Solutions to Homework Seven

CSE 101

1. *Textbook problem 4.11.* Given a directed graph $G = (V, E)$ with positive edge lengths $l(e)$, we want to find the length of the shortest cycle in G.

```
shortest = ∞
For each node u in V:
   dist[·] = dijkstra(G, l, u)
   For each node v in V:
      if (v, u) ∈ E:
         shortest = min{shortest, dist[v] + l(v, u)}
if shortest = ∞:
   output ‘‘acyclic’’
else:
   output shortest
```

   Pick any edge $e = (v, u)$ in the shortest cycle. Then this cycle consists of the shortest path from u to v (which can be found by Dijkstra's algorithm), followed by the single edge e. The procedure above iterates through all possibilities for u in the outer loop, and all possibilities for v in the inner loop.

   A naive implementation of Dijkstra's algorithm has a running time of $O(|V|^2)$, so each node u is processed in this amount of time. The total running time is then $O(|V|^3)$.

2. *Textbook problem 4.19.* We are given a directed graph $G = (V, E)$ with positive edge lengths $l(e)$ and positive vertex costs $c(u)$. We now define the `cost` of a path to be its length, plus the costs of all vertices on the path. We will find paths of minimum cost by reducing to the usual shortest paths problem.

   To this end, consider modifying the edge lengths:

$$l'(u, v) = l(u, v) + c(v)$$

   (in other words, increase the length of each edge by the cost of its endpoint). Then the cost of any path $s \to u_1 \to u_2 \to \cdots \to u_k$ is

$$\begin{aligned} & c(s) + l(s, u_1) + c(u_1) + l(u_1, u_2) + c(u_2) + \cdots + l(u_{k-1}, u_k) + c(u_k) \\ = \quad & c(s) + l'(s, u_1) + l'(u_1, u_2) + \cdots + l'(u_{k-1}, u_k). \end{aligned}$$

   In other words, the cost of a path is simply its l'-length, plus $c(s)$. So we can find paths of minimum cost by running Dijkstra's algorithm on G, with edge lengths l'.

```
for all (u, v) ∈ E:
   l'(u, v) = l(u, v) + c(v)
dist[·] = dijkstra(G, l', s)
Increase all dist values by c(s)
```

   The running time of this algorithm is that of Dijkstra's algorithm.

3. *Textbook problem 4.20.* Let's assume G is undirected (the directed case is similar). Here's the algorithm.

```
ds[·] ← dijkstra(G,l,s)
dt[·] ← dijkstra(G,l,t)
pick (u', v') ∈ E' to minimize min{ds[u'] + l(u', v') + dt[v'], ds[v'] + l(u', v') + dt[u']}
```

Justification: The distance from s to t using a new stretch of road (u', v') in addition to G is simply the sum of the distances $s \to u' \to v' \to t$ (or alternatively $s \to v' \to u' \to t$). The two calls to Dijkstra's algorithm precompute all distances from s and all distances from t (equivalently, *to* t).

Running time: There are two calls to Dijkstra's algorithm, plus a linear-time scan over E', for a total running time of $O(|E'| + |E| \log |V|)$.

4. *One-dimensional set cover.* We are given points $x_1, \ldots, x_n$ on the line, as well as intervals $[s_1, e_1], \ldots, [s_m, e_m]$. The goal is to find the minimum number of intervals that cover all the x_i.

   Here's the idea: suppose you sort the points so that $x_1 \leq x_2 \leq \cdots \leq x_n$. You have to cover x_1, so you might as well begin by picking the interval that covers it and as many of $x_2, x_3, \ldots$ as possible. If this interval covers $x_1, \ldots, x_k$, then remove them and start again at x_{k+1}.

   ```
   Sort points so that x_1 ≤ x_2 ≤ ··· ≤ x_n
   Repeat until all points are covered:
      Let x_j be the first uncovered point
      Pick the interval that covers x_j and as many as possible of x_{j+1}, x_{j+2}, ...
   ```

   To argue that this scheme is correct, suppose the algorithm chooses intervals

   $$\mathcal{C} = [s_{(1)}, e_{(1)}], [s_{(2)}, e_{(2)}], \ldots, \text{ where } s_{(1)} \leq s_{(2)} \leq \cdots .$$

   And let an optimal solution be

   $$\mathcal{C}^* = [s^*_{(1)}, e^*_{(1)}], [s^*_{(2)}, e^*_{(2)}], \ldots, \text{ where } s^*_{(1)} \leq s^*_{(2)} \leq \cdots .$$

   Claim. *Substituting* $[s^*_{(1)}, e^*_{(1)}] \to [s_{(1)}, e_{(1)}]$ *in* $\mathcal{C}^*$ *also yields a valid solution.*

   Proof. $[s^*_{(1)}, e^*_{(1)}]$ must contain x_1. Since our algorithm starts by explicitly choosing the interval with x_1 and the most other points, it follows that any x_i in $[s^*_{(1)}, e^*_{(1)}]$ is also in $[s_{(1)}, e_{(1)}]$. □

   This means that the first interval chosen by the algorithm is part of an optimal solution. Once it removes the covered points, it gets a smaller version of the original problem, and can recurse.

5. *String reconstruction, revisited.*

   (a) Here is the logic:

   $$\begin{aligned} & \text{There is a path from 1 to } n \text{ in graph } G \\ \Leftrightarrow\ & \text{There are } 1 = i_1 < i_2 < \cdots < i_k = n+1 \text{ such that all } (i_j, i_{j+1}) \in E \\ \Leftrightarrow\ & \text{There are } 1 = i_1 < i_2 < \cdots < i_k = n+1 \text{ such that all } x[i_j \cdots i_{j+1} - 1] \text{ are valid words} \\ \Leftrightarrow\ & x[\cdot] \text{ is a valid sequence of words} \end{aligned}$$

   Creating the graph involves testing $O(n^2)$ strings $x[i \cdots j]$ to see if they are valid words; thereafter, all that is needed is a call to `explore`. Therefore the total running time is $O(n^2)$.

   (b) Create the same graph as in (a). Notice that it is a DAG. Find the path from 1 to $n+1$ that has the minimum number of edges, using our algorithm (from class) for shortest paths in DAGs. The running time is still $O(n^2)$.

   Here's the code:

   ```
   Create graph G
   (dist[·], prev[·]) = dag-shortest-paths(G, 1)
   S = (empty stack)
   j = n + 1
   ```

```
while j > 1:
    push(S, x[prev[j] ··· j − 1])
    j = prev[j]
while S is not empty:
    print pop(S)
```

6. (a) *Problem 6.3.* Yuckdonald's.

Subproblem: Let $P(j)$ be the maximum profit achievable using only the first j locations. We want $P(n)$.

Recursive formulation: In figuring out $P(j)$, we have two choices: either place a restaurant at location j or don't. In the former case, we must skip all locations less than k miles to the left of location j. To help with this, define $\texttt{prev}[j]$ to be the largest index i with $m_i \leq m_j - k$, or 0 if there is no such index. Then,

$$P(j) = \max(p_j + P(\texttt{prev}[j]),\ P(j-1)).$$

For consistency, we'll set $P(0) = 0$.

Algorithm:

```
(First compute the prev[·] array)
i = 0
for j = 1 to n:
    while m_{i+1} ≤ m_j − k:
        i = i + 1
    prev[j] = i
(Now the dynamic programming begins)
P[0] = 0
for j = 1 to n:
    P[j] = max(p_j + P[prev[j]], P[j − 1])
return P[n]
```

Running time: $O(n)$.

(b) The greedy strategy doesn't work. Consider a case where $k = 10$ and the locations are $m_1 = 10, m_2 = 20, m_3 = 25, m_4 = 30, m_5 = 40$, with profits $p_1 = 100, p_2 = 100, p_3 = 101, p_4 = 100, p_5 = 100$. The greedy scheme chooses locations $1, 3, 5$, with profit 301, but the best solution is $1, 2, 4, 5$, with profit 400.