n}\frac{1}{j}$$

being the n-th *Harmonic number*. We conclude that in a maze with $n \geq 1$ chambers, Randomix goes through an expected number of

$$H_{n-1} + 1 < \ln(n-1) + 2$$

chambers, much less than what Determinatus would like him to.

Throughout the course, we will be concerned with *expected* runtimes of algorithms. It is important that the expectations we compute hold for *all* inputs. In our case, we have made no specific assumption about the acyclic orientation, meaning that the bound of $H_{n-1} + 1$ holds for *all* of them.[3] In this sense, the expectations we compute are upper bounds for the expected performance of the algorithms *in the worst case*.

[3] This is no real surprise here, because all these orientations are isomorphic, see also Exercise 1.3.

1.3 Adversary Models

The maze game is actually a game between the algorithm (of Randomix) and the *adversary* (Determinatus) who wants to make the algorithm perform as poorly as possible. In this interpretation, the adversary is responsible for providing the worst case under which we want to analyze the algorithm's performance. Exercise 1.1 shows that the adversary can indeed force the worst possible number of n chambers against any deterministic algorithm.

For randomized algorithms (like the one Randomix is proposing), it is important that the adversary is *oblivious*, and we will always consider this case. It means that in constructing a bad input for the algorithm, the adversary may look at the specification of the algorithm (Determinatus indeed does, to no avail), but *not* at the actual random choices the algorithm makes.

In our case, Determinatus confronts Randomix with a maze whose structure does *not* change during the game. If he were unfair enough to watch the outcomes of Randomix's die and then quickly reposition guards in order to make the actual outcome a bad outcome for Randomix, the Gaul would be doomed. Under such a powerful adversary model, randomized algorithms lose their advantage over deterministic ones.

In the *offline* scenario where the algorithm gets the complete input and then starts its computations, the oblivious adversary model makes perfect sense: once the algorithm has taken control, the input won't change depending on subsequent computations. In the *online* world, where decisions have to be taken without knowing parts of the input, the stronger adversary model might be interesting. We will not consider online problems during this course, so it's justified to stick with oblivious adversaries.

1.4 Sources of Randomness

While Randomix has a die at his disposal, the computer hasn't. The usual way out in implementing randomized algorithms consists of using a *pseudorandom number generator*. This is a deterministic procedure for generating a sequence of numbers that 'look' random. There are two problems with this: first, the adversary will know the generator that is being used from the specification of the algorithm, so there is no unpredictability, let alone true randomness, left in the game.

The practical justification for using pseudorandom numbers is that in real life, there is no adversary. In this case, it seems that simulating true randomness by pseudorandomness would result in similar behavior for all practical purposes.

But this is not true. Many pitfalls in using pseudorandom numbers have been published, and in some cases the deviations from the truly random behavior are dramatic. This even holds for the 'best' generators that are available, but on top of that, there are bad generators. The most famous example is the generator `randu` that was widely used in the sixties. It generates pseudorandom floating point numbers in the interval $[0, 1]$. When you plot the sequence $x_1, x_2, \ldots$ of numbers output by `randu`, it appears to be fairly random, but when you plot the triples $(x_1, x_2, x_3), (x_2, x_3, x_4), \ldots$ in the unit cube, all these triples live in only fifteen parallel planes, not what you expect from random triples, see Figure 1.4.

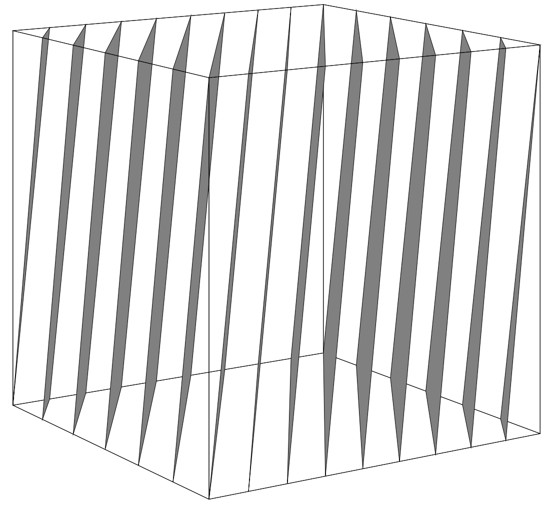


Figure 1.4: The fifteen planes of `randu`

Can we get hold of truly random numbers to be used in a computer program? This is impossible to answer, because one cannot test whether a given finite sequence of numbers has been generated from a truly random distribution. This means that all we can do is trust the distribution to be truly random. In fact, there are websites from which you can download random numbers, extracted for example from radioactive decay (`www.fourmilab.ch/hotbits/`) or atmospheric noise (`www.random.org`). While these numbers pass all kinds of statistical tests, you can never be sure that they are really random. For example, there has been the question of whether the numbers provided by `www.random.org` have shown unusual behavior on September 11, 2001. (The answer seems to be 'no', but the page explaining this is a little spooky.)

In the course, we will sweep the issue of random number generation under the rug. We will analyze our algorithms with the assumption that we can (like Algorithm 1.1 does it) sample a number uniformly at random from a finite set, at some constant cost. What we *will* care about in some cases is the *number* of random numbers we need, because it is an interesting theoretical question how many random resources are necessary to perform a certain task within a certain time.

1.5 Preview of the Course

The question of how fast we can find a sink in a graph whose edge orientations become known only by visiting incident vertices, will be central to the course. We will consider it in connection with different graphs and different classes of orientations. It turns out that many techniques useful for the design and analysis of randomized algorithms can be developed in this scenario. We will see concrete applications (the Emperor's Maze is certainly not the most important one), but we will also apply the techniques to other problems not having to do with sink-finding.

Bibliographic Remarks

The 'standard reference' for randomized algorithms is the very good book by Raghavan and Motwani [11]. Another very interesting book, focusing mainly on randomized algorithms in the context of computational geometry, has been written by Mulmuley [12].

Exercises

Exercise 1.1 *Let $H = (V, E), E \subseteq \binom{V}{2}$ be some undirected graph on the vertex set V. An* orientation *of H can be specified by a directed graph $G = (V, D), D \subseteq V \times V$ such that D contains for every edge $\{v, w\} \in E$ exactly one of the pairs (v, w) and (w, v).*

Suppose that $G = (V, D)$ is an orientation of H which contains exactly one sink, and the goal is to find this sink. In the vertex access model*, information about D can only be obtained through an oracle which for any given vertex $v \in V$ reveals the set of neighbors of v along outgoing edges,*

$$\text{out}(v) := \{w \in V \mid (v, w) \in D\}.$$

Whenever we call the oracle with v, we say that v is evaluated*.*

Given the graph H and any orientation G of H with exactly one sink, specified by a vertex evaluation oracle, we are interested in the smallest number of vertex evaluations a deterministic algorithm needs in the worst case in order to evaluate *the sink[4] of G.*

Let $t(H)$ be this minimum number. For example, if $|V| = 1$, then $t(H) = 1$. If H consist of just two vertices and the edge spanned by them, then $t(H) = 2$, because in one of the two orientations G of H, the first vertex evaluated by the algorithm is not the sink.

(i) Prove that $t(K_n) = n$, where K_n is the complete graph on n vertices.

(ii) Prove that $t(C_n) = 2^{n-1} + 1$, where C_n is the vertex-edge graph of the n-dimensional cube.

(iii) Assume that H has a vertex cover *(set of vertices W such that every edge contains a vertex from W) of size k. Prove that $t(H) \leq k + 1$ in this case. Are there graphs H for which $t(H)$ is smaller than one plus the size of the smallest vertex cover?*

Exercise 1.2 *For $n \geq 1$, let X_n be the random variable for the number of comparisons of type '$i > 1$' in Algorithm 1.1. Prove that*

$$\text{prob}(X_n = n) = \frac{1}{(n-1)!}.$$

Prove an explicit formula for

$$\text{prob}(X_n = i), \quad 1 \leq i < n.$$

*(**Hint:** Browse Emo Welzl's* Basic Examples of Probabilistic Analysis*!)*

[4] Just *knowing* the vertex which is the sink is not enough; we'll see later that this simplifies certain computations.

Exercise 1.3 *Let $H = (V, E)$ be an undirected graph with $|V| = n$ vertices and let $G = (V, D)$ be an* acyclic *orientation of it as defined in Exercise 1.1.*

(i) Prove that there is a bijection

$$\sigma : V \to [n] := \{1, \ldots, n\}$$

such that for all $v, w \in V$, $(v, w) \in D$ implies $\sigma(v) > \sigma(w)$. Such a σ is called a topological sorting *of G.*

(ii) Prove that for given H and acyclic orientation G of H, the following statements are equivalent.

(a) G has a unique *topological sorting,*

(b) for all $v, w \in V$, there is either *a directed path from v to w* or *a directed path from w to v,*

(c) G has a directed Hamiltonian path (a directed path that visits every vertex exactly once).

(iii) Conclude that K_n has $n!$ acyclic orientations, each of them corresponding to a unique topological sorting.

Chapter 2

Smallest Enclosing Balls

In this chapter, we study a classical problem from computational geometry, the problem of computing the ball $\bigcirc(S)$ of smallest radius (equivalently, smallest volume) that contains a given set S of n points in d-dimensional Euclidean[1] space $\mathbb{E}^d$, see Figure 2.1.

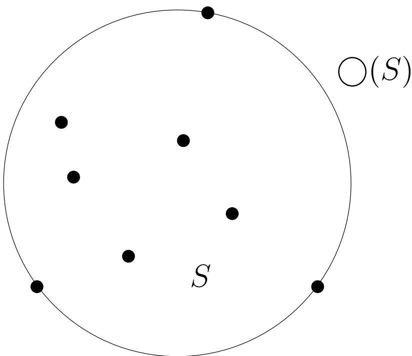


Figure 2.1: The smallest enclosing ball of a point set

2.1 Basics

We consider the following generalization of the smallest enclosing ball problem (in this chapter, S is always a finite set).

Definition 2.1 *Let $R \subseteq S \subseteq \mathbb{E}^d$. If there is a unique closed ball $\mathcal{B}$ of smallest radius that contains S and has the points in R on its boundary (we also say that $\mathcal{B}$* goes through *R and* covers *S), this ball $\mathcal{B}$ is denoted by $\bigcirc(R, S)$. $\bigcirc(\emptyset, S)$ will be abbreviated as $\bigcirc(S)$. Moreover, we set $\bigcirc(\emptyset) := \emptyset$, the 'empty' ball.*

Lemma 2.2 *With $R \subseteq S$ as above, $\bigcirc(R, S)$ exists if there is some ball $\mathcal{B}$ through R that covers S.*

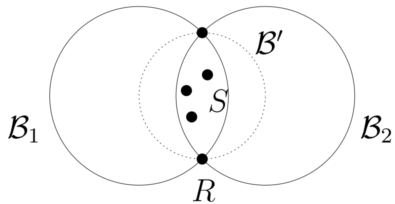


Figure 2.2: Proof of Lemma 2.2

Proof. A standard compactness argument from calculus shows that if there is some ball through R, covering S, then there is also a smallest one.[2] It remains to show that the smallest ball is unique. Assume there were two smallest balls $\mathcal{B}_1, \mathcal{B}_2$ through R that cover S. Then the picture would look like in Figure 2.2, and we could find an even smaller ball $\mathcal{B}'$ (dotted) through R that covers S, a contradiction. Exercise 2.1 asks you to prove the existence of $\mathcal{B}'$ formally, here we are satisfied with the geometric intuition. □

Because for any finite set S, there is always a ball that covers S, we obtain

Corollary 2.3 *For $S \subseteq \mathbb{E}^d$, $\bigcirc(S)$ exists.*

In general, the sets R and S can be considered as *constraints* under which we want to find a ball with smallest radius. Because adding constraints cannot decrease the smallest radius, the following is obvious.

Fact 2.4 *Let $R \subseteq R', S \subseteq S' \subseteq \mathbb{E}^d$. Provided, the respective balls exist, we get*

$$\bigcirc(R, S) \leq \bigcirc(R', S'),$$

where balls are compared by radius. (Note that if $\bigcirc(R, S)$ and $\bigcirc(R', S')$ have the same radius, they are both smallest balls through R, covering S, so they must be equal by Lemma 2.2.)

The next lemma will be the basis of our randomized algorithm(s) for the efficient computation of $\bigcirc(S)$.

Lemma 2.5 *Let $R \subseteq S \subseteq \mathbb{E}^d, p \in S \setminus R$ such that $\bigcirc(R, S)$ (and consequently also $\bigcirc(R, S \setminus \{p\})$) exists. If*

$$p \notin \bigcirc(R, S \setminus \{p\}),$$

then p is on the boundary of $\bigcirc(R, S)$, equivalently

$$\bigcirc(R, S) = \bigcirc(R \cup \{p\}, S).$$

Proof. Assume that $\bigcirc(R, S)$ does not have p on its boundary, but in its interior. Because p is outside of $\bigcirc(R, S \setminus \{p\})$, the two balls are different, and the situation is as in Figure 2.3.

[1] For us, $\mathbb{E}^d$ is just the vector space $\mathbb{R}^d$, equipped with the standard scalar product $\langle p, q \rangle = p^T q$ that allows us to measure the Euclidean norm $\|p\| = \sqrt{\langle p, p \rangle}$ of a point.

[2] The set of centers of balls through R, covering S, whose radius is at most the radius of $\mathcal{B}$, is a compact set. The function that maps each such center c to the radius of the smallest ball with center c through R that covers S, is a continuous function which therefore attains its minimum over the compact set. The corresponding center is the center of some smallest ball through R, covering S.

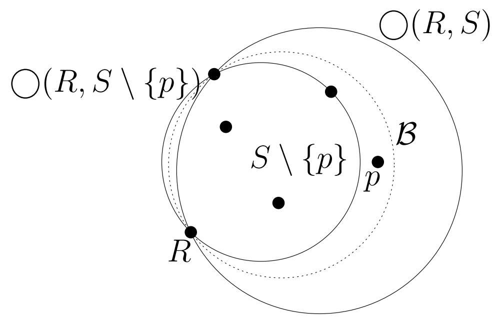


Figure 2.3: Proof of Lemma 2.5

Just like in the proof of Lemma 2.2, we can use Exercise 2.1 and slightly shrink $\bigcirc(R, S)$ to obtain a ball $\mathcal{B}$ (dotted) which is smaller than $\bigcirc(R, S)$, but still goes through R and covers S, which is a contradiction. $\square$

For the remainder of this chapter, we make a *general position assumption*. Although the algorithms (can be made to) work without it, general position simplifies the correctness proofs and gives us more time for the performance analysis.

Assumption 2.6 *For all sets $F \subseteq S$ such that $\bigcirc(F, F)$ exists, and for all points $p \in S \setminus F$, $\bigcirc(F, F)$ does* not *go through p.*

This assumption forbids *degenerate* situations like in Figure 2.4. In the plane, it implies that no ball goes through four points, but the left part of the figure shows that even balls through three points may be degenerate. We will point out where we make use of the assumption.

General position can be achieved by a technique called *symbolic perturbation* that independently moves the points by infinitesimal amounts, so that the degeneracies disappear. As indicated above, general position is convenient for us, but not necessary in order to establish the results of this chapter.

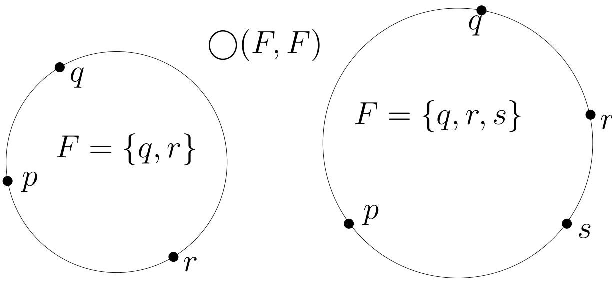


Figure 2.4: The smallest ball through F has another point p on its boundary

2.2 Algorithm `LeaveItThenTakeIt`

In this section, we develop our first algorithm `LeaveItThenTakeIt`(R, S) for computing $\bigcirc(R, S)$.

Precondition 2.7

(i) $\bigcirc(R, S)$ *exists, and*

(ii) R *is* affinely independent*, meaning that whenever there is a sequence of real numbers* $(\lambda_p)_{p \in R}$ *such that*

$$\begin{array}{rcl} \sum_{p \in R} \lambda_p p & = & 0, \\ \sum_{p \in R} \lambda_p & = & 0, \end{array}$$

then $\lambda_p = 0$ *for all* $p \in R$*.*

The empty set is affinely independent. For nonempty sets, affine independence is closely related to linear independence. The following is not difficult to verify (Exercise 2.2).

Fact 2.8 *For* $R \subseteq \mathbb{E}^d$, $R \neq \emptyset$*, the following statements are equivalent.*

(i) R *is affinely independent.*

(ii) For all $p \in R$*, the set of vectors* $\{q - p \mid q \in R \setminus \{p\}\}$ *is linearly independent.*

(iii) For some $p \in R$*, the set of vectors* $\{q - p \mid q \in R \setminus \{p\}\}$ *is linearly independent.*

The important consequence for us is

Corollary 2.9 *Let* $R \subseteq \mathbb{E}^d$ *be affinely independent. Then* $|R| \leq d + 1$.

Under the preconditions above, `LeaveItThenTakeIt`(R, S) will return a set F, satisfying the following

Postcondition 2.10

(i) $R \subseteq F \subseteq S$*,*

(ii) $\bigcirc(F, F) = \bigcirc(R, S) = \bigcirc(R, F)$*, and*

(iii) F *is affinely independent*

We call such a set F a *basis* of the pair (R, S).[3] Moreover, Assumption 2.6 implies that F is the *unique* basis.[4] Exercise 2.3 shows that $\bigcirc(F, F)$ is easy to compute in case F is affinely independent. The algorithm internally makes use of this fact and can actually be assumed to deliver $\bigcirc(F, F)$ along with F. Here is the algorithm.

[3]By now, it is not clear that every pair (R, S) which satisfies Precondition 2.7 has a basis at all; this will follow from the algorithm's correctness proof below.

[4]Assume (R, S) has two different bases F, F'. Then there is a point $p \in F' \setminus F$, say, and because of $\bigcirc(F, F) = \bigcirc(F', F')$, p lies on the boundary of $\bigcirc(F, F)$, a violation of the general position assumption.

Algorithm 2.11

```
LeaveItThenTakeIt(R, S):
  IF R = S THEN
     F := R
  ELSE
     choose p ∈ S \ R uniformly at random
     F := LeaveItThenTakeIt(R, S \ {p})
     IF p ∉ ◯(F, F) THEN
        F := LeaveItThenTakeIt(R ∪ {p}, S)
     END
  END
  RETURN F
```

2.2.1 Correctness

In order to prove correctness of the algorithm, we use induction on the size of $S \setminus R$. If $|S \setminus R| = 0$, the algorithm outputs $F = R = S$, and Postcondition 2.10 holds by affine independence of R. For $|S \setminus R| > 0$, the preconditions on (R, S) and Lemma 2.5 ensure that $\bigcirc(R, S \setminus \{p\})$ and—if `LeaveItThenTakeIt`$(R \cup \{p\}, S)$ is called—$\bigcirc(R \cup \{p\}, S)$ exist. We conclude that the preconditions hold for `LeaveItThenTakeIt`$(R, S \setminus \{p\})$, and using the induction hypothesis, the call returns an affinely independent set F such that

$$\bigcirc(F, F) = \bigcirc(R, S \setminus \{p\}) = \bigcirc(R, F).$$

Case (a) $p \in \bigcirc(F, F)$. Then $\bigcirc(F, F)$ is a ball through R that covers S, hence

$$\bigcirc(R, S) \leq \bigcirc(F, F) = \bigcirc(R, S \setminus \{p\}) \leq \bigcirc(R, S),$$

where the latter inequality is Fact 2.4. Therefore, $\bigcirc(F, F) = \bigcirc(R, S)$ and the postconditions hold.

Case (b) $p \notin \bigcirc(F, F)$. If we can show that $R \cup \{p\}$ is affinely independent, the preconditions for `LeaveItThenTakeIt`$(R \cup \{p\}, S)$ are satisfied, so by induction, the call returns an affinely independent set F with

$$\bigcirc(F, F) = \bigcirc(R \cup \{p\}, S) = \bigcirc(R \cup \{p\}, F).$$

By Lemma 2.5, $\bigcirc(R \cup \{p\}, S) = \bigcirc(R, S)$ in this case, so

$$\bigcirc(F, F) = \bigcirc(R, S) \tag{2.1}$$

follows. It remains to prove that

$$\bigcirc(F, F) = \bigcirc(R, F)$$

in order to establish the postcondition. Assume this does not hold. Then $\bigcirc(F, F)$ and $\bigcirc(R, F)$ are different balls that both go through R and cover F; in addition, $\bigcirc(F, F)$ covers $S \setminus F$, and by Assumption 2.6, no point in $S \setminus F$ is on the boundary of $\bigcirc(F, F)$. The situation is therefore as in Figure 2.5, and we can for the third time invoke Exercise 2.1: slightly shrinking $\bigcirc(F, F)$ yields a ball $\mathcal{B}$ (dotted) which is smaller than $\bigcirc(F, F)$, but still goes through R and covers S, a contradiction to (2.1).

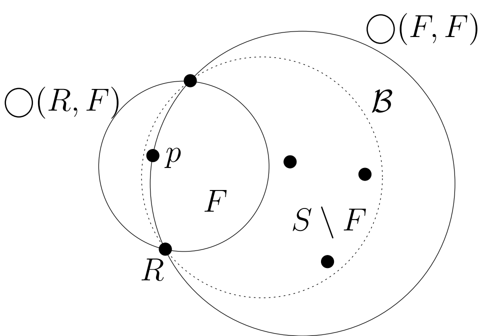


Figure 2.5: Proof of Postcondition 2.10 in Case (b)

It remains to prove affine independence of $R \cup \{p\}$ if $p \notin \bigcirc(F, F) = \bigcirc(R, S \setminus \{p\})$. For this, let us consider real values $\lambda_q, q \in R \cup \{p\}$ such that

$$\sum_{q \in R \cup \{p\}} \lambda_q q = 0, \tag{2.2}$$

$$\sum_{q \in R \cup \{p\}} \lambda_q = 0. \tag{2.3}$$

Let c (c', respectively) and ρ (ρ', respectively) be center and squared radius of $\bigcirc(R, S \setminus \{p\})$ ($\bigcirc(R \cup \{p\}, S)$, respectively). We know that

$$\rho = \|q - c\|^2 = q^T q - 2c^T q + c^T c, \quad q \in R, \tag{2.4}$$

$$\rho < \bar{\rho} := \|p - c\|^2 = p^T p - 2c^T p + c^T c, \tag{2.5}$$

$$\rho' = \|q - c'\|^2 = q^T q - 2c'^T q + c'^T c', \quad q \in R \cup \{p\}. \tag{2.6}$$

Using (2.2) and (2.3), equations (2.4) and (2.5) imply

$$\lambda_p(\bar{\rho} - \rho) = \sum_{q \in R \cup \{p\}} \lambda_q q^T q,$$

while (2.6) yields

$$0 = \sum_{q \in R \cup \{p\}} \lambda_q q^T q.$$

It follows that $\lambda_p = 0$, and because R is affinely independent, we also get $\lambda_q = 0, q \in R$. This means, $R \cup \{q\}$ is affinely independent.

2.2.2 Runtime

While the correctness of Algorithm 2.11 holds for any choice of p, its efficiency crucially depends on p being chosen at random (Exercise 2.5).

Each call of `LeaveItThenTakeIt` performs a *basis computation*[5] ('$F := R$') or a *violation test* ('$p \notin \bigcirc(F,F)$'), and only constant-time operations otherwise (excluding the operations within the recursive calls). This means, the total number of basis computations and violation tests throughout all recursive calls is a good measure of the algorithm's performance (Exercise 2.4 shows that it would be enough to count the violation tests.)

Violation tests. Let $t_k(n)$ be the maximum *expected* number of violation tests in a call to `LeaveItThenTakeIt`(R,S), where (R,S) satisfies Precondition 2.7, $|S \setminus R| = n$ and $d+1-|R| = k$.[6] Note that by Corollary 2.9, k is a nonnegative number. We get

$$t_k(0) = 0, \quad (2.7)$$

$$t_0(n) = t_0(n-1)+1, \quad n > 0. \quad (2.8)$$

While (2.7) is obvious, (2.8) requires a little argument: if $|R| = d+1$ already, there cannot be a second recursive call with parameters $(R \cup \{p\}, S)$, because the correctness proof shows that in this case, $R \cup \{p\}$ would have to be affinely independent. This is impossible for a set with $d+2$ elements.

Another way to look at it is that in case of $|R| = d+1$, there is a unique ball through R—this follows from Exercise 2.3(ii). Therefore, if $\bigcirc(R,S)$ exists (which is our precondition), we must have $\bigcirc(R,S) = \bigcirc(R,R)$, meaning that we will never find a point $p \in S \setminus R$ outside of $\bigcirc(R,R)$.

For $k, n > 0$, we get

$$t_k(n) \leq t_k(n-1) + 1 + p_k(n) t_{k-1}(n-1), \quad (2.9)$$

where $p_k(n)$ is the maximum probability of the event '$p \notin \bigcirc(F,F)$' in a call to `LeaveItThenTakeIt`(R,S), with (R,S) as in the definition of $t_k(n)$. The following is the important

Observation 2.12 $p_k(n) \leq k/n$.

Proof. If $p \notin \bigcirc(F,F) = \bigcirc(R, S\setminus\{p\})$, then p is contained in the unique basis B of (R,S). If not, Postcondition 2.10 and Fact 2.4 would yield

$$\bigcirc(R,S) = \bigcirc(B,B) = \bigcirc(R,B) \leq \bigcirc(R, S\setminus\{p\}) \leq \bigcirc(R,S),$$

hence $\bigcirc(F,F) = \bigcirc(R,S\setminus\{p\}) = \bigcirc(R,S)$, which is a contradiction to $p \notin \bigcirc(F,F)$. It follows that

$$\text{prob}(p \notin \bigcirc(F,F)) \leq \text{prob}(p \in B) = \frac{|B \setminus R|}{|S \setminus R|} \leq \frac{d+1-|R|}{n} = \frac{k}{n}.$$

[5] we call this *computation*, because in an actual implementation, $\bigcirc(F,F)$ would be computed and returned along with F; subsequent violation tests with $\bigcirc(F,F)$ are then very cheap.

[6] for fixed (R,S), the number of violation tests is a random variable $X(R,S)$; then we define $t_k(n) := \max\{E(X(R,S)) \mid (R,S) \text{ satisfies Precondition 2.7}, |S\setminus R| = n \text{ and } d+1-|R| = k\}$. This maximum exists, because there are only finitely many *combinatorially different* pairs (R,S).

$\square$

Lemma 2.13 *For* $k, n \geq 0$,

$$t_k(n) \leq c_k n,$$

where

$$c_k = \begin{cases} 1, & k = 0, \\ 1 + kc_{k-1}, & k > 0 \end{cases}.$$

Proof. We proceed by induction over n, noting that the bound holds for $n = 0$ by (2.7) and for $k = 0$ by (2.8). For $n, k > 0$, we inductively obtain

$$\begin{aligned} t_k(n) &\leq t_k(n-1) + 1 + \frac{k}{n} t_{k-1}(n-1) \\ &\leq c_k(n-1) + 1 + \frac{k}{n} c_{k-1}(n-1) \\ &\leq c_k(n-1) + 1 + kc_{k-1} \\ &= c_k n. \end{aligned}$$

$\square$

Corollary 2.14 *For* $k, n \geq 0$,

$$t_k(n) \leq k! \sum_{i=0}^{k} \frac{1}{i!}\, n = \begin{cases} n, & k = 0, \\ \lfloor ek! \rfloor\, n, & k > 0 \end{cases},$$

where $e \approx 2.718$ *is the Euler constant.*

Proof. For $k = 0$, the statement is immediate. For $k > 0$, we get

$$\frac{c_k}{k!} = \frac{1}{k!} + \frac{c_{k-1}}{(k-1)!} = \sum_{i=1}^{k} \frac{1}{i!} + \frac{c_0}{0!} = \sum_{i=0}^{k} \frac{1}{i!}.$$

This implies

$$c_k = k! \sum_{i=0}^{k} \frac{1}{i!} = ek! - \sum_{i=k+1}^{\infty} \frac{k!}{i!} = ek! - \varepsilon,$$

where $\varepsilon < 1$ for $k > 0$. Because c_k is an integer, the bound follows. $\square$

It follows that for *fixed* dimension d, the expected number of violation tests is *linear* in $|S|$.

Theorem 2.15 *Let* $S \subseteq \mathbb{E}^d$ *be a set of* n *points. Algorithm* `LeaveItThenTakeIt`$(\emptyset, S)$ *computes* $\bigcirc(S)$ *with an expected number of at most*

$$t_{d+1}(n) \leq \lfloor e(d+1)! \rfloor\, n$$

violation tests.

Basis computations. Let $b_k(n)$ be the maximum expected number of basis computations in a call to `LeaveItThenTakeIt`(R, S), where (R, S) satisfies Precondition 2.7, $|S \setminus R| = n$ and $d + 1 - |R| = k$. With the same arguments as for $t_k(n)$, we get $b_k(0) = b_0(n) = 1$ and

$$b_k(n) \leq b_k(n-1) + \frac{k}{n} b_{k-1}(n-1), \quad k, n > 0. \tag{2.10}$$

With

$$B_{k+1}(n+1) := b_k(n) \frac{n!}{k!},$$

(2.10) is equivalent to

$$B_{k+1}(n+1) \leq n B_{k+1}(n) + B_k(n),$$

which looks like the recurrence for the *cycle number*[7] $\left[\begin{array}{c} n+1 \\ k+1 \end{array} \right]$:

$$\left[\begin{array}{c} n+1 \\ k+1 \end{array} \right] = n \left[\begin{array}{c} n \\ k+1 \end{array} \right] + \left[\begin{array}{c} n \\ k \end{array} \right].$$

Checking the base cases reveals that in general, $B_{k+1}(n+1) \not\leq \left[\begin{array}{c} n+1 \\ k+1 \end{array} \right]$: we have

$$B_{k+1}(1) = \frac{1}{k!} > \left[\begin{array}{c} 1 \\ k+1 \end{array} \right] = 0, \quad k > 0.$$

Still, $b_k(n)$ can be bounded via cycle numbers. Inductively, one can prove

Lemma 2.16 *For $k, n \geq 0$,*

$$b_k(n) \leq \frac{1}{n!} \sum_{i=0}^{k} \binom{k}{i} i! \left[\begin{array}{c} n+1 \\ i+1 \end{array} \right].$$

Using the inequality

$$\left[\begin{array}{c} n+1 \\ i+1 \end{array} \right] \leq \frac{n!}{i!} (H_n)^i$$

(easily provable by induction as well, and also to be found in Emo Welzl's reading assignments), we obtain

$$b_k(n) \leq \sum_{i=0}^{k} \binom{k}{i} (H_n)^i = (1 + H_n)^k.$$

Theorem 2.17 *Let $S \subseteq \mathbb{E}^d$ be a set of n points. Algorithm* `LeaveItThenTakeIt`$(\emptyset, S)$ *computes $\bigcirc(S)$ with an expected number of at most*

$$b_{d+1}(n) \leq (1 + H_n)^{d+1}$$

basis computations.

---

[7] $\left[\begin{array}{c} n \\ k \end{array} \right]$ is the number of permutations of n elements with k cycles, see for example Emo Welzl's reading assignments.

This bound is remarkable, because it shows that the expected number of basis computations is not only smaller than the expected number of violation tests, plus two (which we know from Exercise 2.4), but *much* smaller: it is only polylogarithmic instead of linear. One basis computation can be done in time $O(d^3)$ (by Exercise 2.3, computing a ball $\bigcirc(F, F)$ amounts to solving a system of linear equations), while a violation test costs $O(d)$, because the relevant ball has been computed in a previous basis computation. This means, the actual time required to perform all violation tests asymptotically dominates the time for the basis computations. We can summarize our findings as follows.

Theorem 2.18 *Let $S \subseteq \mathbb{E}^d$ be a set of n points. Algorithm* `LeaveItThenTakeIt`$(\emptyset, S)$ *computes $\bigcirc(S)$ in expected time*

$$O(d(d+1)!\, n).$$

Small cases. If $|S| = n \approx d$, the bound of Theorem 2.18 is not too impressive, and we want to improve on it. Let us concentrate on the case $n \leq d+1$. Exercise 2.6 shows that in this case, the bound of Corollary 2.14 on the expected number of violation tests is an overestimate, and that

$$t_{d+1}(n) = 2^n - 1, \quad n \leq d+1 \tag{2.11}$$

holds. The exercise also shows that the randomization in `LeaveItThenTakeIt`becomes irrelevant in this case. In the next section, we develop a (randomized) algorithm which performs better if S is an *affinely independent* set (which in particular means $|S| \leq d+1$).

2.3 More Basics

Before we can describe the algorithm, we need some further simple facts that will guarantee its correctness. Recall that we are still working under the general position Assumption 2.6. In this case, we have

Observation 2.19 *Let $R \subseteq S \subseteq \mathbb{E}^d$ such that $\bigcirc(R, S)$ exists, $p \in S \setminus R$. Then*

$$\bigcirc(R,S) = \bigcirc(R, S \setminus \{p\}) \quad \Leftrightarrow \quad p \in \bigcirc(R, S \setminus \{p\}),$$
$$\bigcirc(R,S) = \bigcirc(R \cup \{p\}, S) \quad \Leftrightarrow \quad p \notin \bigcirc(R, S \setminus \{p\}).$$

In particular, $\bigcirc(R, S)$ equals either $\bigcirc(R, S \setminus \{p\})$ or $\bigcirc(R \cup \{p\}, S)$*, but not both.*

The first equivalence is obvious, the second one follows from Lemma 2.5, plus the observation that

$$\bigcirc(R, S \setminus \{p\}) \neq \bigcirc(R \cup \{p\}, S).$$

This holds, because otherwise, p would be on the boundary of $\bigcirc(F, F)$, with F being the basis of $(R, S \setminus \{p\})$, just what we excluded with our general position assumption.

In addition to the well-known *violation tests* ('$p \notin \bigcirc(F, F)$'), the algorithm will perform *looseness tests*. Let us introduce both notions formally.

Definition 2.20 *Let $F \subseteq \mathbb{E}^d$ be affinely independent.*

(i) $p \notin F$ violates F *if and only if* $\bigcirc(F, F) \neq \bigcirc(F, F \cup \{p\})$.

(ii) $p \in F$ *is* loose *in* F *if and only if* $\bigcirc(F, F) \neq \bigcirc(F \setminus \{p\}, F)$.

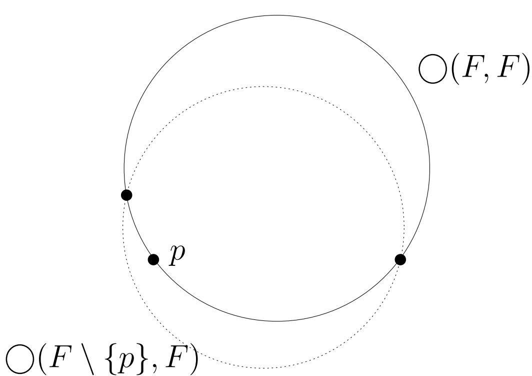


Figure 2.6: p is loose in F; the other two points are 'tight'

Figure 2.6 illustrates the situation. Testing looseness is easy, because Observation 2.19 yields the following

Corollary 2.21 *Let $F \subseteq \mathbb{E}^d$ be affinely independent. $p \in F$ is loose in F if and only if p does not violate $F \setminus \{p\}$.*

Now we are in a position to state the main lemma underlying the algorithm of this section. It says that F is a basis of (R, S) if and only if no point in $S \setminus F$ violates F and no point in $F \setminus R$ is loose in F.

Lemma 2.22 *Let $R \subseteq S \subseteq \mathbb{E}^d$ be a pair satisfying Precondition 2.7, and consider an affinely independent set F with $R \subseteq F \subseteq S$. Then the following two statements are equivalent:*

(i) F is a basis of (R, S), meaning that

$$\bigcirc(F, F) = \bigcirc(R, S) = \bigcirc(R, F).$$

(ii) For all points $p \in S \setminus F$,

$$\bigcirc(F, F) = \bigcirc(F, F \cup \{p\}) \quad \textit{(no point in } S \setminus F \textit{ violates } F\textit{)} \tag{2.12}$$

and for all points $p \in F \setminus R$,

$$\bigcirc(F, F) = \bigcirc(F \setminus \{p\}, F) \quad \textit{(no point in } F \setminus R \textit{ is loose in } F\textit{).} \tag{2.13}$$

Proof. Let F be a basis of (R, S). Using Fact 2.4, we get

$$\bigcirc(F, F) \leq \bigcirc(F, F \cup \{p\}) \leq \bigcirc(F, S) = \bigcirc(F, F), \quad p \in S \setminus F,$$

where the latter equality holds because $\bigcirc(F, F)$ covers S. Similarly,

$$\bigcirc(F, F) \geq \bigcirc(F \setminus \{p\}, F) \geq \bigcirc(R, F) = \bigcirc(F, F), \quad p \in F \setminus R,$$

because F is a basis. It follows that equality holds in all cases, and (ii) is established. In the other direction, (2.12) implies that $\bigcirc(F, F)$ covers S, so if we can show that (2.13) implies $\bigcirc(F, F) = \bigcirc(R, F)$, we are done, because $\bigcirc(R, F)$ covers S, so $\bigcirc(R, F) = \bigcirc(R, S)$ follows.

Assume that $\bigcirc(F, F) \neq \bigcirc(R, F)$ which actually means $\bigcirc(R, F) < \bigcirc(F, F)$. Let E be inclusion-maximal such that $R \subseteq E \subseteq F$ and

$$\bigcirc(E, F) < \bigcirc(F, F). \tag{2.14}$$

By our assumption, E exists; choose a point $p \in F \setminus E$. With Observation 2.19 applied to the pair $(F \setminus \{p\}, F)$, (2.13) yields

$$\bigcirc(F \setminus \{p\}, F \setminus \{p\}) < \bigcirc(F, F). \tag{2.15}$$

Moreover, $\bigcirc(E, F)$ covers p, while $\bigcirc(F \setminus \{p\}, F \setminus \{p\})$ does not (use (2.13) and Corollary 2.21). Both balls go through E and cover $F \setminus \{p\}$.

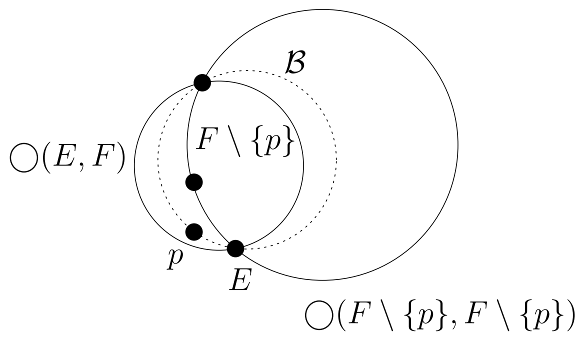


Figure 2.7: Deriving a ball $\mathcal{B}$ which contradicts the choice of E

The situation is therefore as in Figure 2.7, and with Exercise 2.1 applied for the fourth time, we find a ball $\mathcal{B}$ (dotted) which goes through $E \cup \{p\}$, covers F, and is smaller than $\bigcirc(F, F)$ by (2.14) and (2.15). It follows that

$$\bigcirc(E \cup \{p\}, F) < \bigcirc(F, F),$$

a contradiction to our choice of E. $\square$

2.4 Algorithm `TakeItOrLeaveIt`

This section describes the algorithm for computing $\bigcirc(R, S)$ if S is affinely independent. In this case, $\bigcirc(R, S)$ always exists by existence of $\bigcirc(S, S)$ (Exercise 2.3 and Lemma 2.2). Beyond the general position Assumption 2.6, no further preconditions are needed. Just like algorithm `LeaveItThenTakeIt`, `TakeItOrLeaveIt` computes the unique basis F of (R, S), see Postcondition 2.10. Knowing that for any $p \in S \setminus R$, $\bigcirc(R, S)$ equals either $\bigcirc(R, S \setminus \{p\})$ or $\bigcirc(R \cup \{p\}, S)$ (Observation 2.19), the algorithm flips a coin to decide which of the two balls is (recursively) computed first. If later, this choice turns out to be wrong, the algorithm computes the other (correct) ball in a second recursive call.

Exercise 2.7 asks you to prove that the following algorithm indeed returns the basis of (R, S). With the material from the previous section, this can be done without any further geometric reasoning about balls: just as we got really familiar with it, we don't need Exercise 2.1 anymore!

Algorithm 2.23

```
TakeItOrLeaveIt(R, S):
  IF R = S THEN
     F := R
  ELSE
     choose some p ∈ S \ R
     choose a bit β ∈ {0, 1} uniformly at random
     IF β = 0 THEN
        F := TakeItOrLeaveIt(R, S \ {p})
        IF p violates F THEN
           F := TakeItOrLeaveIt(R ∪ {p}, S)
        END
     ELSE
        F := TakeItOrLeaveIt(R ∪ {p}, S)
        IF p is loose in F THEN
           F := TakeItOrLeaveIt(R, S \ {p})
        END
     END
  END
  RETURN F
```

2.4.1 Runtime

As before, we count the expected number of violation and looseness tests as well as the expected number of basis computations ('$F := R$'), which will give us a reasonable estimate of the algorithm's performance.

Violation and looseness tests. Let $t(n)$ denote the expected number of violation and looseness tests in a call to `TakeItOrLeaveIt`(R, S), where $S \subseteq \mathbb{E}^d$ is affinely independent and $|S \setminus R| = n$.[8] We get

$$\begin{aligned} t(0) &= 0, \\ t(n) &= t(n-1) + 1 + \frac{1}{2}t(n-1), \quad n > 0, \end{aligned}$$

because with probability exactly $1/2$, the ball computed in the first recursive call was the wrong one, in which case we need a second call of expected cost $t(n-1)$.

It follows that

$$t(n) = 1 + \frac{3}{2}t(n-1) = 1 + \frac{3}{2} + \left(\frac{3}{2}\right)^2 + \cdots + \left(\frac{3}{2}\right)^{n-1} + \left(\frac{3}{2}\right)^n t(0) = 2\left(\left(\frac{3}{2}\right)^n - 1\right),$$

where we have used $\sum_{i=0}^{m} z^i = (z^{m+1} - 1)/(z - 1)$. This is still exponential, but a substantial improvement over the bound (2.11).

Basis computations. Let $b(n)$ denote the expected number of basis computations in a call to `TakeItOrLeaveIt`(R, S), where $S \subseteq \mathbb{E}^d$ is affinely independent and $|S \setminus R| = n$. We obtain

$$\begin{aligned} b(0) &= 1, \\ b(n) &= b(n-1) + \frac{1}{2}b(n-1), \quad n > 0, \end{aligned}$$

which solves to

$$b(n) = \left(\frac{3}{2}\right)^n.$$

As before, we can argue that any basis computation as well as any violation and looseness test can be performed in time $O(d^3)$, and we get

Theorem 2.24 *Let $S \subseteq \mathbb{E}^d$ be a set of $n \leq d + 1$ affinely independent points. Algorithm* `TakeItOrLeaveIt`$(\emptyset, S)$ *computes $\bigcirc(S)$ in expected time*

$$O\left(d^3 1.5^n\right).$$

The practical implication of this result is that algorithm `TakeItOrLeaveIt` can be used to compute the smallest enclosing ball of an affinely independent point set $S \subseteq \mathbb{E}^{50}$ pretty quickly. In contrast, algorithm `LeaveItThenTakeIt` is already very slow for $d > 30$.

It is a natural question whether we can further reduce the number of *primitive operations* (violation and looseness tests, basis computations). This is indeed possible, and in the next chapter we will do this in the more general framework of *unique sink orientations*.

[8] The fact that this number only depends on n and not on the specific pair (S, R), can inductively be established along the lines of the subsequent analysis.

Bibliographical Remarks

The algorithm `LeaveItThenTakeIt` is due to Welzl [16]. `TakeItOrLeaveIt` is due to Gärtner and Welzl [4]. The proofs via purely geometric arguments using Exercise 2.1 are new. The nondegeneracy assumption 2.6 can be removed. For algorithm `LeaveItThenTakeIt`, this is easy and is already done in Welzl's original paper [16]. Fischer and Gärtner do this for `TakeItOrLeaveIt` [2].

Exercises

Exercise 2.1 *Let $\mathcal{B}$ be a ball with center c and positive squared radius ρ. Then*

$$\mathcal{B} = \{x \in \mathbb{E}^d \mid f_{\mathcal{B}}(x) \leq 1\}, \quad f_{\mathcal{B}}(x) := \|x - c\|^2/\rho.$$

Let $\mathcal{B}_0, \mathcal{B}_1$ be two balls with centers c_0, c_1 and positive squared radii ρ_0, ρ_1. Moreover, $\mathcal{B}_0 \cap \mathcal{B}_1 \neq \emptyset$.

(i) Prove that for any $\lambda \in (0, 1)$, there is a ball $\mathcal{B}_\lambda$ with

$$f_{\mathcal{B}_\lambda}(x) = (1 - \lambda) f_{\mathcal{B}_0} + \lambda f_{\mathcal{B}_1}.$$

(ii) Show that if $\mathcal{B}_0$ and $\mathcal{B}_1$ both go through R and cover S, then the same is true for $\mathcal{B}_\lambda$.

(iii) Let ρ_λ be the squared radius of $\mathcal{B}_\lambda$. Prove that $\rho_\lambda < \max(\rho_0, \rho_1)$ for $\lambda \in (0, 1)$.

Exercise 2.2 *Prove Fact 2.8.*

Exercise 2.3 *Let $F \subseteq \mathbb{E}^d$ be affinely independent, $F \neq \emptyset$.*

(i) Prove that the center of $\bigcirc(F, F)$ is in the affine hull *of F. The affine hull of a set Q is the set of all affine combinations of Q,*

$$\text{aff}(Q) := \{\sum_{q \in Q} \lambda_q q \mid \forall q : \lambda_q \in \mathbb{R}, \sum_{q \in Q} \lambda_q = 1\}.$$

The affine hull is an affine subspace of the linear hull *(set of all linear combinations),*

$$\text{lin}(Q) := \{\sum_{q \in Q} \lambda_q q \mid \forall q : \lambda_q \in \mathbb{R}\}.$$

Argue directly for $|F| = 1$. For $|F| > 1$, proceed in the following steps.

(a) Prove that for any point $p \in F$ and any point $s \in E^d$,

$$s \in \text{aff}(F) \quad \Leftrightarrow \quad s - p \in \text{lin}(F - p),$$

where

$$F - p := \{q - p \mid q \in F \setminus \{p\}\}.$$

(b) For $p \in F$, let M be the $d \times (|F|-1)$-matrix whose columns are the points of $F-p$ in any order. Show that M^TM is invertible.

(c) For $s \in E^d$, consider the point

$$s^* := p + M(M^TM)^{-1}M^T(s-p)$$

and prove (using (a)) that $s^* \in \mathrm{aff}(F)$. (s^* is the projection of s onto $\mathrm{aff}(F)$.)

(d) Prove that $\|(s^*-p)\|^2 + \|s-s^*\|^2 = \|s-p\|^2$.

(e) Assume that the center of $\bigcirc(F,F)$ lies outside of $\mathrm{aff}(F)$, and show (using (d)) that this leads to a contradiction.

(ii) Prove that there is a unique *ball $\mathcal{B}$ through F whose center is in $\mathrm{aff}(F)$. It follows that $\mathcal{B} = \bigcirc(F,F)$. Use the settings and results of part 1 to determine the center c of $\mathcal{B}$ in the form*

$$c = p + M\lambda,$$

where $\lambda = (\lambda_1, \ldots, \lambda_{|F|-1})^T$ is the unique solution to a suitable system of linear equations.

Exercise 2.4 *Prove that in a call of* `LeaveItThenTakeIt`(R,S) *with $R \neq S$, the number of basis computations is at most the number of violation tests, plus one.*

Exercise 2.5 *Let $S = \{p_1, \ldots, p_n\} \subseteq \mathbb{E}^d$ and consider the deterministic variant of Algorithm* `LeaveItThenTakeIt`(S,R) *in which the line*

choose $p \in S \setminus R$ uniformly at random

is replaced by the line

choose p as the point with largest index in $S \setminus R$.

Prove that for any fixed dimension $d \geq 1$, there exists a point set S for which the number of violation tests in the deterministic variant is $\Omega(n^{d+1})$ (the constant hidden in the Ω may depend on d). (Hint: find an example for $d=2$ first, then try to generalize it.)

Exercise 2.6 *Prove that for $n \leq k$,*

$$t_k(n) \leq 2^n - 1.$$

Show that this bound is best possible by constructing a set S of $n = d+1$ points in $\mathbb{E}^d$ for which the expected number of violation tests in `LeaveItThenTakeIt`(R,S) *is exactly $2^{|S \setminus R|} - 1$ for any $R \subseteq S$.*

Exercise 2.7 *Prove that Algorithm* `TakeItOrLeaveIt`(R,S) *returns the unique basis F of (R,S).*

Exercise 2.8 *We have shown that algorithm* `LeaveItThenTakeIt` *computes the smallest enclosing ball of an n-point set $S \subseteq \mathbb{E}^d$ with an expected number of at most $c_{d+1}n$ violation tests, where c_{d+1} is some constant only depending on d. Prove that for any constant $K \geq 1$, there is an algorithm that requires at most*

$$2Kc_{d+1}n$$

violation tests, with probability at least

$$1 - \frac{1}{2^K}.$$

This means, there is an algorithm for smallest enclosing balls that achieves linear runtime not only in expectation, but with arbitrarily high probability.

Exercise 2.9 *The ideas of algorithm* `LeaveItThenTakeIt` *can be used to solve a linear program in d variables and n inequality constraints in time $O(n)$, if d is a constant. Here, we consider linear programs in two variables x_1, x_2, of the form*

$$\begin{array}{ll} \textit{minimize} & c_1x_1 + c_2x_2 \\ \textit{subject to} & a_{i1}x_1 + a_{i2}x_2 \leq b_i, \quad i = 1, \ldots, n, \\ & x_1, x_2 \geq 0, \end{array}$$

where a_{i1}, a_{i2}, b_i are arbitrary real numbers for $i = 1, \ldots, n$, and c_1, c_2 are nonnegative real numbers. Develop a randomized algorithm with expected runtime $O(n)$ for finding a pair $(\tilde{x}_1, \tilde{x}_2)$ with smallest value $c_1\tilde{x}_1 + c_2\tilde{x}_2$ among the pairs that satisfy all $n + 2$ inequality constraints. The inequalities $x_1, x_2 \geq 0$ together with $c_1, c_2 \geq 0$ ensure that this smallest value is bounded from below by 0.

If the linear program is infeasible*, meaning that no pair (x_1, x_2) satisfies all inequalities, the algorithm should output this fact. Moreover, make sure that your algorithm correctly deals with the case where the optimal pair $(\tilde{x}_1, \tilde{x}_2)$ (of the linear program, or of some subprogram) is not unique.*

Chapter 3

Unique Sink Orientations of Cubes

In the previous chapter, we have taken a geometric approach in order to develop algorithms for computing the smallest enclosing ball of a finite point set $S \subseteq \mathbb{E}^d$. In this chapter, we adopt a purely combinatorial view and show that any instance of the problem defined by an *affinely independent* set S of $n \leq d+1$ points has the structure of a *unique sink orientation* (USO) of the n-cube graph. From the global sink of that orientation, the smallest enclosing ball $\bigcirc(S)$ of S can be read off.

It follows that any (in particular, any randomized) algorithm for finding the sink of a general USO can immediately be applied to the smallest enclosing ball problem.[1]

This chapter deals with randomized sink-finding algorithms for general USO, quite in the spirit of Randomix's strategy for quickly finding the exit in Determinatus's maze, see Chapter 1. Even though the problem is more general than smallest enclosing balls, we will be able to improve over the results of the previous chapter. The main reason for this is that our combinatorial view reveals some structure of the smallest enclosing ball problem that we didn't see (and therefore couldn't use) before, probably because we were too focused on the geometry. In particular, Algorithm 2.23 from the previous chapter—which appeared to be pretty fancy back then—will in the USO framework turn out to be a very natural randomized algorithm for finding the sink.

3.1 Definition and Examples

The vertices of the n-dimensional cube can be identified with the subsets of some n-element set N (which for most examples in this chapter will simply be the set $N = [n] := \{1, \ldots, n\}$). The *graph* of the n-cube is the graph $C_n = (V, E)$, where

$$
\begin{aligned}
V &:= 2^N = \{X \mid X \subseteq N\}, \\
E &:= \{\{X, X \oplus \{i\}\} \mid X \in 2^N, i \in N\}.
\end{aligned}
$$

[1] There is a number of other interesting problems which profit from USO techniques, but we don't have time to discuss them here.

Here, $\oplus$ is the *symmetric difference* of two sets.[2] This means that two subsets of N are connected by an edge if and only if they differ in exactly one element, and this element will be the *label* of the edge. Figure 3.1 shows the graph of the 3-cube with edge labels.

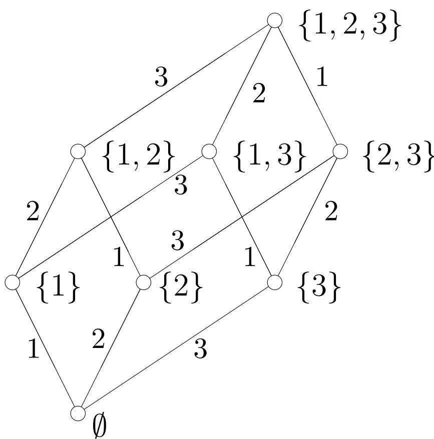


Figure 3.1: The graph C_3 with edge labels

The faces of the cube can be identified with *intervals* of vertices

$$[A, B] := \{X \mid A \subseteq X \subseteq B\}.$$

The faces of the form $[X, X]$ are the vertices themselves, $[\emptyset, N]$ is the whole cube, and $|B \setminus A|$ is the dimension of the face $[A, B]$. Figure 3.2 depicts two faces of the 3-cube in bold.

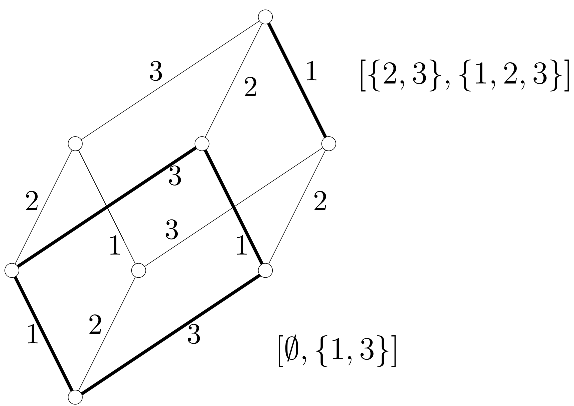


Figure 3.2: Faces of C_3

An *orientation* of C_n is a directed graph $\mathcal{O} = (2^N, D)$, such that D contains exactly one of the ordered pairs $(X, X \oplus \{i\})$ and $(X \oplus \{i\}, X)$, for all $X \subseteq N$ and $i \in N$.

[2] $A \oplus B := (A \cup B) \setminus (A \cap B)$ is the set of elements which are in *exactly* one of the two sets.

Definition 3.1 *An orientation $\mathcal{O}$ of C_n is called a* unique sink orientation *(USO), if for all faces $[A, B]$, the subgraph of $\mathcal{O}$ induced by $[A, B]$ contains exactly one sink. We also say that $[A, B]$ contains a unique sink and denote this sink by $\odot_{\mathcal{O}}(A, B)$.*

In contrast to the setup of Exercise 1.1, we do not only require the whole graph to have a unique sink, but also certain subgraphs, in this case all subgraphs of the cube graph that are induced by cube faces. This makes the orientations more specific, and we will see that the lower bound on the number of vertex evaluations established in Exercise 1.1 does not hold here (Exercise 3.4). It's time for some examples.

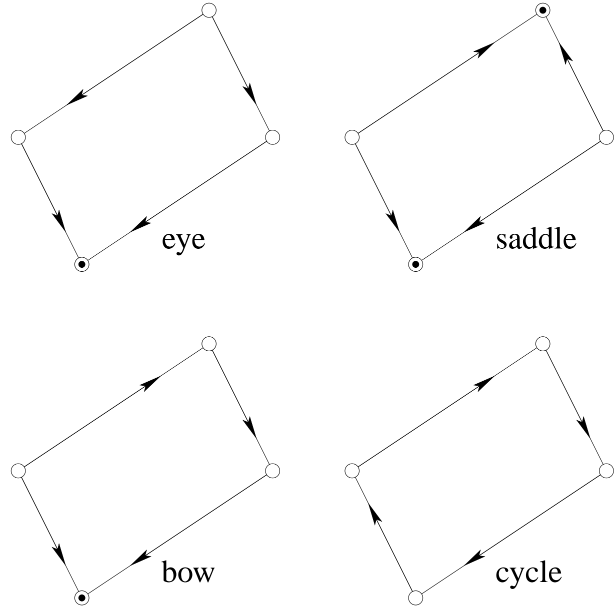


Figure 3.3: Orientations of C_2

The 2-cube has (up to isomorphisms) four different orientations, see Figure 3.3. The *eye* and the *bow* are USO, but the other two aren't: the *saddle* has two sources and two sinks, while the *cycle* has no sink at all. In all cases, we only need to check the single 2-dimensional face, because vertices and edges always have unique sinks.

While cycles cannot occur in a USO of the C_2, there is one USO of the C_3 which has a cycle, see Figure 3.4: there is a unique sink in the whole cube, and every 2-face is a bow.

3.2 Smallest Enclosing Balls Revisited

Here is our motivating example for the concept of USO. Recall that for an affinely independent set $S \subseteq \mathbb{E}^d, |S| = n \leq d + 1$ and a subset $F \subseteq S$, we have defined that a point $p \in S \setminus F$ *violates* F iff $\bigcirc(F, F) \neq \bigcirc(F, F \cup \{p\})$, and that a point $p \in F$ is *loose* in F if $\bigcirc(F, F) \neq \bigcirc(F \setminus \{p\}, F)$ (Definition 2.20).

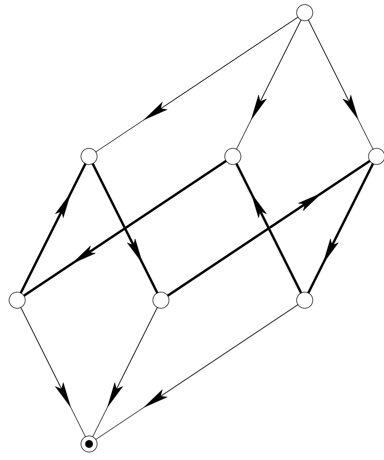


Figure 3.4: Cyclic USO of C_3

Theorem 3.2 *For $S \subseteq \mathbb{E}^d$ affinely independent and in general position according to Assumption 2.6, $|S| = n \leq d+1$, define*

$$\begin{aligned} D_{\rightarrow} &:= \{(F, F \cup \{p\}) \mid F \subseteq S,\ p \in S \setminus F,\ p \text{ violates } F\}, \\ D_{\leftarrow} &:= \{(F, F \setminus \{p\}) \mid F \subseteq S,\ p \in F,\ p \text{ is loose in } F\}. \end{aligned}$$

(i) The graph $O = (2^S, D_{\rightarrow} \cup D_{\leftarrow})$ is a USO of the n-cube.

(ii) For any $A \subseteq F \subseteq B$, the following statements are equivalent.

(a) $F = \odot_{\mathcal{O}}(A, B)$

(b) F is the unique basis *of (A, B) w.r.t. smallest enclosing balls, meaning that $\bigcirc(A, B) = \bigcirc(F, F) = \bigcirc(A, F)$.*

Proof. By Lemma 2.22, F is the basis of (A, B) if and only if no point in $B \setminus F$ violates F and no point in $F \setminus A$ is loose in F, which by definition of $\mathcal{O}$ equivalently means that F is a sink in the face $[A, B]$. By general position, (A, B) has a unique basis, so $[A, B]$ has a unique sink, and both sets coincide. This proves (i) and (ii). □

Figure 3.5 shows the two types of USO on the 2-cube that can arise from the smallest enclosing ball problem over a set of three affinely independent points in the plane. The orientations differ in exactly one edge orientation: in the upper case, r violates $\{p, q\}$, while in the lower case, it doesn't. The smallest enclosing ball of the three points (whose boundary set corresponds to the sink of the USO) is drawn solid.

3.3 The Algorithmic Model

In the previous chapter, we have developed Algorithm 2.23 (`TakeItOrLeaveIt`) which computes the smallest enclosing ball of a set S of affinely independent points with an expected

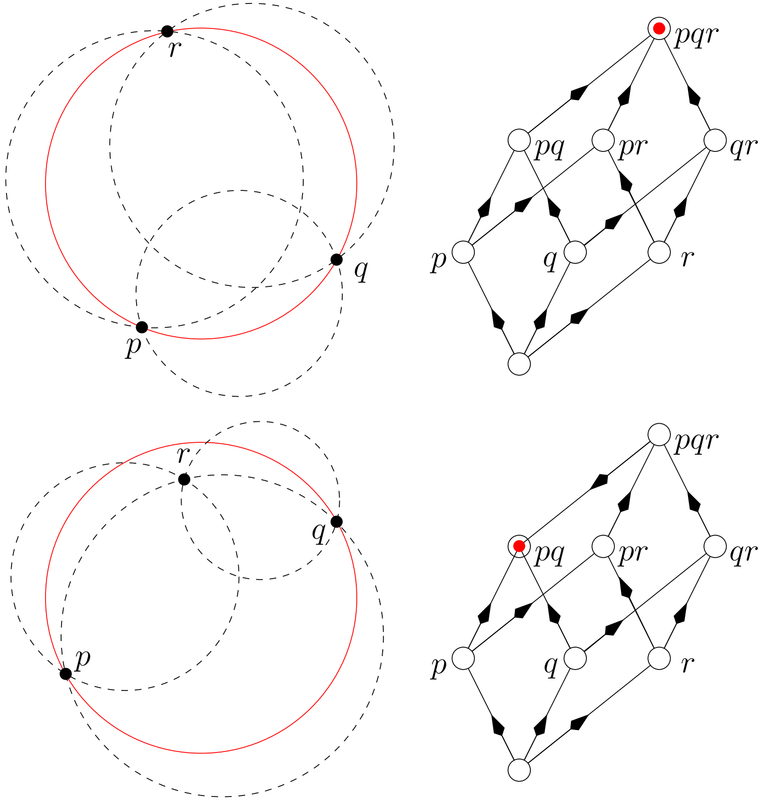


Figure 3.5: USO coming from smallest enclosing balls

number of

$$\left(\frac{3}{2}\right)^{|S|}$$

basis computations (and about twice as much violation and looseness tests, involving a previously computed basis and some other point).

One goal of this chapter is to show that this algorithm can actually be formulated as a (very simple) sink-finding procedure for general USO, where a basis computation translates to a so-called *vertex evaluation*. In itself, this generalization does not give us new results for smallest enclosing balls; however, the point is that the sink in a general USO can be found with even less vertex evaluations than `TakeItOrLeaveIt` needs basis computations, and this *does* give us an improved algorithm also for the special case of smallest enclosing balls.

The model is as follows: the USO is given to us implicitly, and we can obtain information about it through vertex evaluations: evaluating vertex X reveals the orientations of the incident edges. We are looking for a deterministic (randomized) algorithm that minimizes the (expected) number of vertex evaluations necessary to *evaluate* the sink of any USO of the n-cube. More precisely, we define

$$t(n) := \min_{A} \max_{\mathcal{O}} t(A, \mathcal{O}),$$

where the minimum is taken over all deterministic algorithms, the maximum is taken over all

USOs of the n-cube, and $t(A, \mathcal{O})$ is the number of vertex evaluations that algorithm A needs in order to evaluate the sink of $\mathcal{O}$. Similarly,

$$\tilde{t}(n) := \min_A \max_{\mathcal{O}} \tilde{t}(A, \mathcal{O}),$$

where the minimum is taken over all *randomized* algorithms, and $\tilde{t}(A, \mathcal{O})$ is the expected number of vertex evaluations that algorithm A needs in order to evaluate the sink of $\mathcal{O}$.

In still other words, the goal is to find the (randomized) algorithm with the best (expected) worst-case performance.

The following bounds are easy to see.

Observation 3.3

$$\begin{aligned} t(1) &= 2, \quad \tilde{t}(1) = 3/2, \\ t(2) &= 3, \end{aligned} \tag{3.1}$$

and

$$\tilde{t}(n) \le t(n) \le 2^{n-1} + 1. \tag{3.2}$$

Proof. Any deterministic algorithm for the 1-cube can be forced to evaluate two vertices: because the adversary knows which vertex is evaluated first, he will make sure that this one is not the sink. $\tilde{t}(1) \le 3/2$ is clear, because one possible randomized algorithm chooses the vertex to be evaluated first among the two vertices with equal probability. Because the adversary is oblivious, he must commit to an orientation *before* the algorithm makes its random choice (just like Determinatus has to set up a maze before Randomix enters it, see Chapter 1). Then the expected number of vertices evaluated is $1/2(1+2) = 3/2$. It is also clear that one cannot do better: any randomized algorithm has to choose the first vertex to be evaluated by assigning some probability p to one of the vertices and $1-p$ to the other. If $p \ne 1/2$, the adversary (who knows p, just like Determinatus knows Randomix's dice-rolling strategy in Chapter 1) will place the sink on the vertex less likely to be chosen, leading to an expected number of strictly more than $3/2$ vertex evaluations.

In order to see $t(2) \le 3$, we apply (3.2) and for $t(2) \ge 3$, the adversary places the source on the first vertex that is evaluated. Then an easy case analysis shows that for any choice of the second vertex, there is a USO in which the first one is the source, but the second one is *not* the sink. Thus, the adversary can always enforce a third evaluation.

The general bound of $t(n) \le 2^{n-1} + 1$ follows like in Exercise 1.1(ii), and $\tilde{t}(n) \le t(n)$ is obvious, because any deterministic algorithm can be considered as a randomized algorithm that doesn't use its random resources. □

In case of smallest enclosing balls in $\mathbb{E}^d$, an evaluation of vertex X can be implemented in time $O(d^3)$; we need $O(d^3)$ time to compute $\bigcirc(X, X)$, and once we have it, the orientations of the at most d incident edges can be computed in time $O(d)$ per edge, by doing one violation or looseness test.[3] This means, any algorithm for finding the sink in a USO with (an expected number of) t steps can be used to compute the smallest enclosing ball of an affinely independent point set in (expected) time $O(t \cdot d^3)$.

[3] We didn't argue how to perform a looseness test in $O(d)$ time; it can be done, but if you have doubts, an $O(d^3)$ bound will do as well, leading to an overall bound of $O(d^4)$ per vertex evaluation.

3.4 Basic Theory

Any USO $\mathcal{O} = (2^N, D)$ of C_n defines an *outmap* $s : 2^N \to 2^N$, where $s(X)$ is the set of labels of outgoing edges of X,

$$s(X) := \{i \mid (X, X \oplus \{i\}) \in D\}.$$

Obviously,

$$s(\odot_{\mathcal{O}}(\emptyset, N)) = \emptyset,$$

and, more generally,

$$s(\odot_{\mathcal{O}}(A, B)) \cap (B \setminus A) = \emptyset. \tag{3.3}$$

Equation (3.3) holds, because any edge incident to the sink with label $a \in B \setminus A$ is an edge *within* the face $[A, B]$ and is therefore incoming.

Figure 3.6 shows the outmap values associated to a bow of C_2. We denote the outmap defined by a particular orientation $\mathcal{O}$ with $s_{\mathcal{O}}$.

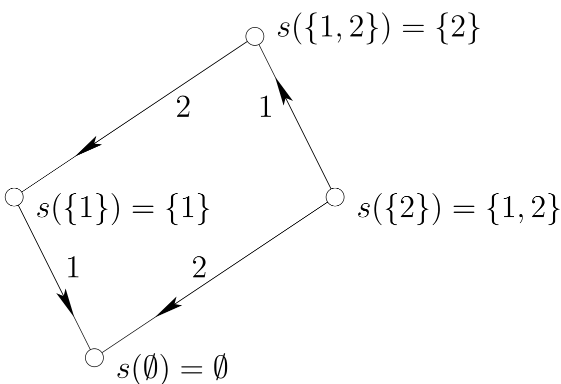


Figure 3.6: Outmap of a USO

It is no coincidence that in Figure 3.6, the outmap is a bijection. This always holds.

Lemma 3.4 *Let $s : 2^N \to 2^N$ be the outmap of a USO $\mathcal{O}$. Then s is a bijection.*

Proof. It suffices to show that s is injective. For this, fix two vertices X and Y such that $s(X) = s(Y) = A \subseteq N$. By iterating Exercise 3.3(i), we see that reorienting all edges of $\mathcal{O}$ with labels in A leads to a USO $\mathcal{O}'$, where

$$s_{\mathcal{O}'}(X) = s_{\mathcal{O}'}(Y) = \emptyset,$$

meaning that both X and Y are global sinks in $\mathcal{O}'$. It follows that $X = Y$, so s is injective. $\square$

Here is a characterization that tells us whether a function $s : 2^N \to 2^N$ is the outmap of a USO. Its (easy) proof is left as Exercise 3.3(ii)

Lemma 3.5 *A function $s : 2^N \to 2^N$ is the outmap of a USO if and only if*

$$(X \oplus Y) \cap (s(X) \oplus s(Y)) \neq \emptyset, \quad \forall X \neq Y \subseteq 2^N.$$

This simple condition hits two birds with one stone: apart from encoding the actual USO property, it guarantees *consistency* of the orientation, meaning that *exactly one* of the two neighboring vertices X and $X \oplus \{i\}$ has an outgoing edge with label i.

Inherited Orientations. In the introduction to this chapter, I have promised some nontrivial structure that we haven't seen for smallest enclosing balls. Here it comes.

Theorem 3.6 *Let $\mathcal{O} = (2^N, D)$ be a USO of the $|N|$-cube with outmap $s = s_{\mathcal{O}}$. Fix $A \subseteq N$ and consider the function $s : 2^{N \setminus A} \rightarrow 2^{N \setminus A}$ defined by*

$$s'(X) := s\left(\odot_{\mathcal{O}}(X, X \cup A)\right).$$

Then s' is the outmap of a USO $\mathcal{O}'$ of the $|N \setminus A|$-cube.

Before we get to the proof, let us discuss what this means. The set A defines $2^{|N \setminus A|}$ subcubes *spanned* by A, of the form

$$[X, X \cup A], \quad X \subseteq 2^{N \setminus A}.$$

In Figure 3.7, $N = \{1, 2, 3, 4\}$. The set $A = \{2, 3\}$ spans four subcubes (the grey blobs).

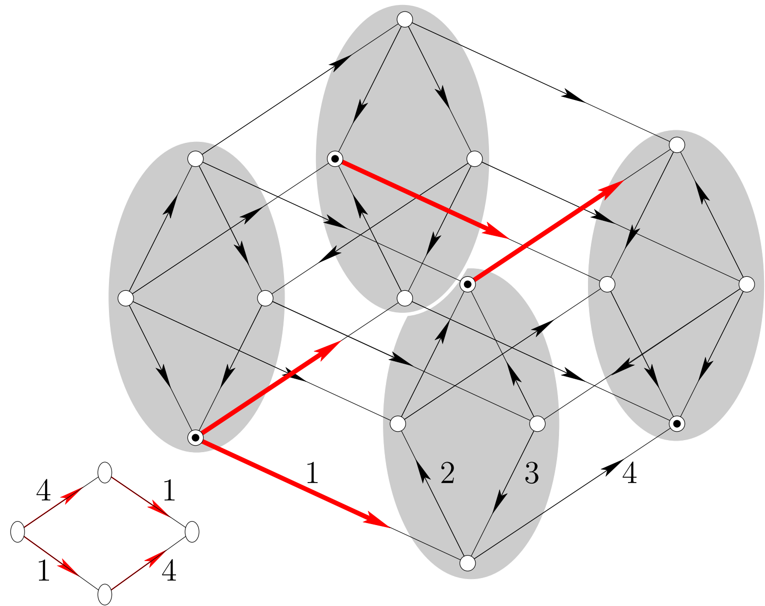


Figure 3.7: Inherited USO

The value $s'(X)$ records the labels of the outgoing edges at the *sink* of the subcube $[X, X \cup A]$ (bold edges in Figure 3.7). By (3.3), $s'(X) \subseteq N \setminus A$.

s' defines an orientation $\mathcal{O}'$ on a cube spanned by $N \setminus A$, see lower left part of Figure 3.7. This cube arises by 'reducing' all subcubes (the grey blobs) to their sinks. The statement of the Theorem is that $\mathcal{O}'$ is in fact a USO again. Moreover, it is clear that the subcube corresponding to the sink of $\mathcal{O}'$ (the rightmost grey blob in Figure 3.7) also contains the global sink.

Proof. We show that s' satisfies the condition of Lemma 3.5. For this, choose $X \neq Y \subseteq 2^{N \setminus A}$ and set

$$\begin{aligned} W &:= \odot_{\mathcal{O}}(X, X \cup A), \\ Z &:= \odot_{\mathcal{O}}(Y, Y \cup A). \end{aligned}$$

Note that $W \neq Z$, because they lie in disjoint subcubes. We claim that

$$(X \oplus Y) \cap (s'(X) \oplus s'(Y)) = (W \oplus Z) \cap (s(W) \oplus s(Z)), \tag{3.4}$$

which proves the Theorem, because the right-hand side is a nonempty set by Lemma 3.5. To see (3.4), first note that

$$s'(X) \oplus s'(Y) = s(W) \oplus s(Z) =: B \subseteq N \setminus A$$

by definition. Secondly, for $a \in N \setminus A$, we have $a \in X \Leftrightarrow a \in W$ (and $a \in Y \Leftrightarrow a \in Z$), hence

$$(X \oplus Y) \cap B = (W \oplus Z) \cap B,$$

which is (3.4). □

The Product Algorithm. Theorem 3.6 suggests the following approach for evaluating the sink in a USO $\mathcal{O}$ of the n-cube, given that you have, for some $k \in \{0, \ldots, n\}$, two (deterministic or randomized) algorithms $\texttt{FindSink}_k$ and $\texttt{FindSink}_{n-k}$ for evaluating the sink in a k-cube and an $(n-k)$-cube, respectively: choose $|A| = k$ and use $\texttt{FindSink}_{n-k}$ to evaluate the sink of $\mathcal{O}'$ as defined by the function s' above. Whenever the algorithm needs to evaluate $s'(X)$, call $\texttt{FindSink}_k$ for the original outmap s on $[X, X \cup A]$. If X is the sink w.r.t. s', this call will eventually evaluate the desired sink w.r.t. s. If $\texttt{FindSink}_k$ and $\texttt{FindSink}_{n-k}$ are best possible deterministic algorithms, this *product algorithm* will call $\texttt{FindSink}_k$ at most $t(n-k)$ times, and each such call evaluates at most $t(k)$ vertices. It follows that the product algorithm requires at most $t(k)t(n-k)$ vertex evaluations. At this point, it becomes clear why our algorithmic model counts the number of steps we need to *evaluate* the sink rather than the number of steps until we *know* it. Namely, just knowing the sink of $\mathcal{O}'$ does not allow us to deduce where the sink of $\mathcal{O}$ is: we only know the k-dimensional subcube containing it. Consequently, some "plus-one-terms" would uglify the analysis of the product algorithm.

If we work with best possible *randomized* algorithms, $\texttt{FindSink}_k$ is called an *expected* number of at most $\tilde{t}(n-k)$ times, each call requiring an *expected* number of at most $\tilde{t}(k)$ vertex evaluations. Because the random choices in both algorithms are independent of each other, the expectations can be multiplied, and the product algorithm needs at most $\tilde{t}(k)\tilde{t}(n-k)$ vertex evaluations in expectation. Because the product algorithm is just one (not necessarily the best) algorithm for the n-cube, we have shown the following result.

Theorem 3.7 *For* $0 \leq k \leq n$,

$$\begin{aligned} t(n) &\leq t(k)t(n-k), \\ \tilde{t}(n) &\leq \tilde{t}(k)\tilde{t}(n-k). \end{aligned}$$

Using $\tilde{t}(1) = 3/2$, this gives us

$$\tilde{t}(n) \leq \tilde{t}(n-1)\frac{3}{2} \leq \left(\frac{3}{2}\right)^n.$$

Moreover, there is a simple product algorithm that achieves this bound: let $\texttt{Guess}_1$ be the simple randomized algorithm for finding the sink of the 1-cube with an expected number of $3/2$ vertex evaluations, see Observation 3.3. Consider the product algorithm $\texttt{Product}_n$ which satisfies

$$\texttt{Product}_1 = \texttt{Guess}_1,$$

and which is, for $n > 1$, recursively defined via $k = n - 1$ and

$$\begin{aligned} \texttt{FindSink}_{n-k} &= \texttt{Guess}_1 \\ \texttt{FindSink}_k &= \texttt{Product}_{n-1}. \end{aligned}$$

Then the expected number of vertex evaluations of $\texttt{Product}_n$ is easily seen to be *exactly*

$$\left(\frac{3}{2}\right)^n. \tag{3.5}$$

Actually, we have rediscovered the USO equivalent of the algorithm TakeItOrLeaveIt for smallest enclosing balls.

Theorem 3.7 also yields

$$\tilde{t}(n) = O\left(\tilde{t}(2)^{n/2}\right). \tag{3.6}$$

Assume we could find a randomized algorithm for 2-cube USO, with an expected number of vertex evaluations smaller than

$$\left(\frac{3}{2}\right)^2 = \frac{9}{4} = \frac{45}{20}$$

in the worst case (this is the bound we get from $\texttt{Product}_2$). Then we could use this algorithm in a recursively defined product algorithm as above, to achieve the bound in (3.6), therefore beating the bound of (3.5). The concrete result would be a randomized algorithm for smallest enclosing balls of affinely independent points that is faster than TakeItOrLeaveIt.

The goal of the next section is to develop such an improved algorithm for the 2-dimensional case, with an expected number of at most

$$\frac{43}{20}$$

vertex evaluations.

3.5 The 2-dimensional Case

The algorithm starts by choosing one of the four vertices uniformly at random. Depending on whether the chosen vertex X is the sink (Case 1), a vertex with one incoming and one outgoing edge (Case 2), or the source (Case 3), the algorithm will proceed differently. We will analyze the performance separately for the *eye* and the *bow* as input, cf. Figure 3.3.

Case 1. The chosen vertex X is the sink. This happens with probability $1/4$, and there's nothing left to do. Table 3.1 records that we need one vertex evaluation, regardless of whether we have an eye or a bow.

	probability	eye	bow
Case 1	$1/4$	1	1
Case 2	$1/2$	2	$5/2$
Case 3	$1/4$		
Strategy 1		2	3
Strategy 2		4	$5/2$

Table 3.1: Number of vertex evaluations in the different cases of the algorithm

Case 2. The chosen vertex X has one incoming and one outgoing edge. This happens with probability $1/2$. We follow the outgoing edge and evaluate its other vertex Y. In case of an eye, Y must be the sink. In case of a bow, Y is the sink with probability $1/2$, and if it is not the sink, we evaluate its other neighbor Z which then must be the sink, see Figure 3.8. For an eye, we always need 2 evaluations in this case, for a bow, the expected number is $(2+3)/2 = 5/2$.

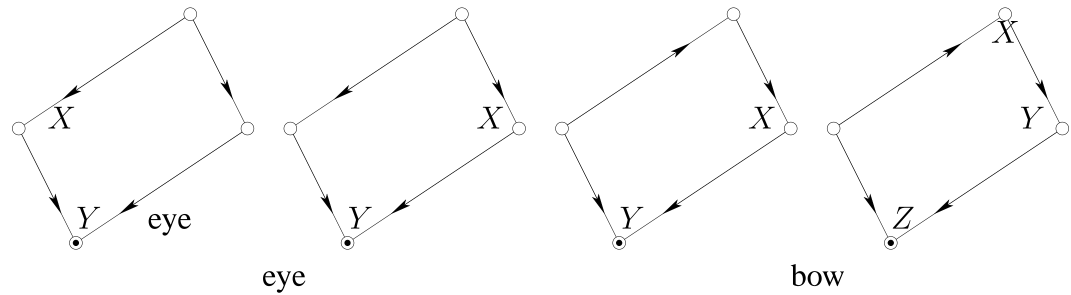


Figure 3.8: Case 2: X is a vertex with one incoming and one outgoing edge

Case 3. The chosen vertex X is the source. This case happens with probability $1/4$. We consider two strategies.

Strategy 1. Evaluate the vertex W antipodal to X. In case of an eye, W is the sink, in case of a bow, W is not the sink, but since we now know all edge orientations, we need just one more evaluation to hit the sink. This strategy needs 2 evaluations for an eye and 3 for a bow.

Strategy 2. Choose one of the neighbors U, U' of X to be evaluated next, uniformly at random. If the chosen neighbor is not the sink, evaluate the other neighbor. In case of an eye, none of U, U' is the sink, so we need a fourth evaluation. In case of a bow, one of U, U' is the sink, and we need $(2+3)/2 = 5/2$ evaluations on average to evaluate it, see Figure 3.9.

Going through Table 3.1, we see that if we are playing Strategy 1 in Case 3, the expected number of vertex evaluations for an eye is

$$\frac{1}{4} \cdot 1 + \frac{1}{2} \cdot 2 + \frac{1}{4} \cdot 2 = \frac{7}{4}.$$

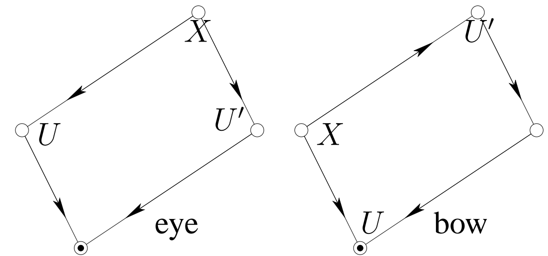


Figure 3.9: Case 3: X is the source

	eye	bow
Strategy 1	$7/4$	$9/4$
Strategy 2	$9/4$	$17/8$

Table 3.2: Summary of algorithm's performance

The other values can be extracted similarly and are summarized in Table 3.2.

Choosing Strategy 1 with probability λ and Strategy 2 with $1-\lambda$, the expected number of vertex evaluations is

$$\frac{7}{4}\lambda + \frac{9}{4}(1-\lambda) \tag{3.7}$$

for the eye and

$$\frac{9}{4}\lambda + \frac{17}{8}(1-\lambda) \tag{3.8}$$

for the bow. While (3.7) increases with λ, (3.8) decreases, which means that the maximum of (3.7) and (3.8) (the worst-case performance of the algorithm) is minimized when the two terms are equal. This happens for $\lambda = 1/5$, in which case both (3.7) and (3.8) evaluate to

$$\frac{43}{20}.$$

Plugging this into the bound (3.6), and using the 2-dimensional algorithm developed above as the basis for a product algorithm that matches this bound, we get an improvement over our previously best 1.5^n bound.

Corollary 3.8 *There is a randomized algorithm that evaluates the sink of any USO of the n-cube with an expected number of at most*

$$O\left(\sqrt{\frac{43}{20}}^{\,n}\right) = O(1.466^n)$$

vertex evaluations.

Bibliographical Remarks

The concept of USOs has first been introduced by Stickney and Watson [13]; the product structure and the product algorithm, as well as the optimal algorithm for $d = 2$ are due to Szabó and Welzl [14]. The reformulation of the smallest enclosing ball problem in terms of USOs also appears in that paper, but it is already implicit in the earlier paper by Gärtner and Welzl [4]. The USO approach can be generalized to smallest enclosing balls of *balls* in any dimension, see Fischer and Gärtner [2].

Exercises

Exercise 3.1 *Recall Randomix's strategy of finding the sink in Determinatus's maze from Chapter 1: at any given vertex, choose one of the outgoing edges at random and go to its other vertex. Repeat until the sink is reached. What is the expected number of vertices visited by this strategy, starting from the source* [4] *of the cyclic USO of the 3-cube, see Figure 3.4? Can you find a USO of the 3-cube where the strategy needs to visit more vertices on average?*

Exercise 3.2 *Prove that an* acyclic *orientation of $C_n, n \geq 2$, is a USO if and only if all subgraphs induced by 2-dimensional faces have unique sinks.*

Exercise 3.3

(i) Given a USO of C_n, prove that reorienting all edges with a fixed label a gives rise to a USO again.

(ii) Prove Lemma 3.5!

Exercise 3.4 *Prove that there exists a* deterministic *algorithm for finding the sink of any USO of the n-cube with*

$$O(\sqrt{3}^n) \approx 1.732^n$$

vertex evaluations.

[4] and also counting the source as a visited vertex

Chapter 4

Zero-Sum Games

In the previous chapter, we have seen an algorithm for finding the sink of a 2-dimensional USO with an expected number of $43/20$ vertex evaluations. Even though this certainly improves the bound of

$$\left(\frac{3}{2}\right)^2 = \frac{45}{20}$$

that we get from the product algorithm (page 38), the question remained open whether the improved bound is best possible. This chapter introduces a technique for proving that this is indeed the case. Even more, the technique allows us to *compute* the best possible algorithm: even if we hadn't been clever enough to come up with the algorithm in Section 3.5, we would get it now.

The technique is based on *game theory*, in particular the theory of *zero-sum games*. Already in the introduction, we have described Randomix's search for a fast strategy to escape Determinatus's maze as a game between the two, and we come back to this view here.

4.1 Basics

We have two players, the *algorithm player*, and the *adversary*. The algorithm player has a set of n algorithms at her disposal,

$$\mathcal{A} = \{A_1, \ldots, A_n\},$$

while the adversary holds a set of m inputs for the algorithms,

$$\mathcal{I} = \{I_1, \ldots, I_m\}.$$

We also call the elements of $\mathcal{A}$ and $\mathcal{I}$ *pure strategies* of the players.

Then, there is an $(n \times m)$-matrix M such that m_{ij} denotes the runtime of algorithm A_i on input I_j. M is the so-called *payoff matrix*. Assuming that the algorithm player has to pay the adversary \$1 for every unit of runtime, m_{ij} is the payoff the adversary receives (equivalently, the amount the algorithm player has to pay—a zero-sum situation) when A_i is run on I_j.

Definition 4.1

(i) A mixed *strategy of a player is a probability distribution over his or her set of pure strategies. We encode a mixed strategy of the algorithm player by an n-vector of probabilities*

$$x = (x_1, \ldots, x_n), \quad \sum_{i=1}^{n} x_i = 1, \; \forall i : x_i \geq 0$$

and a mixed strategy of the adversary by an m-vector of probabilities

$$y = (y_1, \ldots, y_m), \quad \sum_{j=1}^{m} y_j = 1, \; \forall j : y_j \geq 0.$$

(ii) Every mixed strategy x of the algorithm player defines a randomized algorithm $\mathcal{A}(x)$*: choose algorithm A_i with probability x_i.*

(iii) Every mixed strategy y of the adversary defines a random input $\mathcal{I}(y)$*: choose input I_j with probability y_j.*

Now we can describe the game and its goal: independently, the algorithm player chooses a randomized algorithm $\mathcal{A}(x)$, and the adversary chooses a random input $\mathcal{I}(y)$. Given these choices, the payoff is the expected runtime of $\mathcal{A}(x)$ on $\mathcal{I}(y)$, which is

$$\begin{aligned}
& \sum_{i,j} m_{ij} \operatorname{prob}(\mathcal{A}(x) = A_i, \mathcal{I}(y) = I_j) \\
= \; & \sum_{i,j} m_{ij} \operatorname{prob}(\mathcal{A}(x) = A_i) \operatorname{prob}(\mathcal{I}(y) = I_j) \quad \text{(independence of choice)} \\
= \; & \sum_{i,j} m_{ij} x_i y_j \\
= \; & x^T M y.
\end{aligned}$$

The algorithm player wants to minimize the expected runtime, while the adversary wants to maximize it. More precisely, the algorithm player wants to choose x in such a way that

$$\max_y x^T M y,$$

the expected runtime of $\mathcal{A}(x)$ on its worst random input, is as small as possible.

The adversary, on the other hand, attempts to find some y such that

$$\min_x x^T M y,$$

the expected runtime of the best randomized algorithm on $\mathcal{I}(y)$, is as large as possible.

Let us do a simple example to illustrate these concepts. The problem of finding the sink in a USO of C_1 can be formulated as a game between the algorithm player and the adversary, where both can choose between two mixed strategies.

The algorithm player may first evaluate the vertex $\emptyset$ (pure strategy A_1), or the vertex $\{1\}$ (pure strategy A_2). If the sink has been missed, the other vertex is evaluated. Obviously, the algorithm player has other, stupid, strategies: she could evaluate the wrong vertex over and over again. As it is clear that the best randomized algorithm will choose such stupid strategies with probability 0, we can as well omit them from our considerations.

The adversary may provide the USO in which vertex $\emptyset$ is the sink (pure strategy I_1), or the USO in which $\{1\}$ is the sink (pure strategy I_2). The resulting payoff matrix is

$$M = \begin{pmatrix} 1 & 2 \\ 2 & 1 \end{pmatrix}.$$

Figure 4.1 depicts the situation.

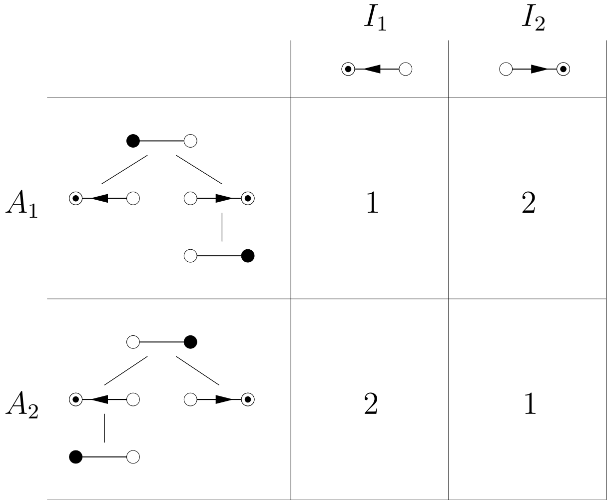


Figure 4.1: The 1-dimensional USO game

A natural question is whether randomized algorithms of type $\mathcal{A}(x)$ cover *all possible* randomized algorithms for solving the problem at hand. If not, the best $\mathcal{A}(x)$ we are computing is not necessarily the best randomized algorithm for the problem.

In fact, there are natural randomized algorithms that do not arise from a probability distribution over a finite set of deterministic ones. For example, the next chapter will discuss a randomized algorithm for finding the sink of a USO, which traverses a chain of adjacent vertices until the sink is found; from each vertex, it follows a *random* outgoing edge to the next vertex. If the USO has cycles, there is no upper bound on the runtime of this algorithm in the worst case,[1] and therefore the algorithm cannot be a combination of finitely many deterministic algorithms.

In the setup we are trying to analyze here, there is only a finite number of possible deterministic algorithms for finding the sink of an n-cube USO, because every such algorithm is uniquely characterized by a finite number of sequences with at most 2^n vertices each,[2] sequence j corresponding to the evaluation order on the j-th USO.

[1]here, the worst case is taken over the initial vertex and the random choices of the algorithm

[2]remember that we got rid of stupid strategies that reevaluate vertices

Therefore, any conceivable randomized algorithm A will—when we run it on all USOs simultaneously—specialize to one of these deterministic algorithms. If x_i is the probability that it specializes to algorithm i, we have $A = \mathcal{A}(x)$.

4.2 Solving the Game

Finding the optimal mixed strategy for the adversary. Assume the algorithm player knows the random input $\mathcal{I}(y)$ chosen by the adversary. Here, knowing it means to know the adversary's probability distribution y. The best randomized algorithm $\mathcal{A}(x)$ on $\mathcal{I}(y)$ (the one with smallest expected runtime which we also call *best response* against y) is given by any optimal solution x to the *linear program*

$$\begin{array}{lll} (\text{LP}_{M,y}) & \text{minimize} & x^T M y \\ & \text{subject to} & \sum_{i=1}^n x_i = 1, \\ & & x_i \geq 0, \quad i = 1, \ldots, n \end{array}$$

in the variables $x_1, \ldots, x_n$. In general, a linear program (LP) is the task of minimizing or maximizing a linear *objective function* in some number of variables, subject to linear (in)equalities involving the variables. Linear programs can efficiently be solved, meaning that a *feasible solution* (a tuple of values for the variables that satisfies all (in)equalities, if that is possible) can be computed, for which the objective function reaches its best possible value among the set of all feasible solutions. Such a solution is called an *optimal solution*, and the corresponding objective function value is the *optimal value* of the LP (also called *minimum value* for minimization and *maximum value* for maximization problems). There is accessible software for solving LPs (the widely used *Maple* program contains a linear programming solver, for example). Linear programs are often written in vector and matrix notation: the above LP, for example, can compactly be written as

$$\begin{array}{lll} (\text{LP}_{M,y}) & \text{minimize} & c^T x \\ & \text{subject to} & Ax = b, \\ & & x \geq 0, \end{array}$$

where

$$c = My \in \mathbb{R}^n, \quad A = (1, \ldots, 1) \in \mathbb{R}^{1 \times n}, \quad b = 1 \in \mathbb{R}^1,$$

and (in)equalities hold component-wise.

Let us denote the minimum value $x^T My$ of ($\text{LP}_{M,y}$) by $f_M(y)$. With this notion, the goal of the adversary is to find an optimal solution y to the problem

$$\begin{array}{ll} \text{maximize} & f_M(y) \\ \text{subject to} & \sum_{j=1}^m y_j = 1, \\ & y_j \geq 0, \quad j = 1, \ldots, m. \end{array}$$

Let Φ_M be the maximum value of $f_M(y)$ in this optimization problem, which is unfortunately no longer a linear program: the function to maximize is not linear, but is itself the

Theorem 4.2 *Consider the two linear programs*

$$\begin{array}{llrcrcl} \text{(LP)} & \text{maximize} & c^T x & + & d^T y & & \\ & \text{subject to} & Ex & + & Fy & \leq & a, \\ & & Gx & + & Hy & = & b, \\ & & & & x & \geq & 0 \end{array}$$

in the variable vectors x and y (and fixed vectors resp. matrices c, d, E, F, G, H of appropriate sizes), and

$$\begin{array}{llrcrcl} (\text{LP}^{\triangle}) & \text{minimize} & a^T w & + & b^T z & & \\ & \text{subject to} & E^T w & + & G^T z & \geq & c, \\ & & F^T w & + & H^T z & = & d, \\ & & & & w & \geq & 0 \end{array}$$

in the variable vectors w and z of appropriate sizes. (LP) and (LP$^{\triangle}$) are called dual *to each other, and their optimal values are equal.*

Proof. We only prove the easy direction here, namely that

$$c^T x + d^T y \leq a^T w + b^T z$$

for all vectors x, y, w, z which satisfy the (in)equalities of their respective programs. This proves that the optimal value of (LP$^{\triangle}$) is an upper bound for the optimal value of (LP), and this fact is known as *weak LP duality*.
Let x, y be any vectors satisfying the (in)equalities of (LP). If $w \geq 0$ and z arbitrary, we can multiply the inequalities of (LP) from the left with w^T and the equalities from the left with z^T. Doing this, the (in)equalities are preserved, and adding them up, we conclude that

$$w^T Ex + w^T Fy + z^T Gx + z^T Hy \leq w^T a + z^T b.$$

If w and z satisfy all (in)equalities of (LP$^{\triangle}$), we get (using $x \geq 0$) the desired inequality

$$c^T x + d^T y \leq \underbrace{w^T Ex + z^T Gx}_{\geq c^T x} + \underbrace{w^T Fy + z^T Hy}_{= d^T y} \leq w^T a + z^T b.$$

$\square$

Remark: In the above form, any maximization (minimization) problem has all its inequalities of type "$\leq$" ("$\geq$"), but this is of course not a restriction: any inequality in the other direction can be multiplied by -1 in order to arrive at a linear program in the form of (LP) or (LP$^{\triangle}$).

Table 4.1: A crash course on LP duality

solution to a minimization problem. Luckily, there is a way out using *linear programming duality*, see Table 4.1.

The duality theorem tells us that $f_M(y)$ is also the maximum value of the linear program dual to ($\text{LP}_{M,y}$), which is the following LP in just one variable u:[3]

$$\begin{array}{lll} (\text{LP}^{\triangle}_{M,y}) & \text{maximize} & u \\ & \text{subject to} & M_i y \geq u, \quad i = 1, \ldots, n. \end{array}$$

Here, M_i is the i-th row of M.

Now, the adversary's task of finding a mixed strategy y with the largest possible value Φ_M of $f_M(y)$ can be solved by simply letting y vary over all possible choices in ($\text{LP}^{\triangle}_{M,y}$). This leads us to the linear program

$$\begin{array}{lll} (\text{LP}_{M}) & \text{maximize} & u \\ & \text{subject to} & M_i y \geq u, \quad i = 1, \ldots, n, \\ & & \sum_{j=1}^{m} y_j = 1, \\ & & y_j \geq 0, \quad j = 1, \ldots, m, \end{array}$$

whose maximum value is Φ_M. A mixed strategy $\tilde{y}$ that leads to this maximum value can be read off the solution to this linear program and defines the optimal random input the adversary is looking for. The value Φ_M is the expected runtime of the best randomized algorithm for this particular random input.

Finding the optimal mixed strategy for the algorithm player. This is now completely symmetric. Given the distribution x of the algorithm player, the worst random input $\mathcal{I}(y)$ for $\mathcal{A}(x)$ (the best response of the adversary against x) is any optimal solution y to the LP

$$\begin{array}{lll} (\text{LP}_{x,M}) & \text{maximize} & x^T M y \\ & \text{subject to} & \sum_{j=1}^{m} y_j = 1, \\ & & y_j \geq 0, \quad j = 1, \ldots, m. \end{array}$$

By the duality theorem, the maximum value $g_M(x)$ of this LP coincides with the minimum value of the dual LP in just one variable v:

$$\begin{array}{lll} (\text{LP}^{\triangle}_{x,M}) & \text{minimize} & v \\ & \text{subject to} & \left(M^T\right)_j x \leq v, \quad j = 1, \ldots, m. \end{array}$$

Consequently, the task of the algorithm player, namely to choose x in such a way that $g_M(x)$ achieves its minimum value Ψ_M, is solved by letting x vary over all possible choices in ($\text{LP}^{\triangle}_{x,M}$), leading to the LP

$$\begin{array}{lll} (\text{LP}^{\triangle}_{M}) & \text{minimize} & v \\ & \text{subject to} & \left(M^T\right)_j x \leq v, \quad j = 1, \ldots, m, \\ & & \sum_{i=1}^{n} x_i = 1, \\ & & x_i \geq 0, \quad i = 1, \ldots, n, \end{array} \tag{4.1}$$

[3]check that it is really the dual!

whose minimum value is Ψ_M. A mixed strategy $\tilde{x}$ that leads to this minimum value can be read off the solution to the linear program and defines the optimal randomized algorithm the algorithm player is looking for. The value Ψ_M is the expected runtime of this algorithm on its worst random input.

The naming of the LP already anticipates the punchline, namely that the latter LP is the dual (check this!) of the adversary's (LP$_M$)!

This implies the following

Theorem 4.3 *Let $\tilde{y}$ be an optimal solution to* (LP$_M$), *Φ_M its optimal value, and let $\tilde{x}$ be an optimal solution to* (LP$^{\triangle}_M$), *Ψ_M its optimal value. Then*

(i) $\Phi_M = \Psi_M = f_M(\tilde{y}) = g_M(\tilde{x}) = \tilde{x}^T M \tilde{y}$, *and*

(ii) $\tilde{x}$ is a best response of the algorithm player against $\tilde{y}$, and $\tilde{y}$ is a best response of the adversary against $\tilde{x}$.

Proof. (i) $\Phi_M = \Psi_M$ is LP duality. By definition, $\Phi_M = f_M(\tilde{y})$ and $\Psi_M = g_M(\tilde{x})$, so $f_M(\tilde{y}) = g_M(\tilde{x})$. Because $f_M(\tilde{y})$ is the minimum value of (LP$_{M,\tilde{y}}$), while $\tilde{x}^T M \tilde{y}$ is just some value that may occur, we get

$$f_M(\tilde{y}) \leq \tilde{x}^T M \tilde{y}.$$

Similarly, it follows that

$$g_M(\tilde{x}) \geq \tilde{x}^T M \tilde{y},$$

so all values must be equal.

(ii) According to (i), $\tilde{x}$ is an optimal solution (and therefore a best response) to (LP$_{M,\tilde{y}}$), and $\tilde{y}$ is an optimal solution (and therefore a best response) to (LP$_{\tilde{x},M}$). □

The theorem allows us to establish two important notions.

Definition 4.4

(i) The value $\Phi_M = \Psi_M$ is called the value *of the zero-sum game defined by the payoff matrix M. It equals the runtime of the randomized algorithm with the best expected worst case performance (see Exercise 4.2 for an exact statement of what this means).*

(ii) An equilibrium *of the game is any pair of mixed strategies $(\tilde{x}, \tilde{y})$ that are best responses with respect to each other.*

We have shown that a zero-sum game always has an equilibrium $(\tilde{x}, \tilde{y})$. The pair has the interesting property that even if the players *know* each others equilibrium strategies, none of them can profit by changing its own mixed strategy. One can say that in an equilibrium, both players win, because both perform as well as they possibly can.

As a small example, let us solve the game for the 1-dimensional USO of Figure 4.1 by the above techniques. Recall that

$$M = \begin{pmatrix} 1 & 2 \\ 2 & 1 \end{pmatrix}$$

in this case, so the program (LP_M) reads as

$$
\begin{array}{llrcrcl}
(\mathrm{LP}_M) & \text{maximize} & u & & & & \\
& \text{subject to} & y_1 & + & 2y_2 & \geq & u, \\
& & 2y_1 & + & y_2 & \geq & u, \\
& & y_1 & + & y_2 & = & 1, \\
& & & & y_1, y_2 & \geq & 0.
\end{array}
$$

We expect the maximum value of u (the value of the game) to be $\tilde{t}(1) = 3/2$, see Observation 3.3. Indeed, if $y_1 = y_2 = 1/2$ and $u = 3/2$, we get a feasible solution to the LP, so the maximum value is at least $3/2$. On the other hand, adding up the first two inequalities of the LP, we obtain

$$2u \leq 3y_1 + 3y_2 = 3$$

for all feasible solutions, because of the equality constraint. Therefore, $3/2$ is also an upper bound on the value of the game, so $3/2$ is the game value. The corresponding optimal values $\tilde{y}_1 = \tilde{y}_2 = 1/2$ define the best mixed strategy of the adversary: choose between the two USOs of the 1-cube uniformly at random.

In this easy case, the dual linear program ($LP_M^{\triangle}$) for finding the optimal mixed strategy for the algorithm player looks completely similar (still, I encourage the reader to write it down) and gives rise to the optimal solution $\tilde{x}_1 = \tilde{x}_2 = 1/2$: choose the first vertex to evaluate uniformly at random between the two vertices.

4.3 Game Trees

We have not developed the machinery in order to deal with the toy problem of the 1-cube, but to find the value of the game in the 2-dimensional case (more precisely, to certify that the upper bound of $43/20$ we have found in the previous chapter is actually the game value). In principle, we can do this now as we did it in the 1-dimensional case in Figure 4.1: write down all possible pure strategies of the algorithm player and evaluate their performance on all possible 12 USOs (the pure strategies of the adversary); this gives the payoff matrix M which is all we need to write down (LP_M). Then feed this linear program to any LP solver and read off the game value.

The only problem is that the number of pure strategies the algorithm player has at her disposal is already quite large (we'll see why), so the LP would be quite large as well. Consequently, the LP solver would be slow, or not even able to process the problem. The approach we discuss in this section leads to a much smaller linear program which even the solver of the *Maple* system[4] can handle in just a few seconds.

First, let us understand why we have an explosion in the number of pure strategies. Adapting the approach of Figure 4.1, a pure strategy can be written as a tree: the root specifies the first vertex to evaluate, and its children correspond to the possible answers of the adversary. For every such answer not corresponding to the sink, the strategy specifies the second vertex to evaluate, and so on, see Figure 4.2.

[4]The very general and versatile *Maple* system is a popular tool, but not primarily because of its speed.

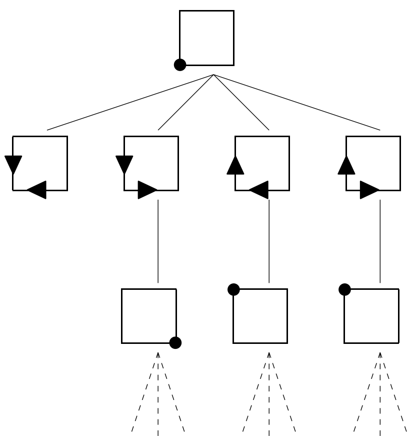


Figure 4.2: Specifying a pure strategy of the algorithm player

Already for the second vertex to evaluate, there are three possible choices[5] for every answer of the adversary (nodes at depth 1), if the sink has not been evaluated so far. This already gives rise to $3 \cdot 3 \cdot 3 = 27$ possible combinations, and any such combination splits up into much more combinations further down the tree.

You can argue that this way of encoding pure strategies is not very efficient, because the already quite large structure of Figure 4.2 is repeated for *all* strategies that agree with the depicted one in the first two vertex evaluations. It should be possible for these strategies to share the encoding of the first two evaluations.

Indeed, we can encode all possible ways in which the game can be played in just one *game tree*, where every path down the tree corresponds to an alternating *move sequence* of moves by the algorithm player and moves by the adversary.

In this formulation, we have to be careful not to let the adversary become too powerful. Recall from the Introduction (Section 1.3) that we are assuming the adversary to be *oblivious*, i.e., the adversary must choose his input—and stick to it—*before* the algorithm player asks her first question. In the game tree approach, we can model this by letting the adversary make the first move. However, it is a *hidden* move unknown to the algorithm player. Only through vertex evaluations, the algorithm player obtains information about the hidden move.

A part of the resulting game tree is depicted in Figure 4.3. Note that after the hidden move, the adversary always has just *one* possible move, because his answer is determined by the choice of the USO in the hidden move. The nodes in which it's the algorithm player's turn are partitioned into *information sets*: such a set collects all nodes corresponding to the same sequence of questions and answers so far. At every node of a fixed information set, the algorithm player has the same knowledge.

At depth 1, all nodes are in the same information set, but later, information sets split up because the algorithm player gets additional knowledge about the adversary's first hidden move.

How can we encode a pure strategy of the algorithm player in the game tree? We must specify at each information set the next vertex to be evaluated. In this way, the next move takes

[5] again, we do not consider strategies that reevaluate vertices

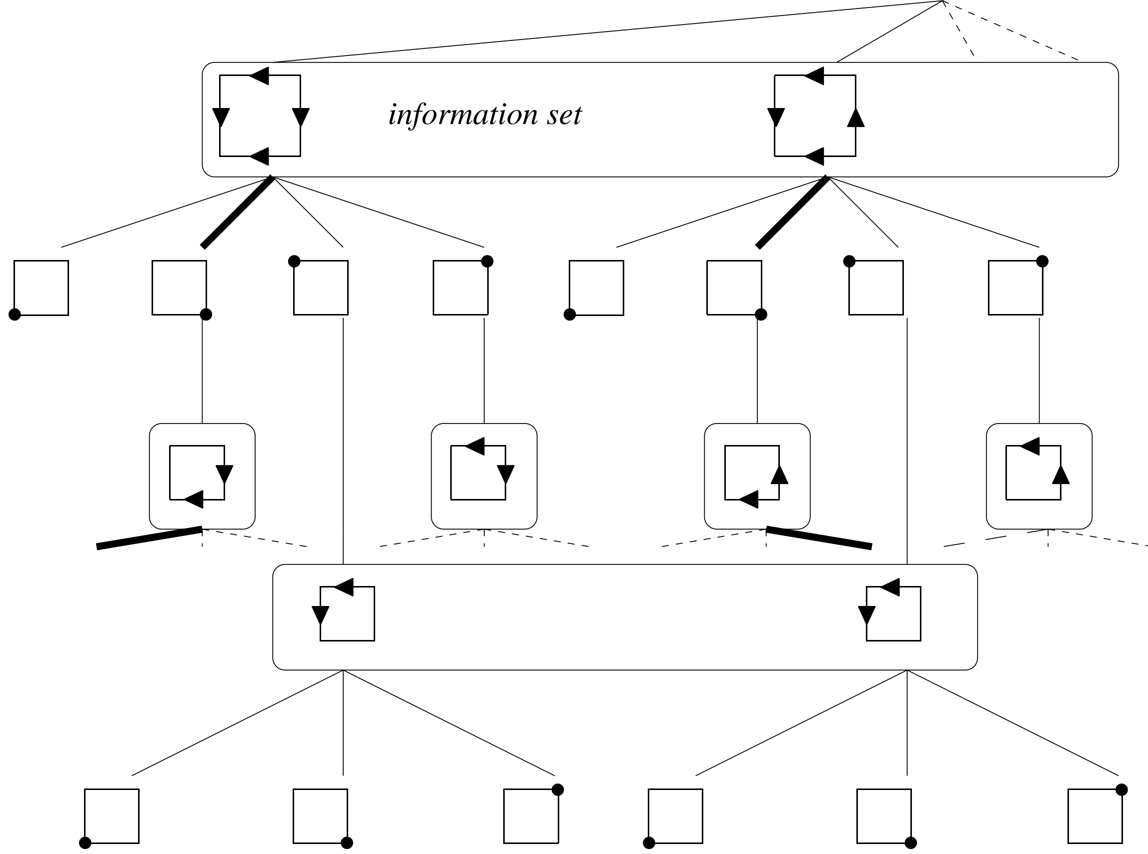


Figure 4.3: Game tree for the 2-dimensional USO game; the part of the information set at level 1 that is shown subdivides into four information sets containing a single node each, and one information set with two nodes.

into account exactly the information the algorithm player has obtained so far. For information sets that are not reachable due to earlier choices, no move must be specified, of course.

The selected move is applied at *all* nodes of the information set. In Figure 4.3, the bold edges belong to a possible pure strategy.

Even here, the number of pure strategies can in general (for example, in the n-dimensional USO game) be exponential in the size of the game tree; we can see this in Figure 4.3. At depth 3 of the tree, there are two *parallel* information sets that are reached through an earlier move, and at each of them, we can independently choose between three moves, leading again to a multiplication of the number of possibilities.

4.4 The Sequence Form

Here comes the statement that finally saves us: we can deal with *mixed* strategies (and this is what we want in the end), without ever looking at pure strategies! Instead, we assign probabilities to *move sequences*.

Theorem 4.5 *Every mixed strategy of the algorithm player can be obtained by specifying for each information set $\mathcal{S}$ and each possible move e in $\mathcal{S}$ the probability $p_{\mathcal{S},e}$ that e is played by the algorithm player, equivalently that the complete* move sequence *ending in e is played.*

We will not prove this here, but the statement is quite intuitive: instead of deterministically fixing how we leave an entered information set (pure strategy), we now "distribute" the probability of entering the information set further among the successor moves.

The randomized algorithm resulting from a mixed strategy specified according to Theorem 4.5 is quite natural: whenever the game reaches a node in an information set S with k possible moves $e_1, \ldots, e_k$, select $e \in \{e_1, \ldots, e_k\}$ with probability

$$\frac{p_{\mathcal{S},e}}{\sum_{\ell=1}^{k} p_{\mathcal{S},e_\ell}}.$$

The sum in the denominator must be nonzero, because it equals the probability of reaching S in the first place.

The number of probabilities we have to specify is bounded by the number of edges in the game tree, which is *linear* in the size of the game tree. To encode a mixed strategy in the "traditional" way, we would have to assign a probability to each of the *exponentially* many pure strategies.

As before, in every node of the same information set, we must distribute the "incoming" probability in the same way, as several edges correspond to the same move. Figure 4.4 shows a part of a mixed strategy in which the probability is x_3 that the upper left vertex is evaluated first; if the outmap pattern is

(which is the case for the two USOs depicted in the figure), the probability x_3 is distributed further among the three possible next moves. On the other hand, if we evaluate the lower right vertex first (with probability x_2), we may see different outmaps, and in the corresponding parallel information sets, x_2 is split independently.

It is important to understand that the algorithm player plays *several* move sequences, and not just one, so the probabilities we assign will not form a probability distribution. Most obviously, if a sequence is played, every prefix of it is played as well. But the actual reason are parallel information sets: in *any* of them, a move will be selected according to the assigned probabilities, although in the actual game, the adversary's answers will only "activate" one of these moves.

4.5 Solving the Game in Sequence Form

The idea is to generalize the LP approach, and we illustrate all steps using the concrete example of the 1-dimensional USO game. Figure 4.5 shows the complete game tree for this case, along with variables for the move probabilities of both players. We also introduce 'names' for the moves which are more intuitive than just the variables.

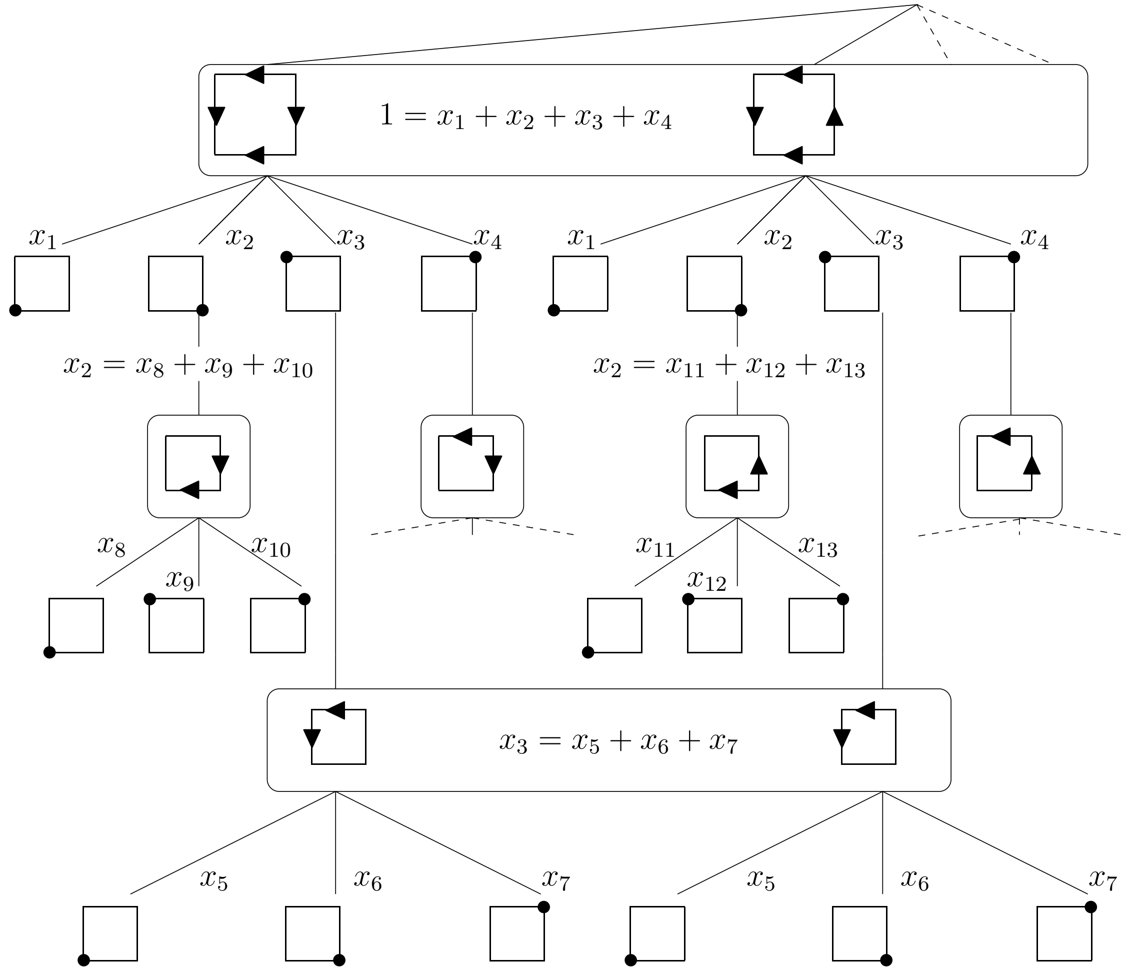


Figure 4.4: Specifying a mixed strategy by assigning probabilities to moves

The game is defined by the joint behavior of all move sequences of the algorithm player on all USOs of the adversary. This allows us to set up a payoff matrix M, whose entry m_{ij} is the number of vertex evaluations of the algorithm player's i-th move sequence on the adversary's j-th USO. It may happen that such a pair does not correspond to a legal way of playing the game, or that it corresponds to an incomplete game. For example, the move sequence LR is impossible on the USO 1, and the move sequence L is incomplete on the USO 2. In both cases, the corresponding entry of M is 0.

In Figure 4.5, the adversary has the USOs 1 and 2 at his disposal, associated with variables y_1, y_2, while the algorithm player has move sequences

$$L, R, LR, RL,$$

with variables $x_1, \ldots, x_4$. This yields the payoff matrix

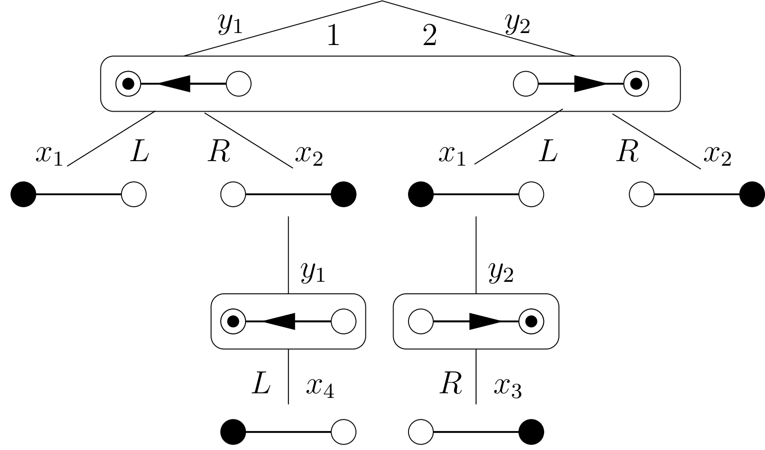


Figure 4.5: Tree for the 1-dimensional USO game with move probabilities and labels

$$M = \begin{matrix} 1 & 2 \\ \downarrow & \downarrow \end{matrix} \\ \begin{pmatrix} 1 & 0 \\ 0 & 1 \\ 0 & 2 \\ 2 & 0 \end{pmatrix} \begin{matrix} \leftarrow L \\ \leftarrow R \\ \leftarrow LR \\ \leftarrow RL \end{matrix} . \tag{4.2}$$

Observation 4.6 *If x_i is the algorithm player's probability of playing the i-th sequence, and y_j is the adversary's probability of choosing the j-th USO, the expected number of vertex evaluations is*

$$x^T My,$$

where $x = (x_i), y = (y_j)$ are the vectors collecting the x_i and y_j, respectively.

Recall that x is not a probability distribution, because the algorithm player plays several move sequences simultaneously, where the one that actually shows up in the game depends on the adversary's answers.

Still, the observation can be proved as before: the contribution of the pair (i, j) to the expected runtime is $x_i y_j m_{ij}$, because x and y are chosen independently. This also holds if $m_{ij} = 0$: an incomplete sequence does not contribute anything, because the runtime will be counted for the complete supersequences. Also, an illegal sequence does not contribute, because it does not get activated during the game. In other words, (i, j) contributes to the runtime if and only if the game reaches a leaf of the game tree when move sequence i is applied to USO j.

As before, we are concerned with computing optimal values for the probabilities x_i and y_j. Assuming, the algorithm player knows the distribution y chosen by the adversary, her best

response is again given by the solution to a linear program

$$
\begin{array}{lll}
(\mathrm{LP}_{M,y}) & \text{minimize} & x^T M y \\
& \text{subject to} & Ex = e, \\
& & x \geq 0,
\end{array}
$$

whose optimal value we denote by $f_M(y)$. The constraint set $Ex = e$ contains one constraint for every information set, specifying that the probability of entering it must be equal to the probability of leaving it. In Figure 4.4, we see the constraints for four of the information sets. In the 1-dimensional case of Figure 4.5, the constraints are

$$
\begin{aligned}
x_1 + x_2 &= 1, \\
x_1 &= x_3, \\
x_2 &= x_4,
\end{aligned}
$$

so we have

$$
E = \begin{pmatrix} 1 & 1 & 0 & 0 \\ 1 & 0 & -1 & 0 \\ 0 & 1 & 0 & -1 \end{pmatrix}, \quad e = \begin{pmatrix} 1 \\ 0 \\ 0 \end{pmatrix}.
$$

The dual of ($\mathrm{LP}_{M,y}$) is the linear program

$$
\begin{array}{lll}
(\mathrm{LP}^{\triangle}_{M,y}) & \text{maximize} & e^T u \\
& \text{subject to} & E^T u \leq My
\end{array}
$$

in the variable vector u, and its optimal value is $f_M(y)$ as well. Thus, the adversary maximizes $f_M(y)$ by solving the linear program

$$
\begin{array}{lll}
(\mathrm{LP}_{M}) & \text{maximize} & e^T u \\
& \text{subject to} & E^T u \leq My \\
& & \sum_{j=1}^m y_j = 1, \\
& & y_j \geq 0, \quad j = 1, \ldots, m,
\end{array}
\tag{4.3}
$$

whose optimal value is Φ_M.

On the other hand, given x, the best response of the adversary is obtained by solving

$$
\begin{array}{lll}
(\mathrm{LP}_{x,M}) & \text{maximize} & x^T M y \\
& \text{subject to} & \sum_{j=1}^m y_j = 1, \\
& & y_j \geq 0, \quad j = 1, \ldots, m,
\end{array}
$$

whose dual is

$$
\begin{array}{lll}
(\mathrm{LP}^{\triangle}_{x,M}) & \text{minimize} & v \\
& \text{subject to} & \left(M^T\right)_j x \leq v, \quad j = 1, \ldots, m,
\end{array}
$$

and both programs have optimal value $g_M(x)$. Therefore, the algorithm player minimizes $g_M(x)$ by solving

$$
\begin{array}{lll}
(\text{LP}_M^{\triangle}) & \text{minimize} & v \\
& \text{subject to} & \left(M^T\right)_j x \leq v, \quad j = 1, \ldots, m, \\
& & Ex = e, \\
& & x \geq 0,
\end{array}
\tag{4.4}
$$

and the resulting optimal value is Ψ_M. As before, $\Phi_M = \Psi_M$ (and this is the *value* of the game), because (4.3) and (4.4) are dual to each other.

Writing down (LP$_M^{\triangle}$) for the 1-dimensional USO game yields the problem

$$
\begin{array}{ll}
\text{minimize} & v \\
\text{subject to} & x_1 + 2x_4 \leq v \\
& x_2 + 2x_3 \leq v \\
& x_1 + x_2 = 1, \\
& x_1 = x_3, \\
& x_2 = x_4, \\
& x \geq 0.
\end{array}
$$

Setting $x_i = 1/2$ for all i and $v = 3/2$ leads to a feasible solution, so the game value is at most $3/2$. On the other hand, adding up the first two inequalities, and using the three equality constraints, we get that

$$2v \geq x_1 + x_2 + 2x_3 + 2x_4 = 3x_1 + 3x_2 = 3,$$

so $v \geq 3/2$. It follows that $3/2$ is the game value, and $x_1 = x_2 = x_3 = x_4 = 1/2$ describes the optimal mixed strategy of the algorithm player. By now, this result does not come as a real surprise anymore.

What about the 2-dimensional case? The game tree is larger, of course, but not of overwhelming size: it consists of twelve subtrees which are symmetric copies of the tree in Figure 4.6, one subtree for each USO of C_2. Various information sets connect nodes across subtrees. I have generated the resulting linear program (LP$_M^{\triangle}$) by computer. It has 226 variables

$$v, x_1, \ldots x_{225},$$

and in addition to the nonnegativity constraints $x \geq 0$, it has

12 inequalities, one for every USO of C_2,

and

142 equalities, one for every information set.

The LP solver of *Maple* needs less than 20 seconds to solve this on my computer. The resulting optimal mixed strategy is depicted in Figures 4.6 and 4.7 that represent two of the twelve subtrees of the complete tree, one for a particular *eye*, and one for a particular *bow*. The probability values in all other ten subtrees, for the other eyes and bows, are completely symmetric, so the mixed strategy is already defined by its behavior on *some* eye and *some* bow. And indeed, the value of the game (the value of the variable v in the solution) is $43/20$.

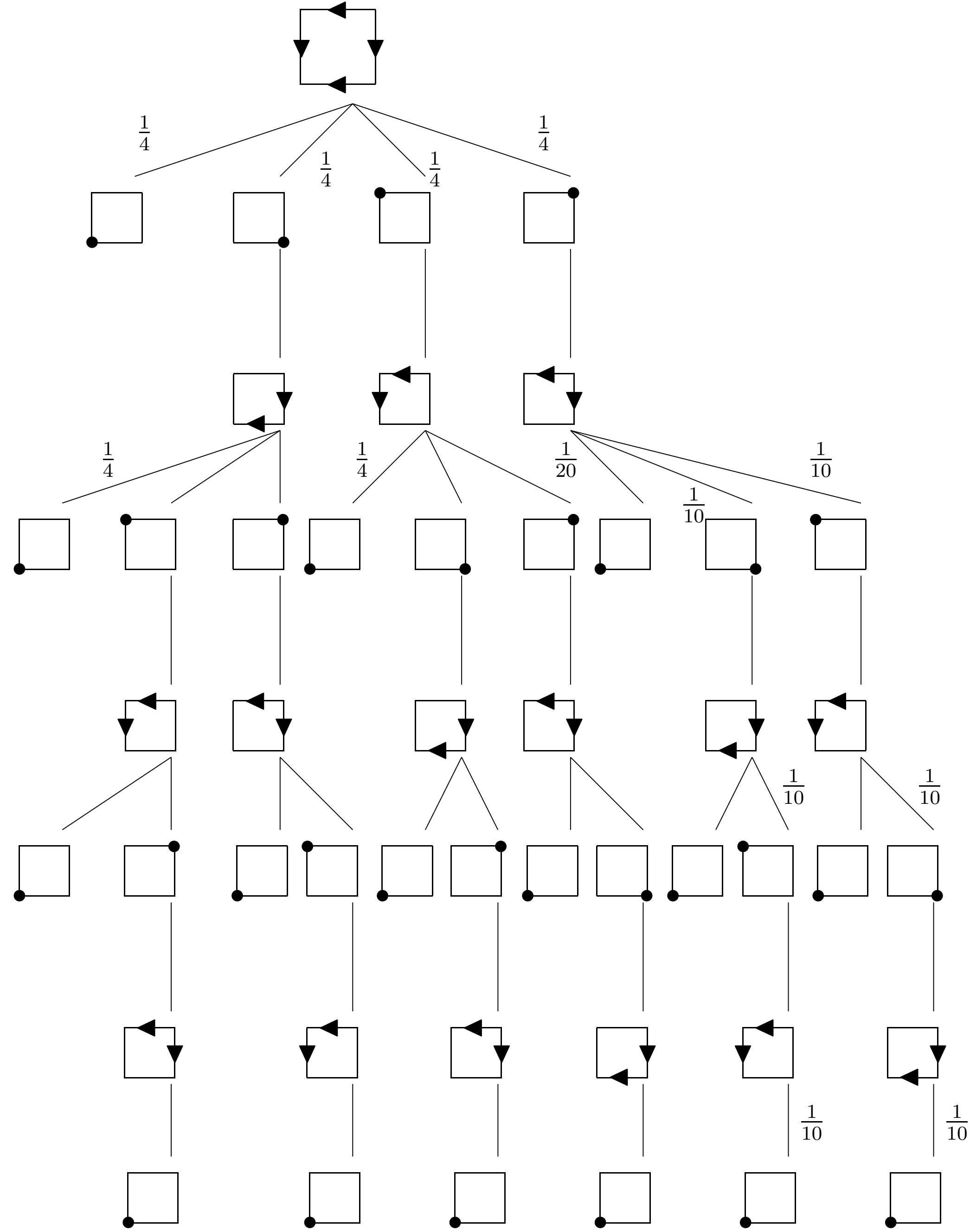


Figure 4.6: The optimal mixed strategy, applied to an eye

Looking at the values (whenever an edge has no value attached to it, the value is 0), you may wonder whether the resulting algorithm is actually the algorithm we have developed in the previous section. A priori, this is not clear, because there may be several algorithms with an expected worst-case performance of $43/20$. Here, the LP solver has in fact delivered our known algorithm. Without going through the details, you can recognize some characteristics of it: for example, the first vertex to be evaluated is chosen uniformly at random; also, in case

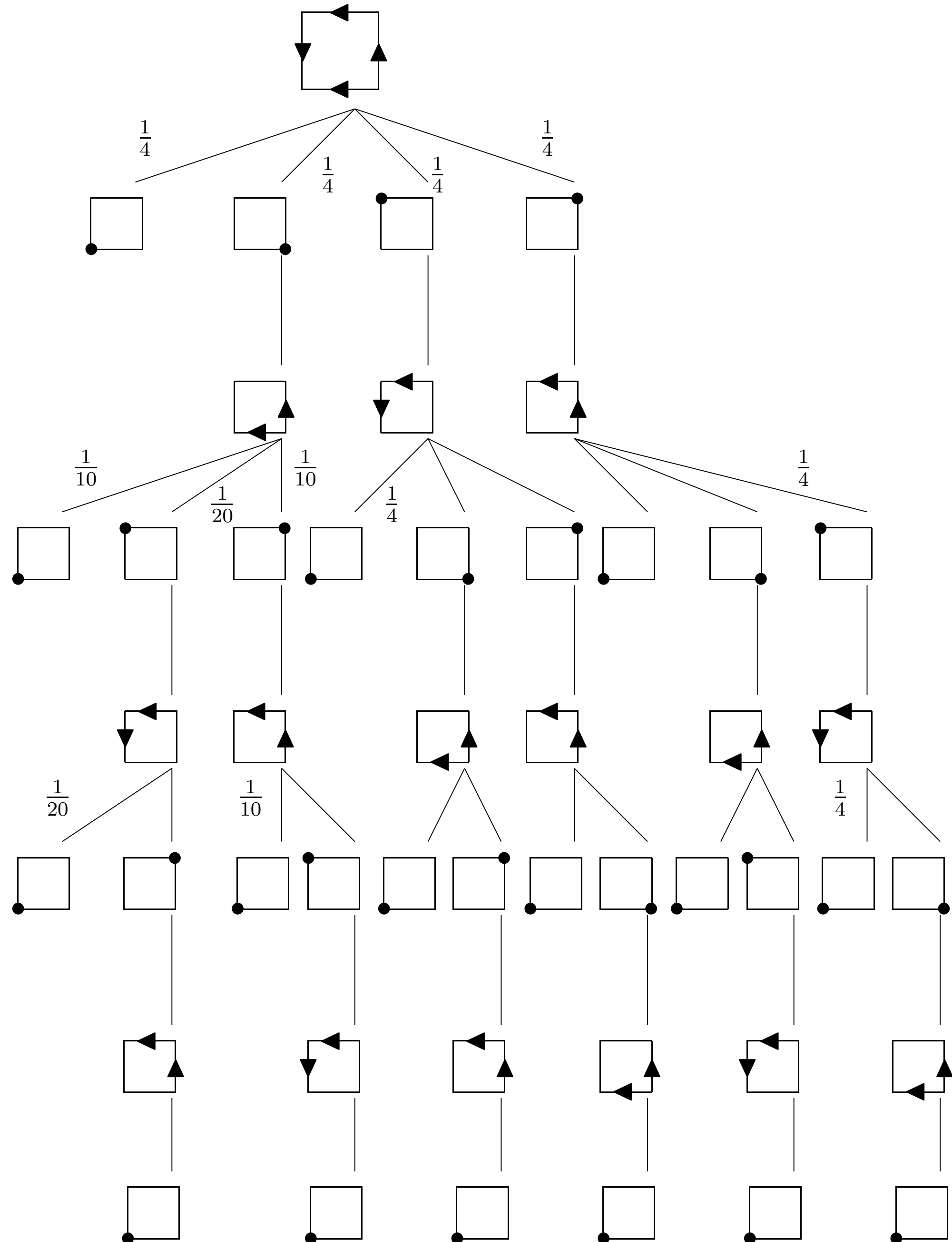


Figure 4.7: The optimal mixed strategy, applied to a bow

of an eye, we may need 4 evaluations, while a bow is always solved with at most 3 evaluations.

4.6 Yao's Theorem

With the machinery we have developed so far, we can now prove *Yao's Theorem*, a powerful tool to establish lower bounds for the expected worst-case performance of the best randomized

algorithm.

As in the beginning of this chapter, we consider the setup in which the algorithm player chooses a randomized algorithm $\mathcal{A}(x)$ defined by a probability distribution x over a finite set of deterministic algorithms, while the adversary chooses a random input $\mathcal{I}(y)$ defined by a probability distribution y over a finite set of inputs. With M being the matrix collecting the runtimes of all algorithms on all inputs,

$$x^T M y$$

is the expected runtime of $\mathcal{A}(x)$ on $\mathcal{I}(y)$.

Using the linear programming approach, we have derived the equality

$$\max_y \min_x x^T M y =: \Phi_M = \Psi_M := \min_x \max_y x^T M y, \tag{4.5}$$

where x and y range over all probability distributions.

An interesting observation is that, for fixed y, we have

$$f_M(y) := \min_x x^T M y = \min_i e_i^T M y, \tag{4.6}$$

and for fixed x,

$$g_M(x) := \max_y x^T M y = \max_j x^T M e_j \tag{4.7}$$

holds, where e_k is the k-th unit vector. The (perhaps surprising) interpretation is that the best response to a fixed strategy can always be chosen as a pure strategy. In other words, when a player knows the other player's strategy, there is no need to randomize anymore. (4.6) follows from

$$x^T M y = \sum_i x_i e_i^T M y,$$

so at least one value $e_i^T M y$ in the right-hand side sum is at most as large as $x^T M y$. But then the smallest possible value of $x^T M y$ over all x (which includes the e_i) must agree with some $e_i^T M y$. The argument for (4.7) is the same.

For any pair (x, y), equations (4.5), (4.6) and (4.7) yield

$$\min_i e_i^T M y = f_M(y) \leq \max_{y'} f_M(y') = \Phi_M = \Psi_M \leq \min_{x'} g_M(x') \leq g_M(x) = \max_j x^T M e_j.$$

Now let $\tilde{x}$ be the optimal mixed strategy for the algorithm player. Then the previous inequality gives *Yao's Theorem*.

Theorem 4.7 *For any mixed strategy y of the adversary,*

$$\min_i e_i^T M y \leq \max_j \tilde{x}^T M e_j.$$

In other words, the expected runtime of the best *deterministic algorithm on the random input $\mathcal{I}(y)$ is a lower bound for the expected worst-case runtime of the* best *randomized algorithm $\mathcal{A}(x)$.*

Thus, in order to prove a lower bound L for the expected worst-case runtime of any randomized algorithm, you choose a suitable distribution y and prove that no deterministic algorithm can be faster than L in expectation, on the random input $\mathcal{I}(y)$. The art here is to find a distribution y that leads to a good lower bound L. Exercise 4.4 asks you to do this for the game of Randomix vs. Determinatus.

Bibliographical Remarks

The presentation of the general method is adapted from the very nice survey by von Stengel [15]. In principle, the application to 2-dimensional USOs was known, but in explicit form, it is new.

Exercises

Exercise 4.1 *Given the payoff matrix*

$$M = \begin{pmatrix} a & b \\ c & d \end{pmatrix}$$

with $a, b, c, d \in \mathbb{R}$*, what are the equilibria* $(\tilde{x}, \tilde{y})$*? In which case is there more than one equilibrium?*

Exercise 4.2 *Let* $\tilde{x}$ *be the optimal mixed strategy computed by the linear program (4.1) on page 48. Prove that*

(i) for any *randomized algorithm* $\mathcal{A}(x)$ *over* $\mathcal{A}$*, there exists an input* I *from* $\mathcal{I}$ *such that the expected runtime of* $\mathcal{A}(x)$ *on* I *is at least* Φ.

(ii) The expected runtime of $\mathcal{A}(\tilde{x})$ *is at most* Φ *for* any *input* I *from* $\mathcal{I}$.

This means, the algorithm $\mathcal{A}(\tilde{x})$ *is the best randomized algorithm in the worst case.*

Exercise 4.3 *The following question has become famous as the* Monty Hall Problem*, and even mathematicians have argued in favor of the wrong answer:*

> *Suppose you're on a game show, and you're given the choice of three doors: Behind one door is a car; behind the others, goats. You pick a door, say No. 1, and the host, who knows what's behind the other doors, opens another door, say No. 3, which has a goat. He then says to you, 'Do you want to pick door No. 2?' Is it to your advantage to take the switch?*

Model this process as a zero-sum game between you and the game show host. Your goal is to maximize your expected payoff (which we may assume to be 1 if you win the car, and 0 otherwise). The game starts with a hidden move by the host who assigns car and goats to the three doors. Then it's your turn to pick a door, followed by the host opening a non-winning door. In the last move of the game, you choose between sticking to your initial choice, or switching to the other closed door.

What is the value of this game? What is your best strategy?

Exercise 4.4 *Use Yao's Theorem to prove that Randomix's strategy of finding the sink in Determinatus's maze (Chapter 1) is the best possible randomized algorithm. If the case of general* n *seems to difficult, try to attack the case* $n = 3$ *explicitly.*

Chapter 5

Random Walks

The theory of random walks is a quite powerful tool in developing and analyzing randomized algorithms. Basically, a random walk "jumps around" on a fixed state space, according to certain transition probabilities between states. For example, Randomix's strategy for escaping Determinatus's maze is a random walk on a state space whose elements are the vertices of a complete graph.

There are two important types of questions in connection with random walks. Questions of the first type ask for the expected number of steps until the walk reaches some prespecified state or set of states. In the second type of questions, we want to know whether we will end up in an approximately random state, given that we perform sufficiently many steps; moreover, we would like to have a bound on the number of steps this takes.

In this chapter, we address both types of questions; we first illustrate them using two easy warm-up examples, and then answer questions of actual interest for unique sink orientations.

5.1 Two Warm-Ups

The casino walk. Suppose you enter a casino with $\$k$ on your hands, $k \geq 0$, and you play roulette (always betting $\$1$ on red), until you either have lost all your money, or you have $\$N$ on your hands, for some prespecified $N \geq k$. Assuming that in every round, the probabilities of winning or loosing $\$1$ are $1/2$ each, what is the expected number of rounds you will play? We note that this question is relevant even outside of the casino, because it appears for example in connection with randomized algorithms for boolean satisfiability, or perfect matchings in regular bipartite graphs.

You might also want to know what the probability of ending up with $\$N$ is, or what happens in the realistic case where the winning probability is less than $1/2$, because the bank always wins if zero comes up. This is covered by Exercise 5.1.

The above process can be modeled as a random walk on the state space $\{0, \ldots, N\}$, where 0 and N are *absorbing*, meaning that the walk ends when one of them is reached. For each $i \in \{1, \ldots, N-1\}$, the probabilities of going to $i-1$ or $i+1$ in the next step are $1/2$ each. We refer to this as the *casino walk*, see Figure 5.1.

Fact 5.1 *For $i \in \{0, \ldots, N\}$, let E_i denote the expected number of steps in the casino walk,*

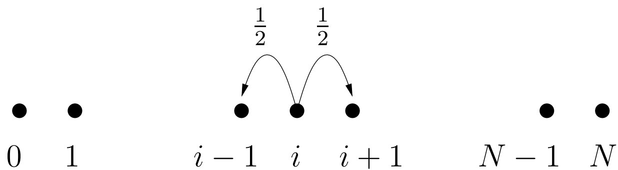


Figure 5.1: The casino walk

starting from i. Then we have

$$E_0 = E_N = 0,$$

and

$$E_i = 1 + \frac{1}{2}E_{i-1} + \frac{1}{2}E_{i+1}, \quad 0 < i < N. \tag{5.1}$$

We want to develop an explicit formula for E_i. The difficulty is that even if we guess the right formula, it cannot be verified by induction, because the value for i depends on values for smaller *and* larger indices. Instead, we use the following

Trick 5.2 *Let*

$$b_i := E_i - E_{i-1} + 2i, \quad 0 < i \leq N.$$

Then $b_i = b_{i-1}$ holds for $1 < i \leq N$. In particular, $b_i = b_1$ for $0 < i \leq N$.

Proof. We compute

$$b_i - b_{i-1} = E_i - 2E_{i-1} + E_{i-2} + 2 = 0,$$

where the latter equality is just twice the defining equation (5.1) for E_{i-1}. □

Using $E_0 = 0$ and $b_1 = E_1 + 2$, we conclude that

$$\begin{aligned} E_i &= b_1 - 2i + E_{i-1} \\ &= ib_1 - 2\sum_{j=1}^{i} j + E_0 \\ &= i(E_1 + 2) - i(i+1) \\ &= i(E_1 - (i-1)), \end{aligned} \tag{5.2}$$

for $0 < i \leq N$ (incidentally, the formula also holds for $i = 0$). Now what is E_1? Using (5.2) for $i = N$ yields

$$0 = E_N = N(E_1 - (N-1)),$$

so $E_1 = N - 1$, and plugging this back into (5.2), we get the following result.

Theorem 5.3 *The expected number E_k of steps in the casino walk, starting from k, satisfies*

$$E_k = k(N-k).$$

For example, if you enter the casino with $\$100$, and you are waiting to get $\$200$, it takes an expected number of $10,000$ bets until you can go home with nothing or $\$200$. Even if the croupier is fast and handles one round of roulette in 30 seconds, you will spend something like 83 hours in the casino.

The provider walk. Suppose that initially, you are a customer of Swisscom mobile, but then you need a new mobile phone. Because providers give out free phones only to new customers, you decide to change the provider; in your everlasting quest for the latest technology, you keep on switching providers every other year.

Because you don't care which provider you end up with, as long as you get a free phone, you switch from your current provider to any of the two others (initially, to Orange or Sunrise), with equal probability $1/2$. What is the expected number of provider changes it takes to be a customer of a more or less random provider?

As before, the process can be modeled as a random walk, this time on a triangle. From any vertex of the triangle, you go to each of the two other ones with probability $1/2$ in the next step. We refer to this as the *provider walk*, see Figure 5.2.

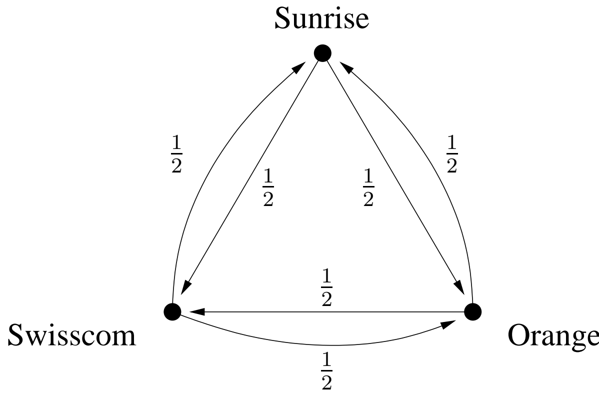


Figure 5.2: The provider walk

Fact 5.4 *For $i \geq 0$ and $A \in \{\text{Swisscom}, \text{Orange}, \text{Sunrise}\}$, let $p_{i,A}$ denote the probability of being customer of provider A after i provider changes. Then we have*

$$p_{0,\text{Swisscom}} = 1, p_{0,\text{Orange}} = p_{0,\text{Sunrise}} = 0,$$

and

$$\begin{aligned}
p_{i,\text{Swisscom}} &= \frac{1}{2}p_{i-1,\text{Orange}} + \frac{1}{2}p_{i-1,\text{Sunrise}}, \\
p_{i,\text{Orange}} &= \frac{1}{2}p_{i-1,\text{Swisscom}} + \frac{1}{2}p_{i-1,\text{Sunrise}}, \\
p_{i,\text{Sunrise}} &= \frac{1}{2}p_{i-1,\text{Swisscom}} + \frac{1}{2}p_{i-1,\text{Orange}},
\end{aligned}$$

for $i > 0$.

Unlike in the casino walk, the following explicit formulae can easily by verified by induction (we don't do this here).

Theorem 5.5 *For $i \geq 0$,*

$$p_{i,\text{Swisscom}} = \frac{1 - (-1/2)^{i-1}}{3}$$

and

$$p_{i,\text{Orange}} = p_{i,\text{Sunrise}} = \frac{1 - (-1/2)^{i}}{3}.$$

Let's check this for small values. If $i = 0$, the theorem indeed yields the boundary conditions of Fact 5.4, and for $i = 1$, it shows

$$(p_{1,\text{Swisscom}}, p_{1,\text{Orange}}, p_{1,\text{Sunrise}}) = (0, \frac{1}{2}, \frac{1}{2}),$$

which is what we expect.

The theorem also tells us that all probabilities tend to $1/3$ for $i \to \infty$. Already for $i = 10$, we have

$$(p_{10,\text{Swisscom}}, p_{10,\text{Orange}}, p_{10,\text{Sunrise}}) \approx (0.33398, 0.33301, 0.33301),$$

which is a pretty good approximation of the uniform distribution. We will never reach the uniform distribution, though: according to Theorem 5.5, the fact that you have started out as a customer of Swisscom will never be "forgotten".

5.2 The `RandomEdge` Algorithm

Trying to mimick Randomix's strategy for finding the exit of Determinatus's maze, we arrive at the following strategy for finding the sink of any given USO: from some initial vertex, proceed along a *random* outgoing edge to an adjacent vertex. Repeat this process until the sink is hit. In fact, the behavior of this strategy on the cyclic USO of C_3 was the subject of Exercise 3.1. Here is the algorithm, written down formally.

Algorithm 5.6

```
RandomEdge_s(X)
  (* s outmap of a USO, X some initial vertex *)
  WHILE s(X) ≠ ∅ DO
    choose i ∈ s(X) uniformly at random
    X := X ⊕ {i}
  END
  RETURN X
```

We are interested in the expected number of vertex evaluations performed by this algorithm in the worst case. An obvious question is whether this expectation exists at all. If the USO has cycles, it is not even clear that the sink is reachable from the initial vertex X, in which case the algorithm would not terminate.[1] The following lemma shows that such pathological situations cannot occur.

[1] if the orientation is acyclic, `RandomEdge` can visit no vertex twice and therefore eventually reaches the unique sink.

Lemma 5.7 *Let $\mathcal{O}$ be a USO of C_n, X the unique sink and Y some other vertex. Then there is a directed path in $\mathcal{O}$ from Y to X, of length $|X \oplus Y|$.*

The proof is Exercise 5.2. Let us remark that no path from Y to X can be shorter: in order to get from Y to X, we must traverse at least one edge in direction i, for all $i \in X \oplus Y$. The lemma can be used to show that `RandomEdge` has finite expected runtime. This in particular implies that the algorithm terminates almost surely (with probability one).

Corollary 5.8 *For any outmap s of an n-cube USO and any initial vertex X, the expected number of vertex evaluations performed by $\texttt{RandomEdge}_s(X)$ is at most*

$$(n+1)n^n.$$

Proof. We subdivide the random walk into *phases*: a phase ends after $n+1$ vertex evaluations, or if the sink has been reached. Thus, all phases but the last consist of exactly $n+1$ vertex evaluations. We aim to show that the expected number of phases is at most n^n which implies the result.

For this, let p_i be the conditional probability that the sink is reached in phase i (equivalently, that phase i is the last phase), given that there are at least i phases. We claim that

$$p_i \geq \frac{1}{n^n}.$$

This holds, because there is a directed path of length at most n from the first vertex of phase i to the sink, and this path is selected by the algorithm with probability at least $1/n^n$. The reader familiar with Bernoulli experiments knows that this proves the expected number of phases to be at most n^n, but let's derive this explicitly.

Let P be the random variable for the number of phases. We have just shown that

$$\begin{aligned}
\text{prob}(P \geq i+1) &= \text{prob}(P \geq i+1 | P \geq i)\,\text{prob}(P \geq i) \\
&= (1-p_i)\,\text{prob}(P \geq i) \\
&\leq \left(1 - \frac{1}{n^n}\right)\text{prob}(P \geq i), \quad i \geq 1,
\end{aligned}$$

which—using $\text{prob}(P \geq 1) = 1$—implies

$$\text{prob}(P \geq i) \leq \left(1 - \frac{1}{n^n}\right)^{i-1}, \quad i \geq 1.$$

Now we get

$$\begin{aligned}
E(P) &= \sum_{i=1}^{\infty} i\,\text{prob}(P = i) \\
&= \sum_{i=1}^{\infty} i\,(\text{prob}(P \geq i) - \text{prob}(P \geq i+1)) \\
&= \sum_{i=1}^{\infty} i\,\text{prob}(P \geq i) - \sum_{i=2}^{\infty}(i-1)\,\text{prob}(P \geq i) \\
&= \sum_{i=1}^{\infty} \text{prob}(P \geq i) \\
&\leq \sum_{i=0}^{\infty} \left(1 - \frac{1}{n^n}\right)^i \\
&= n^n.
\end{aligned}$$

□

Having this upper bound on the expected number of vertex evaluations in Algorithm 5.6, we can think about better bounds. The hope is that $(n+1)n^n$ is a gross overestimate; for acyclic USOs, it obviously is, because no more than 2^n vertex evaluations are possible in this case. But even in the general case, it is not completely implausible that a bound which is *polynomial* in n might hold. After all, Randomix managed to escape a maze with n chambers in $O(\log n)$ steps, so why shouldn't the same method find the sink among 2^n vertices in $\text{poly}(n)$ time?

Unfortunately, the situation is not as nice: below, we show that for every odd value of n, there exists a (highly cyclic) USO $\mathcal{O}$ of C_n for which Algorithm 5.6 needs more than

$$\frac{(n-1)}{2}!$$

steps on average, for *every* initial vertex X distinct from the sink.

5.2.1 Morris's USO

Let $n \geq 1$ be an odd integer. Morris's USO $\mathcal{O}_n$ on C_n generalizes the cyclic USO on C_3 that we have already seen several times. In order to describe the general construction in a convenient way, we change our encoding of vertices and outmap values. Figure 5.3 shows how this works.

With every vertex $X \subseteq N = \{1, \ldots, n\}$, we associate a bitvector of length n, whose i-th entry is 1 if $i \in X$, and 0 otherwise. With every outmap value $s(X) \subseteq N$, we associate a sign vector whose i-th entry is $+$ if $i \in s(X)$, and $-$ otherwise. For a fixed USO, we can identify the vertices with the resulting bit/sign patterns.

In this view, the sink of the USO in Figure 5.3 is the vertex

0	0	0
$-$	$-$	$-$

,

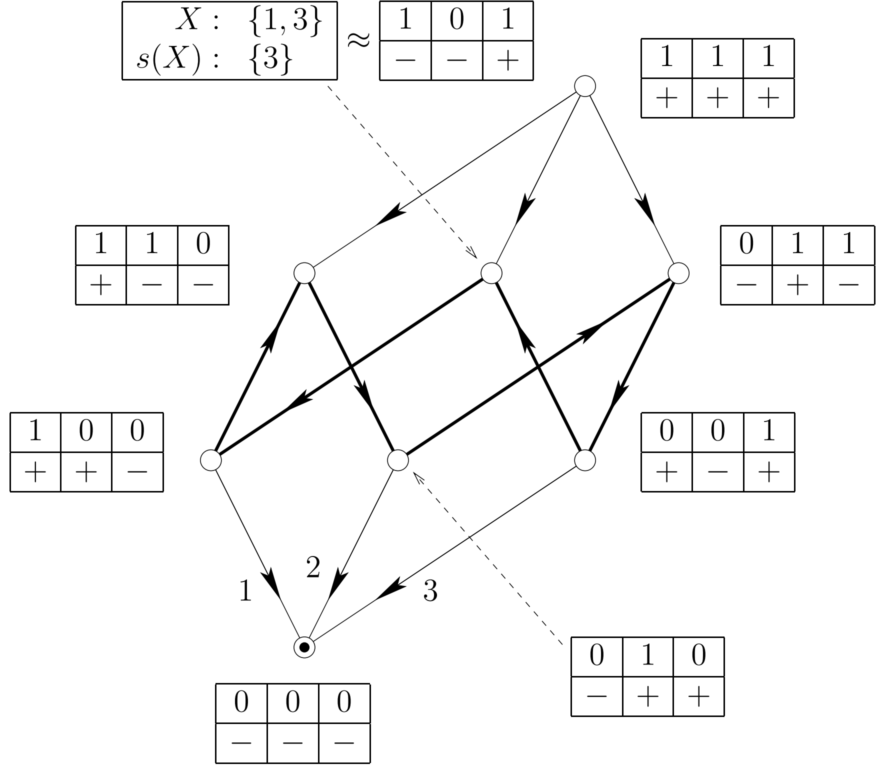


Figure 5.3: Encoding vertices with bit/sign patterns

while the source is

1	1	1
$+$	$+$	$+$

.

In order to specify $\mathcal{O}_n$, we need a rule how to obtain the sign vector corresponding to a given bit vector. If the bit vector is $(0, \ldots, 0)$, the sign vector will be $(-, \ldots, -)$. In our "old" setting, this means that the vertex $\emptyset$ is always the sink.

In any other case, the bit vector has a 1 in some position i, and we use a rule driven by a *finite automaton* to fill in the signs, starting from position $i-1$, and proceeding to the left. When we cannot go left anymore, we "wrap around" and continue with the rightmost position, until all signs are determined. Due to the wrap-around, there is a cycle symmetry in the construction: whenever some vertex (which for us is now a bit/sign pattern) appears, any cyclic shift of it appears as well. This can already be checked for $n = 3$ in Figure 5.3. Figure 5.4 explains the actual construction of $\mathcal{O}_n$, with an example for $n = 5$.

It is not clear yet that the construction determines a USO; so far, the signs just correspond to some mapping $s : 2^N \to 2^N$, but we don't know whether it is the outmap of a USO. The following is the key lemma.

Lemma 5.9 *Let $n \geq 1$ be odd and consider a partial bit/sign pattern with* either *a bit* or *a sign at any of its n positions, e.g.*

	0			1
$+$		$-$	$-$	

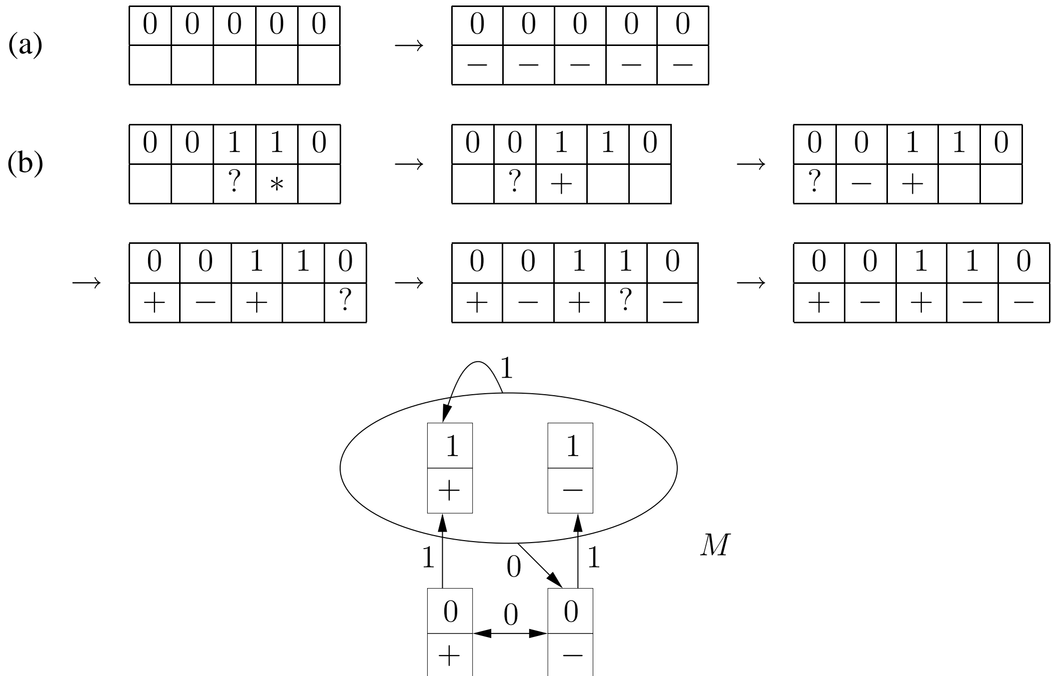


Figure 5.4: Construction of $\mathcal{O}_n$: (a) if the bit vector is the zero vector, then all signs are negative. (b) if at least one bit is 1, we proceed as follows: given the bit/sign combination at position i, the bit at position $i-1$ (modulo wrap-around) determines the sign at position $i-1$, via the finite automaton M. Any combination $(1,*)$ lets us get started, because M has the same behavior in its two states $(1,+)$ and $(1,-)$, indicated by a *superstate* that covers both. The last step of the construction finally determines the sign below the 1 we started with. The result does not depend on the 1 we started with: to the left of any 1-entry (due to wrap-around, this means everywhere), the signs are uniquely determined by M.

in the case $n = 5$. *Then there is a* unique *completion to a full pattern, such that the signs are determined by the bits according to the rules of Figure 5.4.*

Proof. Staring at the automaton M for a while, one realizes that it can also be used to determine the *bit* at position $i-1$, given the *sign*. For example, if the pattern at position i is $(1,+)$, and the sign at position $i-1$ is $-$, the corresponding bit must be 0, because the value 1 would enforce a $+$. Going through all cases reveals a beautiful symmetry: interchanging 1 with $+$ and 0 with $-$ in M yields the automaton for determining the bits. We can even merge the two automata into one which can be used to deduce the missing information at position $i-1$, regardless of whether it is a bit or a sign. Using this automaton, we can uniquely complete all partial patterns containing a 1 or a $+$ as an "anchor" for the completion.

Figure 5.5 depicts the surprisingly simple automaton M' resulting from the merger, along with a completion sequence for the example pattern of the lemma.

It remains to show that any partial pattern containing only entries 0 and $-$ can *only* be completed to the sink, meaning that *all* bits are 0 and *all* signs are $-$.

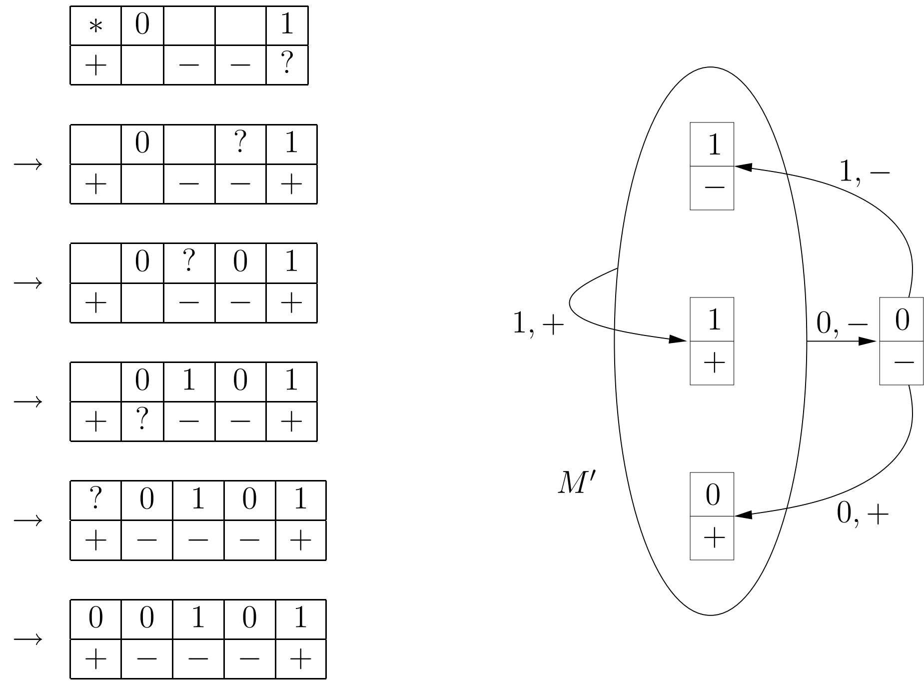


Figure 5.5: Completing a partial bit/sign pattern using automaton M'

For this, consider any other completion and assume it would be obtainable through the original automaton M, equivalently through the automaton M' in Figure 5.5. Because there is either a 0 ar a $-$ at each position, M' would have to switch between the superstate and the state $(0, -)$, for every step it proceeds to the left. This, however, is impossible if n is odd. □

Corollary 5.10 *Let $n \geq 1$ be odd. Then $\mathcal{O}_n$ is a USO.*

The proof is left as Exercise 5.3.

5.2.2 `RandomEdge` on Morris's USO

We fix some odd n and some initial vertex X of $\mathcal{O}_n$ at *level 1*; this is defined as a vertex with exactly one $(1, +)$-combination. For example, all vertices adjacent to the sink (the ones with exactly one bit of value 1) are level-1-vertices, as you can easily derive from the automaton M. In the cyclic orientation of C_3 (see Figure 5.3), level 1 consists of all the vertices on the cycle.

In order to understand the behavior of `RandomEdge` on $\mathcal{O}_n$, we need to understand how the bit/sign pattern changes when we go from X to $X \oplus \{i\}$ during the algorithm. For the bit pattern, this is easy: we just flip the bit at position i. As far as the signs are concerned, we know that X has a $+$ at position i (corresponding to $i \in s(X)$, while $X \oplus \{i\}$ has a $-$ (an easy consequence of the USO property). There are two cases.

Case 1. $i \notin X$. Then the situation is as in Figure 5.6.

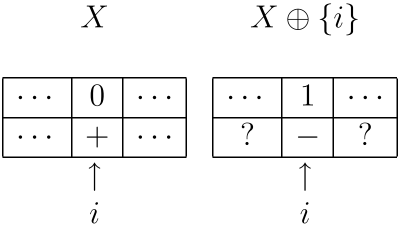


Figure 5.6: $i \notin X$

Because $(0,+)$ and $(1,-)$ are in the same superstate of automaton M' in Figure 5.5, there are no changes in signs at positions other than i. Moreover, since we have not changed the number of $(1,+)$-combinations, the new vertex $X \oplus \{i\}$ is again at level 1.

Observation 5.11 *In case 1, the step from X to $X \oplus \{i\}$ leads from a level-1 vertex to a level-1 vertex and reduces the number of plus-signs in the pattern by one.*

Case 2. $i \in X$. Here, the situation is that of Figure 5.7.

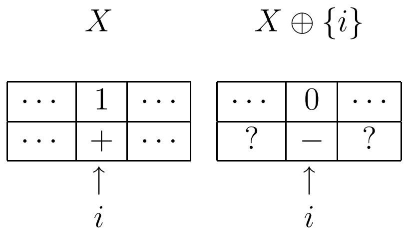


Figure 5.7: $i \in X$

There are two subcases. If X is adjacent to the sink, $X \oplus \{i\}$ is the sink, and the random walk terminates. By Exercise 5.4, this subcase occurs if and only if the number of plus-signs in X is exactly $(n+1)/2$.

Otherwise, X's pattern contains at least a second 1-bit, which must have a $-$ below it because X is at level 1. Let us consider the 1-bit closest to position i on the left. The situation is depicted in Figure 5.8.

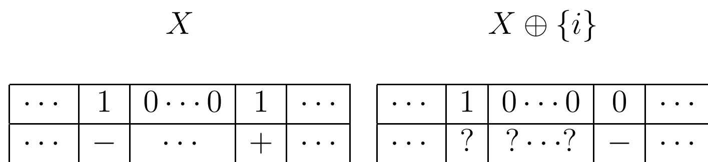


Figure 5.8: $i \in X$ and $X \neq \{i\}$

As before, because $(1,+)$ and $(1,-)$ are in the same superstate of the automaton, sign changes are restricted to the *block* delimited by the two 1-bits. Within the block, *all* signs

change, as you can immediately derive from the automaton. It follows that $X \oplus \{i\}$ is again at level 1: the $(1, +)$-combination has simply moved to the left.

The automaton tells us even more: the signs below the 0's in the block alternate, with the first and last one having negative sign in X. In particular, there is an odd number $2r + 1$ of 0's in the block, r of which have a plus-sign in X. Because the step from X to $X \oplus \{i\}$ reverses all signs in the block, we get

Observation 5.12 *In case 2, the step from X to $X \oplus \{i\}$ leads from a level-1 vertex either to the sink (iff the number of plus-signs in X is $(n + 1)/2$), or it leads to a level-1 vertex and increases the number of plus-signs in the pattern by one.*

Observations 5.11 and 5.12 show that the behavior of `RandomEdge` at level 1 only depends on the number of plus-signs in the current vertex X, not on the vertex itself. Namely, if i is the number of plus-signs in X, exactly one of them has a 1 above it; therefore, case 1 occurs with probability $(i - 1)/i$ and leads to a level-1 vertex with $i - 1$ plus-signs. Case 2 has probability $1/i$ and leads to the sink, or to a level-1 vertex with $i + 1$ plus-signs.

Fact 5.13 *For $i \in \{1, \ldots, (n + 1)/2\}$, let E_i be the expected number of vertex evaluations in* `RandomEdge`*, starting from a level-one vertex X of $\mathcal{O}_n$ with exactly i plus-signs. Moreover, let $E_{(n+3)/2}$ be the expected number of evaluations, starting from the sink. Then we have*

$$E_{(n+3)/2} = 1, \tag{5.3}$$

$$E_i = 1 + \frac{1}{i}E_{i+1} + \frac{i-1}{i}E_{i-1}, \quad 1 \leq i \leq \frac{n+1}{2}. \tag{5.4}$$

This is like the casino walk, except that we have just one absorbing value $(n + 3)/2$, and that the transition probabilities depend on the current value. Still, the idea is, as in the casino walk, to define a quantity

$$b_i = p(i)E_i - q(i)E_{i-1} + t(i),$$

where $p(i), q(i), t(i)$ are functions depending on i that ensure

$$b_i = b_{i-1}. \tag{5.5}$$

In the casino walk we had $p(i) = q(i) = 1, t(i) = 2i$. From (5.5), we can derive some conditions on the functions we may use here. We want that

$$\begin{aligned} b_i - b_{i-1} &= p(i)E_i - (q(i) + p(i-1))E_{i-1} + q(i-1)E_{i-2} + t(i) - t(i-1) \\ &= c(i)\left(\frac{1}{i-1}E_i - E_{i-1} + \frac{i-2}{i-1}E_{i-2} + 1\right) \\ &= 0, \end{aligned}$$

for some function $c(i)$.

This implies the conditions

$$\begin{aligned} q(i-1) &= p(i)(i-2), \\ q(i) + p(i-1) &= p(i)(i-1). \end{aligned}$$

These conditions smell like the functions might involve some factorials, and after some trial-and-error steps, one finally realizes that choosing $p(i) = q(i) = 1/(i-2)!$ satisfies both conditions, and for this choice, we must set $c(i) = (i-1)/(i-2)!$. It remains to find suitable values $t(i)$. With $t(i) - t(i-1) = c(i)$, we must have

$$\begin{aligned} t(i) &= t(i-1) + \frac{i-1}{(i-2)!} \\ &= t(i-1) + \frac{1}{(i-2)!} + \frac{1}{(i-3)!}. \end{aligned}$$

Choosing

$$t(i) := \frac{1}{(i-2)!} + 2\sum_{j=0}^{i-3} \frac{1}{j!},$$

the previous equation holds.

In these rough computations, we have ignored all boundary conditions, so let us check that we have made the right choices. We define

$$b_i := \frac{1}{(i-2)!}\left(E_i - E_{i-1} + \underbrace{T(i-2)}_{t(i)(i-2)!}\right), \quad 2 \le i \le \frac{n+3}{2}, \tag{5.6}$$

where

$$T(k) := t(k+2)k! = 1 + 2k!\sum_{j=0}^{k-1}\frac{1}{j!} = 2k!\sum_{j=0}^{k}\frac{1}{j!} - 1, \quad k \ge 0.$$

It is easy to verify that

$$T(k)(k+1) = T(k+1) - (k+2). \tag{5.7}$$

Lemma 5.14 $b_i = 0$ *for* $2 \le i \le (n+3)/2$.

Proof. By induction. From (5.4) we get

$$b_2 = E_2 - E_1 + 1 = 0.$$

Now assume $i > 2$. Inductively, we get

$$\begin{aligned} b_i &= b_i - b_{i-1} \\ &= \frac{1}{(i-2)!}E_i - \left(\frac{1}{(i-2)!} + \frac{1}{(i-3)!}\right)E_{i-1} + \frac{1}{(i-3)!}E_{i-2} + \frac{T(i-2)}{(i-2)!} - \frac{T(i-3)}{(i-3)!} \\ &= \frac{1}{(i-2)!}E_i - \frac{i-1}{(i-2)!}E_{i-1} + \frac{i-2}{(i-2)!}E_{i-2} + \frac{T(i-2) - (i-2)T(i-3)}{(i-2)!}. \end{aligned}$$

This further yields

$$\begin{aligned} \frac{(i-2)!}{i-1}b_i &= \frac{1}{i-1}E_i - E_{i-1} + \frac{i-2}{i-1}E_{i-2} + \frac{T(i-2) - (i-2)T(i-3)}{i-1} \\ &\overset{(5.7)}{=} \frac{1}{i-1}E_i - E_{i-1} + \frac{i-2}{i-1}E_{i-2} + 1 \\ &\overset{(5.4)}{=} 0. \end{aligned}$$

$\square$

An immediate consequence of $b_{(n+3)/2} = 0$ is

$$E_{(n+1)/2} = T\left(\frac{(n-1)}{2}\right) + 1,$$

but we can also find all other values of E_i. From (5.6) and the lemma, we get

$$E_i = E_1 - \sum_{k=0}^{i-2} T(k), \quad 2 \leq i \leq \frac{n+3}{2}.$$

Using this for $i = (n+3)/2$, the boundary condition (5.3) gives

$$E_1 = 1 + \sum_{k=0}^{(n-1)/2} T(k),$$

which implies the following result.

Theorem 5.15

$$E_i = 1 + \sum_{k=i-1}^{(n-1)/2} T(k), \quad 1 \leq i \leq \frac{n+3}{2}.$$

Asymptotics. The careful reader might remember that in connection with smallest enclosing balls, we have already seen how to evaluate $T(k)$, see Corollary 2.14. Here we get

$$T(k) = 2\lfloor ek! \rfloor - 1, \quad k > 0,$$

where e is the Euler constant. This means, $T((n-1)/2)$ is the dominant term in the right-hand side of the theorem, which implies that

$$E_i = 2e\frac{n-1}{2}! + o\left(\frac{n-1}{2}!\right), \quad 1 \leq i \leq \frac{n+1}{2}.$$

In particular, we get the nice formula

$$E_{(n+1)/2} = 2\lfloor e\frac{n-1}{2}! \rfloor, n \geq 3$$

for the starting value just left of the absorbing value $(n+3)/2$.

By definition of E_i, this corresponds to the expected number of vertex evaluations of `RandomEdge` on Morris's USO $\mathcal{O}_n$, starting from a vertex adjacent to the sink. Because such a vertex must eventually be passed for *any* initial vertex distinct from the sink, the main result of this section follows. It shows that on Morris's USO, `RandomEdge` visits significantly more vertices than there are distinct vertices. In other words, it heavily runs in cycles.

Theorem 5.16 *Starting from any vertex X of $\mathcal{O}_n$ which is not the sink,* `RandomEdge` *takes an expected number of at least*

$$2\lfloor e\frac{n-1}{2}!\rfloor$$

vertex evaluations to reach the sink. For any vertex adjacent to the sink, the bound is exact.

Let us briefly check whether this matches the results we have obtained for Exercise 3.1: in Figure 5.9, all vertices of $\mathcal{O}_3$ are labeled with the expected number of vertex evaluations needed by `RandomEdge`, using the vertex as initial vertex.

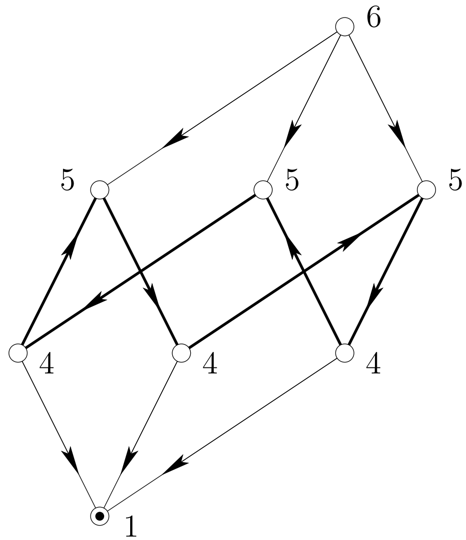


Figure 5.9: `RandomEdge` runtimes for Morris's USO on C_3

Indeed, for the three vertices adjacent to the sink, the number is $4 = 2\lfloor e\rfloor$. Theorem 5.15 also yields the 5's in the figure, because the three corresponding vertices are at level 1 as well.

Does this mean that `RandomEdge` is a bad algorithm? Well, not necessarily: there is the important special case of *acyclic* USOs, and no acyclic USO on the n-cube is known for which `RandomEdge` requires more than roughly n^2 steps. In other words, `RandomEdge` could still be a good (polynomial-time) algorithm for acyclic USOs. Exercise 5.5 asks you to analyze `RandomEdge` on a particular acyclic USO.

We will come back to acyclic USOs in the last chapter, where we collect evidence that the acyclic case might indeed be substantially easier than the general one.

5.3 Random Unique Sink Orientations

Suppose you want to generate a USO of C_n uniformly at random, i.e. you want to have a program which outputs any USO with the same probability $1/u_n$, where u_n is the number of USOs of C_n.

There are two obvious, but inefficient approaches. First, you could generate *all* USOs and then select a random one. For very small n, one can indeed find a complete list of all USOs. The known numbers are

$$\begin{aligned} u_0 &= 1, \\ u_1 &= 2, \\ u_2 &= 12, \\ u_3 &= 744, \\ u_4 &= 5541744. \end{aligned}$$

The statement of Exercise 5.6 can be used to find that

$$\begin{aligned} u_5 &> 6.1 \cdot 10^{13}, \text{ and} \\ u_6 &> 7.5 \cdot 10^{27}, \end{aligned}$$

so the 5-cube has already a number of USOs which cannot be generated in the form of a complete list, using reasonable space and time.

A second approach is to generate a completely random cube orientation, by choosing the orientation of each edge independently at random. If the selected orientation is a USO, output it, otherwise repeat. The problem here is that the probability of obtaining a USO within reasonable time is very small. Already for $n = 4$, only 5541744 out of the 2^{32} orientations are USOs, which is a fraction of only $1/1000$, roughly. This fraction rapidly decreases with the dimension: for $n = 5$, the fraction is around $5 \cdot 10^{-10}$. How did we get this number, not knowing u_5? Just read on...

A third approach is to perform a *random walk* on the set of all USOs: start with an arbitrary USO and go in each step to a random USO in the neighborhood of the current USO. The hope is that after only a few steps, we have reached an almost random USO. This is the idea behind the provider walk of Section 5.1, but things are not that simple here. Two questions need to be addressed.

1. What is the neighborhood of a USO?

2. Even if we have defined neighborhoods, how can we prove that the random walk converges to the uniform distribution?

It turns out that for question 2, the general theory of *Markov chains* will provide an answer, so let's start with this.

5.3.1 Markov Chains

Markov chains are formal models for random walks. We have a *state space* Q (for example, the set of all USOs of C_n) which we assume to be finite.[2] Then we can associate Q with the set of integers $\{1, \ldots, |Q|\}$—this will simplify the further notations.

Furthermore, there is a sequence $(X_t)_{t \in \mathbb{N}}$ of random variables with values in Q. The value of X_t is the state in which the random walk resides after t steps.

[2]General Markov chains do not necessarily have finite state spaces, but we don't need this here.

Definition 5.17 *The pair* $\mathcal{C} = (Q, (X_t)_{t \in \mathbb{N}})$ *is called a* Markov chain*, if for all* $t \in \mathbb{N}$ *and all sequences* $(i_0, \ldots, i_t) \in Q^t$*,*

$$\text{prob}(X_t = i_t | X_0 = i_0, \ldots, X_{t-1} = i_{t-1}) = \text{prob}(X_t = i_t | X_{t-1} = i_{t-1}).$$

This fancy-looking definition simply says that the probability of reaching some state in step t only depends on the state after $t-1$ steps, but not on states the walk went through earlier. In all the random walks, we have seen so far, this was the case; for example, the amount of money you have after t steps of the casino walk only depends on the money after $t-1$ steps (and the outcome of your current bet, of course). It doesn't matter whether you went through a run of good luck before, or whether you almost went broke. We also say that $\mathcal{C}$ "has no memory".

Here is another property shared by all our walks so far: the transition probabilities between states are time-independent:

Definition 5.18

(i) A Markov chain $\mathcal{C} = (Q, (X_t)_{t \in \mathbb{N}})$ *is called* homogeneous*, if for all* $t \in \mathbb{N}, i, j \in Q$*,*

$$\text{prob}(X_t = j | X_{t-1} = i) = \text{prob}(X_1 = j | X_0 = i).$$

(ii) For a homogeneous Markov chain $\mathcal{C}$*, the matrix* $P \in \mathbb{R}^{|Q| \times |Q|}$*, with entries*

$$p_{ij} := \text{prob}(X_1 = j | X_0 = i)$$

is called the transition matrix *of* $\mathcal{C}$*.*

(iii) For a homogeneous Markov chain $\mathcal{C}$*, the directed graph* $G = (Q, E)$*, with*

$$E = \{(i, j) \mid p_{ij} > 0\}$$

is called the graph *of* $\mathcal{C}$*.*

In the homogeneous case, the transition matrix completely specifies the random walk: whenever the walk is in state i, its next state is j with probability p_{ij}, no matter how many steps have already been performed.

Let's do an example. The provider walk of Section 5.1 can be written as a homogeneous Markov chain; its state space can be identified with $Q = \{1, 2, 3\}$, where $1 = \text{Swisscom}, 2 = \text{Orange}, 3 = \text{Sunrise}$. The transition matrix is

$$P = \begin{pmatrix} 0 & \frac{1}{2} & \frac{1}{2} \\ \frac{1}{2} & 0 & \frac{1}{2} \\ \frac{1}{2} & \frac{1}{2} & 0 \end{pmatrix}, \tag{5.8}$$

and the graph is the one of Figure 5.2.

Recall that in the provider walk, we were interested in the probability of being in state i after t steps, for all i. This motivates the next

Definition 5.19 *For a Markov chain $\mathcal{C} = (Q, (X_t)_{t \in \mathbb{N}})$, the row vector*

$$q_t := (\text{prob}(X_t = 1), \ldots, \text{prob}(X_t = |Q|))$$

is called the distribution *of $\mathcal{C}$ at time t.*

Usually, q_0 is known (we had $q_0 = (1, 0, 0)$ in the provider walk), and we want to derive statements about $q_t, t > 0$. The following lemma (whose easy proof is Exercise 5.7) comes in handy.

Lemma 5.20 *Let $\mathcal{C}$ be a homogeneous Markov chain with transition matrix P. Then*

$$q_t = q_0 P^t.$$

Assume we don't start the provider walk with $q_0 = (1, 0, 0)$, but with $q_0 = (1/3, 1/3, 1/3)$, meaning that right in the beginning, you are customer of a uniformly random provider. Then it is easy to check that you will *always* be customer of a uniformly random provider: q_0 is a *stationary distribution* according to the following

Definition 5.21 *Let $\mathcal{C}$ be a homogeneous Markov chain with transition matrix P and $\pi = (\pi_1, \ldots, \pi_{|Q|})$ a row vector such that*

$$\sum_{i=1}^{|Q|} \pi_i = 1, \quad \pi_i \geq 0, \forall i.$$

π is called a stationary distribution *of $\mathcal{C}$, if*

$$\pi P = \pi.$$

Recall from the beginning of this section that we actually want to generate random USOs. Assuming we can manufacture a Markov chain for this in the spirit of the one for "generating" a random mobile phone provider, we still need to prove that this chain has a stationary distribution π which is the uniform one, and that the distribution q_t converges to π as t tends to infinity.

In the provider walk, we were able to do this "manually", but for less trivial chains, this approach quickly becomes unmanageable. However, the theory of Markov chains lists some easy sufficient conditions under which we get the properties we want. Once we have this, the USO case can be dealt with by *designing* a Markov chain that just satisfies these conditions.

Definition 5.22 *A homogeneous Markov chain $\mathcal{C}$ is called* irreducible*, if there is a directed path from any state to any state in the graph of $\mathcal{C}$ (in particular, there must be a directed path from any state to itself). Equivalently, for any pair $(i, j) \in Q^2$, there is a positive probability of eventually reaching j, starting from i.*

Obviously, the provider walk is irreducible: no matter who your current provider is, you can become customer of all three providers in the future. Clearly, if a chain is not irreducible, it cannot be used to generate a random state, because some states might just not be reachable from the initial state.

Definition 5.23 *A state i of a homogeneous Markov chain $\mathcal{C}$ is called k-*periodic *for $k \in \mathbb{N}$, if all directed paths from i back to i in the graph of $\mathcal{C}$ have length $k\ell$ for some $\ell \in \mathbb{N}$. $\mathcal{C}$ is called* aperiodic*, if no state is k-periodic with $k \geq 2$.*

According to this definition, all states are 1-periodic, but higher periods must be avoided by an aperiodic chain. For example, a 2-periodic state i has only directed paths of even length back to itself. When we start the walk in such a state i, we can not expect to converge to a random state, because after any odd number of steps, the probability of being in i is zero.

The provider walk is aperiodic, because any state has directed paths back to itself of all lengths but 1.

Definition 5.24 *A homogeneous Markov chain $\mathcal{C}$ with transition matrix P is called* symmetric *if P is symmetric, meaning that $p_{ij} = p_{ji}$, for all $i, j \in Q$.*

Again, the provider walk satisfies this, because the matrix (5.8) is symmetric. According to the following general results, the provider walk therefore converges—as we know—to a uniformly random provider.

Theorem 5.25 *Let $\mathcal{C}$ be a homogeneous Markov chain. Then the following three statements hold.*

(i) If $\mathcal{C}$ is irreducible, then $\mathcal{C}$ has a unique stationary distribution π.

(ii) If $\mathcal{C}$ is irreducible and aperiodic (this is also known as ergodic*), then*

$$\lim_{t \to \infty} q_t = \pi,$$

where π is the unique stationary distribution.

(iii) If $\mathcal{C}$ is irreducible and symmetric, then the unique stationary distribution π is the uniform one, i.e.

$$\pi = \left(\frac{1}{|Q|}, \ldots, \frac{1}{|Q|} \right).$$

Proofs of these statements are found in most advanced textbooks on probability theory; we only prove here that (iii) follows from (i), because this is easy and instructive.

If $\mathcal{C}$ is symmetric, its transition matrix P is *doubly stochastic*, meaning that all rows and all columns sum up to one. In general, this only holds for the rows, but if it holds for the columns as well, we immediately get

$$\left(\frac{1}{|Q|}, \ldots, \frac{1}{|Q|} \right) P = \left(\frac{1}{|Q|}, \ldots, \frac{1}{|Q|} \right),$$

so the uniform distribution is stationary. By irreducibility, it is the unique stationary distribution.

5.3.2 A Markov Chain for USOs

Now we want to apply the above machinery to a suitable Markov chain over the set of USOs. For this, we first define what it means that two USOs are neighbors of each other. Then we assign to every USO a probability distribution over its set of neighbors. Viewing them as transition probabilities in a random walk, this defines a homogeneous Markov chain on the set of all USOs.

In the following, let us fix the dimension n of the cube and its ground set $N = \{1, \dots, n\}$.

Definition 5.26 *Let $\mathcal{O}, \mathcal{O}'$ be two USOs of C_n, $i \in N$. $\mathcal{O}$ and $\mathcal{O}'$ are called* neighbors in direction i *if and only if $\mathcal{O}$ agrees with $\mathcal{O}'$ in the orientation of all edges, except possibly the ones in direction i. Formally, $\mathcal{O} = (2^N, D)$ and $\mathcal{O}' = (2^N, D')$ are neighbors in direction i, if*

$$(X, X \oplus \{j\}) \in D \quad \Leftrightarrow \quad (X, X \oplus \{j\}) \in D', \quad \forall X \in 2^N, j \in N \setminus \{i\}.$$

According to this definition, every USO is a neighbor of itself in any direction. Figure 5.10 shows a nontrivial example of the neighborhood relation.

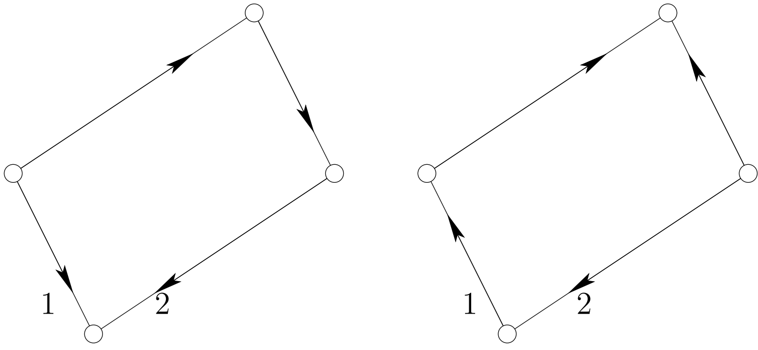


Figure 5.10: Two USOs which are neighbors in direction 1

The Markov chain. The state space Q is the set of all USOs of C_n. The chain will be homogeneous and is therefore completely defined by its transition matrix, containing the transition probabilities for all pairs of USOs. For a USO $\mathcal{O}$, let $\mathcal{N}(\mathcal{O}, i)$ be the neighbors of $\mathcal{O}$ in direction i.

The transition probability of going from $\mathcal{O}$ to $\mathcal{O}'$ is defined as

$$p_{\mathcal{O},\mathcal{O}'} = \frac{1}{n} \sum_{i:\mathcal{O}' \in \mathcal{N}(\mathcal{O},i)} \frac{1}{|\mathcal{N}(\mathcal{O}, i)|}. \tag{5.9}$$

These values are actually legal transition probabilities.

Fact 5.27 *For any $\mathcal{O}$,*

$$\sum_{\mathcal{O}'} p_{\mathcal{O},\mathcal{O}'} = 1.$$

Proof. Letting $[A]$ be the indicator variable for the statement in brackets ($[A] = 1$ if A holds, $[A] = 0$ otherwise), we can write

$$\begin{aligned}\sum_{\mathcal{O}'} p_{\mathcal{O},\mathcal{O}'} &= \sum_{\mathcal{O}'} \frac{1}{n} \sum_{i:\mathcal{O}' \in \mathcal{N}(\mathcal{O},i)} \frac{1}{|\mathcal{N}(\mathcal{O},i)|} \\ &= \sum_{\mathcal{O}'} \frac{1}{n} \sum_{i \in N} [\mathcal{O}' \in \mathcal{N}(\mathcal{O},i)] \frac{1}{|\mathcal{N}(\mathcal{O},i)|} \\ &= \frac{1}{n} \sum_{i \in N} \sum_{\mathcal{O}'} [\mathcal{O}' \in \mathcal{N}(\mathcal{O},i)] \frac{1}{|\mathcal{N}(\mathcal{O},i)|} \\ &= \frac{1}{n} \sum_{i \in N} \frac{|\mathcal{N}(\mathcal{O},i)|}{|\mathcal{N}(\mathcal{O},i)|} = 1.\end{aligned}$$

Note that the sum in the definition of $p_{\mathcal{O},\mathcal{O}'}$ consists either of one term (if $\mathcal{O} \neq \mathcal{O}'$), or of n terms (if $\mathcal{O} = \mathcal{O}'$).

Figure 5.11 shows an example. $\mathcal{O}$ has four neighbors in direction 1, and two neighbors in direction 2, always including itself. The probability of going to a fixed neighbor in direction 1 is therefore $1/2 \cdot 1/4 = 1/8$, while a neighbor in direction 2 is reached with probability $1/2 \cdot 1/2 = 1/4$. For the transition from $\mathcal{O}$ to itself, this sums up to $1/4 + 1/8 = 3/8$.

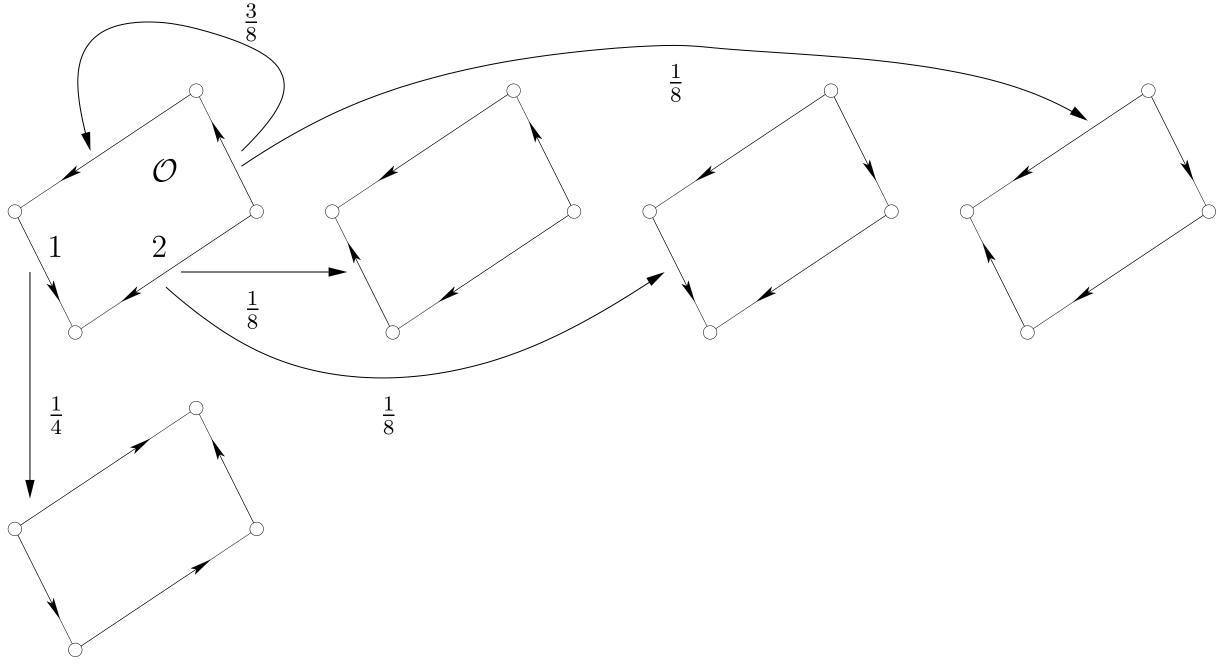


Figure 5.11: Transition probabilities for a fixed bow $\mathcal{O}'$

Theorem 5.28 *The homogeneous Markov chain defined by the transition probabilities in (5.9) is aperiodic, symmetric, and irreducible (this is the order in which we are going to show the properties).*

Proof. Because every USO is a neighbor of itself, it has a directed path of length 1 back to itself. This implies that the chain is aperiodic.[3]

Symmetry follows directly from the symmetry of the neighbor relation, plus the fact that $\mathcal{N}(\mathcal{O}, i) = \mathcal{N}(\mathcal{O}', i)$, if $\mathcal{O}$ and $\mathcal{O}'$ are neighbors in direction i. The latter holds, because the neighborhood relation in direction i (Definition 5.26) is not only symmetric, but also reflexive and transitive, so it is an *equivalence relation*, where the sets $\mathcal{N}(\mathcal{O}, i)$ are the equivalence classes. This means, if $\mathcal{O}, \mathcal{O}'$ are neighbors in direction i, they are equivalent and define the same equivalence class $\mathcal{N}(\mathcal{O}, i) = \mathcal{N}(\mathcal{O}', i)$.

Finally, we need to show that the chain is irreducible, meaning that we can get from any USO to any USO by going to a neighbor in each step. Actually, we show that in at most n steps, we can reach the *uniform* USO (defined by the outmap values $s(X) = X$, for all X); by symmetry, we can go then from the uniform USO back to any USO in another n steps at most.

Starting from any USO, step i reorients only edges in direction i, in such a way that we again get a USO. Then it is clear that this USO is a neighbor of the previous one. To be more precise, step i orients the edges in direction i in such a way that they all go from the larger to the smaller set. After this, the orientation contains the directed edges

$$(X \cup \{i\}, X), \quad X \subseteq 2^{N \setminus \{i\}}. \tag{5.10}$$

We say that the orientation is *combed* in direction i; in this case, it's combed from *top to bottom* (which is natural when you think about combing your hair), but an orientation can also be combed from *bottom to top*, meaning that all edges in direction i go from the smaller to the larger sets.

It is easy to verify (this is actually part of Exercise 5.6) that the process of combing a USO in some direction maintains the USO property.

It follows that after at most n steps, we have a USO which satisfies (5.10) for all i, i.e. it is combed from top to bottom in all directions. This implies $s(X) = X$, so we have the uniform USO. □

Together with Theorem 5.25, this theorem implies that the random walk underlying the Markov chain we have defined indeed converges to the uniform distribution on the set of all USOs.

The distributions q_t can still be explicitly computed within reasonable time for $n = 3$. The *Maple* plot in Figure 5.12 shows how the largest possible entry in q_t (upper graph) and the smallest possible entry (lower graph) develop with t.[4] After around 30 steps, both graphs are already very close to the middle graph which represents the value $1/u_3 = 1/744$.

When you think about how you actually perform the random walk, there is still one issue. It seems we would need the neighbors of the current USO in all directions in order to choose the next one among them, according to the probabilities in (5.9). We could of course, for all i, go through the 2^{n-1} candidates for neighbors in direction i and collect the USOs among them, but that would be pretty inefficient. As we show next, all the neighbors of a given USO are implicitly encoded in the USO itself, and we can make this information explicit. This will also

[3]actually, this is a common trick to make a chain aperiodic: just add a loop from any state to itself, "stealing" the probability of looping from the other transitions

[4]the maxima and minima are taken over all initial distributions q_0

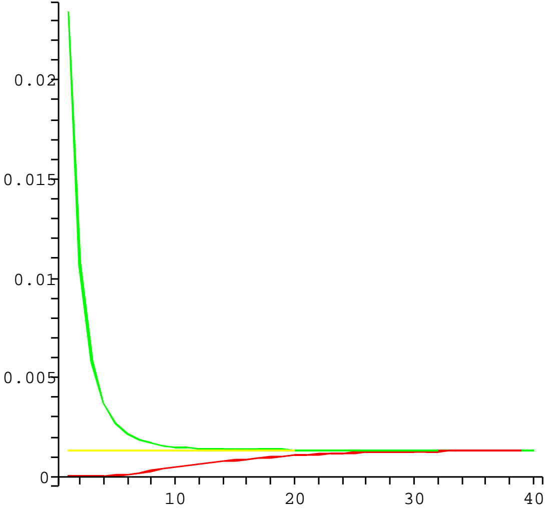


Figure 5.12: Convergence of the USO chain for $n = 3$

show that the number of neighbors in a given direction is always a power of two; in the example of Figure 5.11, we have already come across an incarnation of this fact.

Phases. Consider Figure 5.11 again; why does $\mathcal{O}$ have only two neighbors (including itself) in direction 2? Because the two edges in direction 2 are "in phase": given the orientations of the edges in direction 1, we only get a USO if the two other edges have the same orientation, both from top to bottom (as in $\mathcal{O}$), or from bottom to top , as in its other neighbor. The reason is that there is a pair of vertices X, Y in $\mathcal{O}$ (take any pair of antipodal vertices) such that

$$(X \oplus Y) \cap (s_{\mathcal{O}}(X) \oplus s_{\mathcal{O}}(Y)) = \{2\},$$

and if we reorient just *one* of the two edges incident to X and Y in direction 2, the resulting orientation $\mathcal{O}'$ satisfies

$$(X \oplus Y) \cap (s_{\mathcal{O}'}(X) \oplus s_{\mathcal{O}'}(Y)) = \emptyset.$$

By our outmap characterization (Lemma 3.5), $\mathcal{O}'$ cannot be a USO in this case. This motivates the following

Definition 5.29 *Let $\mathcal{O}$ be a USO over the groundset N, $X, Y \in 2^N, i \in N$. X and Y are called* strongly in phase *with respect to $(\mathcal{O}, i)$ if*

$$(X \oplus Y) \cap (s_{\mathcal{O}}(X) \oplus s_{\mathcal{O}}(Y)) = \{i\}.$$

Remark 5.30 *X and $X \oplus \{i\}$ are strongly in phase w.r.t. $(\mathcal{O}, i)$, for all X.*

For any i, the cube C_n subdivides into a *top facet* (vertices containing i) and a *bottom facet* (vertices not containing i). If X and Y are strongly in phase w.r.t. $(\mathcal{O}, i)$, we have $i \in X \oplus Y$, so X and Y are in different facets. Then $i \in s_{\mathcal{O}}(X) \oplus s_{\mathcal{O}}(Y)$ implies that the two edges incident to X and Y in direction i must both go from top to bottom, or from bottom to top.

The strongly-in-phase relation is symmetric, but it is neither reflexive nor transitive; taking the *transitive closure* gives us a more useful equivalence relation.

Definition 5.31 *Let $\mathcal{O}$ be a USO over N, $X, Y \in 2^N, i \in N$. X and Y are called* in phase *w.r.t. i, written as*

$$X \simeq_{\mathcal{O},i} Y,$$

if there is a sequence of vertices $X = X_0, X_1, \ldots, X_k = Y$ such that X_t and X_{t+1} are strongly in phase w.r.t. $(\mathcal{O}, i)$, for $t = 0, \ldots, k-1$.

It is clear by construction that $\simeq_{\mathcal{O},i}$ is transitive, and it is also reflexive as a consequence of Remark 5.30.

We call the equivalence classes of $\simeq_{\mathcal{O},i}$ the *i-phases* of $\mathcal{O}$. Figure 5.13 shows an example for $n = 3$. Together with the strongly-in-phase pairs of Remark 5.30, the two indicated pairs already prove that the six rightmost vertices are in one common 2-phase. It remains to check that the two leftmost ones are in a different 2-phase.

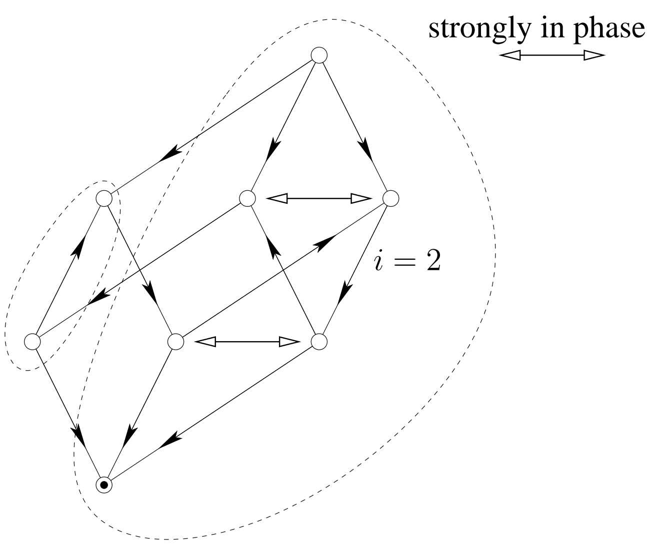


Figure 5.13: The two 2-phases of the cyclic USO

From the considerations above, it follows that *all* edges in direction i between the vertices of a fixed i-phase have the same orientation. For the 2-phase of size 6 in Figure 5.13, there are three such edges, all of them going from top to bottom. Extending previous terminology in the natural way, we say that the i-phases are combed.

Here is the crucial insight:

Theorem 5.32 *If $\mathcal{O}$ and $\mathcal{O}'$ are neighbors in direction i in the sense of Definition 5.26, then $\simeq_{\mathcal{O},i} = \simeq_{\mathcal{O}',i}$, so both USOs have the same i-phases.*

Proof. If $\mathcal{O}$ and $\mathcal{O}'$ are neighbors, they differ only in orientations of edges in direction i. Therefore,

$$(X \oplus Y) \cap (s_{\mathcal{O}}(X) \oplus s_{\mathcal{O}}(Y)) \tag{5.11}$$

can differ from

$$(X \oplus Y) \cap (s_{\mathcal{O}'}(X) \oplus s_{\mathcal{O}'}(Y)) \tag{5.12}$$

at most in the element i. If (5.11) evaluates to $\{i\}$, then (5.12) may evaluate to $\emptyset$, or to $\{i\}$. Because $\mathcal{O}'$ is a USO, the former is not possible, and it follows that X and Y are strongly in phase w.r.t. $(\mathcal{O}, i)$ if and only if they are strongly in phase w.r.t. $(\mathcal{O}', i)$. □

Now we are in a position to describe how the neighbors of $\mathcal{O}$ in direction i can be read off the i-phases of $\mathcal{O}$.

Assume $\mathcal{O}$ has ℓ i-phases. The theorem implies that the number $|\mathcal{N}(\mathcal{O}, i)|$ of neighbors of $\mathcal{O}$ in direction i is at most 2^ℓ. Namely, all neighbors have the same i-phases, and each i-phase can independently of the others be combed in two ways. It also follows that all neighbors of $\mathcal{O}$ can be obtained by reorienting some i-phases. Actually, *all* possible 2^ℓ reorientations of i-phases lead to USOs again (Exercise 5.9), thus

$$|\mathcal{N}(\mathcal{O}, i)| = 2^\ell$$

holds.

Now it is an easy task to choose a random neighbor in direction i, as the transition probabilities (5.9) of our Markov chain require: simply reorient a random subset of the i-phases; equivalently, toss a coin for each i-phase to decide whether this i-phase will be reoriented or not.

5.4 Counting USOs with the Markov Chain

It is often the case that if one can select an element from some state space uniformly at random, then one can also approximately count the number of states. This is particularly useful for state spaces which are too large to be explicitly enumerated. In this section, we see how the Markov chain we have introduced in the previous section can be used to find a reasonably good estimate for the number u_5 of USOs on C_5. Recall that our lower bound of

$$u_5 > 6.1 \cdot 10^{13}$$

already shows that the state space *is* actually too large for an explicit enumeration.

Here is the idea: let $\bar{u}_n$ be the number of USOs of C_n which are combed in a fixed direction, say n. We count a USO for $\bar{u}_n$ regardless of whether it is combed from top to bottom, or from bottom to top. For those who have solved Exercise 5.6, the following does not come as a surprise.

Observation 5.33 $\bar{u}_n = 2(u_{n-1})^2$.

Proof. To obtain a combed USO of C_n, we can combine any USO in the top facet (vertices containing n) with any USO in the bottom facet (vertices not containing n): if all edges between upper and lower facet have the same orientation, the result will be a USO of C_n, combed in direction n. Because upper and lower facets are C_{n-1}'s each, and there are two ways to comb an orientation in direction n, the bound follows. □

The following is actually more a definition than anything else.

Corollary 5.34 *Let $\bar{p}_n$ be the probability that a USO chosen uniformly at random from all USOs of C_n is combed in direction n. Then*

$$\bar{p}_n = \frac{\bar{u}_n}{u_n} = 2\frac{(u_{n-1})^2}{u_n},$$

equivalently

$$u_n = 2\frac{(u_{n-1})^2}{\bar{p}_n}.$$

In the concrete case $n = 5$ we are interested in, u_{n-1} is a known quantity (namely, $u_4 = 5541744$), and $\bar{p}_n$ can be estimated experimentally. For this, we simply use the Markov chain to sample a large number t of (almost) random USOs, and use the fraction $\bar{p}(t)$ of combed ones (in direction n) as an estimate for $\bar{p}_n$. The *law of large numbers* tells us that this is justified: we have

$$\text{“}\lim_{t\to\infty} \bar{p}(t) = \bar{p}_n.\text{”}$$

Despite being quite intuitive, this statement is mathematically incorrect, because $(p(t))_{t\in N}$ is not a sequence of numbers but a sequence of random variables. The correct statement is

$$\lim_{t\to\infty} E\left((\bar{p}(t) - \bar{p}_n)^2\right) = 0,$$

but the interpretation is the same: if we wait long enough, the experimental value $\bar{p}(t)$ will be close to $\bar{p}_n$ with high probability.

But how long do we have to wait? If the Markov chain were perfect in the sense that it generates truly random USOs, we could use the *central limit theorem* and find bounds for t that guarantee a small error. However, we only know that the Markov chain is perfect *in the limit*, but we have no clue how “random” the USO is after 100 steps, say. Figure 5.12 looks encouraging in dimension $n = 3$, but it is conceivable that the convergence rate (also known as the *mixing rate*) of the chain gets much worse as n grows.

The fact that the graph of the chain has small diameter (in our case, any state is reachable from any state in at most $2n$ steps) does not guarantee a good mixing rate: the graph might have *bottlenecks* that are passed by the random walk only with very small probability. No results concerning the mixing rate of the USO chain are known.

In practice, we just let the experiment run until $\bar{p}(t)$ stabilizes to our satisfaction. For example, after letting the walk for the 5-dimensional USOs run for two days, taking every 100th USO into account for $\bar{p}(t)$, one finds that

$$\bar{p}(t) \approx 0.0962185, \tag{5.13}$$

which leads to

$$u_5 \approx 6.3836 \cdot 10^{14}, \tag{5.14}$$

roughly a factor of ten higher than the lower bound we had computed. We must be careful, though: we have no mathematical guarantee that these values are even remotely correct. For example, the chain might in the beginning stabilize to some distribution on some subset of all USOs, and only much later converge to the actual uniform distribution on all USOs. In the wrong distribution that we observe, combed USOs might be much more (or much less) frequent than they should, in which case the results of the experiment would be meaningless.

On the other hand, the author is optimistic and actually believes that there is some truth behind the numbers in (5.13) and (5.14). This optimism also comes from the fact that for $d = 3, 4$, we can compare our experimental results with the known values of u_3, u_4, and they perfectly agree.

Bibliographical Remarks

The casino walk is folklore; it appears for example in the book by Grimmet and Stirzaker [5]. Morris's USO and the analysis of `RandomEdge` is due to Walter Morris [10]. The explicit bound for the expected number of steps is new. Markov chains are classical, Grimmet and Stirzaker treat them in depth. The Markov chain for USOs is due to Matoušek and Wagner [9]. The idea of counting USOs with this chain is new.

Exercises

Exercise 5.1 *Consider the random walk on $\{0, \ldots, N\}$ which starts in k, moves to the left with probability p and to the right with probability $q = 1 - p$. The walk ends if 0 or N is reached.*

(i) What is the probability that the walk reaches N?

(ii) What is the expected number of steps in the walk until either 0 or N is reached?

Remark: *In the course, we have solved (ii) for the case $p = 1/2$. This extension covers the more realistic scenario that your probability of winning a round of roulette by betting on red is strictly smaller than $1/2$, because the bank always wins if zero comes up.*

Exercise 5.2 *Prove that in any USO of C_n, there is a directed path of length $|X \oplus Y|$ from any vertex Y to the unique sink X.*

Exercise 5.3 *Prove Corollary 5.10 of Lemma 5.9.*

Exercise 5.4 *Let $\#_X(1), \#_X(+)$ be the number of 1-bits, respectively plus-signs in the vertex X at level 1 of Morris' USO on C_n, n odd. Prove that*

$$\#_X(1) + \#_X(+)$$

is a constant only depending on n (which one?). Conclude that

$$\#_X(1) = 1 \quad \Leftrightarrow \quad \#_X(+) = \frac{n+1}{2}.$$

Can you generalize this exercise to vertices at level k (vertices with exactly k $(1,+)$ combinations)?

Exercise 5.5 *Let $N = \{1, \ldots, n\}$ and consider the function $s : 2^N \to 2^N$ defined by the following recursive rule.*

$$\begin{aligned} s(\emptyset) &= \emptyset, \\ s(A \cup \{i\}) &= \{1, \ldots, i\} \setminus s(A), \quad A \subseteq \{1, \ldots, i-1\}. \end{aligned}$$

(i) Prove that s defines a USO $\mathcal{O}$ on C_n.

(ii) Prove that there is a directed path in $\mathcal{O}$ which visits all 2^n vertices.

(iii) Prove that the expected number of vertex evaluations performed by the random walk on $\mathcal{O}$ is bounded by $O(n^2)$, for any start vertex.

Exercise 5.6 *Let u_n be the number of unique sink orientations of the n-cube (for example, $u_0 = 1$, $u_1 = 2$ and $u_2 = 12$).*

Prove that

$$u_n \geq 2\,(u_{n-1})^2, \quad n \geq 1,$$

and derive from this an explicit lower bound for u_n.

Exercise 5.7 *Let P be the transition matrix of a homogeneous Markov chain. Prove that the distribution q_t at time t satisfies*

$$q_t = q_0 P^t, \quad t \geq 0.$$

Exercise 5.8 *(i) Find a* non-irreducible *Markov chain with no unique stationary distribution.*

(ii) Find an irreducible, non-aperiodic *Markov chain and some distribution q_0 such that q_t does not converge to the stationary distribution.*

(iii) Find an irreducible, aperiodic, non-symmetric *Markov chain for which the stationary distribution is not the uniform distribution.*

Exercise 5.9 *Let $\mathcal{O}$ be a USO on C_n, and let $\mathcal{V} \subseteq 2^N$ be the set of vertices forming one i-phase. Prove that reorienting all edges*

$$\{X, X \oplus \{i\}\}, \quad X \in \mathcal{V}$$

gives rise to a USO $\mathcal{O}'$. (Because $\mathcal{O}'$ is a neighbor of $\mathcal{O}$ in direction i, $\mathcal{O}'$ has the same i-phases.)

Exercise 5.10 *Suppose you are working as a security person on the tram network of Zurich. Your goal is to be "everywhere at the same time". When you realize that this is not possible, you become less ambitious: you travel from station to station, hopping on and off trams, with the goal of being at every station with the same probability.*

Describe a random walk on the tram network which lets you achieve this goal in the long run.

Chapter 6

Acyclic Unique Sink Orientations

In Chapter 3, we have shown that an expected number of at most

$$O\left(\left(\frac{43}{20}\right)^{n/2}\right) \approx 1.466^n$$

vertex evaluations suffices to find the sink of any USO of C_n. This bound is attained by the *product algorithm*, based on the optimal algorithm for $n = 2$. Not much more is known in the general case. Using the optimal algorithm for $n = 3$ as a basis for the product algorithm,[1] we can slightly improve over the above bound and get an expected number of

$$O\left(\left(\frac{4074633}{1369468}\right)^{n/3}\right) \approx 1.438^n$$

vertex evaluations. This is what we know when no further restrictions on the USO are made.

In this final chapter, we want to show that the above bounds can be substantially improved if the USO is *acyclic*, which means the obvious: there are no directed cycles in the orientation.

6.1 The RandomFacet Algorithm

We develop an algorithm that works for all USOs; only in the analysis, we make use of acyclicity. The algorithm is pretty close in spirit to the simple product algorithm— based on the optimal algorithm for $n = 1$—that previously gave us the

$$\left(\frac{3}{2}\right)^n$$

bound: we choose some direction i of the cube; along direction i, the cube decomposes into two *facets*. Recursively, we find the sink Y in one of the facets. If this sink is not yet the global

[1]The optimal algorithm for $n = 3$ can be found by the game-theoretic methods of Chapter 4, but a substantial refinement (exploiting symmetries) is necessary in order for this approach to stay feasible. For $n = 4$, the optimal algorithm is unknown.

sink (which we can test by looking at the orientation of its incident edge in direction i), we recursively find the sink in the other facet, and the result must be the global sink. When we choose the facet over which we optimize first uniformly at random among the two facets in direction i, we arrive at the product algorithm and its expected complexity $(3/2)^n$.

Here, we will proceed slightly differently: throughout the recursion, the algorithm not only maintains the current face but also a vertex X in that face; the (sub)facet to be searched first will then be chosen uniformly at random among the facets containing X. In other words, the algorithm chooses a *random* direction i, and among the two facets along direction i selects the one containing X as the first facet to search for the sink. In case the facet sink Y is not yet the global sink, we search the other facet, starting from (and this is the key!) the vertex $Y \oplus \{i\}$.

Face/vertex pairs can conveniently be written as follows.

Definition 6.1 *Let $N = \{1, \ldots, n\}$ be some fixed ground set of C_n. For $X, I \in 2^N$, the set*

$$\langle X, I \rangle := \{X \oplus J \mid J \subseteq I\}$$

is the face spanned *by X* in directions I.

It is clear that

$$\langle X, I \rangle = [X \setminus I, X \cup I]$$

in our previous notation of faces as intervals of sets (page 31). Also, every face $[A, B]$ can be written for example as $[A, B] = \langle A, B \setminus A \rangle$, so we do not 'loose' any faces in our new setting. Here is a lemma listing some simple properties (its proof is Exercise 6.1).

Lemma 6.2 *Let $X, I \in 2^N, i \in N$.*

(i) $\langle X, I \rangle = \langle Y, I \rangle$*, for all* $Y \in \langle X, I \rangle$.

(ii) $\langle X, I \rangle = \langle X, I \setminus \{i\} \rangle \cup \langle X \oplus \{i\}, I \setminus \{i\} \rangle$.

(iii) $\langle X, I \setminus \{i\} \rangle$ *contains exactly one of the vertices* Y *and* $Y \oplus \{i\}$*, for any* $Y \in \langle X, I \rangle$.

Figure 6.1 illustrates the process of searching for $\odot(\langle X, I \rangle)$, assuming direction i is chosen for the decomposition into two facets. The figure also emphasizes the fact that the algorithm actually follows a directed path in the orientation: whenever the current vertex Y changes, it is replaced by a neighbor $Y \oplus \{i\}$, reached from Y by following an outgoing edge. Thus, the algorithm can be considered as a random walk, with more sophisticated rules than the simple random walk `RandomEdge` of Algorithm 5.6 from the previous chapter. Here is the algorithm, written down formally.

Algorithm 6.3 *Let s be the outmap of a USO $\mathcal{O}$ of C_n. Given a pair (X, I), the following algorithm evaluates and returns $\odot_{\mathcal{O}}(\langle X, I \rangle)$.*

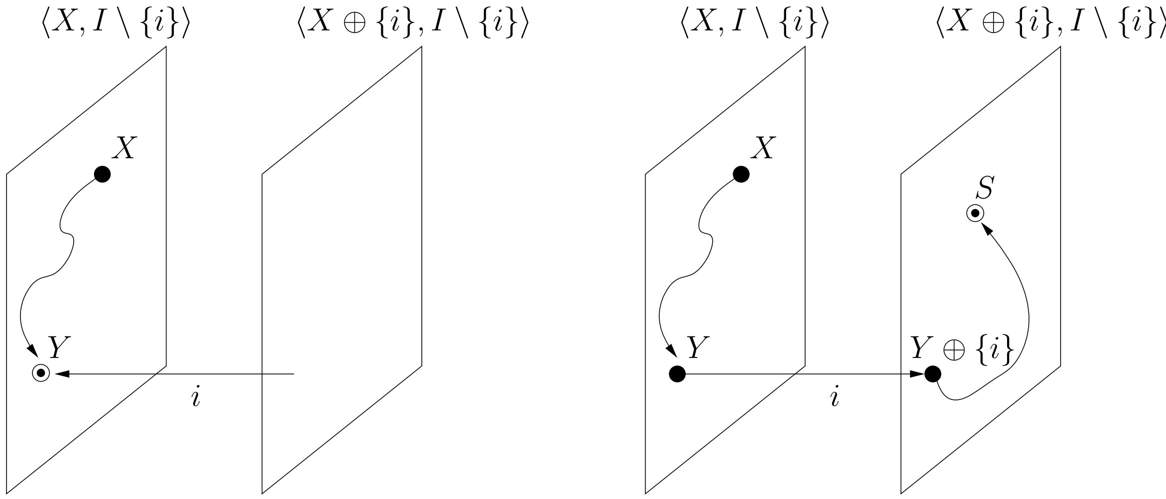


Figure 6.1: To find the sink of $\langle X, I\rangle$, we first recursively find the sink Y in the facet $\langle X, I \setminus \{i\}\rangle$. If Y has an incoming edge in direction i, we are done (left), otherwise, we recursively find the sink S in the other facet $\langle X \oplus \{i\}, I \setminus \{i\}\rangle = \langle Y \oplus \{i\}, I \setminus \{i\}\rangle$, which must then be the sink of $\langle X, I\rangle$ (right).

```
RandomFacet(X, I):
  IF I = ∅ THEN
    evaluate X
    RETURN X
  ELSE
    choose i ∈ I at random
    Y :=RandomFacet(X, I \ {i})
    IF i ∉ s(Y) THEN
      RETURN Y
    ELSE
      RETURN RandomFacet(Y ⊕ {i}, I \ {i})
    END
  END
```

The correctness proof is an easy induction over $|I|$, using Lemma 6.2. Actually, it should already be clear from the above discussion that the algorithm works; moreover, even if the USO has cycles, no vertex will be visited twice by the algorithm (which can also be shown by induction). It follows that the number of vertex evaluations is always at most 2^n, but as we show next, the expected number is much smaller in the acyclic case.

6.2 Analysis of `RandomFacet`

From now on, the USO $\mathcal{O}$ is assumed to be acyclic. In this case, the vertices can be *topologically sorted*, meaning that there is some total order $<$ on the vertices such that for all X, Y,

$$X \text{ reachable from } Y \text{ (along a directed path in } \mathcal{O}) \quad \Rightarrow \quad X < Y. \tag{6.1}$$

In this order, the global sink is necessarily the minimum, because it is reachable from any other vertex (Lemma 5.7). In general, $<$ is not unique. For example, the uniform orientation of C_2 allows two different topological orders, see Figure 6.2.

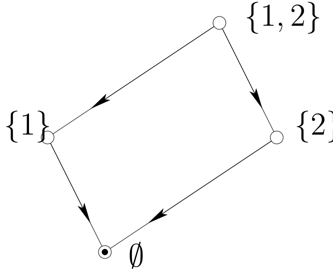


Figure 6.2: The uniform USO of C_2 has two different topological orders of the vertices: $\emptyset < \{1\} < \{2\} < \{1,2\}$ and $\emptyset < \{2\} < \{1\} < \{1,2\}$.

Let us fix a topological order $<$ once and for all.[2] Then we can introduce the crucial concept of *fixed elements*.

Definition 6.4 *Let $X, I \subseteq 2^N$. $i \in I$ is called* fixed *in (X, I), if and only if*

$$X < \odot(\langle X \oplus \{i\}, I \setminus \{i\}\rangle).$$

Otherwise, i is called free *in (X, I).*

What does this mean? We have already noted that $\langle X, I\rangle$ subdivides into two facets along direction i. If i is fixed, the sink of the facet $\langle X \oplus \{i\}, I \setminus \{i\}\rangle$ *not* containing X is not reachable from X along any directed path. By Lemma 5.7, if the sink of the facet is not reachable, *no* vertex of the facet is reachable, and we obtain the following

Corollary 6.5 *Let $i \in I$ be fixed in (X, I). Then*

(i) no vertex in $\langle X \oplus \{i\}, I \setminus \{i\}\rangle$ is reachable from X, and consequently,

(ii) a call to `RandomFacet`(X, I) visits only vertices in the facet $\langle X, I \setminus \{i\}\rangle$, equivalently, it performs no second recursive call.

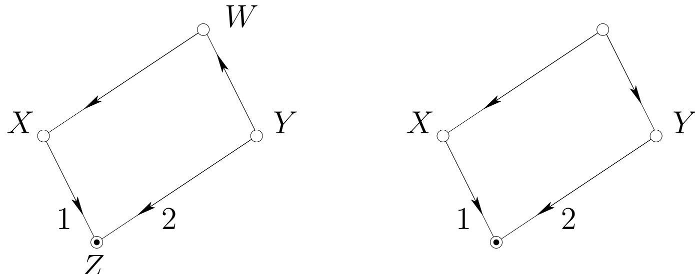


Figure 6.3: The concept of fixed and free elements. Left: 2 is fixed in $(X, \{1,2\})$, but 1 is free. $(Y, \{1,2\})$ and $(W, \{1,2\})$ have no fixed elements. Both 1 and 2 are fixed in $(Z, \{1,2\})$. Right: if $X < Y$, 2 is fixed in $(X, \{1,2\})$, but $(Y, \{1,2\})$ contains only free elements. If $Y < X$, 1 is fixed in $(Y, \{1,2\})$ and $(X, \{1,2\})$ has only free elements.

As an illustration of this concept, consider Figure 6.3, where it also becomes clear that the notion of fixed elements depends on the order $<$, and that even for free elements i, the statements of Corollary 6.5 might hold (this will neither hurt nor benefit us).

Here are two more consequences of Definition 6.4 that we need later.

Lemma 6.6 *Let $X, I \subseteq 2^N, Y \in \langle X, I\rangle, i \in I$.*

(i) If $j \in I \setminus \{i\}$ is fixed in (X, I), then j is also fixed in $(X, I \setminus \{i\})$.

(ii) If $j \in I$ is fixed in (X, I) and $Y < X$, then j is also fixed in (Y, I).

Proof. (i) j being fixed in (X, I) means that

$$X < \odot(\langle X \oplus \{j\}, I \setminus \{j\}\rangle) \leq \odot(\langle X \oplus \{j\}, I \setminus \{i, j\}\rangle),$$

where the latter inequality just says that the sink of a face is equal to or reachable from the sink of any subface. It follows that j is fixed in $(X, I \setminus \{i\})$ as well. (ii) the assumptions yield

$$Y < X < \odot(\langle X \oplus \{j\}, I \setminus \{j\}\rangle). \tag{6.2}$$

This in particular implies that

$$Y \notin \langle X \oplus \{j\}, I \setminus \{j\}\rangle,$$

equivalently that

$$Y \oplus \{j\} \in \langle X \oplus \{j\}, I \setminus \{j\}\rangle,$$

by Lemma 6.2. Using part (ii) of Lemma 6.2 once more, we get

$$\langle X \oplus \{j\}, I \setminus \{j\}\rangle = \langle Y \oplus \{j\}, I \setminus \{j\}\rangle,$$

[2] We could as well work with the *partial order* of the vertices defined by the reachability relation in (6.1), but that would unnecessarily complicate things in the sequel.

so (6.2) gives

$$Y < \odot(\langle Y \oplus \{j\}, I \setminus \{j\}\rangle).$$

This means that j is fixed in (Y, I). ◻

Intuitively, the less free elements there are in (X, I), the faster RandomFacet(X, I) will be. In fact, if (X, I) has k free elements, then all vertices $Y \leq X$ visited during the call to RandomFacet(X, I) must be in a common k-dimensional face with the sink of (X, I), because they cannot differ from the sink in any fixed element. This motivates the following

Definition 6.7 *Let $X, I \subseteq 2^N$. The* hidden dimension *of (X, I) is the number*

$$h(X, I) = |I| - \{i \in I \mid i \textit{ is fixed in } (X, I)\} = \{i \in I \mid i \textit{ is free in } (X, I)\}.$$

For example, in Figure 6.3 (left), we have $h(X, \{1, 2\}) = 1$ and $h(Y, \{1, 2\}) = 2$.

Here is the main result of this chapter. It looks somewhat messy now, but things will clear up later.

Theorem 6.8 *Define $t_k(m)$ to be the maximum expected number of vertex evaluations in a call to RandomFacet(X, I), where $|I| = m$ and $h(X, I) \leq k$. Then $t_0(m) = 1$ and*

$$t_k(m) \leq \frac{1}{m}\left((m-k)t_k(m-1) + \sum_{j=1}^{k}(t_{k-1}(m-1) + t_{j-1}(m-1))\right), \quad k > 0.$$

Proof. If $k = 0$, X must be the sink of $\langle X, I\rangle$, and it is easy to see that exactly one vertex evaluation (for the pair $(X, \emptyset)$) takes place in RandomFacet(X, I).

If $k > 0$, we may assume without loss of generality that $h(X, I) = k$: if the worst pair (X, I) with hidden dimension at most k has $h(X, I) = \ell < k$, we get $t_k(m) = t_\ell(m)$, and proving the theorem for ℓ yields the required bound for k as well. There are two cases now.

Case (a) i is fixed in (X, I). Because all fixed elements $j \neq i$ are also fixed in $(X, I \setminus \{i\})$ by Lemma 6.6(i), we have $h(X, I \setminus \{i\}) \leq k$. Moreover, there will be no second recursive call in this case due to Corollary 6.5, so the expected number of vertex evaluations is bounded by $t_k(m-1)$.

Case (b) i is free in (X, I). Lemma 6.6(i) gives us a bound of $h(X, I \setminus \{i\}) \leq k - 1$ in this case, because we remove a free element. It follows that the expected number of vertex evaluations in the first recursive call can be bounded by $t_{k-1}(m-1)$.

To estimate the performance of the second recursive call, let us order the k free elements $i_1, \ldots, i_k$ in such a way that

$$\odot(\langle X, I \setminus \{i_1\}\rangle) \leq \cdots \leq \odot(\langle X, I \setminus \{i_k\}\rangle).$$

If $i = i_j$, and if there is actually a second recursive call with pair $(Y \oplus \{i\}, I \setminus \{i\})$, we know that for $\ell > j$,

$$\begin{aligned} Y \oplus \{i\} < Y &= \odot(\langle X, I \setminus \{i\}\rangle) \\ &\leq \odot(\langle X, I \setminus \{i_\ell\}\rangle) \\ &= \odot(\langle Y \oplus \{i\} \oplus \{i_\ell\}, I \setminus \{i_\ell\}\rangle), \end{aligned}$$

where the latter equality follows—as in the proof of Lemma 6.6(ii)—from Lemma 6.2.

This, on the other hand, means that $i_{j+1}, \ldots, i_k$ are fixed in $(Y \oplus \{i\}, I)$ and therefore also in $(Y \oplus \{i\}, I \setminus \{i\})$, *in addition* to the elements that were already fixed in (X, I) (Lemma 6.6). It follows that

$$h(Y \oplus \{i\}, I \setminus \{i\}) \leq (k-1) - (k-j) = j - 1.$$

We conclude that the expected number of vertex evaluations in the second recursive call is bounded by $t_{j-1}(m-1)$ if $i = i_j$. Because this happens with probability $1/m$ (which is also the probability that i is any of the $m-k$ fixed elements in case (a)), the claimed bound follows. $\square$

The proof formalizes our intuition that it is better for the algorithm to choose some i for which

$$Y = \odot(\langle X, I \setminus \{i\}\rangle)$$

is rather small in the order $<$, because then Y will already be quite close to the sink. Actually, choosing $i = i_1$ would be best. Unfortunately, the algorithm does not know the order $i_1, \ldots, i_k$; by guessing i randomly, we expect i to be somewhere in the middle which is quite an improvement over the worst case in which we would choose $i = i_k$.

The bound on $t_k(m)$ still looks ugly, but upon closer inspection, it turns out that the bound actually does not depend on m.

Theorem 6.9 *For fixed n, define*

$$T(k) := \max_{m \leq n} t_k(m).$$

Then,

$$\begin{aligned} T(0) &= 1, \\ T(k) &\leq T(k-1) + \frac{1}{k}\sum_{j=1}^{k} T(j-1), \quad k > 0. \end{aligned}$$

Proof. $T(0) = 1$ is obvious. For $k > 0$, consider the value m that leads to the maximum in the definition of $T(k)$. Then we get

$$\begin{aligned} T(k) &= t_k(m) \\ &\leq \frac{1}{m}\left((m-k)t_k(m-1) + \sum_{j=1}^{k}(t_{k-1}(m-1) + t_{j-1}(m-1))\right) \\ &\leq \frac{1}{m}\left((m-k)T(k) + \sum_{j=1}^{k}(T(k-1) + T(j-1))\right), \end{aligned}$$

which is equivalent to

$$\frac{k}{m}T(k) \leq \frac{1}{m}\left(kT(k-1) + \sum_{j=1}^{k} T(j-1)\right).$$

Multiplying this with m/k proves the statement.

Using the method of *generating functions* (or the method of guessing), the following explicit bound on $T(k)$ can be obtained. Once we have it, a simple proof by induction (which we omit here) can be used to verify it.

Theorem 6.10 *For all $k \geq 0$,*

$$T(k) \leq \sum_{j=0}^{k} \frac{1}{j!}\binom{k}{j}.$$

In particular, this gives

$$T(k) \leq e^{2\sqrt{k}}.$$

Proof. We only show how the second bound follows from the first one. For this, we use the easy estimate

$$\binom{a}{b} \leq \frac{a^b}{b!}$$

and compute

$$T(k) \leq \sum_{j=0}^{k} \frac{1}{j!}\binom{k}{j} \leq \sum_{j=0}^{k} \frac{1}{j!}\frac{k^j}{j!} = \sum_{j=0}^{k}\left(\frac{\sqrt{k}}{j!}\right)^2 \leq \left(\sum_{j=0}^{k}\frac{\sqrt{k}}{j!}\right)^2 \leq \left(\sum_{j=0}^{\infty}\frac{\sqrt{k}}{j!}\right)^2 = e^{2\sqrt{k}}.$$

□

Because in any USO of C_n, $h(X,I) \leq n$ for any pair (X,I), the findings of this chapter can be summarized as follows.

Theorem 6.11 *Algorithm 6.3 `RandomFacet` finds the sink of any acyclic n-cube USO with an expected expected number of at most*

$$e^{2\sqrt{n}}$$

vertex evaluations.

This is still exponential, but *much smaller* than the previous bounds. We conclude with a table that compares the expected number of vertex evaluations incurred by the algorithms we have considered in this course, see Table 6.1.

What can we conclude from this table? When we are willing to wait for around 10^9 vertex evaluations (this sounds like a realistic number, given the capabilities of today's computers), we can find the sinks of general USOs in dimensions up to around 50, using the product algorithm. The trivial algorithm quits at around $n = 30$, while `RandomEdge` is a bad idea already much earlier.

If we know that the USO is acyclic, however, we can go up to dimensions around $n = 130$, using `RandomFacet`. For $n = 127$, the algorithm seems to visit many vertices, but comparing this number with the *total* number of vertices, we see that the fraction of visited vertices is approximately

$$10^{-30} = 0.000000000000000000000000000001.$$

	`Trivial` 2^n	`Product` $\frac{3}{2} \cdot \frac{43}{20}^{(n-1)/2}$	`RandomEdge` $\geq 2\lfloor e\frac{n-1}{2}!\rfloor$	`RandomFacet` $\leq \sum_{j=0}^{n} \frac{1}{j!}\binom{n}{j}$
3	8	3.23	4	5.67
7	128	14.9	32	26.0
15	$3.28 \cdot 10^4$	$3.19 \cdot 10^2$	$2.7 \cdot 10^4$	234.0
31	$2.15 \cdot 10^9$	$1.45 \cdot 10^5$	$7.11 \cdot 10^{12}$	$5.5 \cdot 10^3$
63	$9.22 \cdot 10^{18}$	$3.03 \cdot 10^{10}$	$4.47 \cdot 10^{34}$	$5.15 \cdot 10^5$
127	$1.70 \cdot 10^{38}$	$1.32 \cdot 10^{21}$	$1.01 \cdot 10^{88}$	$3.32 \cdot 10^8$
255	$5.79 \cdot 10^{76}$	$2.49 \cdot 10^{42}$	$1.64 \cdot 10^{214}$	$3.30 \cdot 10^{12}$

Table 6.1: Runtime comparisons of four sink-finding algorithms for certain odd values of n; the Trivial algorithm goes through all vertices; the Product algorithm combines (for odd dimensions) the optimal algorithms for dimensions 1 and 2; `RandomEdge` is the simple random walk, and `RandomFacet` is the sophisticated random walk of this chapter, its runtime bound is valid only for the acyclic case.

On the other hand, we cannot argue around the fact that all methods we have studied are impractical as n gets really large (in the thousands, say). The great challenge is to find better algorithms, in the hope of ultimately getting a bound which is polynomial in n. Currently, we are very far away from this goal.

Bibliographic Remarks

The subexponential algorithm for acyclic USOs explicitly appears first in a paper by Ludwig [7], as a specialization of general techniques by Kalai [6] as well as Matoušek, Sharir and Welzl [8]. The explicit $\exp(2\sqrt{n})$ bound is derived by Gärtner [3].

Exercises

Exercise 6.1 *Prove Lemma 6.2!*

Chapter 7

Solutions to Exercises

This final chapter contains solutions to the exercises, organized by chapters. Due to severe time constraints, some solutions are only sketched, and even for the ones that are complete, absolutely no guarantee is given. If you find errors (this actually holds for all chapters), please report them to me.

7.1 Solutions to Chapter 1

Exercise 1.1

(i) The upper bound $t(K_n) \leq n$ is clear, because if we evaluate all vertices, we have surely evaluated the sink. For the lower bound, we argue with an adversary who operates the oracle in such a way that we are forced to ask n questions. The construction is as follows:

Assume we have already evaluated $i-1$ vertices $v_1, \ldots, v_{i-1}, 1 \leq i \leq n$. Because our algorithm is deterministic, the answers we have obtained so far determine the next vertex v_i that we evaluate. The oracle will tell us that

$$\mathrm{out}(v_i) = V \setminus \{v_1, \ldots, v_{i-1}\},$$

i.e. all edges from vertices we have already evaluated are incoming, and all edges to vertices we haven't seen yet are outgoing. This in particular implies that only the n-th vertex we evaluate will actually be the sink, proving the lower bound.

In this construction, it is important that the adversary can always guarantee at least one orientation which is consistent with all answers given so far—otherwise, we could accuse him of cheating.

Such an orientation does indeed exist: after having evaluated v_i, the adversary has revealed that D contains the following edges:

(a) $(v_k, v_\ell), k < \ell \leq i$ (between evaluated vertices),

(b) $(v_k, v), k \leq i, v \in V \setminus \{v_1, \ldots, v_i\}$ (between evaluated and non-evaluated vertices).

Edges between non-evaluated vertices are still unknown to us. In fact, the adversary can put these vertices into *any* order $v_{i+1}, \ldots, v_n$, and the complete orientation induced by

$$D = \{(v_k, v_\ell) \mid k < \ell \leq n\}$$

is acyclic, contains a unique sink v_n and is compatible with all answers. The actual order in which these remaining vertices will appear in the end is exactly the order in which we evaluate them.

(ii) The graph of the n-cube can be realized as follows: the vertices are all the 2^n bitvectors of length n, and two bitvectors are connected by an edge iff they differ in exactly one coordinate, see Figure 7.1.

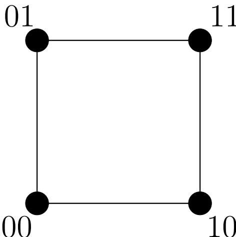


Figure 7.1: Graph of the 2-cube

The upper bound is again easy: evaluate all 2^{n-1} vertices with even parity (even number of one-entries). Because all edges connect a vertex with even parity to one with odd parity, these evaluations reveal the orientations of *all* edges. If the sink has even parity itself, we are done, otherwise, we need one more evaluation.

For the lower bound, the strategy will be similar as in part 1, but we have to be more careful. For example, when evaluating a vertex v, the adversary can in general *not* tell us that $\text{out}(v)$ contains exactly the neighbors of v that have not been evaluated so far. Consider the situation in the 2-cube after evaluating vertices 00 and 11: answering

$$\text{out}(00) = \{01, 10\}, \quad \text{out}(11) = \{01, 10\}$$

would result in the orientation of Figure 7.2 which contains *two* sinks. Because we have been guaranteed that the orientation contains exactly one sink, we would now accuse the adversary of cheating.

Instead, the adversary strategy will be the following: we first construct a Hamiltonian cycle of C_n (a closed path containing every vertex exactly once). Let us prove by induction that such a cycle exists, where the base of the induction is $n = 2$: in this case, we obviously have such a cycle. Let us assume that we have a cycle for C_{n-1}. The vertices of C_n with last coordinate 1 (the upper facet) form a C_{n-1}, and so do the ones with last coordinate 0 (the lower facet). By the induction hypothesis, there are Hamiltonian cycles in both facets, where we can assume that they are copies of each other (connecting the vertices in the same order, within the facets). By just leaving out one edge of each cycle, we get two Hamiltonian *paths*, and by concatenating them with two extra edges between the facets, we get a Hamiltonian cycle in C_n, see Figure 7.3.

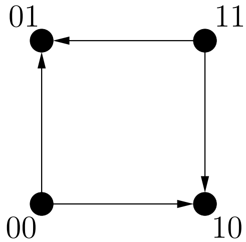


Figure 7.2: A C_2-orientation with two sinks

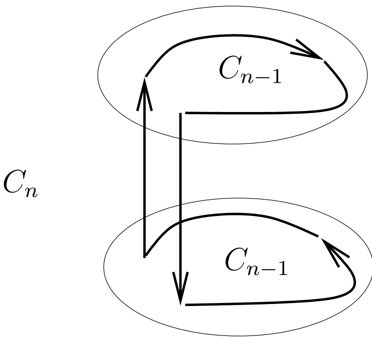


Figure 7.3: Constructing a Hamiltonian cycle

So C_n has a Hamiltonian cycle if $n \geq 2$. (C_1 does not have one, but it is easy to check that the exercise statement is true for $n = 1$.)

Let us assume that Alice can find the sink of C_n with 2^{n-1} evaluations. We can answer her first $2^{n-1} - 1$ question as if this Hamiltonian cycle were directed, and the other edges are all outgoing. (Observe that so far, this orientation does not have a sink at all!) After $2^{n-1} - 1$ queries, there are still three consecutive non-evaluated vertices, or we have two pairs of two consecutive non-evaluated vertices, see Figure 7.4 In both cases, there are still (at least) two

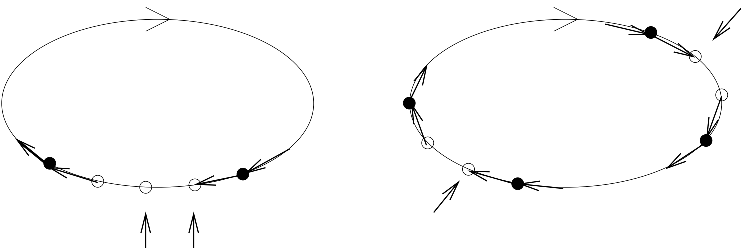


Figure 7.4: The possible choices of the sink after $2^{n-1} - 1$ evaluations.

vertices which will become the unique sink after reorienting one edge about whose orientation Alice has no information. It follows that even if Alice has one more question, she will not be able to deduce the sink.

(iii) If one evaluates the vertices of a *vertex cover*, then the orientation of every edge is known, so it is also known where the sink is. In the worst case, we still have to evaluate it.

Exercise 1.2

If $X_n = n$ then for every $i > 1$ the algorithm has to choose $i-1$ as the next number from $\{1, 2, \ldots, i-1\}$. This probability is $\frac{1}{i-1}$, so prob$(X_n = n) = \frac{1}{(n-1)!}$.

For the rest we need to define the Stirling numbers (of the first kind). Any permutation $\pi \in S_n$ decomposes into cycles, which are sequences $(n_0, n_1, \ldots n_{\ell-1})$ of elements with $\pi(n_i) = n_{(i+1) \bmod \ell}$, for $0 \le i < \ell$. For example, the permutation

i	1	2	3	4	5
$\pi(i)$	3	5	4	1	2

consists of the two cycles

$$(1, 3, 4)(2, 5).$$

The identity has n cycles $(1)(2)\cdots(n)$, while for example any circular shift consists of one cycle of size n. The cycle decomposition is unique if we let every new cycle start with the smallest element not used so far. We define

$$\left[\begin{array}{c} n \\ k \end{array}\right] := |\{\pi \in S_n | \pi \text{ has } k \text{ cycles}\}|.$$

For example $\left[\begin{array}{c} 0 \\ 0 \end{array}\right] := 1$, $\left[\begin{array}{c} n \\ 0 \end{array}\right] = 0$ for $n > 0$, $\left[\begin{array}{c} 0 \\ k \end{array}\right] = 0$ for $k > 0$, and the following recurrence relation holds for Stirling numbers of first kind.

$$\left[\begin{array}{c} n \\ k \end{array}\right] = (n-1)\left[\begin{array}{c} n-1 \\ k \end{array}\right] + \left[\begin{array}{c} n-1 \\ k-1 \end{array}\right], \; n > 0.$$

To see this, observe that there are exactly $\left[\begin{array}{c} n-1 \\ k-1 \end{array}\right]$ permutations with n appearing in a cycle (n) of its own. The ones having n in some nontrivial cycle can be obtained by inserting n into some cycle of a permutation of $n-1$ elements with k cycles. Since there are exactly $n-1$ different ways to do this for every such permutation (j ways for every cycle of size j), the formula follows.

For our algorithm let $p(n, k) :=$ prob$(X_{n+1} = k+1)$. Now we have by the definition of the algorithm that

$$p(n, k) = \sum_{i=0}^{n-1} p(i, k-1).$$

This entails

$$n \cdot p(n, k) - (n-1)p(n-1, k) = p(n-1, k-1).$$

Multiplying the equation by $(n-1)!$ and setting $f(n, k) := n!p(n, k)$ yields

$$f(n, k) = (n-1)f(n-1, k) + f(n-1, k-1).$$

Furthermore $f(0, k) = p(0, k) = \left[\begin{array}{c} 0 \\ k \end{array}\right]$, so $f(n, k) = \left[\begin{array}{c} n \\ k \end{array}\right]$, and

$$p(n, k) = \text{prob}(X_{n+1} = k+1) = \frac{1}{n!}\left[\begin{array}{c} n \\ k \end{array}\right].$$

Exercise 1.3

(i) By induction on n, where the case $n = 1$ is trivial. Assume that the statement is true for graphs on $n-1$ vertices. and let G have n vertices. Because G is acyclic, G must have a source v (vertex with no incoming edge); we remove v and its incident edges from G and inductively get a topological sorting $\sigma : V \setminus \{v\} \to [n-1]$ in the resulting graph G'. Extending σ to V by setting $\sigma(v) = n$ gives a topological sorting of G. To check this, we only need to consider edges involving v, and because v was the source, they must be of type (v, w). Then, however, the requirement of the topological sorting is fulfilled because $\sigma(v) = n > n-1 \geq \sigma(w)$.

(ii) We show three implications:

(a)⇒(c): let σ be the unique topological sorting. We claim that $\sigma^{-1}(n), \dots, \sigma^{-1}(1)$ is a directed Hamiltonian path. Namely, if there were two vertices $\sigma^{-1}(i)$ and $\sigma^{-1}(i-1)$ without a connecting edge, we could swap their order in the topological sorting and obtain a different topological sorting, a contradiction.

(c)⇒(b): For every pair (v, w) there is directed path between them, along the Hamiltonian cycle.

(b)⇒(a): for any pair (v, w), the directed path uniquely determines the order of the elements in the topological sorting: if the path goes from v to w, we must have $\sigma(v) > \sigma(w)$, and if it goes from w to v, the converse holds. Because two different topological sortings would have to order at least one pair in two ways, there can be only one such order.

(iii) For K_n, every acyclic orientation has a unique topological sorting, since for every two vertices v, w there is an edge (a path of length 1) between them. And there are $n!$ different topological sortings.

7.2 Solutions to Chapter 2

Exercise 2.1 (i) The function f_{B_λ} is quadratic in x and therefore we can write $f_{B_\lambda}(x) \leq 1$ as $\|x - c\|^2 \leq \gamma$ for some vector c and a (not necessarily positive) constant $\gamma \in \mathbb{R}$. In order to prove that B_λ is indeed a ball, we must show $\gamma \geq 0$. Since $B_0 \cap B_1 \neq \emptyset$, there exists at least one real point y for which both $f_{B_0}(y) \leq 1$ and $f_{B_1}(y) \leq 1$ hold. It follows that $f_{B_\lambda}(y) = (1-\lambda)\, f_{B_0}(y) + \lambda f_{B_1}(y) \leq 1$. This shows that $\|x - c\|^2 \leq \gamma$ has a real solution, which is only possible if $\gamma \geq 0$.

(ii) is easily seen to hold by plugging $p \in R$ (or $p \in S$, respectively) into $f_{B_\lambda} = (1-\lambda)\, f_{B_0} + \lambda f_{B_1}$.

(iii) Let $f_{B_0} = \|x - c_0\|^2/\rho_0$ and $f_{B_1} = \|x - c_1\|^2/\rho_1$ be the defining functions of the given balls B_0 and B_1. Expanding $f_{B_\lambda} \leq 1$ we obtain

$$f_{B_\lambda} = x^T x \left(\frac{1-\lambda}{\rho_0} + \frac{\lambda}{\rho_1} \right) - 2x^T \left(\frac{1-\lambda}{\rho_0}\, c_0 + \frac{\lambda}{\rho_1}\, c_1 \right) + \text{const} \leq 1,$$

which we can write in the form $\|x - c\|^2/\gamma \leq 1$ for

$$c = \left(\frac{1-\lambda}{\rho_0}\, c_0 + \frac{\lambda}{\rho_1}\, c_1 \right) \Big/ \left(\frac{1-\lambda}{\rho_0} + \frac{\lambda}{\rho_1} \right).$$

This shows that the center c of B_λ is a convex combination of the centers c_0, c_1. That is, as λ ranges from 0 to 1, the center c travels on a line from c_0 to c_1. Notice now that the radius of B_λ is simply the distance from c to a point $p \in \partial B_0 \cap \partial B_1$, because by (ii) the point p lies on the boundary of B_λ for any $\lambda \in [0,1]$. The claim now follows from the fact that the distance from a point c moving on a line (namely from c_0 to c_1) to a fixed point p is a strictly convex function.

Exercise 2.2

(i) $\Rightarrow$ (ii). Fix any $p \in R$ and suppose for a contradiction that the vectors $\{q - p \mid q \in R \setminus \{p\}\}$ are linearly dependent. Then there exist coefficients λ_q, $q \in R \setminus \{p\}$, not all zero, such that

$$0 = \sum_{q \in R\setminus\{p\}} \lambda_q (q - p) = \sum_{q \in R\setminus\{p\}} \lambda_q q - \sum_{q \in R\setminus\{p\}} \lambda_q p.$$

Setting $\lambda_p := \sum_{q \in R\setminus\{p\}} \lambda_q$ we thus obtain $0 = \sum_{q\in R} \lambda_q q$ with $\sum_{q \in R} \lambda_q = 0$ for coefficients λ_q, $q \in R$, which are not all zero, contradiction.

(ii) $\Rightarrow$ (iii) is obvious.

(iii) $\Rightarrow$ (i). Assume the points R are affinely dependent, implying that there exist coefficients λ_q, $q \in \mathbb{R}$, not all zero, such that $\sum_{q\in R} \lambda_q q = 0$ with the coefficients summing up to zero. In order to get a contradiction we need to show that for all $p \in R$, the vectors $\{q - p \mid q \in R \setminus \{p\}\}$ are linearly dependent.

So pick an arbitrary $p \in R$. Using $\sum_{q\in R} \lambda_q = 0$ we get

$$0 = \sum_{q\in R} \lambda_q q = \sum_{q\in R} \lambda_q q - \sum_{q\in R} \lambda_q p = \sum_{q\in R\setminus\{p\}} \lambda_q (q-p).$$

Observe here that at least one of the coefficients λ_q, $q \in R \setminus \{p\}$, is nonzero: λ_p cannot be the only nonzero coefficient because all coefficients together sum up to zero. Thus, the points $\{q - p \mid q \in R \setminus \{p\}\}$ are linearly dependent and we are done.

Exercise 2.3

The case $|F| = 1$ is easy because we then have $\bigcirc(F, F) = F$. Thus the center of $\bigcirc(F, F)$ is the single point in F which trivially lies in $\operatorname{aff}(F) = F$. For $|F| > 1$, we proceed as in the exercise.

(a) If $s \in \operatorname{aff}(F)$ then $s = \sum_{q\in F} \lambda_q q$ for coefficients λ_q summing up to one. It follows that for any $p \in F$,

$$s - p = \sum_{q\in F} \lambda_q q - \sum_{q\in F} \lambda_q p = \sum_{q\in F\setminus\{p\}} \lambda_q (q-p) \in \operatorname{lin}(F-p).$$

This shows direction ($\Rightarrow$); the other direction is shown along the same lines: If $s - p \in \operatorname{lin}(F - p)$, there are coefficients λ_q, $q \in F \setminus \{p\}$, such that $s - p = \sum_{q\in F\setminus\{p\}} \lambda_q (q-p)$. Then

$$s = \sum_{q\in F\setminus\{p\}} \lambda_q (q-p) + p = \sum_{q\in F\setminus\{p\}} \lambda_q q + \Big(1 - \sum_{q\in F\setminus\{p\}} \lambda_q\Big) p,$$

and since the involved coefficients sum up to one we conclude $s \in \operatorname{aff}(F)$.

(b) Recall first that we assume the points F to be affinely independent. This together with Fact 2.8(ii) guarantees that the columns of M are linearly independent. So $Mx = 0$ implies $x = 0$ for all vectors x.

Suppose M^TM is not invertible, i.e., there exists a nonzero vector x with $M^TMx = 0$. Then

$$0 = x^TM^TMx = \|Mx\|^2,$$

and hence $Mx = 0$ for $x \neq 0$, which is impossible as we have just seen.

(c) From (a) we know that $s^* \in \operatorname{aff}(F)$ if and only if $s^* - p \in \operatorname{lin}(F - p)$, or, equivalently, if $s^* - p = Mx$ for some vector x. Take $x := (M^TM)^{-1}M^T(s - p)$.

(d) We first show that $s - s^*$ is orthogonal to $\operatorname{aff}(F)$, i.e., $M^T(s - s^*) = 0$. For this, we use the definition of s^* in order to write

$$M^T(s^* - p) = M^TM(M^TM)^{-1}M^T(s - p) = M^T(s - p);$$

which readily implies $M^T(s - s^*) = 0$. We can then conclude

$$\begin{aligned}
\|s - p\|^2 &= \|(s - s^*) + (s^* - p)\|^2 \\
&= \|s - s^*\|^2 + \|s^* - p\|^2 + 2\,(s - s^*)^T(s^* - p) \\
&= \|s - s^*\|^2 + \|s^* - p\|^2 + 2\,(s - s^*)^TM(M^TM)^{-1}M^T(s - p),
\end{aligned}$$

where the last term reduces to zero.

(e) Let c be the center and ρ be the squared radius of $\bigcirc(F, F)$, and suppose $c \notin \operatorname{aff}(F)$. We will construct a smaller ball B with F on the boundary, which will give us the desired contradiction.

We take c^* as the center of B. By part (d), the distance from c^* to any point $p \in F$ is

$$\|c^* - p\|^2 = \|c - p\|^2 - \|c - c^*\|^2 = \rho - \|c - c^*\|^2. \tag{7.1}$$

Since the right-hand side of this equation is independent of p, the point c^* has the *same* distance to all points in F, and hence the ball B of center c^* and squared radius $\rho - \|c - c^*\|^2$ has F on the boundary.

Furthermore, since $c \notin \operatorname{aff}(F)$ by assumption, we have $\|c - c^*\|^2 > 0$ and thus equation (7.1) yields $\rho - \|c - c^*\|^2 < \rho$. Consequently, B is a ball with F on the boundary and smaller radius than $\bigcirc(F, F)$, a contradiction.

(ii) Let S be the set of balls that go through F and have their center in $\operatorname{aff}(T)$. We will set up a system of equations whose solution space precisely encodes the balls S; subsequently we will see that exactly one solution exists.

Fix any point $p \in F$. We want to find the center c of a ball going through F, or, in other words, a point c such that the distance from c to the fixed point p equals the distance from c to any other point in $F \setminus \{p\}$. Setting $c' := c - p$ and denoting by q' the columns $q - p$ of M, $q \in F \setminus \{p\}$, we can write this as

$$c'^Tc' = (c' - q')^T(c' - q'), \qquad \forall q \in F \setminus \{p\}. \tag{7.2}$$

Subtracting $c'^T c'$ from both sides of (7.2) results in a system of *linear* equations in the unknown c':

$$2c'^T q' = q'^T q', \qquad \forall q \in F \setminus \{p\}. \tag{7.3}$$

So far, we have not yet imposed the constrained that the center c of B must lie in $\text{aff}(F)$. By part (a), $c \in \text{aff}(F)$ is equivalent to $c' \in \text{lin}(F - p)$, and thus we can write c' as $c' = Mx$ for some unknown coefficient vector x. By plugging this into (7.3) we obtain the $|F| - 1$ equations $2x^T M^T q' = q'^T q'$ which we can write as

$$2M^T Mx = m,$$

where m is the vector containing the scalar products $q'^T q'$, $q \in F \setminus \{p\}$ (in the same order as in M).

By part (b), $M^T M$ is a regular matrix and hence the system has exactly one solution, x, say. This shows that there exists a unique ball with center in $\text{aff}(F)$ having F on the boundary. Its center is $c = Mx + p$ and its squared radius is $\|Mx\|^2$ by equation (7.2).

Exercise 2.4 The proof is by induction on $k := |S \setminus R|$. For $k = 0$, the algorithm performs a basis computation '$F := R$' and zero violation tests, so the claim holds.

For $k > 0$, the algorithm calls itself recursively at most twice, each time with parameters for which the induction hypothesis applies. Therefore, the claim still holds after the first recursive call, and since the algorithm then continues with a violation test, the claim also holds after the second recursive call.

Exercise 2.5

Let n be a multiple of $(d + 1)$. Then we put a *block* of $n/(d + 1)$ points close to any of the $d + 1$ corners of a regular simplex, as shown in Figure 7.5 for $d = 2, n = 9$. The labels correspond to the indices of the points in the set S.

The claim is that any $(d + 1)$-element set Q of points containing exactly one point from each block appears as a set R during the algorithm. The bound follows, because there are

$$\left(\frac{n}{d+1}\right)^{d+1}$$

such sets Q, and for any (except possibly the last) ball $\bigcirc(Q, Q)$ appearing in the algorithm, at least one violation test will be performed.

To prove the claim, let $Q = \{q_1, \ldots, q_{d+1}\}$ be such a set, ordered by decreasing index in S.

Always removing the last point of S for the first recursive call, we remove q_1 at some stage and recursively compute $\bigcirc(S', \emptyset)$, for S' the prefix of S that stops just before q_1. By construction, $q_1 \notin \bigcirc(S', \emptyset)$, so there will be a recursive call computing $\bigcirc(S', \{q_1\})$. Subsequently, the algorithm computes $\bigcirc(S'', \{q_1\})$, where S'' is the prefix of S stopping just before q_2. Again, $q_2 \notin \bigcirc(S'', \{q_1\})$, and a recursive call for $\bigcirc(S'', \{q_1, q_2\})$ is spawned. Continuing like this, we eventually get the whole set Q into the second argument, which is what we wanted to prove.

Exercise 2.6 Let $t(n)$ be the maximum expected number of violation tests in a call to `LeaveItThenTakeIt`(R, S), where $|S \setminus R| = n$. Clearly,

$$\begin{aligned} t(0) &= 0, \\ t(n) &= t(n-1) + 1 + p_v t(n-1), \quad n > 0, \end{aligned} \tag{7.4}$$

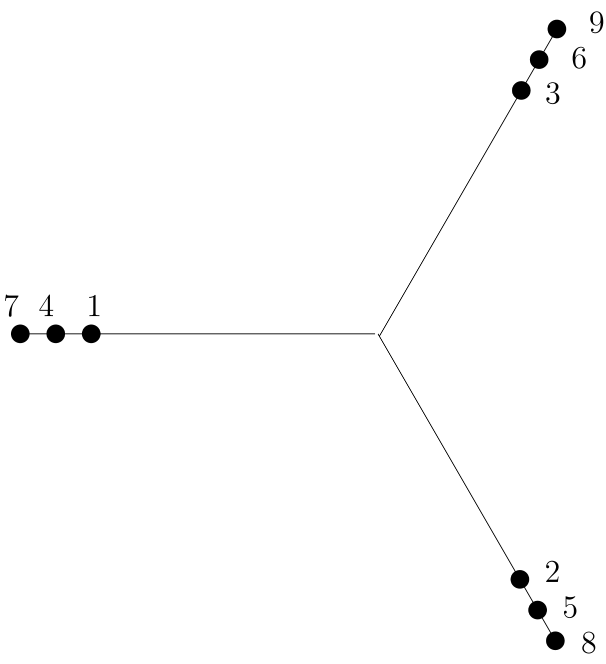


Figure 7.5: Construction for the $\Omega(n^{d+1})$ lower bound

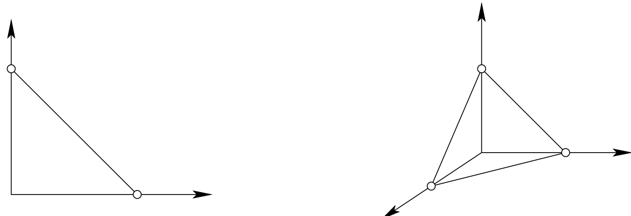


Figure 7.6: The d-dimensional regular simplex embedded in $\mathbb{R}^{d+1}$ for $d = 1$ (left) and $d = 2$ (right).

where p_v is the maximum probability for the event '$p \notin \bigcirc(F, F)$'. Since we definitely have $p_v \leq 1$ we obtain the recursion $t(n) \leq 2t(n-1) + 1$ with $t(0) = 0$, which solves to $t(n) \leq 2^n - 1$. It follows $t_k(n) \leq t(n) \leq 2^n - 1$.

To show that this is best possible, we take S to be the vertices of the *regular simplex* in $\mathbb{R}^d$, and show that algorithm $\texttt{LeaveItThenTakeIt}(R, S)$ needs exactly $2^{|S \setminus R|} - 1$ violation tests for any $R \subseteq S$.

As we will prove below, the vertices S of the d-dimensional regular simplex have the property that for any $R \subseteq S$, $\bigcirc(R, S)$ does not contain any point from $S \setminus R$. Consequently, $p_v = 1$ in equation (7.4), regardless of the outcome of the random choices. It follows that $t(n) = 2t(n-1) + 1$ which together with $t(0) = 0$ solves to $2^n - 1$.

It remains to state what a regular simplex is and to verify that it indeed has the property we have used above. The *d-dimensional regular simplex* is the convex hull of the $d + 1$ standard

basis vectors $S_d := \{e_1, \ldots, e_{d+1}\}$, see Figure 7.6. Observe that the points S_d and hence their convex hull all lie in the d-dimensional hyperplane

$$H := \{x \in \mathbb{R}^{d+1} \mid \sum_{i=1}^{d+1} x_i = 1\}.$$

So if we think of H as $\mathbb{R}^d$, we see that the regular simplex is indeed a d-dimensional object. (We could have defined it directly in $\mathbb{R}^d$, but the resulting formulas for the vertices would be uglier than the given representation in $\mathbb{R}^{d+1}$.)

Fix a set $F = \{e_{i_1}, \ldots, e_{i_k}\} \subseteq S_d$. We want to show that no point in $S_d \setminus F$ is contained in $\bigcirc(F, F)$. The case $|F| \leq 1$ is obvious; so let us assume from now on that F consists of at least two points. In this case, we can use the result from Exercise 2.3(ii) to calculate the center and radius of $\bigcirc(F, F)$ explicitly:

The matrix M (whose columns are the points $e_{i_j} - e_{i_1}$) consists of d rows: one full of -1's, and $d-1$ rows with exactly one 1 at different places and nothing but zeroes elsewhere. From this we immediately get

$$2M^TM = \begin{bmatrix} 4 & 2 & \cdots & 2 \\ 2 & 4 & \ddots & \vdots \\ \vdots & \ddots & \ddots & 2 \\ 2 & \cdots & 2 & 4 \end{bmatrix} \qquad \text{and} \qquad m = \begin{bmatrix} 2 \\ \vdots \\ 2 \end{bmatrix}.$$

The solution of the system $2M^TMx = m$ is $x = (1/k, \ldots, 1/k)$ as you can easily verify, and therefore the entries of $c' = Mx$ read

$$c'_j = \begin{cases} -(k-1)/k, & \text{if } j = i_1, \\ 1/k, & \text{if } j \neq i_1 \text{ and } e_j \in F, \\ 0, & \text{otherwise.} \end{cases}$$

Finally, $c = c' + e_{i_1}$ is the zero vector with $1/k$ in entry i iff $e_i \in F$.[1] For a point $e_j \in F$ we thus get

$$\|e_j - c\|^2 = (1 - 1/k)^2 + (k-1)\,\frac{1}{k^2} = \frac{k-1}{k},$$

that is, the squared radius of $\bigcirc(F, F)$ is $(k-1)/k$. On the other hand, a point $e_j \notin F$ has distance

$$\|e_j - c\|^2 = 1^2 + k\,\frac{1}{k^2} = \frac{k+1}{k},$$

to c and is thus outside $\bigcirc(F, F)$.

Exercise 2.7

Induction on $|S \setminus R|$. If $R = S$, the algorithm returns $F = R$ which is obviously correct. If $|S \setminus R| > 0$, we can assume by the induction hypothesis that the set F returned by the first recursive call is the unique basis of $(R, S \setminus \{p\})$ (if $\beta = 0$) or $(R \cup \{p\}, S)$ if $\beta = 1$. By

[1] So the center of $\bigcirc(F, F)$ is the center of gravity of the points F, something you probably would have guessed beforehand.

Lemma 2.22, F has no loose points and no violators w.r.t. the pair it came from. If no second recursive call happens, we have made sure that F also has no loose elements and no violators w.r.t. (R, S), so F is the basis of (R, S), again by Lemma 2.22.

If there is a second recursive call, we can again inductively assume that it returns the basis of its respective pair. By Observation 2.19, this basis (which we return) must be the basis of (R, S).

Exercise 2.8
Let X be a random variable. The Markov inequality tells us that

$$\text{prob}(X \geq 2E(X)) \leq \frac{1}{2}.$$

It follows that with probability smaller than $1/2$, `LeaveItThenTakeIt` performs more than $2c_{d+1}n$ violation tests. We install a counter, and if the algorithm has not yet terminated after $2c_{d+1}n$ violation tests, we simply abort it and start from scratch. We need to do this with probability smaller than $1/2$.

We require more than $2Kc_{d+1}n$ violation tests if and only if we abort at least K times. Because individual runs of the algorithm are independent, this happens with probability smaller than

$$\frac{1}{2^K},$$

from which the claimed bound follows.

Exercise 2.9 This is covered in detail in the book *Computational Geometry: Algorithms and Applications* by M. de Berg, M. van Kreveld, M. Overmars and O. Schwarzkopf [1].

7.3 Solutions to Chapter 3

Exercise 3.1 Let E be the expected number of visited vertices. The probability that we need to visit only 4 vertices is $\frac{1}{2}$ (after visiting the first 3, we get to the sink with probability $\frac{1}{2}$, and with the same probability, we stay on the cycle of length 6. After taking 2 more steps, we are in the same situation as before. This gives

$$E = 4 \cdot \frac{1}{2} + (E + 2) \cdot \frac{1}{2},$$

so $E = 6$.

Now we prove that for any other USO the expected number of visited vertices, starting from the source, is at most 6. First we claim that (up to symmetry) there is only one USO of the 3-cube which is cyclic. If cycles occur, they must be of length 6: a cycle of length 4 would be a cycle in a 2-dimensional face, which is not possible in a USO. Also, a cycle of length 8 would mean that there is no global sink. It is also clear that no odd cycles can occur, because the cube graph is bipartite.

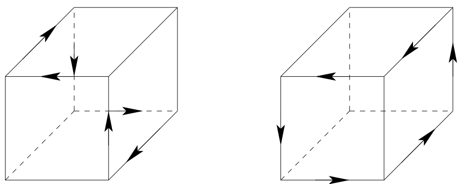


Figure 7.7: The two "possible" cycles of the 3-cube.

Thus, the only possibility is to have some cycle of length 6, and there are (up to symmetries) two ways how such a cycle can look like, see figure 7.7.

The first one gives the USO of the exercise. The second one is not possible, since there is an edge which splits this 6 cycle into two squares. Under any orientation of this edge, we would get a 4-cycle which is not possible in a USO.

Let's focus on the number of vertex evaluations. In order to get more than 6 on average, for some acyclic USO, we must have a path longer than 6 (see Figure 7.8). If the longest path has

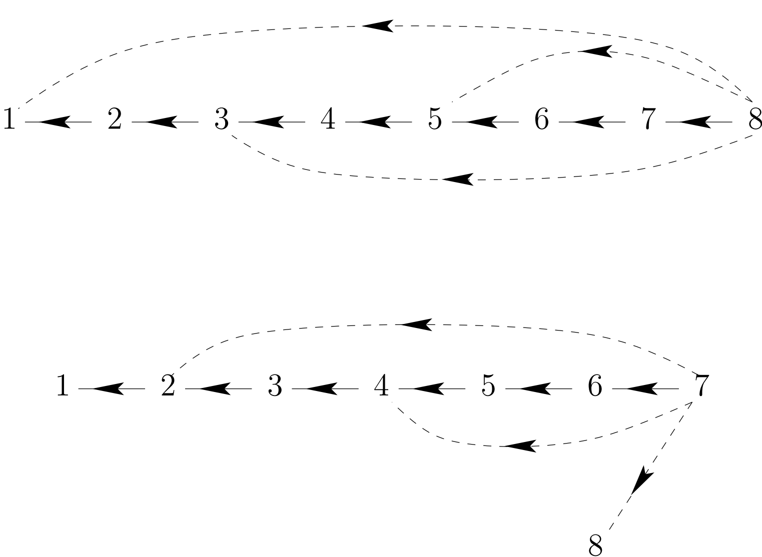


Figure 7.8: Assuming paths of length 8 or 7 from the source

length 8, the source has one edge to the vertex labeled 7 in the figure, and two more outgoing edges. These edges can go only to the vertices denoted by 5,3,1 in the figure (otherwise we would have an (undirected) odd cycle).

It follows that a path of length at most 4 is chosen with probability at least $1/3$ (namely, if the edge going to 3, or to 1—one of them must exist—is chosen). On the other hand, the length-8 path has probability $1/3$ at most, because it can only occur if the edge to 7 is chosen. As all other paths have length 6 at most, it follows that the expected number of vertex evaluations is at most 6.

The case where the longest path has length 7 can be done similarly, see Figure 7.8 (bottom). With probability at least $1/3$, there is a path of length at most 5, starting with the edge from the source to 4 or 1—one of them must exist. As before, the path of length 7 has probability $1/3$

at most, and together with all other paths being of length at most 6, the claim follows.

Exercise 3.2

Assume that all 2-dimensional faces have unique sinks, but that the orientation is not a USO. Consider a face F of smallest dimension which does not have a unique sink. Because there are no cycles, F must have at least two sinks X_1, X_2. Also, they must be antipodal to each other on F, because otherwise, there would be two sinks in some facet of F, contradicting our choice of F. This also implies that F has no other sinks than X_1, X_2, and that every facet of F contains either X_1, or X_2.

Now consider any vertex Y in F which is different from X_1 and X_2. Because Y is not a sink in F, is has some outgoing edge in F. Actually, it must have even two outgoing edges, because otherwise, Y would be a facet sink, meaning that there is some facet of F with two sinks, Y and one of X_1 and X_2, again contradicting our choice of F. Only one of the two outgoing edges can go to X_1 or X_2, because $|X_1 \oplus X_2| \geq 3$ (recall that the 'bad' face F cannot be a 2-face). This means, we can reach a non-sink Y', following an outgoing edge from Y. Continuing with this argument from Y', we can carry on until we see some vertex for the second time. By that moment, however, we have constructed a directed cycle, a contradiction.

Exercise 3.3

(i) Let i be the fixed label. Split C_n into the two facets (subcubes) C^1 and C^2, where C^1 consists of the vertices not containing i, while C^2 collects all vertices containing i.

Any cube face F is either completely in one of the two facets, or it is divided among the two facets. In the first case, the orientation in F is not affected by relabeling (as F does not contain edges labeled i).

In the second case, $F^1 = F \cap C^1$ and $F^2 = F \cap C^2$ are facets of F. Let X_1, X_2 be their sinks, where we assume without loss of generality that X_1 is the sink of F. It easily follows that after relabeling, X_2 is the unique sink of F.

(ii) First assume s has the required property; take a face $[I, J]$. We have to show that $[I, J]$ has a unique sink.

Consider the restricted map $\tilde{s} : [I, J] \to 2^{J \setminus I}$,

$$\tilde{s}(X) = s(X) \cap (J \setminus I).$$

We first show that $\tilde{s}$ is injective. Namely, assume there are $X, Y \in [I, J]$ with $\tilde{s}(X) = \tilde{s}(Y)$. For these vertices we get

$$(s(X) \oplus s(Y)) \cap (J \setminus I) = \emptyset,$$

and since $X \oplus Y \subset J \setminus I$, this contradicts the assumption.

It follows that $\tilde{s}$ is bijective. In particular, there is exactly one sink $X \in [I, J]$ (having $\tilde{s}(X) = \emptyset$).

Now assume s does not have the property and let X, Y witness its failure. But then, since $(X \oplus Y) \cap (s(X) \oplus s(Y)) = \emptyset$, in the face $[X \cap Y, X \cup Y]$ both X and Y have the same outmap value $S = s(X) \cap (X \oplus Y) = s(Y) \cap (X \oplus Y)$. Therefore, after reorienting the edges with label in S, the face $[X \cap Y, X \cup Y]$ has two sinks X, Y, i.e. it is not a USO, and s is not the outmap of a USO.

Exercise 3.4 Using the product algorithm, we get

$$t(n) = t(2)^{\lfloor n/2 \rfloor} t(1)^{n \bmod 2}.$$

Plugging in $t(2) = 3$, the bound follows.

7.4 Solutions to Chapter 4

Exercise 4.1

By interchanging the column player's (adversary's) strategies, we may assume that $a \leq b$. We first argue that the only interesting case is where the relations between the entries of M are as follows.

$$\begin{pmatrix} a & < & b \\ < & & > \\ c & > & d \end{pmatrix}. \tag{7.5}$$

Let's get rid of the other cases. Assume that one of the four relations is an equality, say $a = b$ (the other case are completely symmetric). Then, if $c < d$, the unique optimal strategy of the column player is $\tilde{y} = (0, 1)$, and if $c > d$, it's $\tilde{y} = (1, 0)$. If $c = d$, the column player can choose any strategy.

If $c < d$, the row player (algorithm player) therefore chooses the row with smaller entry in the second column: if $b < d$, she plays $\tilde{x} = (1, 0)$, if $b > d$, it's $\tilde{x} = (0, 1)$. If $b = d$, she can play any strategy. The case $c > d$ is similar. Even if $c = d$, the relation between b and d (equivalently, between a and c) decides the possible optimal strategies for the row player.

It remains to deal with the case where all four relations are strict inequalities, but for example $a < b, c < d$. As before, the column player then uniquely chooses $\tilde{y} = (0, 1)$, and the rows player's optimal behavior follows.

Under the relations in (7.5), it is not difficult to show that the unique optimal solution of (LP$_M$) is assumed when both inequalities involving u are satisfied with equality. This implies

$$\tilde{y} = \left(\frac{d-b}{a-c+b-d}, \frac{a-c}{a-c+b-d} \right).$$

Similarly (by solving the dual), we get

$$\tilde{x} = \left(\frac{d-c}{a-c+b-d}, \frac{a-b}{a-c+b-d} \right).$$

The value of the game is

$$\frac{\det(M)}{a-c+d-b}.$$

Exercise 4.2

(i) Let $\Phi = g_M(\tilde{x})$ be the value of the game. Given algorithm $\mathcal{A}(x)$, the best response of the adversary (the runtime of $\mathcal{A}(x)$ against its worst random input) is $g_M(x)$ by definition.

Because $\tilde{x}$ minimizes $g_M(x)$ over all x, we get $\Phi \leq g_M(x)$, so $\mathcal{A}(x)$ is not faster than Φ on some random input. But then it is also not faster than Φ on some concrete input I_j (recall the arguments from the proof of Yao's Theorem).

(ii) the runtime of $\mathcal{A}(\tilde{x})$ is Φ if the adversary plays optimally. It follows that the runtime is never more than Φ.

Exercise 4.3

The gameshow host has six possible strategies; he has to choose where to put the car (three choices), and for each of these choices specify which door to open in case your first choice is the door that hides the car (otherwise, his answer is unique). Therefore, we can encode the host's strategies with a pair of distinct numbers between 1 and 3.x

You have six strategies as well; there are three choices for the door you pick first, and for each of these, you may switch or not. We can encode these strategies by a number and a letter in $\{Y, N\}$ (Y means you switch). Here, whether you switch or not does not depend on the door the host opens. We can include this, but it wouldn't change anything.

The payoff matrix therefore looks as follows (the host is the row player with the goal of minimizing your payoff, you are the column player and want to maximize the payoff).

	$1Y$	$1N$	$2Y$	$2N$	$3Y$	$3N$
12	0	1	1	0	1	0
13	0	1	1	0	1	0
21	1	0	0	1	1	0
23	1	0	0	1	1	0
31	1	0	1	0	0	1
32	1	0	1	0	0	1

Consider the resulting linear program (LP$_M$). Adding up the first six inequalities, we get

$$4y_1 + 2y_2 + 4y_3 + 2y_4 + 4y_5 + 2y_6 \geq 6u,$$

and plugging in the constraint

$$\sum_{i=1}^{6} y_i = 1,$$

we get

$$2 + 2(y_1 + y_3 + y_5) \geq 6u.$$

Because $y_1 + y_3 + y_5 \leq 1$, $u \leq 2/3$ follows. On the other hand, value $u = 2/3$ is attainable, by setting

$$y_1 = y_3 = y_5 = \frac{1}{3}, \quad y_2 = y_4 = y_6 = 0.$$

It follows that these setting define an optimal strategy: initially, choose between the doors uniformly at random, but then switch in any case.

Exercise 4.4

For this, we need a slightly more formal view of deterministic algorithms for this scenario. We assume that the vertices are labeled, and any deterministic algorithm specifies the label of the next vertex to go to, depending on the sequence of vertices seen so far.

We use Yao's Theorem with the uniform distribution on all inputs, i.e. with each of the $n!$ acyclic orientations of K_n (see Exercise 1.3(iii)) appearing with probability $1/n!$.

Assume we are at the vertex of rank i (the rank is the order in the unique topological sorting corresponding to the orientation, where the sink has rank 1). Because the algorithm has no information about vertices of smaller rank and the orientations of edges between them, any smaller-rank vertex of *fixed* label (the one the algorithm moves to next) still has random rank among the vertices of ranks $1, \ldots, i-1$, averaged over all acyclic orientations. This implies that the expected number $f(i)$ of steps, starting from the vertex of rank i, satisfies the recurrence

$$f(i) = 1 + \frac{1}{i-1} \sum_{j=1}^{i-1} f(j),$$

which is exactly the recurrence we also proved for Randomix's strategy against a *fixed* orientation.

7.5 Solutions to Chapter 5

Exercise 5.1

(i) Let p_i denote the probability that a random walk starting at i ends in N. We know that $p_0 = 0$, $p_N = 1$, and

$$p_i = p \cdot p_{i-1} + q \cdot p_{i+1}, \quad p + q = 1. \tag{7.6}$$

If $p = 0$, then $p_i = 1, i \neq 0$, while for $q = 0$, we get $p_i = 0, i \neq N$, so let's assume that both p and q are nonzero.

Let

$$b_i = q \cdot p_i - p \cdot p_{i-1}. \tag{7.7}$$

We obtain that

$$b_i - b_{i-1} = q \cdot p_i - p \cdot p_{i-1} - q \cdot p_{i-1} + p \cdot p_{i-2} = 0,$$

using that $p + q = 1$ and (7.6). Since $b_1 = q \cdot p_1$ we get from (7.7) that

$$p_i = \frac{p}{q} p_{i-1} + p_1,$$

which gives that

$$p_i = \left(1 + \frac{p}{q} + \cdots + \left(\frac{p}{q}\right)^{i-1}\right) p_1.$$

So

$$p_i = \frac{\left(\frac{p}{q}\right)^i - 1}{\frac{p}{q} - 1} p_1, \quad p \neq \frac{1}{2},$$

and

$$p_i = i \cdot p_1, \quad p = q = \frac{1}{2}.$$

Using that

$$1 = p_N = \frac{\left(\frac{p}{q}\right)^N - 1}{\frac{p}{q} - 1} p_1, \quad p \neq \frac{1}{2}$$

and

$$1 = p_N = N \cdot p_1, \quad p = q = \frac{1}{2},$$

we get that

$$p_i = \frac{\left(\frac{p}{q}\right)^i - 1}{\left(\frac{p}{q}\right)^N - 1}, \quad p \neq \frac{1}{2},$$

and

$$p_i = \frac{i}{N}, \quad p = q = \frac{1}{2}.$$

(ii) Here is a sketch: we proceed as above, but with

$$b_i = p \cdot E_i - q \cdot E_{i-1} + i.$$

In solving this, the following formula comes in handy:

$$\sum_{i=1}^{i=n} i \cdot x^i = \frac{n \cdot x^{n+2} - (n+1)x^{n+1} + x}{(x-1)^2}.$$

Exercise 5.2

One can prove this by induction on $|X \oplus Y|$. If $|X \oplus Y| = 1$ than there is a directed edge between Y and X. Otherwise let $|X \oplus Y| = n$. The cube $[X, Y]$ has X as a sink, so there is an outgoing edge i from Y. By the inductive hypothesis there is a directed path of length $n - 1$ from $Y \oplus \{i\}$ to X. Combined with the edge $(Y, Y \oplus \{i\})$, we get the desired directed path of length n from Y to X.

Exercise 5.3 The face $[A, B]$ corresponds to all bit patterns which have a 1 at positions in A and a 0 at positions not in B. Any sink of $[A, B]$ must have minus-signs at all positions in $B \setminus A$. This means, any sink of $[A, B]$ corresponds to the same partial bit/sign pattern with a bit or sign at any position. The fact that the pattern is completable in a unique way then translates to the fact that $[A, B]$ has a unique sink.

Exercise 5.4 Looking at the automaton in Figure 5.5, it becomes clear that every vertex X can be decomposed into blocks of the three types

$$\begin{array}{|c|}\hline 1 \\ \hline + \\ \hline \end{array}, \quad \begin{array}{|c|c|}\hline 0 & 0 \\ \hline + & - \\ \hline \end{array}, \quad \text{and} \begin{array}{|c|c|}\hline 1 & 0 \\ \hline - & - \\ \hline \end{array}.$$

At level 1, we have only one block of the first type, so there are

$$\frac{n-1}{2}$$

blocks of the two other types. Since each of them contributes either a 1 or a $+$, we get

$$\#_X(1) + \#_X(+) = \frac{n-1}{2} + 2 = \frac{n+3}{2},$$

as desired.

Now consider a vertex X at level k. n being odd implies that k is odd as well. Then we have k blocks of the first type and $(n-k)/2$ of the second and third type. It follows as before that

$$\#_X(1) + \#_X(+) = \frac{n-k}{2} + 2k = \frac{n+3k}{2}.$$

Exercise 5.5

(i) For a face $F = [A, B]$, we define $d(F) = |B \setminus A|$ and $b(F) = |B|$.

We prove by double induction over $(d(F), b(F))$ that every face has a unique sink. This obviously holds for vertices $(d(F) = 0)$ and if $d(F) = b(F)$: in this case, the face is of the form

$$[\emptyset, B]$$

and has $\emptyset$ as its unique sink, as one can easily conclude from the definition of the outmap s.

Now assume $F = [A, B]$ with $b(F) > d(F) > 0$. Let $i = \max(B)$. There are two cases.

Case (a) $i \in B \setminus A$. Then

$$s(X \cup \{i\}) = \{1, \ldots, i\} \setminus s(X) \supseteq \{i\}, \quad X \in [A, B \setminus \{i\}].$$

It follows that the sinks of $[A, B]$ are exactly the sinks of $[A, B \setminus \{i\}]$, and by induction, we know that there is a unique such sink.

Case (b) $i \in A$. Then

$$s(X \cup \{i\}) = \{1, \ldots, i\} \setminus s(X), \quad X \in [A \setminus \{i\}, B \setminus \{i\}].$$

It follows that the sinks in $[A, B]$ are exactly the *sources* in $[A \setminus \{i\}, B \setminus \{i\}]$. Because the latter face and all its subfaces have unique sinks by induction, we have a USO on $[A \setminus \{i\}, B \setminus \{i\}]$, which implies that the face also has a unique source (this easily follows from Exercise 3.3.

(ii) We inductively prove that there is a directed Hamiltonian path through all vertices in $[\emptyset, \{1, \ldots, i\}]$, starting in $\{i\}$ and ending in $\emptyset$. The case $i = 1$ is clear.

For $i > 1$, we inductively assume that there is a path from $\{i-1\}$ to $\emptyset$ in the face $[\emptyset, \{1, \ldots, i-1\}]$. As in case (b) of part (i), we can conclude from this the existence of an 'inverse' directed Hamiltonian path in $[\{i\}, \{1, \ldots, i\}]$ from $\{i\}$ to $\{i-1, i\}$. Concatenating

these paths by inserting the directed edge $(\{i-1, i\}, \{i-1\})$—which exists by definition of s— yields the desired path from $\{i\}$ to $\emptyset$ in $[\emptyset, \{1, \ldots, i\}]$.

(iii) By definition, any vertex containing n (vertex in the upper facet) also has n in its outmap value. It follows that from every such vertex, we have a chance of $1/n$ at least to go to a vertex not containing n (vertex in the lower facet), and from the lower facet there is no way to get back to the upper facet. It follows that after an expected number of at most n steps, we reach a vertex in the lower facet.

Because in the lower facet, the same holds with respect to its upper facet (vertices containing $n-1$), we can iterate this argument and obtain a bound of

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

for the expected number of steps taken by the random walk. The expected number of vertex evaluations is one larger.

Exercise 5.6 Any pair of USOs of the $(n-1)$-cubes $C^1 = [\emptyset, \{1, \ldots, n-1\}]$ and $C^2 = [n, \{1, \ldots, n\}]$ can be extended to a USO of the n-cube $[\emptyset, \{1, \ldots, n\}]$ by combing all edges in direction n (there are two ways of doing this). This gives that

$$u_n \geq 2u_{n-1}^2.$$

Using that $u_0 = 1$ we get that $u_n \geq 2^{2^n-1}$.

Exercise 5.7
This is a trivial consequence of the definitions.

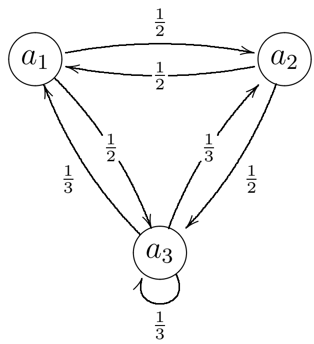
Exercise 5.8

(i) The following Markov chain is non-irreducible and e.g. $(0, 0, 1), (1, 0, 0)$ are stationary distributions.

(ii) The following Markov chain is irreducible and 2-periodic. The initial distribution $(1, 0)$ does not converge to the stationary distribution $(\frac{1}{2}, \frac{1}{2})$.

(iii) The following Markov chain is irreducible, aperiodic and non-symmetric. The stationary

distribution $(\frac{2}{7}, \frac{2}{7}, \frac{3}{7})$ is not the uniform distribution.



Exercise 5.9 Because we reorient only edges in direction i, we know that

$$(X \oplus Y) \cap (s(X) \oplus s(Y))$$

and

$$(X \oplus Y) \cap (s^{'}(X) \oplus s^{'}(Y))$$

can differ only in i.

If the former evaluates exactly to $\{i\}$, X and Y are strongly in phase, so either both their incident edges in direction i get flipped, or none of them. It follows that also the latter expression evaluates to $\{i\}$. If the former contains some $j \neq i$, j will also be contained in the latter. In both cases, the latter is nonempty, and the USO property follows.

Exercise 5.10

The idea is to use Theorem 5.25 and proceed similarly as we generated a USO uniform at random.

The vertices of the graph of our Markov chain will be the stations of the city (we assume that this graph is connected). Let N be the number of stations. Whenever there is a tram between two stations A, B we choose $\frac{1}{N}$ as the transition probability in both directions. In addition, we put loops into every vertex A, where the transition probability for the loop will be chosen such that the probabilities on all outgoing edges of A sum up to 1. This Markov chain is homogeneous, irreducible, aperiodic and symmetric. This ensures that in the long run, we will be at any station with the same probability.

7.6 Solutions to Chapter 6

Exercise 6.1

They key observation one needs to make is that the operator $\oplus$ is associative: both the sets

$$A \oplus (B \oplus C)$$

and

$$(A \oplus B) \oplus C$$

consist exactly of the elements which are not in exactly two of the sets. Using this, the statements are easy.

Bibliography

[1] Mark de Berg, Marc van Kreveld, Mark Overmars, and Otfried Schwarzkopf. *Computational Geometry: Algorithms and Applications*. Springer-Verlag, Berlin, Germany, 2nd edition, 2000.

[2] K. Fischer and B. Gärtner. The smallest enclosing ball of balls: combinatorial structure and algorithms. *International Journal of Computational Geometry and Applications (IJCGA)*, 2004, to appear.

[3] B. Gärtner. The Random-Facet simplex algorithm on combinatorial cubes. *Random Structures & Algorithms*, 20(3), 2002 (preliminary version at RANDOM'98).

[4] B. Gärtner and E. Welzl. Explicit and implicit enforicing o– randomized optimization. In *Lecture Notes of the Graduate Program Computational Discrete Mathematics*, volume 2122 of *LNCS*, pages 26–49. Springer-Verlag, 2001.

[5] G.R. Grimmet and D.R. Stirzaker. *Probability and Random Processes*. Oxford Science Publications, 1982.

[6] G. Kalai. A subexponential randomized simplex algorithm. In *Proc. 24th annu. ACM Symp. on Theory of Computing.*, pages 475–482, 1992.

[7] W. Ludwig. A subexponential randomized algorithm for the simple stochastic game problem. *Information and Computation*, 117(1):151–155, 1995.

[8] J. Matoušek, M. Sharir, and E. Welzl. A subexponential bound for linear programming. *Algorithmica*, 16:498–516, 1996.

[9] J. Matoušek and U. Wagner. Almost uniform smapling of unique sink orientations. `http://www.ti.inf.ethz.ch/ew/workshops/01-lc/problems/node6.html`

[10] Walter Morris. Randomized principal pivot algorithms for P-matrix linear complementarity problems. *Math. Programming, Ser. A*, 92:285–296, 2002.

[11] R. Motwani and P. Raghavan. *Randomized Algorithms*. Cambridge University Press, New York, NY, 1995.

[12] K. Mulmuley. *Computational Geometry: An Introduction Through Randomized Algorithms*. Prentice Hall, Englewood Cliffs, NJ, 1993.

[13] Alan Stickney and Layne Watson. Digraph models of bard-type algorithms for the linear complementary problem. *Mathematics of Operations Research*, 3:322–333, 1978.

[14] Tibor Szabó and Emo Welzl. Unique sink orientations of cubes. In *Proc.* 42nd *IEEE Symp. on Foundations of Comput. Sci.*, pages 547–555, 2000.

[15] Bernhard von Stengel. Computing equilibria for two-person games. In *Handbook of Game Theory*, volume 3. North-Holland, Amsterdam, 2002.

[16] Emo Welzl. Smallest enclosing disks (balls and ellipsoids). In H. Maurer, editor, *New Results and New Trends in Computer Science*, volume 555 of *Lecture Notes Comput. Sci.*, pages 359–370. Springer-Verlag, 1991.