LECTURES 7, 8 and 9

1 Approximation Algorithms

NP-hardness of many combinatorial optimisation problems gives very little hope that we can ever develop an efficient algorithm for their solution. There are two approached that remain. The first one is to solve the problems with non-efficient algorithms. Certainly if problems are of small or moderate size this is a valid approach. This is the approach that you have encountered in earlier courses. Almost all combinatorial optimisation problems can be formulated as integer linear programming (ILP) problems, most of the time using binary (0-1) variables.

Exercise 6.1. Give the ILP-formulation of the combinatorial optimisation problems we encountered during the course: CLIQUE, INDEPENDENT SET, VERTEX COVER, TSP, MST

Having formulated a problem as an ILP problem we can apply the Branch & Bound method, maybe combined with Lagrange Relaxation and/or with smart preprocessing, or Cutting Plane methods, or a combination of the two. Constructing the right ILP-method for solving a given problem is a matter of smart engineering. Some ILP-problems can be solved by just inserting them in an LP/ILP-solver like CPLEX. On others he standard package application takes too long. Especially roostering problems, like classes of universities or schedules of personnel in hospitals, are notorious for being extremely hard to solve, already for small sizes. Solving ILP-problems is an art that can be learned only in practice.

Since you have learned something about ILP-solving, we skip it in this course. For refreshing your memory and for getting a better understanding and some proofs why some of the methods you learned in the past work, you can read Chapters 13 and 14 of [PS].

We choose the other way of coping with NP-hardness of problems by abolishing optimality and settle for approximations of optimal solutions. Hopefully such approximations are good. An enormous amount of research these days in combinatorial optimisation and theoretical computer science is directed in deriving guarantees on the quality of the approximations, in the form of worst-case ratios between the approximate solution value produced and the optimal value.

1.1 0-1 KNAPSACK

Let me first show you how approximation algorithms that look reasonable at first sight, can produce very bad solutions compared to optimal. Consider the 0-1 KNAPSACK problem.

0-1 KNAPSACK.
Instance: A set of n items, each item j has profit $c_j \geq 0$ and weight $w_j \geq 0$, $j = 1, \ldots, n$, and a (weight) capacity of the knapsack b;

Question: Find a subset of the items with maximal total profit and total weight at most b.

A reasonable algorithm seems to be the *greedy* algorithm, which orders all items in non-increasing c_j/w_j ratio (profit per unit weight) and selects them in this order unless the item does not fit given the previous ones selected. In fact this algorithm solves the LP-relaxation of the problem where the first item that does not fit completely is selected only fractionally. The ILP-formulation is

$$\max \sum_{j=1}^{n} c_j x_j$$

$$\text{s.t. } \sum_{j=1}^{n} w_j x_j \leq b$$

$$x_j \in \{0,1\} \; \forall j$$

The LP-relaxation is obtained by relaxing the constraints $x_j \in \{0,1\}$ to $0 \leq x_j \leq 1$ for all $j = 1,\ldots,n$.

Exercise 6.2. Prove that greedy (fractionally including one item) is optimal for the LP-relaxation of 0-1 KNAPSACK.

The following example shows that *greedy* can give an arbitrarily bad solution for 0-1 KNAPSACK:
Take an instance I of the problem with 2 items $c_1 = \varepsilon << 1$, $w_1 = \varepsilon - \varepsilon^2$, $c_2 = b$, $w_2 = b$. Then $c_1/w_1 > c_2/w_2$, hence greedy will first select item 1, but then item 2 does not fit in the remaining capacity of the knapsack. Clearly the optimal solution is to select just item 2. The ratio between the greedy solution value on this instance denoted by $Z^G(I)$ and the optimal solution value denoted by $Z^{OPT}(I)$ is

$$\frac{Z^G(I)}{Z^{OPT}(I)} = \frac{\varepsilon}{b},$$

which can be arbitrarily close to 0 by taking ε arbitrarily small.

Then one is tempted to propose another, also myopic, algorithm that orders the items in non-increasing profit and selects them in this order unless the item does not fit given the previous ones selected. Notice that this algorithm would solve the instance just given optimally. However, consider the following instance I': $n+1$ items, $c_j = b/n$, $w_j = b/n$, $j = 1,\ldots,n$, and $c_{n+1} = b/n + \varepsilon$, $w_1 = b$. This myopic algorithm chooses item $n+1$ and no capacity left for any other item, solution value $Z^M(I') = b/n + \varepsilon$. Clearly the optimal solution selects all items $1,\ldots,n$ yielding value $Z^{OPT}(I') = b$. The ratio

$$\frac{Z^M(I')}{Z^{OPT}(I')} = \frac{b/n + \varepsilon}{b} = \frac{1}{n} + \frac{\varepsilon}{b},$$

which again can be made arbitrarily close to 0 by choosing n arbitrarily large and ε arbitrarily small.

Notice that for instance I' *greedy* would have found the optimal solution. This suggests that greedy works good on instances on which myopic works bad and vice versa. Why

not apply both algorithms to the instances and select the best of the two solutions? How bad can this *best of greedy and myopic* algorithm be? We will prove that it is substantially better than greedy or myopic separately.

Theorem 1. *For every instance I of* 0-1 KNAPSACK *we have* $Z^B(I)/Z^{OPT}(I) \geq \frac{1}{2}$*:*

$$\min_I \frac{Z^B(I)}{Z^{OPT}(I)} \geq \frac{1}{2}. \tag{1}$$

Proof. For any instance I,

$$Z^B(I) \geq \frac{Z^G(I) + Z^M(I)}{2}. \tag{2}$$

Let k be the first item that greedy does not accept, because of lack of remaining capacity. Then clearly

$$Z^{OPT}(I) \leq Z^G(I) + c_k \leq Z^G(I) + \max_j c_j \leq Z^G(I) + Z^M(I). \tag{3}$$

The inequalities (2) and (3) together imply the theorem. □

Thus, this theorem guarantees that we will always, even in the worst-case, find a solution whose value is within $1/2$ of the optimal value.

The proof of this theorem contains all essential ingredients of a worst-case performance analysis:

- For maximisation problems:
  - Find good upper bounds on the optimal solution (*in this case the round-up of the optimal solution of the LP-relaxation, which on its turn is bounded by the value of the greedy heuristic + the profit of the most profitable item*);
  - Find good lower bounds expressible in the upper bounds (*in this case we have two such bounds 1) the profit of the most profitable item as a lower bound on the value of $Z^M(I)$; 2) the average value of the two different solutions as a lower bound on the algorithm's solution value*)
- For minimisation problems:
  - Find good lower bounds on the optimal solution;
  - Find good upper bounds expressible in the lower bounds.

Exercise 6.3. Find an instance on which the algorithm achieves a ratio of $1/2$, implying that the worst-case performance ratio is not better than $1/2$, hence it is in fact equal to $1/2$.

. . . . . **Answer 6.3.** AN EXAMPLE WITH WHICH THE WORST-CASE BOUND OF $1/2$ CAN BE APPROXIMATED ARBITRARILY CLOSELY IS THE FOLLOWING ONE: TAKE $n = 3$ WITH $c_1 = a_1 = 1 + \frac{1}{2}b$, $c_2 = a_2 = c_3 = a_3 = \frac{1}{2}b$, AND b THE CAPACITY. THE PERFORMANCE OF APPROXIMATION ALGORITHM 3 AMOUNTS TO $(\frac{1}{2}b+1)/b$, WHICH GOES TO $1/2$ IF b GOES TO INFINITY. . . . .

We will now show that for 0-1 KNAPSACK we can actually do a lot better! For this purpose we first develop a Dynamic Programming (DP) algorithm for 0-1 KNAPSACK and analyse its running time.

The DP starts with the initial state $(\emptyset, 0)$. In each stage $i+1$ new item $i+1$ is considered. Let M_i be the set of states kept in stage i. $(S,C) \in M_i$ if total profit C can be obtained by the feasible subset $S \subset \{1,\ldots,i\}$, a subset of the first i items, and, among such subsets of the first i items that have total profit C, it is the subset with minimum total weight.

Do the following for every element $(S,C) \in M_i$ kept in stage i. Notice that there are at most $\sum_{j=1}^{n} c_j$ of them.

In stage $i+1$ add at most two states to M_{i+1}: First we tentatively add (S,C), corresponding to rejecting item $i+1$. If there exists already a set $(S',C) \in M_{i+1}$ then compare $\sum_{j\in S} w_j \leq \sum_{j\in S'} w_j$ insert (S,C) in M_{i+1} and evict (S',C) from M_{i+1}, otherwise delete (S,C) from M_{i+1}.

Secondly, check if $\sum_{j\in S} w_j + w_{i+1} \leq b$. If not, go to the next state in M_i. Otherwise, add tentatively $(S\cup\{i+1\}, C+c_{i+1})$ to M_{i+1}, corresponding to selecting item $i+1$. As before, if there exists an element $(S', C+c_{i+1}) \in M_{i+1}$ with $\sum_{j\in S} w_j + w_{i+1} \leq \sum_{j\in S'} w_j$ then insert $(S\cup\{i+1\}), C+c_{i+1})$ in M_{i+1} evicting $(S', C+c_{i+1})$ from M_{i+1}. Otherwise remove $(S\cup\{i+1\}), C+c_{i+1})$ from M_{i+1}.

After n stages the (S,C) element in M_n with the highest C-value is the optimal solution to the 0-1 KNAPSACK instance.

For correctness of this DP algorithm study the proof of Lemma 17.2 in [PS].
In each stage we start with at most $\sum_{j=1}^{n} c_j \leq n \max_j c_j$ states from the previous state, and we tentatively add 2 states. For the first one $((S,C))$ we do at most one comparison if there is already a state with equal C-value in the stage. For the second one $((S\cup\{i+1\}, C+c_{i+1}))$ we have to do two additions $C+c_{i+1}$ and $\sum_{j\in S} w_j + w_{i+1}$ and if accepted at most another comparison. Thus a constant number of operations per state in each stage. At most $n\max_j c_j$ states per stage and exactly n stages. Thus total running time is $O(n^2 \max_j c_j)$.

We notice that this is not polynomial in the size of the input of the 0-1 KNAPSACK instance, since $max_j c_j$ is not polynomial in $\log(\max_j c_j)$, the number of bits to store $\max_j c_j$ in a digital computer: $max_j c_j = 2^{\log(max_j c_j)}$. The algorithm is a *pseudo-polynomial time* algorithm.

Now we propose to make the DP more efficient by truncating the last t digits from every profit c_j, i.e., rounding down every c_j to the nearest integer multiple of 10^t, denoted by $\bar{c}_j$. This may yield a non-optimal solution, but we claim that it is, in the worst-case, not too far from optimal. Let S and S' be the optimal solutions in the original and the

truncated problem instance, respectively. Then

$$\sum_{j\in S} c_j \geq \sum_{j\in S'} c_j \geq \sum_{j\in S'} \bar{c}_j \geq \sum_{j\in S} \bar{c}_j \geq \sum_{j\in S} (c_j - 10^t) \geq \sum_{j\in S} c_j - n10^t.$$

Thus

$$\frac{\sum_{j\in S'} c_j}{\sum_{j\in S} c_j} \geq 1 - \frac{n10^t}{\sum_{j\in S} c_j} \geq 1 - \frac{n10^t}{\max_{j=1,\ldots,n} c_j}.$$

The running time of the DP algorithm on the truncated problem is $O(n^2 \lfloor 10^{-t} \max_j c_j \rfloor)$. Now choose suppose we wish to obtain an approximation ratio of $1-\varepsilon$, then we need to choose t as the smallest integer such that $\frac{n10^t}{\max_{j=1,\ldots,n} c_j} \leq \varepsilon$, hence

$$t = \lfloor \log_{10} \Big(\frac{\varepsilon \max_{j=1,\ldots,n} c_j}{n} \Big) \rfloor.$$

Leading to a running time of

$$O(n^2 \frac{n}{\varepsilon}) = O(n^3 \frac{1}{\varepsilon}).$$

This implies that for any $\varepsilon > 0$ we have an algorithm that achieves a worst-case performance ratio of $1-\varepsilon$ in a running time that is polynomial in the size of the input. Such an algorithm is called a *Polynomial Time Approximation Scheme* (PTAS). In fact, here the running time is also polynomial in $1/\varepsilon$ and in such cases we call it a *Fully Polynomial Time Approximation Scheme* (FPTAS). Thus for 0-1 KNAPSACK we have a FPTAS. We can come arbitrarily close to optimal in time polynomial in the input size and the divergence of the ratio from 1.

This is not achievable for very NP-hard problem. In particular for problems for strongly NP-hard problems an FPTAS is unlikely to exist.

Theorem 2. *Any strongly NP-hard problem does not have a FPTAS unless P=NP.*

Exercise 6.4. Prove the above theorem by proving that every FPTAS can be transformed into a pseudo-polynomial time algorithm by choosing ε appropriately.

1.2 TSP

For TSP the situation is even more dramatic.

Theorem 3. *Unless P=NP, there is no polynomial time approximation algorithm for TSP that can achieve any constant worst-case performance ratio.*

Proof. Suppose by contradiction that such an algorithm with ratio c would exist. We will argue that in that case we would have a polynomial time algorithm to solve HAMILTON CIRCUIT, which can only be true if P=NP.

Take any instance $G = (V,E)$ of HAMILTON CIRCUIT give all edges in E length 1, and all edges not in E length $c|V|+1$. If the graph has a Hamilton Circuit then it must be a TSP-tour of length $|V|$ and the algorithm must give a tour of length at most $c|V|$. But as soon as the algorithm includes one edge not in E it will have a tour-length of at least $c|V|+|V|$. Thus, the algorithm either finds a tour with length $|V|$, corresponding to a Hamilton Circuit in G, or it finds a tour of length greater than $(c+1)|V|$ and since it is c-approximate there is no tour of length less than $|V|+|V|/c > |V|$ and in particular there does not exist a Hamilton Circuit in the graph G.

Thus our algorithm would solve HAMILTON CIRCUIT in polynomial time, which can be true only if P=NP. □.

Notice that in the proof above c can even be any polynomial function of n. So as announced, approximability is dramatic for general TSP.

In [PS] two more positive results have been derived for special classes of TSP, in particular for Δ-TSP, TSP in which the distances satisfy the triangle inequality. I have explained them but do not include in my notes, but instead refer to the text in the book. **Notice that in [PS] the authors don't study the worst-case performance ratio but the worst case relative error.** If they speak of a 1-approximate algorithm A then this means (in a minimisation problem) $\max_I (Z^A(I) - Z^{OPT})/Z^{OPT} \leq 1$ implying that the worst-case performance ratio is $\max_I Z^A(I)/Z^{OPT} \leq 1+1 = 2$.

I emphasize once more the recipe for proving approximation results: in case of minimisation problems, we derive upper bounds on algorithmic solutions and lower bounds on the optimal solution value. In case of maximisation problems we derive lower bounds on algorithmic solution values (like inequality (2) for algorithm B for 0-1 KNAPSACK) and upper bounds on optimal solution values (like inequality (3) for 0-1 KNAPSACK).

1.3 Scheduling

Partly to give you more examples and partly to teach you an enormous class of other combinatorial optimisation problems we will study *scheduling problems*. Such problems are unfortunately not covered in the book [PS].

In a scheduling problem jobs have to be processed on one or more machines. Each job has a processing time, the time needed to process the job on a machine. A *schedule* on a single machine is an assignment of time slots to the jobs, such that the length of the time slot assigned to a job is equal to its processing time, the time slots are non overlapping (each machine can process only one job at a time), and each job is assigned one time slot. In the basic form presented here a feasible schedule is completely determined by the order in which the jobs are processed on the machine. The order then implies for each job j its *starting time* S_j and its completion time C_j.

On multiple machines a schedule consists of an assignment of the jobs to the machines (each job is assigned to one machine) and for each machine a feasible schedule of the

jobs assigned to it, again implying the starting and completion times of the jobs.

As a first example of a scheduling problem consider a problem with n jobs and m machines. The question is to find a feasible schedule of the jobs on the machines such that the makespan, i.e., the earliest time at which all jobs are completed, is minimised.

MAKESPAN.
Instance. Given n jobs and m parallel identical machines, for each job j a processing time p_j, $j = 1, \ldots, n$.
Question. Find a feasible schedule of the jobs on the machines such that the makespan, $\max_{j=1\ldots,n} C_j$, is minimised.

This problem is NP-hard, shown by showing that the decision version is NP-Complete.

Exercise 6.5. Define MAKESPAN-DECISION and prove that it is NP-Complete. *Hint: make the reduction from* PARTITION *to* MAKESPAN *with two machines.* PARTITION *is the problem in which items* $\{1, \ldots, n\}$ *is given, each item* j *having a profit* c_j*, is there a fair division of the items, i.e., is there a subset* $S \subset \{1, \ldots, n\}$ *such that* $\sum_{j \in S} c_j = \frac{1}{2} \sum_{j=1}^{n} c_j$*? (see Chapter 15.7 in [PS]).*

The most simple reasonable algorithm one could think of is *List Scheduling*. It takes the jobs in an arbitrary list, e.g. in the order in which they are specified in the input, and assign every time the next job to the machine which finishes earliest with the jobs assigned to it so far.

As an example consider three jobs $p_1 = p_2 = 1$, $p_3 = 2$ to be scheduled on two machines. List scheduling will assign job 1 to machine 1 and job 2 to machine 2, and then job 3 to machine 1 or 2. The makespan of this schedule is 3. Clearly in this case a makespan of 2 would be optimal in this problem instance. We will show that this is a worst case instance for List Scheduling for MAKESPAN on 2 machines.

Theorem 4. *List scheduling for* MAKESPAN *on m machines has a worst-case performance ratio of* $2 - \frac{1}{m}$*: for every instance I the solution value produced by List Scheduling* $Z^{LS}(I)$ *and the optimal solution value* $Z^{OPT}(I)$*, satisfy*

$$\frac{Z^{LS}(I)}{Z^{OPT}(I)} \leq 2 - \frac{1}{m}.$$

Proof. Notice that the total work (processing time) to be done is $\sum_{j=1}^{n} p_j$. The best we could obtain is if this work could be divided equally over all m machines, yielding a makespan of $\frac{1}{m} \sum_{j=1}^{n} p_j$. This lower bound may not always be achieved but is certainly indeed a lower bound on the optimal makespan:

$$Z^{OPT}(I) \geq \frac{1}{m} \sum_{j=1}^{n} p_j. \tag{4}$$

Also it is easy to see that the longest processing time $p_{\max} = \max_{j=1,\ldots,n} p_j$ is a lower bound on the optimal makespan.

$$Z^{OPT}(I) \geq p_{\max}. \tag{5}$$

For the list scheduling algorithm let job k be the job that finishes last, thus $Z^{LS}(I) = C_k$. The starting time S_k of job k is latest $\frac{1}{m}\sum_{j=1}^{k-1} p_j$, since this is the time job k will start if all machines would take equally long in processing the first $k-1$ jobs. If they do not take equally long then some machine becomes available already earlier for starting job k. Clearly, $C_k = S_k + p_k$

$$\begin{aligned} Z^{LS}(I) = C_k = S_k + p_k &\leq \tfrac{1}{m}\textstyle\sum_{j=1}^{k-1} p_j + p_k \\ &= \tfrac{1}{m}\textstyle\sum_{j=1}^{k} p_j + \left(1 - \tfrac{1}{m}\right) p_k \\ &\leq \tfrac{1}{m}\textstyle\sum_{j=1}^{n} p_j + \left(1 - \tfrac{1}{m}\right) p_{\max}. \end{aligned} \tag{6}$$

Taking (4), (5) and (6) together yields

$$\begin{aligned} Z^{LS}(I) &\leq \tfrac{1}{m}\textstyle\sum_{j=1}^{n} p_j + \left(1 - \tfrac{1}{m}\right) p_{\max} \\ &\leq Z^{OPT}(I) + \left(1 - \tfrac{1}{m}\right) Z^{OPT}(I) \\ &= \left(2 - \tfrac{1}{m}\right) Z^{OPT}(I). \end{aligned}$$

□

The theorem implies that on 2 machines List Scheduling always remains within a multiplicative factor of $3/2$ from optimal. Together with the example instance with 3 jobs given before this leads implies that the worst-case performance ratio for List Scheduling on two machines is in fact exactly $3/2$.

Exercise 6.6. Show by an example that List Scheduling has worst-case ratio exactly $2 - \frac{1}{m}$ for MAKESPAN on m machines.

The problem with List Scheduling is that it may divide all items but the last one perfectly over the machines, and then the last one is a long job. It would be far better to have a small job as last job in the list. This is the idea behind the *Longest Processing Time First Rule* (LPT). This algorithm first orders the jobs according to non-increasing processing time and then apply List Scheduling on this ordered list.

Theorem 5. *LPT for* MAKESPAN *on m machines has a worst-case performance ratio of*

$$\max_I \frac{Z^{LPT}(I)}{Z^{OPT}(I)} \leq \frac{4}{3} - \frac{1}{3m}.$$

Proof. Suppose we have renumbered the jobs such that $p_1 \geq p_2 \geq \ldots \geq p_n$. As before let job k with processing time p_k be the job that finishes last. In the instance on which

LPT achieves the worst-case ratio we may assume that job n is the one that finishes last in the LPT-schedule: if job $k < n$ were the last one then all jobs $j > k$ start later than job k and finish earlier, hence do not effect the LPT-makespan. Thus excluding them from the instance does not effect the LPT-makespan, but they may increase the optimal makespan.

So, job n is the job that finishes last. Let $Z^{OPT}(I)$ be the optimal makespan. Now we distinguish two cases:

CASE 1 $p_n > \frac{1}{3}Z^{OPT}(I)$. In this case, since job n is the job with smallest processing time among all jobs, there are at most 2 jobs on each machine. But LPT gives the optimal solution on instances with at most two jobs per machine. It first puts each of the first m larger jobs on the m machines and then matches job $m+1$ with job m, job $m+1$ with job $m-1$ etc.

CASE 2 $p_n \leq \frac{1}{3}Z^{OPT}(I)$. But then we apply the first few equalities and inequalities in (6) which also applies to LPT, since after ordering the jobs it is List Scheduling:

$$\begin{aligned} Z^{LPT}(I) = C_n = S_k + p_n &\leq \tfrac{1}{m}\textstyle\sum_{j=1}^{n-1} p_j + p_n \\ &= \tfrac{1}{m}\textstyle\sum_{j=1}^{n} p_j + \left(1 - \tfrac{1}{m}\right) p_n. \end{aligned}$$

Given the case $p_n \leq \frac{1}{3}Z^{OPT}(I)$ and given the lower bound (4) on $Z^{OPT}(I)$ we obtain the desired result:

$$\begin{aligned} Z^{LPT}(I) &\leq \tfrac{1}{m}\textstyle\sum_{j=1}^{n} p_j + \left(1 - \tfrac{1}{m}\right) p_n \\ &\leq Z^{OPT}(I) + \left(1 - \tfrac{1}{m}\right) \tfrac{1}{3} Z^{OPT}(I) \\ &= \left(\tfrac{4}{3} - \tfrac{1}{3m}\right) Z^{OPT}(I). \end{aligned}$$

□

Also for this algorithm there is a *tight* example: 2 jobs with processing time 3, and 3 jobs with processing time 2 for the problem on 2 machines. LPT will construct a schedule with makespan 7, whereas 6 is optimal. $7/6 = 4/3 - 1/(3m)$ if $m = 2$.

1.4 A PTAS (Skipped)

We will show that even better performance ratio's than $4/3 - 1/(3m)$ are achievable: for every ε there exists a polynomial time algorithm which has approximation ratio $1 + \varepsilon$.

As before it is based on the principle of rounding. The processing times of the jobs will be rounded down in a way presented later. It will lead to instances with only a fixed number of different processing times. And the trick is, as we will see, that instances with a fixed number of processing times can be solved in polynomial time.

The idea for doing so is bisection search on the question: given time t are m machines sufficient to allow for assigning the jobs to the machines such that no job has more total working load (sum of processing times) than t. This problem is better known under the name BIN PACKING-DECISION: Given n items of size $a_1,\ldots,a_n$ and a bin size b and an integer K, are at most K bins needed to pack all the items such that no bin receives items with total size more than b. We know a lower bound $L = \max\left\{\frac{1}{m}\sum_{j=1}^{n} p_j, \max_j p_j\right\}$ and an upper bound $2L$ on the value of t we need to investigate. We do the bisection search $\lceil \frac{1}{\varepsilon L}$ times to reduce the possible interval to no more than εL.

From now on we assume that we have an estimate t of the optimal makespan on the rounded instance within εL. So let us consider BIN PACKING with k different sizes and bin capacity t. Let n_h be the number of items of the h-th size; hence $\sum_{h=1}^{k} n_h = n$. Let $\mathrm{BINS}(i_1,\ldots,i_k)$ be the number of bins needed to pack i_ℓ items of the ℓ-th size, $\ell = 1,\ldots,k$. We find $\mathrm{BINS}(i_1,\ldots,i_k)$ by recursion. First determine the set $\mathscr{Q}$ of all k-tuples $(q_1,\ldots,q_k)$ that fit in one bin: $\mathrm{BINS}(q_1,\ldots,q_k) = 1$. Then for every initialising the table with $\mathrm{BINS}(q) = 1, \forall q \in \mathscr{Q}$, compute for all $(i_1,\ldots,i_k) \in \{0,1,\ldots,n_1\} \times \{0,1,\ldots,n_2\} \times \ldots \times \{0,1,\ldots,n_k\}$ by

$$\mathrm{BINS}(i_1,\ldots,i_k) = 1 + \min_{q\in\mathscr{Q}} \mathrm{BINS}(i_1 - q_1,\ldots,i_k - q_k).$$

Clearly, computing each entry in this table takes $O(n^k)$ time. There are $O(n^k)$ entries to be computed resulting in an overall running time of $O(n^{2k})$ given any guess of t.

Let us now recapitulate and fill in the details: Given t and a precision ε, jobs with processing times less than $t\varepsilon$ are ignored for the time being. We call them *small jobs*.For all other jobs j, we round the processing time $p_j \in [t\varepsilon(1+\varepsilon)^i, t\varepsilon(1+\varepsilon)^{i+1})$ down to p'_j equal to the left-hand side of the interval. Hence we have no more than $k = \lceil \log_{1+\varepsilon} \frac{1}{\varepsilon} \rceil$ different processing times. And we do not underestimate the makespan by more than a multiplicative factor $1+\varepsilon$.

We solve BIN PACKING-DECISION with $K = m$, $b = t$, and $a_i = t\varepsilon(1+\varepsilon)^i$, $i = 1,\ldots,k$, then we know that the bin-packing in terms of the original processing times gives us a schedule with makespan at most $t(1+\varepsilon)$. The ignored very small items are packed greedily in the leftover spaces in the bins, allowing binsize $t(1+\varepsilon)$ and opening new bins only if needed. Let $\alpha(t,\varepsilon)$ be the number of bins of capacity $t(1+\varepsilon)$ needed. Let $OPT_{bin}(t)$ be the optimal number of bins of capacity t needed to pack all original sizes.

Lemma 1. $\alpha(t,\varepsilon) \le OPT_{bin}(t)$

Proof. If in packing the small items no extra bin was opened the statement must be true since in that case $\alpha(t,\varepsilon)$ is the optimal bin-packing for the rounded-down sized items. If new bins were opened while packing the small items then all bins, except possibly the last one, were filled up till at least size t, implying that the total size of all items is strictly greater than $\alpha(t,\varepsilon)t$ and therefore $OPT_{bin}(t) > \alpha(t,\varepsilon)$. □

Let OPT be the optimal makespan. Then $OPT = \min\{t \mid OPT_{bin}(t) \leq m\}$. Hence the lemma gives that

$$\min\{t \mid \alpha(t,\varepsilon) \leq m\} \leq OPT, \tag{7}$$

and therefore, if we would know $t = OPT$, the algorithm would give us a makespan of $(1+\varepsilon)OPT$.
However, we make an extra error by the bisection search. Let T be the right endpoint of the interval we terminate with. Then $\min\{t \mid \alpha(t,\varepsilon) \leq m\} \in [T - \varepsilon L, T]$. Hence,

$$T \leq \min\{t \mid \alpha(t,\varepsilon) \leq m\} + \varepsilon L,$$

which together with (7) implies

$$T \leq (1+\varepsilon)OPT.$$

As noticed before, using $t = T$ the binpacking based algorithm gives a makespan of at most $(1+\varepsilon)T$. The following theorem follows.

Theorem 6. *The algorithm produces a valid schedule with makespan bounded by*

$$(1+\varepsilon)^2 OPT \leq (1+3\varepsilon)OPT.$$

The running time is $O(n^{2^{\lceil \log_{1+\varepsilon} \frac{1}{\varepsilon} \rceil} \lceil \log_2 \frac{1}{\varepsilon} \rceil})$.

Exercise 6.7. Give an FPTAS for the variant of the makespan problem if there are only 2 machines.

1.5 Variations of scheduling.

There are many variations of scheduling problems. For example instead of minimising the maximum completion time (makespan) we could minimise the average completion time $\frac{1}{n}\sum_{j=1}^{n} C_j$. This problem is solvable in polynomial time by the *Shortest Processing Time First Rule* (SPT). If jobs have weights we could minimise the sum of weighted completion times $\sum_{j=1}^{n} w_j C_j$. This problem is solvable in polynomial time on a single machine by the *Shortest Weighted Processing Time First Rule* (SWPT) (order on non-decreasing p_j/w_j). It is NP-hard on 2 or more machines.

In some scheduling problems jobs have release times r_j, times at which they become available for processing. Then the *flow time* of a job F_j, defined as the completion time minus the release time, $F_j = C_j - r_j$, is a valid measure for quality of service. The objective may then be minimising average flow time $\frac{1}{n}\sum_{j=1}^{n} F_j$ or maximum flow time $\max_{j=1,\ldots,n} F_j$.

Exercise 6.8. Show that for every instance with release times, the schedule that minimises average flow time also minimises average completion time and vice versa.

Exercise 6.9. Show that minimising $\max_{j=1,\ldots,n} F_j$ on a single machine can be solved in polynomial time. And prove that is is NP-hard on 2 or more machines (*Hint: easy*

reduction from makespan).

Minimising $\frac{1}{n}\sum_{j=1}^{n} F_j$ (or minimising $\frac{1}{n}\sum_{j=1}^{n} C_j$) in the presence of release times, is already NP-hard on a single machine. It is easy if preemption of jobs is allowed, meaning that the processing of any job may be interrupted and the job may be resumed later without losing its previous part of the processing. In that case the single machine problem is solved by the *Shortest Remaining Processing Time First Rule* (SRPT).

Scheduling is a fascinating area of research in combinatorial optimization. I advise you to go to the LNMB course Scheduling in Utrecht starting February 1, see the LNMB website http://www.lnmb.nl for further information. The Master and PhD-courses of the LNMB give the credits mentioned for your Master's program.

1.6 Randomised Algorithms and LP-rounding

In this last part on approximation algorithms I introduce two powerful and widely applicable tools in the construction of approximation algorithms. The first one is Randomised algorithms. The algorithms that we have studied so far were all *deterministic algorithms*. Given the input of the problem a deterministic algorithm will always make the same steps and find the same answer. In a randomised algorithm some of the steps made in the algorithm are guided by the outcome of some random experiment, like the flipping of a coin, the throwing of a dice, or, more modernly, the drawing of a random number in the computer. Doing so, the outcome of a randomised algorithm becomes a random variable. We usually study its expectation. The nice fact is that simple randomised algorithms have often surprisingly good approximation behaviour. This is due to the fact that it is much harder to come up with a single bad instance for the algorithm: among all possible realisations of the algorithm there usually are quite some that give very good solutions.

Let me give an extremely easy example of a randomised algorithm and analyse its performance. For this part of the lecture I refer to the scan that you will find on the website.

To improve upon this very simple randomised algorithm, I introduce the next general approximation tool. This concerns the formulation of the problem as an ILP, solve the LP-relaxation and round the optimal LP-solution. Rounding the optimal LP-solution is not always easy but sometimes it is, like in the MAX-SAT problem that we studied. Also for this part I refer to the scan on the website.
An even better algorithm is obtained from the combination of the two previous ones, since the first one is very good for clauses with many literals and the second one for clauses with few literals. Taking the best of the two solutions indeed gives a good approximation ratio. Again see the scan.

Material

[PS] Chapter 17 + the scan of Chapter 16 from V. Vazirani, Approximation Algorithms, Springer Verlag, Berlin, 2003

Exercises:

The Exercises 6.1-6.9 in the text above;

From Chapter 17: Exercises 1, 3, 4(b)
From [V] Chaper 16; Exercises 16.1, 16.2, 16.3*