Technische Universiteit
Eindhoven
University of Technology

Course Notes for

Advanced Algorithms (2IMA10)

Mark de Berg

Contents

Part I

APPROXIMATION ALGORITHMS

Chapter 1

Introduction to Approximation Algorithms

Many important computational problems are difficult to solve optimally. In fact, many of those problems are *NP-hard*[1], which means that no polynomial-time algorithm exists that solves the problem optimally unless P=NP. A well-known example is the *Euclidean traveling salesman problem* (Euclidean TSP): given a set of points in the plane, find a shortest tour that visits all the points. Another famous NP-hard problem is Independent Set: given a graph $G = (V, E)$, find a maximum-size independent set $V^* \subset V$. (A subset is independent if no two vertices in the subset are connected by an edge.)

What can we do when faced with such difficult problems, for which we cannot expect to find polynomial-time algorithms? Unless the input size is really small, an algorithm with exponential running time is not useful. We therefore have to give up on the requirement that we always solve the problem optimally, and settle for a solution close to optimal. Ideally, we would like to have a guarantee on how close to optimal the solution is. For example, we would like to have an algorithm for Euclidean TSP that always produces a tour whose length is at most a factor ρ times the minimum length of a tour, for a (hopefully small) value of ρ. We call an algorithm producing a solution that is guaranteed to be within some factor of the optimum an *approximation algorithm.* This is in contrast to *heuristics*, which may produce good solutions but do not come with a guarantee on the quality of their solution.

1.1 Basic terminology

An *optimization problem* is, informally speaking, a problem for which there are many different solutions that each have a certain value associated to it. The goal is then to find a solution that is *valid* (or: *feasible*) and *best possible.* Optimization problems come in two flavors: minimization problems and maximization problems. For *minimization problems* “best possible” means a solution with minimum value, and for *maximization problems* “best possible” means a solution with maximum value. Euclidean TSP is an example of a minimization problem; valid solutions are tours that visit every point exactly once, and the value associated to a tour is its length. Independent Set is an example of a maximization problem; valid solutions are independent sets and the value associated to an independent set is its number of vertices.

[1] Chapter 36 of [CLRS] gives an introduction to the theory of NP-hardness.

From now on we will use $\textsc{opt}(I)$ to denote the value of an optimal solution to the problem under consideration for input I. For instance, when we study Euclidean TSP then $\textsc{opt}(P)$ will denote the length of a shortest tour on a point set P, and when we study Independent Set then $\textsc{opt}(G)$ will denote the maximum size of any independent set of the input graph G. When no confusion can arise we will sometimes simply write $\textsc{opt}$ instead of $\textsc{opt}(I)$. We denote the value of the solution that a given approximation algorithm computes for input I by $\textsc{Alg}(I)$, or simply by $\textsc{Alg}$ when no confusion can arise. In the remainder we make the assumption that $\textsc{opt}(I) \geqslant 0$ and $\textsc{Alg}(I) \geqslant 0$, which will be the case in all problems and algorithms we shall discuss.

As mentioned above, approximation algorithm come with a guarantee on the relation between $\textsc{opt}(I)$ and $\textsc{Alg}(I)$. This is made precise by the following definition.

Definition 1.1

(i) An algorithm $\textsc{Alg}$ *for a minimization problem is called a* ρ-approximation algorithm, *for some* $\rho > 1$*, if* $\textsc{Alg}(I) \leqslant \rho \cdot \textsc{opt}(I)$ *for all inputs* I*.*

(ii) An algorithm $\textsc{Alg}$ *for a maximization problem is called a* ρ-approximation algorithm, *for some* $\rho < 1$*, if*[2] $\textsc{Alg}(I) \geqslant \rho \cdot \textsc{opt}(I)$ *for all inputs* I*.*

Note that any ρ-approximation algorithm for a minimization problem is also a ρ'-approximation algorithm for any $\rho' > \rho$. For example, any 2-approximation algorithm is also a 3-approximation algorithm. Thus for an algorithm $\textsc{Alg}$ for a minimization problem it is interesting to find the smallest ρ such that $\textsc{Alg}$ is a ρ-approximation algorithm. We can call this the approximation ratio (or: approximation factor) of $\textsc{Alg}$.

Definition 1.2 *The* approximation ratio *of an algorithm* $\textsc{Alg}$ *for a minimization problem is defined as the smallest* ρ *such that* $\textsc{Alg}$ *is a* ρ*-approximation algorithm. In other words, the approximation ratio is equal to* $\sup_I \textsc{Alg}(I)/\textsc{opt}(I)$*, where the supremum is over all possible inputs* I*.*

To prove that the approximation ratio of an approximation algorithm $\textsc{Alg}$ (for a minimization problem) is equal to ρ we must show two things.

- First, we must prove that $\textsc{Alg}$ is a ρ-approximation algorithm: we must prove that $\textsc{Alg}(I) \leqslant \rho \cdot \textsc{opt}(I)$ for *all* inputs I.
- Second, we must show that for any $\rho' < \rho$ there is *some* input I such that $\textsc{Alg}(I) > \rho' \cdot \textsc{opt}(I)$. Often we actually show that there is some input I with $\textsc{Alg}(I) = \rho \cdot \textsc{opt}(I)$.

Remark. In many texts (and possibly also in these Course Notes ...) the term approximation ratio is sometimes used sloppily: after proving that an algorithm is a ρ-approximation algorithm (for some ρ) one sometimes speaks about "an algorithm with approximation ratio ρ", even though it has not been proved that this is the best possible ratio one can prove for the algorithm. When reading such statements you should always verify what is meant: is *the* approximation ratio (in the sense of Definition 1.2) equal to ρ, or is the only thing meant that the algorithm is a ρ-approximation algorithm?

[2] In some texts an algorithm for a maximization problem is called a ρ-approximation algorithm if $\textsc{Alg}(I) \geqslant (1/\rho) \cdot \textsc{opt}(I)$ for all inputs I. Thus, contrary to our definition, the approximation ratio ρ for a maximization problem is always larger than 1.

The importance of lower bounds. It may seem strange that it is possible to prove that an algorithm is a ρ-approximation algorithm: how can we prove that an algorithm always produces a solution that is within a factor ρ of OPT when we do not know OPT? The crucial observation is that, even though we do not know OPT, we can often derive a *lower bound* (or, in the case of maximization problems: an upper bound) on OPT. If we can then show that our algorithm always produces a solution whose value is at most a factor ρ from the lower bound, then the algorithm is also within a factor ρ from OPT. Thus finding good lower bounds on OPT is an important step in the analysis of an approximation algorithm. In fact, the search for a good lower bound often leads to ideas on how to design a good approximation algorithm. This is something that we will see many times in the coming chapters.

1.2 Load balancing

Suppose we are given a collection of n jobs that must be executed. To execute the jobs we have m identical machines, $M_1, \ldots, M_m$, available. Executing job j on any of the machines takes time t_j, where $t_j > 0$. Our goal is to assign the jobs to the machines in such a way that the so-called *makespan*, the time until all jobs are finished, is as small as possible. Thus we want to spread the jobs over the machines as evenly as possible. Hence, we call this problem LOAD BALANCING.

Let's denote the collection of jobs assigned to machine M_i by $A(M_i)$, and the *load* of machine M_i—the total time for which M_i is busy to execute the assigned jobs—by $\mathit{load}(M_i)$. Thus

$$\mathit{load}(M_i) = \sum_{j \in A(M_i)} t_j,$$

and the makespan of the assignment equals $\max_{1 \leqslant i \leqslant m} \mathit{load}(M_i)$. The LOAD BALANCING problem is to find an assignment of jobs to machines that minimizes the makespan, where each job is assigned to a single machine. (We cannot, for instance, execute part of a job on one machine and the rest of the job on a different machine.) LOAD BALANCING is NP-hard.

Our first approximation algorithm for LOAD BALANCING is greedy: we consider the jobs one by one and assign each job to the machine whose current load is smallest.

Algorithm 1.1 Approximation algorithm for LOAD BALANCING.

```
Greedy-Scheduling($t_1, \ldots, t_n, m$)
Initialize $load(M_i) \leftarrow 0$ and $A(M_i) \leftarrow \emptyset$ for all $1 \leqslant i \leqslant m$.
for $j \leftarrow 1$ to $n$ do
    ▷ Assign job $j$ to the machine $M_k$ of minimum load:
    Determine machine $M_k$ such that $load(M_k) = \min_{1 \leqslant i \leqslant m} load(M_i)$
    $A(M_k) \leftarrow A(M_k) \cup \{j\}$; $load(M_k) \leftarrow load(M_k) + t_j$
end for
```

This algorithm clearly assigns each job to one of the m available machines. Moreover, it runs in polynomial time. In fact, if we maintain the set $\{\mathit{load}(M_i) : 1 \leqslant i \leqslant m\}$ in a min-heap, then we can find the machine k with minimum load in $O(1)$ time and update $\mathit{load}(M_k)$ in $O(\log m)$ time. This way the entire algorithm can be made to run in $O(n \log m)$ time. The main question is how good the assignment is. Does it give an assignment whose makespan is close to $\textsc{opt}$? The answer is yes. To prove this we need a lower bound on $\textsc{opt}$, and then we must argue that the makespan of the assignment produced by the algorithm is not much more than this lower bound.

There are two very simple observations that give a lower bound. First of all, the best one could hope for is that it is possible to spread the jobs perfectly over the machines so that each machine has the same load, namely $\sum_{1\leqslant j\leqslant n} t_j/m$. In many cases this already provides a pretty good lower bound. When there is one very large job and all other jobs have processing time close to zero, however, then the upper bound is weak. In that case the trivial lower bound of $\max_{1\leqslant j\leqslant n} t_j$ will be stronger. To summarize, we have

Lemma 1.3 $\textsc{opt} \geqslant \max\left(\frac{1}{m}\sum_{1\leqslant j\leqslant n} t_j \ , \ \max_{1\leqslant j\leqslant n} t_j \right)$.

Let's define $\textsc{lb} := \max\left(\frac{1}{m}\sum_{1\leqslant j\leqslant n} t_j \ , \ \max_{1\leqslant j\leqslant n} t_j \right)$ to be the lower bound provided by Lemma 1.3. With this lower bound in hand we can prove that our simple greedy algorithm gives a 2-approximation.

Lemma 1.4 *Algorithm* Greedy-Scheduling *is a 2-approximation algorithm.*

Proof. We must prove that *Greedy-Scheduling* always produces an assignment of jobs to machines such that the makespan T satisfies $T \leqslant 2\cdot\textsc{opt}$. Consider an input $t_1,\ldots,t_n,m$. Let M_{i^*} be a machine determining the makespan of the assignment produced by the algorithm, that is, a machine such that at the end of the algorithm we have $\mathit{load}(M_{i^*}) = \max_{1\leqslant i\leqslant m} \mathit{load}(M_i)$. Let j^* be the last job assigned to M_{i^*}. The crucial property of our greedy algorithm is that at the time job j^* is assigned to M_{i^*}, machine M_{i^*} is a machine with the smallest load among all the machines. So if $\mathit{load}'(M_i)$ denotes the load of machine M_i just before job j^* is assigned, then $\mathit{load}'(M_{i^*}) \leqslant \mathit{load}'(M_i)$ for all $1 \leqslant i \leqslant m$. Hence, $\mathit{load}'(M_{i^*}) \leqslant (1/m)\cdot\sum_{1\leqslant i\leqslant m} \mathit{load}'(M_i)$. It follows that

$$\mathit{load}'(M_{i^*}) \ \leqslant\ \frac{1}{m}\sum_{1\leqslant i\leqslant m} \mathit{load}'(M_i) \ =\ \frac{1}{m}\sum_{1\leqslant j<j^*} t_j \ <\ \frac{1}{m}\sum_{1\leqslant j\leqslant n} t_j \ \leqslant\ \textsc{lb}. \qquad (1.1)$$

Thus we have $\mathit{load}'(M_{i^*}) < \textsc{lb}$, and we can derive

$$\begin{aligned}
\mathit{load}(M_{i^*}) &= t_{j^*} + \mathit{load}'(M_{i^*}) \\
&\leqslant t_{j^*} + \textsc{lb} \\
&\leqslant \max_{1\leqslant j\leqslant n} t_j + \textsc{lb} \\
&\leqslant 2\cdot\textsc{lb} \\
&\leqslant 2\cdot\textsc{opt} \qquad \text{(by Lemma 1.3)}
\end{aligned}$$

□

So this simple greedy algorithm is never more than a factor 2 from optimal. Can we do better? There are several strategies possible to arrive at a better approximation factor. One possibility could be to see if we can improve the analysis of *Greedy-Scheduling.* Perhaps we might be able to show that the approximation factor is in fact at most $c\cdot\textsc{lb}$ for some $c < 2$. Another way to improve the analysis might be to use a stronger lower bound than the one provided by Lemma 1.3. (Note that if there are instances where $\textsc{lb} = \textsc{opt}/2$ then an analysis based on this lower bound cannot yield a better approximation ratio than 2.)

It is, indeed, possible to prove a better approximation factor for the greedy algorithm described above: a more careful analysis shows that the approximation factor is in fact $(2 - \frac{1}{m})$, where m is the number of machines:

Theorem 1.5 *Algorithm* Greedy-Scheduling *is a* $(2 - \frac{1}{m})$*-approximation algorithm.*

Proof. The proof is similar to the proof of Lemma 1.4. We first slightly change (1.1) to get

$$\mathit{load}'(M_{i^*}) \leqslant \frac{1}{m} \sum_{1 \leqslant i \leqslant m} \mathit{load}'(M_i) = \frac{1}{m} \sum_{1 \leqslant j < j^*} t_j \leqslant \frac{1}{m} \left(\sum_{1 \leqslant j \leqslant n} t_j - t_{j^*} \right) \leqslant \text{LB} - \frac{1}{m} \cdot t_{j^*}. \tag{1.2}$$

Now we can derive

$$\begin{aligned} \mathit{load}(M_{i^*}) &= t_{j^*} + \mathit{load}'(M_{i^*}) \\ &\leqslant (1 - \tfrac{1}{m}) \cdot t_{j^*} + \text{LB} \\ &\leqslant (1 - \tfrac{1}{m}) \cdot \max_{1 \leqslant j \leqslant n} t_j + \text{LB} \\ &\leqslant (2 - \tfrac{1}{m}) \cdot \text{LB} \\ &\leqslant (2 - \tfrac{1}{m}) \cdot \text{OPT} \qquad \text{(by Lemma 1.3)} \end{aligned}$$

□

The bound in Theorem 1.5 is tight for the given algorithm: for any m there are inputs such that *Greedy-Scheduling* produces an assignment of makespan $(2 - \frac{1}{m}) \cdot \text{OPT}$. Thus the approximation ratio is fairly close to 2, especially when m is large. So if we want to get an approximation ratio better than $(2 - \frac{1}{m})$, then we have to design a better algorithm.

A weak point of our greedy algorithm is the following. Suppose we first have a large number of small jobs and then finally a single very large job. Our algorithm will first spread the small jobs evenly over all machines and then add the large job to one of these machines. It would have been better, however, to give the large job its own machine and spread the small jobs over the remaining machines. Note that our algorithm would have produced this assignment if the large job would have been handled first. This observation suggest the following adaptation of the greedy algorithm: we first sort the jobs according to decreasing processing times, and then run *Greedy-Scheduling*. We call the new algorithm *Ordered-Scheduling*.

Does the new algorithm really have a better approximation ratio? The answer is yes. However, the lower bound provided by Lemma 1.3 is not sufficient to prove this; we also need the following lower bound.

Lemma 1.6 *Consider a set of n jobs with processing times $t_1, \ldots, t_n$ that have to be scheduled on m machines, where $t_1 \geqslant t_2 \geqslant \cdots \geqslant t_n$. If $n > m$, then* $\text{OPT} \geqslant t_m + t_{m+1}$.

Proof. Since there are m machines, at least two of the jobs $1, \ldots, m+1$, say jobs j and j', have to be scheduled on the same machine. Hence, the load of that machine is $t_j + t_{j'}$, which is at least $t_m + t_{m+1}$ since the jobs are sorted by processing times. □

Theorem 1.7 *Algorithm* Ordered-Scheduling *is a (3/2)-approximation algorithm.*

Proof. The proof is very similar to the proof of Lemma 1.4. Again we consider a machine M_{i^*} that has the maximum load, and we consider the last job j^* scheduled on M_{i^*}. If $j^* \leqslant m$, then j^* is the only job scheduled on M_{i^*}—this is true because the greedy algorithm schedules the first m jobs on different machines. Hence, our algorithm is optimal in this case. Now consider the case $j^* > m$. As in the proof of Lemma 1.4 we can derive

$$\mathit{load}(M_{i^*}) \quad \leqslant \quad t_{j^*} + \tfrac{1}{m} \textstyle\sum_{1 \leqslant i \leqslant n} t_i.$$

The second term can be bounded as before using Lemma 1.3:

$$\tfrac{1}{m} \textstyle\sum_{1 \leqslant i \leqslant n} t_i \quad \leqslant \quad \max\left(\max_{1 \leqslant j \leqslant n} t_j \; , \; \tfrac{1}{m} \textstyle\sum_{1 \leqslant i \leqslant n} t_i \right) \quad \leqslant \quad \text{OPT}.$$

For the first term we use that $j^* > m$. Since the jobs are ordered by processing time we have $t_{j^*} \leqslant t_{m+1} \leqslant t_m$. We can therefore use Lemma 1.6 to get

$$t_{j^*} \quad \leqslant \quad (t_m + t_{m+1})/2 \quad \leqslant \quad \text{OPT}/2.$$

Hence, the total load on M_{i^*} is at most $(3/2) \cdot \text{OPT}$. □

We can actually even prove better bounds on the approximation ratio of *Ordered-Scheduling*; see Exercises 1.8 and 1.9

1.3 Exercises

Exercise 1.1 In Definition 1.1 it is stated that we should have $\rho > 1$ for minimization problems and that we should have $\rho < 1$ for a maximization problem. Explain this.

Exercise 1.2 In Definition 1.2 the approximation ratio of an algorithm for a minimization problem is defined. Give the corresponding definition for a maximization problem.

Exercise 1.3 Consider the LOAD BALANCING problem on two machines. Thus we want to distribute a set of n jobs with processing times $t_1, \ldots, t_n$ over two machines such that the makespan (the maximum of the processing times of the two machines) is minimized. Professor Smart has designed an approximation algorithm ALG for this problem, and he claims that his algorithm is a 1.05-approximation algorithm. We run ALG on a problem instance where the total size of all the jobs is 200, and ALG returns a solution whose makespan is 120.

(i) Suppose that we know that all job sizes are at most 100. Can we then conclude that professor Smart's claim is false? Explain your answer.

(ii) Same question when all job sizes are at most 10.

Exercise 1.4 Consider a company that has to schedule jobs on a daily basis. That is, each day the company receives a number of jobs that need to be scheduled (for that day) on one of their machines. The company uses the *Greedy-Scheduling* algorithm to do the scheduling. (Thus, each day the company runs *Greedy-Scheduling* on the set of jobs that must be executed on that day.) The following information is known about the company and the jobs: the company has 5 machines, the processing times t_j of the jobs are always between 1 and 25 (that is, $1 \leqslant t_j \leqslant 25$ for all j) and the total processing time of all the jobs, $\sum_{j=1}^{n} t_j$, is always at least 500.

(i) We know from Theorem 1.5 that *Greedy-Scheduling* is a (9/5)-approximation algorithm. Under the given conditions a stronger result is possible: prove that *Greedy-Scheduling* is a ρ-approximation for some $\rho < 9/5$. Try to make ρ as small as possible.

(ii) Give an example of a set of jobs satisfying the condition stated above such that the makespan produced by *Greedy-Scheduling* on this input is ρ' times the optimal makespan. Try to make ρ' as large as possible.

Note: ideally, the value for ρ' that you prove in (ii) is equal to the value for ρ that you prove in (i). If this is the case, your analysis is *tight*—it cannot be improved—and the value is *the* approximation ratio of the algorithm.

Exercise 1.5 We have seen in this chapter that algorithm *Greedy-Scheduling* is a $(2 - \frac{1}{m})$-approximation algorithm. Show that the bound $2-\frac{1}{m}$ is tight, by giving an example of an input for which *Greedy Scheduling* produces a solution in which the makespan is $(2 - \frac{1}{m}) \cdot \text{OPT}$, and argue that the makespan is indeed that large. NB: Your example should be for arbitrary m, it is not sufficient to give an example for one specific value of m.

Exercise 1.6 Give an example of an input on which neither *Greedy-Scheduling* nor *Ordered-Scheduling* gives an optimal solution.

Exercise 1.7 Theorem 1.7 states that *Ordered-Scheduling* is a (3/2)-approximation algorithm for any number of machines. In the proof of the theorem we not only used the lower bound provided by Lemma 1.3, but also the lower bound provided by Lemma 1.6. Show that we really need to use the bound from Lemma 1.6. More precisely, give a specific instance—a specific number m and a specific set of jobs—such that *Ordered-Scheduling* produces a solution whose makespan is strictly greater than $(3/2) \cdot \max\left(\frac{1}{m} \sum_{1 \leqslant j \leqslant n} t_j \;,\; \max_{1 \leqslant j \leqslant n} t_j \right)$.

Exercise 1.8 Theorem 1.7 states that *Ordered-Scheduling* is a (3/2)-approximation algorithm for any number of machines. Prove that the approximation ratio is actually slightly better, by proving that for m machines the algorithm is a $(\frac{3}{2} - \frac{1}{2m})$-approximation algorithm.

Exercise 1.9 Theorem 1.7 states that *Ordered-Scheduling* is a (3/2)-approximation algorithm for any number of machines, and Exercise 1.8 shows that it is actually a $(\frac{3}{2} - \frac{1}{2m})$-approximation algorithm. In this exercise you are asked to prove an even better bound for the special case $m = 2$. (So, for the rest of the exercise, we assume $m = 2$.)

(i) Prove that for $n \leqslant 4$—that is, if the number of jobs is at most four—then *Ordered-Scheduling* is optimal.

(ii) Give an example for $n = 5$ in which *Ordered-Scheduling* gives a makespan that is equal to $(7/6) \cdot \text{OPT}$.

(iii) Prove that for $n = 5$ the algorithm always gives a makespan that is at most $(7/6) \cdot \text{OPT}$.

(iv) Following the proof of Theorem 1.7, let M_{i^*} be a machine determining the makespan, and let j^* be the last job assigned to M_{i^*} by *Ordered-Scheduling*. Prove that then the produced makespan is at most $(1 + \frac{1}{j^*}) \cdot \text{OPT}$.

(v) Prove that the approximation ratio for *Ordered-Scheduling* is 7/6.

Exercise 1.10 Consider the following problem. A shipping company has to decide how to distribute a load consisting of a n containers over its ships. For $1 \leqslant i \leqslant n$, let w_i denote the weight of container i. The ships are identical, and each ship can carry containers with a maximum total weight W. (Thus if $C(j)$ denotes the set of containers assigned to ship j, then $\sum_{i \in C(j)} w_i \leqslant W$.) The goal is to assign containers to ships in such a way that a minimum number of ships is used. Give a 2-approximation algorithm for this problem, and prove your algorithm indeed gives a 2-approximation.
Hint: There is a very simple greedy algorithm that gives a 2-approximation.

Exercise 1.11 Let P be a set of n points in the plane. We say that a point p is *covered* by a square s, if p is contained in the boundary or interior of s. A *square cover* of P is a set S of axis-aligned unit squares (squares of size 1×1 whose edges are parallel to the x- and y-axis) such that any point $p \in P$ is covered by at least one square $s \in S$. We want to find a square cover of P with a minimum number of squares.

(i) Consider the integer grid, that is, the grid defined by the horizontal and vertical lines at integer coordinates. Cells in this grid are unit squares. Suppose we generate a square cover by taking all grid cells covering at least one point from S—see Fig. 1.1. You may assume that no point falls on the boundary between two squares. Analyze the approximation ratio of this simple strategy. (N.B. You should prove that the strategy gives a ρ-approximation, for some ρ,and give an example that where the ratio between the algorithm and the optimal solution is ρ', for some ρ'. Ideally $\rho = \rho'$, which then implies your analysis is tight.)

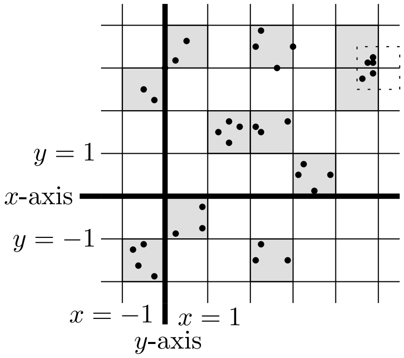

Fig. 1.1: The integer grid, and the square covering produced by it (in grey). Note that the covering is not optimal, since the two squares in the top right corner can be replaced by the dotted square.

(ii) Suppose all points lie in between two horizontal grid lines. More precisely, assume there is an integer k such that for every $p \in P$ we have $k \leqslant p_y < k + 1$, where p_y is the y-coordinate of p. Give an algorithm that computes an optimal square covering for this case. Your algorithm should run in $O(n \log n)$ time.

(iii) Using (ii), give an algorithm that computes in $O(n \log n)$ time a 2-approximation of a minimum-size square cover for an arbitrary set of points in the plane. Prove that your algorithm achieves the desired approximation ratio, and that it runs in $O(n \log n)$ time.

Exercise 1.12 Let $G = (V, E)$ be an (undirected) graph. A *vertex cover* of G is a subset $C \subset V$ such that for every edge $(u, v) \in E$ we have $u \in C$ or $v \in C$ (or both). An *independent set* of G is a subset $I \subset V$ such that no two vertices in I are connected by an edge in E.

(i) Prove the following statement: C is a vertex cover of G if and only if $V \setminus C$ is an independent set of G.

(ii) It follows from (i) that if we can compute a vertex cover for G, we can also compute an independent set for G. Now suppose we want to compute a maximum-size independent set on G, and suppose we have a 2-approximation algorithm *ApproxMinVertexCover* for finding a minimum-size vertex cover. Consider the following algorithm for computing a maximum independent set.

```
ApproxMaxIndependentSet(G)
$C \leftarrow ApproxMinVertexCover(G)$      ▷ $G = (V, E)$ is an undirected graph
return $V \setminus C$
```

Prove or disprove: *ApproxMaxIndependentSet* is a (1/2)-approximation algorithm.

Chapter 2

The Traveling Salesman Problem

Let $G = (V, E)$ be an undirected graph. A *Hamiltonian cycle* of G is a cycle that visits every vertex $v \in V$ exactly once. Instead of Hamiltonian cycle, we sometimes also use the term *tour*. Not every graph has a Hamiltonian cycle: if the graph is a single path, for example, then obviously it does not have a Hamiltonian cycle. The problem Hamiltonian Cycle is to decide for a given graph graph G whether it has a Hamiltonian cycle. Hamiltonian Cycle is NP-complete.

Now suppose that G is a *complete graph*—that is, G has an edge between every pair of vertices—where each edge $e \in E$ has a non-negative length. It is easy to see that because G is complete it must have a Hamiltonian cycle. Since the edges now have lengths, however, some Hamiltonian cycles may be shorter than others. This leads to the *traveling salesman problem*, or TSP for short: given a complete graph $G = (V, E)$, where each edge $e \in E$ has a length, find a minimum-length tour (Hamiltonian cycle) of G. (The length of a tour is defined as the sum of the lengths of the edges in the tour.) TSP is NP-hard. We are therefore interested in approximation algorithms. Unfortunately, even this is too much to ask.

Theorem 2.1 *There is no value c for which there exists a polynomial-time c-approximation algorithm for* TSP*, unless P=NP.*

Proof. As noted above, Hamiltonian Cycle is NP-complete, so there is no polynomial-time algorithm for the problem unless P=NP. Let c be any value. We will prove that if there is a polynomial time c-approximation algorithm for TSP, then we can also solve Hamiltonian Cycle in polynomial time. The theorem then follows.

Let $G = (V, E)$ be a graph for which we want to decide if it admits a Hamiltonian cycle. We construct a complete graph $G^* = (V, E^*)$ from G as follows. For every pair of vertices u, v we put an edge in E^*, where we set $\textit{length}((u, v)) = 1$ if $(u, v) \in E$ and $\textit{length}((u, v)) = c \cdot |V| + 1$ if $(u, v) \notin E$. The graph G^* can be constructed from G in polynomial time—in $O(|V|^2)$ time, to be precise. Let $\text{OPT}(G^*)$ denote the minimum length of any tour for G^*.

Now suppose we have a c-approximation algorithm Alg for TSP. Run Alg on G^*. We claim that Alg returns a tour of length $|V|$ if and only if G has a Hamiltonian cycle. For the "if"-part we note that $\text{OPT}(G^*) = |V|$ if G has a Hamiltonian cycle, since then there is a tour in G^* that only uses edges of length 1. Since Alg is a c-approximation algorithm it must return a tour of length at most $c \cdot \text{OPT}(G^*) = c \cdot |V|$. Obviously such a tour cannot use any edges of length $c \cdot |V| + 1$ and so it only uses edges that were already in G. In other words, if G has a Hamiltonian cycle then Alg returns tour of length $|V|$. For the "only if"-part, suppose Alg returns a tour of length $|V|$. Then obviously that tour can only use edge of length 1—in other words, edges from E—which means G has a Hamiltonian cycle. □

Note that in the proof we could also have set the lengths of the edges in E to 0 and the lengths of the other edges to 1. Then $\text{OPT}(G^*) = 0$ if and only if G has a Hamiltonian cycle. When $\text{OPT}(G^*) = 0$, then $c \cdot \text{OPT}(G^*) = 0$ no matter how large c is. Hence, any approximation algorithm must solve the problem exactly. In some sense, this is cheating: when $\text{OPT} = 0$ allowing a (multiplicative) approximation factor does not help, so it is not surprising that one cannot get a polynomial-time approximation algorithm (unless P=NP). The proof above shows that this is even true when all edge lengths are positive, which is a stronger result.

This is disappointing news. But fortunately things are not as bad as they seem: when the edge lengths satisfy the so-called *triangle inequality* then we *can* obtain good approximation algorithms. The triangle inequality states that for every three vertices u, v, w we have

$$\mathit{length}((u,w)) \leqslant \mathit{length}((u,v)) + \mathit{length}((v,w)).$$

In other words, it is not more expensive to go directly from u to w than it is to go via some intermediate vertex v. This is a very natural property. It holds for instance for *Euclidean TSP*. Here the vertices in V are points in the plane (or in some higher-dimensional space), and the length of an edge between two points is the Euclidean distance between them. As we will see below, for graphs whose edge lengths satisfy the triangle inequality, it is fairly easy to give a 2-approximation algorithm. With a little more effort, we can improve the approximation factor to 3/2. For the special case of Euclidean TSP there is even a PTAS; this algorithm is fairly complicated, however, and we will not discuss it here. We will use the following property of graphs whose edge lengths satisfy the triangle inequality.

Observation 2.2 *Let $G = (V,E)$ be a graph whose edge lengths satisfy the triangle inequality, and let $v_1, v_2, \ldots, v_k$ be any path in G. Then $\mathit{length}((v_1, v_k)) \leqslant \mathit{length}(v_1, v_2, \ldots, v_k)$.*

Proof. By induction on k. If $k = 2$ the statement is trivially true, so assume $k > 2$. By the induction hypothesis, we know that $\mathit{length}((v_1, v_{k-1})) \leqslant \mathit{length}(v_1, v_2, \ldots, v_{k-1})$. Moreover, $\mathit{length}((v_1, v_k)) \leqslant \mathit{length}((v_1, v_{k-1})) + \mathit{length}((v_{k-1}, v_k))$ by the triangle inequality. Hence,

$$\begin{array}{rcl} \mathit{length}((v_1, v_k)) & \leqslant & \mathit{length}((v_1, v_{k-1})) + \mathit{length}((v_{k-1}, v_k) \\ & \leqslant & \mathit{length}(v_1, v_2, \ldots, v_{k-1}) + \mathit{length}((v_{k-1}, v_k) \\ & = & \mathit{length}(v_1, v_2, \ldots, v_k). \end{array}$$

□

2.1 A simple 2-approximation algorithm

A *spanning tree* of a graph G is a tree—a connected acyclic graph—whose vertex set is V; a *minimum spanning tree* of a graph (whose edges have lengths) is a spanning tree whose total edge length is minimum among all spanning trees for G. Spanning trees and tours seem very similar: both are subgraphs of G that connect all vertices. The only difference is that in a spanning tree the connections form a tree, while in a tour they form a cycle. From a computational point of view, however, this makes a huge difference: while TSP is NP-hard,

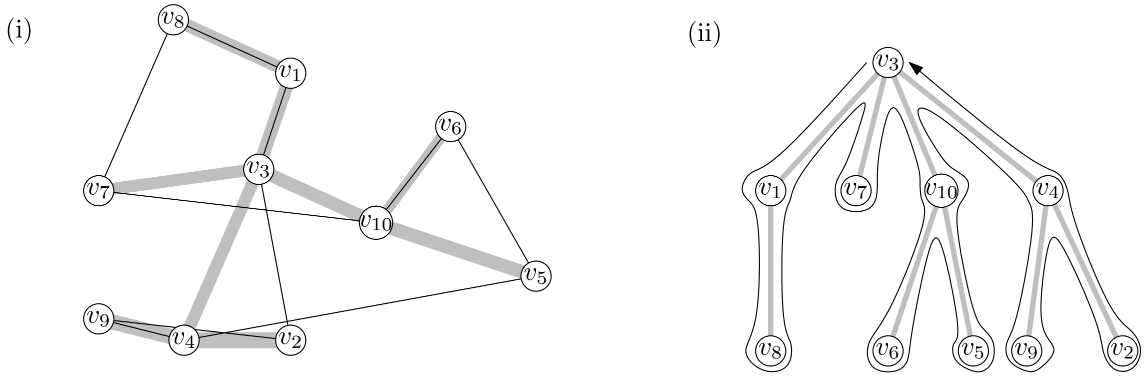

Fig. 2.1: (i) A spanning tree (thick grey) and the tour (thin black) that is found when the traversal shown in (ii) is used. (ii) Possible inorder traversal of the spanning tree in (i). The traversal results from choosing v_3 as the root vertex and visiting the children in some specific order. (A different tour would result if we visit the children in a different order.)

computing a minimum spanning tree can be done in polynomial time with a simple greedy algorithm such as Kruskal's algorithm or Prim's algorithm—see [CLRS] for details.

Now let $G = (V, E)$ be a complete graph whose edge lengths satisfy the triangle inequality. As usual, we will derive our approximation algorithm for TSP from an efficiently computable lower bound. In this case the lower bound is provided by the minimum spanning tree.

Lemma 2.3 *Let* OPT *denote the minimum length of any tour of the given graph G, and let* MST *denote the total length of a minimum spanning tree of G. Then* $\text{OPT} \geqslant \text{MST}$.

Proof. Let Γ be an optimal tour for G. By deleting an edge from Γ we obtain a path Γ' and since all edge lengths are non-negative we have $\mathit{length}(\Gamma') \leqslant \mathit{length}(\Gamma) = \text{OPT}$. Because a path is (a special case of) a tree, a minimum spanning tree is at least as short as Γ'. Hence, $\text{MST} \leqslant \mathit{length}(\Gamma') \leqslant \text{OPT}$. □

So the length of a minimum spanning tree provides a lower bound on the minimum length of a tour. But can we also use a minimum spanning tree to compute an approximation of a minimum-length tour? The answer is yes: we simply make an arbitrary vertex of the minimum spanning tree $\mathcal{T}$ to be the root and use an inorder traversal of $\mathcal{T}$ to get the tour. (An inorder traversal of a rooted tree is a traversal that starts at the root and then proceeds as follows. Whenever a vertex is reached, we first visit that vertex and then we recursively visit each of its subtrees.) Figure 2.1 illustrates this. We get the following algorithm.

Algorithm 2.1 Approximation algorithm for TSP.

```
ApproxTSP($G$)
Compute a minimum spanning tree $\mathcal{T}$ for $G$.
Turn $\mathcal{T}$ into a rooted tree by selecting an arbitrary vertex $u \in \mathcal{T}$ as the root.
Initialize an empty tour $\Gamma$.
InorderTraversal($u, \Gamma$)
Append $u$ to $\Gamma$, so that the tour returns to the starting vertex.
return $\Gamma$
```

Below the algorithm for the inorder traversal of the minimum-spanning tree $\mathcal{T}$ is stated. Recall that we have assigned an arbitrary vertex u as the root of $\mathcal{T}$, where the traversal is started. This also determines for each edge (u, v) of $\mathcal{T}$ whether v is a child of u or vice versa.

```
InorderTraversal($u, \Gamma$)
1: Append $u$ to $\Gamma$.
2: for each child $v$ of $u$ do
3:     InorderTraversal($v, \Gamma$)
4: end for
```

The next theorem gives a bound on the approximation ratio of the algorithm.

Theorem 2.4 ApproxTSP *is a 2-approximation algorithm.*

Proof. Let Γ denote the tour reported by the algorithm, and let MST denote the total length of the minimum spanning tree. We will prove that $\mathit{length}(\Gamma) \leqslant 2 \cdot \text{MST}$. The theorem then follows from Lemma 2.3.

Consider an inorder traversal of the minimum spanning tree $\mathcal{T}$ where we change line 3 to

```
3.     do InorderTraversal($v, \Gamma$); Append $u$ to $\Gamma$.
```

In other words, after coming back from recursively visiting a subtree of a node u we first visit u again before we move on to the next subtree of u. This way we get a cycle Γ' where some vertices are visited more than once, and where every edge in Γ' is also an edge in $\mathcal{T}$. In fact, every edge in $\mathcal{T}$ occurs exactly twice in Γ', so $\mathit{length}(\Gamma') = 2 \cdot \text{MST}$. The tour Γ can be obtained from Γ' by deleting vertices so that only the first occurrence of each vertex remains. This means that certain paths $v_1, v_2, \ldots, v_k$ are shortcut by the single edge (v_1, v_k). By Observation 2.2, all the shortcuts are at most as long as the paths they replace, so $\mathit{length}(\Gamma) \leqslant \mathit{length}(\Gamma') \leqslant 2 \cdot \text{MST}$. □

Is this analysis tight? Unfortunately, the answer is basically yes: there are graphs for which the algorithm produces a tour of length $(2 - \frac{1}{|V|}) \cdot \text{OPT}$, so when $|V|$ gets larger and larger the worst-case approximation ratio gets arbitrarily close to 2. Hence, if we want to improve the approximation ratio, we have to come up with a different algorithm. This is what we do in the next section.

2.2 Christofides's (3/2)-approximation algorithm

Our improved approximation algorithm, which is due to Nicos Christofides, also starts by computing a minimum spanning tree. The difference with our 2-approximation algorithm is that the shortcuts are chosen more cleverly. This is done based on the following two facts.

An *Euler tour* of an undirected graph is a cycle that visits every edge exactly once; note that it may visit a vertex more than once.[1] Determining whether a graph has a Hamiltonian cycle is hard, but determining whether it has an Euler tour is quite easy: a connected undirected graph has an Euler tour if and only if the degree of every vertex is even. Moreover, it is not only easy to determine if a graph has an Euler tour, it is also easy to compute one if it exists.

[1] This terminology is standard but perhaps a bit unfortunate, since an Euler tour is not necessarily a tour under the definition we gave earlier.

Of course a minimum spanning tree—or any other tree for that matter—does not admit an Euler tour, since the leaves of the tree have degree 1. The idea is therefore to add extra edges to the minimum spanning tree such that all vertices have even degree, and then take an Euler tour of the resulting graph. To this end we need the concept of so-called matchings.

Let G be a graph with an even number of vertices. Then a *matching* is a collection M of edges from the graph such that every vertex is the endpoint of at most one edge in M. The matching is called *perfect* if every vertex is incident to exactly one edge in M, and if its total edge length is minimum among all perfect matchings then we call M a *minimum perfect matching*. It is known that a minimum perfect matching of a complete graph with an even number of vertices can be computed in polynomial time. (Notice that a complete graph with an even number of vertices always has a perfect matching.)

Lemma 2.5 *Let $G = (V, E)$ be a graph and let $V^* \subset V$ be any subset of an even number of vertices. Let* OPT *denote the minimum length of any tour on G, and let M^* be a perfect matching on the complete graph $G^* = (V^*, E^*)$, where the lengths of the edges in E^* are equal to the lengths of the corresponding edges in E. Then length$(M^*) \leqslant \frac{1}{2} \cdot \text{OPT}$.*

Proof. Let $\Gamma = v_1, \ldots, v_n, v_1$ be an optimal tour. Let's first assume that $V^* = V$. Consider the following two perfect matchings: $M_1 = \{(v_1, v_2), (v_3, v_4), \ldots, (v_{n-1}, v_n)\}$ and $M_2 = \{(v_2, v_3), (v_4, v_5), \ldots, (v_n, v_1)\}$. Then $\mathit{length}(M_1) + \mathit{length}(M_2) = \mathit{length}(\Gamma) = \text{OPT}$. Hence, for the minimum-length perfect matching M^* we have

$$\mathit{length}(M^*) \leqslant \min(\mathit{length}(M_1), \mathit{length}(M_2)) \leqslant \text{OPT}/2.$$

If $V^* \neq V$ then we can use basically the same argument: Let $n^* = |V^*|$ and number the vertices from V^* as $v_1^*, \ldots, v_{n^*}^*$ in the order they are encountered by Γ. Consider the two matchings $M_1 = \{(v_1^*, v_2^*), \ldots, (v_{n^*-1}, v_{n^*})\}$ and $M_2 = \{(v_2^*, v_3^*), \ldots, (v_{n^*}, v_1)\}$. One of these has length at most $\mathit{length}(\Gamma^*)/2$, where Γ^* is the tour $v_1^*, \ldots, v_{n^*}^*, v_1^*$. The result follows because $\mathit{length}(\Gamma^*) \leqslant \mathit{length}(\Gamma)$ by the triangle inequality. □

The algorithm is now as follows.

Algorithm 2.2 Christofides's algorithm for TSP.

```
ChristofidesTSP(G)
Compute a minimum spanning tree $\mathcal{T}$ for $G$.
Let $V^* \subset V$ be the set of vertices of odd degree in $\mathcal{T}$.
Compute a minimum perfect matching $M$ on the complete graph $G^* = (V^*, E^*)$.
Add the edges from $M$ to $\mathcal{T}$, and find an Euler tour $\Gamma$ of the resulting multigraph.
For each vertex that occurs more than once in $\Gamma$, remove all but one of its occurrences.
return $\Gamma$
```

Theorem 2.6 ChristofidesTSP *is a (3/2)-approximation algorithm.*

Proof. First we note that in any graph, the number of odd-degree vertices must be even—this is easy to show by induction on the number of edges. Hence, the set V^* has an even number of vertices, so it admits a perfect matching. Adding the edges from the matching M to the tree $\mathcal{T}$ ensures that every vertex of odd degree gets an extra incident edge, so all degrees become even. (Note that the matching M may contain edges that were already present in $\mathcal{T}$. Hence, after we add these edges to M we in fact have a *multi-graph*. But this is not a problem for the rest of the algorithm.) It follows that after adding the edges from M, we get a multi-graph that has an Euler tour Γ. The length of Γ is at most $\mathit{length}(\mathcal{T}) + \mathit{length}(M)$, which is at most $(3/2) \cdot \textsc{opt}$ by Lemmas 2.3 and 2.5. By Observation 2.2, removing the superfluous occurrences of the vertices occurring more than once in line 5 can only decrease the length of the tour. □

Christofides's algorithm is still the best known algorithm for TSP for graphs satisfying the triangle inequality. For Euclidean TSP, however, a PTAS exists. As mentioned earlier, this PTAS is rather complicated and we will not discuss it here.

2.3 Exercises

Exercise 2.1 Consider the algorithm *ApproxTSP* presented in Section 2.1. When the edge lengths of the input graph G satisfy the triangle inequality, *ApproxTSP* gives a 2-approximation. Let c be any constant. Give an example of a graph G where the edge weights do *not* satisfy the triangle inequality such that *ApproxTSP* returns a tour of length more than $c \cdot \textsc{opt}$, where $\textsc{opt}$ is the minimum length of any tour for G.

NB: Describing the graph is not sufficient, you should also argue that the algorithm indeed gives a tour of length more than $c \cdot \textsc{opt}$. Note that it is not sufficient to argue that the length of the tour is more than c times the length of a minimum-spanning tree, because $\textsc{opt}$ could be larger than $\textsc{mst}$.

Exercise 2.2 Consider the TSP problem for graphs where all the edges have either weight 1 or weight 2.

(i) Prove that the TSP problem is still NP-hard for these graphs.

(ii) Show that these edge weights satisfy the triangle inequality.

(iii) By part (ii) of this exercise, algorithm *ApproxTSP* gives a 2-approximation for graphs where the edge weights are 1 or 2. Someone conjectures that for these special graphs, the algorithm in fact always gives a (3/2)-approximation. Prove or disprove this conjecture.

Exercise 2.3 Let P be a set of n points in the plane. It is known that a minimum spanning tree (MST) for P can be computed in $O(n \log n)$ time. In this exercise we are also interested in computing an MST $\mathcal{T}$ for P, but we are allowed to add extra points (anywhere we like) and use those extra points as additional vertices in $\mathcal{T}$. Such a tree with extra points is called a *Steiner tree*. Computing a minimum Steiner tree is NP-hard.

(i) Show that it sometimes helps to add extra points by giving an example of a point set P and an extra point q such that an MST for $P \cup \{q\}$ is shorter than an MST for P.

(ii) Let P be a set of points and Q be any set of extra points. Prove that the length of an MST for P is never more than twice the length of an MST for $P \cup Q$. (Hence, simply computing an MST for P gives a 2-approximation for the MST-with-extra-points problem. In fact, one can show that the approximation ratio is even better, but proving the factor 2 is sufficient.)

Exercise 2.4 Theorem 2.4 states that *ApproxTSP* is a 2-approximation algorithm. Give an example of an input graph on n vertices (for arbitrary n) for which the algorithm can produce a tour of length $(2 - (1/n)) \cdot \textsc{opt}$.

Exercise 2.5 Consider the algorithm *ApproxTSP* from the Course Notes. When the edge lengths of the input graph G satisfy the triangle inequality, *ApproxTSP* gives a 2-approximation. Now suppose the edge lengths satisfy the following *weak triangle inequality*: for any three vertices u, v, w we have $\mathit{length}((u,w)) \leqslant 2 \cdot (\mathit{length}((u,v)) + \mathit{length}((v,w)))$.

(i) Explain how Observation 7.2 needs to be modified for this setting, and prove the new version of the observation.

(ii) Now prove a bound on the approximation ratio of *ApproxTSP* for graphs satisfying the weak triangle inequality.

NB: You do not have to write the complete analysis. It suffices to explain how the modified version of Observation 7.2 changes the proof of Theorem 2.4, and what the new bound on the approximation ratio will be.

Chapter 3

Approximation via LP Rounding

Let $G = (V, E)$ be an (undirected) graph. A subset $C \subset V$ is called a *vertex cover* for G if for every edge $(v_i, v_j) \in E$ we have $v_i \in C$ or $v_j \in C$ (or both). In other words, for every edge in E at least one of its endpoints is in C.

3.1 Unweighted Vertex Cover

The unweighted version of the Vertex Cover problem is to find a vertex cover of minimum size for a given graph G. This problem is NP-complete.

Let's try to come up with an approximation algorithm. A natural greedy approach would be the following. Initialize the cover C as the empty set, and set $E' := E$; the set E' will contain the edges that are not yet covered by C. Now take an edge $(v_i, v_j) \in E'$, put one of its two vertices, say v_i, into C, and remove from E' all edges incident to v_i. Repeat the process until $E' = \emptyset$. Clearly C is a vertex cover after the algorithm has finished. Unfortunately the algorithm has a very bad approximation ratio: there are instances where it can produce a vertex cover of size $|V| - 1$ even though a vertex cover of size 1 exists. A small change in the algorithm leads to a 2-approximation algorithm. The change is based on the following lower bound. Call two edges $e, e' \in E$ *disjoint* if they do not have a vertex in common.

Lemma 3.1 *Let $G = (V, E)$ be a graph and let* OPT *denote the minimum size of a vertex cover for G. Let $E^* \subset E$ be any subset of pairwise disjoint edges, that is, any subset such that each pair of edges in E^* is disjoint. Then* OPT $\geqslant |E^*|$.

Proof. Let C be an optimal vertex cover for G. By definition, any edge $e \in E^*$ must be covered by a vertex in C, and since the edges in E^* are disjoint any vertex in C can cover at most one edge in E^*. □

This lemma suggests the greedy algorithm given in Algorithm 3.1. It is easy to check that the **while**-loop in the algorithm indeed maintains the stated invariant. After the **while**-loop has terminated—the loop must terminate since at every step we remove at least one edge from E'—we have $E \setminus E' = E \setminus \emptyset = E$. Together with the invariant this implies that the algorithm indeed returns a vertex cover. Next we show that the algorithm gives a 2-approximation.

Theorem 3.2 *Algorithm* ApproxVertexCover *produces a vertex cover C such that $|C| \leqslant 2 \cdot$* OPT*, where* OPT *is the minimum size of a vertex cover.*

Algorithm 3.1 Approximation algorithm for Vertex Cover.

```
    ApproxVertexCover(V, E)
1:  $C \leftarrow \emptyset$; $E' \leftarrow E$        ▷ Invariant: $C$ is a vertex cover for $G' = (V, E \setminus E')$
2:  while $E' \neq \emptyset$ do
3:      Take an arbitrary edge $(v_i, v_j) \in E'$.
4:      $C \leftarrow C \cup \{v_i, v_j\}$.
5:      Remove $(v_i, v_j)$, and all other edges with $v_i$ or $v_j$ as an endpoint, from $E'$.
6:  end while
7:  return $C$
```

Proof. Let E^* be the set of edges selected in line 3 over the course of the algorithm. Then C consists of the endpoints of the edges in E^*, and so $|C| \leqslant 2|E^*|$. Moreover, any two edges in E^* are disjoint because as soon as an edge (v_i, v_j) is selected from E' all edges in E' that share v_i and/or v_j are removed from E'. The theorem now follows from Lemma 3.1. □

3.2 Weighted Vertex Cover

Now let's consider a generalization of Vertex Cover, where each vertex $v_i \in V$ has a weight *weight*(v_i) and we want to find a vertex cover of minimum total weight. We call the new problem Weighted Vertex Cover. The first idea that comes to mind to get an approximation algorithm for Weighted Vertex Cover is to generalize *ApproxVertexCover* as follows: instead of selecting an arbitrary edge (v_i, v_j) from E' in line 3, we select the edge of minimum weight (where the weight of an edge is defined as the sum of the weights of its endpoints). Unfortunately this doesn't work: the weight of the resulting cover can be arbitrarily much larger than the weight of an optimal cover. Our new approach will be based on linear programming.

(Integer) linear programming. In a linear-programming problem we are given a linear *cost function* of d real variables $x_1, \dots, x_d$ and a set of n linear *constraints* on these variables. The goal is to assign values to the variables so that the cost function is minimized (or: maximized) and all constraints are satisfied. In other words, the Linear Programming problem can be stated as follows:

$$\begin{array}{ll} \text{Minimize} & c_1x_1 + \cdots + c_dx_d \\ \text{Subject to} & a_{1,1}x_1 + \cdots + a_{1,d}x_d \leqslant b_1 \\ & a_{2,1}x_1 + \cdots + a_{2,d}x_d \leqslant b_2 \\ & \vdots \\ & a_{n,1}x_1 + \cdots + a_{n,d}x_d \leqslant b_n \end{array}$$

There are algorithms—for example the so-called *interior-point methods*—that can solve linear programs in time polynomial in the input size.[1] In practice linear programming is often done

[1] Here the input size is measured in terms of the number of bits needed to describe the input, so this is different from the usual notion of input size.

with with famous *simplex method*, which is exponential in the worst case but works quite well in most practical applications. Hence, if we can formulate a problem as a linear-programming problem then we can solve it efficiently, both in theory and in practice.

There are several problems that can be formulated as a linear-programming problem but with one twist: the variables $x_1, \ldots, x_d$ can not take on real values but only integer values. This is called Integer Linear Programming. (When the variables can only take the values 0 or 1, the problem is called 0/1 Linear Programming.) Unfortunately, Integer Linear Programming and 0/1 Linear Programming are considerably harder than Linear Programming. In fact, Integer Linear Programming and 0/1 Linear Programming are NP-complete. However, formulating a problem as an integer linear program can still be useful, as shall see next.

Approximating via LP relaxation and rounding. Let's go back to Weighted Vertex Cover. Thus we are given a weighted graph $G = (V, E)$ with $V = \{v_1, \ldots, v_n\}$, and we want to find a minimum-weight vertex cover. To formulate this as a 0/1 linear program, we introduce a variable x_i for each vertex $v_i \in V$; the idea is to have $x_i = 1$ if v_i is taken into the vertex cover, and $x_i = 0$ if v_i is not taken into the cover. When is a subset $C \subseteq V$ a vertex cover? Then C must contain at least one endpoint for every edge (v_i, v_j). This means we must have $x_i = 1$ or $x_j = 1$. We can enforce this by introducing for every edge $(v_i, v_j) \in E$ the constraint $x_i + x_j \geqslant 1$. Finally, we wish to minimize the total weight of the cover, so we get as a cost function $\sum_{i=1}^{n} \mathit{weight}(v_i) \cdot x_i$. To summarize, solving the weighted vertex-cover problem corresponds to solving the following 0/1 linear-programming problem.

$$\begin{array}{lll} \text{Minimize} & \sum_{i=1}^{n} \mathit{weight}(v_i) \cdot x_i & \\ \text{Subject to} & x_i + x_j \geqslant 1 & \text{for all edges } (v_i, v_j) \in E \\ & x_i \in \{0,1\} & \text{for } 1 \leqslant i \leqslant n \end{array} \tag{3.1}$$

As noted earlier, solving 0/1 linear programs is hard. Therefore we perform *relaxation*: we drop the restriction that the variables can only take integer values and we replace the integrality constraints (3.1) by

$$0 \leqslant x_i \leqslant 1 \qquad \text{for } 1 \leqslant i \leqslant n \tag{3.2}$$

This linear program can be solved in polynomial time. But what good is a solution where the variables can take on any real number in the interval $[0, 1]$? A solution with $x_i = 1/3$, for instance, would suggest that we put $1/3$ of the vertex v_i into the cover—something that does not make sense. First we note that the solution to our new relaxed linear program provides us with a lower bound.

Lemma 3.3 *Let W denote the value of an optimal solution to the relaxed linear program described above, and let* OPT *denote the minimum weight of a vertex cover. Then* OPT $\geqslant W$.

Proof. Any vertex cover corresponds to a feasible solution of the linear program, by setting the variables of the vertices in the cover to 1 and the other variables to 0. Hence, the optimal solution of the linear program is at least as good as this solution. (Stated differently: we already argued that an optimal solution of the 0/1-version of the linear program corresponds to an optimal solution of the vertex-cover problem. Relaxing the integrality constraints clearly cannot make the solution worse.) □

The next step is to derive a valid vertex cover—or, equivalently, a feasible solution to the 0/1 linear program—from the optimal solution to the relaxed linear program. We want to do this in such a way that the total weight of the solution does not increase by much. This can simply be done by *rounding*: we pick a suitable threshold τ, and then round all variables whose value is at least τ up to 1 and all variables whose value is less than τ down to 0. The rounding should be done in such a way that the constraints are still satisfied. In our algorithm we can take $\tau = 1/2$—see the first paragraph of the proof of Theorem 3.4 below—but in other applications we may need to use a different threshold. We thus obtain the algorithm for Weighted Vertex Cover shown in Algorithm 3.2.

Algorithm 3.2 Approximation algorithm for Weighted Vertex Cover.

```
ApproxWeightedVertexCover(V, E)
1: Solve the relaxed linear program corresponding to the given problem:
       Minimize     $\sum_{i=1}^{n} weight(v_i) \cdot x_i$
       Subject to   $x_i + x_j \geqslant 1$          for all edges $(v_i, v_j) \in E$
                    $0 \leqslant x_i \leqslant 1$          for $1 \leqslant i \leqslant n$
2: $C \leftarrow \{v_i \in V : x_i \geqslant 1/2\}$
3: return $C$
```

Theorem 3.4 *Algorithm* ApproxWeightedVertexCover *is a 2-approximation algorithm.*

Proof. We first argue that the set C returned by the algorithm is a vertex cover. Consider an edge $(v_i, v_j) \in E$. Then $x_i + x_j \geqslant 1$ is one of the constraints of the linear program. Hence, the reported solution to the linear program—note that a solution will be reported, since the program is obviously feasible by setting all variables to 1—has $\max(x_i, x_j) \geqslant 1/2$. It follows that at least one of v_i and v_j will be put into C.

Let $W := \sum_{i=1}^{n} weight(v_i) \cdot x_i$ be the total weight of the optimal solution to the relaxed linear program. By Lemma 3.3 we have $\textsc{opt} \geqslant W$. Using that $x_i \geqslant 1/2$ for all $v_i \in C$, we can now bound the total weight of C as follows:

$$\sum_{v_i \in C} weight(v_i) \leqslant \sum_{v_i \in C} weight(v_i) \cdot 2x_i \leqslant 2 \sum_{v_i \in C} weight(v_i) \cdot x_i \leqslant 2 \sum_{i=1}^{n} weight(v_i) \cdot x_i = 2W \leqslant 2 \cdot \textsc{opt}$$

□

The integrality gap. Note that, as always, the approximation ratio of our algorithm is obtained by comparing the obtained solution to a certain lower bound. Here–and this is essentially always the case when LP relaxation is used–the lower bound is the solution to the relaxed LP. The worst-case ratio between the solution to the integer linear program (which models the problem exactly) and its relaxed version is called the *integrality gap.* For approximation algorithms based on rounding the relaxation of an integer linear program, one cannot prove a better approximation ratio than the integrality gap (assuming that the solution to the relaxed LP is used as the lower bound).

3.3 Set Cover

Let $Z := \{z_1, \ldots, z_m\}$ be a finite set. A *set cover* for Z is a collection of subsets of Z whose union is Z. The SET COVER problem is, given a set Z and a collection $\mathcal{S} = S_1, \ldots, S_n$ of subsets of Z, to select a minimum number of subsets from $\mathcal{S}$ that together form a set cover for Z.

SET COVER is a generalization of VERTEX COVER. This can be seen as follows. Let $G = (V, E)$ be the graph for which we want to obtain a vertex cover. We can construct an instance of SET COVER from G as follows: The set Z is the set of edges of G, and every vertex $v_i \in V$ defines a subset S_i consisting of those edges of which v_i is an endpoint. Then a set cover for the input $Z, S_1, \ldots, S_n$ corresponds to a vertex cover for G. (Note that SET COVER is more general than VERTEX COVER, because in the instance of SET COVER defined by a VERTEX COVER instance, every element occurs in exactly two sets—in the general problem an element from Z can occur in many subsets.) In WEIGHTED SET COVER every subset S_i has a weight $\mathit{weight}(S_i)$ and we want to find a set cover of minimum total weight.

In this section we will develop an approximation algorithm for WEIGHTED SET COVER. To this end we first formulate the problem as a 0/1 linear program: we introduce a variable x_i that indicates whether S_i is in the cover ($x_i = 1$) or not ($x_i = 0$), and we introduce a constraint for each element $z_j \in Z$ that guarantees that z_j will be in at least one of the chosen sets. The constraint for z_j is defined as follows. Let

$$\mathcal{S}(j) := \{i : 1 \leqslant i \leqslant n \text{ and } z_j \in S_i\}.$$

Then one of the chosen sets contains z_j if and only if $\sum_{i \in \mathcal{S}(j)} x_i \geqslant 1$. This leads to the following 0/1 linear program.

$$\begin{array}{lll} \text{Minimize} & \sum_{i=1}^{n} \mathit{weight}(S_i) \cdot x_i & \\ \text{Subject to} & \sum_{i \in \mathcal{S}(j)} x_i \geqslant 1 & \text{for all } 1 \leqslant j \leqslant m \\ & x_i \in \{0,1\} & \text{for } 1 \leqslant i \leqslant n \end{array} \tag{3.3}$$

We relax this 0/1 linear program by replacing the integrality constraints in (3.3) by the following constraints:

$$0 \leqslant x_i \leqslant 1 \qquad \text{for } 1 \leqslant i \leqslant n \tag{3.4}$$

We obtain a linear program that we can solve in polynomial time. As in the case of WEIGHTED VERTEX COVER, the value of an optimal solution to this linear program is a lower bound on the value of an optimal solution to the 0/1 linear program and, hence, a lower bound on the minimum total weight of a set cover for the given instance:

Lemma 3.5 *Let W denote the value of an optimal solution to the relaxed linear program described above, and let* OPT *denote the minimum weight of a set cover. Then* OPT $\geqslant W$.

The next step is to use the solution to the linear program to obtain a solution to the 0/1 linear program (or, in other words, to the set cover problem). Rounding in the same way as for the vertex cover problem—rounding variables that are at least 1/2 to 1, and the other variables to 0—does not work: such a rounding scheme will not give a set cover. Instead we use the following *randomized rounding* strategy:

For each S_i independently, put S_i into the cover C with probability x_i.

Lemma 3.6 *The expected total weight of C is at most* OPT.

Proof. By definition, the total weight of C is the sum of the weights of its subsets. Let's define an indicator random variable Y_i that tells us whether a set S_i is in the cover C:

$$Y_i = \begin{cases} 1 & \text{if } S_i \in C \\ 0 & \text{otherwise} \end{cases}$$

We have

$$\begin{aligned} \mathrm{E}[\text{ weight of } C\ ] &= \mathrm{E}[\ \textstyle\sum_{i=1}^{n} \mathit{weight}(S_i)\cdot Y_i\ ] \\ &= \textstyle\sum_{i=1}^{n} \mathit{weight}(S_i)\cdot \mathrm{E}[Y_i] && \text{(by linearity of expectation)} \\ &= \textstyle\sum_{i=1}^{n} \mathit{weight}(S_i)\cdot \Pr[\ S_i \text{ is put into } C\ ] \\ &= \textstyle\sum_{i=1}^{n} \mathit{weight}(S_i)\cdot x_i \\ &\leqslant \textsc{opt} && \text{(by Lemma 3.5)} \end{aligned}$$

□

So the total weight of C is very good. Is C is valid set cover? To answer this question, let's look at the probability that an element $z_j \in Z$ is not covered. Recall that $\sum_{i\in\mathcal{S}(j)} x_i \geqslant 1$. Suppose that z_j is present in ℓ subsets, that is, $|\mathcal{S}(j)| = \ell$. To simplify the notation, let's renumber the sets such that $\mathcal{S}(j) = \{1,\ldots,\ell\}$. Then we have

$$\Pr[\ z_j \text{ is not covered }] \ =\ (1-x_1)\cdot\dots\cdot(1-x_\ell) \ \leqslant\ \left(1-\frac{1}{\ell}\right)^{\ell},$$

where the last inequality follows from the fact that $(1-x_1)\cdot\dots\cdot(1-x_\ell)$ is maximized when the x_i's sum up to exactly 1 and are evenly distributed, that is, when $x_i = 1/\ell$ for all i. Since $(1-(1/\ell))^\ell \leqslant 1/e$, where $e \approx 2.718$ is the base of the natural logarithm, we conclude that

$$\Pr[\ z_j \text{ is not covered }] \ \leqslant\ \frac{1}{e} \ \approx\ 0.268.$$

So any element z_j is covered with constant probability. But this is not good enough: there are many elements z_j and even though each one of them has a reasonable chance of being covered, we cannot expect all of them to be covered simultaneously. (This is only to be expected, since Weighted Set Cover is NP-complete, so we shouldn't hope to find an optimal solution in polynomial time.) What we need is that each element z_j is covered *with high probability.* To this end we simply repeat the above procedure t times, for a suitable value of t: we generate covers $C_1,\ldots,C_t$ where each C_s is obtained using the randomized rounding strategy, and we take $C^* := C_1 \cup\dots\cup C_t$ as our cover. Our final algorithm is shown in Algorithm 3.3.

Theorem 3.7 *Algorithm* ApproxWeightedSetCover *computes a collection C^* that is a set cover with probability at least $1-1/m$ and whose expected total weight is $O(\textsc{opt}\cdot\log m)$.*

Proof. The expected weight of each C_s is at most OPT, so the expected total weight of C^* is at most $t\cdot\textsc{opt} = O(\textsc{opt}\cdot\log m)$. What is the probability that some fixed element z_j is not

Algorithm 3.3 Approximation algorithm for weighted Set Cover.

```
ApproxWeightedSetCover$(X, \mathcal{S})$
▷ Input: A set $X = \{z_1, \ldots, z_m\}$ of element, and a collection $\mathcal{S} = \{S_1, \ldots, S_n\}$
  of subsets, where each $S_i \in \mathcal{S}$ has weight $weight(S_i)$.
Solve the relaxed linear program corresponding to the given problem:
    Minimize     $\sum_{i=1}^{n} weight(S_i) \cdot x_i$
    Subject to   $\sum_{i \in \mathcal{S}(j)} x_i \geqslant 1$     for all $1 \leqslant j \leqslant m$
                 $0 \leqslant x_i \leqslant 1$            for $1 \leqslant i \leqslant n$
$t \leftarrow 2 \ln m$
for $s \leftarrow 1$ to $t$ do                ▷ Compute $C_s$ by randomized rounding:
    for $i \leftarrow 1$ to $n$ do
        Put $S_i$ into $C_s$ with probability $x_i$
    end for
end for
$C^* \leftarrow C_1 \cup \cdots \cup C_t$
return $C^*$
```

covered by any of the covers C_s? Since the covers C_s are generated independently, and each C_s fails to cover z_j with probability at most $1/e$, we have

$$\Pr[\ z_j \text{ is not covered by any } C_s\ ] \leqslant (1/e)^t.$$

Since $t = 2 \ln m$ we conclude that z_j is not covered with probability at most $1/m^2$. Hence,

$$\begin{aligned}\Pr[\ \text{all elements } z_j \text{ are covered by } C^*\ ] &= 1 - \Pr[\ \text{at least one element } z_j \text{ is not covered by } C^*\ ] \\ &\leqslant 1 - \sum_{j=1}^{m} \Pr[\ z_j \text{ is not covered by } C^*\ ] \\ &\leqslant 1 - 1/m\end{aligned}$$

□

3.4 Exercises

Exercise 3.1 Consider the following greedy algorithm for unweighted Vertex Cover:

```
GreedyDegreeVertexCover$(V, E)$
$C \leftarrow \emptyset$; $E' \leftarrow E$
while $E' \neq \emptyset$ do
    Let $v \in V$ be a vertex that covers the largest number of edges in $E'$ (that
    is, a vertex that is an endpoint of the largest number of uncovered edges.)
    $C \leftarrow C \cup \{v\}$
    Remove all edges from $E'$ that have $v$ as an endpoint.
end while
return $C$
```

Prove or disprove: *GreedyDegreeVertexCover* is a 2-approximation algorithm.

Exercise 3.2 Explicitly write the 0/1-LP corresponding to the graph in Fig. 3.1.

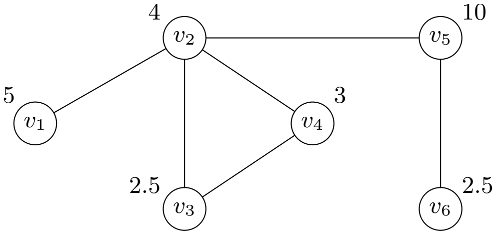

Fig. 3.1: A weighted graph on six vertices. The weights of each vertex is written next to it.

Exercise 3.3 Let $X = \{x_1, \ldots, x_n\}$ be a set of n boolean variables. A boolean formula over the set X is a CNF formula—in other words, is in *conjunctive normal form*—if it has the form $C_1 \wedge C_2 \wedge \cdots \wedge C_m$, where each clause C_j is the disjunction of a number of literals. In this exercise we consider CNF-formulas where every clause has exactly three literals, and there are no negated literals. An example of such a formula is

$$(x_1 \vee x_3 \vee x_4) \wedge (x_2 \vee x_3 \vee x_7) \wedge (x_1 \vee x_5 \vee x_6).$$

Such CNF formulas are obviously satisfiable: setting all variables to TRUE clearly makes every clause TRUE. Our goal is to make the CNF formula TRUE by setting the smallest number of variables to TRUE.

(i) Consider the following algorithm for this problem.

```
Greedy-CNF($\mathcal{C}, X$)
▷ $\mathcal{C} = \{C_1, \ldots, C_m\}$ is a set of clauses, $X = \{x_1, \ldots, x_n\}$ a set of variables.
while $\mathcal{C} \neq \emptyset$ do
    Take an arbitrary clause $C_j \in \mathcal{C}$.
    Let $x_i$ be one of the variables in $C_j$.
    Set $x_i \leftarrow$ TRUE.
    Remove all clauses from $\mathcal{C}$ that contain $x_i$.
end while
return $X$
```

Analyze the approximation ratio of *Greedy-CNF* as a function of n.

(ii) Modify the algorithm such that it becomes a 3-approximation algorithm for the problem, and prove that your algorithm achieves the desired approximation ratio.

(iii) Now give a different 3-approximation algorithm for the problem, based on the technique of LP relaxation.

Exercise 3.4 Show that for any $n > 1$ there is an input graph G with n vertices such that the integrality gap of the LP in *ApproxWeightedVertexCover* for G is (at least) $2 - \frac{2}{n}$. *Hint:* Use a graph where all vertex weights are 1.

Exercise 3.5 Let $d \geqslant 2$ be an integer, and let V be a set of elements called *vertices*. We call a subset $e \subset V$ of size d a *d-edge* on V. A *d-hypergraph* is a pair $G = (V, E)$ where E is a set of d-edges on V. Note that a 2-hypergraph is just a normal (undirected) graph.

A *double vertex cover* of a d-hypergraph $G = (V, E)$ is a subset $C \subset V$ such that for every d-edge $e \in E$ there are two vertices $u, v \in C$ such that $u \in e$ and $v \in e$. We want to compute for a given d-hypergraph $G = (V, E)$ a minimum-size double vertex cover.

(i) Formulate the problem as a 0/1 linear program, and briefly explain your formulation.

(ii) Give a polynomial-time approximation algorithm for this problem, based on the technique of LP rounding. Prove that your algorithm returns a valid solution (that is, a double vertex cover) and prove a bound on its approximation ratio.

(iii) Now consider your LP for the case $d = 3$. Recall that the integrality gap for an LP is the worst-case ratio between the value of an optimal fractional solution and the value of an integral solution. Show that the integrality gap for your LP when $d = 3$ is at least c, for some constant $c > 1$. Try to make c as large as possible.

(iv) Consider the case of arbitrary d again. Give an approximation algorithm that has approximation ratio $O(\log n)$, with high probability.

Exercise 3.6 An electricity company has to decide how to connect the houses in a new neighborhood to the electricity network. Connecting a house to the network is done via a *distribution unit*. There are several possible locations where distribution units can be placed. Thus the problem faced by the company is to decide which of the potential distribution units to actually build, and then through which of these units each house will be served. An additional difficulty is that for each house only some of the distribution units are suitable.

This problem can be modeled as follows. We have a set $U = \{u_1, \ldots, u_n\}$ of potential distribution units, each of which has a cost f_i associated to it; the cost f_i must be paid if the company decides to build unit u_i. Moreover, we have a set $H = \{h_1, \ldots, h_m\}$ of houses that need to be connected to the network. Each house has a set $U(h_j) \subseteq U$ of suitable distribution units, and for each $u_i \in U(h_j)$ there is a cost $g_{i,j}$ that must be paid if the company decides to connect house h_j to unit u_i. The goal of the company is to minimize its total cost, which is the cost of building distribution units plus the cost of connecting each house to one of the distribution units.

(i) Formulate the problem as a 0/1 linear program, and briefly explain your formulation.

(ii) Assume that $|U(h_j)| \leqslant 4$ for all h_j. Give a polynomial-time approximation algorithm for this case, based on the technique of LP rounding. Prove that your algorithm returns a valid solution and prove a bound on its approximation ratio.

Exercise 3.7 Let $G = (V, E)$ be an undirected edge-weighted graph, where the weight of an edge (u, v) is denoted by *weight*(u, v). A *matching* in G is a collection $M \subset E$ of edges such that each vertex $v \in V$ is incident to at most one edge in M. We want to compute a matching in G of maximum total weight. We can model this as a 0/1-LP, as follows. We introduce a variable x_{uv} for every edge $(u, v) \in E$, where setting $x_{uv} := 1$ indicates that we put (u, v) in the matching, and setting $x_{uv} := 0$ indicates that we do not put (u, v) in the matching. Let

$N(u)$ be the set of neighbors of a node $u \in V$, that is, $N(u) := \{v \in V : (u,v) \in E\}$. Then we can model the maximum-matching problem as a 0/1-LP:

$$\begin{array}{lll} \text{Maximize} & \sum_{(u,v)\in E} \mathit{weight}(u,v) \cdot x_{uv} & \\ \text{Subject to} & \sum_{v \in N(u)} x_{uv} \leqslant 1 & \text{for all } u \in V \\ & x_{uv} \in \{0,1\} & \text{for all } (u,v) \in E \end{array}$$

Someone suggests to derive an approximation algorithm from this using the technique of LP-relaxation, as follows.

```
MaxMatching($V, E$)
    Solve the relaxed linear program corresponding to the given problem:
        Maximize     $\sum_{(u,v) \in E} weight(u,v) \cdot x_{uv}$
        Subject to   $\sum_{v \in N(u)} x_{uv} \leqslant 1$          for all $u \in V$
                     $0 \leqslant x_{uv} \leqslant 1$                for all $(u,v) \in E$
    $M \leftarrow \{(u,v) \in E : x_{uv} > 1/2\}$
    return $C$
```

(i) Prove or disprove: the algorithm returns a valid matching.

(ii) Prove or disprove: the algorithm gives a (1/2)-approximation.

Chapter 4

Polynomial-time approximation schemes

When faced with an NP-hard problem one cannot expect to find a polynomial-time algorithm that always gives an optimal solution. Hence, one has to settle for an approximate solution. Of course one would prefer that the approximate solution is very close optimal, for example at most 5% worse. In other words, one would like to have an approximation ratio very close to 1. The approximation algorithms we have seen so far do not quite achieve this: for Load Balancing we gave an algorithm with approximation ratio 3/2, for Weighted Vertex Cover we gave an algorithm with approximation ratio 2, and for Weighted Set Cover the approximation ratio was even $O(\log n)$. Unfortunately it is not always possible to get a better approximation ratio: for some problems one can prove that it is not only np-hard to solve the problem exactly, but that there is a constant $c > 1$ such that there is no polynomial-time c-approximation algorithm unless p=np. Vertex Cover, for instance, cannot be approximated to within a factor 1.3606... unless p=np, and for Set Cover one cannot obtain a better approximation factor than $\Theta(\log n)$.

Fortunately there are also problems where much better solutions are possible. In particular, some problems admit a so-called *polynomial-time approximation scheme*, or *PTAS* for short. Such an algorithm works as follows. Its input is, of course, an instance of the problem at hand, but in addition there is an input parameter $\varepsilon > 0$. The output of the algorithm is then a solution whose value is at most $(1 + \varepsilon) \cdot \textsc{opt}$ for a minimization problem, or at least $(1 - \varepsilon) \cdot \textsc{opt}$ for a maximization problem. The running time of the algorithm should be polynomial in n; its dependency on ε can be exponential however. So the running time can be $O(2^{1/\varepsilon} n^2)$ for example, or $O(n^{1/\varepsilon})$, or $O(n^2/\varepsilon)$, etc. If the dependency on the parameter $1/\varepsilon$ is also polynomial then we speak of a *fully polynomial-time approximation scheme (FPTAS)*. In this lecture we give an example of an FPTAS.

4.1 A dynamic-programming algorithm for Knapsack with integer profits

The Knapsack problem is defined as follows. We are given a set $X = \{x_1, \dots, x_n\}$ of n items that each have a (positive) *weight* and a (positive) *profit*. The weight and profit of x_i are denoted by $\mathit{weight}(x_i)$ and $\mathit{profit}(x_i)$, respectively. Moreover, we have a knapsack that can carry items of total weight W. For a subset $S \subset X$, define $\mathit{weight}(S) := \sum_{x \in S} \mathit{weight}(x)$

and $\mathit{profit}(S) := \sum_{x \in S} \mathit{profit}(x)$. The goal is now to select a subset of the items whose profit is maximized, under the condition that the total weight of the selected items is at most W. From now on, we will assume that $\mathit{weight}(x_i) \leqslant W$ for all i. (Items with $\mathit{weight}(x_i) > W$ can of course simply be ignored.)

We will first develop an algorithm for the case where all the profits are integers. Let $P := \mathit{profit}(X)$, that is, P is the total profit of all items. The running time of our algorithm will depend on n and P. Since P can be arbitrarily large, the running time of our algorithm will not necessarily be polynomial in n. In the next section we will then show how to obtain an FPTAS for Knapsack that uses this algorithm as a subroutine.

Our algorithm for the case where all profits are integers is a dynamic-programming algorithm. For $1 \leqslant i \leqslant n$ and $0 \leqslant p \leqslant P$, define

$$A[i,p] = \min\{\mathit{weight}(S) : S \subset \{x_1, \ldots, x_i\} \text{ and } \mathit{profit}(S) = p\}.$$

In other words, $A[i,p]$ denotes the minimum possible weight of any subset S of the first i items such that $\mathit{profit}(S)$ is exactly p. When there is no subset $S \subset \{x_1, \ldots, x_i\}$ of profit exactly p then we define $A[i,p] = \infty$. Note that Knapsack asks for a subset of weight at most W with the maximum profit. This maximum profit is given by $\textsc{opt} := \max\{p : 0 \leqslant p \leqslant P \text{ and } A[n,p] \leqslant W\}$. This means that if we can compute all values $A[i,p]$ then we can compute $\textsc{opt}$. From the table A we can then also compute a subset S such that $\mathit{profit}(S) = \textsc{opt}$—see below for details. As is usual in dynamic programming, the values $A[i,p]$ are computed bottom-up by filling in a table. It will be convenient to extend the definition of $A[i,p]$ to include the case $i = 0$, as follows: $A[0,0] = 0$ and $A[0,p] = \infty$ for $p > 0$. Now we can give a recursive formula for $A[i,p]$.

Lemma 4.1

$$A[i,p] \;=\; \begin{cases} 0 & \text{if } p = 0 \\ \infty & \text{if } i = 0 \text{ and } p > 0 \\ A[i-1,p] & \text{if } i > 0 \text{ and } 0 < p < \mathit{profit}(x_i) \\ \min(A[i-1,p], A[i-1,p-\mathit{profit}(x_i)] + \mathit{weight}(x_i)) & \text{if } i > 0 \text{ and } p \geqslant \mathit{profit}(x_i) \end{cases}$$

Proof. The first two cases are simply by definition. Now consider third and fourth case. Obviously the minimum weight of any subset of $\{x_1, \ldots, x_i\}$ of total profit p is given by one of the following two possibilities:

- the minimum weight of any subset $S \subset \{x_1, \ldots, x_i\}$ with profit p and $x_i \in S$, or
- the minimum weight of any subset $S \subset \{x_1, \ldots, x_i\}$ with profit p and $x_i \notin S$.

In the former case, $\mathit{weight}(S)$ is equal to $\mathit{weight}(x_i)$ plus the minimum weight of any subset $S \subset \{x_1, \ldots, x_{i-1}\}$ with profit $p - \mathit{profit}(x_i)$, which is given by $A[i-1, p-\mathit{profit}(x_i)]$. (This is also correct when $A[i-1, p-\mathit{profit}(x_i)] = \infty$. In that case there is no subset $S \subset \{x_1, \ldots, x_{i-1}\}$ of profit $p - \mathit{profit}(x_i)$, so there is no subset $S \subset \{x_1, \ldots, x_i\}$ of profit p that includes x_i.) In the latter case, $\mathit{weight}(S)$ is equal to the minimum weight of any subset $S \subset \{x_1, \ldots, x_{i-1}\}$ with profit p, which is $A[i-1,p]$. (Again, this is also correct when $A[i-1,p] = \infty$.) When $p < \mathit{profit}(x_i)$ the former possibility does not apply, which proves the lemma for the third case. Otherwise we have to take the best of the two possibilities, proving fourth case. □

Based on this lemma, we can immediately give a dynamic-programming algorithm.

Algorithm 4.1 Dynamic-programming algorithm for Knapsack with integer profits.

```
*IntegerWeightKnapsack*$(X, W)$
 1: Let $A[0..n, 0..P]$ be an array, where $P = \sum_{i=1}^{n} \mathit{profit}(x_i)$.
 2: for $i \leftarrow 0$ to $n$ do
 3:     $A[i, 0] \leftarrow 0$
 4: end for
 5: for $p \leftarrow 1$ to $P$ do
 6:     $A[0, p] \leftarrow \infty$
 7: end for
 8: for $i \leftarrow 1$ to $n$ do
 9:     for $p \leftarrow 1$ to $P$ do
10:         if $\mathit{profit}(x_i) \leqslant p$ then
11:             $A[i, p] \leftarrow \min(A[i-1, p], \mathit{weight}(x_i) + A[i-1, p - \mathit{profit}(x_i)])$
12:         else
13:             $A[i, p] \leftarrow A[i-1, p]$
14:         end if
15:     end for
16: end for
17: $\textsc{opt} \leftarrow \max\{p : 0 \leqslant p \leqslant P \text{ and } A[n, p] \leqslant W\}$
18: Using $A$, find a subset $S \subseteq X$ of profit $\textsc{opt}$ and total weight at most $W$.
19: return $S$
```

Finding an optimal subset S in line 18 of the algorithm can be done by “walking back” in the table A, as is standard in dynamic-programming algorithms—see also the chapter on dynamic programming from [CLRS]. For completeness, we describe a subroutine *ReportSolution* that finds an optimal subset.

```
*ReportSolution*$(X, A, \textsc{opt})$
$p \leftarrow \textsc{opt}$; $S \leftarrow \emptyset$
for $i \leftarrow n$ downto 1 do
    if $\mathit{profit}(x_i) \leqslant p$ then
        if $\mathit{weight}(x_i) + A[i-1, p - \mathit{profit}(x_i)] < A[i-1, p]$ then
            $S \leftarrow S \cup \{x_i\}$; $p \leftarrow p - \mathit{profit}(x_i)$
        end if
    end if
end for
return $S$
```

It is easy to see that *IntegerWeightKnapsack*, including the subroutine *ReportSolution*, runs in $O(nP)$ time. We get the following theorem.

Theorem 4.2 *Suppose all profits in a* Knapsack *instance are integers. Then the problem can be solved in $O(nP)$ time, where $P := \mathit{profit}(X)$ is the total profit of all items.*

4.2 An FPTAS for Knapsack

To use the result above to obtain an FPTAS for the general case, where the profits can be arbitrarily large and need not even be integers, we apply the following strategy: we replace each *profit*(x_i) by a value *profit*$^*(x_i)$, and then compute an optimal subset for these new profit values using the algorithm from the previous section. To make this work, the values *profit** should satisfy the following three conditions:

(i) each *profit*$^*(x_i)$ should be an integer, so that the algorithm from the previous section can be applied,

(ii) the sum $\sum_{i=1}^{n}$ *profit*$^*(x_i)$ should be sufficiently small—polynomial in n, to be precise—so that algorithm from the previous section will run in polynomial time, and

(iii) each *profit*$^*(x_i)$ should be sufficiently close to *profit*(x_i) so that the error we make by working with *profit** (instead of working with *profit*) is under control.

Next we show how to achieve this. Recall that each *profit*(x_i) is a positive real number. We partition $\mathbb{R}^+$ into intervals of length Δ, where Δ is a parameter that we will pick later. Thus we obtain a collection of intervals $(0, \Delta]$, $(\Delta, 2\Delta]$, $(2\Delta, 3\Delta]$, etc. We then replace each *profit*(x_i) by the value j such that *profit*(x_i) lies in the j-th interval, $((j-1)\Delta, j\Delta]$. In other words, we define *profit*$^*(x_i)$ as follows:

$$\mathit{profit}^*(x_i) := \left\lceil \frac{\mathit{profit}(x_i)}{\Delta} \right\rceil. \tag{4.1}$$

A different way to look at this is the following. We first "round" every profit to the right endpoint of the interval in which it lies; see Fig. 4.1. After the rounding, every profit is an integer multiple of Δ. When then divide all profits by Δ to obtain integral values, thus satisfying condition (i) above. To make sure we also satisfy conditions (ii) and (iii) we need to pick the value of Δ in the right way, as explained next.

Notice that by rounding a profit to the right endpoint of the interval it is contained in, we change the profit by less than Δ. We then scale the whole problem by dividing every profit by the same amount. Intuitively, this scaling should not influence the relative quality of the solution we compute. We will therefore pick Δ based on the intuition that the error we induce on each individual profit is less than Δ. To obtain a PTAS we must compute a solution whose total profit is at least $(1-\varepsilon) \cdot$ OPT. In other words, the total error of the solution should be at most $\varepsilon \cdot$ OPT. If the error in an individual profit is less than Δ then, since a solution consists of at most n items, the total error of the solution is at most $n\Delta$. Thus we want to choose Δ such that $n\Delta = \varepsilon \cdot$ OPT. This suggests to pick

$$\Delta := (\varepsilon/n) \cdot \text{OPT}. \tag{4.2}$$

However, there is a problem: we do not know OPT, so our algorithm cannot set Δ according to (4.2). To overcome this problem, we use a suitable lower bound LB instead of OPT. Clearly

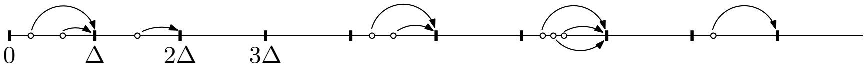

Fig. 4.1: Rounding the profits: each profit (indicated by the small circles) is rounded to the right endpoint of the interval $((j-1)\Delta, j\Delta]$ in which it lies.

(since we assumed each item has weight at most W) we have $\textsc{opt} \geqslant \max_{1 \leqslant i \leqslant n} \mathit{profit}(x_i)$. Thus we set

$$\Delta := (\varepsilon/n) \cdot \textsc{lb}, \tag{4.3}$$

where $\textsc{lb} := \max_{1 \leqslant i \leqslant n} \mathit{profit}(x_i)$. Note that by working with $\textsc{lb}$ instead of $\textsc{opt}$, the interval size Δ can only become smaller. Hence, the error on the individual profits does not increase and so intuitively condition (iii) is satisfied.

What about condition (ii)? For every item x_i we have

$$\mathit{profit}^*(x_i) \leqslant \left\lceil \frac{\max_{1 \leqslant i \leqslant n} \mathit{profit}(x_i)}{\Delta} \right\rceil = \left\lceil \frac{\max_{1 \leqslant i \leqslant n} \mathit{profit}(x_i)}{(\varepsilon/n) \cdot \textsc{lb}} \right\rceil = \lceil n/\varepsilon \rceil.$$

It follows that $\sum_{i=1}^{n} \mathit{profit}^*(x_i) = O(n^2/\varepsilon)$, and condition (ii) is satisfied. Before we formally prove that our algorithm is an FPTAS, we summarize the algorithm in pseudocode.

Algorithm 4.2 PTAS for Knapsack.

```
Knapsack-FPTAS$(X, W, \varepsilon)$
Set $\textsc{lb} \leftarrow \max_{1 \leqslant i \leqslant n} \mathit{profit}(x_i)$ and $\Delta \leftarrow (\varepsilon/n) \cdot \textsc{lb}$.
For all $1 \leqslant i \leqslant n$, let $\mathit{profit}^*(x_i) \leftarrow \left\lceil \frac{\mathit{profit}(x_i)}{\Delta} \right\rceil$.
Compute a subset $S^* \subset X$ of maximum profit and total weight at most $W$ with
    algorithm IntegerWeightKnapsack, using the new profits $\mathit{profit}^*(x_i)$ instead of
    $\mathit{profit}(x_i)$.
return $S^*$
```

Theorem 4.3 Knapsack-FPTAS *computes in $O(n^3/\varepsilon)$ time a subset $S^* \subset X$ of weight at most W whose profit is at least $(1-\varepsilon) \cdot \textsc{opt}$, where $\textsc{opt}$ is the maximum profit of any subset of weight at most W.*

Proof. To prove the running time, recall that $\mathit{profit}^*(x_i) \leqslant \lceil n/\varepsilon \rceil$ for all $1 \leqslant i \leqslant n$. Hence, $\mathit{profit}^*(X)$, the total new profit, is at most $n \cdot \lceil n/\varepsilon \rceil$. Hence, by Theorem 4.2 the algorithm runs in $O(n^3/\varepsilon)$ time.

Above we already argued intuitively that the total error is at most $\varepsilon \cdot \textsc{opt}$. Next we formally prove that the value of the solution we compute is indeed at least $(1-\varepsilon) \cdot \textsc{opt}$. To this end, let S_{opt} denote an optimal subset, that is, a subset of weight at most W such that $\mathit{profit}(S_{\text{opt}}) = \textsc{opt}$. Let S^* denote the subset returned by the algorithm. Since we did not change the weights of the items, the subset S^* has weight at most W. Hence, the computed solution S^* is feasible. It remains to show that $\mathit{profit}(S^*) \geqslant (1-\varepsilon) \cdot \textsc{opt}$.

Because S^* is optimal for the new profits, we have $\mathit{profit}^*(S^*) \geqslant \mathit{profit}^*(S_{\text{opt}})$. Moreover

$$\frac{\mathit{profit}(x_i)}{\Delta} \leqslant \mathit{profit}^*(x_i) \leqslant \frac{\mathit{profit}(x_i)}{\Delta} + 1,$$

where $\Delta = (\varepsilon/n) \cdot \textsc{lb}$. Hence, we have

$$\begin{array}{rcl} \mathit{profit}(S^*) & = & \sum_{x_i \in S^*} \mathit{profit}(x_i) \\ & \geqslant & \sum_{x_i \in S^*} \Delta \cdot (\mathit{profit}^*(x_i) - 1) \\ & = & \Delta \cdot \sum_{x_i \in S^*} \mathit{profit}^*(x_i) - |S^*| \cdot \Delta \\ & \geqslant & \Delta \cdot \sum_{x_i \in S^*} \mathit{profit}^*(x_i) - n \cdot \Delta \\ & \geqslant & \Delta \cdot \sum_{x_i \in S_{\mathrm{opt}}} \mathit{profit}^*(x_i) - n \cdot \Delta \\ & \geqslant & \sum_{x_i \in S_{\mathrm{opt}}} \mathit{profit}(x_i) - n \cdot \Delta \\ & \geqslant & \textsc{opt} - \varepsilon \cdot \textsc{lb} \\ & \geqslant & \textsc{opt} - \varepsilon \cdot \textsc{opt} \end{array}$$

Thus $\mathit{profit}(S^*) \geqslant (1-\varepsilon) \cdot \textsc{opt}$, as claimed. □

4.3 Exercises

Exercise 4.1 Consider the algorithm *Knapsack-FPTAS* described above. Suppose that in step 2 of the algorithm we round the profits down instead of up, that is, we use

$$\mathit{profit}^*(x_i) := \left\lfloor \frac{\mathit{profit}(x_i)}{\Delta} \right\rfloor .$$

Prove or disprove: Theorem 4.3 is still true for this modified version of *Knapsack-FPTAS*.

Exercise 4.2 Consider the following problem. We are given a number $W > 0$ and a set X of n weighted items, where the weight of the i-th item is denoted by w_i. The goal is to find a subset $S \subseteq X$ with the largest possible weight under the condition that the weight of the subset is at most W. Assume that $0 < w_i \leqslant W$ for all $1 \leqslant i \leqslant n$, and that $\sum_{i=1}^{n} w_i > W$.

Suppose that there is an algorithm *Largest-Weight-Subset-Integer*(X, W) that finds an optimal solution when all the weights are integers. We now want to develop an algorithm that computes an optimal solution when the weights are real numbers (in the range $(0 : W]$). Since this problem is hard, we are interested in approximations. More precisely, we want to find a subset of weight at least $(1-\varepsilon) \cdot \textsc{opt}$ that is feasible, that is, has weight at most W; here $\textsc{opt}$ denotes the weight of an optimal solution and ε is a given constant with $0 < \varepsilon < 1$.

(i) Prove that $\textsc{opt} > W/2$.

(ii) Someone suggests the following algorithm for this problem:

```
Largest-Weight-Subset(X, W, ε)
1: LB ← $W/2$
2: For all $1 \leqslant i \leqslant n$ let $w_i^* \leftarrow \left\lceil \frac{w_i}{(\varepsilon/n) \cdot \mathrm{LB}} \right\rceil$, and let $W^* \leftarrow \left\lceil \frac{W}{(\varepsilon/n) \cdot \mathrm{LB}} \right\rceil$.
3: Let $X^*$ be the set of items with the new weights $w_i^*$.
4: $S \leftarrow$ Largest-Weight-Subset-Integer$(X^*, W^*)$.
5: return $S$
```

Prove or disprove: this algorithm gives a feasible solution of weight at least $(1-\varepsilon) \cdot \textsc{opt}$.

(iii) Someone else suggests to modify the algorithm and round the weights down instead of up. Thus step 2 becomes:

2: For all $1 \leqslant i \leqslant n$ let $w_i^* \leftarrow \left\lfloor \frac{w_i}{(\varepsilon/n)\cdot \mathrm{LB}} \right\rfloor$, and let $W^* \leftarrow \left\lfloor \frac{W}{(\varepsilon/n)\cdot \mathrm{LB}} \right\rfloor$.

Prove or disprove: this algorithm gives a feasible solution of weight at least $(1-\varepsilon)\cdot\textsc{opt}$.

Exercise 4.3 Let $\mathcal{G} = (V, E)$ be a connected, undirected, edge-weighted graph. We denote the weight of an edge $e \in E$ by $w(e)$. Assume all edge weights are positive. An *edge cover* for $\mathcal{G}$ is a subset $C \subseteq E$ of the edges such that each vertex $v \in V$ is incident to at least one edge in C. We want to find an edge cover for $\mathcal{G}$ whose total weight is minimized.

(i) Assume each vertex in $\mathcal{G}$ is incident to at most three edges. Give a 3-approximation algorithm for this case, and prove that you algorithm produces a correct cover and that it achieves the required approximation ratio.
Hint: Use the technique of LP-relaxation from the previous chapter.
NB: In the unweighted version of the edge-cover problem and assuming that every vertex is incident to at most three edges, we can simply put all edges into the cover to obtain a 3-approximation.

(ii) Now consider the general case, where we want to solve the weighted version of the problem and there is no restriction on the degrees of the vertices.

Suppose we have an algorithm *IntegerWeightEdgeCover*($\mathcal{G}$) that solves the edge-cover problem optimally if all the weights are positive integers. The running time of *IntegerWeightEdgeCover*($\mathcal{G}$) is $O(2^{w_{\max}}(|V| + |E|))$, where $w_{\max}$ is the maximum weight of an edge in $\mathcal{G}$. Give a PTAS for the case where the weights are real numbers in the range $[0.5, 5]$. Prove that your algorithm achieves the required approximation ratio and analyze its running time.

Exercise 4.4 Consider the Load Balancing problem on two machines. Thus we want to distribute a set of n jobs with processing times $t_1, \ldots, t_n$ over two machines such that the makespan (the maximum of the processing times of the two machines) is minimized. In this exercise we will develop a PTAS for this problem.

Let $T = \sum_{j=1}^{n} t_j$ be the total size of all jobs. We call a job *large* (for a given $\varepsilon > 0$) if its processing time is at least $\varepsilon \cdot T$, and we call it *small* otherwise.

(i) How many large jobs are there at most, and what is the number of ways in which the large jobs can be distributed over the two machines?

(ii) Give a PTAS for the Load Balancing problem for two machines. Prove that your algorithm achieves the required approximation ratio and analyze its running time.

Exercise 4.5 The TSP problem on a set P of points in the plane is to compute a shortest tour visiting all the points in P, that is, a tour whose (Euclidean) length is minimized. Suppose we have an algorithm *IntegerTSP*(P) that, given a set P of n points in the plane with integer coordinates in the range $0, \ldots, m$, computes a shortest tour in $O(nm)$ time. Consider the following PTAS for the general TSP problem, that is, for the case where the coordinates need not be integral and we do not have a pre-specified range in which the coordinates lie. We assume that $\min_{p\in P} p_x = \min_{p\in P} p_y = 0$, where p_x and p_y denote the x- and y-coordinate of the point p.

```
PTAS-TSP$(P, \varepsilon)$
1: $\Delta \leftarrow \dots$
2: For each point $p \in P$ define $p^* = (p^*_x, p^*_y)$, where $p^*_x = \lceil p_x/\Delta \rceil$ and $p^*_y = \lceil p_y/\Delta \rceil$. Let $P^* := \{p^* : p \in P\}$.
3: Compute a shortest tour on $P^*$ using the algorithm IntegerTSP$(P^*)$, and return the reported tour (with each point $p^* \in P^*$ replaced with its corresponding point $p \in P$.
```

(i) Derive a suitable value to be used for Δ in Step 1, so that the resulting algorithm is a PTAS. (Note: In this part of the exercise you don't have to prove that the algorithm is a PTAS.)

(ii) For a tour T on P, define *length*(T) to be the Euclidean length of T. Moreover, define *length*$^*(T)$ to be the length of T if each point $p \in P$ is replaced by p^*. Let T^* be the tour computed by *PTAS-TSP* and let T_{opt} be an optimal tour for the set P. Prove that *length*$(T^*) \leqslant (1+\varepsilon) \cdot$ *length*(T_{opt}) for your choice of Δ, using a proof similar to the proof of Theorem 3.3 in the Course Notes.

(iii) Analyze the running time of the algorithm for your choice of Δ.

Exercise 4.6 Let $G = (V, E)$ be a graph. An *independent set* of G is a subset $W \subseteq V$ such that no two nodes in W are adjacent. In other words, for any two nodes $v, w \in W$ we have $(v, w) \notin E$. A *maximum independent set* is an independent set of maximum size. Maximum Independent Set, the problem of finding a maximum independent set of a given graph G, is NP-hard, so there is no polynomial-time algorithm that solves the problem optimally unless P=NP.

(i) Prove that this implies that there is no FPTAS for Maximum Independent Set unless P=NP. *Hint:* Assume Alg(G, ε) is an FPTAS that computes a $(1-\varepsilon)$-approximation for Maximum Independent Set on a graph G. Now give an algorithm that solves Maximum Independent Set exactly by picking a suitable ε and using Alg(G, ε) as a subroutine. Argue that your choice of ε leads to an exact solution and argue that the resulting algorithm runs in polynomial time to derive a contradiction to the existence of an FPTAS.

(ii) Does your proof also imply that there is no PTAS for Maximum Independent Set unless P=NP? Explain your answer.

Exercise 4.7 Vertex Cover is NP-complete, so there is no polynomial-time algorithm that solves Vertex Cover optimally unless P=NP. Prove that this implies that there is no FPTAS for Vertex Cover unless P=NP. (You are not allowed to use the fact mentioned in the Course Notes that Vertex Cover cannot be approximated to within a factor 1.3606 unless P=NP; your proof that an FPTAS does not exist should only be based on the fact that Vertex Cover is NP-complete.)

Part II

I/O-EFFICIENT ALGORITHMS

Chapter 5

Introduction to I/O-Efficient Algorithms

Using data from satellites or techniques such as LIDAR (light detection and ranging) it is now possible to generate highly accurate digital elevation models of the earth's surface. The simplest and most popular digital elevation model (DEM) is a grid of square cells, where we store for each grid cell the elevation of the center of the cell. In other words, the digital elevation model is simply a 2-dimensional array A, where each entry $A[i, j]$ stores the elevation of the corresponding grid cell. As mentioned, DEMs are highly accurate nowadays, and resolutions of 1 m or less are not uncommon. This gives massive data sets. As an example, consider a DEM representation an area of 100 km × 100 km at 1 m resolution. This gives an array $A[0..m-1, 0..m-1]$ where $m = 100,000$. If we use 8 bytes per grid cell to store its elevation, the array needs about 80GB.

Now suppose we wish to perform a simple task, such as computing the average elevation in the terrain. This is of course trivial to do: go over the array A row by row to compute the sum[1] of all entries, and divide the result by m^2 (the total number of entries).

```
ComputeAverage-RowByRow(A)
▷ $A$ is an $m \times m$ array
$s \leftarrow 0$
for $i \leftarrow 0$ to $m-1$ do
    for $j \leftarrow 0$ to $m-1$ do
        $s \leftarrow s + A[i, j]$
    end for
end for
return $s/m^2$
```

Alternatively we could go over the array column by column, by exchanging the two **for**-loops. Doesn't matter, right? Wrong. You may well find out that one of the two algorithms is much slower than the other. How is this possible? After all, both algorithms run in $O(n)$ time, where $n := m^2$ denotes the total size of the array, and seem equivalent. The problem is that the array A we are working on does not fit into the internal memory. As a result, the actual running time is mainly determined by the time needed to fetch the data from the hard disk,

[1] Actually, things may not be as easy as they seem because adding up a very large number of values may lead to precision problems. We will ignore this, as it is not an I/O-issue.

not by the time needed for CPU computations. In other words, the time is determined by the number of I/O-operations the algorithm performs. And as we shall see, a small difference such as reading a 2-dimensional array row by row or column by column can have a big impact on the number of I/O-operations. Thus it is important to make algorithms I/O-*efficient* when the data on which they operate does not fit into internal memory. In this part of the course we will study the theoretical basics of I/O-efficient algorithms.

5.1 The I/O-model

To be able to analyze the I/O-behavior of our algorithms we need an abstract model of the memory of our computer. In the model we assume our computer is equipped with two types of memory: an *internal memory* (the main memory) of size M and a *external memory* (the disk) of unlimited size. Unless stated otherwise, the memory size M refers to the number of basic elements—numbers, pointers, etcetera—that can be stored in internal memory. We are interested in scenarios where the input does not fit into internal memory, that is, where the input size is greater than M. We assume that initially the input resides entirely in external memory.

An algorithm can only perform an operation on a data element—reading a value stored in a variable, changing the value of a variable, following a pointer, etc.—when the data element is available in internal memory. If this is not the case, the data should first be *fetched* from external memory. For example, in line 4 of *ComputeAverage-RowByRow* the array element $A[i,j]$ may have to be fetched from external memory before s can be updated. (In fact, in theory the variables s, i, j may have to be fetched from external memory as well, although any reasonable operating system would ensure that these are kept in internal memory.)

I/O-operations (fetching data from disk, or writing data to disk) are slow compared to CPU-operations (additions, comparisons, assignments, and so on). More precisely, they are very, very slow: a single I/O-operation can be 100,000 times or more slower than a single CPU-operation. This is what makes I/O-behavior often the bottleneck for algorithms working on data stored on disk. The main reason that reading to (or writing from) disk is so slow, is that we first have to wait until the read/write head is positioned at the location on the disk where the data element is stored. This involves waiting for the head to move to the correct track on the disk (*seek time*), and waiting until the disk has rotated such that the data to be read is directly under the head (*rotational delay*). Once the head and disk are positioned correctly, we can start reading or writing. Note that if we want to read a large chunk of data stored consecutively on disk, we have to pay the seek time and rotational delay only once, leading to a much better average I/O-time per data element. Therefore data transfer between disk and main memory is not performed on individual elements but on *blocks* of consecutive elements: When you read a single data element from disk, you actually get an entire block of data—whether you like it or not. To make an algorithm I/O-efficient, you want to make sure that the extra data that you get are actually useful. In other words: you want your algorithm to exhibit *spatial locality*: when an element is needed by the algorithm, elements from the same block are also useful because they are needed soon as well.

When we need to fetch a block from external memory, chances are that the internal memory is full. In this case we have to *evict* another block—that is, write it back to external memory—before we can bring in the block we need. If we need the evicted block again at some later point in time, we may have to fetch it again. Thus we would like our algorithms to

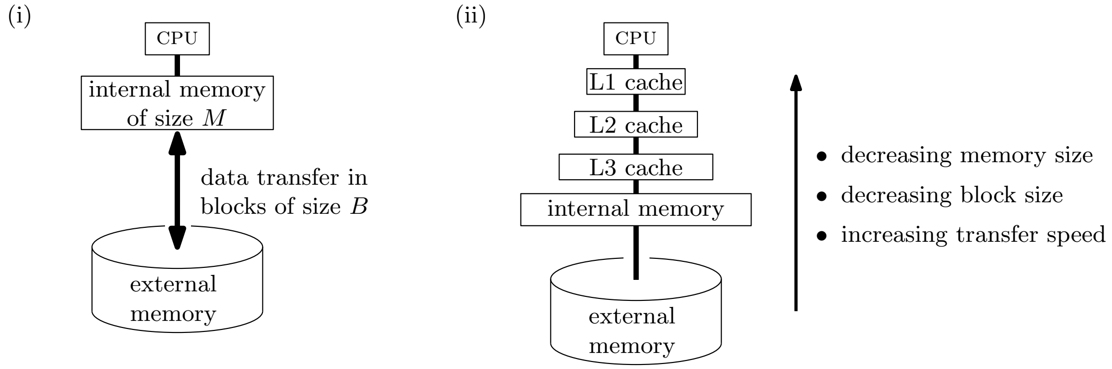

Fig. 5.1: (i) The I/O-model: a two-level memory hierarchy. (ii) A more realistic (but still simplified) multi-level memory hierarchy.

exhibit *temporal locality*: we would like the accesses to any given data element to be clustered in time, so we can keep (the block containing) the element in internal memory for some time and avoid spending an I/O every time we need it. Note that this requires a good *replacement policy* (or, *caching policy*): if we need a block that is currently in internal memory again in the near future, we should not evict it to make room for another block. Instead, we should evict a block that is not needed for a long time.

To summarize, we have the following model; see also Fig. 5.1. We have an internal memory of size M and a disk of unlimited storage capacity. Data is stored in external memory, and transferred between internal and external memory, in blocks of size B. Obviously we must have $M \geqslant B$. Sometimes we need to assume that $M = \Omega(B^2)$; this is called the *tall-cache assumption.*

Analyzing an algorithm in the I/O-model means expressing the number of I/O-operations (block transfers) as a function of M, B, and the input size n. Note that this analysis ignores the number of CPU-operations, which is the traditional measure of efficiency of an algorithm. Of course this is still a relevant measure: for a massive data set, a running time (that is, number of CPU-operations) of $\Theta(n^2)$, say, is problematic even if the number of I/O-operations is small. Hence, an external-memory algorithm should not only have good I/O-efficiency but also a good running time in the traditional sense. Ideally, the running time is the same as the best running time that can be achieved with an internal-memory algorithm.

Controlling the block formation and replacement policy? The I/O-performance of an algorithm is not only determined by the algorithm itself, but also by two other issues: (i) the way in which data elements are grouped into blocks, and (ii) the policy used to decide which block is evicted from internal memory when room has to be made for a new block. In our discussions we will explicitly take the first issue into account, and describe how the data is grouped into blocks in external memory. As for the second issue, we usually make the following assumption: the operating system uses an optimal caching strategy, that is, a strategy that leads to the minimum number of I/Os (for the given algorithm and block formation). This is not very realistic, but as we shall see later, the popular LRU strategy actually comes close to this optimal strategy.

Is the model realistic? The memory organization of a computer is actually much more involved than the abstract two-level model we described above. For instance, besides disk and main memory, there are various levels of cache. Interestingly, the same issues that play a role between main memory and disk also play a role between the cache and main memory: data is transferred in blocks (which are smaller than disk blocks) and one would like to minimize the number of cache misses. The same issues even arise between different cache levels. Thus, even when the data fits entirely in the computer's main memory, the model is relevant: the "internal memory" can then represent the cache, for instance, while the "external memory" represents the main memory. Another simplification we made is in the concept of blocks, which is more complicated than it seems. There is a physical block size (which is the smallest amount of data that can actually be read) but one can artificially increase the block size. Moreover, a disk typically has a disk cache which influences its efficiency. The minimum block size imposed by the hardware is typically around 512 bytes in which case we would have $B = 64$ when the individual elements need 8 bytes. In practice it is better to work with larger block sizes, so most operating systems work with block sizes of 4KB or more.

Despite the above, the abstract two-level model gives a useful prediction of the I/O-efficiency of an algorithm, and algorithms that perform well in this model typically also perform well in practice on massive data sets.

Cache-aware versus cache-oblivious algorithms. To control block formation it is convenient to know the block size B, so that one can explicitly write things like "put these B elements together in one block". Similarly, for some algorithms it may be necessary to know the internal-memory size M. Algorithms that make use of this knowledge are called *cache-aware*. When running a cache-aware algorithm one first has to figure out the values of B and M for the platform the algorithm is running on; these values are then passed on to the algorithm as parameters.

Algorithms that do not need to know B and M are called *cache-oblivious*. For cache-oblivious algorithms, one does not need to figure out the values of B and M—the algorithm can run on any platform without knowledge of these parameters and will be I/O-efficient no matter what their values happen to be for the given platform. A major advantage of this is that cache-oblivious algorithms are automatically efficient across all levels of the memory hierarchy: it is I/O-efficient with respect to data transfer between main memory and disk, it is I/O-efficient with respect to data transfer between L3 cache and main memory, and so on. For a cache-aware algorithm to achieve this, one would have to know the sizes and block sizes of each level in the memory hierarchy, and then set up the algorithm in such a way that it takes all these parameters explicitly into account. Another advantage of cache-oblivious algorithms is that they keep being efficient when the amount of available memory changes during the execution—in practice this can easily happen due to other processes running on the same machine and claiming parts of the memory.

Note that the parameters B and M are not only used in the analysis of cache-aware algorithms but also in the analysis of cache-oblivious algorithms. The difference lies in how the algorithm works—cache-aware algorithms need to know the actual values of B and M, cache-oblivious algorithms do not—and not in the analysis. When analyzing a cache-oblivious algorithm, one assumption is made on the block-formation process: blocks are formed according to the order in which elements are created. For instance, if we create an array A of n elements, then we assume that the first B elements—whatever the value of B may be—are

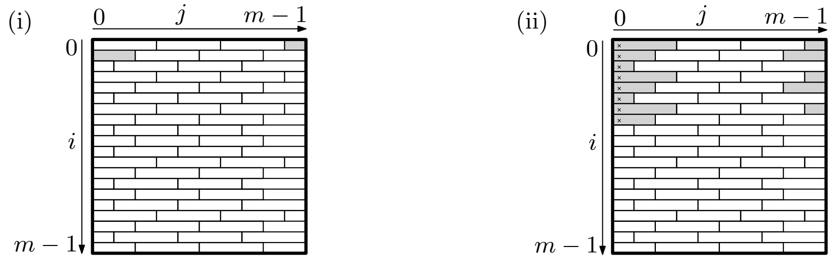

Fig. 5.2: (i) An array stored in row-major order. Blocks may "wrap around" at the end of a row; the block indicated in grey is an example of this. (ii) The blocks accessed to read the first eight elements (the crosses) from the first column.

put into one block, the second B elements are put into the second block, and so on. Another assumption, which was already mentioned earlier, is that the operating system uses an optimal replacement strategy.

5.2 A simple example: Computing the average in a 2-dimensional array

As an easy example of an I/O-analysis, let's consider the problem described earlier: compute the average of the elevation values stored in an array $A[0..m-1, 0..m-1]$. The I/O-efficiency of the two algorithms we discussed—scanning the array row by row or column by column—depends on the way in which A is stored. More precisely, it depends on how the elements are grouped into blocks. Let's assumed A is stored in *row-major order*, as in Fig. 5.2(i).

If we go through the elements of the array row by row, then the access pattern corresponds nicely to the the way in which the elements are blocked: the first B elements that we need, $A[0,0]$ up to $A[0,B-1]$, are stored together in one block, the second B elements are stored together in one block, and so on. As a result, each block is fetched exactly once, and the number of I/Os that *ComputeAverage-RowByRow* uses is $\lceil n/B \rceil$, where $n := m^2$ is the size of the input array.

If we go through the array column by column by column, however, then the first B elements that we need, $A[0,0]$ up to $A[B-1,0]$, are all be stored in different blocks (assuming $B \leqslant M$). The next B elements are again in different blocks, and so on; see Fig. 5.2(ii) for an illustration. Now if the array size is sufficiently large—more precisely, when $m > M/B$—then the internal memory becomes full at some point as we go over the first column. Hence, by the time we go the the second column we have already evicted the block containing $A[0,1]$. Thus we have to fetch this block again. Continuing this argument we see that in every step of the algorithm we will need to fetch a block from external memory. The total number of I/Os is therefore n.

We conclude that the number of I/Os of the row-by-row algorithm and the column-by-column algorithm differs by a factor B. This can make a huge difference in performance: the row-by-row algorithm could take a few minutes, say, while the column-by-column algorithm takes days. Note that both algorithms are cache-oblivious: we only used M and B in the analysis, not in the algorithm.

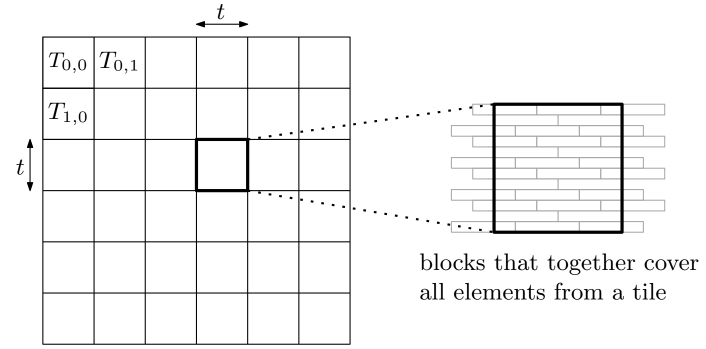

Fig. 5.3: Partitioning of a matrix into tiles, and the blocks needed to cover a given tile.

5.3 Matrix transposition

Many applications in scientific computing involve very large matrices. One of the standard operations one often has to perform on a matrix A is to compute its *transpose* A^{T}. To simplify the notation, let's consider a square matrix $A[0..m-1, 0..m-1]$. Its transpose is the matrix $A^{\mathrm{T}}[0..m-1, 0..m-1]$ defined by

$$A^{\mathrm{T}}[i,j] := A[j,i]. \tag{5.1}$$

Let's assume that A is stored in row-major order, and that we also want to store A^{T} in row-major order. Assume moreover that we do not need to keep A, and that our goal is to transform A into A^{T}. Here's a simple algorithm for this task.

```
Naive-Transpose(A)
1: for $i \leftarrow 1$ to $m-1$ do          ▷ $A[0..m-1, 0..m-1]$ is an $m \times m$ matrix
2:     for $j \leftarrow 0$ to $i-1$ do
3:         swap $A[i,j]$ and $A[j,i]$
4:     end for
5: end for
```

After the algorithm finishes, the array A stores the transpose of the original matrix in row-major order. (Another way to interpret the algorithm is that it takes a 2-dimensional array and changes the way in which the elements are stored from row-major order to column-major order.) It's not hard to see that this algorithm will incur $\Theta(n)$ I/Os if $m > M/B$. Indeed, it effectively traverses A in row-order (to obtain the values $A[i,j]$ in line 3) and in column-order (to obtain the values $A[j,i]$) at the same time. As we already saw, traversing an array in column order when it is stored in row-major order takes $\Theta(n)$ I/Os.

An I/O-efficient cache-aware algorithm. If we know the value of M, it is easy to obtain a more I/O-efficient algorithm. To this end imagine grouping the elements from the matrix into square sub-matrices of size $t \times t$, as in Fig. 5.3. (We will determine a suitable value for t later.) We call these sub-matrices *tiles*. Let's assume for simplicity that m, the size of one row or column, is a multiple of the tile size t; the algorithm is easily adapted when this is not the case. Thus we can number of tiles as $T_{i,j}$, where $0 \leqslant i,j \leqslant m/t-1$, in such a way that $T_{i,j}$ is the sub-matrix $A[it..(i+1)t-1, jt..(j+1)t-1]$.

Observe that for any given tile $T_{i,j}$, there is exactly one tile (namely $T_{j,i}$) that contains the elements with which the elements from $T_{i,j}$ should be swapped. The idea is to choose the tile size t such that we can store two complete tiles in internal memory. Then we need to read and write each tile only once, which leads to a good I/O-efficiency. It may seem sufficient to take $t := \sqrt{M/2}$, since then a tile contains at most $M/2$ elements.[2] The matrix A is stored in blocks, however, and the total size of the blocks containing the elements from a tile can be bigger than the tile size itself—see Fig. 5.3. So how should we choose t such that the total size of the blocks covering any given tile $T_{i,j}$ does not exceed $M/2$? Note that each row of $T_{i,j}$ has at most two blocks sticking out: one on the left and one on the right. Hence, the blocks that cover a $t \times t$ tile have total size at most $t^2 + 2t(B-1)$. We thus want

$$t^2 + 2t(B-1) \leqslant M/2,$$

which is satisfied when we take $t := \sqrt{M/2 + B^2} - B$. This gives us the following algorithm.

Algorithm 5.1 Cache-aware algorithm for matrix transposition.

```
CacheAware-Transpose(A, M, B)
t ← $\sqrt{M/2 + B^2} - B$          ▷ $M$ is size of internal memory, $B$ is block size
for $i$ ← 0 to $m/t$ do              ▷ $A[0..m-1, 0..m-1]$ is an $m \times m$ matrix
    for $j$ ← 0 to $i$ do
        Read the tiles $T_{i,j}$ and $T_{j,i}$ from external memory.
        Swap the elements of these tiles according to Equation (5.1).
        Write the tiles back to external memory.
    end for
end for
```

Theorem 5.1 *Let $A[0..m-1, 0..m-1]$ be a matrix of size $n := m^2$. Under the tall-cache assumption, we can transpose A with a cache-aware algorithm that performs $O(n/B)$ I/Os.*

Proof. Consider algorithm *CacheAware-Transpose*. It handles $O(n/t^2)$ pairs of tiles. To read a given tile from external memory, we read at most $t/B + 2$ blocks per row of the tile. Hence, in total we need at most $2t(t/B + 2)$ I/Os to read two tiles. Similarly, the number of I/Os needed to write a tile back to external memory is $2t(t/B + 2)$. The total number of I/Os over all pairs of tiles is therefore bounded by

$$O\left(\frac{n}{t^2} \cdot 4t(t/B + 2)\right) = O\left(\frac{n}{B} + \frac{n}{t}\right),$$

where $t = \sqrt{M/2 + B^2} - B$. Under the tall-cache assumption we have

$$t = \sqrt{M/2 + B^2} - B \geqslant \sqrt{B^2/2 + B^2} - B = (\sqrt{3/2} - 1)B$$

which implies $n/t = O(n/B)$, thus proving the theorem. □

It is instructive to think about why the tall-cache assumption is actually needed. Intuitively,

[2] Since t should be an integer, we should actually take $t := \left\lfloor \sqrt{M/2} \right\rfloor$. For simplicity we omit the floor function here and in the computations to follow.

the reason is as follows. To obtain $O(n/B)$ I/Os, we can use only $O(t^2/B)$ I/Os per tile. Stated differently, we want to use only $O(t/B)$ I/Os per row of the tile. Because there can be two blocks that are sticking out, the actual number of blocks per row is $\lfloor t/B \rfloor + 2$. The "+2" in this formula disappears in the O-notation, that is, $\lfloor t/B \rfloor + 2 = O(t/B)$, *but only when* $t/B = \Omega(1)$. In other words, we need $t = \Omega(B)$ to be able to "pay" for fetching the two blocks sticking out. Hence, a tile should have size $\Omega(B^2)$. As we want to be able to store two tiles completely in internal memory, we thus need $M = \Omega(B^2)$.

A cache-oblivious algorithm. The matrix-transposition algorithm described above needs to know the memory size M and block size B. We now give a cache-oblivious algorithm for matrix transposition. The new algorithm uses an idea that is useful in many other contexts: If we apply divide-and-conquer, then the problem size in the recursive calls decreases gradually from n (the initial problem size) to $O(1)$ (the base case). Hence, there will always be a moment when the subproblem to be solved has roughly size M and can be solved entirely in internal memory. Hence, if we can perform the conquer-step in the algorithm in an I/O-efficient and cache-oblivious manner, then the whole algorithm is I/O-efficient and cache-oblivious.

Our recursive algorithm will have four parameters (besides the array A): indices i_1, i_2, j_1, j_2 with $i_1 \leqslant i_2$ and $j_1 \leqslant j_2$. The task of the algorithm is to swap the elements in the sub-matrix $A[i_1..i_2, j_1..j_2]$ with the elements in the sub-matrix $A[j_1..j_2, i_1..i_2]$, according to Equation (5.1). We will make sure that $A[i_1..i_2, j_1..j_2]$ either lies entirely above the main diagonal of the matrix A, or the diagonal of the sub-matrix is a part of the diagonal of A. This ensures that each element is swapped exactly once, as required. Initially we have $i_1 = j_1 = 0$ and $i_2 = j_2 = m - 1$.

In a generic step, the algorithm splits the sub-matrix $A[i_1..i_2, j_1..j_2]$ into four sub-matrices on which it recurses. If diagonal of A crosses the sub-matrix—when this happens we must have $i_1 = j_1$ and $i_2 = j_2$—then one of the four smaller sub-matrices lies below the diagonal of A. Hence, we should not recurse on this sub-matrix. The test in line 8 of the algorithm below takes care of this. The recursion stops when $i_1 = i_2$ or $j_1 = j_2$. It is not hard to verify that in this case we either have a 1×1 sub-matrix, or a 2×1 sub-matrix, or a 1×2 sub-matrix. This base case ensures that we never make a call on an empty sub-matrix.

Algorithm 5.2 Cache-oblivious algorithm for matrix transposition.

```
   CacheOblivious-Transpose($A, i_1, i_2, j_1, j_2$)
 1: if $i_1 = i_2$ or $j_1 = j_2$ then
 2:     swap $A[i_1..i_2, j_1..j_2]$ and $A[j_1..j_2, i_1..i_2]$ according to Equation (5.1)
 3: else
 4:     $i_{\mathrm{mid}} \leftarrow \lfloor (i_1 + i_2)/2 \rfloor$; $j_{\mathrm{mid}} \leftarrow \lfloor (j_1 + j_2)/2 \rfloor$
 5:     CacheOblivious-Transpose($A, i_1, i_{\mathrm{mid}}, j_1, j_{\mathrm{mid}}$)
 6:     CacheOblivious-Transpose($A, i_{\mathrm{mid}} + 1, i_2, j_1, j_{\mathrm{mid}}$)
 7:     CacheOblivious-Transpose($A, i_{\mathrm{mid}} + 1, i_2, j_{\mathrm{mid}} + 1, j_2$)
 8:     if $i_1 \geqslant j_{\mathrm{mid}} + 1$ then
 9:         CacheOblivious-Transpose($A, i_1, i_{\mathrm{mid}}, j_{\mathrm{mid}} + 1, j_2$)
10:     end if
11: end if
```

Theorem 5.2 *Let $A[0..m-1, 0..m-1]$ be a matrix of size $n := m^2$. Under the tall-cache assumption, we can transpose A with a cache-oblivious algorithm that performs $O(n/B)$ I/Os.*

Proof. Consider algorithm *CacheOblivious-Transpose*. One way to think about the algorithm is that it recursively partitions A into sub-matrices until the sub-matrices are such that two of them (including the blocks that are "sticking out") fit in internal memory. At this point the sub-matrices have the same size as the tiles in the cache-aware algorithm. A recursive call on such a sub-matrix will now partition the sub-matrix into even smaller pieces, but from the I/O point of view this is irrelevant: all data needed for the call and sub-calls from now on fits in internal memory, and the assumption that the operating system uses an optimal replacement policy guarantees that all relevant blocks are read only once. Hence, the same computation as in the analysis of the cache-aware algorithm shows that the cache-oblivious algorithm performs $O(n/B)$ I/Os.

A more precise proof can be given by writing a recurrence for $T(t)$, the number of I/Os the algorithm performs when called on a sub-matrix of total size $t \times t$. (For simplicity we ignore the fact that the dimensions of the array may differ by one in a recursive call, that is, we ignore that a call can also be on a $t \times (t+1)$ or $(t+1) \times t$ sub-matrix.) Following the arguments from the proof of Theorem 5.1, we see that when $2t^2 + 4t(B-1) < M$, the entire computation fits in internal memory. Hence, we have

$$T(t) \leqslant \begin{cases} 4t(t/B+2) & \text{if } 2t^2+4t(B-1) < M \\ 4T(t/2) & \text{otherwise} \end{cases}$$

Using the tall-cache assumption one can now show that $T(t) = O(t^2/B)$. Since in the first call we have $t = m = \sqrt{n}$ this proves the theorem. □

5.4 Replacement policies

When describing our I/O-efficient algorithms we do not always explicitly describe how block replacement is being done—in particular we do not describe this for cache-oblivious algorithms. Instead we make the assumption that the operating system employs an optimal block-replacement policy. In this section we show that this is not as unrealistic as it may seem. In particular, we show that the popular LRU policy is close to being optimal. First, let's define LRU and another replacement policy called MIN. Both policies only evict a block when necessary, that is, when the internal memory is full and we need to make room to be able to bring in a block from external memory. The difference in the two policies is which block they evict.

LRU **(Least Recently Used).** The LRU replacement policy always evicts the block that has not been used for the longest time. In other words, if τ_i denotes the last time any data element in the i-th block was accessed then LRU will evict the block for which τ_i is smallest. The idea is that if a block has been used recently, then chances are it will be needed again soon because the algorithm (hopefully) has good temporal locality. Thus it is better to evict a block that has not been used for a long time.

MIN **(Longest Forward Distance).** The MIN replacement policy always evicts the block whose next usage is furthest in the future, that is, the block that is not needed for the longest period of time. (If there are blocks that are not needed at all anymore, then such a block is chosen.) One can show that MIN is optimal: for any algorithm (and memory size and block size), the MIN policy performs the minimum possible number of I/Os.

There is one important difference between these two policies: LRU is an *on-line* policy—a policy that can be implemented without knowledge of the future—while MIN is not. In fact, MIN cannot be implemented by an operating system, because the operating system does not know how long it will take before a block is needed again. However, we can still use it as a "golden standard" to assess the effectiveness of LRU.

Suppose we run an algorithm ALG on a given input I, and with an initially empty internal memory. Assume the block size B is fixed. We now want to compare the number of I/Os that LRU needs to the number of I/Os that MIN would need. It can be shown that if LRU and MIN have the same amount of internal memory available, then LRU can be much worse than MIN; see Exercise 5.7. However, when we give LRU slightly more internal memory to work with, then the performance comes close to the performance of MIN. To make this precise, we define

LRU (ALG,M) := number of I/O-operations performed when algorithm ALG is run
with the LRU replacement policy and internal-memory size M.

We define MIN (ALG,M) similarly for the MIN replacement policy. We now have the following theorem.

Theorem 5.3 *For any algorithm* ALG, *and any M and M' with $M \geqslant M'$, we have*

$$\text{LRU}(\text{ALG}, M) \leqslant \frac{M}{M - M' + B} \cdot \text{MIN}(\text{ALG}, M').$$

In particular, $\text{LRU}(\text{ALG}, M) < 2 \cdot \text{MIN}(\text{ALG}, M/2)$.

Proof. Consider the algorithm ALG when run using LRU and with internal-memory size M. Let $t_1, t_2, \ldots, t_s$, with $s = \text{LRU}(\text{ALG}, M)$, be the moments in time where LRU fetches a block from external memory. Also consider the algorithm when run using MIN and internal-memory size M'. Note that up to $t_{M'/B}$, LRU and MIN behave the same: they simply fetch a block when they need it, but since the internal memory is not yet full, they do not need to evict anything.

Now partition time into intervals $T_0, T_1, \ldots$ in such a way that LRU fetches exactly M/B blocks during T_j for $j \geqslant 1$, and at most M/B blocks during T_0. Here we count each time any block is fetched, that is, if the same block is fetched multiple times it is counted multiple times. We analyze the behavior of MIN on these time intervals, where we treat T_0 separately.

- During T_0, LRU fetches at most M/B blocks from external memory. Since we assumed that we start with an empty internal memory, the blocks fetched in T_0 are all different. Since MIN also starts with an empty internal memory, it must also fetch each of these blocks during T_0.

- Now consider any of the remaining time intervals T_i. Let b be the last block accessed by the algorithm before T_i. Thus, at the start of T_i, both LRU and MIN have b in internal memory. We first prove the following claim:

  Claim. Let $\mathcal{B}_i$ be the set of blocks accessed during T_i. (Note that $\mathcal{B}_i$ contains all blocks that are accessed, not just the ones that need to be fetched from external memory.) Then $\mathcal{B}_i$ contains at least M/B blocks that are different from b.

To prove the claim, we distinguish three cases. First, suppose that b is evicted by LRU during T_i. Because b is the most recently used block in LRU's memory at the start of T_i, at least $M/B - 1$ other blocks must be accessed before b can become the least recently used block. We then need to access at least one more block (from external memory) before b needs to be evicted, thus proving the claim. Second, suppose that some other block b' is evicted twice by LRU during T_i. Then a similar argument as in the first case shows that we need to access at least M/B blocks in total during T_i. If neither of these two cases occurs, then all blocks fetched by LRU during T_i are distinct (because the second case does not apply) and different from b (because the first case does not apply), which also implies the claim.

At the start of T_i, MIN has only M'/B blocks in its internal memory, one of which is b. Hence, at least $M/B - M'/B + 1$ blocks from $\mathcal{B}_i$ are not in MIN's internal memory at the start of T_i. These blocks must be fetched by MIN during T_i. This implies that

$$\frac{\text{number of I/Os performed by LRU during } T_i}{\text{number of I/Os performed by MIN during } T_i} \leqslant \frac{M/B}{M/B - M/B' + 1} = \frac{M}{M - M' + B}.$$

We conclude that during T_0 LRU uses at most the same number of I/Os as MIN, and for each subsequent time interval T_i the ratio of the number of I/Os is $M/(M - M' - B)$. The theorem follows. □

5.5 Exercises

Exercise 5.1 Consider a machine with 1 GB of internal memory and a disk with the following properties:

- the average time until the read/write head is positioned correctly to start reading or writing data (seek time plus rotational delay) is 12 ms,
- once the head is positioned correctly, we can read or write at a speed of 60 MB/s.

The standard block size used by the operating system to transfer data between the internal memory and the disk is 4 KB. We use this machine to sort a file containing 10^8 elements, where the size of each element is 500 bytes; thus the size of the file is 50 GB. Our sorting algorithm performs roughly $2\frac{n}{B}\lceil \log_{M/B} \frac{n}{B} \rceil$ I/Os for a set of n elements, where M is the number of elements that fit into the internal memory and B is the number of elements that fit into a block. Here we assume that 1 KB = 10^3 bytes, 1 MB = 10^6 bytes, and 1 GB = 10^9 bytes.

(i) Compute the total time in hours spent on I/Os by the algorithm when we work with the standard block size of 4 KB.

(ii) Now suppose we force the algorithm to work with blocks of size 1 MB. What is the time spent on I/Os in this case?

(iii) Same question as (ii) for a block size of 250 MB.

Exercise 5.2 Consider the algorithm that computes the average value in an $m \times m$ array A column by column, for the case where A is stored in row-major order. Above (on page 46) it was stated that the algorithm performs n I/Os, where $n := m^2$ is the total size of the array, because whenever we need a new entry from the array we have already evicted the block containing that entry. This is true when LRU is used and when $m > M/B$, but it is not immediately clear what happens when some other replacement policy is used.

(i) Suppose that $m = M/B + 1$, where you may assume that M/B is integral. Show that in this case there is a replacement policy that would perform only $O(n/B + \sqrt{n})$ I/Os.

(ii) Prove that when $m > 2M/B$, then any replacement policy will perform $\Omega(n)$ I/Os.

(iii) In the example in the introduction to this chapter, the array storing the elevation values had size $100,000 \times 100,000$ and each elevation value was an 8-byte number. Suppose that the block size B is 1024 bytes. Is it realistic to assume that we cannot store one block for each row in internal memory in this case? Explain your answer.

(iv) Suppose m, M, and B are such that $m = M/(2B)$. Thus we can easily store one block for each row in internal memory. Would you expect that in this case the actual running times of the row-by-row algorithm and the column-by-column algorithm are basically the same? Explain your answer.

Exercise 5.3 A *stack* is a data structure that supports two operations: we can *push* a new element onto the stack, and we can *pop* the topmost element from the stack. We wish to implement a stack in external memory. To this end we maintain an array $A[0..m-1]$ on disk. The value m, which is the maximum stack size, is kept in internal memory. We also maintain the current number of elements on the stack, s, in internal memory. The *push*- and *pop*-operations can now be implemented as follows:

```
Push(A, x)
if $s = m$ then
    return “stack overflow”
else
    $A[s] \leftarrow x$; $s \leftarrow s + 1$
end if
```

```
Pop(A)
if $s = 0$ then
    return “stack is empty”
else
    return $A[s-1]$; $s \leftarrow s - 1$
end if
```

Assume A is blocked in the standard manner: $A[0 \ldots B-1]$ is the first block, $A[B \ldots 2B-1]$ is the second block, and so on.

(i) Suppose we allow the stack to use only a single block from A in internal memory. Show that there is a sequence of n operations that requires $\Theta(n)$ I/Os.

(ii) Now suppose we allow the stack to keep two blocks from A in internal memory. Prove that any sequence of n *push*- and *pop*-operations requires only $O(n/B)$ I/Os.

Exercise 5.4 Let $A[0..n-1]$ be an array with n elements that is stored on disk, where n is even. We want to generate a random subset of $n/2$ elements from A, and store the subset in an array $B[0..(n/2)-1]$.

The following algorithm computes such a random subset. It simply picks $n/2$ times a random element from A. (Note that the same element may be picked multiple times). To this end, the algorithm uses a random-number generator *Random*(a, b) that, given two integers a and b with $a \leqslant b$, generates an integer $r \in \{a, a+1, \dots, b\}$ uniformly at random.

```
RandomSubset-I(A)
▷ $A[0..n-1]$ is an array with $n$ elements, for $n$ even.
for $i \leftarrow 0$ to $(n/2) - 1$ do
    $r \leftarrow Random(0, n-1)$
    $B[i] \leftarrow A[r]$
end for
```

We can also generate a random subset of $n/2$ distinct elements, by going over all the n elements in A and deciding for each element whether to select it into the subset. More precisely, when we arrive at element $A[i]$ (and we still have $n - i$ elements left to choose from, including $A[i]$) and we already selected m elements (and thus we still need to select $n/2 - m$ elements) then we select $A[i]$ with probability $(n/2 - m)/(n - i)$.

```
RandomSubset-II(A)
▷ $A[0..n-1]$ is an array with $n$ elements, for $n$ even.
$m \leftarrow 0$                              ▷ $m$ is the number of elements already selected.
for $i \leftarrow 0$ to $n - 1$ do
    $r \leftarrow Random(1, n-i)$
    if $r \leqslant n/2 - m$ then
        $B[m] \leftarrow A[i]$; $m \leftarrow m + 1$
    end if
end for
```

Assume the memory size M is much smaller than the array size n. Analyze the expected number of I/Os performed by both algorithms. Take into account the I/Os needed to read elements from A and the I/Os needed to write elements to B.

Exercise 5.5 Suppose we are given two $m \times m$ matrices X and Y, which are stored in row-major order in 2-dimensional arrays $X[0..m-1, 0..m-1]$ and $Y[0..m-1, 0..m-1]$. We wish to compute the product $Z = XY$, which is the $m \times m$ matrix defined by

$$Z[i, j] := \sum_{k=0}^{m-1} X[i, k] \cdot Y[k, j],$$

for all $0 \leqslant i, j < m$. The matrix Z should also be stored in row-major order. Let $n := m^2$. Assume that m is much larger than M and that $M \geqslant B^2$.

(i) Analyze the I/O-complexity of the matrix-multiplication algorithm that simply computes the elements of Z one by one, row by row. You may assume LRU is used as replacement policy.

(ii) Analyze the I/O-complexity of the same algorithm if X and Z are stored in row-major order while Y is stored in column-major order. You may assume LRU is used as replacement policy.

(iii) Consider the following alternative algorithm. For a square matrix Q with more than one row and column, let Q_{TL}, Q_{TR}, Q_{BL}, Q_{BR} be the top left, top right, bottom left, and bottom right quadrant, respectively, that result from cutting Q between rows $\lfloor m/2 \rfloor$ and $\lfloor m/2 \rfloor + 1$, and between columns $\lfloor m/2 \rfloor$ and $\lfloor m/2 \rfloor + 1$. We can compute $Z = XY$ recursively by observing that

$$\begin{aligned} Z_{TL} &= X_{TL}Y_{TL} + X_{TR}Y_{BL}, \\ Z_{TR} &= X_{TL}Y_{TR} + X_{TR}Y_{BR}, \\ Z_{BL} &= X_{BL}Y_{TL} + X_{BR}Y_{BL}, \\ Z_{BR} &= X_{BL}Y_{TR} + X_{BR}Y_{BR}. \end{aligned}$$

Analyze the I/O-complexity of this recursive computation. You may assume that m is a power of two, and that an optimal replacement policy is used.
Hint: Express the I/O-complexity as a recurrence with a suitable base case, and solve the recurrence.

(iv) If you solved (ii) and (iii) correctly, you have seen that the number of I/Os of the recursive algorithm is smaller than the number of I/Os in part (ii). Does the algorithm from (iii) have better spatial locality than the algorithm from (i)? And does it have better temporal locality? Explain your answers.

Exercise 5.6 Let $A[0..n-1]$ be a sorted array of n numbers, which is stored in blocks in the standard way: $A[0..B-1]$ is the first block, $A[B..2B-1]$ is the second block, and so on.

(i) Analyze the I/O-complexity of binary search on the array A. Make a distinction between the case where the internal memory can store only one block and the case where it can store more than one block.

(ii) Describe a different way to group the elements into blocks such that the I/O-complexity of binary search is improved significantly, and analyze the new I/O-complexity. Does it make a difference if the internal memory can store one block or more than one block?

(iii) Does your solution improve spatial and/or temporal locality? Explain your answer.

Exercise 5.7 This exercise shows that the bound from Theorem 5.3 is essentially tight.

(i) Suppose that an algorithm operates on a data set of size $M + B$, which is stored in blocks $b_1, \ldots, b_{M/B+1}$ in external memory. Consider LRU and MIN, where the replacement policies have the same internal memory size, M. Show that there is an infinitely long sequence of block accesses such that, after the first M/B block accesses (on which both LRU and MIN perform an I/O) LRU will perform an I/O for every block access while MIN only performs an I/O once every M/B access.
NB: On an algorithm with this access sequence we have $\text{LRU}(\text{ALG}, M) \to \frac{M}{B} \cdot \text{MIN}(\text{ALG}, M)$ as the length of the sequence goes to infinity, which shows that the bound from Theorem 5.3 is essentially tight for $M = M'$.

(ii) Now suppose LRU has an internal memory of size M while MIN has an internal memory of size M'. Generalize the example from (i) to show that there are infinitely long access sequences such that $\text{LRU}(\text{ALG}, M) \to \frac{M}{M-M'+B} \cdot \text{MIN}(\text{ALG}, M')$ as the length of the sequence goes to infinity. (This shows that the bound from Theorem 5.3 is also essentially tight for $M > M'$.)

Exercise 5.8 Consider an image-processing application that repeatedly scans an image, over and over again, row by row from the top down. The image is also stored row by row in memory. The image contains n pixels, while the internal memory can hold M pixels and pixels are moved into and out of cache in blocks of size B, where $M \geqslant 3B$.

(i) Construct an example—that is, pick values for B, M and n—such that the LRU caching policy results in M/B times more cache misses than an optimal caching policy for this application (excluding the first M/B cache misses of both strategies).

(ii) If we make the image only half the size, then what is the performance ratio of LRU versus optimal caching?

(iii) If we make the image double the size, then what is the performance ratio of LRU versus optimal caching?

Exercise 5.9 Consider an algorithm that needs at most $cn\sqrt{n}/(MB)$ I/Os if run with optimal caching, for some constant c. Prove that the algorithm needs at most $c'n\sqrt{n}/(MB)$ I/Os when run with LRU caching, for a suitable constant c'.

Exercise 5.10 Let $X[0..n-1]$ and $Y[0..n-1]$ be two arrays of n numbers, where $n \gg M$. Suppose we have a function $f : \mathbb{R}^2 \to \mathbb{R}$ and we want to compute $\min\{f(X[i], Y[j]) : 0 \leqslant i < n \text{ and } 0 \leqslant j < n\}$. If we have no other information about f, then the only thing we can do is compute $f(X[i], Y[j])$ for all pairs i, j with $0 \leqslant i < n$ and $0 \leqslant j < n\}$. A simple way to do this is using the following algorithm.

```
FindMin(X, Y)
z ← +∞
for i ← 0 to n − 1 do
    for j ← 0 to n − 1 do
        z ← min(z, f(X[i], Y[j]))
    end for
end for
return z
```

(i) Analyze the number of I/Os performed by *FindSum*.

(ii) Give a cache-aware algorithm that solves the problem using $O(n^2/(MB))$ I/Os.

(iii) Give a cache-oblivious algorithm that solves the problem using $O(n^2/(MB))$ I/Os.

Chapter 6

Sorting and Permuting

In this chapter we study I/O-efficient algorithms for sorting. We will present a sorting algorithm that performs $O(\frac{n}{B}\log_{M/B}\frac{n}{B})$ I/Os in the worst case to sort n elements, and we will prove that this is optimal (under some mild assumptions). For simplicity we assume the input to our sorting algorithm is an array of n numbers. (In an actual application the array would store a collection of records, each storing a numerical *key* and various other information, and we want to sort the records by their key.) We also assume that the numbers we wish to sort are distinct; the algorithms can easily be adapted to deal with the case where numbers can be equal.

6.1 An I/O-efficient sorting algorithm

Let $A[0..n-1]$ be an array of n distinct numbers. We want to sort A into increasing order. There are many algorithms that can sort A in $O(n\log n)$ time when A fits entirely in internal memory. One such algorithm is *MergeSort*. We will first show that *MergeSort* already has fairly good I/O-behavior. After that we will modify *MergeSort* to improve its I/O-efficiency even more. As usual, we assume that initially the array A is stored consecutively on disk (or rather, we assume that $A[0..B-1]$ is one block, $A[B..2B-1]$ is one block, and so on).

MergeSort is a divide-and-conquer algorithm: it partitions the input array A into two smaller arrays A_1 and A_2 of roughly equal size, recursively sorts A_1 and A_2, and then merges the two results to obtain the sorted array A. In the following pseudocode, *length*(A) denotes the length (that is, number of elements) of an array A.

```
MergeSort(A)
1: $n \leftarrow length(A)$
2: if $n > 1$ then                                         ▷ else $A$ is sorted by definition
3:     $n_{\mathrm{left}} \leftarrow \lfloor n/2 \rfloor$; $n_{\mathrm{right}} \leftarrow \lceil n/2 \rceil$
4:     $A_1[0..n_{\mathrm{left}}-1] \leftarrow A[0..n_{\mathrm{left}}-1]$; $A_2[0..n_{\mathrm{right}}-1] \leftarrow A[n_{\mathrm{left}}..n-1]$
5:     MergeSort($A_1$); MergeSort($A_2$)
6:     Merge $A_1$ and $A_2$ to obtain the sorted array $A$
7: end if
```

In line 6 we have to merge the sorted arrays A_1 and A_2 to get the sorted array A. This can be done by simultaneously scanning A_1 and A_2 and writing the numbers we encounter to their correct position in A. More precisely, when the scan of A_1 is at position $A_1[i]$ and the scan of A_2 is at position $A_2[j]$, we write the smaller of the two numbers to $A[i+j]$—in other words, we set $A[i+j] \leftarrow \min(A_1[i], A_2[j])$—and we increment the corresponding index (i or j). When the scan reaches the end of one of the two arrays, we simply write the remaining numbers in the other array to the remaining positions in A. This way the merge step takes $O(n)$ time. Hence, running time $T(n)$ of the algorithm satisfies[1] $T(n) = 2T(n/2) + O(n)$ with $T(1) = O(1)$, and so $T(n) = O(n \log n)$.

Let's now analyze the I/O-behavior of *MergeSort*. Note that the merge step only needs $O(n/B)$ I/Os, because it just performs three scans: one of A_1, one of A_2, and one of A. Hence, if $T_{\text{IO}}(n)$ denotes the (worst-case) number of I/Os performed when *MergeSort* is run on an array of length n, then

$$T_{\text{IO}}(n) = 2T_{\text{IO}}(n/2) + O(n/B). \tag{6.1}$$

What is the base case for the recurrence? When writing a recurrence for the running time of an algorithm, we usually take $T(1) = O(1)$ as base case. When analyzing the number of I/Os, however, we typically have a different base case: as soon as we have a recursive call on a subproblem that can be solved completely in internal memory, we only need to bring the subproblem into internal memory once (and write the solution back to disk once). In our case, when $n \leqslant M/2$ the internal memory can hold the array A as well as the auxiliary arrays A_1 and A_2, so we have

$$T_{\text{IO}}(M/2) = O(M/B).$$

Together with (6.1) this implies that $T_{\text{IO}}(n) = O((n/B)\log_2(n/M))$. This I/O bound is already quite good. It is not optimal, however: the base of the logarithm can be improved. Note that the factor $O(\log_2(n/M))$ in the I/O-bound is equal to the number of levels of recursion before the size of the subproblem drops below $M/2$. The logarithm in this number has base 2 because we partition the input array into two (roughly equal sized) subarrays in each recursive step; if we would partition into k subarrays, for some $k > 2$, then the base of the logarithm would be k. Thus we change the algorithm as follows.

Algorithm 6.1 A cache-aware I/O-efficient version of *MergeSort*.

```
EM-MergeSort($A$)
$n \leftarrow length(A)$
if $n > 1$ then                                  ▷ else $A$ is sorted by definition
    Pick a suitable value of $k$ (see below).
    Partition $A$ into $k$ roughly equal-sized subarrays $A_1, \ldots, A_k$.
    for $i \leftarrow 1$ to $k$ do
        EM-MergeSort($A_i$)
    end for
    Merge $A_1, \ldots, A_k$ to obtain the sorted array $A$
end if
```

We would like to choose k as large as possible. However, we still have to be able to do the merge step with only $O(n/B)$ I/Os. This means that we should be able to keep at least one block from each of the subarrays $A_1, \ldots, A_k$ (as well as from A) in main memory, so that we

[1] More precisely, we have $T(n) = T(\lfloor n/2 \rfloor) + T(\lceil n/2 \rceil) + O(n)$, but (as usual) we omit the floor and ceiling for simplicity.

can simultaneously scan all these arrays without running into memory problems. Hence, we set $k := M/B - 1$ and we get the recurrence

$$T_{\text{IO}}(n) = \sum_{i=1}^{k} T_{\text{IO}}(n/k) + O(n/B). \tag{6.2}$$

with, as before, $T_{\text{IO}}(M/2) = O(M/B)$. The solution is $T_{\text{IO}}(n) = O((n/B)\log_{M/B}(n/M))$, which is equivalent[2] to

$$T_{\text{IO}}(n) = O((n/B)\log_{M/B}(n/B)).$$

(See also Exercise 6.2.) Thus we managed to increase the base of the logarithm from 2 to M/B. This increase can be significant. For instance, for $n = 1,000,000,000$ and $M = 1,000,000$ and $B = 100$ we have $\log_2 n \approx 29.9$ and $\log_{M/B} n = 2.25$, so changing the algorithm as just described may give a 10-fold reduction in the number of I/Os. (This computation is perhaps not very meaningful, because we did not analyze the constant factor in the number of I/Os. These are similar, however, and in practice the modified algorithm indeed performs much better for very large data sets.)

The above leads to the following theorem.

Theorem 6.1 *An array $A[0..n-1]$ with n numbers can be sorted using $O((n/B)\log_{M/B}(n/B))$ I/Os using a k-way variant of* MergeSort*, where $k := M/B - 1$.*

Notice that in order to obtain the best performance, we take $k := M/B - 1$. Thus the choice of k depends on M and B, which means that the modified version of the algorithm is no longer cache-oblivious. There are also cache-oblivious sorting algorithms that perform only $O((n/B)\log_{M/B}(n/B))$ I/Os.

Many external-memory algorithms use sorting as a subroutine, and often the sorting steps performed by the algorithm determine its running time. Hence, it is convenient to introduce a shorthand for the number of I/Os needed to sort n elements on a machine with a memory size M and block size B. Thus we define $\text{Sort}(n) := O((n/B)\log_{M/B}(n/B))$.

6.2 The permutation lower bound

In the previous section we saw a sorting algorithm that performs $O((n/B)\log_{M/B}(n/B))$ I/Os. In this section we show that (under certain mild conditions) this is optimal. In fact, we will prove a lower bound on the worst-case number of I/Os needed to sort an array of n numbers, *even if we already know the rank of each number*, that is, even if we already know where each number needs to go. To state the result more formally, we first define the *permutation problem* as follows.

Let $A[0..n-1]$ be an array where each $A[i]$ stores a pair (pos_i, x_i) such that the sequence $pos_0, pos_1, \ldots, pos_{n-1}$ of ranks is a permutation of $0, \ldots, n-1$. The permutation problem is to rearrange the array such that (pos_i, x_i) is stored in $A[pos_i]$. Note that the sorting problem is at least as hard as the permutation problem, since we not only need to move each number in the array to its correct position in the sorted order, but we also need to determine the correct position. Below we will prove a lower bound on the number of I/Os for the permutation problem, which thus implies the same lower bound for the sorting problem.

[2] The second expression is usually preferred because n/B is the size of the input in terms of the number of blocks; hence, this quantity is a natural one to use in bounds on the number of I/Os.

To prove the lower bound we need to make certain assumptions on what the permutation algorithm is allowed to do, and what it is not allowed to do. These assumptions are as follows.

- The algorithm can only *move* elements from external memory to internal memory and vice versa; in particular, the algorithm is not allowed to copy or modify elements.
- All read and write operations are performed on *full blocks*.
- When writing a block from internal to external memory, the algorithm can choose any B items from internal memory to form the block. (A different way of looking at this is that the algorithm can move around the elements in internal memory for free.)

We will call this the *movement-only model*. Observe that the second assumption implies that n is a multiple of B. In the movement-only model a permutation algorithm can be viewed as consisting of a sequence of read and write operations, where a read operations selects a block from external memory and brings it into internal memory—of course this is only possible if there is still space in the internal memory—and a write operation chooses any B elements from the internal memory and writes them as one block to a certain position in the external memory. The task of the permutation algorithm is to perform a number of read and write operations so that at end of the algorithm all elements in the array A are stored in the correct order, that is, element (pos_i, x_i) is stored in $A[pos_i]$ for all $0 \leqslant i < n$.

Theorem 6.2 *Any algorithm that solves the permutation problem in the movement-only model needs* $\Omega((n/B)\log_{M/B}(n/B))$ I/Os *in the worst case, assuming that* $n < B\sqrt{\binom{M}{B}}$.

Proof. The global idea of our proof is as follows. Let X be the total number of I/Os performed by the permutation algorithm, in the worst case. Then within X I/Os the algorithm has to be able to rearrange the input into blocks in many different ways, depending on the particular permutation to be performed. This means that algorithm should be able to reach many different "states" within X I/Os. However, one read or write can increase the number of different states only by a certain factor. Thus, if we can determine upper bounds on the increase in the number of states by doing a read or write, and we can determine the total number of different final states that the algorithm must be able to reach, then we can derive a lower bound on the total number of I/Os. Next we make this idea precise.

We assume for simplicity that the permutation algorithm only uses the part of the external memory occupied by the array A. Thus, whenever it writes a block to memory, it writes to $A[jB..(j+1)B-1]$ for some $0 \leqslant j < n/B$. (There must always be an empty block when a write operation is performed, because elements are not copied.) This assumption is not necessary—see Exercise 6.7—but it simplifies the presentation. Now we can define the *state* of (the memory of) the algorithm as the tuple $(\mathcal{M}, \mathcal{B}_0, \ldots, \mathcal{B}_{n/B-1})$, where $\mathcal{M}$ is the set of elements stored in the internal memory and the $\mathcal{B}_j$ is the set of elements in the block $A[jB..(j+1)B-1]$; see Fig. 6.1. Note that the order of the elements within a block is unimportant for the state. Because we only read and write full blocks, each subset $\mathcal{B}_j$ contains exactly B elements. Moreover, because elements cannot be copied or modified, any element is either in $\mathcal{M}$ or it is in (exactly) one of the subsets $\mathcal{B}_j$. Initially the algorithm is in a state where $\mathcal{M} = \emptyset$ and the subsets $\mathcal{B}_j$ form the blocks of the given input array. Next we derive bounds on the number of different states the algorithm must be able to reach, and on the increase in the number of reachable states when we perform a read or a write operation.

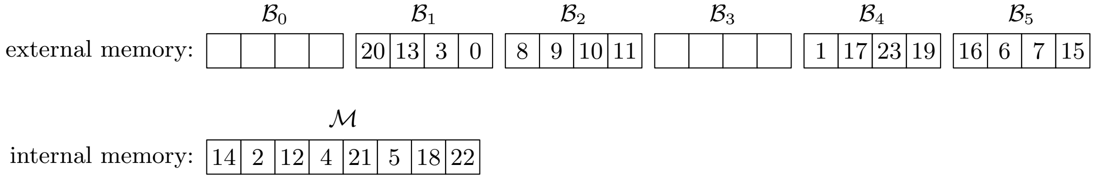

Fig. 6.1: A possible state of the permutation algorithm after several read and write operations. The numbers in the figure indicate the desired positions pos_i of the input elements. Note that $\mathcal{B}_2$ already contains the correct elements.

- We first derive a bound on the number of different states the algorithm must be able to reach to be able to produce all possible permutations. For any given input—that is, any given set of positions $pos_0, \ldots, pos_{n-1}$—there is one output state, namely where $\mathcal{M} = \emptyset$ and each $\mathcal{B}_j$ contains the elements with $jB \leqslant pos_i \leqslant (j+1)B - 1$. However, not every different input leads to a different output state, because we do not distinguish between different orderings of the elements within the same block. Observe that the number of different permutations we can make when we fix the sets $\mathcal{B}_0, \ldots, \mathcal{B}_{n/B}$ is $(B!)^{n/B}$, since for each set $\mathcal{B}_j$ we can pick from $B!$ orderings. Thus the $n!$ different inputs actually correspond to only $n!/(B!)^{n/B}$ different output states.

- Now consider a read operation. It transfers one block from external memory, thus making one of the $\mathcal{B}_j$'s empty and adding its elements to $\mathcal{M}$. The algorithm can choose at most n/B different blocks for this, so the total number of different states the algorithm can be in after a read increase by a factor of at most n/B.

- In a write operation the algorithm chooses B elements from the internal memory, forms a block out of these elements, and writes the block to one of the at most n/B available (currently empty) blocks $\mathcal{B}_j$. The number of different subsets of B elements that the algorithm can choose is $\binom{M}{B}$ and the number of possibilities to choose $\mathcal{B}_j$ is at most n/B. Hence, a write operation increases the number of different states by a factor $\binom{M}{B} \cdot (n/B)$.

Now let X_{r} denote the number of read operations the algorithm performs in the worst case, and let X_{w} denote the number of write operations the algorithm performs in the worst case. By the above, the algorithm can then reach at most

$$\left(\frac{n}{B}\right)^{X_{\mathrm{r}}} \cdot \left(\binom{M}{B} \cdot \frac{n}{B}\right)^{X_{\mathrm{w}}}$$

different states, and this number must be at least $n!/(B!)^{n/B}$ for the algorithm to function correctly on all possible inputs. Since both initially and at the end all elements are in external memory, we must have $X_{\mathrm{r}} = X_{\mathrm{w}} = X/2$. Hence,

$$\begin{aligned}
\tfrac{n!}{(B!)^{n/B}} &\leqslant \left(\tfrac{n}{B}\right)^{X_{\mathrm{r}}} \cdot \left(\tbinom{M}{B} \cdot \tfrac{n}{B}\right)^{X_{\mathrm{w}}} \\
&= \left(\tfrac{n}{B}\right)^{X/2} \cdot \left(\tbinom{M}{B} \cdot \tfrac{n}{B}\right)^{X/2} \\
&= \left(\tfrac{n}{B}\right)^{X} \cdot \tbinom{M}{B}^{X/2}.
\end{aligned}$$

It remains to show that this inequality leads to the claimed lower bound on X. The above implies that

$$X \cdot \log\left(\frac{n}{B} \cdot \binom{M}{B}^{1/2}\right) \geqslant \log\left(\frac{n!}{(B!)^{n/B}}\right). \tag{6.3}$$

We first bound the right-hand side of Inequality (6.3). From Stirling's approximation, which states that $n! \sim \sqrt{2\pi n}(n/e)^n$, it follows that $\log(n!) = n\log(n/e) + O(\log n)$. Hence, the right-hand side in (6.3) can be bounded as

$$\begin{aligned}
\log\left(\tfrac{n!}{(B!)^{n/B}}\right) &= \log(n!) - \log\left((B!)^{n/B}\right) \\
&= n\log(n/e) + O(\log n) - (n/B)\log(B!) \\
&= n\log(n/e) + O(\log n) - (n/B)\left(B\log(B/e) - O(\log B)\right) \\
&= n\log(n/B) + O(\log n) - O((n/B)\log B) \\
&= \Omega(n\log(n/B)).
\end{aligned}$$

Now consider the factor $\log\left(\frac{n}{B}\cdot\binom{M}{B}^{1/2}\right)$ in the left-hand side of (6.3). Using the condition in the theorem that $n < B\sqrt{\binom{M}{B}}$ and using the inequality $\binom{M}{B} \leqslant \left(\frac{eM}{B}\right)^B$, we obtain

$$\begin{aligned}
\log\left(\tfrac{n}{B}\cdot\tbinom{M}{B}^{1/2}\right) &< \log\tbinom{M}{B} \\
&\leqslant B\log\left(\tfrac{eM}{B}\right) \\
&< 2B\log\left(\tfrac{M}{B}\right).
\end{aligned}$$

Hence,

$$\begin{aligned}
X &\geqslant \frac{\log\left(\frac{n!}{(B!)^{n/B}}\right)}{\log\left(\frac{n}{B}\cdot\binom{M}{B}\right)} \\
&= \frac{\Omega(n\log(n/B))}{2B\log(M/B)} \\
&= \Omega((n/B)\log_{M/B}(n/B)).
\end{aligned}$$

□

Theorem 6.2 has the condition that $n < B\sqrt{\binom{M}{B}}$. This condition is satisfied for all reasonable values of n, M, and B since then $\binom{M}{B}$ is extremely large.

6.3 Exercises

Exercise 6.1 Consider the recurrence for the number of I/Os performed by (the unmodified version of) *MergeSort*:

$$T_{\text{IO}}(n) \leqslant \begin{cases} T_{\text{IO}}(\lfloor n/2\rfloor) + T_{\text{IO}}(\lceil n/2\rceil) + O(n/B) & \text{if } n > M/2 \\ O(M/B) & \text{otherwise} \end{cases}$$

Prove that $T_{\text{IO}}(n) = O((n/B)\log_2(n/M))$.

Exercise 6.2 Show that $\log_{M/B}(n/M) = \Theta(\log_{M/B}(n/B))$.

Exercise 6.3 Analyze the running time (not the number of I/Os) of algorithm *EM-MergeSort*. Does it still run in $O(n \log n)$ time? If not, explain how to implement the merge step to make it run in $O(n \log n)$ time.

Exercise 6.4 In this chapter we saw an I/O-efficient version of *MergeSort*, a standard algorithm for sorting a set of n numbers in $O(n \log n)$ time. Another standard sorting algorithm is *QuickSort*.

```
QuickSort(A)    ▷ $A[0..n-1]$ is an array of distinct numbers
1: if $n > 1$ then                                   ▷ else $A$ is sorted by definition
2:     Compute an element $A[i^*]$ such that the number of elements in $A$ smaller
       than or equal to $A[i^*]$ is $\lceil n/2 \rceil$ and the number of elements in $A$ larger than
       $A[i^*]$ is $\lfloor n/2 \rfloor$. (Thus $A[i^*]$ is a median of the elements in $A$.)
3:     Scan $A$ and write all elements smaller than or equal to $A[i^*]$ to an array $B_1$
       and write all elements larger than $A[i^*]$ to an array $B_2$.
4:     QuickSort($B_1$); QuickSort($B_2$)
5:     Scan $B_2$ and write its contents to the first $\lceil n/2 \rceil$ positions in $A$, and then
       scan $B_2$ and write its contents to the last $\lfloor n/2 \rfloor$ positions in $A$.
6: end if
7: return $A$.
```

In this exercise we consider the I/O-performance of *QuickSort*.

(i) Assume that Step 2 can be performed in $O(n/B)$ I/Os. Give a recurrence for $T_{IO}(n)$, the number of I/Os that *QuickSort* performs when run on an array of size n. Make sure you use a suitable base case in your recurrence.

(ii) Prove that the recurrence you gave in part (i) solves to $T_{IO} = O((n/B) \log_2(n/M))$.

(iii) The number of I/Os performed by *QuickSort* is $O((n/B) \log_2(n/M))$ under the assumption that we can perform Step 2 with $O(n/B)$ I/Os, which is not so easy. Therefore one often uses a randomized version of *QuickSort*, where the pivot element $A[i^*]$ is chosen randomly. (More precisely, the index i^* is chosen from the set $\{0, \ldots, n-1\}$ uniformly at random.) Intuitively, one would expect the pivot to be close enough to the median, and indeed one can show that the expected number of I/Os is $O((n/B) \log_2(n/M))$. This randomized version of *QuickSort* is nice and simple, but its I/O-performance is sub-optimal because the logarithm has base 2. Describe how to modify the randomized version of *QuickSort* such that its I/O-performance is improved. The goal would be to obtain a randomized version of *QuickSort* for which the expected number of I/Os is $O(\textsc{Sort}(n))$.

NB: Keep your explanation brief. A short description of your modification and a few lines of explanation why the expected number of I/Os would hopefully be $O(\textsc{Sort}(n))$ suffices.

Exercise 6.5 Let $A[0 \ldots n-1]$ be an array of n distinct numbers. The *rank* of an element in $A[i]$ is defined as follows:

$$\text{rank}(A[i]) := (\text{number of elements in } A \text{ that are smaller than } A[i]) + 1.$$

Define the *displacement* of $A[i]$ to be $|i - \text{rank}(A[i]) + 1|$. Thus the displacement of $A[i]$ is equal to the distance between its current position in the array (namely i) and its position when A is sorted.

(i) Suppose we know that A is already "almost" sorted, in the sense that the displacement of any element in A is less than $M - B$. Give an algorithm that sorts A using $O(n/B)$ I/Os, and argue that it indeed performs only that many I/Os.

(ii) The $O(n/B)$ bound on the number of I/Os is smaller than the $\Omega((n/B)\log_{M/B}(n/B))$ lower bound from Theorem 6.2. Apparently the lower bound does not hold when the displacement of any element in A is less than $M - B$. Explain why the proof of Theorem 6.2 does not work in this case.
NB: You can keep your answer short—a few lines is sufficient—but you should point to a specific place in the proof where it no longer works.

Exercise 6.6 Let $X[0..n-1]$ and $Y[0..n-1]$ be two arrays, each storing a set of n numbers. Let $Z[0..n-1]$ be another array, in which each entry $Z[i]$ has three fields: $Z[i].x$, $Z[i].y$ and $Z[i].sum$. The fields $Z[i].x$ and $Z[i].y$ contain integers in the range $\{0, \dots, n-1\}$; the fields $Z[i].sum$ are initially empty. We wish to store in each field $Z[i].sum$ the value $X[Z[i].x] + Y[Z[i].y]$. A simple algorithm for this is as follows.

```
ComputeSums(X, Y, Z)
for $i \leftarrow 0$ to $n-1$ do
    $Z[i].sum \leftarrow X[Z[i].x] + Y[Z[i].y]$
end for
```

(i) Analyze the number of I/Os performed by *ComputeSums*.

(ii) Give an algorithm to compute the values $Z[i].sum$ that performs only $O(\text{Sort}(n))$ I/Os. At the end of your algorithm the elements in arrays X, Y, and Z should still be in the original order.

Exercise 6.7 In the proof of Theorem 6.2 we assumed that the algorithm only uses the blocks of the input array A, it does not use any additional storage in the external memory. Prove that without this assumption the same asymptotic lower bound holds.

Exercise 6.8 Consider the permutation lower bound of Theorem 6.2.

(i) The lower bound holds when $n \leqslant B(eM/B)^{B/2}$. Show that this condition is satisfied for any practical values of n, M, and B. To this end, pick reasonable values of M and B, and then compute how large n should be to violate the condition.

(ii) Prove that $\Omega(n)$ is a lower bound on the number of I/Os when $n > B(eM/B)^{B/2}$.

Exercise 6.9 The *MergeSort* algorithm can sort any set of n numbers in $O(n \log n)$ time. As we have seen, *MergeSort* can be adapted such that it can sort any set of n numbers in $O((n/B)\log_{M/B}(n/B))$ I/Os. For the special case where the input consists of integers in the range $1, \dots, n^c$, for some fixed constant c, there exist sorting algorithms that run in $O(n)$ time. Would it be possible to adapt such an algorithm so that it can sort any set of n integers in the range $1, \dots, n^c$ in $O(n/B)$ I/Os? Explain your answer.

Chapter 7

Buffer trees and time-forward processing

In this chapter we study time-forward processing, a simple but powerful technique to design I/O-efficient algorithms. Time-forward processing needs an I/O-efficient priority queue. Hence, we first present such a priority queue, which is based on a so-called buffer tree.

7.1 Buffer trees and I/O-efficient priority queues

A *dictionary* is a data structure for storing a set S of elements, each with a *key*, in which one can quickly *search* for an element with a given key. In addition, dictionaries support the *insertion* and *deletion* of elements. One of the standard ways to implement a dictionary in internal memory is by a balanced binary search tree—a red-black tree, for instance—which supports these three operations in $O(\log n)$ time. In external memory a binary search tree is not very efficient, because the worst-case number of I/Os per operation is $O(\log n)$ if the nodes are grouped into blocks in the wrong way. A more efficient alternative is a so-called *B-tree*. B-trees are also search trees, but they are not binary: internal nodes have $\Theta(B)$ children (instead of just two), where the maximum degree is chosen such that a node fits into one memory block. More precisely, the (internal and leaf) nodes in a B-tree store between $d_{\min} - 1$ and $2d_{\min} - 1$ keys. Hence, the degree of the internal nodes is between $d_{\min}$ and $2d_{\min}$; the only exception is the root, which is allowed to store between 1 and $2d_{\min} - 1$ keys. All leaves are at the same depth in the tree, as illustrated in Fig. 7.1. The value $d_{\min}$ should be

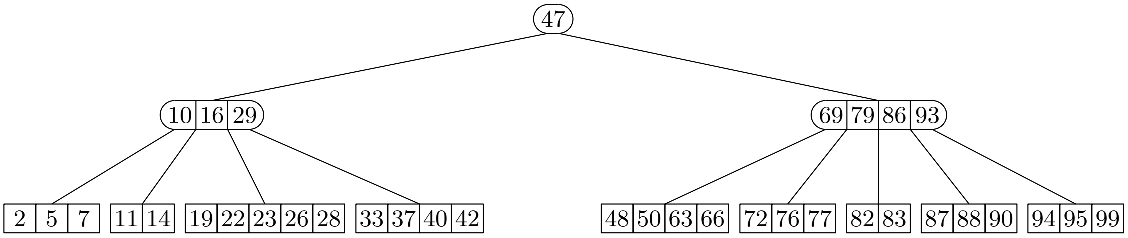

Fig. 7.1: Example of a B-tree. For simplicity the figure shows a B-tree with $d_{\min} = 3$, but in practice $d_{\min}$ is typically (at least) about one hundred.

chosen as large as possible but such that any node still fits into one block in external memory. This means that $d_{\min} = \Theta(B)$.

Because B-trees are balanced—all leaves are at the same level—and the internal nodes have degree $\Theta(B)$, B-trees have depth $O(\log_B n)$. Hence, a search operation can be done in $O(\log_B n)$ I/Os. Also insertions and deletions can be performed with only $O(\log_B n)$ I/Os. The change in the base of the logarithm as compared to binary search trees makes B-trees much more efficient in external memory.

The $O(\log_B n)$ worst-case number of I/Os per operation (search, insert, or delete) is essentially optimal. However, in many applications we have to perform many operations, and we care more about the total number of I/Os than about the worst-case number of I/Os for an individual operation. In such cases the *buffer tree* can be a more efficient solution. The buffer-tree technique can be used for several different purposes. Here we describe how to apply it to obtain an I/O-efficient priority queue.

The basic buffer tree. Let S be a set of elements, where each element $x \in S$ has a key $\mathit{key}(x)$. Suppose we are given a sequence of n operations $\mathit{op}_0, \ldots, \mathit{op}_{n-1}$ on S. Typically the sequence contains a mixture of insert-, delete- and query-operations; the type of query operations being supported depends on the particular structure being implemented by the buffer tree. The goal is to process each of the operations and, in particular, to answer all the queries. However, we do not have to process the operations one by one: we are allowed to first collect some operations and then execute them in a *batched* manner. Next we describe how a buffer tree exploits this. First, we ignore the queries and focus on the insert- and delete-operations. After that we will show how to deal with queries for the case where the buffer tree implements a priority queue.

The basic buffer tree—see also Fig. 7.2—is a tree structure $\mathcal{T}_{\text{buf}}$ with the following properties:

(i) $\mathcal{T}_{\text{buf}}$ is a search tree with respect to the keys in S, with all leaves at the same level.

(ii) Each leaf of $\mathcal{T}_{\text{buf}}$ contain $\Theta(B)$ elements and fits into one block.

(iii) Each internal node of $\mathcal{T}_{\text{buf}}$, except for the root, contains between $d_{\min} - 1$ and $4d_{\min} - 1$ elements, where $d_{\min} = \Theta(M/B)$; the root contains between 1 and $4d_{\min} - 1$ elements.

(iv) Each internal node ν has a *buffer* $\mathcal{B}_\nu$ of size $\Theta(M/B)$ associated to it, which stores operations that have not yet been fully processed. Each operation has a *time-stamp*, which indicates its position in the sequence of operations: the first operation gets time stamp 0, the next operation gets time stamp 1, and so on.

If we consider only properties (i)–(iii) then a buffer tree is very similar to a B-tree, except that the degrees of the internal nodes are much larger: instead of degree $\Theta(B)$ they have degree $\Theta(M/B)$. Thus a single node no longer fits into one block. This means that executing a single operation is not very efficient, as even accessing one node is very costly. The buffers solve this problem: they collect operations until the number of still-to-be-processed operations is large enough that we can afford to access the node. Besides the buffer tree $\mathcal{T}_{\text{buf}}$ itself, which is stored in external memory, there is one block b_{buf} in internal memory that is used to collect unprocessed operations. Once b_{buf} is full, the operations in it are inserted into $\mathcal{T}_{\text{buf}}$. After the operations from b_{buf} have been inserted into $\mathcal{T}_{\text{buf}}$ we start filling up b_{buf} again, and so on. Inserting a batch of $\Theta(B)$ operations from b_{buf} into $\mathcal{T}_{\text{buf}}$ is done as follows.

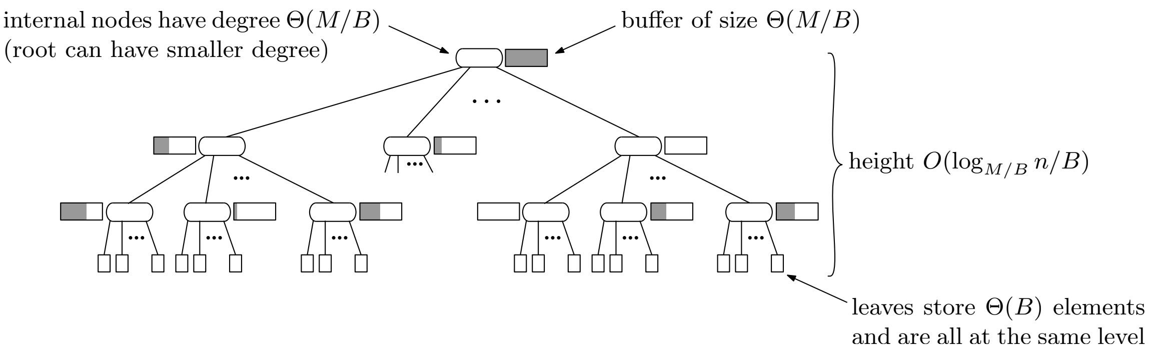
internal nodes have degree $\Theta(M/B)$
(root can have smaller degree)
buffer of size $\Theta(M/B)$
height $O(\log_{M/B} n/B)$
leaves store $\Theta(B)$ elements
and are all at the same level

Fig. 7.2: A buffer tree. The grey bars inside the buffers indicate how full they are. Note that the buffer of the root is completely full. Thus the next insertion into the buffer will cause it to be flushed.

- If the buffer $\mathcal{B}_{root}$ of the root still has space left to store the operations in b_{buf}, these operations are written to $\mathcal{B}_{root}$.
- If the buffer of the root overflows, it is *flushed*. This means that it contents are pushed down into the buffers of the appropriate children of the root. These buffers could in turn overflow, which causes them to be flushed as well, and so on. More precisely, flushing an internal node ν is done recursively as follows.
  - Load the buffer $\mathcal{B}_\nu$ into internal memory and sort the operations by their key.
  - Simultaneously scan the sorted list of operations and the keys stored in ν, and push the operations down into the appropriate children. (The child to which an operation involving key x should be pushed is the next node on the search path to x.) Pushing an operation down into a child means that we add the operation to the buffer of that child. This is done in a batched manner: for each child μ we collect the operations that should be pushed down into μ in groups of $\Theta(B)$ operations, which we write as one block to $\mathcal{B}_\mu$. (The last block of operations to be pushed down into μ could contains fewer than $\Theta(B)$ operations.)
  - For each child of ν whose buffer has overflown due to the operations being pushed down from ν, flush its buffer using the same procedure.

  For internal nodes that are just above the leaf level, the flushing procedure is slightly different. For such a node ν, we load its buffer into internal memory, together with all the leaves below ν. (Recall that ν has degree $\Theta(M/B)$ and leaves have size B, so all of this fits in internal memory.) Let S' be the set of elements stored in these leaves. We then perform the operations from $\mathcal{B}_\nu$ on S'. Finally, we construct a subtree for the new set S', which replaces the old subtree (consisting of ν and the leaves below it).

This finishes the sketch of the buffer tree. We have swept several important details under the rug. In particular, when flushing a node just above leaf level and constructing a subtree for the new set S', we may find that S' contains too many or too few elements to keep properties (ii) and (iii). Thus we have to re-balance the tree. This can be done within the same I/O-bounds; we omit the details.

How many I/Os does a buffer tree perform to execute all operations from the given sequence? Clearly the process of flushing a buffer is quite expensive, since it requires loading the entire buffer into internal memory. However, flushing also involves many operations, and the number of I/Os needed to load (and write back) a full buffer $\mathcal{B}_\nu$ is $O(|\mathcal{B}_\nu|/B)$. In other words, each operation in $\mathcal{B}_\nu$ incurs a cost of $O(1/B)$ I/Os when it is in a buffer being flushed. When this happens the operation is moved to a buffer at a lower level in the tree, and since the depth of the tree is $O(\log_{M/B}(n/B))$, the total cost incurred by an operation is $O((1/B)\log_{M/B}(n/B))$. The total number of I/Os over all n operations is therefore $O((n/B)\log_{M/B}(n/B)) = O(\textsc{Sort}(n))$. (Again, we sweep several details under the rug, about how nodes just above leaf level are handled and about the re-balancing that is needed.)

An I/O-efficient priority queue based on buffer trees. Now let's see how we can use the buffer-tree technique to implement an I/O-efficient priority queue. Let $S = \{x_1, \dots, x_n\}$ be a set of elements, where each element x_i has a priority *prio*(x_i). A *min-priority queue* on S is an abstract data structure that supports the following operations:[1]

- *Extract-Min*, which reports an element $x_i \in S$ with the smallest priority and removes the element from S.
- *Insert*(x), which inserts a new element x with given priority *prio*(x) into S.

(A max-priority queue is similar, except that it supports an operation that extracts the element of maximum priority.) One way to implement a priority queue in internal memory is to use a heap. Another way is to use a binary search tree, where the priorities play the role of the keys, that is, the priorities determine the left-to-right order in the tree. An *Extract-Min* operation can then be performed by deleting the leftmost leaf of the tree. Since a buffer tree provides us with an I/O-efficient version of a search tree, it seems we can directly use the buffer tree as an I/O-efficient priority queue. Things are not that simple, however, because the smallest element is not necessarily stored in the leftmost leaf in a buffer tree—there could be an insert-operation of a smaller element stored in one of the buffers. Moreover, in many applications it is not possible to batch the *Extract-Min* operations: priority queues are often used an event-driven algorithms that need an immediate answer to the *Extract-Min* operation, otherwise the algorithm cannot proceed. These problems can be overcome as follows.

Whenever we receive an *Extract-Min* we flush all the nodes on the path to the leftmost leaf. We then load the $M/4$ smallest elements from $\mathcal{T}_{\text{buf}}$ into internal memory—all these elements are stored in the leftmost internal node and its leaf children—and we delete them from $\mathcal{T}_{\text{buf}}$. Let S^* be this set of elements. The next $M/4$ operations can now be performed on S^*, without having to do a single I/O: when we get an *Extract-Min*, we simply remove (and report) the smallest element from S^* and when we get an *Insert* we just insert the element into S^*. Because we perform only $M/4$ operations on S^* in this manner, and S^* initially has size $M/4$, the answers to the *Extract-Min* operations are correct: an element that is still in $\mathcal{T}_{\text{buf}}$ cannot become the smallest element before $M/4$ *Extract-Min* operations have taken place. Moreover, the size of S^* does not grow beyond $M/2$, so S^* can indeed be kept in internal memory.

After processing $M/4$ operations in this manner, we empty S^* by inserting all its elements in the buffer tree. When the next *Extract-Min* arrives, we repeat the process: we flush all

[1] Some priority queues also support a *Decrease-Key*(x, Δ) operation, which decreases the priority of the element $x \in S$ by Δ. We will not consider this operation.

nodes on the path to the leftmost leaf, we delete the $M/4$ smallest elements and load them into internal memory, thus obtaining a set S^* on which the next $M/4$ operations are performed.

We need $O((M/B)\log_{M/B}(n/B))$ I/Os to flush all buffers on the path to the leftmost leaf. These I/Os can be charged to $M/4$ operations preceding the flushing, so each operation incurs a cost of $O((1/B)\log_{M/B}(n/B))$ I/Os. This means that the total number of I/Os over n *Insert* and *Extract-Min* operations is still $((n/B)\log_{M/B}(n/B))$. We get the following theorem.

Theorem 7.1 *There is an I/O-efficient priority queue that can process any sequence of n* Insert *and* Extract-Min *operations using $((n/B)\log_{M/B}(n/B))$ I/Os in total.*

7.2 Time-forward processing

Consider an *expression tree* $\mathcal{T}$ with n leaves, where each leaf corresponds to a number and each internal node corresponds to one of the four standard arithmetic operations $+$, $-$, $*$, $/$. Fig. 7.3(i) shows an example of an expression tree. Evaluating an expression tree in linear time is easy in internal memory: a simple recursive algorithm does the job. Evaluating it in an I/O-efficient manner is much more difficult, however, in particular when we have no control over how the tree is stored in external memory. The problem is that each time we follow a pointer from a node to a child, we might have to do an I/O. In this section we describe a simple and elegant technique to overcome this problem; it can evaluate an expression tree in $O(\textsc{Sort}(n))$ I/Os. The technique is called *time-forward processing* and it applies in a much more general setting, as described next.

Let $\mathcal{G} = (V, E)$ be a directed, acyclic graph (DAG) with n nodes, where each node $v_i \in V$ has a label $\lambda(v_i)$ associated to it. The goal is to compute a recursively defined function f on the nodes that satisfies the following conditions. Let $N_{\text{in}}(v_i)$ be the set of in-neighbors of v_i, that is, $N_{\text{in}}(v_i) := \{v_j : (v_j, v_i) \in E\}$.

- When $|N_{\text{in}}(v_i)| = 0$ then $f(v_i)$ depends only on $\lambda(v_i)$. Thus when v_i is a source in $\mathcal{G}$ then $f(v_i)$ can be computed from the label of v_i itself only.
- When $|N_{\text{in}}(v_i)| > 0$ then $f(v_i)$ can be computed from $\lambda(v_i)$ and the f-values of the in-neighbors of v_i.

We will call a function f satisfying these conditions a *local function* on the DAG $\mathcal{G}$. Note that if we direct all edges of an expression tree towards the root then we obtain a DAG; evaluating the expression tree then corresponds to computing a local function on the DAG. Local functions on DAGs can easily be computed in linear time in internal memory. Next we show to do this I/O-efficiently if the vertices of the DAG are stored in topological order in external memory. More precisely, we assume that $\mathcal{G}$ is stored as follows.

Let $v_0, \dots, v_{n-1}$ be a topological order on $\mathcal{G}$. Thus, if $(v_i, v_j) \in E$ then $i < j$. We assume $\mathcal{G}$ is stored in an array $A[0..n-1]$, where each entry $A[i]$ stores the label $\lambda(v_i)$ as well a list containing the indices of the out-neighbors of v_i; see Fig. 7.3(ii). We denote this value and list by $A[i].\lambda$ and $A[i]$.OutNeighbors, respectively. Blocks are formed as illustrated in the figure.

We will use an array $F[0..n-1]$ to store the computed f-values, so that at the end of the computation we will have $F[i] = f(v_i)$ for all i. Now suppose we go over the nodes $v_0, \dots, v_{n-1}$ in order. (Recall that this is a topological order.) In internal memory we could keep a list for each node v_j that stores the already computed f-values of its in-neighbors.

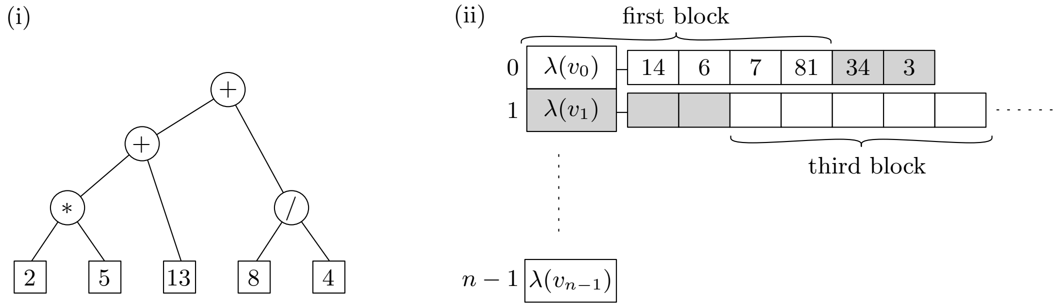

Fig. 7.3: (i) An expression tree. The value of the tree is 25. (ii) Representation of a vertex-labeled DAG in external memory. Vertex v_1 has edges to v_{14}, v_6, v_7, v_{81}, v_{34}, and v_3. The way in which the representation would be blocked for $B = 5$ is also indicated. (The grey elements form the second block.)

When we handle a node v_i, its list will contain the f-values of all its in-neighbors since these have been handled before v_i. Hence, we can compute $f(v_i)$, store the value in $F[i]$ and insert it into the lists of all out-neighbors of v_i. The latter step is expensive in external memory, because we may have to spend an I/O to access each of these lists. Thus we may spend one I/O per edge in $\mathcal{G}$, which means the algorithm would not be I/O-efficient.

In time-forward processing we therefore proceed differently. Instead of keeping separate lists for each of the nodes, we maintain one pool of f-values that we still need in the future. More precisely, we will keep an I/O-efficient min-priority queue $\mathcal{Q}$ storing pairs $(f(v_i), j)$ where j is such that v_j is an out-neighbor of v_i. The value j is the priority of the pair $(f(v_i), j)$; it serves as a time-stamp so that we can extract $f(v_i)$ at the time it is needed. Whenever we compute a value $f(v_i)$, we insert a pair $(f(v_i), j)$ into Q for every out-neighbor v_j of v_i. This leads to the following algorithm.

Algorithm 7.1 Computing a local function on a DAG with time-forward processing.

```
TimeForward-DAG-Evaluation($G$)
1: Initialize an array $F[0..n-1]$ and a min-priority queue $\mathcal{Q}$.
2: for $i \leftarrow 0$ to $n-1$ do
3:     Perform Extract-Min operations on $\mathcal{Q}$ as long as the extracted pairs have
       priority $i$. (This requires one more Extract-Min than there are pairs with
       priority $i$. The extra extracted pair, which has priority $i'$ for some $i' > i$,
       should be re-inserted into $\mathcal{Q}$.)
4:     Compute $f(v_i)$ from $A[i].\lambda$ and the extracted $f$-values; set $F[i] \leftarrow f(v_i)$.
5:     for each $j$ in $A[i]$.OutNeighbors do
6:         Insert the pair $(f(v_i), j)$ into $\mathcal{Q}$.
7:     end for
8: end for
```

How the value $f(v_i)$ is computed in Step 4 depends on the specific function f. Often the computation is straightforward and scanning the f-values of the in-neighbors of v_i is sufficient to compute $f(v_i)$. The computation could also be more complicated, however. If the number of in-neighbors of v_i is so large that their f-values do not all fit into main memory, then computing $f(v_i)$ could even become the I/O bottleneck of the algorithm. Therefore we assume in the theorem below that $f(v_i)$ can be computed in $O(\text{SORT}(1+|N_{\text{in}}(v_i)|))$ I/Os. Note then when each vertex has less than M in-neighbors, which is usually the case in practice, this condition is trivially satisfied.

Theorem 7.2 *Let $\mathcal{G} = (V, E)$ be a* DAG *stored in topological order in external memory, using an adjacency-list representation. Let f be a local function on $\mathcal{G}$ such that each $f(v_i)$ can be computed in $O(\text{SORT}(1+|N_{\text{in}}(v_i)|))$* I/Os *from $\lambda(v_i)$ and the f-values of the in-neighbors of v_i. Then we can compute the f-values of all nodes in $\mathcal{G}$ using $O(\text{SORT}(|V| + |E|))$* I/Os *in total.*

Proof. We first prove that algorithm *TimeForward-DAG-Evaluation* correctly computes f. We claim that when v_i is handled, the f-values of the in-neighbors of v_i are all stored in $\mathcal{Q}$, and that they have the lowest priority of all elements currently in $\mathcal{Q}$. This claim is sufficient to establish correctness.

Define the *level* of a node v_i to be the length of the longest path in $\mathcal{G}$ that ends at v_i. We will prove the claim by induction on the level of the nodes. The nodes of level 0 are the nodes in $\mathcal{G}$ without in-neighbors; for these nodes the claim obviously holds. Now consider a node v_i with level $\ell > 0$. The in-neighbors of v_i have already been handled when we handle v_i, because the nodes are handled in topological order. Moreover, their level is smaller than ℓ and so by the induction hypothesis their f-values have been computed correctly. Hence, these f-values are present in $\mathcal{Q}$ when v_i is handled. Moreover, there cannot be any pair $(f(v_k), i')$ with $i' < i$ in $\mathcal{Q}$, since such a pair would have been extracted when $v_{i'}$ was handled. This proves the claim for nodes at level $\ell > 0$.

To prove the I/O bound, we observe that we perform $O(|V|+|E|)$ *Extract-Min* operations and $O(|V|+|E|)$ *Insert* operations on $\mathcal{Q}$. By Theorem 7.1 these operations can be performed using $O((1/|B|)\log_{M/B}(|V| + |E|))$ I/Os in total. The total number of I/Os needed for the computations of the f-values is $\sum_{i=0}^{n-1} O(\text{SORT}(1 + |N_{\text{in}}(v_i)|))$, which is $O(\text{SORT}(|V| + |E|))$ since $\sum_{i=0}^{n-1} |N_{\text{in}}(v_i)| = |E|$. □

The condition that $\mathcal{G}$ is stored in topological order seems rather restrictive. Indeed, topologically sorting a DAG in external memory is difficult and there are no really I/O-efficient algorithms for it. However, in many applications it is reasonable to assume that the DAG is generated in topological order, or that the DAG has such a simple structure that topological sorting is easy. This is for instance the case when evaluating an expression tree. Note that the second condition of Theorem 7.2, namely that the f-value of a node can be computed in $O(\text{SORT}(1 + |N_{\text{in}}(v_i)|))$ I/Os, is trivially satisfied since $|N_{\text{in}}(v_i)| \leqslant 2$ in a binary expression tree.

Interestingly, evaluating a local function on DAG can also be used for a number of problems on undirected graphs. We give one example of such an application. Recall that an *independent set* of a graph $\mathcal{G} = (V, E)$ is a subset $I \subseteq V$ such that no two nodes in I are connected by an edge. An independent set is called *maximal* if no node can be added to I without losing the independent-set property. (In other words, each node in $V \setminus I$ has an edge to some node in I.) Computing a maximal independent set can be done by applying Theorem 7.2, as shown next. We assume that the input graph $\mathcal{G}$ is stored in an adjacency-list representation as above (including the blocking scheme), except that we do not assume that $v_0, \ldots, v_{n-1}$ is a topological order—indeed, $\mathcal{G}$ is undirected so the concept of topological order does not apply to $\mathcal{G}$. Note that in an adjacency-list representation of an undirected graph, each edge

$(v_i, v_j) \in E$ will be stored twice: once in the adjacency list of v_i and once in the adjacency list of v_j.

Corollary 7.3 *Let $\mathcal{G} = (V, E)$ be an undirected graph, stored in an adjacency-list representation in external memory. We can compute a maximal independent set for $\mathcal{G}$ in $O(\textsc{Sort}(|V| + |E|))$ I/Os.*

Proof. We first turn $\mathcal{G}$ into a directed graph $\mathcal{G}^*$ by directing every edge (v_i, v_j) from the node with smaller index to the node with higher index. Observe that $\mathcal{G}^*$ is a DAG—there cannot be cycles in $\mathcal{G}^*$. Moreover, the representation of $\mathcal{G}$ in fact corresponds to a representation for $\mathcal{G}^*$ in which the nodes are stored in topological order by definition. Thus, to obtain a suitable representation for $\mathcal{G}^*$ we do not have to do anything: we can just interpret the representation of $\mathcal{G}$ as a representation of $\mathcal{G}^*$ (where we ignore the "reverse edges" that $\mathcal{G}$ stores).

Now define a function f on the nodes as follows. (In this application, we do not need to introduce labels $\lambda(v_i)$ to define f.)

- If $|N_{\text{in}}(v_i)| = 0$ then $f(v_i) = 1$.
- If $|N_{\text{in}}(v_i)| > 0$ then

$$f(v_i) = \begin{cases} 0 & \text{if } v_i \text{ has at least one in-neighbor } v_j \text{ with } f(v_j) = 1 \\ 1 & \text{otherwise.} \end{cases}$$

It is easy to see that the set $I := \{v_i : f(v_i) = 1\}$ forms a maximal independent set. Note that f is a local function, and that each $f(v_i)$ can be computed in $O(\textsc{Sort}(1 + |N_{\text{in}}(v_i)|))$ I/Os—in fact, in $O(\textsc{Scan}(1 + |N_{\text{in}}(v_i)|))$ I/Os—from the f-values of its in-neighbors. Hence, we can apply Theorem 7.2 to obtain the result. □

7.3 Exercises

Exercise 7.1 Consider a balanced binary search tree $\mathcal{T}$ with n nodes that is stored in external memory. In this exercise we investigate the effect of different blocking strategies for the nodes of $\mathcal{T}$.

(i) Suppose blocks are formed according to an in-order traversal of $\mathcal{T}$ (which is the same as the sorted order of the values of the nodes). Analyze the minimum and maximum number of I/Os needed to traverse any root-to-leaf path in $\mathcal{T}$. Make a distinction between the case where the internal memory can store only one block and the case where it can store more than one block.

(ii) Describe an alternative way to form blocks, which guarantees that any root-to-leaf path can be traversed using $O(\log_B n)$ I/Os. What is the relation of your blocking strategy to B-trees?

Exercise 7.2 Let $\mathcal{T}$ be a balanced binary search tree $\mathcal{T}$ with n nodes.

(i) Consider a blocking strategy—that is, a strategy to group the nodes from $\mathcal{T}$ into blocks with at most B nodes each— with the following property: for each block b, the nodes in b form a connected part of $\mathcal{T}$. Note that this implies that whenever a root-to-leaf path leaves a block, it will not re-enter that block. Prove that for such a blocking strategy there is always a root-to-leaf path that visits $\Omega(\log_B n)$ blocks.

(ii) Now prove that for *any* blocking strategy, there is a root-to-leaf path that visits $\Omega(\log_B n)$ blocks. *Hint:* Argue that any grouping of nodes into blocks can be modified into a grouping that satisfies the condition in (i), without increasing the number of visited blocks by any root-to-leaf path.

Exercise 7.3 Let $\mathcal{T}$ be a balanced binary search tree. We want to groups the nodes from $\mathcal{T}$ into blocks such that any root-to-leaf path accesses only $O(\log_B n)$ blocks, as in Exercise 7.1(ii). This time, however, your blocking strategy should be cache-oblivious, that is, it cannot use the value B. (Thus you cannot say “put these B nodes together in one block”.) In other words, you have to define a numbering of the nodes of $\mathcal{T}$ such that blocking according to this numbering—putting nodes numbered $1, \ldots, B$ into one block, nodes numbered $B + 1, \ldots, 2B$ into one block, and so on—gives the required property for any value of B.
Hint: Partition $\mathcal{T}$ into subtrees of size $\Theta(\sqrt{n})$ in a suitable manner, and use a recursive strategy.

Exercise 7.4 The I/O-efficient priority queue described above keeps a set S^* in internal memory that contains $M/4$ elements when S^* is created. The next $M/4$ operations are then performed on S^*, after which the elements that still remain in S^* are inserted into the buffer tree (and S^* is emptied). Someone suggests the following alternative approach: instead of only performing the next $M/4$ operations on S^*, we keep on performing *Extract-Min* and *Insert* operations on S^* until $|S^*| = M$. Does this lead to correct results? Explain your answer.

Exercise 7.5 Let S be an initially empty set of numbers. Suppose we have a sequence of n operations $op_0, \ldots, op_{n-1}$ on S. Each operation op_i is of the form $(type_i, x_i)$, where $type_i \in \{Insert, Delete, Search\}$ and x_i is a number. You may assume that when a number x is inserted it is not present in S, and when a number x is deleted it is present. (After a number has been deleted, it could be re-inserted again.) The goal is to report for each of the *Search*-operations whether the number x_i being searched for is present in S at the time of the search. With a buffer tree these operations can be performed in $O(\textsc{Sort}(n))$ I/Os in total. Show that the problem can be solved more directly (using sorting) in $O(\textsc{Sort}(n))$ I/Os, without using a buffer tree.

Exercise 7.6 A *coloring* of an undirected graph $\mathcal{G} = (V, E)$ is an assignment of colors to the nodes of $\mathcal{G}$ such that if $(v_i, v_j) \in E$ then v_i and v_j have different colors. Suppose that $\mathcal{G}$ is stored in the form of an adjacency list in external memory. Assume the maximum degree of any node in $\mathcal{G}$ is $d_{\max}$. Give an algorithm that computes a valid coloring for $\mathcal{G}$ that uses at most $d_{\max} + 1$ colors. Your algorithm should perform $O(\textsc{Sort}(|V| + |E|))$ I/Os.

Exercise 7.7 Let $\mathcal{G} = (V, E)$ be an undirected graph stored in the form of an adjacency list in external memory. A *minimal vertex cover* for $\mathcal{G}$ is a vertex cover C such that no vertex can be deleted from C without losing the cover property. (In other words, there is no $v \in C$ such that $C \setminus \{v\}$ is also a valid vertex cover). Prove that it is possible to compute a minimal vertex cover for $\mathcal{G}$ using only $O(\textsc{Sort}(|V| + |E|))$ I/Os.

Exercise 7.8 Let $\mathcal{G} = (V, E)$ be a directed acyclic graph (DAG) stored in the form of an adjacency list in external memory (with blocks formed in the usual way), with the nodes stored in topological order. Define the *level* of a node $v \in V$ to be the length of the longest path in $\mathcal{G}$ that ends at v. For example, if $|N_{\text{in}}(v)| = 0$ (where $N_{\text{in}}(v)$ is the set of in-neighbors of v), then $\mathit{level}(v) = 0$. Show that it is possible to compute the levels of all vertices in V using only $O(\textsc{Sort}(|V| + |E|))$ I/Os.

Exercise 7.9 Let $\mathcal{G} = (V, E)$ be an undirected graph stored in the form of an adjacency list in external memory (with blocks formed in the usual way), where each node v_i has a label $\lambda(v_i) \in \mathbb{R}$. A *clique* in G is a subset $C \subseteq V$ such that $(u, v) \in E$ for all pairs $u, v \in C$. A clique C is *maximal* if there is no vertex $u \in V \setminus C$ such that $C \cup \{u\}$ is a clique. Show that it is possible to compute a maximal clique in $\mathcal{G}$ using $O(\textsc{Sort}(|V| + |E|)$ I/Os.

Exercise 7.10 Let $\mathcal{G} = (V, E)$ be an undirected graph stored in the form of an adjacency list in external memory (with blocks formed in the usual way), where each node v_i has a label $\lambda(v_i) \in \mathbb{R}$. We call v_i a *local maximum* if its label is larger than the labels of all its (in- and out-)neighbors.

(i) Show that it is possible to compute all local maxima in $\mathcal{G}$ using $O(\textsc{Sort}(|V| + |E|)$ I/Os by applying Theorem 7.2 twice.

(ii) Give a simple, direct algorithm to compute all local maxima using $O(\textsc{Sort}(|V| + |E|))$ I/Os.

Part III

STREAMING ALGORITHMS

Chapter 8

Introduction to streaming algorithms

In many applications data is being collected on-the-fly and at a very high rate. Often is it is undesirable or even impossible to store all the incoming data. Still we would like to compute certain statistics about the data stream.

For example, suppose Google wants to know how often its search engine is being used throughout the year. For this they can simply maintain a counter, which is incremented every time a new search is performed. But now suppose they want to know the number of different users they have. More precisely (and more feasibly[1]), let's say Google wants to know the number of distinct IP addresses from which searches are performed. Then the counter should only be incremented when the search is performed from a new IP address. In principle this is easy to check: just maintain a database containing all IP addresses seen so far. But is this really necessary, or is there a more space-efficient way to count the number of distinct IP addresses?

As another example, suppose we monitor the traffic over a given link in a packet-switching network. In such a network, data to be sent from a given source to a given destination is bundled into packets. The collection of all packets being transmitted for a (source, destination) pair is called a *flow*. Now suppose we notice that a certain link in the network is heavily congested and we want to investigate the reason for this. Then it is interesting to know if there are so-called *heavy hitters*: flows that contribute a significant fraction of the total amount of traffic on the link.[2] To check this, we can monitor the packets being sent over the link, and maintain for each (source IP address, destination IP address) pair a counter indicating how many packets from that flow we have already seen. This way we can easily check if there are flows of size more than, say, 1% of the total traffic on the link. Again the question is: do we really need to store for all flows—that is, for all (source, destination) pairs—the number of packets they consist of? Note that even though the number of different flows can be huge, there must be fewer than 100 flows contributing more than 1% of the total traffic.

Streaming algorithms are algorithms that operate on data streams and that use an amount of storage that is much smaller than the total number of items in the data stream. Often they compute simple statistics on the data like the ones mentioned above. As it turns out, using

[1] although one should not be surprised if Google actually knows who is performing the search, especially when the webcam is on . . .

[2] The term *elephant flow* is also used for such flows.

sublinear storage often means it is impossible to compute the quantity of interest exactly, even when we just want to compute simple statistics such as the number of distinct items in the stream. Streaming algorithms therefore typically compute an approximation of the quantity of interest. Of course we then also want to have guarantees on the maximum error in the reported answer.

8.1 Basic terminology

The input to a streaming algorithm is a sequence $\sigma := \langle a_1, a_2, \ldots, a_m \rangle$ of so-called *tokens*. In the most basic (but still widely applicable) setting, each token is an integer from the universe $[n] := \{0, \ldots, n-1\}$. We will often refer to the elements from $[n]$ as *items*. Typically the size of the universe, n, is known beforehand. The size of the data stream, m, on the other hand, is typically not known; this means that after each token we should be able to report the quantity of interest over the stream seen so far. This basic model is called the *vanilla model*. (The term *time-series model* has been used as well.) The goal of a streaming algorithm is to compute a certain function $\Phi(\sigma)$ on the input stream. The value of $\Phi(\sigma)$ is often a single (real or integral) number, although more complicated outputs are also possible.

In many cases the order in which the tokens arrive is irrelevant for the function we want to compute. We can then represent σ by a *frequency vector* $F_\sigma[0, \ldots, n-1]$, where $F_\sigma[j]$ equals the number of occurrences of item j in the stream σ. Before the stream starts we have $F_\sigma[j] = 0$ for all $j \in [n]$, and the arrival of a token a_i corresponds to an increment of $F_\sigma[a_i]$. We can now also imagine a more general setting, where a token a_i is a pair (j, c) with $j \in [n]$ being an item and c being an integer. The arrival of token $a_i = (j, c)$ then corresponds to setting $F_\sigma[j] := F_\sigma[j] + c$. This model is called the *turnstile model*. When we know that $F_\sigma[j] \geqslant 0$ at all times—intuitively, the number of "departures" cannot exceed the number of "arrivals"—then the model is called the *strict turnstile model*. Sometimes we even know that $c > 0$ for all tokens, in which case the model is called the *cash-register model*.

The efficiency and quality of a streaming algorithm can be measured in several ways.

- The most important efficiency measure is the *amount of storage* used by the algorithm, which can depend on both the size of the stream and the size of the universe. Contrary to what is usually the case in algorithms analysis, the amount of storage used by a streaming algorithm is often quantified by the *number of bits* instead of by the number of elementary objects (integers, pointers, et cetera) being stored.

  If we let $S(m, n)$ denote the worst-case number of bits of storage used by the algorithm for an input stream of size m with items from a universe of size n, then we ideally have $S(m, n) = O(\log(n+m))$. Note that we need $\lceil \log n \rceil$ bits[3] to store a single number from $[n]$, so using $O(\log(n+m))$ bits allows us to store only a constant number of tokens. Many streaming algorithms actually need slightly more storage, for example, $O(\text{polylog}(n+m))$ bits.[4]

- As we will see, it is often impossible to compute $\Phi(\sigma)$ exactly using sublinear space. Thus another important measure is the *quality of the answer*. More precisely, if $\text{ALG}(\sigma)$

[3] All logarithms are assumed to have base 2 unless stated otherwise.

[4] The notation polylog$(n+m)$ is a shorthand for "$\log^k(n+m)$ for some constant k."

is the output of the streaming algorithm then we would like the difference between $\Phi(\sigma)$ and $\text{ALG}(\sigma)$ to be small.

Moreover, many streaming algorithm use *randomization*, and may sometimes report an answer that does not lie within the desired error bounds. In such cases the *probability* with which the algorithm reports a good approximation is also relevant.

To make these quality aspects formal, we say that a randomized algorithm ALG gives an (ε, δ)*-approximation* of Φ, for given $\varepsilon > 0$ and $0 < \delta < 1$, if for any input stream σ we have

$$\Pr\left[|\text{ALG}(\sigma) - \Phi(\sigma)| \leqslant \varepsilon \cdot \Phi(\sigma)\right] \quad \geqslant \quad 1 - \delta.$$

Here we assume that $\Phi(\sigma) \geqslant 0$, which will be the case in all problems we study. Note that the condition on the error is equivalent to $(1-\varepsilon)\cdot\Phi(\sigma) \leqslant \text{ALG}(\sigma) \leqslant (1+\varepsilon)\cdot\Phi(\sigma)$. This is similar to what is used when defining a $(1+\varepsilon)$-approximation for an optimization problem. The difference is that for an optimization problem we only have one side of the condition, since there the value of the computed solution is bounded on one side by the optimal value. Ideally, we have an algorithm that (similar to a PTAS) can be tuned such that, for any given $\varepsilon > 0$ and $\delta > 0$, the condition above is satisfied.

Alternatively, one sometimes uses, for a given $c > 1$, the following condition on the output: $\Phi(\sigma)/c \leqslant \text{ALG}(\sigma) \leqslant c \cdot \Phi(\sigma)$.

- Other efficiency measures are the *time to process each token* and the *time needed to compute the output*; preferably both are $O(\text{polylog}(n+m))$.

In the sequel we will usually focus on the amount of storage and the quality of the answer—we do not explicitly analyze the time needed to process each token and the time needed to compute the output. (In many cases such an analysis would be straightforward, by the way.)

Multi-pass algorithms and sliding-window algorithms. In many applications of streaming algorithms the data stream is generated by some underlying process, but it is never explicitly stored in its entirety. There are also situations, however, where either the data is stored explicitly (in the cloud, say) or where we can run the process generating the stream again. This opens up the possibility of *multi-pass algorithms*, which go over the data stream multiple times. Another interesting variant is to compute the statistic of interest only over a *window* consisting of the last W tokens in the stream. The assumption is that W is too large to store all items from the current window in the internal memory.

In these Course Notes, however, we shall be mostly concerned with single-pass algorithms that do not involve a window.

8.2 Frequent items

Let $\sigma := \langle a_1, a_2, \ldots, a_m\rangle$ be an input stream over a universe $[n]$, and let $F_\sigma[0,\ldots,n-1]$ be the corresponding frequency vector. Let ε be a fixed constant with $0 < \varepsilon < 1$. We say that an item $j \in [n]$ is an *ε-frequent item in σ* (or: *ε-heavy hitter*) if $F_\sigma[j] > \varepsilon m$. We can now define the following problem.

FREQUENT ITEMS: Given a stream σ and a value $0 < \varepsilon < 1$, compute the set $I_\varepsilon(\sigma)$ of ε-frequent items in σ.

When $\varepsilon = 1/2$, an ε-frequent item is also called a *majority item.* A special case of FREQUENT ITEMS is thus the following problem.

> MAJORITY: Given a stream σ, decide if there is a majority item in σ and, if so, report it.

These two problems look very simple. However, even MAJORITY in the vanilla streaming model cannot be solved exactly in sub-linear space, as the following theorem shows.

Theorem 8.1 *Any deterministic streaming algorithm that solves* MAJORITY *exactly on streams with m tokens over the universe $[n]$, where $m \leqslant n/2$, must use $\Omega(m \log(n/m))$ bits of storage, even in the vanilla streaming model.*

Proof. Suppose a deterministic streaming algorithm ALG for MAJORITY uses at most s bits of storage. Then ALG can be in 2^s different states at any point in time. In particular, ALG can be in 2^s different states after processing the first $m/2$ tokens in the stream. (We assume for simplicity that m is even.)

On the other hand, the number of different frequency vectors $F[1,\dots,n]$ that can be generated by a stream of $m/2$ tokens from $[n]$, which equals the number of ways in which we can put $m/2$ balls into n bins, is

$$\binom{n+m/2-1}{n-1} = \binom{n+m/2-1}{m/2} \geqslant \left(\frac{n+m/2-1}{m/2}\right)^{m/2} = 2^{(m/2)\log(2(n+m/2-1)/m)}. \quad (8.1)$$

Hence, when $s < (m/2)\log(2(n+m/2-1)/m)$ there must be two sequences $\sigma_1 := \langle a_1,\dots,a_{m/2}\rangle$ and $\sigma_1' := \langle a_1',\dots,a_{m/2}'\rangle$ with the following property: σ_1 and σ_1' define different frequency vectors but ALG is in exactly the same state after processing σ_1 as it would be after processing σ_1'. Now let $\sigma_2 := \langle a_{m/2+1},\dots,a_m\rangle$ be a sequence such that $\sigma_1 \circ \sigma_2$ contains a certain $j \in [n]$ as a majority item, while j is not a majority item in $\sigma_1' \circ \sigma_2$. (Note that such a sequence σ_2 always exists. Indeed, take some $j \in [n]$ for which $F_{\sigma_1}[j] > F_{\sigma_1'}[j]$, and let exactly $m/2 - F_{\sigma_1}[j] + 1$ of the tokens in σ_2 be equal to j. Then j is a majority item in $\sigma_1 \circ \sigma_2$, but not in $\sigma_1' \circ \sigma_2$.)

Since ALG is deterministic and the state of ALG after processing σ_1 is the same as it would be after processing σ_1', we can conclude that ALG will report the same answer for the stream $\sigma_1 \circ \sigma_2$ as it would for $\sigma_1' \circ \sigma_2$. But this is incorrect, since j is a majority item for $\sigma_1 \circ \sigma_2$, but not for $\sigma_1' \circ \sigma_2$. Hence, any deterministic streaming algorithm that solves MAJORITY exactly must use at least $(m/2)\log(2(n+m/2-1)/m)$ bits of storage. □

Remark 8.2 The condition $m \leqslant n/2$ in the theorem ensures that $\log(n/m) \geqslant 1$. For other cases we can derive similar bounds following the same proof; we only have to change the estimation in Equation (8.1).

Theorem 8.1 is rather discouraging. Somewhat surprisingly, however, we need only very little storage—$O(\log(n+m))$ bits, to be precise—to solve the following variation of MAJORITY: compute an item j such that j is the only possible majority item in σ. Thus, either j is the majority item or σ does not have a majority item, but we do not know which is the case.[5]

[5] If we are allowed to make a second pass over the stream, then it is trivial to check whether j is a majority item or not. This shows that (at least for MAJORITY) two-pass algorithms are much more powerful than single-pass algorithms.

The algorithm is based on the following idea. Suppose that, given a sequence of numbers, we repeatedly delete any pair of adjacent and distinct numbers, until either all remaining numbers are equal or the sequence has become empty. Then the remaining number, if any, is the only candidate for a majority item. Indeed, for each item j that we delete we also delete another item (not equal to j), and since the number of occurrences of a majority item is larger than the total number of occurrences of other items, it is impossible to delete all occurrences of a majority item. It is easy to convert this idea into a streaming algorithm.

We can generalize this approach to computing ε-frequent items. Given a threshold ε, the algorithm will compute a set $I \subseteq [n]$ that is guaranteed to contain all ε-frequent items in the input stream. The algorithm, described in detail in Algorithm 8.1, maintains a set I of items that are candidates to be ε-frequent, where $|I| < 1/\varepsilon$. For each $j \in I$ a counter $c(j)$ will be maintained that serves as an estimation on the number of occurrences of j.

Algorithm 8.1 Streaming Algorithm for Frequent Items

```
Input:
    A stream $\langle a_1, \dots, a_m \rangle$ in the vanilla model.
Initialize:
    $I \leftarrow \emptyset$.
Process($a_i$):
    if $a_i \in I$ then $c(a_i) \leftarrow c(a_i) + 1$
    else
        Insert $a_i$ into $I$ with counter $c(a_i) = 1$
        if $|I| \geqslant 1/\varepsilon$ then
            for all items $j \in I$ do
                $c(j) \leftarrow c(j) - 1$; delete $j$ from $I$ when $c(j) = 0$
            end for
        end if
    end if
Output:
    Report $I$.
```

Next we argue that the set I reported by the algorithm is a superset of the set $I_\varepsilon(\sigma)$ of ε-frequent items. To do so, we interpret the algorithm in a slightly different way, namely as providing an estimate $\widetilde{F}_\sigma[j]$ of the frequency of each item $j \in [n]$ in the stream σ. The estimate is defined as

$$\widetilde{F}_\sigma[j] := \begin{cases} c(j) & \text{if } j \in I \\ 0 & \text{otherwise} \end{cases}$$

The following lemma shows the quality of the estimate.

Lemma 8.3 *For all items $j \in [n]$ we have*

$$\max(0, F_\sigma[j] - \varepsilon m) \quad \leqslant \quad \widetilde{F}_\sigma[j] \quad \leqslant \quad F_\sigma[j].$$

Proof. Consider an item $j \in [n]$ and the value of its counter $c(j)$ during the execution of the algorithm, where we define $c(j) = 0$ when $j \notin I$. Note that we increment the counter $c(j)$

if and only if we encounter item j in the stream (where we interpret line 3 as an increment of $c(j)$). Hence,

$$F_\sigma[j] - (\text{number of decrements to } c(j)) \quad = \quad \widetilde{F}_\sigma[j] \quad \leqslant \quad F_\sigma[j].$$

Whenever we decrement $c(j)$ when processing some token a_i we actually decrement $\lceil 1/\varepsilon \rceil$ counters. Moreover, the total number of decrements over the entire algorithm cannot exceed m; this follows because there are m increments in total and no counter becomes negative. Hence,

$$(\text{number of decrements to } c(j)) \cdot \lceil 1/\varepsilon \rceil \leqslant m.$$

We conclude that $c(j)$ cannot be decremented more than εm times, which implies $\widetilde{F}_\sigma[j] \geqslant F_\sigma[j] - \varepsilon m$. To conclude the proof we note that the algorithm maintains the invariant that $c(j) \geqslant 1$ for $j \in I$ and, by definition, $c(j) = 0$ for $j \notin I$. Hence, $\widetilde{F}_\sigma[j] \geqslant 0$. □

Lemma 8.3 implies that any ε-frequent item j must have $\widetilde{F}_\sigma[j] > 0$. In other words if j is ε-frequent we must have $j \in I$.

Theorem 8.4 *Let ε be a given parameter with $0 < \varepsilon < 1$. There is a streaming algorithm that uses $O((1/\varepsilon)\log(n+m))$ bits of storage and that computes, given a stream $\sigma = \langle a_1, \ldots, a_m \rangle$ in the vanilla model, for each $j \in [n]$ an estimate $\widetilde{F}_\sigma[j]$ with*

$$\max(0, F_\sigma[j] - \varepsilon m) \quad \leqslant \quad \widetilde{F}_\sigma[j] \quad \leqslant \quad F_\sigma[j].$$

In particular, the set of items with a non-zero estimate—these items and their estimates are explicitly stored by the algorithm—contains all ε-frequent items.

Proof. The bound on the estimates provided by the algorithm was already proved in Lemma 8.3. The algorithm maintains a set of at most $1/\varepsilon$ items, each with a counter. To store a single item $j \in [n]$ we need $O(\log n)$ bits. Since the counters cannot exceed m, a single counter needs $O(\log m)$ bits. The bound on the storage follows. □

8.3 Exercises

Exercise 8.1 Consider the following problem in the vanilla streaming model.

> Element Uniqueness: Given a stream $\sigma = \langle a_1, \ldots, a_m \rangle$ over the universe $[n]$, with $m \leqslant n$, decide if all items in σ are distinct.

Either prove that any deterministic streaming algorithm that solves Element Uniqueness exactly must use $\Omega(m \log(2n/m))$ bits in the worst case, or give a deterministic streaming algorithm that solves Element Uniqueness exactly using a sub-linear number of bits. If you give an algorithm, you should also prove its correctness and analyze the number of bits of storage it uses.

Exercise 8.2 Consider the following problem in the vanilla streaming model.

> Two Missing Items: Given a stream $\sigma = \langle a_1, \ldots, a_{n-2} \rangle$ over the universe $[n]$ in which all items in σ are different, compute the items $j_1, j_2 \in [n]$ that are missing from σ.

Note that only streams of length $n-2$ are considered and that all items in the stream are distinct, which implies there are exactly two missing items.

Either prove that any deterministic streaming algorithm that solves Two Missing Items exactly must use $\Omega(n)$ bits in the worst case, or give a deterministic streaming algorithm that solves Two Missing Items exactly using a sub-linear number of bits. If you give an algorithm, you should also prove its correctness and analyze the number of bits of storage it uses.

Exercise 8.3 The problem of finding frequent items in a stream can be generalized to the cash-register model. To this end, we say that an item $j \in [n]$ is *ε-frequent* if $F_\sigma[j] > \varepsilon \cdot \sum_{k=1}^{n} F_\sigma[k]$.

Now consider Algorithm 8.1 from the Course Notes, which computes a subset $I \subseteq [n]$ that contains all α-frequent items in a stream in the vanilla model. Explain how to adapt the algorithm so that it also works in the cash-register model, and argue that it correctly computes a superset of the ε-frequent items. NB: the time to process a token (j,c) should not depend on c.

Exercise 8.4 Consider Algorithm 8.1, which computes a set I containing all ε-frequent items in a stream. Suppose we change the algorithm as follows. If $|I| \geqslant 1/\varepsilon$ then, instead of decrementing the counters $c(j)$ of all $j \in I$, we only decrement the counters of all $j \in I$ with $c(j) = 1$. These counters are thus set to zero and the corresponding items j are removed from I. Someone claims that this algorithm still computes a set I containing all ε-frequent items; after all, the counters $c(j)$ that are not decremented stay closer to the true count of the number of occurrences of item j seen so far.

Prove or disprove this claim. To prove the claim, give a formal proof of the statement. To disprove the claim, give a concrete example of an input stream where the algorithm fails to report an ε-frequent item.

Exercise 8.5 Consider the following sliding-window version of Frequent Items. We are given an infinite stream $\sigma = \langle a_1, a_2, a_3 \ldots \rangle$ over the universe $[n]$ and a window size W, and we want to maintain a set I that contains all ε-frequent items (and possibly other items as well) within the current window. More precisely, after processing an item a_i, the following should hold. Define the window $\sigma(i, W)$ as

$$\sigma(i,W) \ := \ \begin{cases} a_{i-W+1}, \ldots, a_i & \text{if } i \geqslant W \\ a_1, \ldots, a_i & \text{if } i < W. \end{cases}$$

Let m_i denote the size of the window $\sigma(i, W)$; thus $m_i = \min(i, W)$. We define an item j to be *ε-frequent in $\sigma(i, W)$* if the number of occurrences of j in $\sigma(i, W)$ is at least $\varepsilon \cdot m_i$. Our goal is now to maintain a small set I such that, immediately after processing the token a_i we have: if j is an ε-frequent in $\sigma(i, W)$ then $j \in I$.

Describe a streaming algorithm for this problem that uses $O((1/\varepsilon)\log(n+W))$ bits.

Chapter 9

Streaming and randomization

Many problems cannot be solved deterministically in a streaming setting when only sub-linear storage is allowed. Hence, most streaming algorithms are randomized. To be able to analyze a randomized algorithm one needs to know some probability theory. The required knowledge of probability theory for this course, which is rather basic, will be reviewed in Section 9.1. In Section 9.2 we then introduce some basic concepts concerning randomized algorithm, and we describe and analyze a trivial randomized streaming algorithm for approximating the median in a data stream.

9.1 Some probability theory

One of the basic concepts in probability theory is that of a *random variable*. Intuitively, a random variable is a variable whose value depends on the outcome of an experiment for which each possible outcome is assigned a probability. (More formally, a random variable is obtained by assigning a value to each event in a given sample space.) For example, we can roll a die (our experiment) and define a random variable X whose value equals the number of spots on the top face of the die (the outcome of the experiment). Or we can define a random variable Y that is 0 if the number of spots is even and 1 if it is odd. If we assume the die is fair then each of the six possible outcomes is equally likely. We then have a *uniform distribution*. For a fair die we thus have $\Pr[X = i] = 1/6$ for all $i \in \{1, \ldots, 6\}$ and $\Pr[Y = i] = 1/2$ for $i \in \{0, 1\}$.

In the course we will only need *discrete random variables*, which are random variables whose values come from a discrete set. The *expected value* (or *expectation*) of a discrete random variable X, is denoted by $\mathrm{E}[X]$ (or μ_X, or simply μ) and defined as

$$\mathrm{E}[X] := \sum_{x_i} x_i \cdot \Pr[X = x_i],$$

where the sum is taken over all possible values of X. In the example of the fair die we have $\mathrm{E}[X] = \sum_{i=1}^{6} i \cdot (1/6) = 3.5$ and $\mathrm{E}[Y] = 0 \cdot (1/2) + 1 \cdot (1/2) = 0.5$. A useful property of expected values is the following.

Lemma 9.1 (linearity of expectation) *For any two random variables X, Y and any constant c we have*

$$\mathrm{E}[X + Y] = \mathrm{E}[X] + \mathrm{E}[Y] \text{ and } \mathrm{E}[c \cdot X] = c \cdot \mathrm{E}[X].$$

Note that we do *not* always have $\mathrm{E}[X \cdot Y] = \mathrm{E}[X] \cdot \mathrm{E}[Y]$. If the variables X and Y are *independent*, however, then $\mathrm{E}[X \cdot Y] = \mathrm{E}[X] \cdot \mathrm{E}[Y]$ holds. (Intuitively, two random variables X and Y are independent if knowledge of the value of one of them does not change the probability distribution for the other. Formally, two random variables are independent when $\Pr[(X = x_i) \wedge (Y = y_i)] = \Pr[X = x_i] \cdot \Pr[Y = y_i]$ for all x_i and y_i.)

An *indicator random variable* is a random variable associated to a certain event that gets the value 1 if the event takes place and 0 if the event does not take place. Thus the random variable Y in the die example above is an indicator random variable for the event "the outcome of the roll of the die is even". Indicator random variables play an important role in the analysis of many randomized algorithms. By definition, if X is an indicator random variable then $\mathrm{E}[X] = \Pr[X = 1]$.

Often we want to bound the probability that a random variable has a value that is much larger (or much smaller) than its expectation. The *Markov Inequality* and the *Chebyshev Inequality* provide such bounds. Both inequalities apply in fairly general settings. The downside is, however, that the bounds are not very strong—if more is known about the distribution of the random variable, better bounds are often possible.

Lemma 9.2 (Markov Inequality) *Let X be a non-negative random variable with finite expectation μ. Then for any $t > 0$ we have* $\Pr[\ X \geqslant t \cdot \mu\ ] \leqslant 1/t$.

The Markov Inequality can only be used to bound the probability that X exceeds its expected value by some factor t. (Note that the statement becomes useless for $t \leqslant 1$, as then the bound on the probability is at least 1.) Chebyshev's Inequality can also be used to bound the probability that a random value is smaller than its expected value by a certain amount. The inequality is stated in terms of the *standard deviation*, which is defined as $\sqrt{\mathrm{Var}[X]}$, where $\mathrm{Var}[X]$ denotes the *variance* of X. The variance is defined as

$$\mathrm{Var}[X] := \mathrm{E}\left[X^2\right] - (\mathrm{E}[X])^2,$$

or, equivalently, as $\mathrm{Var}[X] := \mathrm{E}\left[(X - \mathrm{E}[X])^2\right]$. Note that for an indicator random variable the formula becomes $\mathrm{Var}[X] := \mathrm{E}[X] - (\mathrm{E}[X])^2$. The standard deviation of a random variable X is often denoted by σ (or σ_X) and so the variance is often denoted by σ^2 instead of by $\mathrm{Var}[X]$.

Lemma 9.3 (Chebyshev Inequality) *Let X be a random variable with finite expectation μ and variance* $\mathrm{Var}[X]$. *Then for any $t > 0$ we have* $\Pr\left[\ |X - \mu| \geqslant t\sqrt{\mathrm{Var}[X]}\ \right] \leqslant 1/t^2$.

To use Chebyshev's Inequality we have to know the expectation and variance of the random variable X we are interested in. In our applications, X is often defined as the sum of other random variables: $X := \sum_i X_i$. Linearity of expectation then helps us to determine $\mathrm{E}[X]$, since it implies $\mathrm{E}[\sum_i X_i] = \sum_i \mathrm{E}[X_i]$. For the variance we can use a similar result, except that we now require the variables to be independent.

Lemma 9.4 (variance of sum of independent variables) *For any two independent random variables X, Y we have*

$$\mathrm{Var}[X + Y] = \mathrm{Var}[X] + \mathrm{Var}[Y].$$

Note that the lemma implies that $\mathrm{Var}\left[\sum_i X_i\right] = \sum_i \mathrm{Var}\left[X_i\right]$ when the X_i are independent.

If $X := \sum_i X_i$ and the X_i are indicator random variables then there is a much stronger bound than the Chebyshev Inequality, as stated in the following lemma. (The constant e in the lemma is the base of the natural logarithm, so $e \approx 2.71828\ldots$)

Lemma 9.5 (Chernoff bound for Poisson trials) *Let $X_1, \ldots, X_k$ be k independent indicator random variables, where $p_i := \Pr\left[X_i = 1\right]$ with $0 < p_i < 1$. Let $X := \sum_{i=1}^{k} X_i$ and let $\mu := \mathrm{E}\left[X\right] = \sum_{i=1}^{k} p_i$. Then for any $\delta > 0$ we have*

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

One way to look at indicator random variables is that we are doing a number of experiments, called *Poisson trials*, which can either be successful ($X_i = 1$) or fail ($X_i = 0$). Lemma 9.5 shows that for Poisson trials the probability that the number of successes is greater than its expectation by some multiplicative factor is exponentially small in the expectation. For example, for $\delta = 2$ we have $e^{\delta}/(1+\delta)^{1+\delta} < 1/2$, so we get $\Pr\left[X > 3\mu\right] \leqslant (1/2)^{\mu}$.

9.2 A randomized streaming algorithm

A *randomized algorithm* is an algorithm whose working not only depends on the input but also on certain random choices made by the algorithm. When designing randomized algorithms we will assume that we have a random number generator *Random*(a, b) available that generates for two integers a, b with $a < b$ an integer r with $a \leqslant r \leqslant b$ uniformly at random. In other words, $\Pr\left[r = i\right] = 1/(b - a + 1)$ for all $a \leqslant i \leqslant b$. We assume that *Random*(a, b) runs in $O(1)$ time (even though it would perhaps be more fair if we could only get a single random bit in $O(1)$ time, and in fact we only have *pseudo-random number generators*). Next we give a simple example of a randomized streaming algorithm.

Let $\sigma := \langle a_1, \ldots, a_m \rangle$ be a stream of m distinct items from the universe $[n]$. The *rank* of an item a_i is 1 plus the number of items in σ that are smaller than a_i:

$$\mathit{rank}(a_i) = 1 + |\{a_{i'} \in \sigma : a_{i'} < a_i\}|.$$

Thus the smallest item has rank 1 and the largest item has rank m. A *median* of σ is a number of rank $\lfloor (m+1)/2 \rfloor$ or $\lceil (m+1)/2 \rceil$. We want to develop a streaming algorithm for the following problem.

> MEDIAN: Given a stream $\sigma = \langle a_1, \ldots, a_m \rangle$ of m distinct items over the universe $[n]$, compute a median of σ.

Unfortunately, computing the exact median with a deterministic streaming algorithm that uses sub-linear storage is impossible; this can be shown with a proof similar to the proof of Theorem 8.1. We therefore set the bar a bit lower: we settle for a randomized algorithm and we shall be satisfied with an item whose rank is close to $(m+1)/2$.

Our randomized algorithm for finding an item close to the median is extremely simple: we pick an index $r \in \{1, \ldots, m\}$ uniformly at random by setting $r \leftarrow$ *Random*$(1, m)$, and report the item a_r from the stream. The algorithm can be implemented in a streaming setting using only $O(\log(n+m))$ bits of storage. Note that the algorithm requires knowledge of the length m of the stream.

To analyze the quality of the output of our algorithm we define $X := \mathit{rank}(a_r)$, where a_r is the item reported by the algorithm. Obviously X depends on the random index r, so X is a random variable. Since r is picked uniformly at random from $\{1, \ldots, m\}$ and we assumed that all a_i are distinct, we have $\Pr[X = i] = 1/m$ for all $1 \leqslant i \leqslant m$. Hence,

$$\mathrm{E}[X] = \sum_{i=1}^{m} i \cdot (1/m) = (m+1)/2.$$

The fact that the expectation of the rank of the reported item is $(m+1)/2$ is what we would hope, but it is not enough. Indeed, if we would always either report the largest item in the stream or the smallest item, each with probability 1/2, then the expectation of the rank is also $(m+1)/2$ but the reported item is always far from the median. We are thus interested in the probability that the reported item has a rank close to $(m+1)/2$.

We define a *ε-approximate median* to be an item a_i with

$$\left\lfloor (\frac{1}{2} - \varepsilon)(m+1) \right\rfloor \leqslant \mathit{rank}(a_i) \leqslant \left\lceil (\frac{1}{2} + \varepsilon)(m+1) \right\rceil,$$

where ε is a parameter with $0 \leqslant \varepsilon \leqslant 1/2$. Let's say that we are satisfied with a (1/4)-approximate median. Since the number of (1/4)-approximate medians in the stream is roughly[1] $m/2$, and the reported item is chosen uniformly at random, our algorithm reports a (1/4)-approximate median with probability 1/2. We can increase the success rate by lowering our standards—for example, if we are satisfied with a (2/5)-approximate median then the success probability increases to 4/5—but there is a better solution: we can boost the success rate with a standard technique, which we describe next.

The median trick. The technique is very simple: we run the algorithm k times, for some parameter $k \geqslant 1$, and then report the median of the answers we get. Now the probability that the median of the answers is a good approximation of the real answer increases as k increases[2]—see Lemma 9.6. In a (single-pass) streaming scenario we cannot re-run the algorithm, of course. Hence, we have to run the different instances of the algorithm in parallel, which increases the storage by a factor k. The resulting algorithm is described in Algorithm 9.1. Next we analyze the new algorithm.

Lemma 9.6 *Algorithm 9.1 uses $O(k \log(n+m))$ bits of storage and it reports a (1/4)-approximate median with probability at least $1 - 2(e/4)^{k/4}$.*

Proof. The algorithm needs to store the set R, which takes $O(k \log m)$ bits of storage, and the set J, which takes $O(k \log n)$ bits. Hence, we use $O(k \log(n+m))$ bits in total.

To bound the success probability of the algorithm, we define for each $i \in \{1, \ldots, k\}$ random variables X_i and Y_i as

$$X_i := \begin{cases} 1 & \text{if } \mathit{rank}(a_{r_i}) > \lceil 3(m+1)/4 \rceil \\ 0 & \text{otherwise} \end{cases}$$

[1] The precise number is $\lceil 3(m+1)/4 \rceil - \lfloor ((m+1)/4 \rfloor + 1$ but we ignore rounding issues for simplicity.

[2] This is similar to the *Law of Large Numbers*, which states that if we perform an experiment a large number of times, the mean of the outcomes converges to the expected value. We cannot report the mean of the k answers, however, because the mean need not be an item from the stream.

Algorithm 9.1 Streaming algorithm for Median that uses the median trick. The algorithm assumes m is known, and only reports something after seeing all m tokens in the stream.

```
Input:
    A stream $\langle a_1, \dots, a_m \rangle$ of $m$ distinct items in the vanilla model.
Initialize:
    Choose a suitable integer $k \geqslant 1$ to obtain the desired success probability.
    Pick $k$ indices $r_1, \dots, r_k$ independently and uniformly at random from $\{1, \dots, m\}$.
    Set $R \leftarrow \{r_1, \dots, r_k\}$ and $J \leftarrow \emptyset$, where $R$ and $J$ are considered multi-sets.
Process($a_i$):
    if $i \in R$ then
        $J \leftarrow J \cup \{a_i\}$
    end if
Output:
    Return the median of the set $J$.
```

and

$$Y_i := \begin{cases} 1 & \text{if } \mathit{rank}(a_{r_i}) < \lfloor (m+1)/4 \rfloor \\ 0 & \text{otherwise.} \end{cases}$$

We also define $X := \sum_{i=1}^{k} X_i$ and $Y := \sum_{i=1}^{k} Y_i$. Now suppose the item we report is not a (1/4)-approximate median. Then either its rank is greater than $\lceil 3(m+1)/4 \rceil$, or its rank is smaller than $\lfloor (m+1)/4 \rfloor$. In the former case more than half of the items in J must have rank greater than $\lceil 3(m+1)/4 \rceil$, and so $X > k/2$. Similarly, in the latter case we have $Y > k/2$. We bound the probabilities of these events using Lemma 9.5 (Chernoff bound for Poisson trials), as shown next.

Observe that $\mathrm{E}[X_i] = \mathrm{E}[Y_i] = 1/4$ because r_i is chosen uniformly at random. Hence,

$$\mathrm{E}[X] = \mathrm{E}\left[\sum_{i=1}^{k} X_i\right] = \sum_{i=1}^{k} \mathrm{E}[X_i] = \sum_{i=1}^{k} \frac{1}{4} = k/4.$$

Lemma 9.5 thus gives

$$\Pr[X > k/2] = \Pr[X > 2\,\mathrm{E}[X]] \leqslant \left(\frac{e}{4}\right)^{k/4}$$

Similarly, $\Pr[Y > k/2] \leqslant (e/4)^{k/4}$. Hence,

$$\Pr[\text{the algorithm reports a (1/4)-approximate median}] \geqslant 1 - 2(e/4)^{k/4}.$$

□

Lemma 9.6 implies that setting $k = 20$ will give a success probability of over 71%, while setting $k = 40$ already gives a success probability of over 95%. In fact, by setting k sufficiently large we can obtain any desired success probability.

Theorem 9.7 *Let σ be a stream of m distinct items over the universe $[n]$. For any $\delta > 0$, there is a streaming algorithm that uses $O(\log(1/\delta)\log(n+m))$ bits of storage and that reports a (1/4)-approximate median from σ with probability at least $1 - \delta$.*

Proof. Algorithm 9.1 needs to store the set R, which takes $O(k \log m)$ bits of storage, and the set J, which takes $O(k \log n)$ bits. Hence, we use $O(k \log(n+m))$ bits in total. It remains to observe that we can get the desired success probability by taking $k := \lceil 8 \log(2/\delta) \rceil$, since this implies that $(e/4)^{k/4} \leqslant \delta/2$. □

It is important to understand that the bound $1-\delta$ in Theorem 9.7 on the success probability holds for *any* input stream—we do *not* assume that the input stream is random. The randomness is enforced because *the algorithm* picks the indices r_i at random.

Remark 9.8 The technique above is called the *median trick* because we take the median of the outcomes of several independent runs of the algorithm, not because it was applied to the median problem. Indeed, we can apply the median trick to other problems as well.

9.3 Exercises

Exercise 9.1 Prove that for $m < n/2$ any deterministic streaming algorithm that solves MEDIAN exactly must use $\Omega(m \log(n/m))$ bits in the worst case.

Exercise 9.2 The streaming algorithm for MEDIAN (Algorithm 9.1) reports a (1/4)-approximate median with probability at least $1-\delta$. Now suppose we want an algorithm that reports a (1/10)-approximate median with probability at least 0.95. Show how to pick the value of k in the initialization such that the algorithm reports an (1/10)-approximate median with the desired probability, where you should pick k as small as possible.

Exercise 9.3 Suppose we have a randomized streaming algorithm ALG whose goal is to estimate some function $\Phi(\sigma)$ of an input stream σ, where $\Phi(\sigma) > 3$. Let $B(n,m)$ be the number of bits of storage used by ALG, where m is the length and n is the size of the underlying universe. Suppose furthermore that we know that ALG outputs a value $\widetilde{\Phi}(\sigma)$ such that $\mathrm{E}\left[\widetilde{\Phi}(\sigma)\right] = \Phi(\sigma)$ and $\mathrm{Var}\left[\widetilde{\Phi}(\sigma)\right] = (1/3) \cdot \Phi(\sigma)$.

(i) Show that there is a constant c with $0 < c < 1$ such that

$$\Pr\left[|\widetilde{\Phi}(\sigma) - \Phi(\sigma)| \geqslant c \cdot \Phi(\sigma)\right] \leqslant 1/6.$$

(ii) Describe a streaming algorithm (using ALG as a subroutine) that, given a parameter $\delta > 0$, computes an estimate $\widehat{\Phi}(\sigma)$ such that $\Pr\left[|\widehat{\Phi}(\sigma) - \Phi(\sigma)| \leqslant c \cdot \Phi(\sigma)\right]$ with probability at least $1-\delta$, where c is the constant you determined in (i).

Prove that $\widehat{\Phi}(\sigma)$ indeed has the desired accuracy with probability at least $1-\delta$. Also analyze the amount of storage used by your algorithm.

Exercise 9.4 The streaming algorithm for MEDIAN assumes that m, the length of the stream, is known beforehand. Adapt the algorithm to the case where m is not known beforehand. In other words, after receiving any item a_i, your algorithm should report a (1/4)-approximate median of $\langle a_1, \dots, a_i \rangle$ with probability at least $1-\delta$. Argue that your algorithm indeed reports a (1/4)-approximate median with the desired probability.

Exercise 9.5 Give a 3-pass streaming algorithm that solves MEDIAN exactly and that, with probability at least 0.95, uses $O(\sqrt{m}\log n)$ bits of storage. Prove that your algorithm has the required properties. *Hint:* Use a random sampling approach. In the analysis the following inequality may be useful: $(1-1/k)^k < 1/2$ for all $k \geqslant 1$.

Exercise 9.6 Give a multi-pass streaming algorithm that solves MEDIAN exactly and that uses $O(\log n)$ bits of storage. The expected number of passes of your algorithm should be $O(\log n)$.

Exercise 9.7 Suppose that we want to compute a (1/10)-approximate median in a stream σ of m distinct items, and that we have a streaming algorithm that uses $O(\log(n+m))$ bits of storage and returns a (1/10)-approximate median with probability at least 0.05. Moreover, we know that the rank of the returned token never exceeds $\lceil m/2 \rceil$.

Explain how to boost the success probability of the algorithm so that it returns a (1/10)-approximate median with probability at least 0.95, and analyze the number of bits of storage used by your algorithm.

Exercise 9.8 Let $\sigma = \langle a_1, \ldots, a_m \rangle$ be a stream of m distinct items in the vanilla model. We wish to compute an element of rank $m/4$ in σ. Since this is hard to do exactly, we are satisfied with an item a_i such that $m/8 \leqslant \mathit{rank}(a_i) \leqslant 3m/8$.

(i) Give a streaming algorithm for this problem with the following properties: the probability that the rank of the returned item lies in the correct range is 218/512, the probability that the rank of the returned item is too small is 169/512, and the probability that the rank of the returned item lies is too large is 125/512. Prove that your algorithm has the desired properties and analyze its storage requirements.
NB: You may ignore rounding issues, and assume that an element chosen uniformly at random from the stream has probability 1/4 to lie in the correct range, and probability 1/8 and 5/8 that it is too small resp. too large.

(ii) Describe how to boost the success probability of your algorithm: present an algorithm that, for a given value $\delta > 0$, returns an item whose rank lies in the correct range with probability at least $1-\delta$, and analyze the storage requirements of your algorithm.

Chapter 10

Hashing-based streaming algorithms

In this chapter we consider two streaming algorithms that use hashing to achieve a suitable "randomization" of the input. The first algorithm deals with the problem of counting the number of distinct items in a stream, the second deals with the problem of finding frequent items.

10.1 Counting the number of distinct items

We want to develop a streaming algorithm for the following problem.

> Distinct Items: Given a stream $\sigma := \langle a_1, \ldots, a_m \rangle$ in the vanilla model over the universe $[n]$, count the number of distinct items in σ.

Using the proof technique from Theorem 8.1, it is not hard to show that Distinct Items cannot be solved exactly using sub-linear space. Hence, we have to settle for an approximate answer. We will actually have to go one step further, by also allowing for randomization. Thus our goal is give an (ε, δ)-approximation algorithm for Distinct Items.

A first solution. The idea of our approach is as follows. Suppose that the number of distinct items in σ is D. Suppose furthermore that the D items would be chosen uniformly at random from $[n]$, and consider the maximum number $j_{\max}$ in the stream. Trivially, when $D = n$ then $j_{\max} = n - 1$. On the other hand, when $D = 1$—we have $a_i = j$ for all $1 \leqslant i \leqslant m$, where j is chosen uniformly at random from $[n]$—then the probability that $j_{\max} = n - 1$ is very small. Thus when $j_{\max} = n - 1$ the probability is high that σ contains many distinct items. Similarly, when $j_{\max} \leqslant n/2$ then we would expect that σ contains only a few distinct items, because an item drawn uniformly at random from $[n]$ has probability $1/2$ to be greater than $n/2$. Thus $j_{\max}$, which is trivial to compute with a streaming algorithm, can tell us something about the expected number of distinct items in the stream.

However, the argument only works if the items are chosen uniformly at random from $[n]$. This means that for certain input streams the algorithm will always have a large error, which is not what we want: we want to the probability to have a large error to be small *for any input stream.* To achieve this, we use the following trick: we map each item $j \in [n]$ to a random item $h(j) \in [n]$ by using a suitable hash function h, and we work with the stream $\langle h(a_1), \ldots, h(a_m)\rangle$. Algorithm 10.1 describes the (extremely simple) algorithm in detail.

Algorithm 10.1 Streaming Algorithm for Distinct Items

```
Input:
    A stream $\langle a_1, \ldots, a_m \rangle$ in the vanilla model.
Initialize:
    Construct a suitable random hash function $h : [n] \to [n]$. Set $j_{\max} \leftarrow -1$.
Process($a_i$):
    if $h(a_i) > j_{\max}$ then
        $j_{\max} \leftarrow h(a_i)$
    end if
Output:
    Compute the largest integer $i^*$ such that $j_{\max} > (1 - \frac{1}{2^{i^*}})n$. Return $2^{i^*+\frac{1}{2}}$.
```

Lemma 10.1 *Let D be the number of distinct items in σ, and let $\widetilde{D}$ denote the estimate returned by the algorithm. Then*

$$\Pr\left[D/(4\sqrt{2}) \leqslant \widetilde{D} \leqslant 4\sqrt{2}D\right] \geqslant \frac{1}{2}$$

Proof. In the following proof we assume that the hash function h is completely random. More precisely, we assume that $h(j)$ is chosen uniformly at random from $[n]$ for each $j \in$ independently. For each integer i and each $j \in [n]$ we define the indicator random variable $X_{i,j}$ as

$$X_{i,j} := \begin{cases} 1 & \text{if } h(j) > (1 - \frac{1}{2^i})n \\ 0 & \text{otherwise} \end{cases}$$

Let J be the set of all distinct items in σ and define

$$X_i := \sum_{j \in J} X_{i,j}.$$

In other words, X_i is the number of items in J whose hash value is greater than $(1 - \frac{1}{2^i})n$. Note that $X_i > 0$ if and only if $i^* \geqslant i$, where i^* is defined as in Algorithm 10.1. (Thus $\widetilde{D} = 2^{i^*+\frac{1}{2}}$.) Moreover, since X_i is integral we have $X_i > 0$ if and only if $X_i \geqslant 1$.

We first prove that $\Pr\left[\widetilde{D} > 4\sqrt{2}D\right] \leqslant 1/4$. To this end observe that $\widetilde{D} > 4\sqrt{2}D$ if and only if $X_s \geqslant 1$, where s is the smallest integer such that $2^{s+\frac{1}{2}} > 4\sqrt{2}D$. To bound the probability that $X_s \geqslant 1$ we use that each $h(j)$ is chosen uniformly at random from $[n]$. Hence

$$\Pr[X_{s,j} = 1] = \Pr\left[h(j) > \left(1 - \frac{1}{2^s}\right)n\right] = 1/2^s.$$

This implies

$$\mathrm{E}[X_s] = \mathrm{E}\left[\sum_{j \in J} X_{s,j}\right] = \sum_{j \in J} \mathrm{E}[X_{s,j}] = \sum_{j \in J} \Pr[X_{s,j} = 1] = D/2^s,$$

where the second equality follows from linearity of expectation. Markov's Inequality, together with the definition of s, now gives

$$\Pr[X_s \geqslant 1] = \Pr\left[X_s \geqslant \frac{2^s}{D} \cdot \mathrm{E}[X_s]\right] \leqslant \frac{D}{2^s} \leqslant \frac{1}{4}.$$

Next we show that $\Pr\left[\widetilde{D} < D/(4\sqrt{2})\right] \leqslant 1/4$. Thus we have to bound $\Pr[i^* < t]$, where t is the largest integer such that $2^{t+\frac{1}{2}} \leqslant D/(4\sqrt{2})$. Observe that $i^* < t$ if and only if $X_t = 0$, so we want to bound $\Pr[X_t = 0]$. We already saw that $\mathrm{E}[X_t] = D/2^t$. Hence,

$$\Pr[X_t = 0] \leqslant \Pr\left[|X_t - \mathrm{E}[X_t]| \geqslant D/2^t\right].$$

Recall *Chebyshev's Inequality*, which states that for a random variable Z, and any $k > 1$, we have

$$\Pr\left[|Z - \mathrm{E}[Z]| \geqslant k\sqrt{\mathrm{Var}[Z]}\right] \leqslant 1/k^2,$$

where $\mathrm{E}[Z]$ and $\mathrm{Var}[Z]$ denote the expectation and variance of Z, respectively. Since the variables $X_{t,j}$ are independent, we have

$$\mathrm{Var}[X_t] = \mathrm{Var}\left[\sum_{j \in J} X_{t,j}\right] = \sum_{j \in J} \mathrm{Var}[X_{t,j}].$$

Moreover, since the $X_{t,j}$ are indicator random variables we have

$$\mathrm{Var}[X_{t,j}] = \mathrm{E}[X_{t,j}] - (\mathrm{E}[X_{t,j}])^2 < \mathrm{E}[X_{t,j}] = 1/2^t.$$

Hence, $\mathrm{Var}[X_t] < D/2^t$. Applying Chebyshev's Inequality now gives

$$\Pr[X_t = 0] \leqslant \Pr\left[|X_t - \mathrm{E}[X_t]| \geqslant D/2^t\right] = \Pr\left[|X_t - \mathrm{E}[X_t]| \geqslant \sqrt{D/2^t} \cdot \sqrt{\mathrm{Var}[X_t]}\right] \leqslant 2^t/D.$$

Plugging in the definition of t we conclude that $\Pr[X_t = 0] \leqslant 1/4$. This finishes the proof. □

In the proof above we assumed that the hash function h is completely random, that is, we assumed that $h(j)$ is chosen uniformly at random from $[n]$ for each $j \in [n]$ independently. How can we achieve this? Even if we assume—which is what we will do—that we have a function *Random*$(0, n-1)$ available that generates a number uniformly at random from $[n]$, then we still have a problem. The reason is that each time we encounter j in the stream, we need to use the same hash value $h(j)$, and we cannot afford to store for each j a randomly chosen value $h(j)$. Fortunately we do not need h to be completely random. The proof works as long as h is *pairwise independent*, meaning that $h(j)$ and $h(j')$ are independent for any two j, j'. A pairwise independent hash function h can be obtained by picking two numbers $a, b \in [n]$ uniformly at random and defining $h(j) := (aj + b) \mod n$. Thus, in the initialization phase we pick a and b uniformly at random from $[n]$, store a and b as well as n, and then throughout the algorithm use $h(j) := (aj + b) \mod n$.

We conclude that the algorithm can be implemented such that it needs to store only four numbers, namely a, b, n, and $j_{\max}$. This leads to the following lemma.

Lemma 10.2 *Algorithm 10.1 can be implemented using $O(\log n)$ bits of storage.*

An improved solution. The bounds in Lemma 10.1 are rather weak: even when we allow the estimate to be a factor $4\sqrt{2}$ away from the real number of distinct items, the success probability is still only 1/2. Of course we can get a higher success probability if we increase the approximation factor—it's easy to adapt the computations in the proof to this end. However, better results can be obtained using the median trick: run the algorithm t times in parallel and return the median of the computed estimates. This leads to the following result.

Theorem 10.3 *Let σ be a stream of m distinct items over the universe $[n]$. Let D denote the number of distinct items in σ. For any $\delta > 0$, there is a streaming algorithm that uses $O(\log(1/\delta)\log n)$ bits of storage and that reports an estimate $\widetilde{D}$ of D such that*

$$\Pr\left[D/(4\sqrt{2}) \leqslant \widetilde{D} \leqslant 4\sqrt{2}D\right] \geqslant 1-\delta.$$

Proof. Similar to the proof of Lemma 9.6. □

10.2 A sketch for frequent items

Recall that an ε-frequent item in a stream $\sigma = \langle a_1, \ldots, a_m\rangle$ in the vanilla model is an item $j \in [n]$ with $F_\sigma[j] > \varepsilon m$. In other words, an ε-frequent item is an item that occurs more than εm times. In Chapter 8 we gave a streaming algorithm that computes for a given given $\varepsilon > 0$ a superset of the set of all ε-frequent items. The algorithm could also be viewed as providing, for a given parameter $\varepsilon > 0$ that determines the accuracy, for each $j \in [n]$ an estimate $\widetilde{F_\sigma}[j]$ such that

$$\max(0, F_\sigma[j] - \varepsilon m) \quad \leqslant \quad \widetilde{F_\sigma}[j] \quad \leqslant \quad F_\sigma[j]. \tag{10.1}$$

The algorithm was described for the vanilla model, but it can easily be adapted to the cash-register model. Recall that in the cash-register model each token a_i is a pair (j, c), where $j \in [n]$ is an item and $c > 0$ is an integer indicating the increment to the frequency $F_\sigma[j]$. In the cash-register model, the number of tokens in the stream is not necessarily equal to the sum of the frequencies. We therefore have to adapt the definition of ε-frequent item as well as the guarantee on the estimates given in (10.1). To this end we define $||F_\sigma||_1 := \sum_{j\in[n]} |F_\sigma[j]|$, and we call an item ε-frequent when $F_\sigma[j] > \varepsilon \cdot ||F_\sigma||_1$. When adapting the algorithm from Chapter 8 to the cash-register model we obtain estimates $\widetilde{F_\sigma}[j]$ such that

$$\max(0, F_\sigma[j] - \varepsilon \cdot ||F_\sigma||_1) \quad \leqslant \quad \widetilde{F_\sigma}[j] \quad \leqslant \quad F_\sigma[j].$$

Thus the algorithm finds a superset of the set of ε-frequent items. Unfortunately, the algorithm from Chapter 8 cannot be adapted to the turnstile model, where we can also have tokens (j, c) with $c < 0$.

In this chapter we discuss a different algorithm that yields similar estimates to the ones above. More precisely, it can output for any $j \in [n]$ an estimate $\widetilde{F_\sigma}[j]$ such that

$$F_\sigma[j] \quad \leqslant \quad \widetilde{F_\sigma}[j] \quad \leqslant \quad F_\sigma[j] + \varepsilon \cdot ||F_\sigma||_1. \tag{10.2}$$

The new algorithm will be a so-called (linear) sketch, which has several advantages. For instance, the new algorithm also works in the turnstile model. The downside of the new algorithm is that it uses more storage and that it is randomized.

Sketches. A streaming algorithm can be viewed as an algorithm that computes, given an input stream σ, a "data structure" $\mathcal{D}(\sigma)$ that can be used to approximate certain statistics over the stream. In the algorithm from Chapter 8, for example, $\mathcal{D}(\sigma)$ is a set I of at most $\lfloor 1/\varepsilon \rfloor$ items $j \in [n]$ with for each item $j \in I$ a counter c_j; the statistic that can be estimated is the frequency of any given item. In certain applications it is desirable to combine the data structures $\mathcal{D}(\sigma_1)$ and $\mathcal{D}(\sigma_2)$ for two streams σ_1 and σ_2 to obtain the data structure $\mathcal{D}(\sigma_1 \circ \sigma_2)$ that we would get for $\sigma_1 \circ \sigma_2$. If this is possible we call $\mathcal{D}(\sigma)$ a *sketch* of the stream σ.

The streaming algorithm from Chapter 8 is not a sketch: we cannot combine the sets I_1 and I_2 computed for streams σ_1 and σ_2 into a suitable set $I_{1,2}$ of size at most $\lfloor 1/\varepsilon \rfloor$ that provides the desired estimates for the stream $\sigma_1 \circ \sigma_2$. Below we present a sketch for estimating frequencies.

The Count-Min sketch. The idea behind the sketch is to randomly group the items into k groups $G_0, \ldots, G_{k-1}$. More precisely, we assign each item $j \in [n]$ to group $G_{h(j)}$, where $h(j) \in [k]$ is chosen uniformly at random. Define the frequency $F_\sigma[G_i]$ of group G_i in a stream σ as $F_\sigma[G_i] := \sum_{j \in G_i} F_\sigma[j]$. Note that we can easily compute the group frequencies $F_\sigma[G_i]$ by a streaming algorithm that maintains k counters. Now suppose we use $F_\sigma[G_{h(j)}]$ as an estimate of $F_\sigma[j]$. Since items are assigned to groups uniformly at random, we have

$$\mathrm{E}\left[F_\sigma[G_i]\right] = (1/k) \cdot ||F_\sigma||_1 \quad \text{for all } 0 \leqslant i < k.$$

This implies that the expected error in our estimates is at most $\varepsilon \cdot ||F_\sigma||_1$ when we set $k := \lceil 1/\varepsilon \rceil$. However, achieving an error that is small in expectation is not good enough: we also want the success probability—the probability that the error is within the required bounds—to be large. We therefore have to make some adjustments to the algorithm.

First, we double the value of k to get a reasonable success probability. To boost the success probability even further we then use an approach similar to the median trick: we run the algorithm t times in parallel, for a suitable value of t, where each run uses its own independently chosen hash function h_s. We then take the best of the estimates provided by these runs. Recall that in the strict turnstile model all frequencies are non-negative. This means that $F_\sigma[G_{h(j)}]$ cannot underestimate $F_\sigma[j]$. Hence, the best estimate for some $F_\sigma[j]$ is given by the smallest estimate provided by any of the t runs of the algorithm. We thus arrive at the algorithm described in Algorithm 10.2.

The following theorem states bounds on the performance of the Count-Min sketch.

Theorem 10.4 *Let σ be a stream in the strict turnstile model with items from the universe $[n]$, and let $\varepsilon > 0$ and $\delta > 0$ be two given parameters. Then the Count-Min Sketch uses $O((1/\varepsilon)\log(1/\delta)\log m_{\max})$ bits of storage, where $m_{\max}$ is the maximum total frequency at any time during the algorithm, and it provides for each $j \in [n]$ an estimate $\widetilde{F_\sigma}[j]$ such that*

$$F_\sigma[j] \quad \leqslant \quad \widetilde{F_\sigma}[j] \quad \leqslant \quad F_\sigma[j] + \varepsilon \cdot ||F_\sigma||_1$$

with probability at least $1 - \delta$.

Proof. The algorithm uses a table with $tk = O((1/\varepsilon)\log(1/\delta))$ counters, where each counter uses $O(\log m_{\max})$ bits to store a group frequency. This proves the storage bound.

To prove the bounds on the error and success probability, consider the estimate $\widetilde{F_\sigma}[j]$ for a given item $j \in [n]$. For $0 \leqslant s < t$, define

$$X_s := C[s, h_s(j)] - F_\sigma[j].$$

Algorithm 10.2 The Count-Min Sketch.

```
Input:
    Parameters $\varepsilon$ and $\delta$ determining the accuracy and success probability.
    A stream $\langle a_1, \dots, a_m \rangle$ in the strict turnstile model, where $a_i = (j_i, c_i)$.
Initialize:
    Set $k \leftarrow \lceil 2/\varepsilon \rceil$ and $t \leftarrow \lceil \log(1/\delta) \rceil$.
    Initialize all entries in a table $C[0..t-1][0..k-1]$ to zero.
    Independently pick random hash functions $h_s : [n] \to [k]$ for $0 \leqslant s < t$, each
    from a family of pairwise independent hash functions.
Process$(j_i, c_i)$:
    for $s \leftarrow 0$ to $t-1$ do
        $C[s, h_s(j_i)] \leftarrow C[s, h_s(j_i)] + c_i$
    end for
Output:
    As an estimate of $F_\sigma[j]$, return $\widetilde{F_\sigma}[j] := \min_{0 \leqslant s < t} C[s, h_s(j)]$.
```

In other words, X_s is the error if we estimate $F_\sigma[j]$ by $C[s, h_s(j)]$. Notice that

$$\widetilde{F_\sigma}[j] = F_\sigma[j] + \min_{0 \leqslant s < t} X_s.$$

As already observed, for streams σ in the strict turnstile model we have $F_\sigma[j] \geqslant 0$ for all $j \in [n]$. Hence, $X_s \geqslant 0$ for any s, which implies that $F_\sigma[j] \leqslant \widetilde{F_\sigma}[j]$. To bound the probability that $\min_{0 \leqslant s < t} X_s > \varepsilon \cdot ||F_\sigma||_1$ we first consider a fixed s. For $j' \neq j$, define an indicator random variable $Y_{j'}$ as

$$Y_{j'} := \begin{cases} 1 & \text{if } h(j') = h(j) \\ 0 & \text{otherwise} \end{cases}$$

Then we have

$$\begin{aligned} \mathrm{E}\left[X_s\right] &= \mathrm{E}\left[\textstyle\sum_{j' \neq j} F_\sigma[j'] \cdot Y_{j'}\right] \\ &= \textstyle\sum_{j' \neq j} F_\sigma[j'] \cdot \mathrm{E}\left[Y_{j'}\right] \\ &= \textstyle\sum_{j' \neq j} F_\sigma[j'] \cdot \Pr\left[Y_{j'} = 1\right] \\ &= \textstyle\sum_{j' \neq j} F_\sigma[j'] \cdot (1/k) \\ &\leqslant (1/k) \cdot ||F_\sigma||_1 \\ &\leqslant (\varepsilon/2) \cdot ||F_\sigma||_1. \end{aligned}$$

Using Markov's Inequality, which we can do because X_s is non-negative, we thus get

$$\Pr\left[X_s > \varepsilon \cdot ||F_\sigma||_1\right] < 1/2. \tag{10.3}$$

We report the smallest over all $C[s, h_s(j)]$ as our final estimate, and so the error of the final estimate is too large if and only if $X_s > \varepsilon \cdot ||F_\sigma||_1$ for all $0 \leqslant s < t$. Since the hash functions h_s are chosen independently, and Inequality (10.3) holds for all s, this happens with probability at most $(1/2)^t$. Because $t = \lceil \log(1/\delta) \rceil$, the failure probability is thus at most δ. □

The Count-Min Sketch can also be used in the general turnstile model, where the frequencies can become negative. Note that in this case we do not always have $C[s, h_s(j)] \geqslant F_\sigma(j)$, and so the smallest $C[s, h_s(j)]$ does not necessarily give the best estimate. Hence, in the general turnstile model we report the median of the set $\{C[s, h_s(j)] : 0 \leqslant s < t\}$ rather than the minimum. With this adaptation, one can prove that with probability at least $1 - \delta$ the reported estimate $\widetilde{F_\sigma}[j]$ satisfies

$$F_\sigma[j] - 3\varepsilon \cdot ||F_\sigma||_1 \quad \leqslant \quad \widetilde{F_\sigma}[j] \quad \leqslant \quad F_\sigma[j] + 3\varepsilon \cdot ||F_\sigma||_1.$$

To interpret this results correctly, note that in the general turnstile model the quantity $||F_\sigma||_1$ is no longer the sum of the frequencies of all items; it's the sum of the absolute values of the frequencies. (It's actually not surprising that the error depends on the sum of the absolute values of the frequencies rather than on the sum of the frequencies themselves, since even when the latter sum is zero some form of approximation is still necessary.)

10.3 Exercises

Exercise 10.1 Prove that for $m < n$ any deterministic streaming algorithm that solves Distinct Items exactly must use $\Omega(m \log(n/m))$ bits in the worst case.

Exercise 10.2 Prove Theorem 10.3.

Exercise 10.3 Suppose person A has run the Count-Min sketch algorithm on a stream σ_1 with m_1 items, and person B has run the Count-Min sketch algorithm on a stream σ_2 with m_2 items. The items in both streams come from the universe universe $[n]$.

Now suppose we want to compute the Count-Min sketch for $\sigma_1 \circ \sigma_2$ from the sketch for σ_1 and the sketch for σ_2. Explain under which conditions this is possible, and explain how to compute the sketch for $\sigma_1 \circ \sigma_2$ in case the conditions are met. (Keep your answer short.)

Exercise 10.4 Explain where in the proof of Theorem 10.4 we use the fact that we are working in the strict turnstile model. (In other words, explain why the proof doesn't work in the general turnstile model.)

Exercise 10.5 Consider the CountMin sketch to estimate the frequencies of the items in a stream in the vanilla model. Suppose $\varepsilon = 0.2$ and $\delta = 0.5$ so that in the initialization phase we set $k = 10$ and $t = 1$. Give an example of an input stream σ such that the probability is very high that for at least one of the items $j \in \sigma$ the estimate of its frequency is much larger than its actual frequency. More precisely, give an example such that (for m large enough) the probability that there is an item j with $\widetilde{F_\sigma}[j] - F_\sigma[j] > m/2$ is at least 0.99.