Technische Universität München, Zentrum Mathematik
Lehrstuhl für Angewandte Geometrie und Diskrete Mathematik

Discrete Optimization (MA 3502)

Prof. Dr. Nicole Megow | Dr. Fidaa Abed | Dr. Michael Ritter

Exercise Sheet 7

Problem 7.1 (1-Tree Lagrangian Relaxation for TSP)
Let $V = [n]$, $G = (V, E)$ a complete graph and $c : E \longrightarrow \mathbb{R}_{\geq 0}$ a weight functions, which associates to each edge $e \in E$ a cost c_e. Consider the following ILP formulation of the TSP:

$$\min \sum_{e \in E} c_e x_e$$

$$\sum_{e \in E} x_e = n \tag{1}$$

$$\sum_{e \in \delta(1)} x_e = 2 \tag{2}$$

$$\sum_{e \in \delta(i)} x_e = 2 \qquad i \in [n] \setminus 1 \tag{3}$$

$$\sum_{e \in E(S)} x_e \leq |S| - 1 \qquad \emptyset \subset S \subset V \setminus v_1 \tag{4}$$

$$x \in \{0, 1\} \tag{5}$$

a) Use the inequalities (3) to construct a Lagrangian relaxation for this problem. Which well-known combinatorial problem is the relaxed problem similar to, and how can it be solved efficiently?

b) What is the subgradient of the Lagrange function?

c) Use the subgradient method to compute z_{LD} for the weighted graph depicted in Figure 1. Use the steplength $\xi_i = \frac{\sqrt{2}}{i}$ for $i \in \mathbb{N}$ and start with $\lambda^1 = 0$. What does this imply for the optimal solution for this problem?

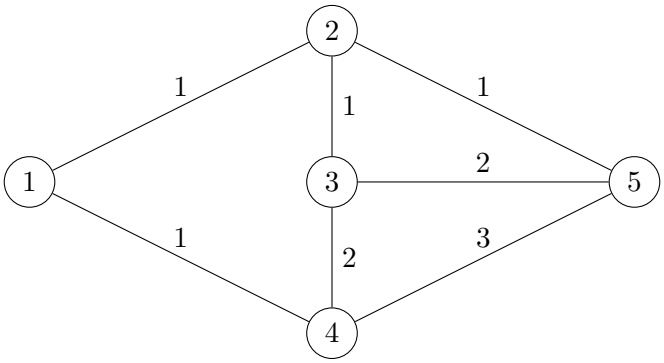

Figure 1: The graph for Problem 7.1

Answer to Problem 7.1

a) We relax the constraints $\sum_{e\in\delta(i)} x_e = 2,\ i \in [n]\setminus\{1\}$ and get the following problem

$$
\begin{aligned}
z_{LR}(\lambda) = \min\ & \sum_{e\in E} c_e x_e + \sum_{v\in V\setminus\{1\}} \lambda_v \left(\sum_{e\in\delta(v)} x_e - 2\right) \\
& \sum_{e\in E} x_e = n \\
& \sum_{e\in\delta(1)} x_e = 2 \\
& \sum_{e\in S} x_e \le |S| - 1 \qquad\qquad \emptyset \subset S \subset V\setminus v_1 \\
& x \in \{0,1\} \\
& \lambda \in \mathbb{R}^{n-1}
\end{aligned}
$$

Notice that since we relaxed equality constraints, we do not demand $\lambda \ge 0$, but instead $\lambda \in \mathbb{R}^{n-1}$. Also, λ_1 is missing, because the degree inequality corresponding to node 1 has not been relaxed. Of course, one could also include λ_1 and just set it to 0.

The objective function can be rewritten as

$$\sum_{e=\{v,w\}\in E} \underbrace{(c_e + \lambda_v + \lambda_w)}_{=:c'_e} x_e - 2\sum_{v\in V\setminus\{1\}} \lambda_v.$$

Note that the term $(-2\sum_{v\in V\setminus\{1\}} \lambda_v)$ is constant for fixed λ and can therefore be ignored when computing an optimal solution x (but of course not for the objective value!).

Without the constraint $\sum_{e\in\delta(1)} x_e = 2$ and after changing the right hand side of the first inequality from n to $n-2$, the relaxed problem is just a minimum spanning tree problem on the graph $\tilde{G} = (\tilde{V} := V\setminus\{1\}, \tilde{E} := E(V\setminus\{1\}))$. It is therefore easy to see that an optimal solution to this *one-tree* problem (with edge-weights c'_e) can be found by computing a minimum spanning tree on $V\setminus\{1\}$ with respect to c' and then adding two edges incident to 1 of minimal cost. An optimal solution for the relaxed problem is given by the one-tree computed as described above. The optimal value $z_L(\lambda)$ is given by the weight of the optimal one-tree minus $2\sum_{v\in V\setminus\{1\}} \lambda_v$.

b) The components of the subgradient are given by $s_v = \sum_{e\in\delta(v)} x_e - 2$ for $v \in V\setminus 1$.

c) For $\lambda^{(1)} = 0$, the optimal one-tree is given by the edges $T^1 = \{\{1,2\},\{1,4\},\{2,3\},\{2,5\},\{3,4\}\}$, yielding $z_L(0) = 6$, see Figure 2. The subgradient at $\lambda^{(1)} = 0$ contains

$$\nabla z_L(\lambda^{(1)}) = \begin{pmatrix} 1 \\ 0 \\ 0 \\ -1 \end{pmatrix}$$

In a first update step we therefore get

$$\lambda^{(2)} = \lambda^{(1)} + \xi_1 \frac{\nabla z_L(\lambda^{(1)})}{\|\nabla z_L(\lambda^{(1)})\|} = 0 + \frac{\sqrt{2}}{\sqrt{2}} \cdot \begin{pmatrix} 1 \\ 0 \\ 0 \\ -1 \end{pmatrix} = \begin{pmatrix} 1 \\ 0 \\ 0 \\ -1 \end{pmatrix}.$$

This yields the new edge weights depicted in Figure 2 (right hand side) and the new 1-tree

$$\{\{1,2\},\{2,5\},\{5,3\},\{3,4\},\{4,1\}\}.$$

The value of $z_L(\lambda^{(2)}) = 7 - 2(1-1) = 7$. As our current solution is also a TSP tour of length 7, we know we have an optimal solution (its value coincides with the current lower bound), thus we may stop the procedure at this point.

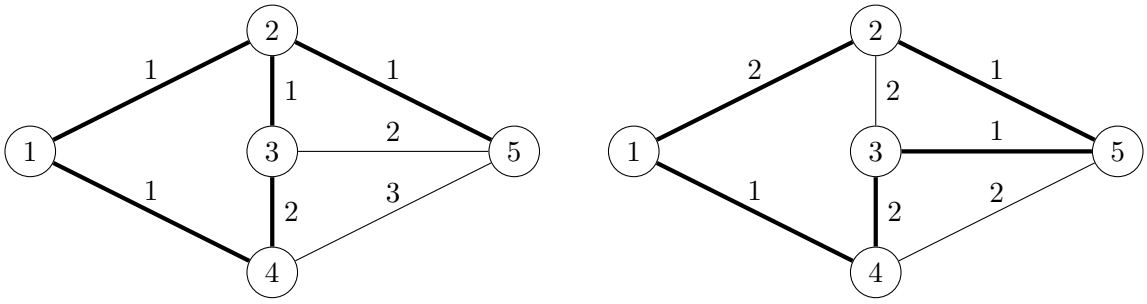

Figure 2: graph for Problem 7.1

In the last step, one could also have chosen a different optimal solution for the 1-tree problem ($\{\{1,2\},\{1,4\},\{2,3\},\{2,5\},\{4,5\}\}$), which results in additional iterations (the number depends on the subsequent choice of optimal solutions).

Problem 7.2 (An Approximation Algorithm for the Knapsack Problem)
Consider an instance of the knapsack problem on $n \in \mathbb{N}$ items given by values $v \in \mathbb{N}^n$, weights $w \in \mathbb{N}^n$ and a bound $b \in \mathbb{N}$, where $w_i \leq b$ for all $i \in \{1,\dots,n\}$. Let z^* denote the value of a maximum knapsack and z_{greedy} the value of a knapsack obtained through the following algorithm:

Input: An integer n, a vector $v \in \mathbb{N}^n$ of values, a vector $w \in \mathbb{N}^n$ of weights and a bound $b \in \mathbb{N}$ such that $w_i \leq b$ for all $i \in \{1,\dots,n\}$.
Output: A feasible knapsack solution $I \subset \{1,\dots,n\}$.
Sort the items such that $\frac{v_1}{w_1} \geq \dots \geq \frac{v_n}{w_n}$.
Determine $k' := \max\left\{k \in \{1,\dots,n\} : \sum_{i=1}^{k} w_i \leq b\right\}$.
if $\sum_{i=1}^{k'} v_i > v_{k'+1}$
then
  return $I := \{1,\dots,k'\}$
else
  return $I := \{k'+1\}$

a) Show that the algorithm is a 2-approximation for the knapsack problem, i. e. $z_{\text{greedy}} \geq 1/2 \cdot z^*$.

b) Prove that the constant $1/2$ is best possible for this algorithm, i. e. there is no constant $r < 2$ such that $z_{\text{greedy}} \geq 1/r \cdot z^*$. for all possible instances of the knapsack problem.

c) Show that the last step of the algorithm is essential, i. e. prove that by always returning the knapsack $I := \{1,\dots,k'\}$ the ratio $\frac{z_{\text{greedy}}}{z^*}$ can become arbitrarily small.

Answer to Problem 7.2

a) Let us first remark that the algorithm presented above always yields a feasible solution to the knapsack instance, as the items are selected to adhere to the weight limit b. Furthermore, the

algorithm needs to compute at most n sums, each can be computed by adding just one number to the previous sum, and do a comparison for each of these sums. One more comparison at the end means we get a running time of $\mathcal{O}(n)$, thus the algorithm is polynomial in the encoding size of the input.

For the approximation ratio, we first need to prove a bound on the optimal solution z^*. Using the notations defined above we claim that

$$z^* \leq \sum_{i=1}^{k'} v_i + \frac{b - \sum_{i=1}^{k'} w_i}{w_{k'+1}} v_{k'+1} \leq \sum_{i=1}^{k'+1} v_i.$$

Assume that a positive fraction of any item $i > k' + 1$ would be contained in the optimal knapsack. Then there is some item $i' \leq k' + 1$ contained in that solution with less than the amount given in the above formula. Exchanging a little part of item i for the same part of item i' will lead to a different solution that is at least as good as the original one. Repeating this process will eventually yield the solution claimed above without decreasing the objective value, thus that solution is also optimal.

The greedy solution consists either of $\{1, \ldots, k'\}$ or just of $\{k'\}$, whichever has higher value. Therefore,

$$z^* \leq \sum_{i=1}^{k'} v_i + v_{k'+1} \leq z_{\text{greedy}} + z_{\text{greedy}} = 2z_{\text{greedy}}.$$

b) Let $\varepsilon > 0$ a small, positive number and consider the following 3 item instance of the knapsack problem:

$$\begin{aligned} w_1 = v_1 &= \frac{b}{2} + \varepsilon \\ w_2 = v_2 &= \frac{b}{2} \\ w_3 = v_3 &= b \end{aligned}$$

The relative value $\frac{v_i}{w_i} = 1$ for all three items, so we can assume they are sorted by increasing number. Then, the greedy algorithm would return the solution $\{1\}$ with value $\frac{b}{2} + \varepsilon$, whereas an optimal solution would be to take just the third item, yielding a value of b. Thus, for every constant $r < 2$, we can choose a suitable value for ε to prove $z^* \not\leq r \cdot z_{\text{greedy}}$.

A simple idea to overcome this problem would be the following modification of the algorithm: Determine the index j_0 of the highest-valued item, i. e.

$$v_{j_0} = \max\{v_j : j \in \{1, \ldots, n\}\}.$$

Then, in the last step of the algorithm, compare the two possible solutions $\{1, \ldots, k\}$ and $\{j_0\}$ and return the one with higher total value. Obviously, that renders our above example worthless, because the algorithm would now return the third item with value b.

Of course, the counterexample can easily be modified to deal with this new situation: Simply split the third item into many small items, so to make sure that the algorithms never chooses

one of these smaller items:

$$\begin{aligned} w_1 = v_1 &= \frac{b}{2} + \varepsilon \\ w_2 = v_2 &= \frac{b}{2} \\ w_3 = v_3 &= \frac{b}{n-2} \\ &\vdots \\ w_n = v_n &= \frac{b}{n-2} \end{aligned}$$

Choose n large enough so that $\frac{b}{n-2} < \frac{b}{2} + \varepsilon$, then the greedy algorithm again returns the solution $\{1\}$ with a value of $\frac{b}{2} + \varepsilon$, whereas a better solution would be the knapsack $\{3, \ldots, n\}$ of total value b.

c) For small $\varepsilon \in (0, b)$, consider the following 2 item instance:

$$\begin{aligned} v_1 &= \varepsilon & w_1 &= \varepsilon \\ v_2 &= b & w_2 &= b \end{aligned}$$

The ratios $\frac{v_1}{w_1} = \frac{v_2}{w_2} = 1$, so we may assume that the items are already sorted. The greedy algorithm without the last step would then return $\{1\}$ with a value of ε, whereas the optimal solution is clearly $\{2\}$ with a total value of b, hence

$$\frac{z_{(greedy)}}{z^*} = \frac{\varepsilon}{b},$$

which can become arbitrarily small by choosing ε suitably.

A general remark: The instances in this exercise are often chosen such that the ratios $\frac{v_i}{w_i}$ are all equal and then a "bad" sorting is assumed. It is of course possibly to enforce the desired sorting and still keep the examples working by choosing some small $\delta \in (0, 1)$ and substituting $v_i := v_i + \delta^i$ for each item i, which would lead to a unique order of the items and just slightly more complicated proofs.

Problem 7.3 (An Approximation Algorithm for Load Balancing)
Consider the Load Balancing Problem:

Input: n jobs with processing times $p_1, \ldots, p_n \in \mathbb{N}$, a number $m \in \mathbb{N}$ of processors.

Task: Find an assignment of jobs to machines that minimizes the maximum load over all machines. The load of machine $i \in [m]$ is the sum of processing times of jobs assigned to i.

a) Show that the following simple list scheduling algorithm (LS) is a $(2 - 1/m)$-approximation for Load Balancing.

LS: Consider jobs in an arbitrary, but fixed order. Assign a job to the machine that has currently minimum load.

b) Show that the analysis is tight, i. e. construct an instance of Load Balancing where LS yields a solution that is a factor $(2 - 1/m)$ away from an optimum solution.

c) Can you modify the algorithm to improve upon the approximation ratio?

Answer to Problem 7.3

a) For clearer notation, the algorithm is formalized in 1. Let C^* denote the optimal makespan and C^g the makespan achieved through the greedy algorithm. The running time is dominated by the main loop which does n iterations, finding the currently least used machine is a matter of at most m comparisons. As m is dominated by n, we have a running time of $\mathcal{O}(n^2)$, which is polynomial in the encoding size of the input. The algorithm always yields an assignment of all jobs to the machines, so the solution obtained is clearly feasible.

The optimal makespan is bound below by the processing time for each single job and also by the average processing time per machine (the LP relaxation), i. e.

$$C^* \geq p_j \quad \text{for all } j \in [n],$$
$$C^* \geq \frac{1}{m} \sum_{j \in [n]} p_j.$$

Let $k \in [n]$ be the last job that is completed in the LS solution, then the starting time of that job is $C^g - p_k$. At the time that job was scheduled on a machine, the completion times of all other machines where at least $C^g - p_k$, thus the total processing time without job k is at least m times that number:

$$\sum_{j \in [n] \setminus \{k\}} p_j \geq m \cdot (C^g - p_k)$$

This yields the following inequality proving the claimed approximation ratio:

$$\begin{aligned} C^g = (C^g - p_k) + p_k &\leq \frac{1}{m} \sum_{j \in [n] \setminus \{k\}} p_j + p_k \\ &= \frac{1}{m} \sum_{j \in [n]} p_j - \frac{p_k}{m} + p_k \\ &\leq C^* + p_k \left(1 - \frac{1}{m}\right) \\ &\leq C^* + C^* \left(1 - \frac{1}{m}\right) = \left(2 - \frac{1}{m}\right) C^* \end{aligned}$$

b) Consider an instance with 2 processors and 3 jobs with processing times $p_1 = 1$, $p_2 = 1$, $p_3 = 2$. The LS algorithm assigns jobs 1 and 3 to processor 1 and job 2 to processor 2, resulting in a total makespan of 3 (on processor 1). However, an optimal assignment would be to assign jobs 1 and 2 to processor 1 and job 3 to processor 2, resulting in a makespan of 2. Thus

$$3 = C^g = \frac{3}{2} \cdot 2 = \frac{3}{2} C^* = (2 - \frac{1}{2}) C^*$$

c) We modify the algorithm by sorting the list of jobs in decreasing order according to their processing times beforehand. We claim that this change yields a $(\frac{4}{3} - \frac{1}{3m})$-approximation.

To simplify notation, let us assume that $p_1 \geq p_2 \geq \cdots \geq p_n$ and let again $k \in [n]$ denote the last job to finish, C^* the optimal makespan and C^g the makespan of the solution computed by our

algorithm. We first consider the case where $p_k \leq \frac{1}{3}C^*$, then the same argument as above yields

$$\begin{aligned} C^g = (C^g - p_k) + p_k &\leq \frac{1}{m} \sum_{j \in [n] \setminus \{k\}} p_j + p_k \\ &= \frac{1}{m} \sum_{j \in [n]} p_j - \frac{p_k}{m} + p_k \\ &\leq C^* + p_k \left(1 - \frac{1}{m}\right) \\ &\leq C^* + \frac{C^*}{3} \left(1 - \frac{1}{m}\right) = \left(\frac{4}{3} - \frac{1}{3m}\right) C^* \end{aligned}$$

Now suppose $p_k > \frac{1}{3}C^*$. We focus on just the tasks $p_1, \ldots, p_k$ and disregard the remaining tasks, this will only yield a better optimal makespan. As $C^* < 3p_k$ and p_k is the job with the shortest processing time, each machine can accommodate at most 2 jobs. The jobs assigned to one machine i can then be denoted by $i_1 < i_2$ where $p_{i_1} \geq p_{i_2}$. Also, we may assume that machines are sorted such that the larger first jobs are scheduled on the machines with smaller indices, i. e. for $i < i'$ we have $p_{i_1} \geq p_{i'_1}$. For the second job on each machine, we may assume that $i < i'$ implies $p_{i_2} \leq p_{i'_2}$, otherwise we could simply exchange the two jobs without altering the optimal makespan.

Let us compare that solution to the schedule our algorithm produces: If our algorithm yields a schedule with at most 2 jobs per machine, then the schedule is precisely the same as that of the optimal solution. So we are left with the case that our algorithm puts a third job on some machine, denote that job by $t \in [n]$. From our discussion above, we know that there is a solution with at most 2 jobs per machine, thus $n \leq 2m$, consequently our solution also contains a machine with just one job (no job is impossible, as the third job would then have been scheduled to the empty machine), we will denote that job by $s \in [n]$. This job's processing time must be longer than that of the two jobs already on the machine that job t is assigned to, hence

$$p_s \geq 2p_t \geq 2p_k > \frac{2}{3}C^*.$$

However, in the optimal solution, job s is processed on one machine together with a second job, thus the processing time on that machine is at least $p_s + p_k$ and therefore

$$C^* \geq p_s + p_k > \frac{2}{3}C^* + \frac{1}{3}C^* = C^*,$$

clearly a contradiction.

Input: A number $n \in \mathbb{N}$, processing times $p_1, \ldots, p_n \in \mathbb{N}$, a number m.
Output: A feasible assignment of jobs to machines $J_1, \ldots, J_m \subset [m]$, where J_i contains all jobs scheduled on machine i.

for $i = 1, \ldots, m$ **do**
  Set $J_i \leftarrow \emptyset$
for $i = 1, \ldots, n$ **do**
  Find the smallest $k \in [m]$ with $\sum_{j \in J_k} p_j = \min\left\{\sum_{j \in J_i} p_j : i \in [m]\right\}$.
  Set $J_k \leftarrow J_k \cup \{i\}$.
return $(J_1, \ldots, J_m)$

Algorithm 1: LS-Algorithm for Load Balancing