CS41 Lab 5

October 2017

Some notation reminders: A *tree* is an undirected graph that is connected and acyclic. Given a graph $G = (V, E)$, a *spanning tree* for G is a graph $G' = (V, T)$ such that $T \subseteq E$ and G' is a connected graph. (It is also common to refer to T as the spanning tree). Suppose your graph has edge weights: $\{w_e : e \in E\}$. The cost of a spanning tree T equals $\sum_{e \in T} w_e$. A *minimum spanning tree* is a spanning tree of minimal cost.

In the Minimum Spanning Tree (MST) problem, you are given a connected (undirected) graph $G = (V, E)$ with edge weights $\{w_e : e \in E\}$, and you must compute and output a minimum spanning tree T for G.

1. **Minimum Spanning Trees: implementation.** Two common greedy MST algorithms are:

   - Prim's algorithm: Maintain a set of connected nodes S. Each iteration, choose the cheapest edge (u, v) that has one endpoint in S and one endpoint in $V \setminus S$.
   - Kruskal's algorithm: Start with an empty set of edges T. Each iteration, add the cheapest edge from E that would not create a cycle in T.

   We saw pseudocode for both these algorithms in class. Implementation details **matter a lot** in considering which of these algorithms to use.

   (a) What is the asymptotic running time of Prim's algorithm?
   If you were to implement it (in, say, C++) what data structures would you need? Would you need any additional data structures beyond structures you've seen from CS35? If so, try to design an implementation for them.

   (b) What is the asymptotic running time of Kruskal's algorithm?
   If you were to implement it (in, say, C++) what data structures would you need? Be specific. Would you need any additional data structures beyond structures you've seen from CS35? If so, try to design an implementation for them.

2. **Minimum Spanning Trees: edge weights.** In class we saw the cut property, which stated that for any nonempty subset $S \subsetneq V$ of vertices, the edge $e = (u, v)$ of minimal weight such that $u \in S$ and $v \in V \backslash S$ is in every minimal spanning tree of the given graph.

   There may be more than one minimum spanning tree of a graph. The cut property is worded very carefully: the edge e is in *every* minimum spanning tree.

   However, if edge weights are not distinct then there might be two edges which are tied: both have the smallest weight.

   (a) Given a connected undirected graph G with edge weights from the set $\{1, 2, 3, 4, 5\}$, is there a minimum spanning tree that does not contain some edge e of weight 1 (the minimum weight)?
   If yes, give a graph where this is true. If no, argue why it is not true.

(b) Given a connected undirected graph G with edge weights from the set $\{1, 2, 3, 4, 5\}$, is there a minimum spanning tree that does contain some edge e of weight 5 (the maximum weight)?
If yes, give a graph where this is true. If no, argue why it is not true.

(c) The problem with our cut property seems to be that when edge weights are not distinct, it does not say which edge should be in a minimum spanning tree. Rewrite the cut property so that it covers the case where edge weights are not distinct. (If there is not necessarily a *single* edge of minimum weight, then what should the cut property say?)

(d) Use your new version of the cut property to prove that Prim's algorithm returns a minimal spanning tree, in the case when edge weights are not distinct.

3. **Making change with coins.** Consider the problem of making change for n cents out of the fewest number of coins. Assume that n and the coin values are positive integers (cents).

(a) Describe a greedy algorithm to solve the problem using the US coin denominations of quarters (25), dimes (10), nickels (5), and pennies (1). Prove your algorithm is optimal.

(b) Suppose the country of Algorithmland uses denominations that are powers of c for some integer c. This country uses $k+1$ denominations of $c^0, c^1, \ldots, c^k$. Show that your greedy algorithm works in Algorithmland as well.

4. **A simpler wedding planner problem.** There are n boolean variables x_1, x_2, $\ldots$, x_n. A *literal* is either a variable x_i or its negation $\bar{x}_i$. A 2-*constraint* consists of the OR of two literals $f \vee g$. Think of each variable as a person, a literal as a decision (invite Bob or don't invite Bob?), and a constraint as a description of what the happy couple want in terms of invitations (e.g. $x_i \vee \bar{x}_j$ means invite i or don't invite j).

An assignment sets truth values for each variable. For example, if $n = 3$, one such assignment is $\{x_1 = \text{True}, x_2 = \text{False}, x_3 = \text{True}\}$. An assignment A satisfies a constraint $f \vee g$ if at least one of the literals f or g is satisfied. For example, $x_i \vee \bar{x}_j$ is satisfied if either $x_i = \text{True}$ or $x_j = \text{False}$ (or both).

Give a linear-time algorithm that takes a list of n variables and m 2-constraints and produces a satisfying assignment or returns that no such assignment exists.