CS/ECE 374 ✧ Spring 2021
Homework 9

Due Thursday, April 22, 2021 at 10am

Groups of up to three people can submit joint solutions. Each problem should be submitted by exactly one person, and the beginning of the homework should clearly state the Gradescope names and email addresses of each group member. In addition, whoever submits the homework must tell Gradescope who their other group members are.

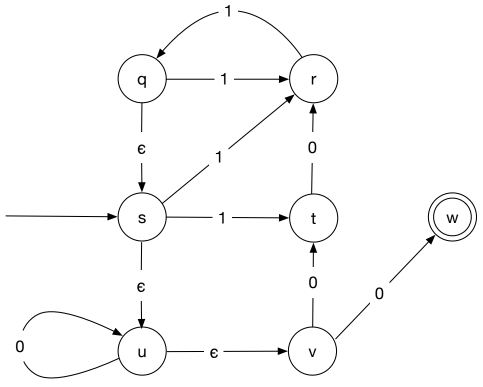
---

1. Let $G = (V, E)$ a directed graph with edge lengths $\ell(e), e \in E$. The edge lengths can be negative. Let $R \subset E$ be red edges (an edge can be red or uncolored). Given s, t and an integer h_r, the goal is is to find the length of a shortest s-t *walk* that contains at most h_r red edges. Note that if the same red edge is repeated ℓ times in a walk then it is counted ℓ in the bound h_r.

   - Describe an instance or example in which the shortest walk length is $-\infty$.
   - Describe an instance or example in which the shortest walk length is finite but there is no path achieving it (in other words one needs a cycle in the walk).
   - Describe an efficient algorithm that finds the shortest walk length from s to t under the given constraints, or reports that it is $-\infty$. The running time of your algorithm should be polynomial in m, n, h_r where $m = |E|, n = |V|$. *Hint:* Under what conditions will the answer be $-\infty$? If walk length is finite what is the maximum number of edges it can contain?

   You may want to see the Lab 11 problem on risky edges.

2. Recall that an NFA N is specified as $(Q, \delta, \Sigma, s, F)$ where Q is a finite set of states, Σ is a finite alphabet, $s \in Q$ is the start state, $F \subseteq Q$ is the set of final (or accepting) states and $\delta : Q \times (\Sigma \cup \{\epsilon\}) \to 2^Q$ is the transition function. Recall that δ^* extends δ to strings: $\delta^*(q, w)$ is the set of states reachable from state q on input string w.



   - In the NFA shown in the figure what is $\delta^*(q, 0)$?

- Describe an efficient algorithm that takes three inputs, a description of an NFA $N = (Q, \Sigma, \delta, s, F)$, a state $q \in Q$, and a symbol $a \in \Sigma$, and computes $\delta^*(q, a)$ (in other words the set of all states that can be reached from q on input a which is now interpreted as a string). You may want to first think about how to compute $\delta^*(q, \epsilon)$ (the ϵ-reach of state q). Express the running time of your algorithm in terms of n the number of states of N and $m = \sum_{p \in Q} \sum_{b \in \Sigma \cup \{\epsilon\}} |\delta(p, b)|$ which is the natural representation size of the NFA's transition function.
- Describe an efficient algorithm that takes two inputs, a description of a NFA $N = (Q, \Sigma, \delta, s, F)$, and a string $w \in \Sigma^*$, and outputs whether N accepts w. Express your running time as a function of $\ell = |w|$ and n and m as in the preceding part.

No proofs necessary but briefly justify your algorithms and running time.

3. In number theory Oppermann's conjecture states the following: For every $n > 1$ there is at least one prime number between $n(n-1)$ and n^2 and at least one prime number between n^2 and $n(n+1)$. This conjecture is still open although it was postulated in 1877. The goal of this problem is to show why being able to solve the Halting problem can easily resolve such open problems.

   - Write pseudocode for a program CheckOppermann($int\, n$) that checks whether Oppermann's conjecture is true or false for a given integer n. You can use a subroutine IsPrime($int\, n$) that checks whether an integer n is a prime number (you can write one yourself if you want). In writing this routine you are not concerned about efficiency but your program should terminate and give the correct answer.
   - Using the program in the preceding part as a sub-routine, write pseudocode for a program OppermannConj() that does not take any input and tries all n in a sequence looking for a counter example. Your program should have the feature that it will halt (if run on a computer) if and only if Oppermann's conjecture is false.
   - Suppose there was a program P that given another program Q can always answer correctly whether Q will halt or not. How can you check whether Oppermann's conjecture is true or not by using P and the preceding part?

   No proofs necessary but your code should be clear.