Lecture 5 (Sep 15, 2015): Bin Packing and Max SAT

Lecturer: Mohammad R. Salavatipour *Scribe: Samuel Fischer*

5.1 Bin packing (continuation)

In the last lecture we introduced the bin packing problem, in which elements of given sizes shall be packed into a minimal number of unit size bins. We introduced a simple greedy algorithm (the first fit algorithm) to solve the problem approximately and showed that it leads to a solution not worse than $2\ opt + 1$.

In this lecture we will show that there are algorithms that return results much closer to the optimum.

Theorem 1 *For any $0 < \epsilon \leq \frac{1}{2}$ there is an algorithm A_ϵ that runs in polynomial time and returns a solution to the bin packing problem of cost $\leq (1 + \epsilon)\ opt + 1$.*

Proof. We will prove this theorem in multiple steps. First, we will consider only small items that have at most size $\frac{\epsilon}{2}$. Then, we will consider a simplified instance of the general problem, in which all items have sizes from a fixed set of size values. We will use the solution to this simplified problem to solve the bin packing problem for the "big" elements (with sizes larger than $\frac{\epsilon}{2}$). Finally, we will combine the approaches in order to obtain an algorithm that returns an approximation of the desired quality.

5.1.1 Bin packing with small elements

Suppose all items are small, i.e. their size is $\leq \frac{\epsilon}{2}$. We call this problem instance I_S

Claim 1 *If we apply the first fit algorithm to this problem, there will be less than $\frac{\epsilon}{2}$ free space in all bins except maybe one.*

Proof. We can prove this easily by contradiction: Let us call the ith bin B_i, whereby the index i indicates the order in which the bins are visited by the first fit algorithm. Suppose there were two bins filled not more than $\left(1 - \frac{\epsilon}{2}\right)$. Call their indices j and k with $j < k$ and let s be the size of an arbitrary element in B_k.

Let us now look back into how the algorithm could have achieved this setting: The algorithm would have tried to fill the element s into B_j first, because $j < k$. This would have been successful, since there is at least $\frac{\epsilon}{2} \geq s$ space in B_j. Therefore, the considered item would have been filled into B_j, which contradicts the assumption that it is in B_k. Hence, the algorithm cannot return a setting like the constructed. Thus, there is not more than one bin filled $\leq \left(1 - \frac{\epsilon}{2}\right)$. ■

Using this observation, it is not difficult to assess the approximation quality of the first fit algorithm:

Claim 2 *The first fit algorithms applied to a bin packing problem instance in which all items have sizes less or equal to $\frac{\epsilon}{2}$ returns a cost $\leq (1 + \epsilon)\ opt + 1$.*

Proof. Let $FF(I_S)$ denote the number of required bins returned by the first fit algorithm and *opt* the cost of the optimal solution, i.e. the number of required bins with optimal packing. As shown above, all bins but maybe one are filled with at least $\left(1-\frac{\epsilon}{2}\right)$. Hence,

$$\left(1-\frac{\epsilon}{2}\right)(FF(I)-1) \quad \leq \quad \sum_{i=1}^{n} s_i.$$

Furthermore, the items must occupy at least $\lceil \sum_{i=1}^{n} s_i \rceil$ bins. That is,

$$\sum_{i=1}^{n} s_i \quad \leq \quad opt.$$

Therefore,

$$\begin{aligned} \left(1-\frac{\epsilon}{2}\right)(FF(I)-1) &\leq opt \\ &\Leftrightarrow \\ FF(I) &\leq \left(\frac{1}{1-\frac{\epsilon}{2}}\right) opt + 1 \\ &\leq (1+\epsilon)\, opt + 1. \end{aligned}$$

For the last step we used that $1 < 1+\frac{\epsilon}{2}-\frac{\epsilon^2}{2} = (1+\epsilon)\left(1-\frac{\epsilon}{2}\right) \;\Leftrightarrow\; \frac{1}{1-\frac{\epsilon}{2}} < 1+\epsilon$. ∎

5.1.2 Bin packing with a fixed set of element sizes

Before we proceed to regard the problem instance with the big elements, we consider a simplified version of the bin packing problem. Suppose that the size of each element can only take one out of r different values. Let $\{s_i \ : \ 1 \leq i \leq r\}$ be the set of the occurring sizes.

Definition 1 *A vector $(x_1, \ldots, x_r)$ is called a configuration, if we can pack a bin using x_i items of size s_i, respectively.*

We can obtain an upper bound for the number of all configurations by assuming that each x_i in a configuration can take an integer between 0 and n (the number of considered items). If this were the case, there would be $(n+1)^r$, which is in $\mathcal{O}(n^r)$, different possibilities.

In any solution to the bin packing problem a bin corresponds to a configuration. Let C be the set of all configurations whose items fit into a single bin. That is, $C := \{(x_1, \ldots, x_r) \ : \ \sum_{i=1}^{r} x_i \leq 1\}$.

Let $A[x_1, \ldots x_r]$ be the number of bins needed to pack a set of items containing x_i items of size s_i, respectively. We can solve the problem by applying dynamic programming as outlined in algorithm 1.

Algorithm 1 Bin packing with fixed element sizes linear programming Algorithm.

1: **initialize** all entries of r-dimensional array A with ∞
2: **for all** $(x_1, \ldots, x_r) \in C$ **do**
3: $\quad A[x_1, \ldots, x_r] = 1$
4: **end for**
5: **for** $i_1 \leftarrow 0$ **to** n **do**
6: $\quad$ **for** $i_2 \leftarrow 0$ **to** n **do**
7: $\quad\quad \vdots$
8: $\quad\quad$ **for** $i_r \leftarrow 0$ to n **do**
9: $\quad\quad\quad$ **for all** $(x_1, \ldots, x_r) \in C$ **do**
10: $\quad\quad\quad\quad A[i_1, \ldots, i_r] = \min\{A[i_1, \ldots, i_r],\ A[i_1 - x_1, i_2 - x_2, \ldots, i_r - x_r] + 1\}$
11: $\quad\quad\quad$ **end for**
12: $\quad\quad$ **end for**
13: $\quad\quad \vdots$
14: $\quad$ **end for**
15: **end for**

5.1.3 Bin packing with big elements

We have already regarded the problem considering only the small items. Let us now look at the problem instance I_L in which all items are big, i.e. their size is greater than $\frac{\epsilon}{2}$. Suppose we have n such items.

We will solve the problem by reducing it (by introducing small errors) to a bin packing problem with a small set of element sizes – just as we regarded it in the previous section.

First of all, we sort the items by their sizes in decreasing order. This works in polynomial time. Now we put every k consecutive items in one group, for k to be specified later. The group G_1 contains the k largest items, the group G_2 the next k items and so on. We obtain $\lceil \frac{n}{k} \rceil$ groups.

We discard the items in G_1 and replace in all other groups G_i with $2 \leq i \leq \lceil \frac{n}{k} \rceil$ the sizes of all items with the size of the respective largest item in the group G_i. See figure 5.1 for a visual presentation of the described procedure.

Let us call the obtained problem instance I'_L.

Lemma 1 *For the optimum of the instances I_L and I'_L: opt$(I'_L) \leq$ opt(I_L)*

Proof. To prove this inequality we build a further problem instance I''_L starting from I_L. However, this time

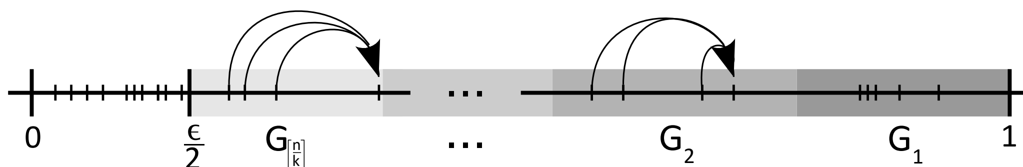

Figure 5.1: Visualizing of the rounding procedure to trasform the bin packing problem with large elements into a bin packing problem in which the element sizes are from a fixed set. On the horizontal line the different element sizes (small dashes) are plotted. The grey boxes show which elements each group contains. The arrows denote by which value an element size is replaced during the replacement procedure.

we proceed in the opposite way than we did in the construction of I'_L:

- Instead of G_1 we discard the group $G_{\lceil \frac{n}{k} \rceil}$
- In $G_1, \ldots, G_{\lceil \frac{n}{k} \rceil - 1}$ we replace the size of all items with the size of the smallest item in their respective group.

Clearly, it is $opt(I''_L) \leq opt(I_L)$, because all item sizes were either decreased or not changed. In addition, I''_L does not contain the items in $G_{\lceil \frac{n}{k} \rceil}$. On the other hand, $opt(I'_L) \leq opt(I''_L)$, since the largest items have been discarded in I'_L and all items in G_{i+1} in instance I'_L are smaller than or equal to those in group G_i in instance I''_L. Thus, $opt(I'_L) \leq opt(I_L)$. ■

To solve the problem I_L we solve I'_L and distribute the k items from G_1 with the first fit algorithms to further bins (at most k of them). The overall resulting number of bins will be smaller or equal $opt(I'_L) + k$.

Of course it is important to make a good choice for the number k of items in each group. Let us set $k := \lceil \epsilon S \rceil$, whereby $S := \sum_{i=1}^{n} s_i$ is the sum of the sizes of all items. Because all items are greater than $\frac{\epsilon}{2}$ and all bins have size 1 (which implies that we need at least $\lceil S \rceil$ bins), it is $n\frac{\epsilon}{2} \leq S \leq opt$.

Using these observations, we can compute the running time of the algorithm. We obtain for the number r of different item sizes (# of constructed groups − 1) in instance I'_L:

$$
\begin{aligned}
r + 1 &= \left\lceil \frac{n}{k} \right\rceil \\
&\Leftrightarrow \\
r &\leq \frac{n}{k} = \frac{n}{\lceil \epsilon S \rceil} \leq \frac{n}{\epsilon n \frac{\epsilon}{2}} = \frac{2}{\epsilon^2}
\end{aligned}
$$

That is, the dynamic programming part of the algorithm runs in $\mathcal{O}\left(n^{\frac{1}{\epsilon^2}}\right)$.

Claim 3 *The proposed procedure leads to a result* $\leq (1 + \epsilon)\, opt(I_L) + 1$.

Proof. It is

$$
\begin{aligned}
opt(I'_L) + k &\leq opt(I_L) + \lceil \epsilon S \rceil \\
&\leq opt(I_L) + \epsilon\ opt(I_L) + 1 \\
&= (1 + \epsilon)\, opt(I_L)) + 1.
\end{aligned}
$$

In the first line we applied lemma 1. For the second inequality we used that $S \leq opt$. ■

5.1.4 Combining the approaches

In order to solve the general bin packing problem we combine the our solutions for the instances I_S and I_L as follows:

In figure 5.2 we show how the result of the combined approach could look like.

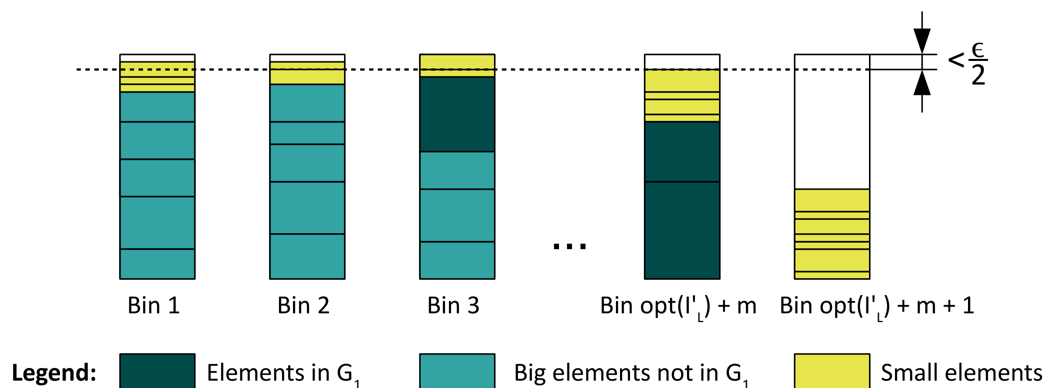

Figure 5.2: Visualization of the combined bin packing algorithms. The turquois elements have been distributed using the approach for large items. In particular, the light turquois elements have bin distributed using algorithm 1. The dark turquois and then the yellow elements were added afterwards using the first fit algorithm. In all but maybe one bins that contain small (yellow) items, there is less than $\frac{\epsilon}{2}$ free space. Distributing only the light turquois elements would require $opt(I'_L)$ bins. The number m denotes the number of additional bins that is required, if also the dark turquois items have to be considered.

Algorithm 2 Bin packing with successively considering large and small items

- Pack the large ($> \frac{\epsilon}{2}$) items into $opt(I_L)$ bins
- Use the first fit algorithm to add the small ($\leq \frac{\epsilon}{2}$) items.

Claim 4 *The returned solution has a value* $\leq \max\{opt(I_L), (1+\epsilon)\, opt_{tot} + 1\}$*, whereby* opt_{tot} *denotes the overall optimum.*

Proof. If all small items fit in the bins of the large items, they will not occupy further space. On the other hand, if they do not fit in the bins of the large items, all (except maybe one) bins will be filled more than $1 - \frac{\epsilon}{2}$ (same argument as in claim 1). ■

Theorem 1 follows directly from claim 3 and claim 4. ■

Further scientific progress has been made with regard to the bin packing problem. For example, [KK88] found an algorithm that returns a result $\leq opt + \mathcal{O}(\log^2 n)$. Later, even better algorithms were developed. However, it is still an open question whether there is an approximation as good as $opt + \mathcal{O}(1)$

5.1.5 Complementary problem: Scheduling on identical parallel machines

The complementary problem to bin packing is scheduling on identical parallel machines.

Scheduling on identical parallel machines:

- Input:
  - A set of k identical machines $M_1, \ldots, M_k$, which run parallel.
  - A set of n jobs $J_1, \ldots, J_n$ with processing times $p_1, \ldots, p_n$, respectively.

- Goal: Assign the jobs to the machines in a way that the "make span", i.e. the latest time any machine finishes its last job, is minimized.

In the bin packing problem we tried to minimize the number of bins (here: machines), whereby each bin had a given size. On the other hand, in the scheduling problem the number of machines (previously: bins) is fixed and we try to minimize the largest computation time (previously: bin size).

Note that even if two problems are complementary to each other, their solutions can be of different computational difficulty.

We will pass on solving the problem for now and go to the next section directly: maximum satisfiability.

5.2 Max SAT (Maximum Satisfiability)

Before we introduce the maximum satisfiability problem, recall the term "CNF" ("clausal normal form", or "conjunctive normal form"). It is a boolean expression that consists of a conjunction ("and-linkage") of clauses. Thereby, a clause is a disjunction ("or-linkage") of literals.

Example 1 *Let* $x_1, \ldots, x_6$ *be literals. Then the statement*

$$(x_1 \vee \bar{x}_2 \vee) \wedge (\bar{x}_5 \vee x_4) \wedge (\bar{x}_1 \vee x_3 \vee x_6) \tag{5.1}$$

is in CNF. The expressions inside the parentheses are clauses. Note that we write $\bar{x}$ *for* $\neg x$*.*

Naively speaking, we want to maximize the "truth value" of a given CNF in the maximum satisfiability problem.

Maximum satisfiability problem:

- Input: A CNF formula over n variables $x_1, \ldots x_n$, whereby each clause c_i has weight w_i.
- Goal: Find a truth assignment that maximizes the total weight of satisfied clauses.

We can distinguish multiple special cases of the Max SAT problem:

- Max k SAT: There are $\leq k$ literals in each clause. The respective decision problem is NP-hard for $k \geq 3$.
- Max E k SAT: There are exactly k literals in each clause.

Theorem 2 *Max k SAT is NP-hard for any $k \geq 2$.*

We state this theorem without proof.

There are plenty of algorithms to solve the problem approximately. As we did for the bin packing problem, we introduce some simple algorithms first, which we will combine later to a more sophisticated approach that leads to an even better approximation. Let us start by looking at the algorithm "random assignment using fair coins":

Theorem 3 (Johnson '74) *Algorithm 3 is a $\frac{1}{2}$-approximation.*

Algorithm 3 (Max SAT 1) Random assignment using fair coins

1: **for all** variables x_i **do**
2: $x_i \leftarrow True$ or $False$ with probability $\frac{1}{2}$, respectively
3: **end for**

Proof. For all clauses c_j define

$$z_j := \begin{cases} 1 & \text{if clause } c_j \text{ is satisfied} \\ 0 & \text{else.} \end{cases}$$

A clause c_j is satisfied if any of its literals is true. We randomly assign truth values to the variables. Therefore, the literals are true or false with probability $\frac{1}{2}$, respectively, too. The probability that the $|c_j|$ literals in j are all false is $\frac{1}{2^{|c_j|}}$. Hence, the expected value of z_j is $\mathbb{E}[z_j] = \left(1 - \frac{1}{2^{|c_j|}}\right) \geq \frac{1}{2}$.

The total weight w of all satisfied clauses is given by

$$w = \sum_j w_j z_j.$$

Hence, the expected total weight is given by

$$\begin{aligned} \mathbb{E}[w] &= \sum_j w_j \mathbb{E}[z_j] \\ &= \sum_j w_j \left(1 - \frac{1}{2^{|c_j|}}\right) \\ &\geq \frac{1}{2} \sum_j w_j. \end{aligned}$$

Since $opt \geq \sum_j w_j$ (we cannot obtain a result better than that all clauses are true), the algorithm gives a $\frac{1}{2}$-approximation. ∎