Exercises

Please do the exercises on your own.

1. (6 pt.) In class, we saw pseudocode for Dijkstra’s algorithm which returned shortest distances but not shortest paths. In this exercise we’ll see how to adapt it to return shortest paths. One way to do that is shown in the pseudocode below:

```
Dijkstra_st_path(G, s, t):
    for all v in V, set d[v] = Infinity
    for all v in V, set p[v] = None
    // we will use the information p[v] to reconstruct the path at the end.
    d[s] = 0
    F = V
    D = [] // D is the list of "done" vertices
    while F isn’t empty:
        x = a vertex v in F so that d[v] is minimal
        for y in x.outgoing_neighbors:
            d[y] = min( d[y], d[x] + weight(x,y) )
            if d[y] was changed in the previous line, set p[y] = x
        F.remove(x)
        D.add(x)
    // use the information in p to reconstruct the shortest path:
    path = [t]
    current = t
    while current != s:
        current = p[current]
        add current to the front of the path
    return path, d[t]
```

Step through `Dijkstra_st_path(G, s, t)` on the graph `G` shown below. Complete the table below (on the next page) to show what the arrays `d` and `p` are at each step of the algorithm, and indicate what path is returned and what its cost is. If it is helpful, the `\LaTeX` code for the table is reproduced at the end of the PSET.
We are expecting: The following things:

- The table below filled out
- The shortest path and its cost that the algorithm returns.

No justification is required.

<table>
<thead>
<tr>
<th>We are expecting: The following things:</th>
</tr>
</thead>
<tbody>
<tr>
<td>• The table below filled out</td>
</tr>
<tr>
<td>• The shortest path and its cost that the algorithm returns.</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th></th>
<th></th>
<th></th>
<th></th>
<th></th>
<th></th>
<th></th>
<th></th>
<th></th>
</tr>
</thead>
<tbody>
<tr>
<td>When entering the first while loop for the first time, the state is:</td>
<td>0</td>
<td>∞</td>
<td>∞</td>
<td>∞</td>
<td>None</td>
<td>None</td>
<td>None</td>
<td>None</td>
</tr>
<tr>
<td>Immediately after the first element of D is added, the state is:</td>
<td>0</td>
<td>3</td>
<td>∞</td>
<td>9</td>
<td>None</td>
<td>s</td>
<td>None</td>
<td>s</td>
</tr>
<tr>
<td>Immediately after the second element of D is added, the state is:</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>Immediately after the third element of D is added, the state is:</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>Immediately after the fourth element of D is added, the state is:</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

2. (2 pt.) Consider the graph below:

Step through the Floyd-Warshall algorithm from Lecture 12 on this graph (using the order suggested by the vertex labels), and write down what your tables $D^{(i)}$ look like for $i = 0, 1, 2, 3$. (You may either code this up or do it by hand).

If it helps, here is the \LaTeX code for one such table:

\begin{tabular}{c|c|c|c|c|c|c|c}
| $D^{(i)}$ | 1 & 2 & 3 | 3 & 2 & 1 | 2 & 1 |
|----------|------|------|------|------|------|------|------|
| 1 & - & - & - | 3 & 2 & 1 | 2 & 1 |
| 2 & - & - & - | 2 & - & - | 1 & 3 |
| 3 & - & - & - | 3 & 2 & 1 | 2 & 1 |
\end{tabular}

[We are expecting: Your filled-in tables. No explanation is required.]
3. (6 pt.) Consider the recurrence relation defined by

\[ T(n) = 2T(n - 1) + T(n - 2) + 1, \]

with \( T(0) = T(1) = 0. \)

(a) (3 pt.) Write a **bottom-up** dynamic programming algorithm that computes \( T(n) \). Your algorithm should run in time \( O(n) \).

[We are expecting: Pseudocode. No explanation is required.]

(b) (3 pt.) Write a **top-down** dynamic programming algorithm that computes \( T(n) \). Your algorithm should run in time \( O(n) \).

[We are expecting: Pseudocode. No explanation is required.]
Problems

You may talk with your fellow CS161-ers about the problems. However:

- Try the problems on your own before collaborating.
- Write up your answers yourself, in your own words. You should never share your typed-up solutions with your collaborators.
- If you collaborated, list the names of the students you collaborated with at the beginning of each problem.

4. (5 pt.) (Dijkstra with negative edges) For both of the questions below, suppose that \( G \) is a connected, directed, weighted graph, which may have negative edge weights (but not infinite edge weights), containing vertices \( s \) and \( t \), and refer to the pseudocode for `Dijkstra_st_path` from Exercise 1. Suppose that there is some path from \( s \) to \( t \) in \( G \).

(a) (2 pt.) Give an example of a graph where there is a path from \( s \) to \( t \), but no shortest path from \( s \) to \( t \). (Note that in a directed graph, a path must follow the direction of the edges; recall that a shortest path is one which minimizes the sum of the edge weights along that path).

[We are expecting:
- A small example (at most 5 vertices)
- An explanation of why there is no shortest path from \( s \) to \( t \).
]

(b) (3 pt.) Give an example of a graph where there is a shortest path from \( s \) to \( t \), but `Dijkstra_st_path(G, s, t)` does not return one.

[We are expecting:
- A small example (at most 5 vertices)
- An explanation of what `Dijkstra_st_path` does on this graph and why it does not return a shortest path.
]
5. (10 pt.) [Longest Paths.] For this problem, let $G = (V, E)$ be a weighted directed acyclic graph (DAG) with $n$ vertices and $m$ edges. In this problem you will design a dynamic programming algorithm to find the length of a longest path in $G$.

(a) (1 pt.) Plucky the Pedantic Penguin is a bit worried about the phrase “longest path in $G$” above. They are concerned because in the graph below, for example, there doesn’t seem to be any longest path from vertex $v_1$ to vertex $v_2$: you can have a path $v_1 \rightarrow v_2 \rightarrow v_3 \rightarrow v_1 \rightarrow v_2 \rightarrow v_3 \rightarrow \cdots \rightarrow v_1 \rightarrow v_2$ that’s arbitrarily long! Explain why Plucky doesn’t need to be worried: that is, explain why, in this case, the length of a longest path is well-defined and finite.

![Graph](image)

[We are expecting: One or two sentences explaining why Plucky doesn’t need to be worried.]

(b) (3 pt.) Suppose that $v_0,v_1,\ldots,v_{n−1}$ is a topological ordering of the vertices in $V$. For $k \in \{0,\ldots,n−1\}$, let $P[k]$ denote the length of the longest path in $G$ which ends at the vertex $v_k$. (Here, “longest” means highest-cost, according to the edge weights).

State a formula which defines $P[k]$ in terms of $P[j]$ for $0 \leq j < k$.

[We are expecting: A formula of the form $“P[k] = (something which may reference P[0], P[1], \ldots, P[k−1], and the structure of G),”$ and base case(s). You don’t need to explain why your formula is correct.]

(c) (4 pt.) Develop a dynamic programming algorithm (either bottom-up or top-down, your choice) that uses your relationship from part (b) to find the length of the longest path in $G$. Your algorithm should run in time $O(n + m)$. You may assume that the vertices are stored in an adjacency-list format so that $V[i].inNeighbors()$ returns a list of the indices $j \in \{0,\ldots,n−1\}$ so that $(v_j, v_i) \in E$. You may also assume that for vertices $v_i,v_j$, you can get the weight $w(v_i,v_j)$ in time $O(1)$.

[We are expecting: Pseudocode. You do not need to include an English explanation, although you may if you think it will make your answer more clear. You do not need to justify the running time.]

(d) (2 pt.) Why did we consider the vertices in a topological order? Give an example where your algorithm fails if it considers the vertices in a different order. (Unless you think that your algorithm works for any order, in which case just say that).

Note: don’t augment your algorithm to first find a topological order and then say “it works if the vertices are not in topological order because my algorithm puts them in topological order.” Rather, this question is asking whether or not the core dynamic programming part of your algorithm still works.

[We are expecting: An example of a graph and an ordering on the vertices where the algorithm fails, and a short explanation of what goes wrong. If you think that your algorithm works on any order, just say that.]

---

1. Edge weights could possibly be negative.
Feedback

There’s no “correct” answer here, and it is completely anonymous.

6. (1 pt.) Please fill out the following poll which asks about office hours.

https://forms.gle/91cq69FanUUQxEESA

Did you fill out the poll?

[We are expecting: The answer “yes.”]
Optional problem

This problem is completely optional. It’s not worth any points (not even bonus points) and we won’t grade it. However, it might be fun to think about!

7. (0 pt.) In this optional problem we’ll see how to compute Fibonacci numbers even faster!

(a) In class, we saw the following bottom-up algorithm to compute Fibonacci numbers:

```
def fasterFibonacci(n):
    F = [1 for i in range(n+1)]
    for i in range(2,n+1):
        F[i] = F[i-1] + F[i-2]
    return F[n]
```

This was much faster than the naive divide-and-conquer approach. But what actually is the runtime of this? In class we said it’s \( O(n) \): we have \( n \) iterations of the loop and we’re just adding two numbers inside the loop. But when we look at the running times for really large \( n \), it looks like this:

![Computing Fibonacci Numbers](image)

That doesn’t look linear! Our argument seems pretty reasonable, at least by the standards we’ve been using in this class. So what’s going on? (Note: We haven’t really been formal enough in this class to give you the tools to argue this formally. The point of this optional problem is just to get you to think a bit.) **Hint:** How large is \( F[n] \)? How many bits does it take to represent \( F[n] \) in binary? How much time does it take to add \( F[n-2] + F[n-1] \)?

(b) Let \( M \) be the matrix \( M = \begin{bmatrix} 0 & 1 \\ 1 & 1 \end{bmatrix} \). Argue that

\[
M^n \cdot \begin{bmatrix} 1 \\ 1 \end{bmatrix} = \begin{bmatrix} F[n] \\ F[n+1] \end{bmatrix}.
\]

(c) Come up with an algorithm that uses \( O(\log n) \) multiplications (of unbounded size) to compute \( F[n] \), when \( n \) is a power of 2.

(d) If you take into account how big the numbers you are multiplying are in the above, how long would your algorithm from the previous part take? What if you use a fast multiplication algorithm like Karatsuba? What if you use a multiplication algorithm which takes time \( O(n \log(n) \log \log(n)) \) to multiply \( n \)-bit numbers? (This latter thing exists, it’s called the Schonhage-Strassen algorithm).
Helpful \LaTeX code

Here is the code for the table from Exercise 1:

\begin{center}
def\arraystretch{1.5}
newcommand{\td}{\texttt{d}}
newcommand{\tp}{\texttt{p}}
begin{tabular}{|p{6cm}|c|c|c|c||c|c|c|c|}
\hline
& \td\[$s$] & \td\[$u$] & \td\[$v$] & \td\[$t$] & \tp\[$s$] & \tp\[$u$] & \tp\[$v$] & \tp\[$t$] \\
\hline
When entering the first while loop for the first time, the state is:
0 & $\infty$ & $\infty$ & $\infty$ & None & None & None & None \\
\hline
Immediately after the first element of $D$ is added, the state is:
0 & $3$ & $\infty$ & $9$ & None & $s$ & None & $s$ \\
\hline
Immediately after the second element of $D$ is added, the state is:
& & & & & & & \\
\hline
Immediately after the third element of $D$ is added, the state is:
& & & & & & & \\
\hline
Immediately after the fourth element of $D$ is added, the state is:
& & & & & & & \\
\hline
\end{tabular}
\end{center}