Mid-term. February 24, 2017

CS180: Algorithms and Complexity
Winter 2017

Guidelines:

- The exam is closed book and closed notes. Do not open the exam until instructed to do so.
- Write your solutions clearly and when asked to do so, provide complete proofs.
- Unless told otherwise you may use results and algorithms we proved in class without proofs or complete details as long as you state what you are using.
- I recommend taking a quick look at all the questions first and then deciding what order to tackle to them in. Even if you don't solve the problems fully, attempts that show some understanding of the questions and relevant topics will get reasonable partial credit.
- You can use extra sheets for scratch work, but try to use the white space (it should be more than enough) on the exam sheets for your final solutions.
- Most importantly, make sure you adhere to the policies for academic honesty set out on the course webpage. The policies will be enforced strictly and any cheating reported.

Problem	Points	Maximum
1		7
2		3
3		4
4		4
5		4
Total		22

Name	Zhaoyang Xue
UID	104629708
Section	1B

1 Problem

The answers to the following should fit in the white space below the question.

1. Write down Kruskal's algorithm. It is sufficient to write down the main while loop and the rule describing how the algorithm proceeds. [1 point]

   ① Set $T \leftarrow \phi$ $R \leftarrow E$
   ② FOR each edge $e \in R$
   → pick the edge ~~with~~ e with the least weight
   → IF adding e to T will not form a cycle in T
   add e to T
   → remove e from R
   ③ RETURN T.

2. State the *cut property* we used in class to analyze Kruskal's and Prim's algorithms. [1 point]

   Among all the edges across the same cut, the edge with the least weight must be in the minimum spanning tree of the graph.

3. Suppose we are given an instance of the Minimum Spanning Tree Problem on a graph G, with edge costs that are all positive and distinct. Let T be a minimum spanning tree for this instance. Now suppose we replace each edge cost c_e by its square, c_e^2, thereby creating a new instance of the problem with the same graph but different costs.

   True or false: T must still be a minimum spanning tree for this new instance. [1 point]

   True

   Since $c_1 > 0$, $c_2 > 0$. if $c_1 > c_2$, then $c_1^2 > c_2^2$
   Therefore, the least-weighted edges across every cut remain the same.
   Therefore, T remains the same.

4. Consider the weighted interval scheduling problem where we are given n jobs as input with the i'th job having start time s_i, finish time f_i, and value v_i. (Thus, the input to the problem is n triples $(s_1, f_1, v_1), \ldots, (s_n, f_n, v_n)$.) Recall that our goal is to find the set of non-conflicting jobs with the highest possible total value. Consider the following greedy algorithm for the question:

   (a) Set $A = \emptyset$, $R = \{1, 2, \ldots, n\}$.

   (b) While $R \neq \emptyset$:

      i. Pick job $i \in R$ with highest v_i/f_i (value to finish time ratio) and add i to A.

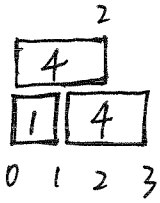
      ii. Remove i and all jobs that conflict with i from R.

   (c) Return A.

   True or false: A achieves the highest possible total value. If true, provide a brief explanation. If false, provide a counterexample. [2 points]

False

counterexample:



$v_1/f_1 = 1/1 = 1$

$v_2/f_2 = 4/2 = 2$

$v_3/f_3 = 4/3$

$\therefore v_2/f_2 > v_3/f_3 > f\, v_1/f_1$

Based on the algorithm, it will pick job2, and the total value is 4 while the optimal solution should be job1 + job3 $\Rightarrow$ total value is 5

5. You have n items with the i'th item having weight w_i. You also have a knapsack with total weight capacity W (i.e., it can safely hold items whose total weight is at most W). Describe an algorithm for picking a *largest* possible subset of items that can be placed safely in the knapsack. That is, describe an algorithm to find a subset $S \subseteq \{1, 2, \ldots, n\}$ of maximum possible size such that $\sum_{i \in S} w_i \leq W$. For full-credit, your algorithm should run in time $O(n \log n)$. You don't have to prove correctness or analyze the time complexity of the algoritm. [2 points]

   [Hint: One approach is to give a greedy algorithm.]

~~(① merge sort them items in increasing order of weights)~~

① set $w = W$, $N = \{1, 2, \ldots, n\}$

~~② From the lightest item to the heaviest~~

② while ~~$w \geq 0$~~ N is not empty

→ pick the lightest item i from $\{1, 2, \ldots, n\}$

→ if $w_i < w$

add w_i to S

→ remove i from N

→ $w = w - w_i$

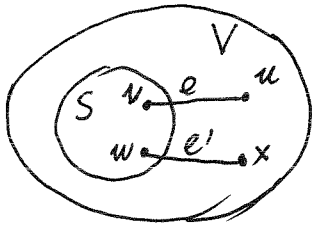
2 Problem

Suppose you have a weighted undirected graph $G = (V, E)$ where all the weights are distinct. Prove that if an edge e is part of a cycle C and has weight more than every other edge in the cycle, then e cannot be part of the minimum spanning tree in G. [3 points]

[Hint: Assume that the statement is false for the sake of contradiction and let T being a MST that contains the edge e. Arrive at a contradiction by a swapping argument as we did in class for proving the cut property.]

~~Suppose the~~
Assume the statement is false and e is in T.
suppose e connects vertices v and u.
Then due to Kruskal's Algorithm, some edge e' in cycle C must be excluded from T. Because adding e' would create a cycle in T.
suppose e' connects vertices w and x.
Since e has the largest weight among all edges within C,
$We > We'$

Now, let's look at a cut S where $v \in S$, $w \in S$
while $u \notin S$, $x \notin S$



e and e' are both edges across the same cut,
so we can replace one by the other.
If we replace e by e', since $We' < We$, the total weight of the new ~~AAST~~ tree would be smaller than the "MST",
which contradicts the hypothesis.

Therefore, the assumption has to be false.
Which means e must not be in T.

3 Problem

Give a dynamic programming algorithm for the following version of knapsack where you have three copies of each item. There are n types of items with weights $w_1, \ldots, w_n$ respectively and *value* $v_1, \ldots, v_n$ respectively and you have **three** copies of each item. Suppose you have a knapsack of total weight capacity W. We a say configuration $(a_1, \ldots, a_n)$ is *safe* if $0 \leq a_i \leq 3$ and $a_1 w_1 + a_2 w_2 + \ldots + a_n w_n \leq W$ (i.e., it is safe to pack a_1 copies of item 1, a_2 copies of item 2, ..., a_n copies of item n into the knapsack). The value of a configuration is the total value of the items in the configuration: for a configuration $(a_1, \ldots, a_n)$, its value is $v_1 a_1 + v_2 a_2 + \cdots + v_n a_n$.

Give an algorithm which given the numbers $w_1, \ldots, w_n$, $v_1, \ldots, v_n$, W as input computes the maximum value achievable over all safe configurations. For full-credit it is sufficient to give a correct algorithm for the problem which runs in time $O(nW)$ and it is not required to prove correctness or analyze the time-complexity of the algorithm. You must provide full description of the algorithm. [4 points]

Let OPT(i, w) represent the maximum total value of choosing from items 1, 2, ..., i (3 copies each) with a total weight ~~compa~~ capacity of w.

Let ~~M[i]~~ M[i,w] of an ~~matrix~~ array holds the value OPT(i,w)

Algorithm: Compute_OPT

① Create an array M[1, 2, ..., n, 1, 2, ..., W], ~~each element M[i,w] = null~~.

② Initialize M[0, w] = 0 for ∀ w = 1, 2, ..., W

M[i, 0] = 0 for ∀ i = 1, 2, ..., n

~~③ If M[i,w] is not empty return M[i,w]~~

④ FOR i = 1, 2, ..., n

FOR w = 1, 2, ..., W

IF $w_i > w$ M[i,w] = M[i-1, w]

ELSE M[i,w] = max
- M[i-1, w] // $a_i = 0$
- M[i-1, w - w_i] + v_i // $a_i = 1$
- M[i-1, ~~2~~(w - 2w_i)] + 2v_i // $a_i = 2$
- M[i-1, ~~3~~(w - 3w_i)] + 3v_i // $a_i = 3$

recurrence relation

⑤ Output M[n, W]

4 Problem

You are given two arrays of integers $X = [x[0], x[1], \ldots, x[m]]$ and $Y = [y[0], y[1], \ldots, y[n]]$ as input. For two subsequences of X, Y of the same length, i.e., sequences of indices $0 \le i_1 < i_2 < \ldots < i_k \le m$ and $0 \le j_1 < j_2 < \ldots < j_k \le n$, the value of the subsequences is defined as

$$\sum_{\ell=1}^{k} \frac{1}{1 + |x[i_\ell] - y[j_\ell]|}.$$

Give an algorithm that given X, Y as input computes the maximum possible value achievable over all subsequences. For full-credit, your algorithm should run in time $O(mn)$ (ignoring the cost of arithmetic, i.e., adding numbers). You don't have to prove correctness or analyze the time-complexity of the algorithm. [4 points]

Example: $X = [1, 4, 2, 5]$, $Y = [1, 2, 10, 4, 100]$. Here, if you look at subsequences $x[0], x[2], x[3]$ and $y[0], y[1], y[3]$ you get value $1/1+1/1+1/2 = 2.5$. Whereas, if look at subsequences $x[0], x[1], x[2], x[3]$ and $y[0], y[1], y[2], y[3]$, you get value $1/1 + 1/3 + 1/9 + 1/2 \sim 1.9444$. So the first subsequence has better value. Your goal is to find the best possible value achievable over all subsequences.

[Hint: Create subproblems like we did for edit-distance in class and develop the appropriate recurrence.]

Basically, what we do is to align X and Y (insert blanks to X and Y)
for example $X_0, X_1, X_2, X_3, X_4 \rightarrow X_0 - X_1\ X_2 - X_3\ X_4$
$Y_0, Y_1, Y_2, Y_3, Y_4 \rightarrow Y_0\ Y_1\ Y_2 - -\ Y_3\ Y_4$

there are only three possibilities for a pair

$\begin{bmatrix} X_i \\ Y_j \end{bmatrix}$: we include both x_i, y_j to the calculation

$\begin{bmatrix} X_i \\ - \end{bmatrix}$: we ignore x_i

$\begin{bmatrix} - \\ Y_j \end{bmatrix}$: we ignore y_j

Therefore, recurrence relation,

$$OPT(a,b) = \max \begin{cases} \frac{1}{1+|x[ia]-y[jb]|} + OPT(a-1, b-1) & \begin{bmatrix} X_a \\ Y_b \end{bmatrix} \\ OPT(a-1, b) & \begin{bmatrix} X_a \\ - \end{bmatrix} \\ OPT(a, b-1) & \begin{bmatrix} - \\ Y_b \end{bmatrix} \end{cases}$$

Compute_Maximum.

① create an array $M[1, 2, \ldots, m, 1, 2, \ldots, n]$
$M[a,b]$ stores $OPT(a,b)$

② Initialize $M[0,0] = \frac{1}{1+|x[0]-y[0]|}$

③ FOR $a = 0, 1, \ldots, m$
FOR $b = 0, 1, \ldots, n$
compute $M[a,b]$ using the recurrence relation.

④ Output $M[m,n]$

5 Problem

Consider the following variant of the RNA sequencing question. Given a sequence $X = (x_1, \ldots, x_n)$, a set of pairs $M = \{(i_1, j_1), (i_2, j_2), \ldots, (i_m, j_m)\}$ is an *allowed* set of pairs if the following hold:

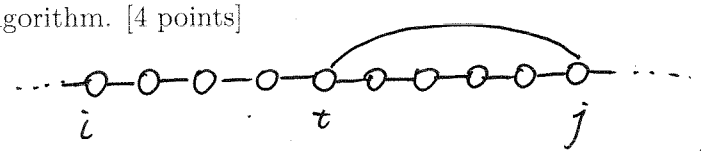
1. Each index appears in at most one pair in M (i.e., no repetitions).
2. Each pair is one of $\{G, C\}$ or $\{A, U\}$. That is, for all $1 \leq p \leq m$, $\{x_{i_p}, x_{j_p}\}$ is one of $\{G, C\}$ or $\{A, U\}$.
3. No sharp edges: For all pairs $(i, j) \in M$, $i < j - 4$.
4. No crossing edges: If pairs $(i, j), (k, \ell) \in M$, then we cannot have $i < k < j < \ell$.

(These are the same rules as we worked with in class.)

The *stability* of an allowed set of pairs M is given by the following formula:

$$stability(M) = \sum_{p=1}^{m} (j_p - i_p)^2.$$

That is, the stability of the collection of pairs is the sum of squares of the number of characters between each pair. Give an efficient algorithm that given a sequence $X = (x_1, \ldots, x_n)$ computes the maximum possible $stability(M)$ over all feasible sets of pairs M. For full-credit, your algorithm should run in $O(n^3)$ time. You do not have to prove correctness or analyze the time complexity of the algorithm. [4 points]



suppose OPT(i,j) represent maximum possible stability over i through j.
an element M[i,j] of an array holds OPT(i,j)

recurrence relation:

$$OPT(i,j) = \max \begin{cases} OPT(i, j-1) & \text{// } j \text{ is not paired} \\ \max \text{ among all } t : (j-t)^2 + OPT(i, t-1) + OPT(t+1, j-1) & \text{// } j \text{ is paired to } t \end{cases}$$

(note that $t < j-4$ and that $\{t, j\}$ is a legal pair.)

Compute_Stability.

① create array M[1,2,...,n, 1,2,...,n]

② initialize M[i,i]=0
M[i,i+1]=0
M[i,i+2]=0
M[i,i+3]=0
M[i,i+4]=0

③ FOR i = 1,2,...,n-5
FOR k = 5,6,...,n-i
→ j = i+k
→ find M[i,j] using the recurrence relation

④ Output M[1,n]