COMP523 Tutorial 7 - Solutions

Coordinator: Aris Filos-Ratsikas Demonstrator: Michail Theofilatos

December 27, 2019

Problem 1

A. Let A be a totally unimodular matrix. Prove that the matrix $A' = [A \quad -A \quad I \quad -I]^{\mathrm{T}}$ is totally unimodular, where I is the identity matrix.

B. Prove that the incidence matrix A of any directed graph is totally unimodular.

Solution

A. We will use the following proposition which can be proven using elementary properties of the determinant (proof omitted here, you can use the proposition as a given).

Proposition 1. *Let A be a tutotally unimodular matrix. Then the matrices $-A$, A^T, $[A \quad I]$ and $[A \quad -A]$ are totally unimodular. Furthermore, if we multiply any row or any column of A by -1, we obtain a matrix A' which is totally unimodular.*

Then, we can obtain the matrix $A' = [A \quad -A \quad I \quad -I]^{\mathrm{T}}$ by repeated applications of the proposition above: I is totally unimodular (known fact: $\det(I) = 1$), so $-I$ is totally unimodular as well. A and $-A$ are also both totally unimodular and so are their transpose matrices.

B. A matrix B is totally unimodular if every square submatrix of A has determinant $-1, 0$ or $+1$. Note that since A is the incidence matrix of a directed graph has entries which are either 0, 1 or -1. Futhermore, it holds that every column of A has exactly one 1 and one -1, and all the other entries of the column are 0.

Before we proceed, we highlight some properties of the determinant for square matrix A':

- If A' has a column of only 0s, then the determinant is 0.
- If the rows of A' are not *linearly independent*, then the determinant is 0.

We will prove that A is totally unimodular by induction on the size of its square submatrices. For the base case, the claim is true for a 1×1 matrix. For the induction step, assume that it holds for all $(k-1) \times (k-1)$ square submatrices of A (induction hypothesis); we will prove that it holds for all $k \times k$ submatrices of A. Let A' be an arbitrary $k \times k$ submatrix of A. We consider a few cases:

- If A' has a column of only 0s, then the determinant of A' is 0.
- If A' has a column j^* with exactly one 1 on row i^*, then consider the matrix A'' obtained from A', after removing row i^* and column j^*. A'' is a $(k-1) \times (k-1)$ matrix and it holds that $\det(A'') = (-1)^{i+j}\det(A')$. This means that either $\det(A') = \det(A'')$ or $\det(A') = -\det(A'')$. By the induction hypothesis, A'' has determinant either 0, -1 or 1, and therefore so does A'.
- If A' has exacty one 1 and one -1 in every column, then, if we add up the rows of A' we get $\mathbf{0}$. This implies that the rows are not linearly independent and therefore the determinant is 0.

Problem 2

Recall the 0/1-Knapsack problem: There is a set N of n items with weights w_i and values v_i and a knapsack with capacity C. The goal is to select a subset $S \subseteq N$ of the n items to put into the knapsack, such that $\sum_{i \in S} w_i \leq W$ holds, and $\sum_{i \in S} v_i$ is maximised.

Design a polynomial time approximation algorithm for the 0/1-Knapsack problem, which achieves an approximation ratio of 2.

Solution

Here is a first attempt at a 2-approximation for the 0/1-knapsack problem: We will use the algorithm that solves the *fractional knapsack* optimally, which we saw in a previous tutorial. Sort the items in terms of their effectiveness, or their "bang-per-buck" v_i/c_i. Greedily put items $a_1, \ldots, a_{i-1}$ in the knaspack according to that order, until an item a_i is encountered that can not fit in the knapsack.

However, it is not hard to see that this algorithm has a bad approximation ratio. Consider an example with one item of size 1 and value 2 and one item with size C and value C. The algorithm will put the first item in the knapsack (and then the second will not fit anymore) for a total value of 2, whereas the optimal solution would be to put the second item in the knapsack, for a value of C. The approximation ratio of the algorithm is at least $C/2$, which can be arbitarily large as C increases.

We will see that a small modification to the algorithm above actually works. Sort the items in terms of their effectiveness, or their "bang-per-buck" v_i/c_i. Greedily put items $a_1, \ldots, a_{i-1}$ in the knaspack according to that order, until an item a_i is encountered that can not fit in the knapsack. Pick the better of $\{a_1, a_2, \ldots, a_{i-1}\}$ and $\{a_i\}$ and put it in the knapsack. Let x be the solution of our algorithm.

Next, we argue about the correctness of the algorithm. Let x^* be an optimal solution to the 0/1-Knapsack instance (where x^* is a set of items) and let $v(x^*)$ be the total value of that solution. Furthermore, let y^* denote the solution to the *fractional* version of the 0/1-knapsack instance, where items are allowed to be partially added to the knapsack and let $v(y^*)$ be its value. Obviously, it holds that $v(x^*) \leq v(y^*)$. Now, observe that since the bang-per-buck greedy algorithm is optimal for the fractional version (this was shown in the lectures), it holds that y^* consists of items $a_1, \ldots, a_{i-1}$ and a λ fraction of item a_i.

We have

$$v(x^*) \leq v(y^*) = [v(a_1) + \ldots v(a_{i-1})] + \lambda v(a_i) \leq 2v(x)$$

This completes the proof.

We remark that this is not the best possible approximation algorithm for the problem. There is actually a Fully Polynomial Time Approximation Scheme (FPTAS) for the algorithm, i.e., an algorithm which runs in time polynomial in the input size and $1/\varepsilon$ and produces an $1+\varepsilon$ approximation to the optimal solution.