

#4 Assignment - CMPT 405

Luiz Fernando Peres de Oliveira - 301288301 - lperesde@sfu.ca

November 16, 2018

#1

Let w_{ij} be the weight of every $(i, j) \in E$ and x_{ij} be variables such that $x_{ij} = 1$ if the shortest path contains $i \rightarrow j$ and $x_{ij} = 0$, otherwise. The shortest path from a source $s \in V$ to a target $t \in V$ in a weighted graph $G = (V, E, w)$ can be found by minimizing the summation of $w_{ij}x_{ij}$ for every (i, j) . See below:

$$x_{ij} = \begin{cases} 1 & \text{if the shortest path contains } i \rightarrow j \\ 0 & \text{otherwise} \end{cases}$$

By the principle of network flow, we have that for each single node i , the amount of a flow f_i is equal the difference between the amount of outgoing flow from i and the amount of incoming flow to i :

$$f_i = \sum_j x_{ij} - \sum_k x_{ki}$$

As we are looking for the shortest path from s to t , we know that our network will "travel" from the source to the target, cancelling any flow f_u for single vertices u between s and t in our network, where $u \neq s$ and $u \neq t$. Because there is no incoming flow in s , $f_s = 1$. Likewise, because there is no outgoing flow in t , $f_t = -1$. Thus:

$$f_i = \begin{cases} 1 & \text{if } i = s \\ -1 & \text{if } i = t \\ 0 & \text{otherwise} \end{cases}$$

Assuming that $x_{ij} \geq 0$, the linear program for the shortest problem is:

$$\min \sum_{(i,j) \in E} w_{ij}x_{ij}$$

The resulting dual will have one variable y_u for each vertex u in the graph. The values of y have the constraint that $y_j - y_i \leq w_{ij}$ and the objective function is

the maximization of $y_s - y_t$:

$$\begin{aligned} \max \quad & y_s - y_t \\ \text{subject to} \quad & y_j - y_i \leq w_{ij}, \forall (i, j) \in E \end{aligned}$$

Dual Encoding: The dual can be interpreted as the encoding of Bellman-Ford, because when BF terminates, it has computed for each vertex j a value y_j , such that for each edge $(i, j) \in E$, we have the same constraints as the dual: $y_j \leq y_i + w_{ij}$. The objective function is also the maximization of $y_s - y_t$.

#2

In a similar way of question #1, let w_e be the weight of every $e \in E$ and x_e be 0 – 1 variables such that $x_e = 1$ if the edge e is in the matching and $x_e = 0$, otherwise.

$$x_e = \begin{cases} 1 & \text{inclusion of edge } e \text{ in the matching} \\ 0 & \text{otherwise} \end{cases}$$

We need to choose at each step an augmenting path p that produces the largest possible increase in weight so that the matching obtained by flipping the edges has maximum weight.

The objective function maximizes the weight of all edges e in the matching and, because we are augmenting paths in our matchings, we use constraints to limit one edge per vertex so that the path is created in the form $x_e \leq 1$, for all vertices u , such that $e = (u, v)$. The linear program is then:

$$\begin{aligned} \max \quad & \sum_e w_e x_e \\ \text{subject to} \quad & \sum_{e=(u,v)} x_e \leq 1, \forall u \in V \end{aligned}$$

#3

To prove that the constraint matrix in previous example is totally unimodular for bipartite graphs, we must prove that every entry is equal to 1, 0 or –1.

Let x_{ij} be an alias for x_e in the previous example, such that $e = (i, j)$. Each row i in x represent a vertex v_i and each column j represents an edge $e = (i, j)$. $x_{ij} = 1$ only if our previous algorithm included (i, j) in the matching.

Proof. Consider an arbitrary square submatrix x' of x . The goal here is to show that the determinant of x' is in $\{1, 0, -1\}$.

Case 1: x' has a column with only 0. Then the determinant of $x' = 0$.

Case 2: x' has a column with only 1. By induction, x'' has determinant equal 1, 0 or -1 and so does x' .

$$x' = \begin{bmatrix} 1 & \dots \\ 0 & x'' \end{bmatrix}$$

Case 3: Each column of x' has exactly two 1. Because we are dealing with a bipartite graph, we have two distinct sets $x^{[+1]}, x^{[-1]}$, $x^{[+1]} \cap x^{[-1]} = \emptyset$, such that each edge $e = (i, j) \in x$ will have i in one of this sets and j in the other set (otherwise it would not be bipartite). Because the rows are linearly dependent, by multiplying $+1$ in the rows in $x^{[+1]}$ and -1 on the columns in $x^{[-1]}$, the determinant of $x' = 0$.

$$x' = \begin{bmatrix} x^{[+1]} \\ x^{[-1]} \end{bmatrix}$$

□

Counterexample for non-bipartite graphs:

In spite of working for bipartite graphs, the same does not hold for non-bipartite graphs. It can be easily proved by a simple counterexample (use the same linear program as described on #2): $G = (V, E)$, $V = \{a, b, c\}$, $E = \{(a, b) = 1/3, (b, c) = 1/3, (a, c) = 1/3\}$.

#4

In this problem, we are trying to find the minimum value $\max(x \cdot a, (1 - x) \cdot b)$, for two vectors a and b of length n , $x \in \{0, 1\}^n$. The algorithm to solve such problem is as follows:

$$x_i = \begin{cases} 1 & \text{if we choose element } i \text{ in vector } a \\ 0 & \text{if we choose element } i \text{ in vector } b \end{cases}$$

$$\text{minimize } \max(x \cdot a, (1 - x) \cdot b)$$

In order to find the minimum $\max(x \cdot a, (1 - x) \cdot b)$ our algorithm will try every single possibility exhaustively. For example, for $n = 3$, our algorithm would try all eight possibilities: 000, 001, 010, 011, 100, 101, 110 and 111. Because we have two choices at a time ($0 - 1$ problem), it will take $O(2^n n)$ time to be solved, meaning that is not likely to have an algorithm that will solve the problem in polynomial time as the greedy approach (always taking the smallest number at a time) will fail for some input. Take $a = [3, 4, 10]$ and $b = [5, 5, 9]$, for example. The DP approach can be done by modifying the $0 - 1$ knapsack problem (which is not polynomial as well), with utility equals weight, where we take an item from b if 0, and from a if 1. Let M be the problem $\max(x \cdot a, (1 - x) \cdot b)$. We want to show that this is a NP-hard problem:

Proof. Reducing knapsack to our problem: It is known that the knapsack problem is NP-complete (and was shown in class how to reduce it to the partition problem, by using utility equals weight and then comparing the sets). The idea of this problem is to show a reduction in polynomial time of the knapsack problem to our algorithm. Let K be an alias for Knapsack problem.

To show that $K \leq_p M$, we only need to make a small change on the abstraction of the problem: on the knapsack problem we have the option of either (i) take element k or (ii) not take element k in a k_{th} iteration. In our reduction, we keep the binary choice by using (i) whenever I choose a a_k element of vector a and (ii) whenever I choose a b_k element of vector b in a k_{th} iteration.

Reduction in polynomial time: The reduction can be done in polynomial time, and, in fact, it does not change the complexity of the problem, because we only changed the way of interpreting the problem.

K has a solution iff M has a solution:

Consider W being the total capacity of weight of a knapsack, then:

\implies If there exists a set of numbers in K that sum up to less or equal than W , then there exists a set of some of numbers using vectors a, b such that they sum up to less or equal than W .

\Leftarrow Let's say that there exists some numbers in a, b that sum up to less or equal than W , then, these numbers in K would have chosen some numbers in either (i) or (ii), that sum up to equal or less than W .

NP-hard problem: Because our problem M was proved to be **NP-complete**, we know that its optimization version (minimum of $\max(x \cdot a, (1 - x) \cdot b)$) is NP-hard.

□

#5

As proved in assignment #1, the wall painting problem is not a problem that can be done by a Greedy algorithm nor it is a problem that can be solved in polynomial time. We should arrange the n_i tiles in column i , where all n_i tiles have (possibly) different heights, such that we paint the wall with as few colours as possible. The idea of the algorithm to solve the problem is to treat each column as a set of tiles and the whole wall as a set of sets of tiles. In this way, we may rearrange the tiles in all possibilities until we find the smallest number of used colours. Let W be the alias for the wall painting problem. We want to show that this is a NP-hard problem:

Proof. Reducing the subset sum to our problem: As a matter of a reduction $SS \leq_p W$, consider the subset sum problem for solving the wall painting problem, and as said above, consider the wall to be a multiset, where every column i is a set. For some integer k representing the colours, does there exist a subset t of W such that the sum of its elements is equal to k ? In this case

here, we might use float values other than the a_{ij} ones, such that all elements adjacent in a row add up to one. For example, if I have three columns with two tiles of same size each, each tile t_{ij} would equal $\frac{1}{3}$ (so that the row adds up to 1). By doing that, it is possible to change the interpretation of the subset sum question to: Does there exist an arrangement for a subset t of W such W can be painted using at most k colours?

Reduction in polynomial time: The reduction can be done in polynomial time because the data structures of the wall in problem W can be in fact created in linear time, once we only need to create one set for each column and a larger set (the wall) that owns all sets.

SS has a solution iff W has a solution:

\implies The subset sum has a solution if some subset of the master set of W is equal to k . If so, it means that there is a number of colours k that can be used to paint the wall.

\impliedby If there is a number of colours k that satisfies our problem W , then, there is a subset sum that adds up to k (by using some of t_{ij}).

NP-hard problem: Because our problem W was proved to be **NP-complete**, we know that its optimization version, using as few colours as possible, is NP-hard. \square