Algorithmic Combinatorics Wi 2018–19
Tibor Szabó
Anurag Bishnoi

Exercise Sheet 2

Due date: 16:15, 30th October

You should try to solve all of the exercises below, but clearly mark which two solutions you would like us to grade – each problem is worth 10 points. We encourage you to submit in pairs, but please remember to indicate the author of each solution. Until submission, you are forbidden to look at the solutions of any of these exercises on the internet.

Exercise 1 Give an algorithm to find out whether a graph $G = (V, E)$ is bipartite or not that runs with worst-case time complexity of $O(|V| + |E|)$. Prove the correctness of your algorithm.

Exercise 2 Show that the first m edges added in Kruskal's algorithm form an m-edge forest of minimum weight.

Exercise 3 The purpose of this exercise is to show that the greedy algorithm can fail spectacularly for some problems.

(a) Explain how you would change Kruskal's algorithm (as simply as possible) for the Travelling Salesman Problem on complete graphs with weighted edges: try to greedily build a Hamilton cycle of minimum weight.

(b) Show that for every $\alpha > 1$, there is an edge-weighted graph for which the greedy algorithm builds a Hamilton cycle whose weight is at least α times larger than the optimum.

Exercise 4

(1) Give an example of a connected weighted graph in which some edges are allowed to have negative weights such that Dijkstra's algorithm fails on this graph.

(2) The following algorithm is designed to find lightest paths in *directed* graphs that may have negative edge weights.

```
Algorithm: LIGHTPATHS
Input: A directed graph G = ([n], E⃗), edge weights ω : E⃗ → ℝ, root vertex u ∈ [n]
Result: LIGHTPATHS(G, ω, u) computes, when possible, for every vertex v ∈ [n] a
        lightest directed path (with total weight) from u to v in (G, ω).

/* Initialisation:  start with infinite distance bounds and empty
   paths, except for the root u                                          */
Set dist[u] = 0;
for v ∈ [n] \ {u} do
    Set dist[v] = ∞ ;
    Set prev[v] = null;
end
/* Repeatedly check edges to see if we can improve current paths         */
for 1 ≤ i ≤ n − 1 do
    for edge (x, y) ∈ E⃗ do // check if edge gives shorter paths from u
        if dist[x] + ω((x, y)) < dist[y] then
            Set dist[y] = dist[x] + ω((x, y));
            Set prev[y] = x;
        end
    end
end
/* Run through edges once more to check for %%%%%%%% %%%%%                */
for edge (x, y) ∈ E do
    if dist[x] + ω((x, y)) < dist[y] then
        Return error: graph has a %%%%%%%% %%%%% ;
    end
end
```

(a) Unfortunately the pseudocode got corrupted, and some words were lost. What words should replace the '%' characters towards the end?

(b) Prove that the algorithm runs correctly. What is its running time?

The next couple of exercises concern SAT — the Boolean satisfiability problem — for which we now define the necessarily terminology. A *Boolean variable* is a variable that can take one of two variables: True or False. A *Boolean formula* $f(x_1, x_2, \ldots, x_n)$ is simply a function $f : \{\text{True}, \text{False}\}^n \to \{\text{True}, \text{False}\}$. In other words, it takes as input a number of Boolean variables, $x_1, x_2, \ldots, x_n$, and for every possible truth assignment to these variables, evaluates to either True or False. A Boolean formula f is said to be *satisfiable* if there is some assignment of truth values to its inputs for which f evaluates to True.

Every Boolean formula can be represented by combining the Boolean input variables with three logical operators: '$\wedge$' (and), '$\vee$' (or) and '$\neg$' (not). In particular, every formula has a *Conjunctive Normal Form (CNF)*. A *literal* is either a variable x_i or its negation $\neg x_i$. A *clause* is the 'or' of several literals, so it is satisfied if any one of its literals is. Finally, the CNF formula is the 'and' of all its clauses, and is thus satisfied if and only if all of its clauses are. The Boolean satisfiability problem is the decision problem asking whether or not a given CNF formula is satisfiable.

A k-CNF formula consists of clauses with exactly k literals[1], each corresponding to different variables[2]. For example, the following is a 4-CNF formula:

$$f(x_1, x_2, x_3, x_4, x_5, x_6) = (x_1 \vee \neg x_2 \vee x_4 \vee \neg x_5) \wedge (x_2 \vee x_3 \vee \neg x_4 \vee x_6) \wedge (x_1 \vee \neg x_2 \vee x_5 \vee \neg x_6)\,.$$

This formula is satisfiable, since, for example, $f(\text{True}, \text{False}, \text{True}, \text{True}, \text{False}, \text{False}) = \text{True}$. In general, the k-satisfiability (k-SAT) problem is the Boolean satisfiability problem restricted to k-CNFs.

Exercise 5

(a) Provide an example of an unsatisfiable instance of k-SAT.

(b) Show that every instance of k-SAT with fewer than 2^k clauses must be satisfiable, and show that the bound on the number of clauses is tight.

Exercise 6 (Bonus)

(a) Show that 2-SAT is in $\mathcal{P}$.

(b) Prove that SAT can be (polynomially) reduced to 3-SAT.

[1] Some authors would only ask that there only be at most k literals. However, these are essentially equivalent, since given a clause C with fewer than k literals, we can introduce a new variable y, and replace C with the logically-equivalent $(C \vee y) \wedge (C \vee \neg y)$.

[2] Having literals with the same variable is redundant: $x \vee x$ is just x, while $x \vee \neg x$ is always True.