Optimization and Approximation ENS Lyon

Lecture 12

Lecturer: Alantha Newman *January 3, 2017*

1 Deterministic Rounding via Extreme Point Structure

In this lecture, we show how extreme point structure can be used to design *deterministic rounding* procedures for linear programs.

Let $P = \{\mathbf{x} \in \mathbb{R}^n \mid \mathbf{Ax} \geq \mathbf{b},\ \mathbf{x} \geq 0\}$ be a polyhedron in $\mathbb{R}^n$. Let $\mathbf{A}$ be an $m \times n$ matrix, where $m < n$. Then the total number of constraints is $m + n$. Recall that an *extreme point* of P is a point in P that cannot be expressed as a convex combination of two other points in P. A *basic feasible solution* of P is a point in P that is tight for at least n constraints of which n are linearly independent. (Furthermore, recall that $\mathbf{x}^* \in P$ is an extreme point if and only if it is a basic feasible solution.) Thus, we have the following observation, which is directly implied by the definition of a basic feasible solution.

Claim 1. *A basic feasible solution of $P \in \mathbb{R}^n$ has at least n tight constraints.*

Since $n - m$ of these tight constraints must be nonnegativity constraints, it follows that at least $n - m$ variables have zero value (e.g. the nonbasic variables in a simplex dictionary). This implies that there are at most m variables with nonzero value. Let us state this claim more generally.

Claim 2. *Let $P = \{\mathbf{x} \in \mathbb{R}^n \mid \mathbf{A}_1\mathbf{x} \geq \mathbf{b}_1,\ \mathbf{A}_2\mathbf{x} = \mathbf{b}_2,\ \mathbf{x} \geq 0\}$ be a polyhedron, where m equals the number of rows in $\mathbf{A}_1$ plus the number of rows in $\mathbf{A}_2$. If $\mathbf{x}^*$ is a basic feasible solution of P, then $\mathbf{x}^*$ has at most m nonzero entries.*

These seemingly simple statements have many useful applications.

1.1 Knapsack

We are given a knapsack of capacity B and a set of N items $S = \{s_1, s_2, \ldots, s_N\}$ such that each item s_i has value v_i and requires b_i units of space in the knapsack. The objective of the knapsack problem is to find a subset of S that fits in the knapsack and has the highest total value. An indicator variable x_i is used to denote whether or not item s_i is chosen. Here is a linear programming relaxation for the knapsack problem.

$$\begin{aligned} \max \sum_{s_i \in S} x_i v_i, & \\ \text{subject to: } \sum_{s_i \in S} x_i b_i \leq B, & \\ x_i \leq 1, \quad \text{for all } s_i \in S, & \\ x_i \geq 0, \quad \text{for all } s_i \in S. & \qquad (P_{knapsack}) \end{aligned}$$

The polyhedron $(P_{knapsack})$ has $2N + 1$ constraints in total. Let $\mathbf{x}^*$ be a basic feasible solution for $(P_{knapsack})$. Claim 1 implies that $\mathbf{x}^*$ is tight for at least N constraints. It follows that for at least $N - 1$ items, the corresponding value of x_i^* is integral. So there is at most one item whose corresponding x_i^* value is fractional. Thus, if we were allowed to include one item fractionally in the knapsack, we could find an optimal solution using this relaxation.

2 Scheduling on Unrelated Parallel Machines

Consider the following scheduling problem. There are N jobs and M machines. For each job and each machine, the time required to process job i on machine j is denoted by p_{ij}, where p_{ij} is nonnegative.

The goal is to assign each job to one of the M machines so as to minimize the maximum processing time over all machines. The processing time on a machine is often referred to as the *makespan* and our goal is therefore to minimize the maximum makespan.

An instance of this scheduling problem can be modeled as a complete bipartite graph $K_{N,M}$ where p_{ij} is the edge weight for edge (ij). A valid solution is a subset of edges that has degree exactly one at each vertex corresponding to a job.

Figure 1: The jobs are the red circles on the left and the machines are the blue squares on the right. The edge weights are indicated by the numbers on the left. In this example, an optimal solution has makespan of five.

We can model this scheduling problem with the following integer program. The variable x_{ij} indicates whether or not job i is assigned to machine j.

$$\begin{aligned} & \min \ T \\ \text{subject to: } & \sum_{i=1}^{N} x_{ij} p_{ij} \leq T, \quad \text{for all machines } j : 1 \leq j \leq M, \\ & \sum_{i=1}^{M} x_{ij} = 1, \quad \text{for all jobs } i : 1 \leq i \leq N, \\ & x_{ij} \in \{0, 1\}. \end{aligned} \tag{1}$$

Constraints of the form (1) are called *assignment constraints*, since they force each job to be assigned to exactly one machine in an integer solution. To obtain a linear program, we relax the integrality constraint. Since the unit upper bound on the variables is implied by the assignment constraint, it is redundant and can be removed.

$$\begin{aligned} & \min \ T \\ \text{subject to: } & \sum_{i=1}^{N} x_{ij} p_{ij} \leq T, \quad \text{for all machines } j : 1 \leq j \leq M, \\ & \sum_{i=1}^{M} x_{ij} = 1, \quad \text{for all jobs } i : 1 \leq i \leq N, \\ & x_{ij} \geq 0. \end{aligned} \tag{$P_{makespan}$}$$

We cannot immediately use this linear program to obtain a, say, constant-factor approximation algorithm, because the integrality gap can be arbitrarily large. Consider an input instance where there is only one job with processing time $p_{1j} = M$ for all machines j. The optimal solution will assign a $\frac{1}{M}$-fraction of the job to each machine, so the optimal solution will have value $T = 1$. However, the makespan must be at least M, since each job must be scheduled on a single machine. So the integrality gap is M.

2.1 Improved Lower Bound

To circumvent this problem, we observe that for each job i, the minimum value of p_{ij} over all machines j is also a lower bound on the value of the minimum makespan. In other words:

$$OPT \quad \geq \quad \max_{i \in [1,N]} \min_{j \in [1,M]} p_{ij}.$$

We incorporate this observation via the following preprocessing step, which "prunes" edges from the input graph. For some fixed value of T, we define S_T to be the subset of edges in the complete bipartite graph $K_{M,N}$ whose weights are at most T:

$$S_T \quad = \quad \{(ij) \mid p_{ij} \leq T\}.$$

Now we obtain a modified linear program.

$$\begin{aligned} \sum_{i:(ij) \in S_T} x_{ij} p_{ij} &\leq T, \quad \text{for all machines } j : 1 \leq j \leq M, \\ \sum_{j:(ij) \in S_T} x_{ij} &= 1, \quad \text{for all jobs } i : 1 \leq i \leq N, \\ x_{ij} &\geq 0. \end{aligned} \tag{P_T}$$

We let T^* denote the minimum value of T for which (P_T) is feasible. (We can efficiently find this value using a binary search in the range $[0, W]$, where $W = \sum_{i=1}^{N} \sum_{i=1}^{M} p_{ij}$.) Our next goal is to prove the following theorem.

Theorem 3. *There is an efficient algorithm to produce a schedule with makespan at most* $T^* + \max_{(ij) \in S_{T^*}} p_{ij}$, *which is at most* $2T^* \leq 2OPT$.

Since T^* is a lower bound on the minimum makespan, Theorem 3 states that there is a 2-approximation algorithm.

2.2 Extreme Points of (P_T)

The proof of Theorem 3 is based on several key lemmas about the structure of extreme points of (P_T). The first claim follows directly from Claim 2.

Lemma 4. *An extreme point* $\mathbf{x}^*$ *of* (P_T) *has at most* $M + N$ *nonzero entries.*

Note that the total number of variables is NM, so Lemma 4 states that the support of an extreme point is very sparse.

Definition 5. *A* pseudotree *is a connected graph that contains at most one cycle.*

For example, a spanning tree or a 1-tree (a spanning tree plus one edge) is an example of a pseudotree.

Definition 6. *A graph is a* pseudoforest *if each component is a pseudotree.*

Let $\mathbf{x}^*$ be an extreme point of (P_T) and let $G(\mathbf{x}^*)$ denote the subgraph of $K_{M,N}$ corresponding to the support of $\mathbf{x}^*$ (i.e., an edge $(ij) \in G(\mathbf{x}^*)$ iff $x^*_{ij} > 0$). By Lemma 4, we observe that $G(\mathbf{x}^*)$ has at most $M + N$ edges, so it is a (globally) sparse graph. The following lemma says that, in fact, $G(\mathbf{x}^*)$ is locally sparse (i.e., it is sparse on every component).

Lemma 7. $G(\mathbf{x}^*)$ *is a pseudoforest.*

Proof. $G(\mathbf{x}^*)$ has $M+N$ vertices. By Lemma 4, it has at most $M+N$ edges.

Case 1. $G(\mathbf{x}^*)$ is connected. Then it is a pseudotree, since it is either a spanning tree or a 1-tree.

Case 2. $G(\mathbf{x}^*)$ is not connected. Then we need to show that each component is a pseudotree. Towards a contradiction, let us assume that there is some component C that is not a pseudotree (i.e., it has more edges than vertices). Let $\bar{C}$ denote the rest of the graph $G(\mathbf{x}^*) \setminus C$. Then C and $\bar{C}$ correspond to two restricted scheduling problems. We let $\mathbf{x}^*_C$ and $\mathbf{x}^*_{\bar{C}}$ denote the restrictions of $\mathbf{x}^*$ to these two instances. (Note that $\mathbf{x}^* = \mathbf{x}_C + \mathbf{x}_{\bar{C}}$.)

But since C is not a pseudotree, $\mathbf{x}^*_C$ cannot be an extreme point for the instance C, since this would violate Case 1. Therefore, we can write $\mathbf{x}^*_C$ as a convex combination of two other points in the respective polyhedron:

$$\mathbf{x}^*_C \quad = \quad \lambda \mathbf{y}_1 + (1-\lambda)\mathbf{y}_2,$$

for $\lambda \in (0,1)$. Thus, we have:

$$\mathbf{x}^* \quad = \quad \lambda(\mathbf{x}_{\bar{C}} + \mathbf{y}_1) + (1-\lambda)(\mathbf{x}_{\bar{C}} + \mathbf{y}_2).$$

This contradicts the assumption that $\mathbf{x}^*$ is an extreme point. So we can conclude that $G(\mathbf{x}^*)$ is a pseudoforest. □

2.3 Rounding an Optimal Extreme Point

Now let $\mathbf{x}^*$ denote an optimal extreme point of (P_{T^*}). We will use the fact that $G(\mathbf{x}^*)$ is a pseudoforest to round $\mathbf{x}^*$ and obtain a schedule with value at most $T^* + \max_{(ij)\in S_{T^*}} p_{ij}$.

First, for all edges $(ij) \in S_{T^*}$ where $x^*_{ij} = 1$, assign job i to machine j. Next, we define a *leaf* as a degree-1 vertex in $G(\mathbf{x}^*)$. (Note that a vertex corresponding to a job i cannot be a leaf.) Now do the following until there are no more leaves: Choose a leaf (i.e., machine) and assign to it any job that is adjacent in $G(\mathbf{x}^*)$. When there are no more leaves, if there are unassigned jobs remaining, find an even cycle (there is at most one such cycle per component) and find a perfect matching to match remaining jobs to machines.

3 Bin Packing

We now turn to the *bin packing* problem. We are given access to an infinite number of unit-capacity bins and a set of n items with sizes indicated by s_i, where $0 < s_1 \leq s_2 \leq \cdots \leq s_n \leq 1$. The goal is to pack all of the items using as few bins as possible. We say that an instance $\mathcal{I}$ of bin packing corresponds to the set of item sizes in the input (i.e., $\mathcal{I} = \{s_1, s_2, \ldots, s_n\}$). We use $SIZE(\mathcal{I})$ to denote the sum of all the item sizes in instance $\mathcal{I}$, and we use $OPT(\mathcal{I})$ to denote the optimal number of bins required to pack the items in instance $\mathcal{I}$. It is easy to see that $\lceil SIZE(\mathcal{I}) \rceil \leq OPT(\mathcal{I})$. For example, suppose we are given items with sizes $\{\frac{1}{4}, \frac{1}{3}, \frac{2}{3}, \frac{5}{8}\}$. We can pack these four items into two unit-capacity bins. The size of this instance is $1\frac{7}{8}$, so two bins is optimal.

The bin packing hard is NP-hard via a reduction from the partition problem. (Partition problem: Is there a partition of the items in $\mathcal{I}$ such that the size of the items in each partition equals $SIZE(\mathcal{I})/2$?) So it is NP-hard to decide if we need two bins or three. This rules out a PTAS, but there is a so-called asymptotic PTAS which uses $(1+\epsilon)OPT(\mathcal{I}) + 1$ bins and runs in time $n^{O(\frac{1}{\epsilon})}$ time [DLVL81]. There is also an efficient algorithm that packs the items in instance $\mathcal{I}$ into $OPT(\mathcal{I}) + \log^2 OPT(\mathcal{I})$ bins. This algorithm crucially uses extreme point structure and we present these ideas by focusing on a special case.

3.1 3-Partition

The *3-partition* problem is a special case of bin packing. An instance is a set of item sizes $\mathcal{I} = \{\frac{1}{4} < s_1 \leq s_2 \leq \cdots \leq s_n < \frac{1}{2}\}$. In this restricted input instance, we can see that in an optimal solution, each bin contains two or three items (except for maybe one bin). Moreover, every pair of items can be packed together in a bin. The goal is to find the minimum number of unit-capacity bins needed to pack all items (i.e., items s_i, s_j and s_k can be placed in a bin if $s_i + s_j + s_k \leq 1$).

We now describe an integer program for the 3-partition problem. Let $\mathcal{P}$ be the set of valid patterns. In other words, let pattern $p = \{s_i, s_j, s_k\}$. Then $p \in \mathcal{P}$ if $s_1 + s_j + s_k \leq 1$. Let the variable $x_p = 1$ if pattern p is chosen and $x_p = 0$ otherwise. If a pattern p is chosen (i.e., if $x_p = 1$) then one bin is reserved for the items in pattern p. After relaxing to allow $x_p \in [0, 1]$, we have the following linear program:

$$\begin{aligned} \min \sum_{p \in P} x_p & \\ \sum_{p: s_i \in p} x_p \geq 1, & \quad \text{for all items } s_i \\ x_p \geq 0. & \end{aligned} \tag{P_{bin}}$$

We can also specify an input instance as a set of (unique) item sizes $\frac{1}{4} < s_1 < s_2 < \cdots < s_k < \frac{1}{2}$, where each item size s_i has a multiplicity b_i (i.e., there are b_i items with size s_i in the input instance). Let t_{ip} denote the number of times an item with size s_i appears in pattern p (i.e., an item size s_i can appear in a pattern once, twice or three times). Then we can also use the following linear program:

$$\begin{aligned} \min \sum_{p \in P} x_p & \\ \sum_{p: s_i \in p} t_{ip} x_p \geq b_i, & \quad \text{for all item sizes } s_i \\ x_p \geq 0. & \end{aligned} \tag{Q_{bin}}$$

In both linear programs, there are at most $O(n^3)$ variables, since there are at most this many valid patterns. In (P_{bin}), there are n constraints, not counting nonnegativity constraints. In (Q_{bin}), the number of such constraints equals the number of different item sizes. Let $OPT_{LP}(\mathcal{I})$ denote the optimal value of (Q_{bin}) on instance $\mathcal{I}$. We note that $SIZE(\mathcal{I}) \leq OPT_{LP}(\mathcal{I}) \leq OPT(\mathcal{I})$. Now we present a rounding algorithm.

There are two key tools in the rounding algorithm:

1. Grouping items and rounding up sizes of items to create fewer item sizes.
2. Sparse extreme point structure.

3.1.1 Grouping

We partition the items in instance $\mathcal{I}$ into groups $\mathcal{G}(\mathcal{I}) = \{G_1, G_2, \dots\}$ using the following procedure. We will use $\mathcal{G}$ as a function mapping each item to a group, so that the notation $\mathcal{G}(\mathcal{I}')$ is also valid when $\mathcal{I}' \subset \mathcal{I}$.

GROUP-ITEMS($\mathcal{I}$)

$i = 1$; $j = 1$.
$G_j = \emptyset$; $\mathcal{G}(\mathcal{I}) = \emptyset$.

While $i \leq n$:

1. Add s_i to group G_j; $i = i + 1$.
2. If $SIZE(G_j) \geq 2$, then:
   (a) Add G_j to $\mathcal{G}(\mathcal{I})$.
   (b) $j = j + 1$; $G_j = \emptyset$.

Output $\mathcal{G}(\mathcal{I})$.

Suppose at the end of this procedure, we have r groups, $\mathcal{G}(\mathcal{I}) = \{G_1, G_2, \ldots, G_r\}$. For each group G_j, we have $SIZE(G_j) \geq 2$, except possibly for the last group G_r. Moreover, if we remove the largest element from a group G_j, then the size is strictly less than 2. Thus, each group has at most nine elements (since $s_i > \frac{1}{4}$).

The next step is to modify the instance by discarding some items from instance $\mathcal{I}$. These items are placed in the set $\mathcal{D}(\mathcal{I})$, and they will be packed separately. We discard the items in the following groups:

- G_r,
- G_{r-1},
- any group G_j where $|G_j| > |G_{j+1}|$.

DISCARD-ITEMS($\mathcal{I}$, $\mathcal{G}(\mathcal{I})$)

$\mathcal{I}' = \mathcal{I}$; $\mathcal{D}(\mathcal{I}) = \emptyset$.
$r = |\mathcal{G}(\mathcal{I})|$; $j = 1$.

While $j < r - 1$:

    If $|G_j| > |G_{j+1}|$:

        Remove all items in group G_j from $\mathcal{I}'$ and add them to $\mathcal{D}(\mathcal{I})$.

Remove all items in groups G_r and G_{r-1} from $\mathcal{I}'$ and add them to $\mathcal{D}(\mathcal{I})$.
Output $\mathcal{I}', \mathcal{D}(\mathcal{I})$.

Claim 8. *The number of discarded elements in $\mathcal{D}(\mathcal{I})$ is at most* 54.

Proof. The number of items in groups G_{r-1} and G_r is at most 18. For each group G_j, $5 \leq |G_j| \leq 9$. So the number of groups for which $|G_j| > |G_{j+1}|$ is at most four and these groups contain at most 36 items in total. Therefore, at most 54 items are discarded. (Note that this bound can be improved to 22 items, since we really only need to remove $|G_j| - |G_{j+1}|$ items from group G_j. But for the sake of simplicity, we discard the whole group.) □

Finally, we modify the instance $\mathcal{I}'$ by rounding up the size of each item in group G_j to the size of the largest item in group G_j.

ROUND-UP-ITEMS($\mathcal{I}', \mathcal{G}(\mathcal{I}')$)

$j = 1$.

For each group $G_j \in \mathcal{G}(\mathcal{I}')$:

1. $s_{max} = \max\{s_i : s_i \in G_j\}$.
2. For all $s_i \in G_j : s_i = s_{max}$.

Output $\mathcal{I}'$.

Even though the item sizes in instance $\mathcal{I}'$ are rounded-up or increased, due to the space vacated by the discarded items, we can bound the value $OPT_{LP}(\mathcal{I}')$ by $OPT_{LP}(\mathcal{I})$.

Claim 9. *$OPT_{LP}(\mathcal{I}') \leq OPT_{LP}(\mathcal{I})$ and $SIZE(\mathcal{I}') \leq SIZE(\mathcal{I})$.*

Proof. We can map items in group G_j to the "slots" allocated by the linear program for the items in group G_{j+1}. (The fact that we discarded the elements in $\mathcal{D}(\mathcal{I})$ from $\mathcal{I}$ implies that this mapping is feasible.) □

Now we are ready to state our main algorithm.

$\mathcal{D} = \emptyset$. (Global variable)

BIN-PACK($\mathcal{I}$)

1. Use GROUP-ITEMS($\mathcal{I}$) to find $\mathcal{G}(\mathcal{I})$.
2. Partition instance $\mathcal{I}$ into $\mathcal{I}'$ and $\mathcal{D}(\mathcal{I})$ using DISCARD-ITEMS($\mathcal{I}, \mathcal{G}(\mathcal{I})$).
3. Round-up item sizes for items in $\mathcal{I}'$ using ROUND-UP-ITEMS($\mathcal{I}', \mathcal{G}(\mathcal{I}')$).
4. Add items in $\mathcal{D}(\mathcal{I})$ to the discard set $\mathcal{D} = \mathcal{D} \cup \mathcal{D}(\mathcal{I})$.
5. Solve (Q_{bin}) on $\mathcal{I}'$.
6. Pack $\lfloor x_p \rfloor$ bins using pattern p. Call the packed items $\mathcal{I}_1$ and unpacked items $\mathcal{I}_2$.
7. BIN-PACK($\mathcal{I}_2$).
8. Pack items in $\mathcal{D}$ using any algorithm.

3.1.2 Extreme Point Structure

The reason we round up the item sizes is to create fewer item sizes, which in turn implies that the support of the extreme point of (Q_{bin}) on instance $\mathcal{I}'$ is sparse. Since the support is sparse, there must be many patterns p for which the integer part of x_p is nonzero. This is used to bound the number of iterations (i.e., recursive calls) to BIN-PACK by $O(\log(OPT(\mathcal{I})))$.

Claim 10. *The number of different item sizes in instance $\mathcal{I}'$ is at most $SIZE(\mathcal{I})/2$.*

Proof. Each group $G_j \in \mathcal{G}(\mathcal{I}')$ has size $SIZE(G_j) \geq 2$, and all items in group G_j have the same size in instance $\mathcal{I}'$. □

Claim 10 states that an optimal solution for (Q_{bin}) on instance $\mathcal{I}'$ has at most $SIZE(\mathcal{I})/2$ nonzero variables.

Claim 11. *In an execution of* Bin-Pack$(\mathcal{I})$*, the number of recursive calls to* Bin-Pack *is* $O(\log(OPT(\mathcal{I})))$.

Proof. From Claim 10, it follows that there are at most $SIZE(\mathcal{I})/2$ constraints in (Q_{bin}), and therefore, there are at most $SIZE(\mathcal{I})/2$ many nonzero variables in an optimal extreme point solution for (Q_{bin}) on instance $\mathcal{I}'$. Moreover, for each pattern p such that x_p is nonzero, the contribution to $SIZE(\mathcal{I}_2)$ is $x_p - \lfloor x_p \rfloor \leq 1$. Thus, we have:

$$SIZE(\mathcal{I}_2) \quad \leq \quad \frac{SIZE(\mathcal{I})}{2}.$$

This implies that the number of recursive calls to Bin-Pack by Bin-Pack$(\mathcal{I})$ is $O(\log(SIZE(\mathcal{I}))) \leq O(\log(OPT))$. □

Claim 12. *The number of discarded items in* $\mathcal{D}$ *is* $O(\log(OPT))$.

Proof. This follows directly from Claims 8 and 11. □

Claim 13. *Items packed as* $\mathcal{I}_1$ *(integer parts in Step 6. of* Bin-Pack*) use* $OPT_{LP}(\mathcal{I})$ *bins.*

Proof. From Claim 9, we have:

$$OPT_{LP}(\mathcal{I}_1) + OPT_{LP}(\mathcal{I}_2) \leq OPT_{LP}(\mathcal{I}') \leq OPT_{LP}(\mathcal{I}).$$

By recursively applying Claim 9, we see that the each item packed as $\mathcal{I}_1$ (i.e., all items not assigned to $\mathcal{D}$) can be mapped to a slot (of an item at least as large) in the optimal solution for (Q_{bin}) on the original instance $\mathcal{I}$. □

Theorem 14. *The total number of bins is* $OPT_{LP}(\mathcal{I}) + O(\log(OPT(\mathcal{I})))$.

Proof. By Claim 13, the non-discarded items can be packed into at most $OPT_{LP}(\mathcal{I})$ bins. Meanwhile, the discarded items in $\mathcal{D}$ can be packed in $O(\log(OPT))$ additional bins. □

References

[DLVL81] W. Fernandez De La Vega and George S. Lueker. Bin packing can be solved within $1+\varepsilon$ in linear time. *Combinatorica*, 1(4):349–355, 1981.

[KK82] Narendra Karmarkar and Richard M. Karp. An efficient approximation scheme for the one-dimensional bin-packing problem. In *23rd Annual Symposium on Foundations of Computer Science*, pages 312–320. IEEE, 1982.

[LST90] Jan Karel Lenstra, David B Shmoys, and Éva Tardos. Approximation algorithms for scheduling unrelated parallel machines. *Mathematical programming*, 46(1-3):259–271, 1990.

[WS11] David P. Williamson and David B. Shmoys. *The Design of Approximation Algorithms*. Cambridge University Press, 2011.

This lecture is based in part on lecture notes (Lecture 7) from a course on approximation algorithms at EPFL (`http://theory.epfl.ch/osven/courses/Approx13`) and Section 4.6 of [WS11]. The algorithm for scheduling on unrelated parallel machines is due to Lenstra, Shmoys and Tardos [LST90] and the algorithm for bin packing is due to Karmarker and Karp [KK82].