Problem Set 8

All parts are due on April 19, 2018 at 11PM. Please write your solutions in the LaTeX and Python templates provided. Aim for concise solutions; convoluted and obtuse descriptions might receive low marks, even when they are correct. Solutions should be submitted on the course website, and any code should be submitted for automated checking on `py.mit.edu/6.006`.

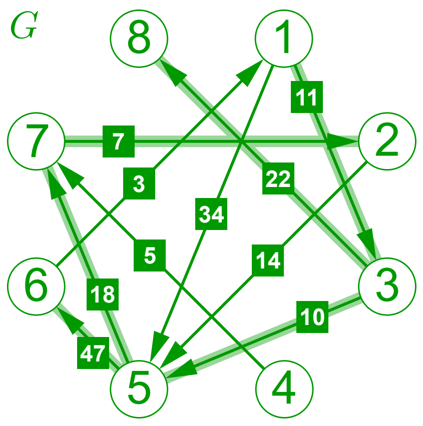
Problem 8-1. [30 points] **Shortest Paths**

(a) [10 points] **Dijkstra Practice:** Below is is a list of triples representing a weighted directed graph G on 8 integer labeled vertices. A triple (i, j, w) represents a directed edge in the graph from vertex v_i to vertex v_j having weight w.

```
G = [(1, 3, 11),
     (1, 5, 34),
     (2, 5, 14),
     (3, 5, 10),
     (3, 8, 22),
     (4, 7,  5),
     (5, 6, 47),
     (5, 7, 18),
     (6, 1,  3),
     (7, 2,  7)]
```

- [2 points] Draw graph G by placing vertices to lie roughly on a circle ordered clockwise by index. Be sure to label edge weights!
- [8 points] Run Dijkstra to compute the shortest path distance $\delta_s(v_i)$ from source vertex $s = v_1$ to every vertex v_i in the graph.

Solution:



Running Dijkstra's algorithm maintains shortest path estimates for each vertex. Students need only list values of $\delta_s(v_i)$ at the completion of the algorithm, but for completeness, we provide shortest path estimates $d_s(v_i)$ after each vertex is processed by Dijkstra:

	$d_s(v_1)$	$d_s(v_2)$	$d_s(v_3)$	$d_s(v_4)$	$d_s(v_5)$	$d_s(v_6)$	$d_s(v_7)$	$d_s(v_8)$
Initial	0	∞	∞	∞	∞	∞	∞	∞
v_1	(0)	∞	11	∞	34	∞	∞	∞
v_3	(0)	∞	(11)	∞	21	∞	∞	33
v_5	(0)	∞	(11)	∞	(21)	68	39	33
v_8	(0)	∞	(11)	∞	(21)	68	39	(33)
v_7	(0)	46	(11)	∞	(21)	68	(39)	(33)
v_2	(0)	(46)	(11)	∞	(21)	68	(39)	(33)
v_6	(0)	(46)	(11)	∞	(21)	(68)	(39)	(33)
v_4	(0)	(46)	(11)	(∞)	(21)	(68)	(39)	(33)
δ	0	46	11	∞	21	68	39	33

Rubric:

- 2 points for correct drawing of graph
- If incorrect, give points for $\delta_s(v_i)$ based on graph drawn
- 1 point for each correct $d_s(v_i)$ for $i \in [1, 8]$

(b) [5 points] **Longest Paths:** A path in a graph is **simple** if it visits any vertex at most once. Fellman Bord claims that one can find a longest simple path between two vertices in an arbitrary weighted graph by negating edge weights and running Bellman-Ford once in the modified graph. Argue that Fellman Bord's algorithm correctly finds a longest simple path, or argue that the algorithm is not correct for all inputs.

Solution: Bellman-Ford will not necessarily compute the longest paths in the original graph, since there might be a negative-weight cycle reachable from the source, and the algorithm will abort. Finding a longest simple path in a graph is actually an NP-hard problem, meaning that it is unlikely that a polynomial time algorithm exists to solve this problem; we will discuss what NP-Hard means in more detail later in the term.

Rubric:

- 5 points for a correct argument
- Partial credit may be awarded

(c) [15 points] **Longer Shortest Paths:** Given a connected undirected graph $G = (V, E)$ with positive edge weights, describe algorithms to find a minimum weight path (if one exists) between two specified vertices that uses: (1) **exactly** k edges in $O(k|E|)$ time, and (2) **at least** k edges in $O(k|E| + |V| \log |V|)$ time.

Solution: Let s and t be the two specified vertices. For part (1), to find a shortest path from s to v using exactly k vertices, we construct a graph on $(k+1)|V|$ vertices: (v, i) for each $v \in V$ and $i \in [0, k]$. Connect a directed edge from vertex $(u, i-1)$ to (v, i)

for each edge $(u, v) \in E$ and for every $i \in [1, k]$. Since traversing an edge adds one to the second index of any vertex, this graph is acyclic and a path from vertex $(s, 0)$ to a vertex (v, k) will represent a (possibly non-simple) path traversing exactly k edges from s to t in the original graph. To find such a path of minimum weight, relax edges from vertices in a topological sort order in $O((k+1)|V| + k|E|) = O(k|E|)$ time, since $|V| = O(|E|)$ in a connected graph, resulting in shortest path weight $\delta_s(v)_k$ for a path from s to v using exactly k edges.

For part (2), we take the original graph and add an auxiliary node a, with a directed edge (a, v) to each vertex $v \in V$, weighted by $\delta_s(v)_k$, the shortest path weight from s to each vertex using exactly k edges. A minimum weight path π from s to t using at least k edges will have as a prefix a minimum weight path using exactly k edges ending at some vertex u. Further, the (possibly empty) subset of the path π from u to t cannot have cycles (as they would have positive weight and could be removed), so finding a shortest path from a to t in this new graph will correspond to a minimum weight path using at least k edges. This graph has only positive weights, so we can apply Dijkstra in $O(|V| \log |V| + |E|)$ time (using a Fibonacci Heap to implement its queue). Constructing the graph takes $O(k|E|)$ time using the method from part (1), so this algorithm takes $O(k|E| + |V| \log |V|)$.

Rubric:

- 6 points for a correct algorithm for part (1)
- 1 point for running time analysis for part (1)
- 6 points for a correct algorithm for part (2)
- 2 points for running time analysis for part (2)
- Partial credit may be awarded

Problem 8-2. [30 points] **Consulting**

For each of the following scenarios, apply a graph algorithm that best solves the problem: describe a graph related to the problem, choose or modify an existing algorithm to apply to the graph, justify your choice of algorithm, and state your algorithm's running time in terms of problem parameters.

(a) [10 points] **Rover Scheduling:** NASA scientists need to plan a task schedule for the Mars rover. Each task requires a certain amount of time to complete. One task may be required to complete prior to another. For example, before transmitting soil sample data to Earth, the samples must have been collected, and the antennae battery must have been charged. Given a list of rover tasks including their durations and dependencies, describe an algorithm to compute the shortest time to complete all tasks.

Solution: Construct a graph with a vertex for each task, and a directed edge from task a to task b if a must be completed prior to b, weighted by the **negation** of the time required to complete task a. Further, add vertices s and t, with a directed edge from s to every other vertex in the graph and a directed edge from every vertex in the graph to t, each with weight 0. If there are n tasks and m dependencies, this graph has size $O(n + m)$. Use depth-first search to check whether this graph contains and cycles; if it does, then no task on the cycle can be first, so no plan can be constructed. Otherwise, finding the shortest path from s to t will correspond to the longest time to complete any sequence of tasks, with all other tasks being performed in parallel. Since the graph is acyclic, relax edges in topological sort order to compute the path of minimum weight in $O(n + m)$ linear time, which is as faster than other methods.

Rubric:

- 4 points for graph description
- 2 points for choosing a correct algorithm (even if inefficient)
- 2 points for justifying Topo-Sort Relaxation
- 2 points for correct running time analysis based on chosen algorithm
- Partial credit may be awarded

(b) [10 points] **Secret Correspondence:** A network of spies operate in Easteros. Each spy has a list of associates with whom they communicate regularly. Whenever a particular pair of spies meet, there is a known probability that the information discussed will be leaked. The Mother of Turtles needs to send a secret message via this network from her spy in the capitol to a scouting spy at the boarder of the kingdom. When a message is transferred through the network via a sequence of spies, the **risk** of transferring the message is the maximum probability of **any individual pair of spies** in the sequence leaking the information. Assuming the spy network is sufficiently connected to be able to transmit the message, describe an algorithm to determine a sequence of spies to convey the message that minimizes risk.

Solution: Construct a graph with a vertex for each of the n spies and an unweighted edge between two spies if they are associated, weighted by the risk of the association (say there are m associations). The goal is to find a path from some spy s to another spy t in this network such that the maximum weight (risk) of any edge in the path is minimized. Since all weights are positive, run Dijkstra to compute the shortest path from s to t, but when relaxing edge (u, v), only update the value of $d(v)$ if $d(v) > \max(d(u), w(u, v))$. Just like the triangle inequality, this condition is safe (will not undershoot the minimum) and obeys optimal substructure (paths that minimize risk contain sub-paths that also minimize risk). Otherwise, Dijkstra will run normally, finding a path satisfying the desired property in $O(m + n \log n)$ by using a Fibonacci Heap as the priority queue used in Dijkstra. Dijkstra is faster than Bellman-Ford, and the graph may have cycles and many weights, so linear time methods do not apply.

Rubric:

- 4 points for graph description
- 2 points for choosing a correct algorithm (even if inefficient)
- 2 points for justifying Dijkstra
- 2 points for correct running time analysis based on chosen algorithm
- Partial credit may be awarded

(c) [10 points] **Purchasing Parts:** Faylee Krye is the mechanic on board a Sirefly-class spaceship named Ferenity. While docked at a spaceport, she heads into town to purchase six parts for the ship: a power converter, a flux capacitor, a hyper drive, an ion shield generator, a binary motivator, and an air freshener. Faylee has downloaded a map of the town including the location of the parts distributors and time estimates to travel long any particular road in town (you may assume that the number of parts distributors is large). Looking down the list of distributors, Faylee observes that each store sells at most one of the parts, so she will need to visit six different stores to buy everything. Assume that the time spent in any store is negligible compared to transit time. Describe an algorithm to find a route for Faylee to travel from the spaceport and back, purchasing all the needed spaceship parts in the minimum amount of time.

Solution: Let n be the number of stores and road intersections, let m be the number of the roads between stores and intersection in the town, and let s be location of the spaceport. Construct a graph on $2^6 \times n$ vertices of the form $(b_1, b_2, b_3, b_4, b_5, b_6, v)$, denoting that Faylee arrives at store or intersection v while either possessing or not possessing item i, depending on whether boolean b_i is True (1) or False (0) respectively. For every directed road (u, v) between stores or intersections u and v, add many directed edges weighted by the time to traverse the road. Specifically add an edge from vertex $(b_1, b_2, b_3, b_4, b_5, b_6, u)$ to vertex $(b_1, b_2, b_3, b_4, b_5, b_6, v)$ (Faylee did not buy anything at u), one edge for every assignment of the b_is; and if u is a store selling a desired part, say part 1 without loss of generality, add a directed edge from vertex $(0, b_2, b_3, b_4, b_5, b_6, u)$ to vertex $(1, b_2, b_3, b_4, b_5, b_6, v)$, one edge for every assignment of the remaining b_is. This graph has $2^6 n$ vertices and $(2^6 + 2^5)m$ edges, so the graph has size $O(n + m)$ if n and m are large compared to 2^6. A route that corresponds to purchasing all six items and returning home will be a path from vertex

$(0,0,0,0,0,0,s)$ to vertex $(1,1,1,1,1,1,s)$ in the graph. Under the assumption that it takes positive time to traverse any road, we can run Dijkstra to find the shortest such path in $O(m+n\log n)$ time (using a Fibonacci Heap). Dijkstra is faster than Bellman-Ford, and the graph may have cycles and many weights, so linear time methods do not apply.

Rubric:

- 4 points for graph description
- 2 points for choosing a correct algorithm (even if inefficient)
- 2 points for justifying Dijkstra
- 2 points for correct running time analysis based on chosen algorithm
- Partial credit may be awarded

Problem 8-3. [40 points] **Crypto-Exchange**

Chad Chaddingsworth is an unlikeable millennial investor who trades crypto-currencies for a living on the BoinCase exchange. The BoinCase exchange allows users to convert between real currencies and crypto-currencies at posted exchange rates that change each day. For example, yesterday Chad purchased 5 CitBoins for 10 US dollars at an exchange rate of 0.5 CitBoins per US dollar. BoinCase exchange rates are always positive, but not necessarily symmetric, so the exchange rate from US dollars to CitBoins might be smaller (or possibly larger) than 2.0. Given a daily list of BoinCase exchange rates, Chad wants to find a sequence of trades that he can perform that day and make money. More specifically, beginning with x US dollars to invest at the beginning of the day, Chad wants to end the day with more than x US dollars by performing a sequence of conversions on the exchange.

(a) [15 points] Design an algorithm to determine whether Chad can make money on the exchange, and if so, provide a sequence of currencies from US dollars to US dollars for which the sequence of conversions makes money. **Hint:** Products can be transformed into sums using logarithms.

Solution: Let $r(i,j)$ represent the conversion rate from currency i to currency j. Our goal is to find a sequence of currency exchanges $(v_0,\ldots,v_k)$ such that

$$\prod_{i=1}^{k} r(v_{i-1},v_i) > 1.$$

We can convert this optimization to finding a negative sum by taking a negative logarithm of both sides: $\sum_{i=1}^{k} -\log(r(v_{i-1},v_i)) < 0$.

Construct a graph with a vertex for each of the n currencies on the exchange. Add a directed edge from vertex u to vertex v weighted by $-\log(r(u,v))$. Assuming there is an exchange rate between every pair of currencies, there will be $n(n-1)$ edges in the graph. Run Bellman-Ford from the vertex associates with US dollars to find a negative weight cycle of weight $-c$. If the vertex associated with US dollars is on the cycle, return the cycle. Otherwise, convert from dollars to an arbitrary vertex v on the cycle and back to dollars. This single loop has weight d. Compute k such that $d - kc < 0$. Then return the single loop with k repetitions of the negative weight cycle in between, representing a conversion sequence that will make money. This algorithm takes $O(n \times n^2)$ time to find the negative weight cycle using Bellman-Ford, but the output sequence may be arbitrarily large depending on the value of k; however an implicit representation of the sequence could be returned in $O(n^3)$ time.

Rubric:

- 3 points for graph description
- 1 point for logarithmic edge weights
- 3 points for application of Bellman-Ford
- 5 points for a correct algorithm to find a cycle to and from dollars
- 3 points for a correct running time analysis
- Partial credit may be awarded

(b) [25 points] Write the Python function `make_money(exchange_rate)` that implements your algorithm from part (a). Currencies are labeled with integers from 0 to $n - 1$ inclusive, with 0 label of US dollars. The input `exchange_rate` is a length n list of length n lists, where element `exchange_rate[i][j]` corresponds to the exchange rate for converting currency i into currency j on the exchange. Your function should return a list of integer labels between 0 and $n-1$ inclusive, starting and ending with label 0, corresponding to a sequence of currencies that can be converted in order to make money. If no sequence of conversions can make money in this way, then your function should return `None`. Submit your code online at `py.mit.edu/6.006`.

Solution:

```python
import math
def make_money(exchange_rate):
    V = len(exchange_rate)  # Bellman-Ford
    rate = [[-math.log(r) for r in row] for row in exchange_rate]
    d = [float('inf') for _ in exchange_rate]
    parent = [None for _ in exchange_rate]
    d[0] = 0
    for _ in range(V - 1):  # relax edges in V - 1 rounds
        changed = False
        for i in range(V):
            for j in range(V):
                if (d[i] + rate[i][j]) < d[j]:
                    changed = True
                    d[j] = d[i] + rate[i][j]
                    parent[j] = i
        if not changed:
            break
    start = None             # check for cycle
    for i in range(V):
        for j in range(V):
            if (d[i] + rate[i][j]) < d[j]:
                parent[j] = i
                start = j
                break
        if start is not None:
            break
    if start is None:        # no cycle
        return None
    seen = {}                # walk back to find vertex on cycle
    while start not in seen:
        seen[start] = True
        start = parent[start]
    weight = rate[0][start] + rate[start][0]
    if weight < 0:           # if cycle from 0 to vertex is negative, done!
        return [0, start, 0]
    cycle_weight, cycle, node = 0, [start], parent[start]
    while node != start:
        if node == 0:        # if 0 on cycle, start from 0
            start, cycle, cycle_weight = 0, [0], 0
        cycle_weight += rate[parent[node]][node]
        cycle.append(node)
        node = parent[node]
    cycle_weight += rate[parent[node]][node]
    cycle.reverse()
    if cycle[-1] == 0:       # if 0 on cycle, return cycle
        return [0] + cycle
    k = math.ceil(-weight / cycle_weight)
    return [0, start] + k * cycle + [0]      # otherwise, traverse k times
```