Randomized Algorithms

August 28, 2014

Contents

1 Preliminaries

In this chapter, the usage of randomization is motivated and some basic tools, notations and concepts are introduced.

1.1 Why Randomization?

For starters, consider the following (very applied) example:

Example (Cooking Dinner) *Assume you want to make steak for dinner. Looking in the cook book, it tells you to roast the steak in the pan, three minutes on each side. For your seafood-fancy friend you also want to fry shrimps, lets say two dozens. Of course, you could proceed similar with the shrimps as with the steak: Fry each individually for a certain time frame on each side. But as you want to get done today, you just throw all of them in the pan and stir occasionally; trusting that by chance each shrimp will get fried on each side sufficiently if frying long enough.*

The cooking dinner example illustrates the basic concepts behind randomization: It is often much *easier* to perform some random action than to follow a deterministic algorithm (occasionally stirring versus frying individually), it can *save time* to complete the task, and if the result is not perfect up to now, just repeat (stir again, wait a bit). During the course of the lecture, we will consider randomized algorithms and data structures, which

- exhibit smaller run times/space consumption than deterministic algorithms in practise,
- have theoretical run times beyond deterministic lower bounds,
- are easier to implement/more elegant than deterministic strategies,
- allow for treating run time against output quality,
- provide a good strategy for games or search in unknown environments.

Furthermore, we will have a look at randomized algorithms for online problems, where the complete input data is not given in advance but revealed in an online fashion. Again, randomized methods will be shown to be as good as or even superior to deterministic algorithms in several aspects. Finally, randomization will also be shown to be a useful tool for (non-constructive) existence proofs.

1.2 Some Basic Stochastics

1.2.1 Expected Value

In a random experiment, we consider a random variable X which can take discrete numerical values. The expected value of X assuming the random experiment would be conducted infinitely often is denoted by $E(X)$ and formally defined in the following.

Definition (Expected Value) *Let $x_1, \cdots, x_k$ be the finite set of values a random variable X can take, and $p_1, \cdots, p_k \in [0,1]$ the respective probabilities of realization $p_i = P(X = x_i)$. Then the expected value of X is:*

$$E(X) := \sum_{i=1}^{k} x_i \cdot p_i$$

Example (Rolling a Dice) *If X describes the score of a fair dice, the set of values X can take is $\{1,2,3,4,5,6\}$ and the probabilities are $p_1 = p_2 = \cdots = p_6 = 1/6$. So the expected value $E(X)$ is $1/6 \sum_{i=1}^{6} i = 21/6 = 3.5$*

In the context of analyzing randomized algorithms, the expected value might be deceiving, though. Consider instead of a conventional dice a dice which has a 1 on three sides and a 6 on the other three. The expected value for this dice will be 3.5 as well, while X never will take a value close to 3.5. We come back to that point when investigating concentration bounds (see Section 1.4.1) and when analyzing QuickSort (see Section 2.2).

1.2.2 First Success

Often it will be of interest when a certain event $(X = x_i)$ will happen the first time. We denote with p the probability for success (so $p = P(X = x_i)$) and with q the probability of the counterevent, so $q = P(X \neq x_i)$. Obviously it yields $q = 1 - p$. Now in every round we perform our random experiment and assign to X the respective value. We assume, that the experiments are independent of each other, i.e. the outcome of the experiment in round j does not have an influence of the outcome of the experiment in later rounds. Now let the random variable Y denote the number of rounds until $X = x_i$ for the first time. Obviously, for Y being equal to some $j \in \mathbb{N}$, we had no success in the previous $j - 1$ rounds, and then success in the j^{th} round. The probability for no success in $j - 1$ rounds can easily be expressed as q^{j-1}, the probability for success in the last round simply by p. So altogether we get $P(Y = j) = (1 - p)^{j-1}p$. Now applying our formula for the expected value, we get:

$$E(Y) = \sum_{j=1}^{\infty} j \cdot P(Y = j) = \sum_{j=1}^{\infty} j \cdot (1-p)^{j-1} p = \frac{p}{1-p} \sum_{j=1}^{\infty} j \cdot (1-p)^j = \frac{p}{1-p} \cdot \frac{1-p}{p^2} = \frac{1}{p}$$

We manifest this in a lemma for later reference.

Lemma 1.1. *If a random experiment with success probability $p > 0$ is performed repeatedly with the outcomes being independent of each other, then the expected number of rounds until the first success is* $^1/_p$.

Lets have a look at some illustrating examples.

Example (Rolling a Dice again) *In many games one needs to throw a 6 to get started. The probability of getting a 6 with the first trial is $p = {}^1/_6$. So the expected number of rounds until the first 6 shows up is $^1/_p = 6$. Of course, this is the same expected value as for waiting for the first 1 or 2 or 3 or 4 or 5.*

Example (Throwing a Coin) *With a fair coin the probability of the coin showing 'head' is $^1/_2$, so expectedly we see 'head' the first time after 2 rounds. If e.g. the probability for 'head' would be only a quarter, we have to wait on average 4 rounds for the first success.*

1.2.3 Linearity of the Expected Value

Another important aspect of the expected value of a random variable is its linearity. So if we consider multiple random variables $X_1, X_2, \cdots, X_k$ the expected value of the sum of those is equal to the sum of their expected values, i.e.

$$E(X_1 + X_2 + \cdots + X_k) = E(X_1) + E(X_2) + \cdots + E(X_k)$$

Exercise. Prove the linearity of the expected value.

As a consequence it is possible to break events down in the union of simpler events and compute the expected value for the complicated setting via accumulation of expected values for the simpler events.

Example (Guessing Cards) *Lets say you have deck of n distinct cards. You want to demonstrate your prediction skills by turning over one card at a time, but claiming its identity in advance.*

As this example sounds like a fun game for (lame) parties, lets first assume you are already completely drunk and you cannot remember the cards you already turned over. How many cards will you guess correctly on average?
Let $X_1, \cdots, X_n$ be a set of random variables with X_i being 1 if you guessed card number i correct, and 0 otherwise. With X we denote the random variable describing how many cards you guessed correctly in total. So $E(X) = E(X_1 + X_2 + \cdots + X_n)$. Because of the linearity of $E(X)$ we can rewrite the formula as $E(X) = \sum_{i=1}^{n} E(X_i)$. It remains to analyse $E(X_i)$ for every possible i. This can easily by done by applying the definition of the expected value and inserting the probability of guessing a single card correctly which obviously is $1/n$. Hence we get $E(X_i) = 1 \cdot 1/n + 0 \cdot (n-1)/n = 1/n$. Altogether we can compute the expected value of X as $E(X) = \sum_{i=1}^{n} 1/n = 1$. This means, on average, you just guess one card correctly. Surprisingly the term does *not* depend on the total size of the card deck. So no matter if you have 52 cards or 20 million, the outcome will expectedly be the same.

What about not being totally drunk and therefore being able to remember which cards you already turned around?
Let X and X_i be defined as before. What changes is the pool of cards from which you still can choose in each round. In fact, in round i, there are only $n - i + 1$ cards left. Hence the expected value adapts:

$$E(X_i) = P(X_i = 1) = \frac{1}{n - i + 1}$$

Summing up again, we get:

$$E(X) = \sum_{i=1}^{n} E(X_i) = \sum_{i=1}^{n} \frac{1}{n - i + 1} = \sum_{i=1}^{n} \frac{1}{i} = H(n)$$

Here $H(n)$ is the n^{th} harmonic number. $H(n)$ can be bounded as follows: $\ln(n+1) < H(n) < 1 + \ln(n)$. Therefore speaking in terms of O-Notation we have $E(X) = H(n) \in \Theta(\log n)$. So now (with memory), the expected number of correctly guessed cards is indeed dependent on the total number of cards in the deck and exceeds the value for memory-free guessing for large n significantly.

Example (Collecting Goodies) *Lets assume that if you buy groceries at the local supermarket for more than 10€ you get one of n toys for free (which one you cannot choose). You want to collect all of them. How often do you have to shop there expectedly for this purpose?*

A lower bound is trivial. Obviously you have to go there at least n times to get all toys. But after n visits, the chances that you have already collected all of them are minor. Note, the more toys you already have the smaller the probability to get a new one (and not some other multiple times). If you already have $n - 1$ different toys, the chance to get the last missing one is only $1/n$. So now let X be the variable describing the number of visits at the supermarket before you have collected all toys. Like for the guessing cards problem, it is not easy to compute $E(X)$ straight away; but using the linearity of the expected value we can reduce the complexity. The decomposition we make is into subproblems asking for the number of visits you have to make to go from zero toys to one toy, from one toy to two different toys, from two to three different toys and so on. The respective random variables are called $X_1, X_2, X_3, \cdots$. So what is the expected value of each X_i? If you already have $i - 1$ toys collected, $n - i + 1$ new toys remain. So your chance of success for the next visit is $p = (n - i + 1)/n$. As we have learned in Lemma 1.1, this means we have to wait expectedly $n/(n-i+1)$ rounds for the success, and so this is the value of $E(X_i)$. To get $E(X)$ it remains to sum up over all $E(X_i)$:

$$E(X) = \sum_{i=1}^{n} E(X_i) = \sum_{i=1}^{n} \frac{n}{n-i+1} = n \sum_{i=1}^{n} \frac{1}{n-i+1} = n \sum_{i=1}^{n} \frac{1}{i} = n \cdot H(n)$$

So on average one needs $\Theta(n \log n)$ visits at the supermarket to collect all toys. Of course, this only holds if the chances to get each of the toys are equal.

1.2.4 Conditional Probability

The last concept we need to analyze randomized algorithms later is conditional probability and conditional expectation. In all our previous examples we considered the outcomes of several random experiments to be independent of each other. So if you roll a dice a second time, the chances of the possible scores are not influenced by the result of the first roll at all. But of course, this is not true for all kinds of random experiments. The typical example for changing probabilities is picking balls from an urn.

Example (Urn Model) *In an urn you have a collection of differently coloured balls, e.g. four black ones and four white ones. You cannot see inside the urn, so if you pick a ball its colour is only revealed at the moment you remove it from the urn. There are two different types of urn models, one where you return the balls you picked (possibly under some additional constraint, e.g. only if it was a black one), and the model where a ball once removed from the urn is never reconsidered.*

So what are the chances in an urn with four black (b) and four white (w) balls to pick a white ball? Obviously in the first round its simply $P(w) = {}^4\!/_8 = {}^1\!/_2$. If we picked a ball but do not return it in the urn afterwards, what are the chances of picking a white ball in the second round? Well, it depends which ball we removed from the urn in the first round. If it was a white ball, the chance to pick a white ball again decreased, namely to ${}^3\!/_7$ (only three out of seven balls are white now in the urn). If a black ball was picked in the first round, the probability to pick a white ball in the second round is ${}^4\!/_7$ accordingly. We indicate this by using conditional probability $P(A|B)$, which means the probability for A given that B happened. So for our example we have e.g. $P($w in second round$|$w in first round$) = {}^3\!/_7$. Considering e.g. the fifth round, it might be impossible to pick a white ball, if they all were removed from the urn in the previous rounds, so $P($w in fifth round$|$ w in round 1,2,3,4$) = 0$. In general, the conditional probability can be computed as

$$P(A|B) = \frac{P(A \cap B)}{P(B)}$$

which means the probability that both A and B happen divided by the probability that B happens (of course this makes only sense if $P(B) > 0$). For the expected value we can incorporate conditions as well. So computing the expected value of X given some event Y, the formula adapts as follows.

$$E(X|Y) = \sum_{i=1}^{n} x_i P(X = x_i|Y)$$

We will need this kind of probabilities e.g. for analyzing a randomized algorithm for the MinCut problem 3.2.

1.3 Las Vegas and Monte Carlo Algorithms

1.3.1 Examples and Features

There are two different types of randomized algorithms, which we will introduce by considering the following example.

Example (Drug Detection) *Assume you are a janitor at a high school. It is a typical high school, so half of the n pupils do drugs and hide their stash in their lockers. The dean of the school wants you to break lockers open until you found some drugs, in order to punish the respective pupil (as a warning for all others).*

A deterministic strategy would be to break open one locker after the other in the order of the locker numbers. If all addicts have high locker numbers, you need to break open $n/2 + 1$ lockers. In fact, for every deterministic strategy, if the pupils would find out about your strategy, they could rearrange the drugs such that you would always need $n/2 + 1$ tries. If instead of choosing a deterministic order, you change your strategy to randomly permute the locker numbers and attack them in that order, you also might need to crack $n/2 + 1$ lockers; but maybe you are lucky and have success much earlier. At least the pupils now cannot come up with a bad rearrangement of the drug locations, such that you need $n/2$ tries for sure. We call that strategy *RandA*. If you only have a limited amount of time, lets say to break open k lockers, than the strategy *RandB* is to choose k lockers randomly and look for drugs there. So *RandA* makes sure you always find drugs, but you cannot tell a priori how much time it will take to do so. *RandB* on the other side gives you a fixed time, but maybe in that time you will find no drugs. What are the chances of being lucky using *RandB*? The success probability when choosing a single random locker is $1/2$ (as half of the pupils do drugs). So the chance to pick only drug-free lockers in k rounds is $1/2^k$. Therefore the probability of success is $1 - 1/2^k$.

RandA and *RandB* capture the basic characteristics of Las Vegas and Monte Carlo algorithms. A Las Vegas algorithm always outputs the correct result or guarantees success, but the runtime is random (like for *RandA*). A Monte Carlo algorithm has a chance of error or fail, but the runtime does not depend on chance (like for *RandB*). We will look at two further examples to emphasize the advantages of such randomized algorithms compared to deterministic strategies.

Example (Find Large Number) *Given an array A of size n filled with natural numbers. We seek to find a number which is equal to or larger than the median of these n numbers.*

A deterministic strategy would be e.g. to scan over the array and always remember the maximum seen so far. This would result in an correct output after having looked at each of the n elements. You can improve slightly by stopping already after having inspected the first $n/2$ elements, and output their maximum. Nevertheless, the runtime is still in $\mathcal{O}(n)$. In fact, no deterministic strategy can perform better in the worst case. Now consider the following algorithm:

Algorithm 1: Monte Carlo Find

```
begin
    pick $k$ elements at random;
    find max of these elements;
    return max;
end
```

The probability that a random element is not equal to or larger than the median is $\leq 1/2$. Hence the probability that all k elements are incorrect (and therefore their max) is $\leq 1/2^k$. So if we set k to $c \cdot \log_2 n$ for some positive constant c, we have a chance of error of only n^{-c} and a runtime of $\mathcal{O}(\log n)$. So lets assume we have a million elements in A. If we choose $k = 20$ the chance to fail is smaller than the chance of being struck by a lightning (about 1:1 million), and for $k = 25$ the chance of failure is smaller than the chance of having identical quadruplets (about 1:15 million). At the same time we only spend 20 or 25 operations on computing the result, while the deterministic strategy needs $500,000$. As the probability to fail depends solely on k and not on n, we get the same results considering a billion elements or a quadrillion (while the costs for the deterministic strategy increase, though).

Example (Find Repeated Element) *Given an array A of size $n \geq 4$ filled with $n/2$ distinct elements and $n/2$ repeated elements. Determine the repeated element.*

A naive deterministic strategy would be to sort A and then parse through the sorted array until two consecutive elements are the same. The runtime of this approach would be $\mathcal{O}(n \log n)$. In fact, one can get down to a deterministic runtime of $\mathcal{O}(n)$ because the median can be found deterministically in that time. But we can do even better by using a Las Vegas algorithms:

Algorithm 2: Las Vegas Find Repeated Element

```
begin
    while true do
        pick $i$ u.a.r. in $\{1, \cdots, n\}$;
        pick $j$ u.a.r. in $\{1, \cdots, n\} - \{i\}$;
        if $A[i] = A[j]$ then
            return $A[i]$;
end
```

Obviously, the algorithm is correct. It only returns an element which occurs at least twice in the array, hence it has to be the repeated element. What is the expected runtime of this Las Vegas algorithm? The success probability in a single round can be expressed as

$$P(success) = \frac{n/2}{n} \cdot \frac{n/2 - 1}{n-1} \approx \frac{1}{4}$$

for n large enough. So according to Lemma 1.1 we only need to sample 4 pairs on average before having a result, which is clearly constant time. Nevertheless – in theory – the algorithm could run forever.

1.3.2 From Las Vegas to Monte Carlo and Back

Any Las Vegas algorithm can be easily turned into a Monte Carlo algorithm, the other way around requires some more effort as we will explore in the following.

Given a Las Vegas algorithm with expected running time at most $f(n)$. To get a fixed runtime as required for a Monte Carlo algorithm, we just stop the computation after $\alpha \cdot f(n)$ time. If the algorithm found a solution in this bounded number of steps, we have the correct result. Otherwise we return 'error'. What is the probability of error for this approach? The famous Markov inequality can be applied to bound the error in this scenario.

●**Definition (Markov Inequality)** *For a non-negative random variable X it yields:*

$$P(X > t) \leq \frac{E(X)}{t}$$

The intuition behind this inequality is that the value of a random variable cannot deviate too much from the expected value with high probability, because then the expected value would be higher in the first place.

Exercise. Prove the Markov inequality.

We can reformulate the Markov inequality to make it more suitable for our purpose by stating that

$$P(X > t \cdot E(X)) \leq \frac{1}{t}.$$

Now if X is the runtime of our Las Vegas algorithm and $E(X)$ accordingly the expected runtime $f(n)$, we get

$$P(X > \alpha \cdot f(n)) \leq \frac{1}{\alpha}.$$

So the probability that the Las Vegas algorithm does not find a solution in the first $\alpha \cdot f(n)$ steps is less than $1/\alpha$, which automatically constitutes the error probability for the respective Monte Carlo algorithm.

Now lets consider the other way around: Given a Monte Carlo algorithm, can we construct a corresponding Las Vegas algorithm? A Las Vegas algorithm stops at the moment a correct result is found. So it is crucial to have a predicate telling the algorithm that the actual candidate is indeed a correct result. Remembering our initial example with the drugs in the lockers, the janitor knows for sure that it was a correct locker if he finds drugs in it. But in the example of finding an element equal to or larger than the median, it is not so easy to make sure that a suggested element is indeed correct. Here a scan over the array would be required that makes sure at least $n/2 - 1$ smaller elements exist. So the time to check would be already as large as the time to solve the problem deterministically. More generally, lets say we are given a Monte Carlo algorithm with runtime $f(n)$ and success probability $p(n)$. If we have a solution verifier that runs in $g(n)$, then there exists a Las Vegas algorithm with expected running time

$$E(X) \leq \frac{f(n) + g(n)}{p(n)}.$$

This runtime holds, because according to Lemma 1.1 we need expectedly $1/p(n)$ rounds for the first success and every round requires to run the Monte Carlo algorithm and check the respective result for correctness.

1.4 How to Analyze Randomized Algorithms

When analyzing randomized algorithms, we are of course interested in the same aspects as for deterministic algorithms, like space consumption and runtime (best case, average case, worst case). Only now these values can turn into expected values for Las Vegas algorithms. For Monte Carlo algorithms we furthermore have to carefully analyze the success probability. We give some details for analysis of the two algorithm classes in the remainder of the section.

1.4.1 (Very) High Probability and Concentration Bounds

To analyze the behaviour of a random variable X, we already identified the expected value as a useful tool. Nevertheless $E(X)$ cannot always give enough information to analyze certain randomized algorithms and randomized structures sufficiently. As $E(X)$ is some averaged value, it tells nothing about the distribution of the probabilities for the possible events. But often we are interested in how concentrated the values of X are around the expected value. Remember our dice example. A dice with possible scores 1,2,3,4,5,6 and a dice with 1,1,1,6,6,6 have the same expected value of 3.5. Nevertheless with the second dice we will not score anything close to the expected value. Considering e.g. the run time of a Las Vegas algorithm as a random experiment, we would like to make sure that the runtime of the algorithm is almost always small. To formalize the 'almost always' bit, there exist fixed terms in probability theory.

●**Definition (With High Probability)** *An event x occurs* with high probability (whp) *if with growing value of n the probability of x converges to 1 or alternatively,*

$$P(X = x) \geq 1 - n^{-c}$$

for some constant c. The event occurs with very high probability (wvhp) *if*

$$P(X = x) \geq 1 - 2^{-cn}.$$

Remember our example where we aimed for finding a number equal to or larger than the median. For the Monte Carlo algorithm we computed a failure probability of $\leq n^{-c}$, that means $P(success) \geq 1 - n^{-c}$. So we would say our Monte Carlo algorithm succeeds with high probability.

We already got in touch with one way to get more information about how concentrated the events are around the mean, namely applying the Markov inequality. But with this inequality we can only upper bound the probability that the value of X is much higher than the expected value. To illustrate the limitations of the Markov inequality better, we consider the following example.

● **Example (Points in Boxes)** *Let n points be u.a.r. distributed over the unit square ($[0,1]^2$). Divide the square in $n/\log^2 n$ square boxes, each with side length $\log n/\sqrt{n}$. For given $\epsilon \in (0,1)$, a box is called ϵ-nice if it contains at least $(1-\epsilon)\log^2 n$ points and at most $(1+\epsilon)\log^2 n$ points at the same time. We would like to prove that a fixed box is ϵ-nice with high probability.*

Let now X be a random variable counting the number of points in a fixed box. As we have $n/\log^2 n$ many boxes and uniformly distributed points, the expected number of points in a fixed box is $E(X) = \log^2 n$. According to the Markov inequality, we get:

$$P(X > (1+\epsilon)E(X)) \leq \frac{1}{1+\epsilon}$$

This bound is independent of n and we cannot even conclude that a single fixed box contains less than $(1+\epsilon)\log^2 n$ with high probability. But another classical concentration bound comes to help here.

● **Definition (Chebyshev Inequality)** *Let X be a random variable, $E(X)$ the expected value and σ the standard deviation. Then for all $t > 0$,*

$$P(|X - E(X)| > t\sigma) \leq \frac{1}{t^2}.$$

The standard deviation is defined as the square root of the variance $Var(X)$ with $Var(X) = E(X^2) - E(X)^2$. The intuition behind the Chebyshev inequality is, that the probability for a random variable to fall within t times the standard deviation around the expected value is large, namely $1 - {}^{1}/_{t^2}$. Plugging in e.g. $t = 2$, we can conclude that 75% of the times a random value falls in the interval $[E(X) - 2\sigma, E(X) + 2\sigma)]$.

For our points in the box example, the variance can be expressed as $\log^2 n(1 - log^2 n/n)$. With that information the Chebyshev inequality tells us,

$$P(|X - E(X)| > \epsilon E(X)) \leq \frac{Var(X)}{\epsilon^2 E(X)^2} = \frac{n - \log^2 n}{\epsilon^2 n \log^2 n}.$$

This term obviously tends to 0 with $n \to \infty$. Hence the probability of the counterevent, namely the box being ϵ-nice, converges to 1 with growing n. So according to our definition, a box being ϵ-nice has a high probability.

1.4.2 Analyzing Las Vegas Algorithms

Considering a Las Vegas algorithm, a random variable X is required to express the run time of the algorithm. So the expected value of X is also called the expected runtime of the algorithm. We differentiate between certain types of Las Vegas algorithms.

●**Definition (Completeness)** *A Las Vegas algorithm is called* complete*, if it is guaranteed to solve each problem instance (for which a solution exists) in time t_{max}, with t_{max} being an instance-dependent constant. Further a Las Vegas algorithm is called* approximately complete *if the probability to solve the problem converges to* 1 *with the run time approaching* ∞*. Las Vegas algorithms being neither complete nor approximatively complete are called* essentially incomplete.

Remember our example of finding the repeated element in an array by random sampling pairs of elements. This algorithm might run forever (never picking two elements which are equal). Nevertheless the probability to solve the problem can be expressed in dependency of the number of rounds k as $P(success) = 1 - (3/4)^k$, hence with growing k the probability converges to 1, classifying the algorithm as *approximately complete*. In our drug detection example, we randomly permuted the lockers and opened them in that order until we find some drugs. So after at most $n/2 + 1$ rounds, the algorithm will terminate, providing us with a $t_{max} \in \mathcal{O}(n)$ upper bound on the run time. Hence this algorithm is *complete*.

1.4.3 Analyzing Monte Carlo Algorithms

In Monte Carlo algorithms, we try to analyze the probability of success or error respectively. Thereby, if the problem to solve is a decision problem, we distinguish between *one-sided* and *two-sided* error. Remember the drug detection example again. Say now, we do not know for sure if there a pupils taking drugs and we apply the above described Monte Carlo algorithm (opening k random lockers) to decide if they do. If we find no drugs in k rounds, the answer returned will be 'no'. But of course, this might be false, as we might just have missed the lockers containing drugs. On the other hand, if we detect drugs, the answer is 'yes' and if 'yes' is returned as an answer, we know that this is true. In this case the error is only *one-sided*, as a false negative answer is possible, but a false positive answer is not. This principle is also often used in testing. If one wants to decide whether complex objects, e.g matrices, are equal, one could (instead of checking all entries) sample k entries and check for their equality. If a mistake is found in that sample, the matrices are not the same for sure. If all k entries are equivalent, the result is inconclusive and the answer 'matrices are equal' might be wrong. For an example of a possible *two-sided* error, consider the question whether in a delivery of $10,000$ light bulbs more than 80% are intact. The corresponding Monte Carlo algorithm is to select 10 light bulbs and test them. If 8 or more light bulbs work, we return 'true', otherwise 'false'. Obviously, both answers might be wrong. Even if all 10 light bulbs work, the remaining $9,990$ might be broken and even if zero light bulbs work in the test, the others might. So we have to be careful when stating the quality of a Monte Carlo algorithm, in particular we have to make sure that every possible kind of error was regarded.

Having determined a success probability for a Monte Carlo algorithm, a common approach to improve the chance of success is to repeat the algorithm. If the underlying problem is a decision problem, this increases the chance of getting a conclusive result. If the problem is a search or optimization problem, we return the best result of all runs as the final answer. Often, we aim for getting a (very) high probability of success. The introduced concentration bounds (Markov and Chebyshev inequality) will be useful in determining how often to run a Monte Carlo algorithm to achieve those high success probabilities.

2 Randomized Algorithms and Data Structures

In this chapter, we have a look on some very basic applications of randomization in the area of optimization, sorting, searching and testing. For further reading, have a look at `http://www.aw-bc.com/info/kleinberg/assets/downloads/ch13.pdf`. For SkipLists more details can be found here: `http://www.cs.umd.edu/~meesh/420/Notes/MountNotes/lecture11-skiplist.pdf`, for sorting here: `http://www.cs.cmu.edu/~avrim/451f11/lectures/lect0913.pdf`, for hashing here:

https://www.cs.princeton.edu/courses/archive/fall08/cos521/hash.pdf. A nice presentation on VC-dimension and ϵ-Nets can be found here https://www.cs.ucsb.edu/~suri/cs235/VcDimension.pdf.

2.1 The Maximum 3-Satisfiability Problem

We start with a well-known NP-complete problem, for which we will devise a very simple and surprisingly good randomized algorithm.

●**Definition (Max 3-SAT)** *Given a set of clauses $C_1, \cdots, C_k$ each of length 3 over a set of boolean variables $X = \{x_1, \cdots, x_n\}$ (for example $(x_1 \vee \neg x_2 \vee x_3) \wedge (\neg x_1 \vee x_4 \vee \neg x_5)$ with $k = 2, n = 5$). The problem of determining an assignment of the variables such that the maximum number of clauses is satisfied is called the Maximum 3-Satisfiability Problem or Max 3-SAT for short.*

The simplest randomized approach that comes to mind is to assign true or false to the variables u.a.r, i.e. each with a probability of $1/2$. How good is this approach expectedly?

Lemma 2.1. *The expected number of satisfied clauses by a random assignment is within an approximation factor $7/8$ of the optimum.*

Proof. Let Y denote the random variable that counts the number of satisfied clauses. As $E(Y)$ is rather hard to analyze, we use the linearity of the expected value here. So observe that $Y = Y_1 + \cdots Y_k$ with Y_i being 1 if clause C_i is satisfied (and 0 otherwise). So $E(Y_i) = P(C_i\ satisfied)$. As in each C_i the variables are concatenated via *or*, C_i is not satisfied only if *all* variables in the clause got the wrong value assigned. The probability for this to happen is $1/2^3 = 1/8$. Thus $P(C_i\ satisfied) = 7/8 = E(Y_i)$. For the desired expected value of Y we get $E(Y) = \sum_{i=1}^{k} E(Y_i) = \sum_{i=1}^{k} 7/8 = 7/8k$. Since no assignment can satisfy more than k clauses, our naive randomized algorithm provides expectedly a $7/8$-approximation. □

So the expected numbers of satisfied clauses is $7/8k$, completely independent of the inner structure of the clauses. What does it mean for the Max 3-SAT problem, that we can derive such an expected value? Well, for any random variable it yields, that there must be some point at which it assumes a value at least as large as its expected value. So if we expect $7/8$ of the clauses to be satisfied, it automatically means that we can assign to Y a value at least as high as $7/8k$. This immediately implies the next lemma.

Lemma 2.2. *For every instance of Max 3-SAT there exists an assignment that satisfies at least a $7/8$ fraction of the clauses.*

Of course, we have no clue how this assignment has to look like. Nevertheless we could prove its existence via a randomized algorithm. This principle of proving existence of a structure by showing that a random construction produces it with positive probability is an application of the so called *probabilistic method*, which we will investigate further in the final chapter. Still, it is quite surprising that we can come to such a general statement about Max 3-SAT (which has as such nothing to do with randomization) on the basis of analyzing a random assignment.

?**Exercise.** Prove that for every instance of Max 3-SAT with $k \leq 7$ clauses all clauses can be satisfied.

The described randomized algorithm is obviously a Monte Carlo algorithm. The runtime is fixed (indeed it is linear in n) and the result might be a Max 3-SAT solution, but it does not has to be. Based on the previous observation that always $7/8k$ clauses can be satisfied, a natural goal is to have an algorithm which always produces an assignment that realizes at least this number of satisfied clauses. So we turn the Monte Carlo algorithm into a Las Vegas algorithm by running it again and again until enough clauses are satisfied. The time to check the result of the Monte

Carlo algorithm is obviously linear in k. So the total runtime crucially depends on the number of rounds that the Las Vegas algorithm will take expectedly to return the desired assignment. Hence we need to calculate the success probability p. For that purpose, we first define with $p_j = P(Y = j)$, $j = 0, \cdots, k$ the probabilities to satisfy exactly j clauses. Then we can express the success probability as $p = \sum_{j \geq 7/8k} p_j$. We know that the expected value of Y is $7/8k$. Plugging in the definition of the expected value, we get.

$$7/8k = E(Y) = \sum_{j=0}^{k} j \cdot p_j$$

This can be reformulated as follows.

$$7/8k = \sum_{j<7/8k} j \cdot p_j + \sum_{j \geq 7/8k} j \cdot p_j$$

We now define k' to be the largest natural number strictly smaller than $7/8k$ (so in case $7/8k \in \mathbb{N}$ we have $k' = 7/8k - 1$, otherwise we have $k' = \lfloor 7/8k \rfloor$). On that basis, the following inequality must hold.

$$7/8k \leq \sum_{j<7/8k} k' \cdot p_j + \sum_{j \geq 7/8k} k \cdot p_j = k' \sum_{j<7/8k} p_j + k \sum_{j \geq 7/8k} p_j$$

We further observe that $\sum_{j<7/8k} p_j = 1 - p$ and therefore we get:

$$7/8k \leq k'(1-p) + kp \leq k' + kp$$

In terms of p this equals:

$$p \geq \frac{7/8k - k'}{k}$$

According to the definition of k' we know that $7/8k - k' \geq 1/8$. Hence we have a valid lower bound for our success probability p, namely $1/8k$. Combining this value with our Lemma 1.1, we have proven the following lemma.

Lemma 2.3. *There exists a Las Vegas algorithm for Max 3-SAT which satisfies at least $7/8$ of the k clauses in an expected number of $8k$ rounds, with each round requiring $\mathcal{O}(n + k)$ time.*

So the total expected runtime is $\mathcal{O}(nk + k^2)$, which is polynomial in the input.

2.2 k-Select, Approximate Median and QuickSort

We already came across the problem to compute the median of n numbers, when we considered the repeated element problem and the task of finding a large number in an array. A more generalized version is to ask for the k^{th} largest element in an array A of n elements. This problem will be called *k-Select* in the following.

2.2.1 Selection

There exists a deterministic algorithm, which solves k-Select in $\mathcal{O}(n)$. This algorithm *DetSelect*$(A[n], k)$ is sketched in the following.

1. group the elements in A into $n/5$ groups of size 5 each
2. compute the median of each group naively (e.g. by sorting)
3. recursively apply steps 1. and 2. until the median of those medians is known, call this p
4. use p as a pivot to split A in $A_1 := \{a \in A | a < p\}$ and $A_2 := \{a \in A | a > p\}$
5. if $|A_1| = k - 1$ return p as the result
6. if $|A_1| \geq k$ recursively call *DetSelect*(A_1, k), otherwise call *DetSelect*$(A_2, k - |A_1| - 1)$

Lemma 2.4. *Deterministic k-Select runs in $\mathcal{O}(n)$.*

Proof. Let $T(n, k)$ denote the runtime of the algorithm for a call with an array of size n and parameter k. With $T(n)$ we refer to the worst-case runtime over all possible values of k, so $T(n) = \max_{k=1,\cdots,n} T(n, k)$. Obviously if we can show that $T(n) \in \mathcal{O}(n)$, we are done. Now lets have a look at the steps performed in the algorithm. The first step obviously is linear in n and creates only linear many groups each of constant size. So step 2 also runs in linear time, because we only spend constant time on each of the groups. The recursive call in step 3 takes time at most $T(n/5)$ as the search space is reduces accordingly. Splitting the array in step 4 is possible in linear time again. So the crucial question for the runtime is how large the subarray will be on which we recurse in step 6. What is the worst case here? Well, remember that p is the median of medians of the $m = {}^n\!/\!_5$ groups. So in at least $\lceil {}^m\!/\!_2 \rceil$ of the groups at least three of the five elements have to be $\leq p$. Hence the total number of elements $\leq p$ is lower bounded by $3\lceil {}^m\!/\!_2 \rceil \geq {}^{3n}\!/\!_{10}$. The same bound can be proven for elements $\geq p$. So the maximum number of elements in A_1 or A_2 respectively is bounded by ${}^{7n}\!/\!_{10}$. Therefore the recurrence we get is:

$$T(n) = cn + T({}^n\!/\!_5) + T({}^{7n}\!/\!_{10})$$

We claim that this yields $T(n) \leq 10cn$ and prove it inductively. Obviously this is true for small constant n, e.g. $n = 5$. For the induction step, we consider $T(n + 1) = c(n + 1) + T({}^{(n+1)}\!/\!_5) + T({}^{7(n+1)}\!/\!_{10})$. As ${}^{(n+1)}\!/\!_5$ and ${}^{7(n+1)}\!/\!_{10}$ are both smaller than n, the induction hypothesis can be applied and we get

$$T(n+1) \leq c(n+1)+10c\cdot{}^1\!/\!_5(n+1)+10c\cdot{}^7\!/\!_{10}(n+1) = c(n+1)+2c(n+1)+7c(n+1) = 10c(n+1).$$

Accordingly $T(n) \in \mathcal{O}(n)$, proving the lemma. □

So the sticking point of the proof was the pivot element p splitting the array in two almost balanced subarrays. Indeed p was constructed in a way that ensured it to be not too far away from the median. How about finding an element close to the median based on randomization? Lets say we want to find an element, which is at most δ positions away from the median (considering the elements in order). Lets construct a very simple Monte Carlo algorithm for that.

Algorithm 3: Monte Carlo Approximate Median

```
begin
    pick $r$ u.a.r. in $\{1, \cdots, n\}$;
    $a \leftarrow A[r]$;
    $rank \leftarrow 1$;
    for $i$=1 to $n$ do
        if $A[i] < a$ then
            $rank \leftarrow rank + 1$;
    if $rank \in [{}^n\!/\!_2 - \delta, {}^n\!/\!_2 + \delta]$ then
        return a;
    else
        return error;
end
```

The runtime is clearly linear, but the output obviously must not be a δ-approximation of the median. What is the probability of success? There are 2δ many elements for which the algorithm would not report 'error'; hence $P(success) = {}^{2\delta}\!/\!_n$. This reflects the intuition that the closer we like to get to the median the smaller the probability for success. Now lets say, we do not run the Monte Carlo algorithm once, but c times. The runtime then increases to $\mathcal{O}(cn)$. The chance that all c trials fail can be described as $(1 - {}^{2\delta}\!/\!_n)^c$. If we want to guarantee success, we turn the algorithm into a Las Vegas algorithm. Based on the previously computed success probability this algorithm needs expectedly $\mathcal{O}(n/\delta)$ rounds to come up with an approximate median.

So theoretically we could plug in the Monte Carlo or Las Vegas median finder in the deterministic algorithm for simplification. But there is an even better way to integrate randomization in k-Select. If choosing the pivot element p to split A in A_1 and A_2 u.a.r. we can prove an expected runtime of $\mathcal{O}(n)$.

Lemma 2.5. *Randomized k-Select runs expectedly in $\mathcal{O}(n)$.*

Proof. For analysis we assume that the element we want to identify always ends up in the larger of the two A_1 and A_2. So there are two possible pivots (the smallest and the largest element) to end up with $\max(|A_1|, |A_2|) = n - 1$ elements to consider in the next round, accordingly two pivots for $\max(|A_1|, |A_2|) = n - 2$ and so on. Obviously, $\max(|A_1|, |A_2|)$ can never be lower than $n/2$. As p is chosen u.a.r. we can determine our expected runtime as:

$$T(n) = \frac{2}{n} \sum_{i=n/2}^{n-1} T(i) + cn$$

We prove by induction that $T(n) \in \mathcal{O}(n)$. For $n = 1$ correctness is obvious. Now consider

$$T(n+1) = \frac{2}{n+1} \sum_{i=(n+1)/2}^{n} T(i) + c(n+1).$$

Expanding the last term of the sum we can reformulate this as

$$T(n+1) = \frac{2}{n+1} \sum_{i=(n+1)/2}^{n-1} T(i) + cn + \frac{2}{n+1} T(n) + c$$

The first part is obviously $< T(n)$, so we get

$$T(n+1) < T(n) + \frac{2}{n+1} T(n) + c$$

Together with the induction hypothesis we end up with $T(n+1) \in \mathcal{O}(n)$. □

This is a nice example of a randomized algorithm which is easier to implement and easier to analyse than its deterministic counterpart, but performs expectedly equally well.

2.2.2 Randomized Sorting

A very similar principle can be used when the goal is to sort an array. There is a variety of deterministic algorithms for sorting which all exhibit a runtime that matches the sorting lower bound of $\mathcal{O}(n \log n)$ for general inputs, e.g. insertion sort, heap sort, and merge sort. The algorithm we will now focus on is QuickSort, a randomized sorting algorithm which is often implemented for real world use. In a deterministic version we could choose the pivot always as the median (in linear time), with the method described above. This would split the set of elements to recur on exactly in the middle. So the runtime would be $T(n) = T(n/2) + T(n/2) + \mathcal{O}(n)$, with the median selection and the creation of A_1 and A_2 being responsible for the last term. So the deterministic runtime is $T(n) \in \mathcal{O}(n \log n)$. One could think now, that the analysis of the randomized version can be conducted just like that. If we choose the pivot u.a.r. among all elements in A, we choose expectedly an element with the rank of the median (just like for a dice with scores $1, 2, \cdots, n$). But remember, that the expected value does not tell us, if the random variable X can get even close to $E(X)$. So if we would choose p as the minimum or the maximum of the elements in A (each with probability $1/2$), the expected value would just be the same. The runtime for that approach would be $\mathcal{O}(n^2)$, though. So we require a more careful analysis for proving the expectedly good runtime of QuickSort.

Algorithm 4: QuickSort(A)

```
1 begin
2     if $|A| \leq 1$ then
3         return $A$;
4     select $p$ in $A$ u.a.r.;
5     $A_1 \leftarrow \{a \in A | a < p\}$;
6     $A_2 \leftarrow \{a \in A | a > p\}$;
7     return QuickSort($A_1$)+$p$+QuickSort($A_2$);
8 end
```

Lemma 2.6. *QuickSort runs expectedly in $\mathcal{O}(n \log n)$.*

Proof. Choosing the pivot p u.a.r means that every split is equally likely: if p has rank 0, we get a partitioning in $|A_1| = 0/|A_2| = n-1$, if it has rank 1 we get $1/n-2$ and so on. In general for the rank being i we get $i/n-i-1$ and the chance to get this split is $1/n$. This leads us to the following formula for the expected runtime:

$$T(n) = \frac{1}{n}\sum_{i=0}^{n-1}(T(i) + T(n-i-1)) + cn$$

Here, the last term is still required to describe the comparisons of all elements to the pivot to get A_1, A_2. The pivot selection itself is now a constant time operation in contrast to the deterministic variant. Obviously we can reformulate our expected runtime as:

$$T(n) = \frac{2}{n}\sum_{i=1}^{n-1} T(i) + cn$$

We will prove that $T(n) \leq 2n\log n + \mathcal{O}(n)$, again by induction. The base case for $n = 1$ is clear. Now we replace all $T(i)$ terms with the induction hypothesis:

$$\begin{aligned}
T(n) &\leq \frac{2}{n}\sum_{i=1}^{n-1}(2i\log i + \mathcal{O}(i)) + cn \\
&\leq \frac{2}{n}\sum_{i=1}^{n-1} 2i\log n + \frac{2}{n}\sum_{i=1}^{n-1}\mathcal{O}(i)) + cn \\
&= \frac{4\log n}{n}\sum_{i=1}^{n-1} i + \frac{2}{n}\sum_{i=1}^{n-1}\mathcal{O}(i)) + cn \\
&\leq \frac{4\log n}{n}\frac{(n-1)(n-2)}{2} + \frac{2}{n}\frac{\mathcal{O}(n^2)}{2} + cn \\
&\leq 2n\log n + \mathcal{O}(n)
\end{aligned}$$

Accordingly we get $T(n) \in \mathcal{O}(n \log n)$. □

This proof is rather technical and does not give much insight there the log-term actually stems from. Therefore we also consider an alternative proof, which benefits from the linearity of the expected value.

Proof. Let $a_1, a_2, \cdots, a_n$ be the sorted sequence of the elements in A. We would like to count all comparisons QuickSort makes to determine the total runtime. For that purpose let X_{ij} denote the random variable that is 1 if a_i and a_j got compared during the course of the algorithm and 0 otherwise. Obviously $T(n) = \sum_{j=1}^{n}\sum_{i<j} E(X_{ij})$. To get the expected value of X_{ij}, we need to

calculate the probability for a comparison of a_i and a_j. If those two got compared, it means that at the point of comparison either a_i or a_j was selected as pivot element, and before that none of the elements between a_i and a_j in the sorted sequence became pivot (otherwise a_i and a_j would have been split up). As the pivot is chosen u.a.r., the chance for X_{ij} being 1 can be expressed as $2/(j-i+1)$ because there are $j-i+1$ elements in the sequence from a_i to a_j and only two of them enforce the comparison. So we get:

$$T(n) = \sum_{j=1}^{n}\sum_{i<j} E(X_{ij}) = \sum_{j=1}^{n}\sum_{i<j} \frac{2}{j-i+1} = \sum_{j=1}^{n}\sum_{i<j}\frac{2}{i} = \sum_{j=1}^{n} 2H(j) \leq n \cdot 2H(n) \approx 2n\log n + 2n$$

Hence also this analysis leads us to $T(n) \in \mathcal{O}(n \log n)$. □

❓**Exercise.** Implement a deterministic sorting algorithm which runs in $\mathcal{O}(n \log n)$ and QuickSort. Let both algorithms run a set of 10,000,000 numbers, measure the runtime and count the comparisons. Run QuickSort on the same input 100 times and have a look at the distribution of runtime and number of comparisons.

2.2.3 Sorting Lower Bounds

We consider now the class of comparison-based sorting algorithms, i.e. the only operations allowed are pairwise comparisons ("is $a_i < a_j$?") and rearrangement of the elements based on the outcome of the comparison. (E.g. RadixSort is not in this class as it uses a form of hashing to compute the result.) For deterministic comparison-based algorithms (incarnations thereof are e.g. MergeSet and InsertionSort), we will show a worst-case lower bound of $\Omega(n \log n)$ comparisons to be performed to get the correct output permutation.

Lemma 2.7. *Any deterministic comparison-based algorithm must perform $\Omega(n \log n)$ comparisons to sort n elements in the worst case. So for any deterministic comparison-based sorting algorithm $\mathcal{A}$ and for all $n \geq 2$ there exists an input I of size n such that $\mathcal{A}$ makes at least $\log_2(n!) = \Omega(n \log n)$ comparisons to sort I.*

Proof. W.l.o.g. we assume the input is $\{1, \cdots, n\}$ with the elements being in some unknown order. The basic idea is that two different input orders cannot both be correctly sorted by applying the exact same permutation. So if two inputs I_1, I_2 are consistent with all the comparisons the algorithm made so far, then the algorithm cannot be done yet. So implicitly, the sorting algorithm has to pin down which input order was given. Now let S be the set of input orderings consistent with all answers to comparisons made so far. Initially S is the set of all possible permutations and therefore has a size of $n!$. At the end of the algorithm the set size must be reduced to 1. We can think of the algorithm as a way to split s always in two groups: the one that contains all inputs for which the answer to the actual question is *yes*, and the group there the answer is *no*. Being pessimistic, we always assume our input to be in the larger of those two groups. Accordingly the algorithm needs to make at least $\log_2(n!)$ many comparisons before it can halt. Hence we get:

$$\log_2(n!) = \sum_{i=2}^{n} \log_2(i) = \Omega(n \log n)$$

□

Lets consider a small illustrating example.

Example ($n = 3$) *Initially S contains the following permutations: $\{123\}, \{132\}, \{213\}, \{231\}, \{312\}$ and $\{321\}$. Assume the first question is "$A[0] < A[1]$?". If we only consider the inputs with the* *answer being* yes*, we have reduced S to $\{123\}, \{132\}, \{231\}$. If we now ask for "$A[1] > A[2]$?", we just remove one ordering and end up with S being $\{132\}, \{231\}$. So we need a third and final question to decide for the correct input, we choose "$A[0] > A[2]$?" and get only for $\{231\}$ that the answer is* yes*, hence the input is isolated now.*

So somebody who knows how your deterministic algorithm makes its decisions can construct an example instance to force the algorithm to perform $\Omega(n \log n)$ comparisons. But maybe on average the algorithm performs much better? Here with on average we mean that the input is chosen randomly among all $n!$ possible input permutations. The next lemma will destroy that hope.

Lemma 2.8. *For any deterministic comparison-based algorithm, the average-case number of comparisons is at least $\lfloor \log_2(n!) \rfloor$.*

Proof. We consider the decision tree that results from isolating all the inputs by answering sequences of questions. The branches in the tree indicate then sequences of *yes/no* answers, the leaves correspond to input permutations (so we have $n!$ many leaves). The depth of a leaf then corresponds to the number of comparisons necessary to determine the input permutation. If this binary tree would be completely balanced (i.e. maximum gap between the depth of two leaves is 1), all leaves would have a depth of $\lfloor \log(n!) \rfloor$ or $\lceil \log(n!) \rceil$ respectively. Hence in that case the lemma is proven. We now show that the tree that minimizes the average depth indeed is the completely balanced one. So consider now an unbalanced tree. Let the maximum depth in this tree be D and the minimum depth of a leaf be d and the average depth be avg. The operation we perform now is to choose a parent of two siblings at maximum possible depth $(D-1)$, and reassign its children to a leaf at depth d. As with this operation we remove two leaves at depth D and one at depth d while creating one new at depth $D-1$ (the former parent) and two new at depth $d+1$, we get overall:

$$avg' = \frac{avg \cdot n! - 2D - d + (D-1) + 2(d+1)}{n!} = \frac{avg \cdot n! - D + d + 1}{n!}$$

And because $D - d \geq 2$ by definition of an unbalanced tree, it yields:

$$avg' \leq \frac{avg \cdot n! - 1}{n!} < avg$$

Hence any unbalanced tree can be modified such that the average depth shrinks. Therefore such a tree cannot be one that minimizes the average depth, so the best possible average we can achieve is indeed be realized by the balanced tree. □

But what about using a randomized algorithms like QuickSort? The bounds do not automatically apply here! But we will show in the following that the sorting lower bound for randomized comparison-based algorithm matches the deterministic lower bound.

Lemma 2.9. *The expected number of comparisons a randomized comparison-based sorting algorithm has to perform is $\Omega(n \log n)$.*

Proof. The crucial idea to prove this lemma is to interpret a randomized algorithm A as a probability distribution over all deterministic algorithms. Every sequence of questions that leads to a conclusive result can be seen as one deterministic algorithm A_s. The randomized algorithm chooses one of those algorithms u.a.r. and hence we can determine the average runtime of the randomized algorithm as $T(n) = {}^1/_{n!} \sum_I \sum_{s \in S} P(s)(\text{runtime of } A_s \text{ on } I)$. This can be reformulated as $T(n) = \sum_{s \in S} P(s) {}^1/_{n!} \sum_I (\text{runtime of } A_s \text{ on } I)$. We showed that the average deterministic runtime is $\geq \lfloor \log_2(n!) \rfloor$. Plugging this in, we get $T(n) \geq \sum_{s \in S} P(s) \lfloor \log_2(n!) \rfloor = \lfloor \log_2(n!) \rfloor$. □

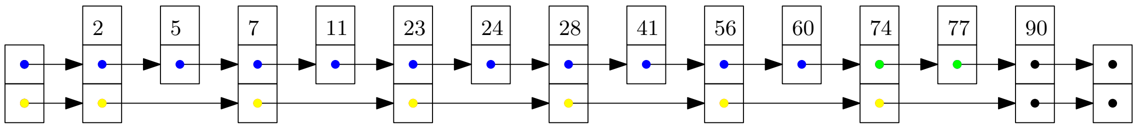
So staying inside the comparison-based model, the construction of a randomized sorting algorithm performing better than QuickSort in terms of O-Notation is not possible.

2.3 SkipLists

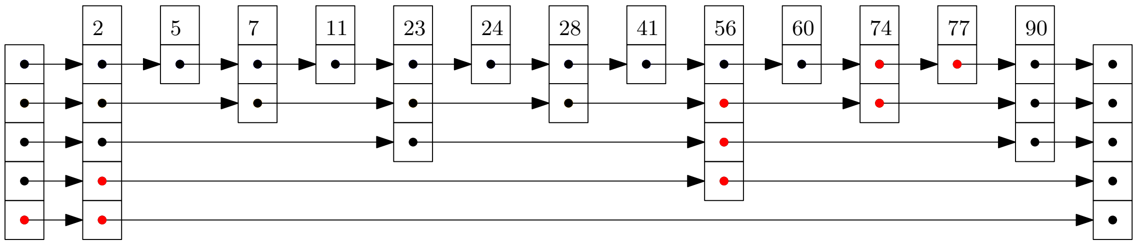
After considering optimization and selection/sorting, we now come to the application of search. So given a set of numbers, e.g. represented as array, we want to find the position at which a certain element a is located or the message that a is not contained in the set. Naively, we could just parse through the array and stop as soon as the element is detected and report the respective position. This costs us at most n (if the last element equals a or a is not contained). Even if the elements are sorted and stored in a linked list, the complexity stays the same for the search operation. But one can do much better. For example a search trees (like AVL trees, red-black trees) allow to identify an element in $\mathcal{O}(\log n)$ time. The same time holds for insertion and deletion of elements. A randomized data structure called SkipLists has the same asymptotic time bounds as many complicated search trees, but in practice SkipLists are often simpler, faster and require less memory. We will start by constructing a deterministic SkipList and subsequently have a look where randomization can improve the performance.

2.3.1 Deterministic SkipLists

We start with a simple linked list of the sorted elements. Now consider the linked list, where we leave out every second element in the sorted sequence and instead link 'over them' – that is the skipping part. What is the maximum search time if we have those two lists? Well, lets first consider the list which skipped every second element. We can just parse through it, until we either identified a or the next link points to an element which is larger than a. Now we switch to the list without skipped nodes, and consider the current element there. Either the next link in this list points to a or a is not contained in the set. For an illustration consider the following example:



In the image, the elements considered when searching for 77 only in the complete linked list are coloured blue, the ones for the combination of the complete list and the skip list in yellow; green elements are considered by both searches. Searching e.g. for 75 would result in the very same image. We observe, that the search time when including the skip list is almost halved. In fact, every search for n elements now requires at most ${}^{n}/_{2} + 1$ elements to be visited. But why stop at skipping every second element?



In the image above, we augmented the data structure with skip lists for every 4^{th}, every 8^{th} and every 16^{th} node. For the latter only a single element remained in the list. The search path for 77 is indicated by the red dots. We see, that using this data structure we only need to follow four links to get to the result (compared to 12 in the simple linked list). The algorithm behind it is simple: We start in the last list and follow the links until we found our searched element or the next element in the list is larger (or the end of the list). If we are not done yet, we switch to the list above (at that position) and proceed similar. What is the general search time now?

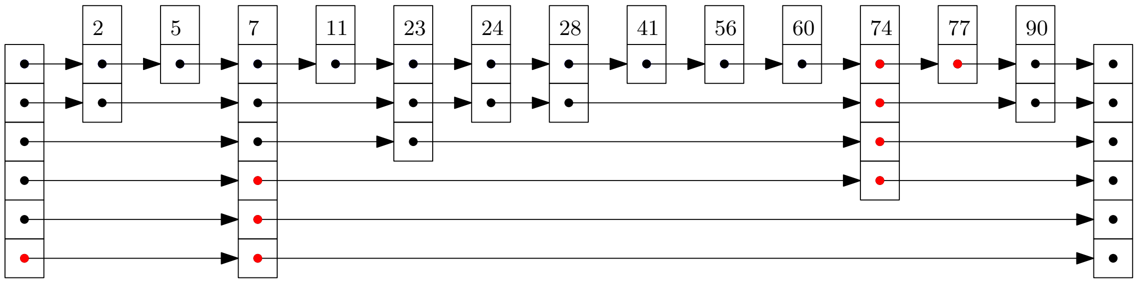
Lemma 2.10. *Deterministic SkipLists exhibit a search time of* $\mathcal{O}(\log n)$.

Proof. The data structure now consists of $\log n$ lists, with list i keeping only every 2^ith node and skipping the rest. If we switch from list i to $i-1$ the search space in list $i-1$ is always limited to the next element in this list. Because the element after that is also contained in list i, and the switch indicated that this element is larger than the element we search for, we never will follow the respective link in list $i-1$. So in each level only constant time is spend, leading to an overall search time of $\mathcal{O}(\log n)$. □

So the search time indeed is fine. But the problem comes with inserting new elements. Just like for a perfectly balanced search tree, a new element can enforce the complete reconstruction of the data structure requiring $\mathcal{O}(n)$ time. Therefore we will now introduce some slack, by constructing the SkipLists in a randomized manner.

2.3.2 Randomized SkipLists

The idea is to choose the number of lists in which an element will appear not deterministically based on its position in the sorted list, but use coin flips instead. So for every element we flip a fair coin until it shows *head*. We count the number of times it showed *tail* before that and let that determine in which lists the element appears. Reconsidering our previous example, we could e.g. get something like this:



The search algorithm stays exactly the same. What about the search time?

Lemma 2.11. *Randomized SkipLists exhibit an expected search time of* $\mathcal{O}(\log n)$.

Proof. First we observe that the expected number of levels is $\mathcal{O}(\log n)$. For level 0 we have n elements. Every of those survives with a probability of $1/2$ and therefore expectedly $n/2$ of the elements are contained in level 1. For level 2, the probability of survival is $1/4$ and the expected number of elements $n/4$ accordingly. For level i, we have a probability of $1/2^i$ and expectedly $n/2^i$ elements, so for $i > \log n$ we expect empty lists. (The same argumentation could be made by determining the expected max level of a single element.) So expectedly, the randomized SkipList structure has the same number of elements in each level as the deterministic variant. We now prove that indeed the search time will be the same expectedly. For that purpose, we will use a so called *backward analysis*. So instead of counting how many links we need to follow to get to the element we searched for when starting at the head of the list with the maximum level, we now start at the element in level 0 and construct the reverse search path until we meet the head of the maximum level list. Counting the number of links backwards obviously does not change the result. Observe, that in the reverse search path, we will always take a step up if we can (so switch to the next higher level). Otherwise we take a step to the left. What is the probability to make a step up? Well, it is $1/2$, because the chance for an element present in level i to be also present in level $i+1$ was determined by a fair coin flip. Hence the chance to go left is $1 - 1/2 = 1/2$ as well. So let X_i denote the number of steps we need to walk through i levels, we get:

$$E(X_i) = 1 + \frac{1}{2}E(X_{i-1}) + \frac{1}{2}E(X_i)$$

The 1 counts the current step, with probability $1/2$ we came from the level below (so invoking X_{i-1}) and with the same probability we came from the right. This can be rewritten as $E(X_i) =$

$2 + E(X_{i-1})$. If we expand this term, we observe that

$$E(X_i) = 2 + E(X_{i-1}) = 4 + E(X_{i-2}) = 6 + E(X_{i-3}) = \cdots = 2j + E(X_{i-j}) = \cdots = 2i.$$

As i corresponds to the level, we have an expected search time of at most $E(X_{\log n}) = 2\log n \in \mathcal{O}(\log n)$. $\square$

What about inserting a new element? First we need to search for this new element, which costs us $\mathcal{O}(\log n)$ expectedly. This will provide us with the position in the complete list, where the new element has to be inserted. Then we determine the number of levels it will appear in by tossing a fair coin and waiting for *head* just like before. As we expect the max level to be in $\mathcal{O}(\log n)$, this step will not dominate the runtime. Now we have to reset the links that are affected by the new created entries. How many links do we have to change expectedly? Well, in the final structure only one link per SkipList can point to the element and only one link can go outwards. Hence the number of affected links (the ones that spanned the new element before, but are now interrupted) corresponds to the number of levels the element appears in, and hence is also in $\mathcal{O}(\log n)$. Deleting an element works very similar, by redirecting links pointing to the element to the next element in the list which appears in the corresponding level. Therefore insertion and deletion are also $\mathcal{O}(\log n)$ operations in randomized SkipLists, making this data structure a good alternative for more complicated search structures and algorithms.

❓**Exercise.** Analyse the (expected) space consumption of deterministic and randomized SkipLists.

2.4 Randomized Hashing

We now consider the basic dictionary problem. Here we are given a subset S of n items from a large universe U and want to store them in order to perform look-ups as quickly as possible (e.g. S are the names of students in the course, U is the set of all names). This is not so different from the problem considered before, namely to search efficiently for a specific item in a set. But now we have a finite (although possible very large) universe of elements to deal with. Using hashing, this will enable even better access times to single elements than for the search structures considered before, but at the cost of a higher space consumption. Hashing is widely used in many areas, as e.g. cryptography and complexity theory. Also in AI-search programs hashing is used to remember which parts of the state-space already have been considered. After investigating how hashing works in detail and how randomization can be used to improve this data structure even further, we will subsequently see how hashing can also be used to decide if two complex structures are equal in a Monte Carlo like manner.

2.4.1 Universal Hashing Principle

The general procedure can be described quite simply. We have an array of size m to store our items, and a so called hash function $h : U \to \{0, \cdots, m-1\}$ which maps each element in the universe to a specific position in the array. At each position we have a linked list to store all elements with the same hash function value. Therefore the look-up time for an element can be described as the time needed to evaluate the hash function plus linear time in the length of the respective list (as we need to scan those elements for the one we search). Hence we would like to have a hash function that avoids collisions as far as possible, i.e. the number of elements with the same hash function value.

One way to spread elements nicely over the hash array would be select the position u.a.r., but then the ability to find a specific element again is destroyed. So the hash function itself needs to be deterministic. Well, then lets just fix some deterministic hash function.

Lemma 2.12. *For any hash function h, if $|U| \geq (n-1) \cdot m + 1$, there exists a set $S \subseteq U$ of n items that are all hashed to the exact same location.*

Proof. Consider the opposite for contradiction. So every location in the hash table contains at most $n-1$ elements from U. Then the total number of elements contained is $\leq (n-1)m < |U|$. □

Example (Bad Hashing) *Let U be the set of integers $\leq 2^m$ and $h(u) = u \mod m$ the used hash function. If we choose S as $s_1 = m+c, s_2 = 2m+c, \cdots, s_n = nm+c$ with $c < m$ all elements in S will be hashed to position c.*

So we are screwed. As soon as we fix a deterministic hash function, someone evil can construct an input that leads to a search time of $\mathcal{O}(n)$ for an element, which is far away from the quasi-constant time we would like to have. But while we cannot use randomization in the hash function itself, we can apply randomization in the hash function construction process. So instead of a single hash function h, we create a family of hash function H and pick $h \in H$ randomly. If implemented correctly, we will be able to show that for any kind of input S we will have expectedly a good performance. This principle is known as *universal hashing.*

●**Definition (universal)** *A family of hash functions H is called* universal, *if $\forall u, v \in U, u \neq v$: $P_{h\in H}(h(u) = h(v)) \leq {}^1\!/\!_m$ or in other words $|\{h \in H : h(u) = h(v)\}| \leq {}^{|H|}\!/\!_m$. Moreover we say H is* nearly universal *if $\forall u, v \in U, u \neq v : P_{h\in H}(h(u) = h(v)) \leq {}^2\!/\!_m$*

So two items in U create a collision with probability at most ${}^1\!/\!_m$, if h is uniformly drawn from a universal hash family H.

Lemma 2.13 (Expected Number of Collisions)**.** *If H is universal, then for any $S \subseteq U$ with $|S| = n$ and any $u \in U$ the expected number of collisions of u and elements in S is at most ${}^n\!/\!_m$.*

Proof. Let X denote the number of collisions with u and $E(X)$ the expected value. We can express X as $\sum_{i=1}^{n} X_i$ with $X_i = 1$ if u collides with element s_i in S and $X_i = 0$ otherwise. So $E(X_i)$ equals the probability of u and s_i colliding. As H is universal, we get $E(X_i) \leq {}^1\!/\!_m$. Using the linearity of the expected value we get $E(X) = \sum_{i=1}^{n} E(X_i) \leq n \cdot {}^1\!/\!_m = {}^n\!/\!_m$. □

According to this lemma, if $m \in \mathcal{O}(n)$, we expect a constant number of collisions for a single element and therefore constant query times. So this is exactly what we looked for. And now the dependency on the input 'behaving well' is no longer there. But of course, to use this properties in practice, we need to be able to construct such a hash family set H at first.

2.4.2 Designing a Universal Hash Family

We now consider the keys to hash as integers and w.l.o.g. to be the set $\{1, \cdots, |U|\}$ (which can easily be realized by enumerating the objects). The existence of a universal hash family is clear, as we simply could consider the set of all functions $f : U \to \{0, \cdots, m-1\}$.

Exercise. Prove that this hash family is universal.

But of course, managing the set of all functions is not a good idea for practical use. So we would like to choose h from a much smaller space, but with the same properties.

Lemma 2.14. *Let p be a prime $> |U|$. Let H be the hash family that contains*

$$h_a(u) = (au \mod p) \mod m$$

for each $a \in \{1, \cdots, p-1\}$. Then H is nearly universal.

Proof. We have to show that the collision probability is bounded by $^2/_m$. For that purpose, we first show that for two elements $u, v \in U, u \neq v$ it yields $(au \mod p) \neq (av \mod p)$ which means a collision cannot arise from the first part. Assume otherwise for contradiction. Then we have $a(u-v) = kp$ with $k \in \mathbb{N}$. As we know that $|u-v| \leq |U| < p$ (by choice of p) and $a < p$ as well, the product of a and $(u-v)$ cannot by divisible by p, which contradicts the equality above. So no collisions would occur with our hash function if the hash table size would be p. But as the hash table size is $m \ll p$, the second part of the hash function where we compute the result modulo m can indeed create collisions. With $(au \mod p) = q, (av \mod p) = q'$, for a collision to happen we require $q - q' = km$ with $k \in \mathbb{Z}$, or in other words $q' = q - km$. We know that $k \neq 0$, because otherwise already the first part would have enforced the collision, which we disproved. Moreover $|k|$ has to be upper bounded by $^{(p-1)}/_m$ as $q \leq p-1$ and $q' \geq 0$. So the probability of a collision is the probability of selecting the specific value for a that led to q, q' multiplied with $2^{(p-1)}/_m$ (as this denotes the number of possible values for k). As there are $p-1$ choices for a, the probability for a certain a is $^1/_{(p-1)}$ and so the total collision probability for the hash set family H is bounded by $2^{(p-1)}/_m \cdot {}^1/_{(p-1)} = {}^2/_m$ □

Why does this solve our problem? If $|U| \leq 2^w$ we can choose p between 2^w and 2^{w+1}. Such a prime always exist. Bertrand postulated that for $n > 1$ there exists always at least one prime p with $n < p < 2n$. A stronger version of this claim was proven by Chebyshev. So setting $n = 2^w$ we can be sure that we will detect a feasible prime number when scanning the next 2^w elements.

At the beginning of the computation we can select a random hash function from H and store all necessary information in constant space (namely only p and a). Moreover the hash function can be evaluated in constant time. Plugging in near universality in Lemma 2.13, we have an expected number of collisions of $^{2n}/_m$. Therefore for a hash table of size cn with c being a small constant, we only expect $^2/_c$ collisions per entry. Hence the total space consumption is linear in the set size S and a look-up is expectedly a constant time operation. Note, that there are very similar construction schemes for real universal functions (using an additional parameter).

2.4.3 Fingerprinting

One of the important applications of hashing is comparing two objects in an efficient manner. Consider the following example:

Example (Example) *Alice and Bob both have a lecture script and want to find out if their versions are exactly the same. For that purpose Alice could send Bob her script, Bob could check for differences and then report back to Alice.*

The obvious disadvantages of this procedure are the large amount of communication necessary (transmitting the whole script) and the long evaluation time for Bob. On the positive side, the answer will always be correct (if Bob can be trusted). But what if we do not have the bandwidth or the budget for this amount of communication or not the time to compare both scripts letter for letter? One simple idea is to hash both scripts using a suitable function with a fixed output size. Then Alice could just transmit her hash function and the hash function value for her script. Bob could use the same hash function on his script and then just compare the hash values. If they are not equal, he knows the scripts are different. But of course, if the hash values are the same, the scripts might still be different. In fact, if someone knows about Alice hash function, a script could be constructed that leads to a false positive equality report for sure. So just like before, fixing a deterministic hash function a priori is not a good idea for Alice.

Exercise. Assume Alice constructs a universal hash family H and chooses the specific hash function h on demand. Let for all $h \in H$ the hash value be k bits long. What is the probability of two different scripts to be mapped to the same value? What about repeating the process c times?

Now let Alice and Bob have a very sensitive message and not something boring like a script.

The message contains 1000 bits. Alice is aware that sending the message over the internet invites people to eavesdrop. Assuming that the evil internet spies have unlimited computational resources, schemes like RSA are not safe enough. Instead, when Alice and Bob met in person last time, they flipped a fair coin 2000 times and both wrote down the resulting bit sequence. The plan now is as follows: Alice encrypts her message by using bitwise XOR of her message and the first 1000 coin flip bits and sends this information to Bob, who decrypts it similarly. But somehow somebody evil with great interest in the message heard of their plan. So he send a spy to Bob to copy the coin flip bits. Bob, waking up in the night due to some noise in his office, interrupts the spy while he transmits the secret sequence. The spy gets away, but Bob is able to keep the device the spy used for transmission. Bob sees, that 200 bits were sent.

❓**Exercise.** If the 200 bits are consecutive bits of the secret sequence, how should Alice and Bob proceed?

But surprisingly the 200 bits do not look familiar to Bob at all. So the 200 bits the spy sent contain information about the secret sequence also in an encrypted way.

❓**Exercise.** What should Alice and Bob do now?

Alice and Bob talk over the internet (where the evil people will listen) about the problem. They then agree to use hashing. In fact Alice constructs a hash function h that maps $\{0,1\}^{2000}$ to $\{0,1\}^{1000}$. Then Alice sends the description of h to Bob (via internet, so the spies know about). Then they both hash their secret sequence. Alice then uses the resulting 1000 bits to encrypt her message as originally planned and sends the result to Bob.

❓**Exercise.** Why does the hashing part help to keep the sensitive information unreadable?

Having 200 bits about the original secret sequence, how much information could the spies gain now? If h was chosen u.a.r. from a universal hash family, we expect the spies to learn only $2^{1000-2000+200} = 2^{-800}$ bits about the message, which is 0 for all practical purposes.

2.4.4 De-randomized MaxCut via Hashing

The MaxCut problem is defined as follows.

●**Definition (Definition)** *Given a graph $G(V, A)$, partition V in two groups (red and blue vertices) such that the number of edges with differently coloured end points is maximized.*
More formally, we want to have a function $\phi : V \to \{red, blue\}$ to maximize

$$c(\phi) := |\{\{v, w\} \in A | \phi(v) \neq \phi(w)\}|.$$

The problem is NP-hard in general. A simple randomized Monte Carlo algorithm would be to colour each vertex red or blue with the respective probability of $1/2$. What is the expected size of the cut? Let X denote the number of cut edges, then we get:

$$E(X) = \sum_{\{v,w\} \in A} P(\phi(v) \neq \phi(w)) = \sum_{\{v,w\}} \frac{1}{2} = \frac{|A|}{2}$$

Ok, so expectedly we are not more than a factor 2 from the theoretical possible optimum. This is nice and – like for Max 3-SAT – this tells us, that there has to exist a colouring that leads to at least $|A|/2$ bi-chromatic edges. Again we could now construct a Las Vegas algorithm by running the Monte Carlo algorithm several times. But what about designing a deterministic polytime algorithm with the same approximation guarantee? Well, let H be a universal hash family that maps V to $\{0, 1\}$ with 0 encoding red and 1 encoding blue. If we choose $h \in H$ u.a.r. the expected value from above still applies, so this can be seen as an alternative randomized algorithm. Now remember that we can construct H such that $|H|$ is polynomial in the size of the universe. Therefore we can de-randomize our algorithm, by simply trying all those hash functions in H and return the colouring with the maximum c-value.

2.5 Random Sampling, VC-dimension and ϵ-Nets

Sampling is the principle to choose a small subset from a very large universe in order to gain information about the whole universe by extrapolating results derived from computations on the subset. A typical real-world example is election winner prediction via (comparably small) surveys. Another sampling application is *detection*, i.e. identifying elements of the universe with a certain characteristic. For example, not all cars are monitored by the police all of the time, as this would be impossible. But there are traffic checks at some positions and at some times in the hope to catch frequent speeders.

2.5.1 Basic Sampling Theorem

Let U be our universe of elements (e.g. people, objects, points) and $S \subseteq U$ a not too small subset. Then a random sample of U is likely to intersect S as expressed in the following theorem.

Theorem 2.15 (Sampling)**.** *Let $S \subseteq U$ be a subset of the universe such that $|S| \geq \epsilon|U|$ for some $\epsilon \in]0,1[$. Then a random sample R of size $1/\epsilon \ln 1/d$ from U intersects S with a probability of $1-d$.*

Proof. The probability for a single element u.a.r. chosen from U to be also in S is ϵ. Accordingly the probability of the element not to be contained in S is $1-\epsilon$. Hence the probability that none of the sampled elements is in S can be expressed as:

$$P(R \cap S = \emptyset) = (1-\epsilon)^{1/\epsilon \ln 1/d}$$

As $(1 - 1/n)^n$ converges to e^{-1}, we can upper bound this term as follows.

$$P(R \cap S = \emptyset) \leq e^{-\ln 1/d} = d$$

Hence the probability that R indeed intersects S is at least $1-d$. □

So this basic sampling theorem guarantees that for a given set S a random sample works well.

Example (Traffic Control) *In Germany, there are about 32 million drivers. About half a million people are banned from driving each year. Counting minor offences and regarding the estimated number of unreported cases, it seems safe to assume that at least 4 million drivers are speeders. This provides us with an ϵ-value of at least $1/8$. So if we want to catch a speeder with a probability of 90%, we have to draw a random sample of size $8 \ln 10$, i.e. we should check* 19 *drivers.*

But what if we are not only interested in a single subset S but a family of such sets $S_1, \cdots, S_m$? Of course, we could draw a suitable random sample for each of those sets and then build their union. But we will see that there are better ways to determine good samples representing a family of subsets sufficiently.

Exercise. How many randomly chosen voters should you consider today in Germany in a survey if you want to be 80% sure to include an FDP-supporter? Select four other parties and compute the sample size in order to have a supporter for each party with a probability of 80%.

2.5.2 Hitting Sets and Universal Sampling

As seen before the size of a good sample does not depend on the size of the universe itself, but solely on the ratio of the sizes of S and U (expressed by ϵ). In general we call a sample *universal* if its size is independent of $|U|$. Now we are given a family of different sized subsets $S_1, \cdots, S_m \subseteq U$. Can we get a single universal sample set to hit all S_i? The problem to hit a family of subsets of a universe with as few elements as possible is better known as HittingSet problem, one of the classical NP-complete problems.

Definition (HittingSet) *Let $(U, \mathcal{S})$ be a set system, i.e. U is a universe of elements and $\mathcal{S}$ a collection of subsets of U. Find a subset $R \subseteq U$ of the elements, such that $\forall S \in \mathcal{S} : R \cap S \neq \emptyset$ and $|R|$ is minimized.*

The problem is not only hard, but also hard to approximate. In fact, it was proven that there exists no polytime algorithm which can guarantee an approximation factor for HittingSet better than $\log n$ with $n = |U|$. So for general inputs, the size of the sample we need is dependent on the size of the universe, hence a universal sample is out of reach. If we restrict our family of subsets to S with $|S| \leq k$, there exist methods that solve HittingSet in polytime with an approximation guarantee of k. But again this is not what we want to have. Let S^* be the largest set in $\mathcal{S}$, if $|S^*| = \epsilon|U|$ we can only guarantee a $\epsilon|U|$ approximation, which obviously depends on the size of the universe. And it seems counter-intuitive, that the larger the sets the worse the approximation factor – as large sets should be easier to hit than small ones. So for general inputs, there is no hope for a good universal sample. But there are certain features of set systems, which allow to get beyond the inapproximabilty bounds of the HittingSet problem. In fact, the notions of ϵ-nets and VC-dimension describe when this is possible.

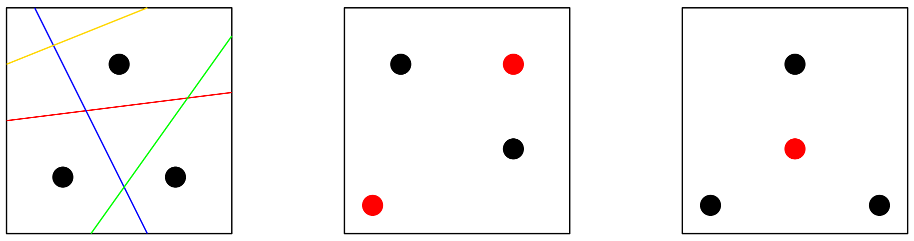
2.5.3 VC-dimension

The VC-dimension measures the complexity of a set system.

Definition (VC-Dimension) . *The VC-dimension d of a set system $(U, \mathcal{S})$ is defined as the size of the largest subset of U that can be shattered. Thereby, a subset $U' \subseteq U$ is called shattered if for any subset $A \subseteq U'$ there exists $B \in \mathcal{S}$ with $U' \cap B = A$.*

Lets consider some examples for clarity.

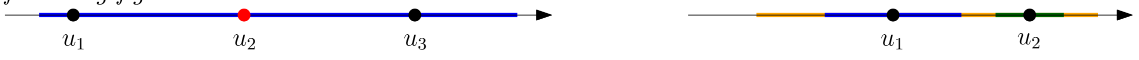
Example (Points and Half-Spaces) *Let U be points in $\mathbb{R}^2$ and $\mathcal{S}$ a collection of half-spaces. This set system has a VC-dimension of at most $d = 3$.*



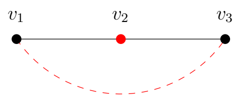
The left image shows three points in general position. This set of points can be shattered as indicated by the coloured lines. The image in the middle and the right one both show a set of four points that cannot be shattered, as the red point(s) cannot not be separated from the black ones with a straight line. To argue that no set of four points can be shattered (as required to nail the VC-dimension to 3), we make the following observation: For four points either the convex hull of the points is determined by 1, 2, 3 or four points obviously. If it is 1, all points are at the same position. As soon as two points have the same position they are inseparable and cannot be shattered. If it is 2, the points form a straight line, hence points that are not end points cannot be separated from both endpoints. If it is 3, we have the case shown in the right image, if it is 4 we have the case illustrated in the middle. So no point set of 4 or more points in the plane can be shattered by half-spaces, proving a VC-dimension of at most 3.

Exercise. Show that the VC-dimension of a system of half-spaces in $\mathbb{R}^n$ (i.e. hyperplanes) is at most $n + 1$. First show that there exist systems of $n + 1$ points that can be shattered. To show that there is no valid set of $n+2$ points that can be shattered, you can apply Radon's theorem. This theorem says that if S is a set of $n+2$ points in n dimensions, then S can be partitioned into two (disjoint) subsets S_1 and S_2 whose convex hulls intersect.

Example (Points and Intervals) *Let now U be points in $\mathbb{R}$ and $\mathcal{S}$ a collection of intervals. This set system has a VC-dimension of at most $d = 2$. Consider three elements u_1, u_2, u_3 in U with $u_1 < u_2 < u_3$. Any interval that contains u_1 and u_3 has to contain u_2 as well, so u_2 cannot be separated from the other elements. For any set of two points $u_1 < u_2$ we can have an interval containing both points, as well as one containing u_1 but not u_2 and vice versa as illustrated in the following figure.*



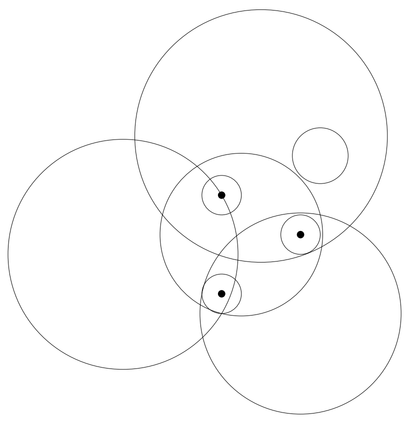
Example (Vertices and Unique Shortest Paths) *Given an undirected connected graph $G(V,E)$, let $U = V$ be the universe and $\mathcal{S}$ a collection of unique shortest paths in that graph (i.e. between any two points there exists only one shortest path). The VC-dimension of this system is 2. Consider a path of three vertices v_1, v_2, v_3, then like for the interval example, any path that contains v_1 and v_3 has to contain v_2 as well (otherwise there would be multiple shortest paths between v_1 and v_3). This is illustrated via the dashed line in the following image:*



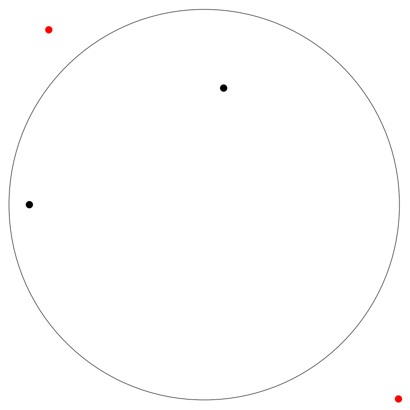
Two distinct vertices can be shattered as every vertex is a shortest path on its own and and a path containing both vertices can easily be constructed.

Exercise. What is the VC-dimension of a system of unique directed shortest paths?

Example (Points and Circles) *Let U again be points in the plane and $\mathcal{S}$ a set of circles. Three points can be shattered as shown in the next image:*



What about four points? Again consider the convex hull. If one point is inside the convex hull, the points on the convex hull cannot be separated from the such a point via a circle (this is true for all convex objects). So now assume all four points are indeed on the convex hull and form a quadrangle. We cannot use the same argument as for straight lines here, as indeed two diagonal points can be separated now, as shown here:



But what about the other two nodes outside this circle? Can they be separated via a circle from the other points as well? The answer is no, because if such a circle would exist, then the symmetric difference of the two considered circles would consist of 4 disjoint regions, which is impossible.

Exercise. What is the VC-dimension of a system of axis parallel rectangles? What is the VC-dimension of triangles in the plane?

Exercise. Think of an example of a set system with unbounded VC-dimension.

We show now, that the complexity of the system provides an upper bound on the number of possible sets in the system which is much tighter than the naive bound of 2^n.

Lemma 2.16 (Shatter Lemma)**.** *For a set system with VC-dimension at most d, a subset $U' \subseteq U$ of size m can intersect at most*

$$\sum_{i=0}^{d} \binom{m}{i}$$

sets in $\mathcal{S}$.

Proof. We will prove the lemma by showing that a set system with $|U| = n$ and VC-dimension d has at most $\sum_{i=0}^{d} \binom{n}{i}$ sets in total. For the proof via induction, consider some fixed element $u \in U$. For u we define a new set system $(U', \mathcal{S}')$ with $U' = U \setminus u$ and $\mathcal{S}' = \{S \setminus u | S \in \mathcal{S}\}$. How much can the set sizes of $\mathcal{S}$ and $\mathcal{S}'$ differ? The only way the removal of u can lead to fewer sets is then two sets $S_a, S_b \in \mathcal{S}$ are equal after removal of u (trivial sets can be excluded w.l.o.g.). We define $\mathcal{S}''$ as the collection that contains all those sets with $S_a = S_b \cup u$. Then obviously $|\mathcal{S}| = |\mathcal{S}'| + |\mathcal{S}''|$. We observe that according to the induction hypothesis $|\mathcal{S}'| \leq \sum_{i=0}^{d} \binom{n-1}{i}$ as $|U'| = n - 1$ and the VC-dimension of this system cannot increase by removal of an element. For $\mathcal{S}''$ the VC-dimension can be at most $d-1$, because if we can shatter a subset $A \subseteq U \setminus u$ with $\mathcal{S}''$ then we can shatter $A \cup u \subseteq U$ with $\mathcal{S}$ as well. Therefore it yields $|\mathcal{S}''| \leq \sum_{i=0}^{d-1} \binom{n-1}{i}$.

Summing up the bounds for $|\mathcal{S}'|$ and $|\mathcal{S}''|$ we finally get:

$$|\mathcal{S}| \leq \sum_{i=0}^{d} \binom{n-1}{i} + \sum_{i=0}^{d-1} \binom{n-1}{i} \leq \sum_{i=0}^{d} \binom{n}{i}$$

$\square$

Note that $\sum_{i=0}^{d} \binom{n}{i} \leq n^d$. So instead of being exponential in n for fixed d we now have a polynomial set size. Note that in general this bound is the best one can show, but for individual set systems even better bounds might exist. For our example with points and half-spaces, this lemma gives us a bound of n^3 but it is possible to show that $|\mathcal{S}| \leq n^2$ in this case.

How can we use this characteristic to draw good samples? Brönnimann and Goodrich proved in 1995 that hitting sets within a factor of $\mathcal{O}(d \log(d|OPT|))$ of the optimal can be computed in polynomial time with d being the VC-dimension and OPT denoting the optimal solution. Hence, if d is a constant, we have a $\log|OPT|$ approximation. This is not what we want to have in the end (as the optimum is not completely independent of $|U|$), but we are already below the inapproximability bound for HittingSet for many instances. As the results of Brönimann and Goodrich are based on ϵ-nets we will come back to their results after introducing ϵ-nets formally.

Note, that there are many applications for the VC-dimension besides the one we consider here. For example, in machine learning approaches the VC-dimension can tell us how complex a classifier needs to be to separate groups of data points from each other.

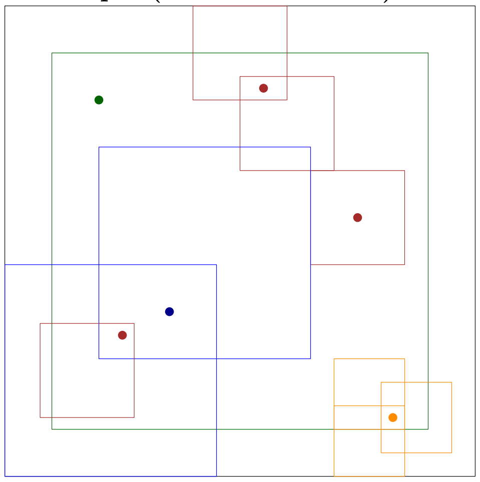
2.5.4 ϵ-Nets

The idea of an ϵ-net is closely related to HittingSet as to be observed in the formal definition.

Definition (ϵ-net) *Let $(U, \mathcal{S})$ be a set system and $\epsilon \in [0,1]$ a real number. A set $R \subseteq U$ is called an ϵ-net for $(U, \mathcal{S})$ if $\forall S \in \mathcal{S}$ with $|S| \geq \epsilon|U| : R \cap S \neq \emptyset$.*

So obviously a HittingSet for $(U, \mathcal{S})$ is also an ϵ-net for an arbitrary value of ϵ, the other way around a $0 - net$ is a HittingSet. For all $\epsilon > 0$ an ϵ-net does not need to be a HittingSet for the whole set system as well, though, but only a HittingSet for all sets in the collection that are large enough. Consider the following example for more clarity.

Example (Geometric ϵ-Net)



Let U be the black rectangle and $\mathcal{S}$ the green, blue, brown and orange sub-rectangles. For $\epsilon = 1/2$ the green dot would be a valid ϵ-net as only the green rectangle has to be hit. If $\epsilon = 1/4$ also the blue rectangles have to be considered. Selecting only the blue vertex suffices for this purpose as it is contained in the green and the two blue rectangles. For $\epsilon = 1/16$ the brown rectangles are of interest as well. Now the set of the three brown dots is a valid net. Shrinking ϵ to a value that requires to hit also the orange rectangles, a feasible ϵ-net would be the union of brown and orange points. As we now hit all elements in $\mathcal{S}$, this ϵ-net is also a Hitting Set.

How to compute an ϵ-net for given ϵ? Let $\mathcal{S}_\epsilon$ be defined as $\mathcal{S}_\epsilon := \{S \in \mathcal{S} | |S| \geq \epsilon|U|\}$. If we draw a random sample R of U with $|R| = r$, the probability of a certain $S \in \mathcal{S}_\epsilon$ not to be hit is $(1-\epsilon)^r$ as we have learned when considering the basic sampling theorem. So the probability that some $S \in \mathcal{S}_\epsilon$ is not hit is upper bounded by $|\mathcal{S}_\epsilon|(1-\epsilon)^r$. Accordingly the probability of all relevant sets

being hit is at least $1 - |\mathcal{S}_\epsilon|(1-\epsilon)^r$. If we choose $r = (c + \log|\mathcal{S}_\epsilon|)/\epsilon$ we get a success probability of at least $1 - e^{-c}$.
But of course this only makes sense if r does not get too large; we will see more sophisticated ways to construct ϵ-nets in the next chapter.

We also need to consider the weighted version of ϵ-nets for our analysis later on. The only thing that changes here, is that no longer the size of the subset matters, but a preassigned weight.

●**Definition (Weighted ϵ-Nets)** *Let $(U, \mathcal{S})$ be a set system, $w : \mathcal{S} \to \mathbb{R}^+$ a weight function, and $\epsilon \in [0,1]$ a real number. A set $R \subseteq U$ is called an ϵ-net for $(U, \mathcal{S})$ if $\forall S \in \mathcal{S}$ with $w(S) \geq \epsilon w(U) : R \cap S \neq \emptyset$.*
So setting $w(S) = |S|$ we have our conventional ϵ-net definition. We will see that weighting the sets cleverly will be an important ingredient to retrieve good samples.

Exercise. Let U be points in a square of side length a, and $\mathcal{S}$ a set of completely inscribed circles. Prove that for every $\epsilon \leq 1$ an ϵ-net of size π/ϵ exists, or in other words a set of circles with an area of at least $\epsilon \cdot a^2$ can be hit by π/ϵ points.

We will see in the next chapter that the existence of an ϵ-net of size c/ϵ for all values of ϵ (with c being a constant) gives rise to a constant approximation for HittingSet.

Exercise. Golden Exercise: Prove that for any system of undirected unique shortest paths and $\epsilon \in [0,1]$ there exists an ϵ-net of size c/ϵ for some constant value c. Attention: This is a so far unsolved problem! If you solve it, there is no need for you to show up at the oral exam :).

2.5.5 Approximation Algorithms Based on Small VC-dimension and ϵ-Nets

Remember our final goal was to achieve better approximations for the HittingSet problem than the pessimistic general inapproximability bound of $\log n$ (which makes no assumption about the VC-dimension of the set system). In fact, we would like to guarantee an approximation factor which is independent of $|U| = n$, i.e. some constant factor approximation, coinciding with a strategy to obtain an universal sample for a collection of subsets of the universe.

The basic ϵ-Net Theorem states that a random sample of sufficient size (not depending on $|U|$) is an ϵ-net with positive probability.

Theorem 2.17. *Let $(U, \mathcal{S})$ be a set system with VC-dimension d, and $\epsilon, \delta \in [0,1]$ parameters. A random sample of size*

$$\frac{d}{\epsilon}\log\frac{d}{\epsilon} + \frac{1}{\epsilon}\log\frac{1}{\delta}$$

is an ϵ-net with probability $\geq \delta$.

A slightly different (and also often found) formulation will be used to simplify the proof:

Theorem 2.18 (ϵ-Net Theorem)**.** *Let $(U, \mathcal{S})$ be a set system with finite VC-dimension $d \geq 2$ and $\epsilon \in]0, 1/2[$ a parameter, then there exists an ϵ-net for $(U, \mathcal{S})$ of size $\mathcal{O}(d/\epsilon \log 1/\epsilon)$.*

We will invoke the probabilistic method to prove this theorem, i.e. we will show that a random sample of size $cd/\epsilon \ln(1/\epsilon)$ has a positive probability to be an ϵ-net for $(U, \mathcal{S})$. If this is the case, existence follows. We first need a small helping lemma.

Lemma 2.19 (Chernoff-Bound)**.** *Let $X = \sum_{i=1}^n X_i$ be a random variable, with the X_i being independent of each other and $P(X_i = 1) = p$. Then $P(X \geq np/2) \geq 1/2$ if $np \geq 8$.*

Proof. Remember Chebyshev's inequality $P(|X - E(X)| > t\sigma) \leq {}^1\!/_{t^2}$ for $t > 0$. For a binomial distribution we know that $E(X) = np$ and $Var(X) = \sum_{i=1}^{n} Var(X_i) \leq np$. So we get:

$$P\left(X < \frac{np}{2}\right) \leq P\left(|X - E(X)| > \frac{np}{2}\right) \leq \frac{4}{np} \leq \frac{1}{2}$$

□

Now we have all ingredients to prove the ϵ-Net Theorem.

Proof. Remember that we want to show that a random sample of size $r = {}^{cd}\!/_{\epsilon} \ln({}^1\!/_{\epsilon})$ has a positive probability to be an ϵ-net for $(U, \mathcal{S})$. For this proof we argue about two random samples R, R' both of size r. We define E as the event that R is not an ϵ-net, i.e. it exists $S \in \mathcal{S} : R \cap S = \emptyset$. The goal is to show that $P(E) < 1$, as this would imply $P(\overline{E}) > 0$. To show that this indeed is true, we consider a second event E' defined as the case that there is $S \in \mathcal{S} : S \cap R = \emptyset$ and $|R' \cap S| \geq {}^{\epsilon r}\!/_2$. As the event E' implies E but has an additional requirement, we obviously get $P(E') \leq P(E)$ (which does not help much as we need an upper bound on E, but we will get there). Now consider $P(E'|R)$ that means R is fixed and R' is chosen randomly and implies E'. If R is an ϵ-net, then E' is impossible and the conditional probability is 0. Otherwise, let $S^* \in \mathcal{S}$ be a witness for R not being an ϵ-net. Then we get $P(E|R) = 1$ and $P(E'|R) \geq P(|R' \cap S^*| \geq {}^{\epsilon r}\!/_2)$.

Exercise. Why $\geq$ and not $=$?

The last term can be lower bounded via the Chernoff lemma by ${}^1\!/_2$. Therefore we can conclude $P(E|R) \leq 2P(E'|R)$. As this is true for all R, this is the same as stating $P(E) \leq 2P(E')$. It remains to show that $2P(E') < 1$. For this, let A be a random sample of size $2r$. Then subsequently we divide the elements in A into R, R' again using randomization. So for fixed A there are $\binom{2r}{r}$ possibilities to create R, R'. Let A be fixed and additionally fix $S \in \mathcal{S}$. We define $P_S = P(R \cap S = \emptyset, |R' \cap S| \geq {}^{\epsilon r}\!/_2 | A)$. Obviously, if $|A \cap S| < {}^{\epsilon r}\!/_2$ then $P_S = 0$. Otherwise $P_S \leq P(R \cap S = \emptyset | A)$. This term can be bounded as follows:

$$P(R \cap S = \emptyset | A) \leq \frac{\binom{2r - {}^{\epsilon r}\!/_2}{r}}{\binom{2r}{r}} \leq \left(1 - \frac{{}^{\epsilon r}\!/_2}{2r}\right)^r \leq e^{-\left(\frac{{}^{\epsilon r}\!/_2}{2r}\right) r} = e^{-\frac{\epsilon r}{4}} = e^{-\frac{cd \ln({}^1\!/_{\epsilon})}{4}} = \left(\frac{1}{\epsilon}\right)^{-\frac{cd}{4}}$$

Now we use the shatter lemma to determine the number of sets in $\mathcal{S}$ that A can intersect in order to compute $P(E'|A)$. So we set $m = 2r$ in the lemma, and get:

$$P(E'|A) \leq \left(\frac{{}^{2re}\!/_{\epsilon}}{d}\right)^d \cdot \left(\frac{1}{\epsilon}\right)^{-\frac{cd}{4}} = \left(\frac{2ec}{\epsilon} \ln\left(\frac{1}{\epsilon}\right) \cdot \left(\frac{1}{\epsilon}\right)^{-\frac{c}{4}}\right)^d$$

Inserting that ${}^1\!/_{\epsilon} > 2$ we get:

$$P(E'|A) \leq \left(ec \ln\left(\frac{1}{\epsilon}\right) \frac{1}{2^c}\right)^d$$

With $d \geq 2$ and c large enough we can therefore assure that $P(E'|A) < {}^1\!/_2$ for any A and hence $P(E') < {}^1\!/_2$. □

Matousek showed that such a net of size $\mathcal{O}({}^d\!/_{\epsilon} \log {}^1\!/_{\epsilon})$ can be found in polynomial time (with the VC-dimension determining the exponent). So we are able to compute good ϵ-nets based on small VC-dimension of the set system right now, but our final goal was to compute an approximate HittingSet solution. Therefore, we now want to have a closer look at the results of Brönimann and Goodrich. Remember, their technique allows to compute a HittingSet of size $(dr^* log(dr^*)$ with

$r^* = |OPT|$. Their deterministic polytime algorithm requires two ingredients: a net-finder and a verifier. The verifier has to return for a net-suggestion R that either R is a valid net or a violator, i.e. some set S that is not hit by R. A net-finder can formally be described as follows.

●**Definition (Net-Finder)** *For a non-decreasing function s an s-net-finder for a set system $(U, \mathcal{S})$ with a weight function $w : U \to \mathbb{R}^+$, is an algorithm that for given r computes an $\epsilon = 1/r$-net of size $s(r)$.*

The main results of Brönimann and Goodrich are summarized in the following theorem.

Theorem 2.20. *Given a set system $(U, \mathcal{S})$, an s-net-finder and a verifier, then there exists an algorithm which computes a HittingSet of size $s(4r^*)$ in polytime.*

The algorithm runs as follows. At the beginning all elements in U have a weight of 1 and accordingly $\forall S \in \mathcal{S} : w(S) = |S|$. Now we apply the net-finder with $s = 1/2r^*$ (we will see in the end why we can assume r^* to be known). Then we run the verifier. If the returned net is a HittingSet, we are done. Otherwise we have a witness $S^* \in \mathcal{S}$ for which yields S^* is not hit by the actual net. We proceed by doubling all the weights of elements in S^*. When the net-finder and the verifier is invoked again. The whole process is repeated until the verifier reports the found net to be a valid HittingSet for the set system. Note, that the total weight of the universe cannot increase too much by the weight doubling, as we always know that $1/2r^* > w(S^*)$. In fact, the number of iterations we have to perform before the weight increase ensures us to have found a HittingSet can easily be bounded as stated in the next lemma.

Lemma 2.21. *With r^* being the optimal size of a HittingSet, no more than $6r^* \log(n/r^*)$ iterations are possible. The total weight of the universe cannot exceed n^4/r^{*3}, i.e. it stays polynomial in the input size.*

Proof. Let k be the number of iterations. Observe again, that the set S^* returned as violator in a round satisfies $w(S^*) < w(U)/2r^*$ as otherwise it would have been large enough to be hit by the ϵ-net automatically. Therefore the total weight of U is multiplied in each round with at most $(1 + 1/2c)$. So we can bound the weight of the universe as follows.

$$w(U) \leq n \left(1 + \frac{1}{2c}\right)^k \leq ne^{(k/2c)}$$

The weight of the computed solution L can be estimated as $w(L) \geq \sum_{u \in L} w(u)$. As in each round at least the weight of a single element gets doubled, we conclude that $w(L) \geq r^* 2^{k/r^*}$ as L must naturally contain at least r^* many elements; and as we perform k doubling rounds we have to include at least one element with weight $w() \geq 2, w() \geq 4 \cdots w() \geq 2^k$. So we get the following inequality:

$$r^* 2^{k/r^*} \leq w(L) \leq w(U) \leq ne^{(k/2c)}$$

Obviously $w(L) \leq w(U)$ because L is a subset of U. If we rearrange the formula to separate k we get:

$$k(2log(2) - log(e)) \leq 2r^*(log(n) - log(r^*))$$

And so:

$$k \leq 6r^* log(n/r^*)$$

If we insert this term in the upper bound for $w(U)$ we get $ne^{3\log(n/r^*)} = n^4/r^{*3}$. □

Why it is ok to assume r^* to be known? Lets say we have a guess r of this size, with $r \leq r^*$ (e.g. $r = 1$). If r is too small we will observe this in our algorithm, because after

$$6r \log(n/r)$$

the verifier will still report that we have not found a feasible HittingSet. The trick is to run the algorithm again in this case, now with the guess $2r$ and so on. As soon as $2^z r \geq r^*$ the algorithm returns a solution.

Exercise. What is the runtime of the complete algorithm? What is the overhead compared to knowing r^* from somewhere?

So Brönimann and Goodrich provided us with a polytime algorithm that for constant VC-dimension finds a solution within $\log(r^*)$ of the optimum r^*. But already in their paper they come up with examples where the approximation ratio is even better, namely a constant (for selected geometric problems). A result by Clarkson gives some insight when this is possible in general.

Theorem 2.22. *If for a set system $(U, \mathcal{S})$ and arbitrary $\epsilon \in [0,1]$ there exists an ϵ-net of size c/ϵ with c an ϵ-independent constant, then we can find an approximate HittingSet for $(U, \mathcal{S})$ in polytime which is at most a factor c away from the size of the optimal solution r^*.*

So remember our circles in the square example where we showed the existence of an ϵ-net of size π/ϵ for every possible ϵ. According to this result, we can find a π-approximation of the corresponding HittingSet problem in polytime. If you accomplish the golden exercise, there would be a c-approximation for HittingSet problems on unique shortest paths. This would have a variety of real-world applications as we will study in more detail in the next section.

2.5.6 Application: Route-Planning

The rather abstract notions of ϵ-nets and VC-dimension of set systems can indeed be used to design practical algorithms. One important area of application is learning, but here we will focus now on route planning. The connection between VC-dimension, ϵ-nets and route planning techniques was only recently explored and is an active area of research right now (e.g. by Microsoft).

Exercise. Implement a random graph generator. For this purpose randomly locate points inside a rectangle and then connect points with an edge which have a pairwise distance smaller than some bound B. Then implement a shortest path extractor (for our unweighed graph here BFS suffices) to get all shortest path which are at least k long (i.e. contain k or more nodes). Find a HittingSet for those paths. Use random sampling (until a HittingSet is found), and the conventional greedy algorithm (always choose the next point as the one hitting most so far unhit sets) and compare the results. If you like, come up with your own (competitive) algorithm.

We will now have a closer look at some applications of HittingSets for path systems (which you could all tackle with your implementation).

Transit Nodes For navigation systems and route planning engines the most important task is to compute shortest paths between two locations in a street graph $G(V, E)$ as efficient as possible. The conventional way to do this is to use Dijkstra's algorithm. Unfortunately for larger networks Dijkstra requires processing times in the order of seconds, which is prohibitive for interactive use. Therefore several speed-up techniques have been developed in the last years to accelerate query answering.

The *transit node* approach allows to find shortest paths in large graphs in the order of microseconds, so a million times faster than plain Dijkstra. The basic idea behind this approach is that if you leave from a specific location (e.g. your home) and you drive to another far away location, there exists a concise set of nodes of which at least one will be included in the optimal shortest path (think of all near-by slip roads). And moreover, your direct neighbours leave the local area via exactly the same set of nodes as you. The union of all such important nodes over which people are forced to travel if they drive a longer tour is called the transit node set $T \subseteq V$. The transit nodes for a single location $v \in V$ are called the access nodes of v. To speed up shortest path computations with transit nodes, the following approach is used: First, we need to find a set T not too large. Then for every pair of nodes $t, t' \in T$ the shortest path distance is computed with Dijkstra and stored in a look-up table. Also for every node $v \in V$ the access node set along with the shortest path distance to each access node are pre-computed. Consider now a query asking for the shortest path from v to w. If their distance is very small (for some notion of small), we run a local Dijkstra. Otherwise, for every access node t of v and every access node t' of w the total path distance via those two transit nodes is computed (with the help of the look-up table). In the end the shortest path is determined by t, t' which minimize the distance from v to w. A good transit node implementation leads to access nodes sets of a size about 5 for v and w, so a query consists mainly of $5 + 5 \cdot 5 + 5 = 35$ look-ups.

So the crucial part is to find a concise set T, because otherwise the space consumption gets too high (we need to store $\Omega(|T|^2)$ shortest path distances). There exist many good heuristics nowadays to find a feasible set T. One way would be to formulate the problem as an instance of HittingSet: Lets have a fixed notion of a small query, e.g. nodes with the shortest path exhibiting only 20 nodes. So to hit all longer paths, we need to extract shortest paths which have a length of 21 nodes (in fact all with ≥ 21 nodes, but this is redundant). If we can find a small HittingSet for this set system, we have a concise set of transit nodes. Knowing that the VC-dimension of a system of shortest paths is a constant, in particular $d = 2$ (for undirected shortest paths) or $d = 3$ (for directed shortest paths), we can apply the ϵ-net theorem here. Let k be the number of nodes required for a long shortest path (here $k = 21$), our ϵ is ${}^{k}/_{n}$ with $n = |V|$ the size of the universe. Accordingly we can find a HittingSet of size $\mathcal{O}({}^{n}/_{k} \log({}^{n}/_{k}))$ in polytime. And a random sample of this size will be the required solution with high probability. The results by Brönimann and Goodrich tell us that also an approximation guarantee of $\log(|OPT|)$ can be achieved.

Facility Location Problems With the same approach as for transit node computation we can tackle facility location problems in street networks. In a facility location problem, we want to place certain objects on nodes in the street graph. Think e.g. about gas stations. It would be reasonable to require that every shortest path of length 50km or longer has to have a gas station on it somewhere, so people do not run out of fuel. In another scenario, we could demand to have a sign at least all 20km on interstates and highways which tells about the nearest cities in driving direction. Note, that it is not crucial to have a fixed length for each path in the set system. Consider e.g. battery-powered electric vehicles. Their cruising range is terrain dependent, as on flat terrain they hold on much longer compared to going uphill. If we want to cover the street network with loading stations for electric vehicles such that an initially fully loaded vehicle never runs out of energy while driving on a shortest path, the length of the paths in the set system (namely all minimal length path on which the vehicle would get stranded) differ vastly.

Exercise. If the path lengths in the system are denoted by distances (50km, 20km) instead of number of nodes (as in the transit node example), how can we transform the problem such that the ϵ-net theorem can be applied nevertheless?

Two interesting heuristics in the context of facility location problems are adaptive sampling and pruning. Both algorithms require the nodes in V to be ordered (e.g. randomly).

In the adapative sampling approach we start with an empty HittingSet C. When we parse through the nodes one by one in the fixed order and check if all shortest paths of required length originating in the current node v are already hit by C (by running Dijkstra sufficiently long). If this is not the case, we add v to C. Obviously, after having considered all nodes, the resulting set C is a HittingSet.

The pruning approach looks at the problem from the completely other side. It starts with $C = V$, so every node is in the cover at the beginning. Then it also parses through the list of nodes. Now the crucial question for a node v is, if the node has to maintain in C in order to maintain the characteristic that C is a HittingSet.

Exercise. Design an algorithm that for a HittingSet C of all long paths in the graph and a node $v \in C$ decides whether v is important for the solution.

❓**Exercise.** Implement adaptive sampling and pruning. Which approach is faster? Compare the resulting set sizes for several runs of the algorithms with varying node orders (e.g. new random order in each round).

Graph Simplification Consider a huge street graph stored on a server, and a mobile client which wants to render a part of the graph on a certain zoom-level. In a zoomed-out view it does not make sense to transmit every single node in the network to the client (as this is costly in terms of bandwidth and rendering), but it may suffice for a path to send e.g. every 10th node. To simplify the total graph based on this idea, we would like to have a HittingSet for every zoom level, so for a high zoom level we like to hit all paths of length k with k being small, for a low zoom level k should be large. Especially for large values of k the $\log(|OPT|)$ approximation algorithm promises concise HittingSets.

❓**Exercise.** To avoid artificial artefacts, we would like to have that the HittingSet C_1 for the lowest zoom level is a subset of C_2 for the next higher zoom level and so on. Design an algorithm to compute such a sequence of HittingSets for given $k_1, k_2, \cdots, k_r$.

Some Hints for the Golden Exercise Let $G(V, E)$ be the graph and P_ϵ the set of all unique shortest paths in G which contain $\epsilon \cdot n$ nodes (with $n = |V|$). We want to show that for $\epsilon \in [0, 1]$ every path p in the collection P_ϵ can be hit using only c/ϵ many nodes for some global constant c. We first proof a simple lemma.

Lemma 2.23. *If $\forall p, p' \in P_\epsilon : p \cap p' = \emptyset$ or $|p \cap p'| \geq \epsilon \cdot n/2$ then we can find a HittingSet for P_ϵ of size $2/\epsilon$.*

Proof. Let P^* be a maximal set of paths in P_ϵ that are pair-wisely intersection free. So every path in P_ϵ but not in P^* has to intersect a path in P^*, and the intersection has to contain at least $\epsilon \cdot n/2$ nodes. As we know that the VC-dimension of the set system is at most 2, we know that the intersection of two unique shortest paths has to form a continuous subpath. So if we choose the two nodes in the middle of every path in P^* to be part of the HittingSet, all paths in P_ϵ are hit automatically. How many paths can there be in P^*? As the paths are intersection free and each has a length of $\epsilon \cdot n$, at most $n/\epsilon \cdot n = 1/\epsilon$ paths can be contained in P^*. Selecting two nodes from each of the paths we have a total HittingSet size of $2/\epsilon$. □

Consider $\epsilon > 2/3$. The lemma above immediately provides us with a HittingSet of size at most 2, as for ϵ in this range all paths in P_ϵ have to have an overlap of at least $\epsilon \cdot n/2$ nodes.

We go on considering $\epsilon \in]1/2, 2/3]$. If the lemma from above applies, we have a HittingSet of $2/\epsilon$. But what if this is not the case?

Lemma 2.24. *For $\epsilon \in]1/2, 2/3]$, the set system P_ϵ can be hit with 3 nodes.*

Proof. If the lemma from above does not apply, we know that there exist $p, p' \in P_\epsilon$ with $0 < |p \cap p'| < \epsilon \cdot n/2$. Let the intersection path of p, p' be denoted as p^* and $|p^*| = \delta \cdot n$. Then p, p' are dissected each in three sub-paths, a prefix, p^* and a suffix (with the prefix or the suffix possibly being empty). We now suggest the following nodes to form the HittingSet: a node in the middle of p^*, a node in the middle of the prefix or suffix path of p, using the longest of these two, and the same for p'. We have to show that this set hits all possible paths. How could an unhit path be composed? If it intersects p^* it can contain at most $\delta \cdot n/2$ nodes of p^* and $(\epsilon \cdot n - \delta \cdot n)/2$ further nodes from p or p' plus nodes not in p or p', which are $n - 2\epsilon \cdot n + \delta \cdot n$ many. This makes in total

$$\frac{\epsilon \cdot n}{2} + n - 2\epsilon \cdot n + \delta \cdot n$$

many nodes. Using the fact that $\delta < \epsilon/2$ we get a bound on the length of a possible unhit path of $n - \epsilon \cdot n$. For $\epsilon > 1/2$ this term is clearly upper bounded by $\epsilon \cdot n$. So the unhit path is not long enough to be in P_ϵ. If the unhit path does not intersect p^*, it can have at most

$$2\frac{\epsilon \cdot n - \delta \cdot n}{2}$$

nodes shared with p and p'. That makes together with the nodes neither in p nor in p' again at most $n - \epsilon \cdot n < \epsilon \cdot n$ for $\epsilon > 1/2$ nodes. □

3 Getting Beyond Deterministic Bounds via Randomization

In this chapter we consider problems with lower bounds for any deterministic strategy which can be broken when introducing randomization. Further information for the Closest Pair problem are available here: https://www.cs.ucsb.edu/~suri/cs235/ClosestPair.pdf, for Min-Cut here http://sarielhp.org/teach/courses/473/notes/12_mincut.pdf, for Sublinear Algorithms see https://www.cs.princeton.edu/~chazelle/pubs/mstapprox.pdf and for the Long Path problems http://www.google.com/url?sa=t&rct=j&q=&esrc=s&source=web&cd=1&ved=0CBwQFjAA&url=http%3A%2F%2Fwww.cs.cmu.edu%2Fafs%2Fcs%2Facademic%2Fclass%2F15859-f04%2Fwww%2Fscribes%2Flec1.ps&ei=kKrXU_uuPKrMsQTo5IKoDA&usg=AFQjCNFDqOMBLezg7FfmhRHqtVeux303hg&sig2=2z20cj_sRiIU7At5o1kzpQ&bvm=bv.71778758,d.cWc.

3.1 The Closest Pair (CP) Problem

Given a set S of n points in the plane, the goal is to identify two nodes such that no other pair of nodes exhibits a smaller distance. The problem can easily be solved in quadratic time, by computing the distances between any two nodes and keeping track of the minimum. But we can do better when using a Divide-and-Conquer algorithm, namely we can find the CP in $\mathcal{O}(n \log n)$.

3.1.1 Deterministic Divide-and-Conquer Algorithm

Dividing works as follows:

1. If $|S| \leq 10$, brute force closest pair, let the distance be d, else go to 2.
2. Compute the median x-coordinate x^* of all points in S.
3. Subdivide S into S_1, S_2 with $p \in S_1$ if $x(p) < x^*$, $p \in S_2$ if $x(p) > x^*$, and the points with $x(p) = x^*$ such that $|S_1| = |S_2| = |S|/2$
4. Perform recursively steps 1.-3. on S_1 and S_2.

Obviously the recursion has a depth of $\log n$ and a runtime of $\mathcal{O}(n \log n)$. Now comes the conquering part. Let d_1, d_2 be the closest pair distances of nodes only in S_1 or only in S_2 respectively, and $\delta = \min(d_1, d_2)$. The closest pair of points in $S = S_1 \cup S_2$ is the either the pair of points responsible for δ or it is formed by a node $p_1$in S_1 and a node p_2 in S_2. With x^* being the median of the x-coordinates of points in S it becomes clear, that the x-coordinates of p_1, p_2 have to lie in a δ-interval around x^*. But of course theoretically all $n/2$ points in S_1 and all $n/2$ points in S_2 could fulfil this requirement. Then the amount of work that has to be done to check all possible pairs is $n^2/4$. Therefore with this approach we would not win anything compared to the naive approach. But we can do better, by respecting the y-coordinates of the points as well. In fact for a point $p \in S_1$ all possible CP candidates have to lie in a $\delta \times 2\delta$ rectangle, as of course also the y-coordinate of p_2 has to be within δ of $y(p_1)$. Why does this help? If all $n/2$ points in S_2 lie inside this rectangle, we would gain nothing. But the crucial observation here is, that this is impossible. We know, that the CP in S_2 has a distance of at least δ. A rectangle with side lengths

$\delta \times 2\delta$ containing 7 or more points enforces a CP with distance $< \delta$. Therefore we have at most 6 potential candidates for p_2. Hence checking all candidate pairs can be performed in $6^n/2 \in \mathcal{O}(n)$ time. But this requires that we have access to those p_2 candidates for each p_1 efficiently. One way to achieve this would be sorting the points by y-coordinate, but this would require $\mathcal{O}(n \log n)$ time on each level. But sorting on the lowest level can be accomplished in constant time. When joining the sets back in the conquer step, we can merge two presorted sequences in linear time to get the final sorted sequence. So the total runtime of this Divide-and-Conquer algorithm can be expressed as $T(n) = 2T(n/2) + \mathcal{O}(n)$, which boils down to $T(n) \in \mathcal{O}(n \log n)$.

3.1.2 Deterministic Lower Bound

But maybe there is an algorithm requiring only $\mathcal{O}(n \log \log n)$ time or even better? As long as we talk about deterministic algorithms, the answer is no. We show that by reducing the Element Uniqueness Problem to CP, for which a lower bound of $\mathcal{O}(n \log n)$ is known, that CP cannot be solved faster with a deterministic algorithm. The Element Uniqueness Problem asks for a set of numbers $x_1, x_2, \cdots, x_n$ if all of those elements are unique. Obviously, the problem can be transformed into a CP problem, by creating a point (x_i, x_i) for each number in linear time. If the resulting CP has a distance of 0, the elements in the set are not all unique for sure (this can be checked in constant time). So if we could solve CP in $T(n)$ faster than $\mathcal{O}(n \log n)$, then we could also solve Element Uniqueness in $T(n) + \mathcal{O}(n)$, which would result in a contradiction.

3.1.3 Randomized Incremental Algorithm

We now introduce a randomized algorithm for CP which runs in $\mathcal{O}(n)$. It is based on considering the points iteratively and maintaining a dynamic grid data structure.

The algorithm works as follows. Let $p_1, p_2, \cdots, p_n$ the points in some fixed order. Lets assume for the moment we know the CP distance δ of the points $p_1, \cdots, p_{i-1}$ from somewhere. Now we are interested what happens when adding p_i to the set. Does p_i create a new CP, and if yes with which distance? Of course, we could naively compute the distance to all previous nodes. But then the total runtime would be quadratic again in the end. So we would like to have an oracle which tells us which points are possibly close enough to p_i to form a CP with p_i; and the number of such candidates should be small. For that purpose, we construct a grid with grid length δ and sort the points $p_1, \cdots, p_{i-1}$ in the respective cells. This can be achieved in time $\mathcal{O}(i)$. Now let c_{kl} be the cell that contains p_i. How many cells do we have to check for a possible CP mate? As every cell has a side length of δ, all candidates have to lie either in c_{kl} or in of the surrounding cells (i.e. $c_{ab}, a \in \{k-1, k, k+1\}, b \in \{l-1, l, l+1\}$). So there are only 9 cells in total to be checked. How many points can lie inside those cells? Well, as the actual CP distance is δ, a single cell can accommodate no more than 4 points. So in total only 36 distances have to be computed, in order to check if p_i forms a new CP or not.

Exercise. Consider the CP problem in dimension d and an analogue grid construction scheme. How many cells and points have to be checked then?

If p_i does not reveal a new CP, the grid structure stays unaffected when considering p_{i+1}. If p_i leads to a new CP distance $\delta' < \delta$, then the grid needs to be reconstructed. But with the reconstruction taking $\mathcal{O}(i)$ time, the overall runtime could still be quadratic if we have to reconstruct the grid in many iterations.

Exercise. Construct a CP instance, for which the grid based algorithm needs quadratic runtime.

Now randomization comes to help. We do not use an arbitrary order of points, but a random one (with an equal probability for each permutation). What is the probability of an iteration to be expensive, i.e. that the grid has to be reconstructed after inserting p_i? Well, the probability that p_i is part of the CP when considering the points in $p_1, \cdots, p_i$ is $\leq {}^2\!/_i$ as 2 of the i points form a CP.

❓**Exercise.** Why $\leq$ and not $=$?

So the expected runtime can be described as

$$T(n) = \sum_{i=3}^{n} ({}^2\!/_i \cdot \mathcal{O}(i) + {}^{i-2}\!/_i \cdot \mathcal{O}(1))$$

using the linearity of the expected value. Obviously both summands collapse to a constant value, resulting in an expected runtime of:

$$T(n) = \sum_{i=3}^{n} \mathcal{O}(1) = \mathcal{O}(n)$$

3.2 The MinCut Problem

We now consider another problem from the area of graph theory, more precisely of robust network design. In the MinCut problem, we are given a network, and the goal is to remove as few edges as possible to disrupt the network in two separated components. For example, let the nodes in the network be members of a terrorist group and edges indicate which members can communicate directly. The question would be which communications have to be prevented to split the members into two groups without any communication in between. And as this is risky business, we would like to sabotage a minimum number of pair-wise communication protocols.

More formally, we are given an undirected graph $G(V, E)$ and ask for a partitioning of V into $V = A \uplus B$ inducing a minimum cut, where a cut is defined as the number of edges that have one endpoint in A and the other in B, i.e. $c(A, B) = |\{e = \{v, w\} \in E | v \in A, w \in B\}|$.

There exists a variety of deterministic algorithms that solve the MinCut problem to optimality. Many of those are based on flow algorithms. The fastest algorithm runs in $\mathcal{O}(n^3)$ and is quite difficult to understand and implement. In the following, we will describe a series of randomized algorithms for solving the MinCut problem. A rather simple randomized algorithm, called *FastCut*, will produce the optimal MinCut with high probability in $\mathcal{O}(n^2 \log^3(n))$.

3.2.1 Guessing Algorithm

The basic operation in all the randomized algorithms introduced in the following is the *edge contraction.* Here an edge $e = \{v, w\}$ is removed from the graph, and the two endpoints v, w are joined into a single vertex with label vw. All other edges which were previously incident to v or w are maintained, resulting in a new graph G' which possibly is a multi-graph (if edges $\{u, v\}, \{u, w\}$ were present before, we now have two edges $\{u, vw\}$).

❓**Exercise.** Show that edge contraction can be implemented in $\mathcal{O}(n)$ using a suitable graph representation.

If we recursively contract edges in G until only two nodes remain (because contraction reduces the number of nodes by one in each round), we end up with a partitioning of V in two subsets, namely the vertices that contributed to the label of the one vertex and the ones that contributed to the label of the other. The remaining edges in this graph (neglecting self-loops, which can be thrown away anyhow) form then also a cut in the original graph.

What can we say about the MinCut in the original graph and the MinCut in a graph after edge contraction? Obviously, the size of the MinCut cannot decrease with this operation. Contracting an edge could be interpreted as fixing that the two endpoints are in one partition in the end. So only edges are removed that do not contribute to the cut in the end. But unfortunately the MinCut size might increase during the edge contraction process. Think of the K_n, so the complete graph on n vertices and an extra vertex v^* which is connected to one vertex v in the complete graph via a single edge. The MinCut in this graph has size 1. But if we contract the edge $\{v, v^*\}$, the MinCut size becomes suddenly $n-1$. But if we pick edges for contraction randomly, the chance to pick this particular edges is rather small. We will show now that this is true for any kind of graph. In fact, we will show that there is a positive (but small) probability, that iteratively contracting randomly selected edges results in a correct MinCut in the end. The pseudocode of the simple guessing algorithm looks like this: So the algorithm requires $n-2$ rounds of edge contraction,

Algorithm 5: GUESSMINCUT(G)

```
1 begin
2     for $i = n$ downto 2 do
3         pick random edge $e \in E$;
4         $G \leftarrow G \setminus e$;
5     return only cut in G;
6 end
```

and each round takes linear time. Therefore the total runtime of the guessing algorithm is $\mathcal{O}(n^2)$. But what is the success probability of this algorithm?

Let the optimal MinCut size be denoted by k. How many edges have there to be in the graph at least? Well, if there would be a node with a degree smaller than k, then this node would induce a MinCut smaller than k. Therefore every node in the graph has to have a degree of at least k, providing us with a lower bound of $kn/2$ many edges. Our algorithm is successful if never a MinCut edge gets contracted. In the first round the chance to screw up, i.e. picking a MinCut edge, is only:

$$\frac{k}{\frac{kn}{2}} = \frac{2}{n}$$

So the success probability is

$$1 - \frac{2}{n}.$$

But intuitively the more edges we contracted the more likely it gets to pick a MinCut edge. As we are only correct in the end if we avoid MinCut edges all of the time, the success probability is the product of the success probabilities of each round, i.e.

$$P(n) \geq \left(1 - \frac{2}{n}\right) P(n-1)$$

with $P(n)$ denoting the success probability in a graph with n nodes. The base case of the recurrence is $P(2) = 1$, as there nothing has to be done by our algorithm and we are correct for sure. We can expand the recurrence into a product with most of the terms cancelling out:

$$P(n) \geq \prod_{i=3}^{n} \left(1 - \frac{2}{i}\right) = \prod_{i=3}^{n} \frac{i-2}{i} = \frac{\prod_{i=3}^{n} i-2}{\prod_{i=3}^{n} i} = \frac{2}{n(n-1)}$$

So the chance to find the actual MinCut with the guessing algorithm is pretty low. But at least we have a positive success probability, i.e. if we run the algorithm multiple times the chance of success increases.

Algorithm 6: KARGERMINCUT(G)

```
begin
    $k \leftarrow \infty$;
    $minX \leftarrow NULL$;
    for $i = 1$ to $B$ do
        $X \leftarrow$ GUESSMINCUT(G);
        if $|X| < k$ then
            $k \leftarrow |X|$;
            $minX \leftarrow X$;
    return $minX$;
end
```

3.2.2 Amplification

The guessing algorithm is a typical MC algorithm, the runtime is fix but the outcome depends on the random order in which the edges are chosen. But turning the MC algorithm into a LV algorithm is not that easy, because we do not have an easy checker which tells us if the produced MinCut is indeed the optimum. What can be easily accomplished instead is to decide which of two given results is better by comparing the cut sizes. This leads to the following algorithm, where the runtime is modulated via a parameter B and the outcome depends on chance. The algorithm is named after its designer Karger.

So the Karger MinCut algorithm returns the correct result if at least one of B calls to the guessing algorithm results in a valid MinCut. Accordingly, it only fails if B calls fail, so the success probability can be expressed as:

$$1 - \left(1 - \frac{2}{n(n-1)}\right)^B$$

Using the well-known inequality

$$1 - x \leq e^{-x}$$

we can lower bound the success probability with

$$1 - e^{\frac{-2B}{n(n-1)}}.$$

So choosing a large B we can push the success probability to a value close to 1. If we set $B = c \cdot \left(\begin{array}{c} n \\ 2 \end{array}\right) \cdot \ln(n)$ for some constant c, then KARGERMINCUT is correct with probability at least

$$1 - e^{-c\ln(n)} = 1 - \frac{1}{n^c}.$$

So we conclude that the algorithm is successful with high probability, when allowing a total runtime of $B \cdot \mathcal{O}(n^2) = \mathcal{O}(n^4 \log(n))$. For such an easy algorithm this is not too bad, but still above the known deterministic runtime. Note that in dense graphs there can be $\mathcal{O}(n^2)$ edges, so $\mathcal{O}(n^2)$ is a natural lower bound for all MinCut algorithms. We will see in the next paragraph how we can improve Karger's algorithm to get a runtime of $\tilde{\mathcal{O}}(n^2)$.

3.2.3 FastCut

Where could Karger's algorithm be improved? The basic observation here is, that the first few contractions are pretty 'safe', because the chance to select a MinCut edge is small. The later we are in the process the bigger the chance to screw up. So at some point contraction becomes 'dangerous'. Re-running the guessing algorithm several times does not incorporate that fact. But obviously it would be beneficial to spend more time on performing 'dangerous' contractions in order to get a better success probability. This is achieved using the following algorithm, which is called the FastCut algorithm.

Algorithm 7: FASTCUT(G)

```
begin
    if $(n+1)/\sqrt{2} > 2$ then
        $X_1 \leftarrow FASTCUT(CONTRACT(G, (n+1)/\sqrt{2})$;
        $X_2 \leftarrow FASTCUT(CONTRACT(G, (n+1)/\sqrt{2})$;
        return $\min\{X_1, X_2\}$;
    else
        return only cut;
end
```

Algorithm 8: CONTRACT(G,m)

```
begin
    for $i = n$ downto $m$ do
        pick a random edge $e \in E$;
        $G \leftarrow G \setminus e$;
    Return $G$;
end
```

So FastCut uses two trials to not contract any MinCut edges until the graph is down to $(n+1)/\sqrt{2}$ many nodes. Then it spends four trials to get down to $(n+1)/2$, then eight trials and so on. The calls form then a binary tree of depth $\mathcal{O}(\log(n+1))$. As soon as there exists a single path from the root to a leave in this tree that corresponds to not contracting a MinCut edge all the way through, the algorithms returns the correct MinCut. The probability to screw up in the 'safe' phase where we start with n nodes and reduce them to $(n+1)/\sqrt{2}$ nodes can be calculated like in our previous analysis as:

$$1 - \prod_{i=(n+1)/\sqrt{2}+1}^{n} \frac{i-2}{i} = \frac{(n+1)/\sqrt{2} \cdot ((n+1)/\sqrt{2}-1)}{n(n-1)} < \frac{1}{2}$$

Therefore the total success probability can be computed as:

$$P(n) \geq 1 - \left(1 - \frac{1}{2}P\left(\frac{n+1}{\sqrt{2}}\right)\right)^2 = P\left(\frac{n+1}{\sqrt{2}}\right) - \frac{1}{4}P\left(\frac{n+1}{\sqrt{2}}\right)^2$$

Lemma 3.1. *The success probability $P(n)$ is at least $1/\log_2(n+1)$.*

Proof. We prove the lemma by induction. The base case is $P(2) = 1 = 1/log_2(2)$. Now we plug the induction hypothesis in the recurrence:

$$P(n) \geq P\left(\frac{n+1}{\sqrt{2}}\right) - \frac{1}{4}P\left(\frac{n+1}{\sqrt{2}}\right)^2 = \frac{1}{\log_2((n+1)/\sqrt{2})} - \frac{1}{4(\log_2((n+1)/\sqrt{2}))^2}$$

$$= \frac{1}{\log_2(n+1) - 1/2} - \frac{1}{4(\log_2(n+1) - 1/2)^2}$$

$$\Rightarrow P(n) \geq \frac{4\log_2(n+1)-3}{4\log^2(n+1)-4\log_2(n+1)+1} = \frac{1}{\log_2(n+1)} + \frac{1/\log_2(n+1)}{4\log_2(n+1)^2-4\log_2(n+1)+1}$$

For $n > 2$ the latter term is positive, hence we end up with

$$P(n) \geq \frac{1}{\log_2(n+1)}.$$

□

So the success probability of FastCut is much higher than for simple guessing. What about the runtime $T(n)$? It can be expressed as

$$T(n) = \mathcal{O}(n^2) + 2T\left(\frac{n+1}{\sqrt{2}}\right) = \mathcal{O}(n^2 \log n)$$

because every call to contract costs us at most quadratic time, and the remaining time is spend on the two recursive calls.

To get the overall runtime, we have to calculate the number of calls we have to make to FastCut to get the correct MinCut with high probability. Using the same estimation as before we see that $B = c\log_2(n)\ln(n)$ suffices. So we have to run FastCut expectedly $\mathcal{O}(\log(n)^2)$ times to get the MinCut with high probability, yielding a total runtime of $\mathcal{O}(n^2\log^3(n))$. This is better than the best deterministic algorithm, and at the same time the FastCut algorithm is much easier to implement.

3.3 Sublinear Algorithms

We will now see that randomized algorithms even have the power to get good approximative results with positive probability in quasi constant time.

3.3.1 Counting the Number of Connected Components

Two nodes are in one connected component (CC) in an undirected graph $G(V, E)$ iff there exists a path between them in G. A simple deterministic algorithm to count the number of CCs is to run BFS from an arbitrary start vertex to get all nodes in the same CC as the vertex. As long as vertices remain, a new vertex is chosen that was in none of the previously computed CCs, and used as start vertex for another BFS run. BFS runs in $\mathcal{O}(|V| + |E|)$, i.e. it requires time linear in the size of the graph. But if the graph is dense then $|E| \in \mathcal{O}(|V|^2)$, and for larger graphs running BFS still might be too expensive. Also BFS requires linear space. If the graph is huge this might also be prohibitive. We therefore now have a look at some randomized algorithm, there runtime and space consumption depend only on an input parameter ϵ but not on $n = |V|$ at all!

Theorem 3.2. *There exists a randomized algorithm that for given $G(V, E)$ and ϵ outputs an ϵ-additive approximation to the number of CCs in G with a probability of $2/3$. So if the algorithm returns k' and the correct solution is k, with a probability of $2/3$ it yields $|k' - k| \leq \epsilon n$.*

The randomized algorithm will be based on taking a random sample of vertices and perform certain operations for them. To analyse the algorithm, we define for a vertex $v \in V$ the number of vertices in the CC of v as n_v. So $n_v = |C(v)|$ with $C(v)$ being the CC in which v is contained. We make two simple observations about n_v:

- $\sum_{v \in C_i} \frac{1}{n_v} = 1$ where C_i is the CC number i. This is true since $\forall v \in C_i : \frac{1}{n_v} = \frac{1}{|Ci|}$ and $|C_i| \cdot \frac{1}{|C_i|} = 1$.
- $\sum_{v \in V} \frac{1}{n_v} = k$, because every CC contributes 1 to the sum.

So n_v gives us some local information about v. If v is part of a very large CC, then $1/n_v$ is small and vice versa. Intuitively, if there are many CCs in the graph, there need also to be many small ones. We will use these facts to restrict the amount of work per vertex in a random sample, leading finally to the sublinear runtime without losing the property of having a good estimate. The following formal definition is the first step in that direction.

●**Definition (Definition)** *Let $\hat{n}_v = \min\{n_v, 2/\epsilon\}$. This means, $\hat{n}_v = n_v$ if v is in a CC of size $< 2/\epsilon$ and $2/\epsilon$ otherwise. We say v is in a small component in the first case, and in a large component in the second case.*

Lemma 3.3. $\forall v \in V : 0 \leq \frac{1}{\hat{n}_v} - \frac{1}{n_v} < \frac{\epsilon}{2}$

Proof. If v is in a small component $\frac{1}{\hat{n}_v} - \frac{1}{n_v} = 0$, otherwise $0 < \frac{1}{\hat{n}_v} - \frac{1}{n_v} = \frac{\epsilon}{2} - \frac{1}{n_v} < \frac{\epsilon}{2}$. □

We now let $\hat{k} = \sum_{v \in V} \frac{1}{\hat{n}_v}$ be a first estimate for k.

Lemma 3.4. $|\hat{k} - k| < \frac{\epsilon n}{2}$

Proof. Using the previous lemma, we know that $\frac{1}{\hat{n}_v} - \frac{1}{n_v} < \frac{\epsilon}{2}$, therefore also

$$\sum_{v \in V} \left(\frac{1}{\hat{n}_v} - \frac{1}{n_v}\right) < \sum_{v \in V} \frac{\epsilon}{2}$$

must hold. This is the same as stating:

$$\sum_{v \in V} \frac{1}{\hat{n}_v} - \sum_{v \in V} \frac{1}{n_v} < n\frac{\epsilon}{2}$$

With $\hat{k} = \sum_{v \in V} \frac{1}{\hat{n}_v}$ and $k = \sum_{v \in V} \frac{1}{n_v}$ we get the desired inequality. □

We are now ready to formulate the randomized algorithm. The basic idea is to draw a random sample of vertices of sufficient size, than compute $\hat{n}_v$ for all these vertices locally, then compute the average over all calculated values $1/\hat{n}_v$ and multiply this average with n to get a good estimate on the number of CCs in the graph. We claim that this algorithm outputs with positive probability

Algorithm 9: SAMPLECC(G, ϵ)

```
1 begin
2     pick $s \in \mathcal{O}(1/\epsilon^2)$ vertices in a random sample $S = \{v_1, \cdots, v_s\}$;
3     for $v_i \in S$ do
4         run BFS from $v_i$ until the search visits at most $2/\epsilon$ vertices;
5         set $\hat{n}_{v_i}$ to the number of visited nodes in the BFS run;
6     return $k' \leftarrow \frac{n}{s} \sum_{v \in S} \frac{1}{\hat{n}_v}$;
7 end
```

something close to the correct value k in practically constant time.

Theorem 3.5. *With probability $2/3$ the algorithm returns a k' which satisfies $|k' - k| < \epsilon n$, while requiring only $\mathcal{O}(1/\epsilon^4)$ time.*

The runtime is dominated by the local BFS runs that have to be performed for each sample vertex. If the neighbourhoods of the sample vertices are dense, there might be $\mathcal{O}(2^2/\epsilon^2)$ edges to consider in each run. So the total runtime can be expressed as $\mathcal{O}(1/\epsilon^2) \cdot \mathcal{O}(1/\epsilon^2) = \mathcal{O}(1/\epsilon^4)$.

To prove the ϵn-additive approximation, we need a helping lemma. The idea is that we first show that k' approximates $\hat{k}$ well, and then combine this with our previous observation that $\hat{k}$ and k are close.

Lemma 3.6. *$P(|k' - \hat{k}| > \epsilon n/2) < 1/3$*

Proof. Let X_i be the random variable that holds $\frac{1}{\hat{n}_{v_i}}$ for $v_i \in S$. As every vertex in V is equally likely to be selected as v_i, the expected value of X can be expressed as

$$E(X_i) = \frac{1}{n} \sum_{v \in V} \frac{1}{\hat{n}_v} = \frac{\hat{k}}{n}.$$

Let $X = \sum_{i=1}^{s} X_i$ be the random variable that accumulates these values. Then we get:

$$E(X) = E(\sum_{i=1}^{s} X_i) = \sum_{i=1}^{s} E(X_i) = \sum_{i=1}^{s} \frac{\hat{k}}{n} = s\frac{\hat{k}}{n}$$

Observe that the algorithm actually outputs $\frac{n}{s}X$. The question now is how far $k' = \frac{n}{s}X$ is from its expected value, knowing that:

$$E(k') = E(\frac{n}{s}X) = \frac{n}{s}E(X) = \frac{n}{s}s\frac{\hat{k}}{n} = \hat{k}$$

The tool we learned for such an estimation is the Chernoff bound. It essentially tells us in this context that

$$P(|X - E(X)| \geq \frac{\epsilon}{2}s) < e^{-\epsilon^2 \cdot s/8}.$$

Using $s = 12/\epsilon^2$ we get

$$P(|X - E(X)| \geq \frac{\epsilon}{2}s) < e^{-1.5} < \frac{1}{3}.$$

And accordingly

$$P(|\frac{n}{s}X - \frac{n}{s}E(X)| < \frac{\epsilon}{2}s\frac{n}{s} = \frac{\epsilon n}{2}) < \frac{1}{3}$$

which concludes the proof. □

So now we know that that algorithm outputs an $\epsilon n/2$-additive approximation to $\hat{k}$, but we want an ϵn-additive approximation for k. So what is the probability of k' and k to be more than ϵn apart? Well, if $|k' - k| > \epsilon n$, we also know that $|k' - \hat{k}| + |\hat{k} - k| \geq |k' - k| > \epsilon n$ (triangle inequality). For $|\hat{k} - k|$ we have proven before that the term is smaller than $\epsilon n/2$. Therefore $|k' - k| > \epsilon n$ coincides with $|k' - \hat{k}| > \epsilon n/2$. For the latter we know that the probability of this to happen is smaller than $1/3$. Therefore altogether $P(|k' - k| > \epsilon n) < 1/3$, which means with a probability of $2/3$ we get an ϵn-additive approximation.

What if we want to have a smaller probability of error, that is we want to decrease our confidence to $1 - \delta$, with δ arbitrary small?

Lemma 3.7. *If we pick $s = \mathcal{O}(1/\epsilon^2 \log 1/\delta)$ samples in the above algorithm, we get $P(|k' - \hat{k}| > \epsilon n/2) < \delta$.*

The runtime only increases then to $\mathcal{O}(1/\epsilon^4 \log 1/\delta)$ which for given constant values of ϵ and δ still is constant.

3.3.2 Estimating the Weight of Minimum Spanning Trees

So we can count the number of CC in constant time, with a small probability of a large additive error. While this is a nice result on its own, it can also used as a subroutine when estimating the weight of a minimum spanning tree in sublinear time (again allowing an additive error and some screw up probability). So now we consider a connected weighted graph $G(V, E)$ with weights $w : E \to \{1, 2, \cdots, W\}$. The goal is to estimate the weight of the minimum spanning tree $w(MST)$. Deterministic algorithms like Prim and Kruskal produce the correct result naively in $\mathcal{O}(m \log m)$.

In order to do this in sublinear time, we try to express the weight of the MSt as function of some local quantities. In fact, we will look at specific subgraphs of G and count the number of CCs in them, which finally leads to an estimate $\hat{w}(MST)$. We start by considering G_1 which only contains a subset of the edges in E, namely the ones with $w = 1$. Let k_1 be the number of CC in G_1, and T an MST in G. The crucial idea now is to estimate the number of edges in T with a weight strictly larger than 1 by using k_1. For that, we need the following two observations:

- If T would have an edge with weight > 1 in a CC of G_1 it would close a cycle, so this is impossible.
- To connect the k_1 CC in G_1 in T we need exactly $k_1 - 1$ edges with a weight > 1, because every additional edge decreases the number of disconnected components by one.

Now lets define G_i generally as the subgraph of G with edges of weight $\leq i$, we see that the argumentation from above transfers. So we always know that in T there are $k_i - 1$ edges of weight $> i$. Observe that this is also true for $i = 0$, as then G_0 just consists of n single vertices and every MST has exactly $n - 1$ edges.

Lemma 3.8. $w(MST) = \sum_{i=0}^{W} (k_i - 1)$

Proof. Let m_i be the number of edges with weight i in T. Then $w(MST) = \sum_{i=1}^{W} i \cdot m_i$. We can rewrite this term as $\sum_{i=1}^{W} m_i + \sum_{i=2}^{W} m_i + \cdots \sum_{i=W}^{W} m_i$. This assures that each m_i is counted exactly i times. We just have seen before that $\sum_{i=j}^{W} m_i$ is nothing alse as $k_j - 1$. Hence we get $w(MST) = \sum_{i=0}^{W} (k_i - 1)$. □

Now we can finally formulate the sampling algorithm. Lets analyze this algorithm. If all estimates

Algorithm 10: SAMPLEMST(G, ϵ)

```
1 begin
2     for $i = 1$ to $W$ do
3         $\hat{k}_i \leftarrow$ SAMPLECC$(G_i, \epsilon' = \epsilon/W)$ with $\delta = 1/3W$;
4     return $\hat{w}(MST) = \sum_{i=1}^{W} \hat{k}_i$;
5 end
```

of $\hat{k}_i$ are within an additive error of $\epsilon n/W$, so $|\hat{k}_i - k_i| \leq \epsilon n/W$, then

$$|\sum_{i=1}^{W} (\hat{k}_i - k_i)| \leq W \cdot \frac{\epsilon n}{W} = \epsilon n.$$

So using $s = \mathcal{O}(1/\epsilon^2 \log W/\delta)$ sample vertices, we get for each i that

$$P(|\hat{k}_i - k_i| \geq \frac{\epsilon n}{W}) < \frac{1}{3W}$$

and the probability that one estimate is bad is bounded by $W \frac{1}{3W}$, i.e. a third. The runtime of the SAMPLEMST algorithm is $\mathcal{O}(W 1/\epsilon^4 \log W/\delta)$, which again is sublinear when assuming ϵ, δ and W to be constant.

3.4 Long Path Problems

Given a graph $G(V, E)$ and an integer k, the goal is to find a simple path of length k in G. For $k = n - 1$ this problem equals the task to find a Hamilton path in a graph (so a simple path over all vertices), which is an NP-complete problem. We will tackle the problem for not too large k using randomized techniques.

3.4.1 The DAG algorithm

The central idea of the first algorithm is to exploit the fact that the search for the longest path in a directed acyclic graph is not hard, but can be accomplished in linear time. For this purpose, the DAG gets sorted topologically, i.e. each node receives a label $l(v)$ such that for all edges $(v, W) \in E$ yields $l(v) < l(w)$. This is similar to demanding that the graph could be drawn by placing all vertices on a horizontal line and edges only go from left to right (if the graph has a cycle this obviously would not be possible). In such a sorted DAG, the longest path can be computed via a simple sweeping algorithm. For each node a maximal path length label d is maintained which is initially zero. Then parsing through the nodes in l-order, we relax all outgoing edges and increase the path length label at the tail nodes whenever possible. After considering all nodes, the maximal node labels tells the length of the longest path (if predecessors are stored along, this path can also be backtracked in linear time). Obviously, the algorithm considers each edge only once and performs constant time operations on it, so the total runtime is in $\mathcal{O}(m)$.

So how can we use the availability of an efficient longest path algorithm for DAGs in our setting? Well, any undirected graph can be turned into a DAG by fixing some order l of the nodes and then direct all edges such that $l(v) < l(w)$ is true for all of them. Then in this DAG we can search for the longest path and check if its length exceeds k. But of course the node order influences the length of the longest path in the DAG. If we choose a permutation u.a.r. the following lemma applies.

Lemma 3.9. *If there is a path p of length k in G then p will be a path in a randomly constructed DAG of G with probability $\alpha = \dfrac{2}{(k+1)!}$.*

Proof. The path p remains a path in the DAG only if all its nodes are either labelled increasingly or decreasingly. So of the $(k+1)!$ possible node permutations only 2 lead to success, which means the probability that p is a path in the DAG is $\dfrac{2}{(k+1)!}$. □

The lemma also tells us that if G contains a path of length k then the DAG has a path of length k with probability at least α. As the success probability is pretty small, we use again amplification to improve our chances.

Algorithm 11: LONGPATH1(G)

```
1 begin
2     for $i = 1$ to $t = 1/\alpha$ do
3         construct a DAG for $G$ using a random node permutation;
4         find the longest path $p$ in the DAG;
5         if $|p| \geq k$ then
6             return $p$;
7     return fail;
8 end
```

Theorem 3.10. *LONGPATH1 finds a path of length $k = \mathcal{O}\left(\dfrac{\log n}{\log\log n}\right)$ if one exists in randomized polynomial time.*

Proof. The probability to fail in all t trials to find a sufficiently long path can be bounded as:

$$P(fail) \leq (1-\alpha)^t \leq e^{-t\alpha} \leq \frac{1}{e}$$

Hence the algorithm finds a path of length k with probability at least $1 - 1/e \geq 0.63$. The runtime of the algorithm is $t\mathcal{O}(m)$ as the DAG construction and path search are in $\mathcal{O}(m)$ and the process is repeated t times. So the runtime is in $\mathcal{O}(m^{(k+1)!/2}) = \mathcal{O}(m(k+1)^k)$. Setting k to $\dfrac{c \log n}{\log \log n}$ we get a total runtime of $\mathcal{O}(mn^c)$ which is polynomial for some constant c. □

3.4.2 The Colouring Algorithm

A completely different reduction of the problem is to colour all nodes and then look for a colourful path. In fact, lets have a set of $k+1$ colours, and every node in the graph gets coloured with one of these colours chosen u.a.r. from the set. A colourful path is a path in which no colour occurs twice. Obviously, the length of the longest colourful path is k as there are only $k+1$ distinct colours. A colourful path can be found in time linear in m but exponential in k.

❓Exercise. Describe an algorithm which computes a colourful path of length k in time $\mathcal{O}(mk2^k)$.

We will now investigate in the success probability of constructing a long colourful path using a random colour assignment.

Lemma 3.11. *If p is a path of length k in G, then the probability of p being colourful is*

$$P(success) \approx \frac{\sqrt{2\pi(k+1)}}{e^{k+1}}.$$

Proof. The path p is colourful if all its nodes have different colours. The number of way in which the vertices could be coloured is $(k+1)^{(k+1)}$, as for any f the $k+1$ nodes $k+1$ possible colours are available. The number of ways in which p is colourful is $(k+1)!$, so:

$$P(success) = \frac{(k+1)!}{(k+1)^{k+1}}$$

Stirling's approximation says that $a! \approx \left(\frac{a}{e}\right)^a \sqrt{2\pi a}$. Plugging in $k+1 = a$ we get:

$$P(success) \approx \frac{\sqrt{2\pi(k+1)}\left(\frac{k+1}{e}\right)^{k+1}}{(k+1)^{k+1}} = \frac{\sqrt{2\pi(k+1)}}{e^{k+1}}$$

□

Lets denote this success probability as β to state the second algorithm for long path extraction.

Theorem 3.12. *LONGPATH2 finds a path of length $k = \mathcal{O}(\log n)$ if one exists in randomized polytime.*

Proof. The failure probability can be denoted as:

$$P(fail) \leq (1-\beta)^t \leq \frac{1}{e}$$

The runtime of the algorithm is t times $\mathcal{O}(mk2^k)$, so $\mathcal{O}(mk2^k e^k)$. For $k = c\log n$ this results in a runtime of $\mathcal{O}(mn^{\mathcal{O}(c)})$, which is polynomial for constant c. □

So we observe that the colouring algorithm is slightly better than the DAG based algorithm, as it allows to handle larger values of k in randomized polynomial time. Recent results (based on other techniques) show that with randomization one can even allow $k = \mathcal{O}(e^{\sqrt{\log n / \log\log n}})$ and still achieve polynomial runtime.

Algorithm 12: LONGPATH2(G)

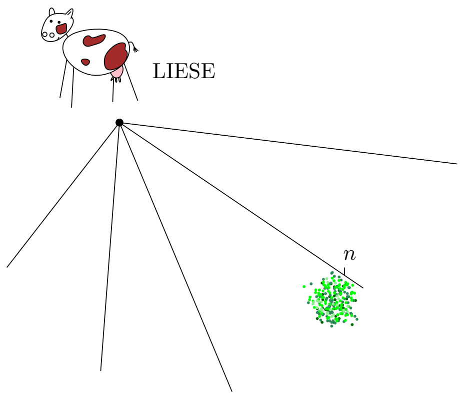
```
1 begin
2     for $i = 1$ to $t = 1/\beta$ do
3         colour the nodes in $G$ randomly with $(k+1)$ colours;
4         find the longest colourful path $p$;
5         if $|p| \geq k$ then
6             return $p$;
7     return fail;
8 end
```

4 Randomization in Games and AI

We will now consider some problems related to explore unknown terrains or detect unknown variables in an efficient manner. For more information on the Cow-Path problem see https://www.cs.duke.edu/~reif/paper/tate/cowpath.pdf.

4.1 The Cow-Path Problem

Many applications in AI involve the exploration of an unknown environment while minimizing the effort. The cow-path problem is one incarnation of this setting. Imagine a cow, Liese, which stands at a crossroad (the origin) with w lanes leaving in different directions. One lane only leads to a grazing field at distance n from Lieses actual location, see the following illustration:



As Liese is already a bit old and cannot see very well, she only notices the grazing field as soon as she stands on it. If she would choose one direction and walks on that lane until she finds grass, she might be walking forever if the initial direction is wrong. The basic question is how Liese should explore the lanes in a way that leads her to the field without her having to walk a very long distance. If she would know the correct lane, she only needs to walk distance n obviously. But what if she does not? We will restrict us as first to two lanes only and investigate good strategies for this scenario.

So now we assume Liese only has to choose between walking to the left or to the right at any time. What if she would be aware about the distance n? Then she could choose a lane and walk n in this direction. If the field is not there, she turns around an walks $2n$ in the other direction. So in a maximum walk distance of $3n$ she will be happily grazing. If she chooses the initial direction u.a.r (so left or right each with probability of $1/2$), the expected walk distance for this strategy is $1/2 3n + 1/2 n = 2n$. That means in expectation she only walks twice as much as if somebody told her the correct direction. This factor 2 is then also called the (expected) competitive factor of the strategy.

●**Definition (Competitiveness)** *The competitive factor c of an algorithm A for exploring an unknown environment is the maximal ratio of the solution returned by A and the optimal solution for the setting where the complete environment is known a priori. If A is a randomized algorithm, we call c the expected competitive factor.*

So lets take more information away from Liese. Now she is not aware of the distance to the field anymore, but just knows that in one direction there has to be grass at some point. How should she act? Obviously, any strategy that does not involve exploring lanes in an alternating manner might force poor Liese to walk forever on a wrong lane. So to guarantee anything, we need Liese to do the following: Walk $f(i)$ steps to the left in round i if i is even. If there is no grass on the way, turn around, walk back to the origin, and walk $f(i+1)$ steps to the right in round $i+1$, and so on until the desired grass is found. Clearly, any sensible function f needs to fulfil $f(i+2) > f(i)$ in order to make Liese explore new path sections in every round. An easy way to get a function that sticks to that property is to set $f(i) = i$. How long will Liese be on her way then?

Lemma 4.1. *For $f(i) = i$, Liese has to walk a distance of $\Theta(n^2)$ to reach the grazing field.*

Proof. In round i Liese walks $2i$ steps, namely i in the defined direction and i back to the origin. At latest in round $n+1$ she reaches the field after n steps (if in round n she walked in the wrong direction). So the walking distance can be bounded by:

$$\sum_{i=1}^{n} 2i + n = 2\sum_{i=1}^{n} i + n = 2\frac{n(n+1)}{2} + n = n^2 + 2n$$

If she is lucky and starts with the correct direction, she gets there $2n$ steps earlier, i.e. in distance of n^2. Therefore she has to walk a distance of $\Theta(n^2)$ to reach the goal in both cases. □

So with $f(i) = i$ there is a huge walking overhead compared to the path where Liese knows the grazing field location. In fact, the competitive factor here is n. We will show now, that choosing a better function f results in a constant competitive factor only. The idea is to apply iterative doubling until Liese finds the grass.

Lemma 4.2. *For $f(i) = 2^i$ the alternating strategy is 9-competitive.*

Proof. In each intermediate round Liese now walks $2f(i) = 2^{i+1}$ steps. Let j be the integer such that $2^j < n \leq 2^{j+1}$ holds. In the worst case, Liese walks in the wrong direction in round $j+1$, so the total walking distance would be:

$$\sum_{i=1}^{j+1} 2^{i+1} + n = 2(2^{j+2} - 2) + n = 2^{j+3} - 4 + n = 2^3 \cdot 2^j - 4 + n$$

As we know that $2^j < n$ we can bound the sum by $8n - 4 + n < 9n$, proving the lemma. □

❓**Exercise.** Show that no deterministic strategy can have a competitive ratio better than 9.

What about applying randomization in that scenario? So again Liese chooses to walk to the left or to the right in the first round each with probability $1/2$.

Lemma 4.3. *For $f(i) = 2^i$ and a randomly chosen initial direction, the alternating strategy is expectedly 7-competitive.*

Proof. In the previous lemma we analyzed the worst case, there Liese walks in the wrong direction in the most costly round. What if she is lucky? Then the walk distance can be expressed as:

$$\sum_{i=1}^{j} 2^{i+1} + n = 2(2^{j+1} - 2) + n = 2^{j+2} - 4 + n = 2^2 \cdot 2^j - 4 + n < 4n - 4 + n < 5n$$

The expected walk distance is then $1/2(9n + 5n) = 7n$. □

This is quite good. Liese expectedly only has to walk 7 times the distance if she knows nothing compared to the case where she knows the correct lane. Nevertheless, using randomization this bound can even be improved using the so called SMARTCOW algorithm, where not only the initial direction but also the distance in every round is cleverly randomized. Algorithm 13 holds the pseudocode for an arbitrary number w of lanes and a predefined constant $r > 1$ which corresponds to the minimal walk radius.

Algorithm 13: SMARTCOW

```
1  begin
2      $\sigma \leftarrow$ random permutation of $\{0, \cdots, w-1\}$;
3      $\epsilon \leftarrow$ real value chosen u.a.r. from $[0, 1)$;
4      $d \leftarrow r^{\epsilon}$;
5      $p \leftarrow 0$;
6      while goal not found do
7          explore lane $\sigma(p)$ up to distance $d$;
8          if goal not found then
9              return to origin;
10             $d \leftarrow d \cdot r$;
11             $p \leftarrow (p+1) \mod w$;
12 end
```

It can be shown that this algorithm has a competitive ratio of

$$c(r, w) = 1 + \frac{2}{w} \frac{1 + r + r^2 + \cdots + r^{w-1}}{\ln r}$$

and that this is the optimal ratio as there exists a matching lower bound. For $w = 2$ and $r = 3.59112$, the algorithm achieves a competitive factor of 4.59112. This is almost twice as good as the deterministic strategy with a competitiveness ratio of 9.

Exercise. Implement the cow path setting with w and n as parameters. Compare a good deterministic strategy to the randomized version of it and the SMARTCOW algorithm. For SMARTCOW empirically identify a good value for r.

4.2 Robots Running Against Walls

Now we consider a robot that has to navigate in an unknown geometric terrain. The robot stands at $s \in \mathbb{R}^2$ and has to get to target t within the plane with the coordinates of t known to him. What the robot is not informed about are obstacles on the way to the target. These obstacles must be circumvented in order to reach the goal. Again, we would like to minimize the distance the robot needs to cover to get to the target. To compute an easy upper bound for the competitive ratio of our strategy we compare to the path distance between s and t in a setting without any obstacles (so just straight-line distance). The 'real' competitive ratio would result from a comparison to the optimal path from s to t with all obstacles known a priori. But obviously, when we perform well compared to the straight line distance, we all the more perform well compared to the real baseline.

To make the problem setting clearer, we state some basic assumptions.

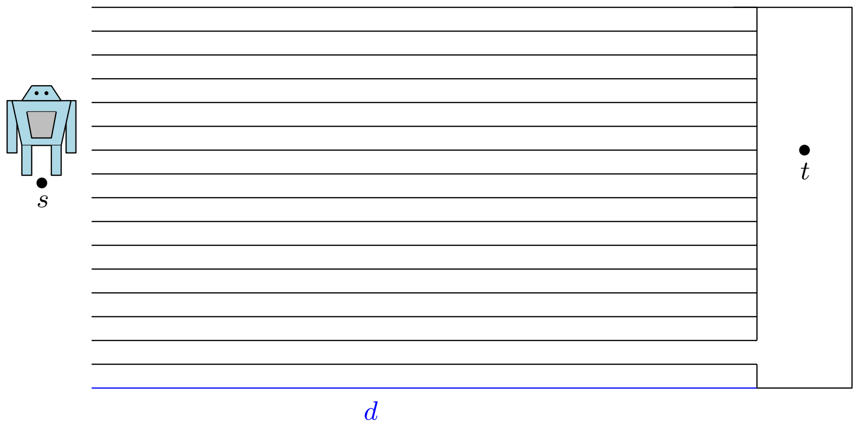
- Obstacles are opaque, impenetrable (no machine gun will help the robot) and non overlapping.
- None of the obstacles contain or enclose s or t.

- The target t might be a single point, a polygon or an infinite wall.
- The unit square can be inscribed in every obstacle, so obstacles cannot be infinitely thin.
- If two obstacle touch, the robot can squeeze between them. This is important if we only allow convex obstacles, because then no non-convex obstacles can be created by sticking together convex ones.
- The robot has no vision. It only can explore objects by bumping into them.

In the following we explore some specific setting for the robot problem, investigating competitive strategies and lower bounds.

4.2.1 Non-convex Obstacles

If non-convex obstacles are allowed, we could construct a maze-like structure with them. For example, consider the following setting:



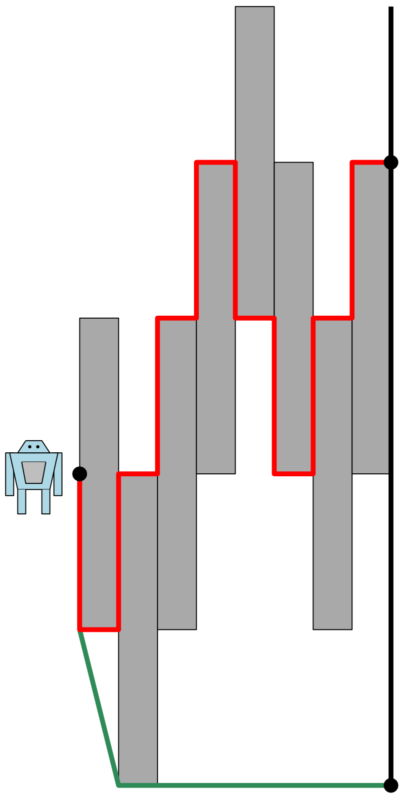
The robot has to choose between different corridors of length d each, with only one leading to the target. Every time he chooses a wrong one, it costs distance $2d$ to find the blockage and leave the corridor again. So if there are k such corridors and the optimal distance is within $2d$, the competitive ratio cannot be guaranteed to be better than k. Hence in the setting with non-convex objects no good setting-independent competitive strategies exists.

4.2.2 The Wall Problem

As we have learned in the last paragraph, non-convex obstacles do not allow for good competitive strategies. Therefore we now turn to convex objects. Lets assume the robot is located initially at point $s = (0, 0)$ embedded in a coordinate system and the target is an infinite vertical wall at distance n from s. Furthermore the obstacles are only allowed to be axes-parallel rectangles. Identifying the best path from source to target in this setting is called the *wall problem.*

We first show that any deterministic strategy has a competitive ratio of $\Theta(\sqrt{n})$, then introduce a deterministic strategy which achieves a competitive factor of $\mathcal{O}(\sqrt{n})$ and then proceed with improvements due to randomization.

Deterministic Lower Bound We will show that no deterministic strategy can guarantee a competitive factor better than $\Theta(\sqrt{n})$. For that, we only consider axes-parallel obstacles that have a width in x-direction of 1 and a height in y-direction of n. Next we assume that the plane is at first obstacle free, but some evil person places obstacles while the robot is moving to elongate the path. In fact, every time the robot reaches a new x-value for the first time, the evil person locates a $1 \times n$ obstacle between the robot and the target wall, such that the center of the left side of the obstacle is at the robot's position. Of course, the person can only do that, if the new obstacle does not overlap previous ones. So the path of the robot and the placement of obstacles could look for example like this (red path - actual path of the robot, green path - optimal path):



Theorem 4.4. *The path the robot takes is at least factor $\Theta(\sqrt{n})$ longer than the optimal path.*

Proof. We first compute a lower bound for the distance walked by the robot. The first observation is that the evil enemy places at least n obstacles in the way of the robot. For each obstacle, the robot has to walk a distance of $n/2$ in y-direction to circumvent it. Hence, the robot walks vertically at least a distance of $n \cdot n/2 = \Theta(n^2)$ before reaching the target (plus at least n in horizontal direction, but this does not matter for the sake of the proof). Now we try to get an upper bound on the length of the optimal path. Obviously as soon as we can prove the existence of a path of length $\mathcal{O}(n\sqrt{n})$ the theorem follows. For a fixed y-coordinate we can go from s to the target wall by first moving vertically from s until we reach y, and then try to walk straight towards the target. Every time the robot hits an obstacle it circumvents it (by arbitrarily choosing the direction of circumvention) and then returns to the desired y-coordinate. Figure 1 shows three different such y-paths. Let d be the vertical distance from s to the desired y-coordinate and k the number of obstacles that intersect the horizontal line defined by y. Then the length of the path is bounded by $d + 2kn + n$ as the robot first needs d to reach y, each circumvention costs at most $2n$ in vertical direction and the total distance in horizontal direction is exactly n. We will now show that we can bound d by $n\sqrt{n}$ and k by $\sqrt{n}$ at the same time. For that purpose consider the y-coordinates in the range $[-n\sqrt{n}, n\sqrt{n}]$ that are multiples of $n/2$. There are $4\sqrt{n} + 1$ such coordinates. Each obstacle can intersect at most 2 such y-coordinates. Therefore, there has to be at least one y-coordinate in this range that intersects no more than $\sqrt{n}$ of the n obstacles. Hence for some y-coordinate at most $d \leq n\sqrt{n}$ away from s it yields the number k of obstacles that intersect the horizontal line induced by y is bounded by $\sqrt{n}$. Plugging in these values in the path length formula $d + 2kn + n$ we get an overall path length of $\mathcal{O}(n\sqrt{n})$, concluding the proof. □

A $\mathcal{O}(\sqrt{n})$-Competitive Deterministic Strategy We now present a strategy whose competitive ratio is $\mathcal{O}(\sqrt{n})$. As this matches the lower bound, the strategy is asymptotically optimal. For

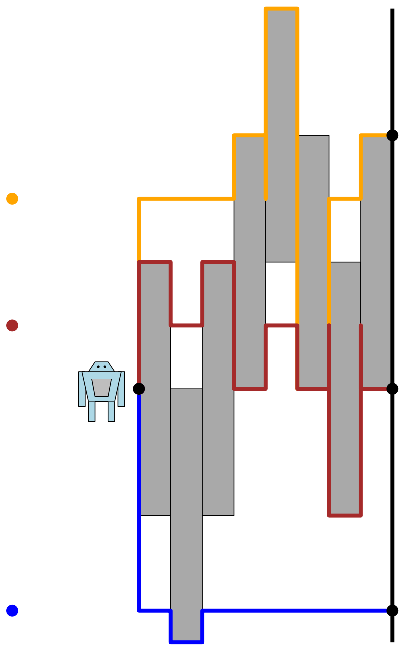


Figure 1: Induced y-paths by fixed y-values (indicated by the yellow, brown and blue dot).

the analysis we require the following observation.

Observation (Manhattan-Paths) *An optimal path with the following limitations (called a Manhattan-path)*

- *movements are only strictly horizontal or strictly vertical*
- *the x-coordinate never decreases*

is only up to a factor of $\sqrt{2}$ longer than an optimal path without any limitations.

The strategy is based on a sweep algorithm in a predefined corridor induced by two parameters y_1 and $y_2 > y_1$.

Algorithm 14: UPWARDSSWEEP(x, y_1, y_2, τ)

```
1  begin
2      $p \leftarrow (x, y_1)$;
3      while actual $y$-coordinate is $< y_2$ do
4          move right until bumping into an obstacle $O$;
5          if one corner of $O$ is less than $\tau$ away then
6              circumvent $O$ via this corner;
7              return to previous $y$-coordinate;
8          else
9              move to upper left corner of $O$;
10 end
```

A DOWNWARDSSWEEP works quite similar, just that we start with (x, y_2), want to reach y_1 and go in line 9 to the lower left corner of the obstacle. Note that to decide if one of the corners is less than τ away (line 5), the robot has to move first τ downwards to check (like in the cow path problem with known n) and then upwards after that (if the corner is too far away).

With a *phase* we refer to an upwards sweep followed by a downwards sweep in the same corridor. We will see that choosing a suitable corridor we will find a good path from the source to the target wall by a concatenation of a bounded number of phases.

Lemma 4.5. *Let y_1, y_2 define a corridor. Let Q be a Manhattan-path that starts and ends at the same points as the path that results from completing a phase in that corridor (upwards + downwards sweep with fixed start point on the line induced by y_1). Then the vertical walk distance of Q is at least τ.*

Proof. Consider the obstacles that actually lead to upwards or downwards moves (i.e. the ones not causing τ-circumventions). Obviously any y-coordinate in the corridor has to intersect at least one of those obstacles (otherwise the robot could walk horizontal forever and the phase would never be completed). If a single obstacle leads to the end of the upwards sweep, then Moreover there needs to be an obstacle with the bottom left corner being more than τ below the upper right corner of the previous obstacle (if not a single obstacle exceeds the corridor to both sides). Q must circumvent at least one of those obstacles, therefore the vertical part of Q has to be at least τ long. □

We will use this lemma to bound the number of phases we need to reach the target wall. Let now W be the sum of the lengths of all vertical parts of an optimal Manhattan-path and lets assume W is known from somewhere. For any $W' > W$, we know that the corridor $[-W', W']$ has to contain the complete optimal path (as otherwise the vertical part would be larger than W). Now we run our phase algorithm in this corridor using

$$\tau = \frac{W'}{\sqrt{n}}.$$

What is the length of the vertical parts of a path constructed by those alternating sweeps? We first bound the distance resulting from τ-circumventions. Because each circumvention brings the robot 1 unit closer to the target wall, there are at most n circumventing moves. Each circumvention costs at most 3τ to determine the corner of the obstacle that is no more than τ away (equal to the cow path problem analysis). Then to complete the circumvention another vertical path of τ is required on the other side of the obstacle. So in total at most $n4\tau$ distance is spend with those moves, which is equal to

$$4n\frac{W'}{\sqrt{n}} \in \mathcal{O}(W'\sqrt{n}).$$

It remains to measure the distance induced by upwards or downwards moves. We know that in every sweep the vertical distance covered by those moves is exactly $2W'$. So how many sweeps do we have to perform? We know that the optimal path requires distance W vertically. The previous lemma told us, that any optimal path spends at least a vertical distance of τ per phase we perform in our algorithm. As the optimal path is contained in our search corridor, we know that after no more than $\lceil \frac{W}{\tau} \rceil$ phases, i.e. $2\lceil \frac{W}{\tau} \rceil$ sweeps, we reach the wall. Hence we can upper bound the total length of upwards/downwards moves by $4W'\lceil \frac{W}{\tau} \rceil$. If we can make sure that $W' \leq 2W$ this formula reduces to $8W(\sqrt{n}+1) \in \mathcal{O}(W\sqrt{n})$. So even together with the vertical distance induced by τ-circumventions we can guarantee to be at most a factor of $\mathcal{O}(\sqrt{n})$ longer than the optimal path (the real optimal path, as this path is at most a factor of $\sqrt{2}$ shorter than the Manhattan-path we considered so far).

So we are done – besides the fact that W is normally not known and therefore cannot easily be plugged in. We came across this problem before, e.g. when we constructed ϵ-nets with the technique of Brönimann and Goodrich. In fact, we can use the same remedy here: iterative doubling. So we first determine a valid lower bound W_0 for the value to estimate, here W. Then, because we know that after a bounded number of rounds we reach the goal if the estimate was good, we can observe during the course of the algorithm is the estimate was good enough. If not we restart with a value $W_1 = 2W_0$ and so on until in a round with $W_i = 2^i W_0$ we have $W_i \geq W$, and hence we reach the goal for sure. The details for the robot navigation application are given in the following.

1. Move right until the first obstacle is hit. Then use a cow path like algorithm to determine the nearest corner of the obstacle. Let this distance be W_0 (this is obviously a valid lower bound for the vertical distance every optimal path has to exhibit).
2. In every round $i \geq 0$, run the phase based algorithm with $W' = W_i = 2^i W_0$. Stop when reaching the wall or after at most $\sqrt{n} + 1$ phases.
3. If the wall was not reached, repeat 2. with $W_{i+1} = 2W_i$.

So what is the total distance? For the sake of analysis let j be the integer for which $2^j W_0 \leq W < 2^{j+1} W_0$. So after iteration $j+1$ we reached the wall for sure. Every iteration costs $\mathcal{O}(\sqrt{n} W_i)$ according to the analysis above. So in total we get

$$\sum_{i=0}^{j+1} \mathcal{O}(\sqrt{n} W_i) = \mathcal{O}(\sqrt{n} W_0 \sum_{i=0}^{j+1} 2^i) = \mathcal{O}(\sqrt{n} 2^j W_0) = \mathcal{O}(\sqrt{n} W).$$

Hence due to iterative doubling we only induce a constant overhead compared to the strategy where W is known a priori. Therefore the strategy with unknown W is $\mathcal{O}(\sqrt{n})$-competitive. As this matches the previously proven lower bound, there is no hope for improvement considering deterministic strategies.

A $\mathcal{O}(\mathbf{n}^{4/9} \log \mathbf{n})$-Competitive Randomized Strategy Again, we will see that the power of randomization can be used to break lower bounds for deterministic strategies.

The randomized strategy for robot navigation in unknown terrains is based on the deterministic iterative doubling strategy. Here, we call the part of the process where the corridor is defined by $[-W_i, W_i]$ with $W_i = 2^i W_0$ as stage i. Every stage will be divided in phases, where parts of the corridor are explored in a randomized way. If this is done in a clever way, the following key properties can be proved:

1. The cost in each stage i is $\mathcal{O}(W_i \cdot n^{4/9} \log n)$.
2. For large enough n and $W_i > W$ (with W denoting the length of vertical parts of an optimal path), the robot reaches the wall by the end of the stage i with a probability of $\geq 3/4$.

If these two properties are fulfilled, the $\mathcal{O}(n^{4/9} \log n)$-competitiveness follows easily as shown in the following theorem.

Theorem 4.6. *An algorithm fulfilling those two properties is $\mathcal{O}(n^{4/9} \log n)$-competitive.*

Proof. Let j be the integer such that $W_{j-1} \leq W < W_j$. Then the expected cost of the algorithm is

$$\mathcal{O}(\sum_{i=1}^{j-1} W_i \cdot n^{4/9} \log n + \sum_{i=j}^{\infty} \left(\frac{1}{4}\right)^{i-j} W_i \cdot n^{4/9} \log n)$$

where the first term corresponds to the minimum amount of effort to reach the target wall and the second term the subsequent iterations until the target wall is indeed reached. We can reformulate this sum as follows:

$$n^{4/9} \log n \cdot \mathcal{O}(\sum_{i=1}^{j-1} 2^i W_0 \cdot + \sum_{i=j}^{\infty} \left(\frac{1}{4}\right)^{i-j} 2^i W_0) = n^{4/9} \log n \cdot \mathcal{O}(\sum_{i=1}^{j-1} 2^i W_0 \cdot + \sum_{i=j}^{\infty} \left(\frac{1}{2}\right)^{i} W_0)$$

Then the first summand is obviously in $\mathcal{O}(2^j W_0) = \mathcal{O}(W)$. The second term converges to $2W_0$ and is therefore in $\mathcal{O}(W)$ as well. So all in all we get $n^{4/9} \log n \cdot \mathcal{O}(W) = \mathcal{O}(W \cdot n^{4/9} \log n)$. Then obviously the ratio of expected costs and optimal solution W is in $\mathcal{O}(n^{4/9} \log n)$. $\square$

For a detailed description of the corresponding algorithm and the respective analysis see the lecture notes or the original paper[1].

4.3 Bidding

Assume you want to buy a certain good in an auction. But you do not know what it is worth. You can submit a bid for the good. If your bid is larger or equal to the value of the good, you will get it. Otherwise you can submit a new bid. The only drawback is, that you have to pay in the end the sum of all your bids to get the good. So if you submit a sequence of bids $b_i, i = 1, \cdots, k$ and the good is worth W, the the competitive ratio can be expressed as $\sum_{i=1}^{k} b_i \cdot W^{-1}$. The question is what a good sequence of bids would be in order to keep the ratio low.

Again, the idea of iterative doubling comes in handy here. So define $b_i = 2^i$. Then we know that $b_k \leq 2W$ has to be true. Therefore the total cost is

$$\sum_{i=1}^{\log(W)+1} 2^i = 2^{\log W+2} - 2 \leq 4W.$$

Therefore the competitive ratio is 4 only, which is quite good. But again a randomized strategy can beat this. For this, we define $b_i = e^X \cdot e^i$ with X randomly distributed in $[0, 1)$. We first have a look at the term $\sum_{i=1}^{k} b_i \cdot b_k^{-1}$. This sum equals in expectation $\sum_{i=0}^{k} e^{i-k} = \sum_{i=0}^{k} e^{-i} \leq \dfrac{e}{e-1}$. Now we consider $b_k \cdot W^{-1}$. This term is distributed as e^X with X uniformly drawn from $[0, 1)$. Therefore the expected value of the ratio is $\int_0^1 e^X dX = e - 1$. The desired ratio $\sum_{i=1}^{k} b_i \cdot W^{-1}$ can then be computed as the product of $\sum_{i=1}^{k} b_i \cdot b_k^{-1}$ and $b_k \cdot W^{-1}$ and therefore equals e in expectation. This is optimal in the sense, that there exists a matching lower bound for randomized strategies.

5 Online-Algorithms

In this chapter, we deal with problems where not the whole setting is known in advance but information are revealed from time to time. More specifically, we assume a stream of requests $\sigma = \sigma_1, \sigma_2, \cdots$ which arrive in an online fashion. Like for the AI-related problems, we will look for strategies with a good competitive factor – now compared to the best strategy for the case that all requests are known a priori. To better distinguish strategies for the latter case from online strategies, we call them offline strategies.

For more details on some of the problems see http://www14.in.tum.de/personen/albers/papers/brics.pdf and for the BahnCard problem in particular http://www.sciencedirect.com/science/article/pii/S0304397500002668.

5.1 Paging

In our first scenario, σ corresponds to page requests. Assume you have a memory system with fast access (a cache) and another one with slow access. The fast access memory can accommodate k pages. The slow access memory is unlimited for the sake of simplicity. Every time a page is requested it has to be loaded into the fast memory. If it is already in there, the request σ_i is cheap. Otherwise, the page first has to be loaded. If the fast memory already contains k pages, additionally one of those has to be selected for eviction (into the slow memory). This means we have an expensive request. We call this a fault. The goal is to develop a strategy to maximize the number of cheap requests while knowing nothing about the nature of future requests at any point in time.

[1] http://dl.acm.org/citation.cfm?id=313852.313890

5.1.1 Deterministic Strategies

What would be good strategies for that problem? Which page in the actual fast access memory should be evicted in case of an expensive request? In the following, three intuitive strategies are listed.

- **Least Frequently Used (LFU)** evict the page from fast memory which has the lowest number of accesses since it was loaded
- **Least Recently Used (LRU)** evict the page that was accessed least recently of all pages in the actual fast memory
- **First-in-first-out (FIFO)** evict the page that has been in fast memory the longest

And in fact, LRU and FIFO are the best you can do (restricting yourself to deterministic strategies) as we will prove in the following.

For comparison, we consider the optimal offline strategy which is called MIN. On a fault, MIN evicts the page whose next request occurs furthest in the future.

Theorem 5.1. *LRU and FIFO are k-competitive (compared to MIN).*

Proof. We will only prove the theorem for LRU. The proof for FIFO is very similar and can be deduced from the proof for LRU.

Consider an arbitrary request sequence σ. We will show that $c_{LRU}(\sigma) \leq k \cdot c_{MIN}(\sigma)$ with c denoting the costs that are proportional to the number of expensive requests (cache misses). W.l.o.g. we assume that the fast memory contains the very same pages at the beginning for LRU and MIN. We partition σ into phases P such that LRU has at most k faults in P_0 and exactly k in every subsequent phase P_i. We will show that MIN has at least one fault in every phase which concludes the proof.

Consider at first P_0. The first fault for LRU implies thet there was a request to a page that was not contained in the initial fast memory. As this is the same initial configuration as for MIN, MIN also has to have a fault on this request.

Now consider some phase $P_i, i > 0$. Let $\sigma(t_i)$ be the first request in P_i and $\sigma(t_{i+1} - 1)$ the last. Furthermore let p be the last page requested in P_{i-1}. We want to show that P_i contains requests to k distinct pages that are all different from p. If this is the case, then MIN obviously has to have a fault in P_i. Suppose that LRU faults twice on the same page q. Let these faults occur at requests $\sigma(t) = q$ and $\sigma(t') = q$ with $t_i \leq t < t' \leq t_{i+1} - 1$. Right after t, page q is in the fast memory. To create a fault on q at time t' it has to be evicted before again. To achieve this, q has to be the last recently used element. Therefore the request sequence between t and t' has to contain $k+1$ distinct requests to pages of which k have to be different from q. So the only problem would be, that LRU faults on p (as MIN would not have to, as p is present in the fast access memory at the beginning of the phase for sure and the $k-1$ fault causing requests for LRU could be served). Let t be the point in time p is evicted. With the same argument as above, p needs to be the LRU element in the cache for that, which enforces at least k requests to pages distinct from p.

As for every k faults made by LRU, MIN has at least one fault as well, we can conclude that LRU is k-competitive. □

No deterministic strategy can outperform LRU as stated in the following theorem.

Theorem 5.2. *For a c-competitive deterministic strategy for the Paging problem it yields $c \geq k$.*

Proof. We proof this by constructing a worst-case request sequence for any kind of deterministic strategy. In fact, let there be $k+1$ pages in the memory system. Then σ is constructed such that always the page not in the cache is requested. So the deterministic strategy has a fault on every request. Let MIN have a fault on request i. Then obviously the next $k-1$ requested pages are already loaded into the fast access memory. So the next fault can occur earliest at request $i+k$. Thus, MIN at most one fault per k consecutive requests, while the deterministic algorithm has exactly k. Therefore a competitiveness factor better than k is not possible. □

The theoretical results imply that the performance of deterministic strategies declines with a growing size of the fast memory system. Interestingly, in practice quite the opposite is true. If the cache is large, LRU and FIFO perform much better.

5.1.2 Marking

We now consider a randomized Paging strategy which outperforms all possible deterministic ones by far. Instead of being k-competitive, MARKING can be proven to be $2H_k$-competitive which is significantly better.

The MARKING strategy proceeds in phases. At the beginning of each phase all pages are unmarked. Every time a page is requested it gets marked. If a page has to be evicted from the cache, this page is chosen u.a.r. from the unmarked pages in the cache. If all pages are marked, a new phase begins and all markers are deleted.

Theorem 5.3. *MARKING is $2H_k$-competitive.*

Proof. Given a request sequence σ. W.l.o.g. we assume MARKING already faults on the very first request. Formally a phase in the MARKING algorithm staring with request $\sigma(i)$ end with $\sigma(j), j > i$ with j being the smallest integer such that the request sequence from i to $j+1$ contains $k+1$ distinct pages. This is obviously the requirement for having only marked pages in the cache at the end. Consider some phase P_i. We say a page is stale if it is unmarked but was marked in phase P_{i-1}. We call a page clean if it is neither marked nor stale. So clean pages are not in the cache for sure, stale pages might. With c we refer to the number of clean pages that are requested in P_i. We are going to show that MIN has at least $c/2$ faults in that phase, while MARKING has at most $c \cdot H_k$.

Let S_{MIN} be the cache content when using the MIN strategy and S_{MARK} the one using MARKING. We define $d_I = |S_{MIN} \setminus S_{MARK}|$ as the number of pages at the beginning of a phase that are in the cache of MIN but not also in the cache of MARKING. Similarly, we define $d_F = |S_{MIN} \setminus S_{MARK}|$ for the end of a phase. We observe that MIN has at least $c - d_I$ faults in P_i as at least this many requested clean pages are not in the cache when using MIN. Also, MIN has at least d_F page faults during the phase, because d_F pages requested during the phase are not in MIN's fast memory (at the end of the phase). We conclude that MIN has at least

$$\max(c - d_I, d_F) \geq \frac{1}{2}(c - d_I + d_F) = \frac{c}{2} - \frac{d_I}{2} + \frac{d_F}{2}$$

many faults in the phase. Summing over all phases, the terms $d_F/2$ and $d_I/2$ telescope, except for the very first and the very last. As the first d_I is 0 (because $S_{MIN} = S_{MARK}$) we can conclude that the number of faults MIN exhibits when processing σ is at least

$$\frac{c}{2} + \frac{d_F}{2} \geq \frac{c}{2}.$$

Next we analyze MARKING. Serving c request to clean pages obviously causes c faults as no clear page can be contained in the fast memory. Furthermore there are $s = k - c \leq k - 1$ requests to stale pages. For $i = 1, \cdots, s$ we compute the expected cost of the i-th request to a stale page. Let $c(i)$ be the number of clean pages that were requested in the phase before request number i. And let $s(i)$ denote the number of stale pages that remain before the request i. Then obviously $s(i) = k - (i-1)$. When MARKING serves the i-th request to a stale page, exactly $s(i) - c(i)$ of the $s(i)$ stale pages are in fast memory, each of them with equal probability. So the expected cost of the request i can be expressed as

$$\frac{s(i) - c(i)}{s(i)} \cdot 0 + \frac{c(i)}{s(i)} \cdot 1 \leq \frac{c}{k - i + 1}.$$

The total expected costs for serving requests to stale pages is then:

$$\sum_{i=1}^{s} \frac{c}{k+1-i} \leq \sum_{i=2}^{k} \frac{c}{i} = c(H_k - 1)$$

Altogether we have to pay c for requests to clean pages and $c(H_k - 1)$ for requests to stale pages. In total this makes cH_k. Compared to the $c/2$ faults that MIN causes at least, MARKING is $2H_k$ competitive in expectation. □

So we showed that evicting pages using a random component is significantly better than with any deterministic strategy.

5.2 When to Rent Skies

Assume you go skiing regularly, but only if the weather is appropriate. Every day you face the decision of either renting skies for an unbeatable offer of 1 Euro per day, or finally buy your own skies for M Euro. You cannot predict how many good skiing days will follow. Maybe the day you buy skies will be the very last. The optimal a posteriori solution would be to always rent if the number of days with good weather T is smaller than M, and otherwise buy the skies on day one. So the optimal costs can be expressed as $\min(T, M)$. The question now is when to buy skies if you are not aware of T. If you just buy immediately, and the good weather stops right after that, the optimal solution would cost 1 but you paid M, so the competitive ratio is M. As M can be arbitrarily large, this strategy obviously is arbitrarily bad. If you decide to never buy your own skies but stick to renting, when you are going to pay T. In case $T > M$ the competitive ratio is $T/M > 1$ and can be arbitrarily large just like before. Accordingly in order to be able to guarantee anything, you have to buy skies eventually.

5.2.1 Simple Deterministic Strategy and Lower Bound

An easy way to make sure that you pay not too much compared to the optimal offline solution is to buy skies after M days of rental.

Lemma 5.4. *The simple deterministic strategy is 2-competitive.*

Proof. If $T < M$, you always rent and that is what the optimal strategy suggests as well. Therefore in this case the competitive ratio is 1. Otherwise, you pay M for rental and after that M for buying the skies, so $2M$ in total. Looking back, the optimal strategy would have been to buy skies immediately, leading to costs of M. So the competitive ratio of this strategy is bounded by $2M/M = 2$ □

In fact, this is not too bad. Especially because with any kind of deterministic strategy you cannot guarantee anything better as shown in the next theorem.

Theorem 5.5. *Every deterministic strategy for the Ski-Rental problem is at least 2-competitive.*

Proof. Every deterministic strategy can be described as 'buy after X days of rental' with some fixed value for X. We already saw that $X = 0$ and $X = \infty$ lead to arbitrary bad solutions compared to the optimal strategy. So X should be positive and finite. In this case the cost of the deterministic strategy can be expressed as $c_{DET} = X + M$. We have argued before that $c_{OPT} = \min(X, M)$. Hence the competitive ratio is

$$\frac{c_{DET}}{c_{OPT}} = \frac{X + M}{\min(X, M)} = \frac{X}{\min(X, M)} + \frac{M}{\min(X, M)}.$$

Obviously both summands are at least 1 as the denominater is smaller or equal to the numerator. Therefore the ratio can be lower bounded by 2. □

5.2.2 Randomized Strategies for Ski Rental

Now lets explore if we can get beyond the deterministic lower bound of 2 for the competitiveness factor if we use a bit of randomization to determine the point in time to buy skies. Lets say we throw a coin once. If it shows HEAD we buy skies after $M/2$ days, otherwise after M days.

The worst case (just like before) would be that after buying skies the weather will never be good enough again to go skiing. So we analyze two cases now, with the first case being $T = M$ and the second case $T = M/2$.

If $T = M$ then $c_{OPT} = M$. The expected cost of the randomized strategy is

$$\frac{1}{2} \cdot 2M + \frac{1}{2} \cdot \frac{3}{2}M = \frac{7}{4}M$$

with the first summand coinciding with the case TAIL shows up and we buy skies after M days and the second summand expressing that we bought the skies already after $M/2$ days.

If $T = M/2$ then $c_{OPT} = M/2$ as well. The expected cost of the randomized strategy is

$$\frac{1}{2} \cdot \frac{M}{2} + \frac{1}{2} \cdot \frac{3}{2}M = M$$

with the first term denoting that TAIL showed up and we rented for $M/2$ days and the second term that we rented for $M/2$ days as well, but bought skies right after that. So the ratio of expected cost and optimal cost is 2 – which unfortunately is not better than the deterministic competitiveness factor.

So lets play a bit with the earliest point in time we might buy skies. Before we said this is $M/2$, now lets generalize this to αM with $\alpha \in (0, 1)$. Again, with a probability of $1/2$ each, we buy skies either after αM or M days of rental. Let reconsider the case analysis. If $T = M$ the expected costs are now

$$\frac{1}{2}(\alpha + 1)M + \frac{1}{2} \cdot 2M = \frac{\alpha + 3}{2}M.$$

So the ratio is $\frac{\alpha + 3}{2}$. If $T = \alpha M$ the expected costs are

$$\frac{1}{2}(\alpha + 1)M + \frac{1}{2}\alpha M = \frac{2\alpha + 1}{2}M.$$

With the optimal cost being equal to αM, the resulting ratio is $1 + \frac{1}{2\alpha}$. The question is, for what value of α the maximum of $\frac{\alpha + 3}{2}$ and $1 + \frac{1}{2\alpha}$ is minimized (as the maximum of those two determines the competitiveness factor). As these terms are antiproportional in dependency of α they have to be equal to minimize the maximum. Therefore we have to solve

$$\frac{\alpha + 3}{2} = 1 + \frac{1}{2\alpha}$$

which is the same as asking for α such that $\alpha^2 + \alpha - 1 = 0$. The solution for this in $(0, 1)$ is $\alpha = \frac{\sqrt{5} - 1}{2}$. Plugging in this value, we get a competitiveness factor of ≈ 1.809 which clearly is better than what we could achieve with the deterministic strategy. In fact, this is not the end of what is possible when applying randomization. If choosing a careful probability distribution for buying skies at time $X \in [0, M]$ one can prove a competitive ratio of $\frac{e}{e - 1} \approx 1.58$.

5.3 The BahnCard Problem

We now consider a generalization of the Ski Rental problem called the BahnCard problem. Assume you can now buy a BahnCard for C Euro which is valid T days and leads to a reduced ticket price of β times the original price with $\beta \in [0, 1]$. So buying a student BahnCard 50 for 136 Euro, we get $C = 136, T = 365, \beta = 0.5$. A normal BahnCard 25 costs $C = 62$ Euro at the moment, and $T = 365, \beta = 0.75$. The BahnCard problem is to decide for a sequence of dates on which to buy a BahnCard in order to pay the smallest amount of money for train journeys (including the BahnCard costs) when knowing nothing about future journeys.

How can we express the Ski rental problem in terms of the BahnCard problem? Well, the cost of buying skies is $C = M$. As we buy unbreakable skies only, of course, we get $T = \infty$. And as soon as we bought a pair of skies, we never have to pay anything for renting again, resulting in $\beta = 0$.

Lets investigate strategies for the more general BahnCard problem now.

5.3.1 Deterministic Strategies

Let $\sigma = \sigma_1, \sigma_2, \cdots$ be the sequence of journeys, each $\sigma_i \in \sigma$ being specified by the departure date t_i and the regular price p_i.

One valid strategy would be to never buy a BahnCard. We observed for the Ski rental problem that never buying skies can be arbitrarily bad. How does this fact generalize? The cost of this strategy is simply the sum of all regular prices, i.e. $\sum_i p_i$. The cost of the optimal solution is the very same, if buying a BahnCard never pays off and otherwise at least $\sum_i \beta p_i + C$. Hence the ratio of those two terms never gets worse than $1/\beta$. For a BahnCard 50, for example, this leads to a competitive ratio of 2 which is not too bad. For a BahnCard 25, the ratio would be even $4/3$. But for a BahnCard 100, where $\beta = 0$, the factor cannot be bounded as the ratio converges to ∞. The same is true for the Ski Rental problem, as already observed in the analysis for this special case. So the NEVER strategy seems only to be useful for large values of β. Therefore, to be able to guarantee something for small values of β or completely independent of β we should possibly buy a BahnCard at some point in time.

In concurrency with our 2-competitive deterministic strategy for the Ski Rental problem, we now present a generalized version of this strategy for the BahnCard problem which can be proven to be $2 - \beta$ competitive. So 2 for $\beta = 0$ is the worst case.

To state the strategy we introduce some further notation. For an interval I, we call $p^I := \sum_{i:t_i \in I} p_i$ the partial costs for I. Accordingly, the partial strategy costs are defined as $c^I := \sum_{i:t_i \in I} cost(\sigma_i)$ with $cost(\sigma_i) = p_i$ if at time t_i we have no valid BahnCard, and $cost(\sigma_i) = \beta p_i$ otherwise. We call $c_{crit} = \dfrac{C}{1-\beta}$ the critical value, because if the summed regular ticket prices exceed this value in an interval, having bought a BahnCard at the beginning of the interval would have been beneficial. Accordingly, we call an interval with smaller partial costs than c_{crit} cheap, and otherwise expensive.

The following deterministic strategy is called SUM. Using SUM we buy a Bahncard at request $\sigma_i = (t_i, p_i)$ if we do not already have a BahnCard valid at t_i and the accumulated ticket costs since the last BahnCard turned invalid exceed c_{crit}.

Theorem 5.6. *SUM is $(2 - \beta)$-competitive.*

Proof. We compare to an optimal offline solution c_{OPT}. Here BahnCards would have been bought at times $\tau_1 < \tau_2 < \cdots < \tau_k$. We now consider the periods induced by these points in time, i.e. $[\tau_j, \tau_{j+1})$ for $j = 0, \cdots, k$ with fixing for technical reasons $\tau_0 = 0$ and $\tau_{k+1} = \infty$. Obviously each period has to start with an expensive interval (otherwise buying a BahnCard at the beginning would make no sense). Then a (possibly empty) cheap interval $[\tau_j + T, \tau_{j+1})$ follows. Using SUM we buy a BahnCard only after the summed ticket prices exceed the critical value, i.e. at most once per period $[\tau_j, \tau_{j+1})$. For all cheap phases according to an optimum solution, we know that $c^I_{SUM} \leq c^I_{OPT}$, because in the optimal solution no BahnCard is valid in this interval and in the SUM solution it might. So we only have to observe expensive phases. let I be such an expensive phase. We subdivide I into I_1, I_2, I_3 (some of them possible empty) with SUM having a valid BahnCard in I_1 and I_3 but not in I_2. Let s_1, s_2, s_3 denote the partial costs for these intervals. When we can calculate the costs for SUM in I as

$$c^I_{SUM} \leq C + s_2 + \beta(s_1 + s_3)$$

and the optimal costs as

$$c^I_{OPT} = C + \beta(s_1 + s_2 + s_3).$$

The ratio of those two is then upper bounded by $\frac{C + c_{crit}}{C + \beta c_{crit}}$ which for $c_{crit} = \frac{C}{1-\beta}$ equals $2 - \beta$. □

Observe that for $C = M, T = \infty$ and $\beta = 0$ this indeed is the same strategy that we had a close look on for the Ski Rental problem. And again, we are going to prove that this strategy is the best one can do in the sense that there exists no deterministic strategy which exhibits a better competitiveness factor.

Theorem 5.7. *No deterministic strategy for the BahnCard problem exhibits a competitiveness factor better than* $2 - \beta$.

Proof. Let $\mathcal{A}$ be a deterministic online algorithm for the BahnCard problem, and let $\epsilon > 0$ a parameter. As long as the deterministic algorithm does not demand to buy a BahnCard we assume a dense series of journeys in $[0, T)$ each with price $p = \epsilon$. As soon as the deterministic strategy decides to buy a BahnCard, we assume there no more journeys in the future (as this is obviously the worst case). Let s be the accumulated costs of the journeys without the very last one. Then the total cost induced by using algorithm $\mathcal{A}$ is s plus the cost of the BahnCard C plus the reduced ticket price for the very last journey, namely $\beta\epsilon$, i.e.

$$c_{\mathcal{A}} = s + C + \beta\epsilon.$$

The optimal cost can be expressed as $c_{OPT} = s + \epsilon$ if $s + \epsilon < c_{crit}$ because then buying a BahnCard does not pay off. Otherwise the best achievable total cost would be $c_{OPT} = C + \beta(s + \epsilon)$ as then buying the BahnCard immediately would be the best. Plugging in $c_{crit} = \frac{C}{1-\beta}$ we observe that the ratio of $c_{\mathcal{A}}$ and c_{OPT} is lower bounded by

$$2 - \beta - \frac{\epsilon(1-\beta)^2}{C}.$$

As ϵ can be chosen arbitrarily small, there can exist no algorithm $\mathcal{A}$ such that a competitive ratio of less than $2 - \beta$ can be guaranteed. □

5.3.2 Randomized Strategy

Just like for the Ski Rental problem, we will observe in the following that with the help of randomization we can break this bound. Here is how the randomized strategy – we refer to it as R-SUM – works: Every time SUM would buy a BahnCard, we buy a BahnCard using R-SUM with probability $p = \frac{1}{1+\beta}$. This small modification of the algorithm improves the (expected) competitiveness significantly as stated in the next theorem.

Theorem 5.8. *R-SUM is* $\frac{2}{1+\beta}$*-competitive.*

Proof. We analyze the strategy in the same way we analyzed SUM. So again, consider τ-periods consisting each of an expensive and a cheap phase. For the cheap phases we still know that R-SUM can only be better than the optimal solution in those intervals, hence we neglect them. So let I be again an expensive phase, split up in the three subintervals I_1, I_2, I_3 just like before, and $c^I_{OPT} = C + \beta(s_1 + s_2 + s_3)$ the respective optimal costs. The expected costs in this interval using R-SUM can be expressed as $c^I_{R-SUM} = pC + s_2 + p\beta(s_1 + s_3) + (1 - p)(s_1 + s_3)$. Computing the ratio of both terms we can conclude that

$$\frac{c_{R-SUM}}{c_{OPT}} \leq \frac{2}{1+\beta}$$

because the quotient grows with s_2 but by definition it yields $s_2 \leq c_{crit}$. □

Observe that for all values $\beta \in (0, 1)$ the term $\frac{2}{1+\beta}$ is better than $2 - \beta$, so the randomized strategy beats the deterministic one occasionally.

6 The Probabilistic Method

the probabilistic method is a way to show the existence of a certain structure or object with specific characteristics by proving that a randomized construction is successful with positive probability. Recall that we already saw applications of the probabilistic method in the lecture, in fact when we discussed MAX 3-SAT, MaxCut and ϵ-nets. Another very famous area of application of the probabilistic method is graph colouring. But we will first investigate in minimal sizes of independent sets in graphs.

6.1 Independent Set

●**Definition (Independent Set)** *Let $G(V,E)$ be a graph. A subset $I \subseteq V$ of the nodes is called an independent set if $\forall v,w \in I : \{v,w\} \notin E$. So no two points in the independent set are connected via an edge in G.*

Determining the largest independent set in a graph is NP-hard. But luckily, using the probabilistic method, we can prove a good lower bound on the size of an independent set. As seen before for MAX 3-SAT such a lower bound can be used to construct a Las Vegas algorithm .

Theorem 6.1. *Let $G(V,E)$ be a graph with $|V| = n$ and $|E| = m$ and no isolated vertices. Then G exhibits an independent set of size $\frac{n^2}{4m}$.*

Proof. We define $d := \frac{2m}{n}$ to be the average degree of a node in G. On that basis we run the following randomized algorithm: (1) Delete each vertex in G (and the respective incident edges) with probability $1 - {}^1\!/\!_d$. (2)For each remaining edge, remove the edge and one of the incident vertices.

This obviously provides us with an independent set in G. But how large is it in expectation? In the first step each vertex has a chance of survival of ${}^1\!/\!_d$. If X denotes the random variable that counts remaining vertices after step 1, we get $E(X) = {}^n\!/\!_d$. Let Y be the number of edges that survive step 1. An edge only survives if both endpoints survive. Initially we have $m = \frac{nd}{2}$ edges. So

$$E(Y) = \frac{nd}{2} \cdot \frac{1}{d^2} = \frac{n}{2d}$$

. In step 2 for each edge at most one endpoint is deleted, so at most Y additional ones. If Z corresponds to the final number of nodes, then obviously $Z \geq X - Y$. The expected value of Z can then be computed as:

$$E(Z) \geq E(X) - E(Y) = \frac{n}{d} - \frac{n}{2d} = \frac{n}{2d} = \frac{n^2}{4m}$$

□

6.2 The Lovasz Local Lemma

A very general observation in the context of the probabilistic method is the following one. If in a set of events all are independent of each other, and each has a probability of less than 1, then there is a positive probability that none of the events occur. So if the events describe all the undesired features of a structure, with the help of the probabilistic method, we can show that there exists a structure exhibiting none of these features if the above statement applies. Unfortunately, we often have to deal with events that depend on each other. In this case the observation no longer applies. But Lovasz proved that with the events being 'mostly' independent, we still can use this technique. The notion of 'mostly' is specified in the following lemma.

Lemma 6.2 (Lovasz Local Lemma)**.** *Let $E_1, E_2, \cdots, E_k$ be events with each E_i occurring with probability at most $p < 1$. If the probability of E_i to happen depends on at most d of the other events and $ep(d+1) \leq 1$, then there exists a nonzero probability that none of the events occur.*

To see how this can be useful, consider the following example.

Example (Colour Points on a Circle) *Let there be $11n$ points placed in some way around a circle. The points are coloured with n different colours, such that always exactly 11 points share a colour. We claim that in any such colouring there must be a set of n points which covers all colours but no two points in this set are adjacent, i.e. neighbours on the circle.*

To prove this, we pick a point of each colour randomly. So each point has the chance of $1/11$ to be chosen. On that basis, we define $11n$ events that correspond to the cases that neighbours were picked. Clearly, we want to avoid all these events. For each pair of neighbours, the probability to pick both points is 0 if they have the same colour and $1/121$ otherwise, so our p is $1/121$. What are the dependencies between the events? Whether a pair of neighbours a, b is chosen depends only on the other vertices in the colour classes of a and b. So an event E_i depends only on events E_j for which at least one point in the respective neighbouring pair shares a colour with a point in the pair corresponding to E_i. There are obviously 11 points on the circle sharing the colour of a, each of which is involved in two events. So there are 21 pairs besides a, b which include a point with the colour of a. The same holds for b, so in total there can be no more than $d = 42$ dependent events. Plugging these values in the Local lemma, we get $e \cdot \dfrac{1}{121} \cdot 43 \approx 0.966 < 1$. So we have a positive probability that none of the bad events occur. Therefore there has to exist a set of the required properties.