CS 170 Homework 6

Due **Monday 10/11/2024, 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, explicitly write "none".

2 Firefighters

PNPLand is made of N cities that are numbered from $0, 1, \ldots N-1$, which are connected by two-way roads. You are given a matrix D such that, for each pair of cities (a, b), $D[a][b]$ is the distance of the shortest path between a and b. The distances $D[a][b]$ are metric, so you can assume that the triangle inequality always holds: $0 \leq D[a][b] \leq D[a][c] + D[c][b]$ for all a, b, c.

We want to pick K distinct cities and build fire stations there. For each city without a fire station, the response time for that city is given by distance to the nearest fire station. We define the response time for a city with a fire station to be 0. Let R be the maximum response time among all cities. We want to create an assignment of fire stations to cities such that R is as small as possible.

In this problem, we'll devise a greedy algorithm to solve this problem. However, a greedy approach will not always give the optimal solution. Instead, we'll just aim to get a solution that is "good enough".

We can formalize the problem as follows: Suppose the optimal assignment of fire stations to cities produces response time R_{opt}. Given positive integers N, K and the 2D matrix D as input, describe an $O(N^2 \cdot K)$ (or faster) greedy algorithm to output an assignment that achieves a response time of $R_g \leq 2 \cdot R_{\text{opt}}$.

Provide a 3-part solution. For your proof of correctness, show that your algorithm achieves the desired approximation factor of 2.

Solution:

Algorithm Description: Start by building the fire station an arbitrary city, eg city 1. For the remaining $K-1$ fire stations, repeatedly find the city with the largest response time and build a fire station there.

Proof of Correctness: Let R_i be the largest response time when creating the ith fire station in the algorithm (for $i > 1$). Observe that at the ith step of the algorithm, the travel time between any two fire stations is $\geq R_i$. We want to show $R_K = R_g \leq 2R_{\text{opt}}$.

Suppose this algorithm results in a city v whose response time is $R_K > 2R_{\text{opt}}$. Then, the travel time between every pair of fire stations is $> 2R_{\text{opt}}$ (otherwise, we would've placed a fire station at city v during the greedy algorithm).

Now consider the set S of these $K+1$ cities (i.e. the K fire stations and v), and the optimal solution O. Since $|S| = K+1$ and $|O| = K$, by the pigeonhole principle, there must exist an optimal fire station $w \in O$ such that w is the closest fire station to two cities $a, b \in S$. Hence, via the triangle inequality we have

$$D[a][b] \leq D[a][w] + D[w][b] \leq 2R_{\text{opt}}.$$

However, since a, b are in S, we must have $D[a][b] > 2R_{\text{opt}}$. This is a contradiction.

Runtime Analysis: There are K iterations, and each iteration takes $O(N^2)$ time because we look at the distances between every pair of fire stations. Thus, the overall runtime is $O(N^2K)$.

3 Updating a MST

You are given a graph $G = (V, E)$ with positive edge weights, and a minimum spanning tree $T = (V, E')$ with respect to these weights; you may assume G and T are given as adjacency lists. Now suppose the weight of a particular edge $e \in E$ is modified from $w(e)$ to a new value $\hat{w}(e)$. You wish to quickly update the minimum spanning tree T to reflect this change, without recomputing the entire tree from scratch.

There are four cases. In each, give a description of an algorithm for updating T, a proof of correctness, and a runtime analysis for the algorithm. Note that for some of the cases these may be quite brief. For simplicity, you may assume that no two edges have the same weight (this applies to both w and $\hat{w}$).

(a) $e \in E'$ and $\hat{w}(e) < w(e)$

(b) $e \notin E'$ and $\hat{w}(e) < w(e)$

(c) $e \in E'$ and $\hat{w}(e) > w(e)$

(d) $e \notin E'$ and $\hat{w}(e) > w(e)$

Solution:

(a) **Main Idea:** Do nothing.

Correctness: T's weight decreases by $w(e) - \hat{w}(e)$, and any other spanning tree's weight either stays the same or also decreases by this much, so T must still be an MST.

Runtime: Doing nothing takes $O(1)$ time.

(b) **Main Idea:** Add e to T. Use DFS to find the cycle that now exists in T. Remove the heaviest edge in the cycle from T.

Correctness: The heaviest edge in a cycle cannot be in the MST (because if it is in the MST, you can remove it from the MST and add some other edge to the MST, and the MST's cost will decrease), and any edge not in an MST is the heavest edge in some cycle (in particular, the cycle formed by adding it to the MST). For any edge not in T except for e, decreasing e's weight does not change that it is the heaviest edge in the cycle, so it is safe to exclude from the MST. By adding e to T and then removing the heaviest edge in the cycle in T, we remove an edge that is also not in the MST. Thus after this update, all edges outside of T cannot be in the MST, so T is the MST.

Runtime: This takes $O(|V|)$ time since T has $|V|$ edges after adding e, so the DFS runs in $O(|V|)$ time.

(c) **Main Idea:** Delete e from T. Now T has two components, A and B. Find the lightest edge with one endpoint in each of A and B, and add this edge to T.

Correctness: Every edge besides e in the MST is the lightest edge in some cut prior to changing e's weight, and increasing e's weight cannot affect this property. So all edges besides e are safe to keep in the MST. Then, whatever edge we add is also the lightest edge in the cut (A, B) and is thus also in the MST.

Runtime: This takes $O(|V| + |E|)$ time, since it might be the case that almost all edges in the graph might have one endpoint in both A and B and thus almost all edges will be looked at.

(d) **Main Idea:** Do nothing.

Correctness: T's weight does not increase, and any other spanning tree's weight either stays the same or increases, so T must still be the MST.

Runtime: Doing nothing takes $O(1)$ time.

4 Minimum Spanning k-Forest

Given a graph $G(V, E)$ with nonnegative weights, a spanning k-forest is a cycle-free collection of edges $F \subseteq E$ such that the graph with the same vertices as G but only the edges in F has k connected components. For example, consider the graph $G(V, E)$ with vertices $V = \{A, B, C, D, E\}$ and all possible edges. One spanning 2-forest of this graph is $F = \{(A, C), (B, D), (D, E)\}$, because the graph with vertices V and edges F has components $\{A, C\}, \{B, D, E\}$.

The minimum spanning k-forest is defined as the spanning k-forest with the minimum total edge weight. (Note that when $k = 1$, this is equivalent to the minimum spanning tree). In this problem, you will design an algorithm to find the minimum spanning k-forest. For simplicity, you may assume that all edges in G have distinct weights.

(a) Define a j-partition of a graph G to be a partition of the vertices V into j (non-empty) sets. That is, a j-partition is a list of j sets of vertices $\Pi = \{S_1, S_2 \dots S_j\}$ such that every S_i includes at least one vertex, and every vertex in G appears in exactly one S_i. For example, if the vertices of the graph are $\{A, B, C, D, E\}$, one 3-partition is to split the vertices into the sets $\Pi = \{\{A, B\}, \{C\}, \{D, E\}\}$.

Define an edge (u, v) to be crossing a j-partition $\Pi = \{S_1, S_2 \dots S_j\}$ if the set in Π containing u and the set in Π containing v are different sets. For example, for the 3-partition $\Pi = \{\{A, B\}, \{C\}, \{D, E\}\}$, an edge from A to C would cross Π.

Show that for any j-partition Π of a graph G, if $j > k$ then the lightest edge crossing Π must be in the minimum spanning k-forest of G.

(b) Give an efficient algorithm for finding the minimum spanning k-forest.

Please give a 3-part solution.

Solution:

(a) It helps to note that when $j = 2, k = 1$ this is exactly the cut property. A similar argument lets us prove this claim.

For some j-partition Π where $j > k$, suppose that e is the lightest edge crossing Π but e is not in the minimum spanning k-forest. Let F be the minimum spanning k-forest. Now, consider adding e to F. One of two cases occurs:

- $F + e$ contains a cycle. In this case, some edge e' in this cycle besides e must cross Π. This means $F + e - e'$ is a spanning k-forest, since deleting an edge in a cycle cannot increase the number of components. Since e by definition is cheaper than e', the forest $F + e - e'$ is cheaper than F, which contradicts F being a minimum spanning k-forest.
- $F + e$ does not contain a cycle. In this case, the endpoints of e are in two different components of F, so $F + e$ has $k - 1$ components. Since Π is a j-partition and $j > k$, some edge e' in F must cross Π. Deleting an edge from a forest increases the number of components in the forest by only 1, so $F + e - e'$ has k components,

i.e. is a k-forest. Since e by definition is cheaper than e', the forest $F + e - e'$ is cheaper than F, which contradicts F being a minimum spanning k-forest.

In either case, we arrive at a contradiction and have thus proven the claim.

(b) There are multiple solutions, we recommend the following one because its proof of correctness follows immediately from part a:

Main Idea: The algorithm is to run Kruskal's, but stop when $n - k$ edges are bought, i.e. the solution is a spanning k-forest.

Correctness: Any time the algorithm adds an edge e, let $S_1 \dots S_j$ be the components defined by the solution Kruskal's arrived at prior to adding e. $S_1 \dots S_j$ form a j-partition and by definition of the algorithm, $j > k$. e is the cheapest edge crossing this j-partition, so by part a e must be in the (unique) minimum spanning k-forest. Since every edge we add is in the minimum spanning k-forest, our final solution must be the minimum spanning k-forest.

Runtime Analysis: This is just modified Kruskal's so the runtime is $O(|E| \log |V|)$ (Kruskal's runtime is dominated by the edge sorting, so the fact that we may make less calls to the disjoint sets data structure because the algorithm terminates early does not affect our asymptotic runtime).

5 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 k in range(n+1)] # val[k] denotes the value of a length-k pipe
    val[0] = 0

    # Fill in the array val[] via bottom-up DP, and return last entry
    for i from 1 to n:
        max_val = 0
        for j from 1 to i:
            max_val = max(max_val, price[j] + val[i-j])
        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 Hypothesis: Assume the optimal price is found for all pipes of length less than or equal to k; i.e. $cutPipe(j) = Val(j)$ for all $j \leq k$.

Inductive Step: If the first cut x_1 that the algorithm makes is not optimal, then there exists 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 since we take the max in the inner for-loop. We have thus arrived at a contradiction.

Hence, by induction $cutPipe(n) = Val(n)$ for all $n > 0$, implying that our algorithm outputs the optimal solution.

Runtime Analysis: The algorithm contains two nested for loops, resulting in a runtime of $O(n^2)$.

6 [Coding] Simple Dynamic Programming

For this week's homework, you'll implement implement some simple dynamic programming algorithms. 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-fa24-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.

hw06

October 25, 2024

1 Dynamic Programming

In this notebook, we'll explore solving the Longest Increasing Subsequence problem and the Longest Path on DAGs problem using dynamic programming.

1.0.1 If you're using Datahub:

- Run the cell below **and restart the kernel if needed**

1.0.2 If you're running locally:

You'll need to perform some extra setup. #### First-time setup * Install Anaconda following the instructions here: https://www.anaconda.com/products/distribution * Create a conda environment: `conda create -n cs170 python=3.11` * Activate the environment: `conda activate cs170` * See for more details on creating conda environments https://conda.io/projects/conda/en/latest/user-guide/tasks/manage-environments.html * Install pip: `conda install pip` * Install jupyter: `conda install jupyter`

Every time you want to work

- Make sure you've activated the conda environment: `conda activate cs170`
- Launch jupyter: `jupyter notebook` or `jupyter lab`
- Run the cell below **and restart the kernel if needed**

[20]:
```
# Install dependencies
!pip install -r requirements.txt --quiet
```

[62]:
```
import otter
assert (otter.__version__ >= "5.5.0"), "Please reinstall the requirements and␣
 ↪restart your kernel."

grader = otter.Notebook("hw06.ipynb")
import time
import tqdm
import pickle
import numpy as np
import networkx as nx

test_cases = pickle.load(open("generated_testcases.pkl", "rb"))
```

```
rng_seed = 42
```

1.0.3 Q1. Longest Increasing Subsequence

First implement the longest increasing subsequence. The algorithm is explained here https://people.eecs.berkeley.edu/~vazirani/algorithms/chap6.pdf#page=3.

The algorithm discussed in lecture and the textbook only returns the length of the longest increasing subsequence. Here we want you to return the actual subsequence (the actual list of elements). To find the actual subsequence, it may be useful to maintain an array seperate from the dp array which can be used to reconstruct the actual sequence.

[28]:
```
def longest_increasing_subsequence (arr, n):
    """
    Return a list containing longest increasing subsequence of the array.
    If there are ties, return any one of them.

    args:
        arr:List[int] = an array of integers
        n:int = an int representing the length of arr

    return:
        List[int] Containing the longest increasing subsequence. Return the␣
 ↪actual
        elements, not the indices of the elements.
    """
    # BEGIN SOLUTION

    # compute the DP array
    dp = [1]*n
    prev = [i for i in range(n)]
    for i in range(n):
        for j in range(i):
            if arr[i] > arr[j] and dp[i] <= dp[j]:
                dp[i] = dp[j] + 1
                prev[i] = j

    # reconstruct the sequence
    end = dp.index(max(dp))
    sequence = []
    while prev[end] != end:
        sequence.append(arr[end])
        end = prev[end]
    sequence.append(arr[end])
    sequence.reverse()

    return sequence
```

```
    # END SOLUTION
```

1.0.4 Debugging

A simplified verion of the otter tests are pasted here for your convenience. Feel free to add whatever print statements or assertions you'd like when debugging.

[29]:
```
def check_subsequence(seq, arr):
    for i in range(len(seq) - 1):
        assert seq[i] < seq[i + 1], f"Your subsequence is not strictly␣
 ↪increasing: {seq}"

    index = 0
    matched = 0
    while matched < len(seq) and index < len(arr):
        if seq[matched] == arr[index]:
            matched += 1
        index += 1
    assert matched == len(seq), f"your list is not a valid subsequence of the␣
 ↪input list."
assert tqdm is not None

problems = test_cases['q1']
for arr, sol in tqdm.tqdm(problems, total=len(problems)):
    student_sol = longest_increasing_subsequence(arr, len(arr))

    assert len(student_sol) == len(sol), f"""The length of your list differs␣
 ↪from the solution. Your list {student_sol}, the solution {sol}"""
    check_subsequence(student_sol, arr)
```

```
100%|      | 358/358 [00:00<00:00, 897.06it/s]
```

Note: your solution should not take inordinate amounts of time to run. If it takes more than 10 seconds to run, it is too slow

We will also check submissions for hardcoded solutions.

[ ]:
```
grader.check("LIS")
```

1.0.5 Representing graphs in code (Part 2!!!)

Unlike last week's assignment, our graphs are now weighted, so we'll need to store weights alongside the edge information. Using an edge list representation, we can represent directed edges (u, v) with weight w by creating a list of tuples `(u, v, w)`.

However, like last week, we'd like to represent our graph using adjacency lists. We can represent the directed edge (u, v) with weight w by storing the tuple `(v, w)` in `adj_list[u]`.

[42]:
```
def generate_adj_list(n, edge_list):
    """
    args:
        n:int = number of nodes in the graph. The nodes are labelled with␣
 ↪integers 0 through n-1
        edge_list:List[Tuple[int,int,int]] = edge list where each tuple (u,v,w)␣
 ↪represents the directed
            and weighted edge (u,v,w) in the graph
    return:
        A List[List[Tuple[int, int]]] representing the adjacency list
    """
    adj_list = [[] for i in range(n)]
    for u, v, w in edge_list:
        adj_list[u].append((v, w))
    for nodes in adj_list:
        nodes.sort()
    return adj_list

def draw_graph(adj_list):
    """Utility method for visualizing graphs

    args:
        adj_list (List[List[Tuple[int, int]]]): adjacency list of the graph␣
 ↪given by generate_adj_list
    """
    G = nx.DiGraph()
    for u in range(len(adj_list)):
        for v, w in adj_list[u]:
            G.add_edge(u, v, weight=w)
    nx.draw(G, with_labels=True)
```

1.0.6 Q2. Longest (Simple) Path in DAGS

It is thought that on general graphs, there is no efficient algorithm to solve the Longest Simple Path problem on general graphs. The main reason for this difficulty is that it is generally difficult to make an algorithm that ignores cycles in a graph.

However, there is an efficient dynamic programming algorithm to find the longest path on DAGs, since these graphs have no cycles. Specifically, the way to find longest paths on a DAG is the exact same as finding the shortest path on a DAG, except at each step you take the maximum rather than minimum distance. See https://people.eecs.berkeley.edu/~vazirani/algorithms/chap6.pdf for more details.

In this context, "longest" means largest sum of edge weights, regardless of the number of edges in the path.

You may assume all test cases are directed acyclic graphs, so you don't need to check for cycles. Some test cases are already topologoically sorted for you, but others are not. Feel free to use code from previous homeworks to help you.

[76]:
```
# BEGIN SOLUTION
# HELPER: Post-number code modified from Hw03
def get_post(adj_list):
    time = 1
    post = []

    n = len(adj_list)
    visited = [False]*n

    def explore(u):
        nonlocal time
        nonlocal visited
        visited[u] = True
        time += 1
        for v, _ in adj_list[u]:
            if not visited[v]:
                explore(v)
        post.append((u, time))
        time += 1
    for i in range(n):
        if not visited[i]:
            explore(i)

    return post
# END SOLUTION
def longest_path_on_DAGS(adj_list):
    """
    Return a list containing the longest path on the dag. If there are ties,␣
 ↪return
    any such path. If there are none, return the empty list.

    args:
    adj_list: an adjacency list representing the DAG.

    return: the longest path as a list of nodes the list [a, b, c, d, e]␣
 ↪correspondes
            to the path a -> b -> c -> d -> e
    """
    # BEGIN SOLUTION
    n = len(adj_list)

    dp = [0 for i in range(n)] # not -infinity, as our path can start from␣
 ↪anywhere!
    prev = [None for i in range(n)]

    topo_order = [node for node, _ in get_post(adj_list)]
```

```
    for u in topo_order[::-1]:
        neighbors = adj_list[u]
        for v, w in neighbors:
            if dp[v] < dp[u] + w:
                dp[v] = dp[u] + w
                prev[v] = u

    # find optimal path end using a linear scan
    t = dp.index(max(dp))

    path = []
    curr = t
    while prev[curr] is not None:
        path.append(curr)
        curr = prev[curr]
    path.append(curr)
    path.reverse()
    return path
    # END SOLUTION
```

1.0.7 Debugging

A simplified verion of the otter tests are pasted here for your convenience. Feel free to add whatever print statements or assertions you'd like when debugging.

[77]:
```
problems = test_cases['q2']
for adj_list in tqdm.tqdm(problems, total=len(problems)):
    G = nx.DiGraph({u: {v: {'weight': w} for v, w in neighbors} for u,␣
 ↪neighbors in enumerate(adj_list)})

    # bans networkx
    nxall = nx
    def error(*args, **kwargs):
        nx = nxall
        raise Exception("You may not call any graph libraries, modules, or␣
 ↪functions.")
    nx = error

    try:
        path = longest_path_on_DAGS(adj_list)
    finally:
        nx = nxall

    # checks that the path returned is a real path in the graph and that it␣
 ↪starts and ends
    # at the right vertices
```

```
    assert nx.is_simple_path(G, path), f"your algorithm did not return a valid␣
↪simple path"

    # checks that the path returned is the longest path
    opt_path_length = nx.dag_longest_path_length(G)
    student_path_length = sum(G[path[i]][path[i+1]]['weight'] for i in␣
↪range(len(path)-1))
    assert student_path_length == opt_path_length, f"your algorithm did not␣
↪return the shortest path"
```

```
100%|      | 21/21 [00:00<00:00, 5974.79it/s]
```

[ ]:
```
grader.check("DAG-longest-path")
```

1.1 Submission

Make sure you have run all cells in your notebook in order before running the cell below, so that all images/graphs appear in the output. The cell below will generate a zip file for you to submit.

[ ]:
```
grader.export(pdf=False, force_save=True, run_tests=True)
```