CS 170 Homework 6

Due **3/4/2024, at 10:00 pm (grace period until 11:59pm)**

1 Study Group

List the names and SIDs of the members in your study group. If you have no collaborators, you must explicitly write "none".

2 2-SAT

In the 2SAT problem, you are given a set of clauses, where each clause is the disjunction (OR) of two literals (a literal is a Boolean variable or the negation of a Boolean variable). You are looking for a way to assign a value true or false to each of the variables so that all clauses are satisfied – that is, there is at least one true literal in each clause. For example, here's an instance of 2SAT:

$$(x_1 \vee \overline{x_2}) \wedge (\overline{x_1} \vee \overline{x_3}) \wedge (x_1 \vee x_2) \wedge (\overline{x_3} \vee x_4) \wedge (\overline{x_1} \vee x_4)$$

Recall that $\vee$ is the logical-OR operator and $\wedge$ is the logical-AND operator and $\overline{x}$ denotes the negation of the variable x. This instance has a satisfying assignment: set x_1, x_2, x_3, and x_4 to `true, false, false, and true`, respectively.

The purpose of this problem is to lead you to a way of solving 2SAT efficiently by reducing it to the problem of finding the strongly connected components of a directed graph. Given an instance I of 2SAT with n variables and m clauses, construct a directed graph $G_I = (V, E)$ as follows.

- G_I has $2n$ nodes: one for each variable and its negation.
- G_I has $2m$ edges: for each clause $(\alpha \vee \beta)$ of I (where α, β are literals), G_I has an edge from from $\overline{\alpha}$ to β, and one from the $\overline{\beta}$ to α.

Note that the clause $(\alpha \vee \beta)$ is equivalent to each of the implications $\overline{\alpha} \implies \beta$ and $\overline{\beta} \implies \alpha$. In this sense, G_I records all implications in I.

(a) Show that if G_I has a strongly connected component containing both x and $\overline{x}$ for some variable x, then I has no satisfying assignment.

(b) Now show the converse of (a): namely, that if none of G_I's strongly connected components contain both a literal and its negation, then the instance I must be satisfiable.

Hint: Pick a sink SCC of G_I. Assign variable values so that all literals in the sink are True. Why are we allowed to do this, and why doesn't it break any implications?

(c) Conclude that there is a linear-time algorithm for solving 2SAT. Provide the algorithm description and runtime analysis; proof of correctness is not required.

Solution:

(a) Suppose there is a SCC containing both x and $\overline{x}$. Notice that the edges of the graph are necessary implications. Thus, if some x and $\overline{x}$ are in the same component, there is a chain of implications which is equivalent to $x \to \overline{x}$ and a different chain which is equivalent to $\overline{x} \to x$, i.e. there is a contradiction in the set of clauses.

(b) Take any sink component, and assign variables so all the literals in this component are True. Because of how we define the graph, there is a corresponding source component which has the negations of all literals in this component. Remove this source/sink component pair, and repeat the process until the graph is empty. Since we set components to true in reverse topological order, there is no implication from a true literal to a false literal. Since no literal and its negation are in the same SCC, we never try to set a variable to be both true and false. So this produces an assignment satisfying all clauses.

(c) Let φ be a formula acting on n literals $x_1, \ldots, x_n$. Construct a graph with $2n$ vertices representing the set of literals and their negations. For each clause $(a \vee b)$ of φ add the edges $\overline{a} \Rightarrow b$ and $\overline{b} \Rightarrow a$. Use the strongly connected components algorithm and for each i, check if there is a SCC containing both x_i and $\overline{x_i}$. If any such component is found, report unsatisfiable. Otherwise, report satisfiable.

(Note: A common mistake is to report unsatisfiable if there is a path from x_i to $\overline{x_i}$ in this graph, even if there is no path from $\overline{x_i}$ to x_i. Even if there is a series of implications which combined give $x_i \to \overline{x_i}$, unless we also know $\overline{x_i} \to x_i$ we could set x_i to False and still possibly satisfy the clauses. For example, consider the 2-SAT formula $(\overline{a} \vee b) \wedge (\overline{a} \vee \overline{b})$. These clauses are equivalent to $a \to b, b \to \overline{a}$, which implies $a \to \overline{a}$, but this 2-SAT formula is still easily satisfiable.)

3 Copper Pipes

Bubbles has a copper pipe of length n inches and an array of nonnegative integers that contains prices of all pieces of size at most n. He wants to find the maximum value he can make by cutting up the pipe and selling the pieces. For example, if length of the pipe is 8 and the values of different pieces are given as following, then the maximum obtainable value is 22 (by cutting in two pieces of lengths 2 and 6).

length	1	2	3	4	5	6	7	8
price	1	5	8	9	10	17	17	20

Give a dynamic programming algorithm so Bubbles can find the maximum obtainable value given any pipe length and set of prices. Clearly describe your algorithm and analyze its runtime (proof of correctness not required).

Solution:

Main idea: We create a recursive formula, where for each subproblem of length k we choose the cut-length i such that $Price(i) + Value(k - i)$ is maximized. Here $Price(i)$ is the price of selling the full pipe of length i and $Value(k - i)$ is the amount obtained after optimally cutting the pipe of length $k - i$.

```
def cutPipe(price, n):
    val = [0 for x in range(n+1)]
    val[0] = 0

    # Build the table val[] in bottom up manner and return
    # the last entry from the table
    for i from 1 to n:
        max_val = 0
        for j from 0 to i-1:
             max_val = max(max_val, price[j] + val[i-j-1])
        val[i] = max_val

    return val[n]
```

Proof: An inductive proof on the length of the pipe will show that our solution is correct. We let $cutPipe(n)$ represent the optimal solution for a pipe of length n. Base case: If the pipe is length 1, $cutPipe(1) = Price(1) = Val(1)$. Inductive: Assume the optimal price is found for all pipes of length less than or equal to k. If the first cut the algorithm makes x_1 is not optimal, then there is an x_1' such that $Val((k+1)-x_1)+Price(x_1) < Val((k+1)-x_1')+Price(x_1')$. By the induction hypothesis, this implies that $cutPipe((k+1)-x_1)+Price(x_1) < cutPipe((k+1)-x_1')+Price(x_1')$. So the algorithm must have chosen x_1' instead of x_1, by construction (a contradiction). Therefore, by induction $cutPipe(n) = Val(n)$ for all $n > 0$.

Run-time: The algorithm contains two nested for-loops resulting in a run-time of $O(n^2)$.

4 Mechanical DP (Optional, Ungraded)

(a) In the longest common substring problem, you are given two strings, $a = a_1a_2 \cdots a_n$ and $b = b_1b_2 \cdots b_m$. You want to find the largest k for which there are indices i and j with $a_ia_{i+1} \cdots a_{i+k-1} = a_ja_{j+1} \cdots a_{j+k-1}$.

For this problem, let us define the following subproblem:

$$dp[i, j] = \text{length of the longest common suffix of } a[1 \ldots i] \text{ and } b[1 \ldots j].$$

(i) For strings a = "compsci" and b = "pompous", fill out remainder of the DP table below.

	c	o	m	p	s	c	i
p					0	0	0
o					0	0	0
m					0	0	0
p	0	0	0		0	0	0
o	0	1	0	0			
u	0	0	0	0			
s	0	0	0	0			

Solution:

	c	o	m	p	s	c	i
p	0	0	0	1	0	0	0
o	0	1	0	0	0	0	0
m	0	0	2	0	0	0	0
p	0	0	0	3	0	0	0
o	0	1	0	0	0	0	0
u	0	0	0	0	0	0	0
s	0	0	0	0	1	0	0

(ii) What's the longest common substring of a and b?

Solution: The longest common substring is "omp", of length 3.

(b) In the longest common subsequence problem, you are given two strings, $a = a_1a_2 \cdots a_n$ and $b = b_1b_2 \cdots b_m$. You want to find the largest k for which there are indices $i_1 < i_2 < \cdots < i_k$ and $j_1 < j_2 < \cdots j_k$ with $a_{i_1}a_{i_2} \ldots a_{i_k} = b_{j_1}b_{j_2} \ldots b_{j_k}$.

For this problem, let us define the following subproblem:

$$dp[i, j] = \text{longest common subsequence of } a[1 \ldots i] \text{ and } b[1 \ldots j].$$

(i) For strings $a = algorithm$ and $b = lithium$, fill out the remainder of the DP table below.

	a	l	g	o	r	i	t	h	m
l	0	1	1	1					
i	0	1	1	1					
t	0	1	1	1					
h	0	1	1	1					
i	0	1	1	1					
u	0	1	1	1					
m	0	1	1	1					

Solution:

	a	l	g	o	r	i	t	h	m
l	0	1	1	1	1	1	1	1	1
i	0	1	1	1	1	2	2	2	2
t	0	1	1	1	1	2	3	3	3
h	0	1	1	1	1	2	3	4	4
i	0	1	1	1	1	3	3	4	4
u	0	1	1	1	1	3	3	4	4
m	0	1	1	1	1	3	3	4	5

(ii) What's the longest common subsequence?

Solution: The longest common subsequence is "lithm", of length 5.

(iii) The table below is a DP table for the longest common subsequence for string X and string Y. What indices of X form the longest common subsequence? List all possibilities.

	$X[0]$	$X[1]$	$X[2]$	$X[3]$	$X[4]$
$Y[0]$	0	0	0	0	0
$Y[1]$	0	1	1	1	1
$Y[2]$	0	1	1	2	2
$Y[3]$	0	1	1	2	3

Solution:

The longest common subsequence is formed by $(X[1], X[3], X[4])$ or $(X[2], X[3], X[4])$.

(iv) Is the following sub-table possible in the longest common subsequence problem? Explain.

	$X[0]$	$X[1]$
$Y[3]$	2	2
$Y[4]$	3	3

Solution: No, it is not possible. Elements in the DP table represent the longest common subsequence thus far. It is not possible for the 0th index of X to have a longest common subsequence of length more than 1.

(v) Is the following sub-table possible in the longest common subsequence problem? Explain.

	$X[0]$	$X[1]$
$Y[0]$	0	1
$Y[1]$	1	2

Solution: No, it is not possible. The element at ($X[1]$, $Y[1]$) must be 1. If $X[1]$ and $Y[1]$ match, then we add 1 to the longest common subsequence formed without $X[1]$ and $Y[1]$ ((X[0], Y[0]) + 1). If $X[1]$ and $Y[1]$ do not match, then the longest common subsequence is the longest common subsequence without $X[1]$ or $Y[1]$ ($\max((X[1], Y[0]), (X[0], Y[1]))$).

5 [Coding] Debugging a MST Algorithm

For this week's coding questions, you won't actually need to implement any algorithms from scratch, but we will walk you through debugging a buggy algorithm implementation!. There are two ways that you can access the notebook and complete the problems:

1. **On Datahub**: click here and navigate to the `hw06` folder.
2. **On Local Machine**: `git clone` (or if you already cloned it, `git pull`) from the coding homework repo,

   `https://github.com/Berkeley-CS170/cs170-sp24-coding`

   and navigate to the `hw06` folder. Refer to the `README.md` for local setup instructions.

Notes:

- *Submission Instructions:* Please download your completed submission `.zip` file and submit it to the Gradescope assignment titled "Homework 6 Coding Portion".
- *Getting Help:* Conceptual questions are always welcome on Edstem and office hours; *note that support for debugging help during OH will be limited.* If you need debugging help first try asking on the public Edstem threads. To ensure others can help you, make sure to:
  1. Describe the steps you've taken to debug the issue prior to posting on Ed.
  2. Describe the specific error you're running into.
  3. Include a few small but nontrivial test cases, alongside both the output you expected to receive and your function's actual output.

  If staff tells you to make a private Ed post, make sure to include *all of the above items* plus your full function implementation. If you don't provide them, we will ask you to provide them.
- *Academic Honesty Guideline:* We realize that code for some of the algorithms we ask you to implement may be readily available online, but we strongly encourage you to not directly copy code from these sources. Instead, try to refer to the resources mentioned in the notebook and come up with code yourself. That being said, we **do acknowledge** that there may not be many different ways to code up particular algorithms and that your solution may be similar to other solutions available online.