# CS180 Exam 2

TOTAL POINTS

**20 / 22**

QUESTION 1

7 pts

1.1 Kruskal's algorithm **1 / 1**

- **- 0** Correct algorithm

1.2 Cut property **1 / 1**

- **- 0** Property stated correctly

1.3 MST change when squaring weights **1 / 1**

- **- 0** Correct answer

1.4 WIS: value-to-finish time **2 / 2**

- **- 0** Correct answer, with a valid example showing why the greedy algorithm won't work

1.5 Greedy for same value knapsack **2 / 2**

- **- 0** Correct algorithm, that sorts the items by weight, and fills up the knapsack

QUESTION 2

2 Proof of cycle property **3 / 3**

- **- 0** Correct proof.

QUESTION 3

3 Knapsack with 3 copies **4 / 4**

- **- 0** Correct algorithm

QUESTION 4

4 Most valuable subsequence **2.75 / 4**

- **- 1.25** Incomplete code but correct logic or single error which clearly affects the final solution while the form of the recurrence is structurally related to correct solution

QUESTION 5

5 RNA with squared norm stability **3.25 / 4**

- **- 0.75** Right subproblems and loop correct and slight mistake in memoization and recurrence or incomplete code with some initialization.

# 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 | |
| UID | |
| Section | |

## 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]

set $R = E$ sorted in increasing order of weights; $T = \emptyset$

while $R \neq \emptyset$

- pick ~~edge~~ $e \notin T$ that is of smallest weight in $R$
- If adding $e$ ~~to~~ to $T$ doesn't create cycle (BFS check)
  add $e$ to $T$
- delete $e$ from $R$

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

~~If an edge e is not part of set of edges~~

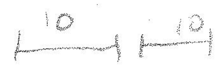
If we have an edge $e = \{u,v\}$ that is of smallest weight that crosses the cut $S$ (where $u$ is a vertex in $S$ & $v \notin S$) then edge $e$ is part of MST

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. Squaring the weights of edges in $G$ doesn't change ordering of weights of edges & kruskal's alg will still pick the same edges

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 $R \neq \emptyset$:

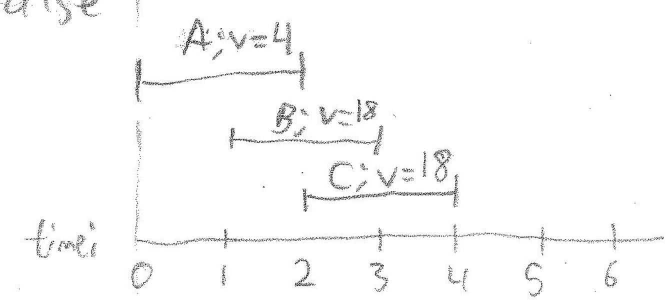
i. Pick job $i \in R$ with highest $v_i/f_i$ (value to finish time ratio) and add $i$ to $A$.

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 !




• Actual optimal: A, C with total weight 22

• Alg actual (wrong): B with total weight 18

$A: \frac{v_A}{f_A} = \frac{4}{2} = 2$

$B: \frac{v_B}{f_B} = \frac{18}{3} = 6$

$C: \frac{v_C}{f_C} = \frac{18}{4} = \frac{9}{2}$

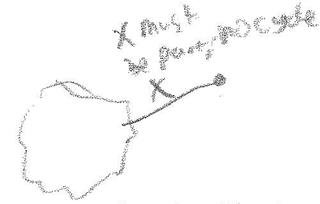
(B will get picked for highest ratio & remove jobs A & C.

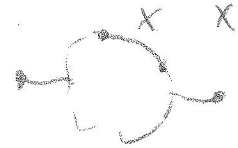
5. You have $n$ items with the $i$'th item having weight $w_i$. You also have a knapsack with total weight capacity $W$ (i.e., it can safely hold items whose total weight is at most $W$). Describe an algorithm for picking a *largest* possible subset of items that can be placed safely in the knapsack. 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. Sort the weights in increasing order (nondecreasing if nondistinct weights)
   a) Initialize set $S$ to hold subset
2. For each $i = 1$ to $n$:
   if $W - w_i \geq 0$:
   add $i$ to $S$ → adding smallest known weighted $i$ to subset
   update $W \leftarrow W - w_i$
3. Return $S$

let X = super big edge




## 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.]

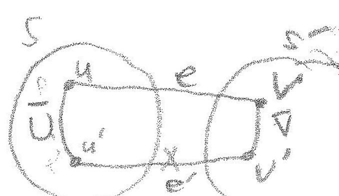
Proof by contradiction:

Lemma: If edge $e$ is part of cycle $C$ & has more weight than other edges in $C$, $e$ can be part of MST in $G$.

So if $T$ is MST of $G$, it contains $e = \{u, v\}$

Because MSTs can't have cycles we thus have to remove an edge $e' = \{u', v'\}$ from $C$. This means that ~~there~~ is a path from $u'$ to $u$ to $v$ to $v'$ via path $P$ around the other direction of cycle, and if $C$ was only cycle in $G$, then removing $e'$ broke all cycles & $T$ is thus a tree & can span to all vertices. Let $c(\text{edge})$ = weight of edge.


Since $c(e) > c(e')$ by lemma definition, this means $T$ although a spanning tree is not the MST. By the cut property, we should only make safe choices such that we choose edge of minimum weight that crosses the cut has to be part of MST. Cut before adding $e$ or $e'$ is:




where $\bar{U}$ = $u$ to $u'$ path & $\bar{U}$ = $v$ to $v'$ path.

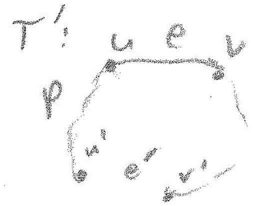
We added $e$ before $e'$ ~~when~~ that crosses cut but $e'$ crosses cut too & is of less weight so $e'$ should have been first. Also $c(\bar{U}) + c(\bar{V}) + c(e) \overset{\geq}{\text{compared to}} c(\bar{U}) + c(\bar{V}) + c(e')$

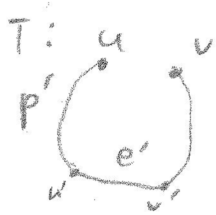
$\bar{V} \cancel{x} c(e) + c(\bar{U}) \overset{?}{\geq} c(\bar{V}) + c(\bar{U}) + c(e')$

can simplify to $c(e) \geq c(e')$ so the sums are uneven. Thus, lemma is false by contradiction since "MST" with $e$ has more weight than ~~MST~~ MST w/o $e$ & has $e'$.

Let $e = \{u, v\}$ & $e' = \{u', v'\}$

Assume $e$ is part of MST, $T'$, & $e$ is heaviest edge in C.
Then we remove $e'$ from C to break cycle. Assume there is path(s) $P$ that connects $u, u', v, v'$ & holds $e$, such that path is part of MST $T'$.
$T'$ connects all vertices and has no cycles so $T'$ is at least spanning tree. We declared length of $P$ that contains $e$ to add total weight to $T'$ that is less than total weight of similar path $P'$ that holds $e'$ & is part of $T$ MST:



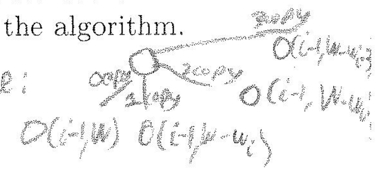

This mean length of $P < P'$ so $T'$ forms a MST. This is a contradiction because we cannot have a MST smaller than what was previously defined as MST so lemma is false.

# 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]

Reccurence:




```
Initialize A of size n by W array:
    For all i = 1 to n:  A[i,0] = 0
    For all w = 1 to W:  A[0,w] = 0
  call opt(n, W).

opt(i, w):
    if i or w is 0, return 0
    if A[i,w] is not empty, return A[i,w]
    else:
      { tempMax = opt(i-1, w)  <- no copies i included
        for K = 1 to 3:
          { if w - (w_i * K) >= 0:
                tempMax = max(tempMax,   opt(i-1, w-w_i*K) + v_i*K)
        A[i,w] = tempMax
    return A[i,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.]

Recurrence: look at last int in arrays:

Case 1: $\boxed{x_m = y_n}$ same int; want $O(m-1, n-1)$

Case 2: $\boxed{x_m \neq y_n}$ diff ints; want: $\rightarrow O(m-1, n)$ or $O(m, n-1)$ or $O(m-1, n-1)$

Initialize array A as m by n 2D array to hold solutions to subproblems;

$A[0,0] = 1/(1+|x[0]-y[0]|)$

for each $i = 1$ to $m$: $A[i,0] = 1/(1+|x[i]-y[0]|)$

for each $i = 1$ to $n$: $A[0,i] = 1/(1+|x[0]-y[i]|)$

for $i = 1$ to $m$;
  for $j = 1$ to $n$:
    if $x[i] = x[j] \rightarrow M[i,j] = \frac{1}{1+|x[i]-y[j]|} + M[i-1, j-1]$ — this is really $= 1$
    else:

$$M[i,j] = \max \begin{cases} \frac{1}{1+|x[i]-y[j]|} + M[i-1, j-1], \\ \frac{1}{1+|x[i-1]-y[j]|} + M[i-1, j], \\ \frac{1}{1+|x[i]-y[j-1]|} + M[i, j-1] \end{cases}$$

return $A[m,n]$

## 5 Problem

Consider the following variant of the RNA sequencing question. Given a sequence $X = (x_1, \dots, x_n)$, a set of pairs $M = \{(i_1, j_1), (i_2, j_2), \dots, (i_m, j_m)\}$ 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 $1 \le p \le m$, $\{x_{i_p}, x_{j_p}\}$ 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 $X = (x_1, \dots, x_n)$ 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] Initialize A as n by n 2D array to hold max stabilities subproblems

~~For all i=1 to n~~

Initialize:
For all k = 0 to 4:
For all i = 1 to n:
A [i, i+k] = 0

for k = 5 to n−1:
for i = 1 to n−k:
j = i+k
temp Max = A [i, j−1] → not include $j^{th}$ index
for t = i to j−4:
if (t, j) is valid matching pair as outlined in rules above:

$$tempMax = max \begin{cases} tempMax, \\ (x_j - x_t)^2 + A[i, t-1] + A[t+1, j] \end{cases}$$
↳ include range t to j

A [i, j] = tempMax

return A [1, n] → highest stable value in A between 1, n