Assignment 4

CSci 3110: Introduction to Algorithms

Sample Solutions

Question 1 Denote the points by $p_1, p_2, \ldots, p_n$, ordered by increasing x-coordinates. We start with a few observations about the structure of bitonic tours and paths, which will help us to derive a dynamic programming algorithm for computing a shortest bitonic tour.

Observation 1 Points p_{n-1} and p_n are neighbours in any bitonic tour that visits points $p_1, p_2, \ldots, p_n$.

Proof. Assume that p_{n-1} is not a neighbour of p_n. Then let p_i and p_j be the two neighbours of p_n. The tour visits points p_i, p_j, p_{n-1}, p_n in the order p_i, p_n, p_j, p_{n-1}. However, both p_i and p_j have smaller x-coordinates than p_{n-1} and p_n. Hence, the tour cannot be bitonic. $\square$

Observation 1 implies that edge (p_{n-1}, p_n) is present in any bitonic tour that visits all points. Hence, to find a shortest such tour, it suffices to concentrate on minimizing the length of the bitonic path from p_{n-1} to p_n that is obtained by removing edge (p_{n-1}, p_n) from the tour. We make the following observation about the structure of this path: Let p_k be the neighbour of p_n on this path. If we remove p_n, we obtain a bitonic path P' from p_k to p_{n-1}. If we remove p_{n-1} from P', we obtain a bitonic path that visits points $p_1, p_2, \ldots, p_{n-2}$ and has p_{n-2} as one of its endpoints. So let us concentrate on bitonic paths between any two points p_i and p_j, $i < j$, that visit all points $p_1, p_2, \ldots, p_j$. We call such a path a *normal* bitonic path. Observe that the path from p_{n-1} to p_n that we want to compute is normal. Next we prove that shortest normal bitonic paths have an optimal substructure.

Observation 2 Given a normal bitonic path P with endpoints p_i and p_j, $i < j$, let p_k be the neighbour of p_j on this path. Then the path P' obtained by removing p_j from P is a normal bitonic path with endpoints p_i and p_k. In particular, $p_{j-1} \in \{p_i, p_k\}$. If P is a shortest normal bitonic path with endpoints p_i and p_j, then P' is a shortest normal bitonic path with endpoints p_i and p_k.

Proof. Clearly, if we remove an endpoint from a bitonic path, the resulting path is still bitonic. Hence, P' is a bitonic path with endpoints p_i and p_k. Moreover, it has to visit all points $p_1, p_2, \ldots, p_{j-1}$ because P visits all points $p_1, p_2, \ldots, p_j$, and p_j is the only point we have removed from P to obtain P'. Now assume that $p_{j-1} \notin \{p_i, p_k\}$. Then P visits points p_i, p_k, p_{j-1}, p_j in the order p_i, p_{j-1}, p_k, p_j. Since p_i and p_k have x-coordinates less than those of p_{j-1} and p_j, P cannot be bitonic, a contradiction.

Now let us prove that P' is shortest if P is shortest. Assume that there exists a shorter normal bitonic path P'' from p_i to p_k. Then we would obtain a shorter bitonic path P''' from

p_i to p_j by appending edge (p_k, p_j) to P''. Indeed, since $x(p_j) > x(p_k)$, P''' is bitonic; it is normal, as it visits all points $p_1, p_2, \ldots, p_j$ and has p_j as an endpoint; and it is shorter than P because $\ell(P) = \ell(P') + \mathrm{dist}(p_k, p_j) > \ell(P'') + \mathrm{dist}(p_k, p_j) = \ell(P''')$. □

From Observation 2, we obtain another simple observation that allows us to derive a formula for computing the length of a shortest normal bitonic path from p_{n-1} to p_n.

Observation 3 Consider the neighbour p_k of p_j in a normal bitonic path P with endpoints p_i and p_j, $i < j$. If $i = j - 1$, we have $1 \leq k < i$. If $i < j - 1$, we have $k = j - 1$.

Proof. In the first case, p_k has to be a point in $\{p_1, p_2, \ldots, p_j\} \setminus \{p_i, p_j\}$. This leaves us with exactly the listed choices. In the second case, we obtain from Observation 2 that one of the endpoints of the subpath of P obtained by removing p_j from P must be p_{j-1}. Since $p_i \neq p_{j-1}$, we have $p_k = p_{j-1}$. □

Making the observation that there exists only one normal bitonic path from p_1 to p_2, namely the one consisting of edge (p_1, p_2), we obtain the following formula for computing the length $\ell(i, j)$ of a shortest normal bitonic path with endpoints p_i and p_j, $i < j$:

$$\ell(i, j) = \begin{cases} \mathrm{dist}(p_i, p_j) & \text{if } i = 1 \text{ and } j = 2 \\ \ell(i, j-1) + \mathrm{dist}(p_{j-1}, p_j) & \text{if } i < j - 1 \\ \min_{1 \leq k < i}(\ell(k, i) + \mathrm{dist}(p_k, p_j)) & \text{if } j > 2 \text{ and } i = j - 1 \end{cases}.$$

Before we present our algorithm to compute a shortest bitonic travelling-salesman tour, we make the following observations about the necessary information to construct such a tour, once we have computed all values $\ell(i, j)$, $1 \leq i < j \leq n$. We can obtain a shortest normal bitonic path from p_i to p_j by choosing the correct neighbour p_k in such a path, recursively finding a shortest normal bitonic path from p_i to p_k, and finally appending edge (p_k, p_j). How do we find this "correct" neighbour? If $i < j-1$, there is only one choice: $p_k = p_{j-1}$, by Observation 3. If $i = j-1$, p_k is the point that minimizes the expression $\ell(k, i) + \mathrm{dist}(p_k, p_j)$ because this is the value we have assigned to $\ell(i, j)$. So, in order to construct a shortest bitonic travelling-salesman tour, we only have to record, for every pair (i, j), which is the neighbour p_k of p_j in a shortest normal bitonic path from p_i to p_j. We store this information in an array N; that is, $N[i, j]$ stores the index k of this neighbour p_k.

One other observation is in order: We have to make sure that, whenever we compute a value $\ell(i, j)$, the values $\ell(i', j')$ this computation relies on have already been computed. Now, according to our formula above, the computation of $\ell(i, j)$ relies on values $\ell(k, j - 1)$. Hence, if we fill in the table column by column—assuming that we use i to index the rows and j to index the columns—everything is in order. What remains to be done is to list the algorithm:

Bitonic-TSP(p)

```
 1 n ← |p|
 2 ▷ Compute ℓ(i, j) and N(i, j), for all 1 ≤ i < j < n.
 3 for j = 2..n
 4    do for i = 1..j − 1
 5           do if i = 1 and j = 2
 6                 then ℓ[i, j] ← dist(p[i], p[j])
 7                      N[i, j] ← i
 8                 else if j > i + 1
 9                         then ℓ[i, j] ← ℓ[i, j − 1] + dist(p[j − 1], p[j])
10                              N[i, j] ← j − 1
11                         else ℓ[i, j] ← +∞
12                              for k = 1..i − 1
13                                 do q ← ℓ[k, i] + dist(p[k], p[j])
14                                    if q < ℓ[i, j]
15                                      then ℓ[i, j] ← q
16                                           N[i, j] ← k
17 ▷ Construct the tour. Stacks S[1] and S[2] will be used to construct the two x-monotone
   parts of the tour.
18 Let S be an array of two initially empty stacks S[1] and S[2].
19 k ← 1
20 i = n − 1
21 j = n
22 while j > 1
23    do Push(S[k], j)
24       j ← N[i, j]
25       if j < i
26          then swap i ↔ j
27               k ← 3 − k
28 Push(S[1], 1)
29 while S[2] is not empty
30    do Push(S[1], Pop(S[2]))
31 for i = 1..n
32    do T[i] ← Pop(S[1])
33 return T
```

The final question to be answered concerns the running time of the algorithm. It is easy to see that Lines 17–33 take linear time. Indeed, the loop in Lines 31–32 is executed n times and performs constant work per iteration. The loop in Lines 29–30 is executed $|S[2]|$ times, and each iteration takes constant time; hence, it suffices to prove that $|S[2]| \leq n$ after the execution of Lines 17–28. To prove this, it suffices to show that the loop in Lines 22–27 is executed at most $n - 1$ times because every iteration pushes only one entry onto stack $S[1]$

or $S[2]$. The loop in Lines 22–27 takes constant time per iteration. It is executed until $j = 1$. However, initially $j = n$, and the computation performed inside the loop guarantees that j decreases by one in each iteration. Hence, the loop is executed at most $n - 1$ times.

To analyze the running time of Lines 1–16, we first observe that this part of the algorithm consists of two nested loops; the code in Lines 5–16 is executed $\Theta(n^2)$ times because i runs from 2 to n and in every iteration of the outer loop, j runs from 1 to $i - 1$. Now, we perform constant work inside the loop unless $i = j - 1$. In the latter case, we execute the loop in Lines 12–16 $i - 1 = \mathcal{O}(n)$ times. Since there are only $n - 1$ pairs (i, j) such that $1 \le i = j - 1 < n - 1$, we spend linear time in only $n - 1$ iterations of the two outer loops and constant time in all other iterations. Hence, the running time of Lines 1–16 is $\mathcal{O}(n^2 \cdot 1 + n \cdot n) = \mathcal{O}(n^2)$.

Question 2 (20 marks)

a. In order to use a greedy algorithm to solve this problem, we have to prove that the problem has optimal substructure and the greedy-choice property. The former is easy to prove: Let n be an amount for which we want to give change. Assume that optimal change for n cents includes a coin of denomination d. Then we obtain optimal change for n cents by giving optimal change for $n-d$ cents and then adding this d-cent coin. Indeed, if we could use less coins to give change for $n - d$ cents, we could also use less coins to give change for n cents by adding a d-cent coin to the set of coins we give as change for $n - d$ cents.

   The greedy-choice property is harder to prove. First, what is an obvious greedy choice to make? Well, if we want to give change for n cents, a greedy way to try to minimize the number of coins we use is to start with a coin of largest denomination d such that $d \le n$. We include this coin in the change and recursively give optimal change for $n - d$ cents. Let us prove that this works with denominations $d_0 = 1, d_1 = 5, d_2 = 10, d_3 = 25$.

   We prove by induction on i and n that optimal change using denominations $d_0, \dots, d_i$ always includes $\lfloor n/d_i \rfloor$ coins of denomination d_i. This then immediately implies that optimal change does indeed always include at least one coin of the highest denomination $d_i \le n$, which is the greedy choice property we want to prove.

   So consider the case $i = 0$. Then the only choice we have is to give $n = \lfloor n/d_0 \rfloor$ pennies.

   If $i = 1$ and $n < 5$, we can only give pennies, which matches the claim that we give $0 = \lfloor n/d_1 \rfloor$ nickels. If $n \ge 5$, there has to be at least one nickel. Otherwise, we could give better change by replacing 5 pennies with a nickel. By the optimal substructure property, we obtain optimal change for n cents by adding optimal change for $n - 5$ cents to the nickel. By the induction hypothesis, optimal change for $n - 5$ cents includes $\lfloor (n-5)/d_1 \rfloor = \lfloor n/d_1 \rfloor - 1$ nickels. Hence, the optimal change for n cents includes $\lfloor n/d_1 \rfloor$ nickels.

   If $i = 2$ and $n < 10$, we can only give pennies and nickels, which matches the claim that we give $0 = \lfloor n/d_2 \rfloor$ dimes. If $n \ge 10$, there has to be at least one dime. Otherwise, the optimal change would have to include at least 2 nickels, which we could replace with a dime to get better change. By the optimal substructure property, we obtain optimal change for n cents by adding optimal change for $n-10$ cents to the dime. By the induction hypothesis, optimal change for $n-10$ cents includes $\lfloor (n-10)/d_2 \rfloor = \lfloor n/d_2 \rfloor - 1$ dimes. Hence, the optimal change for n cents includes $\lfloor n/d_2 \rfloor$ dimes.

If $i = 3$ and $n < 25$, we can only give pennies, nickels, and dimes, which matches the claim that we give $0 = \lfloor n/d_3 \rfloor$ quarters. If $n \geq 25$, there has to be at least one quarter. Otherwise, there would have to be two dimes and a nickel if $n < 30$ or three dimes if $n \geq 30$. In the former case, we could obtain better change by replacing the two dimes and the nickel with a quarter. In the latter case, we could replace the three dimes with a quarter and a nickel. By the optimal substructure property, we obtain optimal change for n cents by adding optimal change for $n-25$ cents to the quarter. By the induction hypothesis, optimal change for $n-25$ cents includes $\lfloor (n-25)/d_3 \rfloor = \lfloor n/d_3 \rfloor - 1$ quarters. Hence, the optimal change for n cents includes $\lfloor n/d_3 \rfloor$ quarters.

From this discussion, we conclude that the following algorithm gives optimal change. The arguments are:

- n: the amount we want to change
- k: the number of denominations $-$ 1 (since indexing starts at 0)
- d: an array of denominations sorted from the lowest to the highest

The algorithm returns an array C of size k such that $C[i]$, $0 \leq i \leq k$, is the number of coins of denomination $d[i]$ that have to be included in optimal change for n cents.

Greedy-Change(n, k, d)

```
 1 ▷ Initially, we have not given any change yet.
 2 for i = 0..d
 3    do C[i] ← 0
 4 ▷ Now give change.
 5 i ← k
 6 while n > 0
 7    do if n ≥ d[i]
 8          then n ← n − d[i]
 9               C[i] ← C[i] + 1
10          else i ← i − 1
11 return C
```

The running time of the algorithm is $\mathcal{O}(k+n)$. To see this, observe that the for-loop in Lines 2–3 is executed k times; the while-loop in Lines 6–10 is executed until $n = 0$. However, in every iteration either i decreases by one or n decreases by $d_i \geq 1$. Hence, the while-loop is executed $\mathcal{O}(k+n)$ times. Since every iteration takes constant time, this establishes the claimed time bound.

b. To give optimal change using denominations $d_0, d_1, \ldots, d_k$ such that $d_i = c^i$, for some integer $c > 1$, we use the algorithm from Question a. Our proof of the optimal substructure property in Question a does not rely on any particular properties of the coin denominations. Hence, it remains valid. What we have to verify is that, for denominations $d_0, d_1, \ldots, d_k$, the greedy strategy of choosing a largest denomination $d_i \le n$ and then giving optimal change for the amount $n - d_i$ gives optimal change. Again, we prove by induction on i and n that optimal change for n cents using denominations $d_0, d_1, \ldots, d_i$ includes $\lfloor n/d_i \rfloor$ coins of denomination d_i.

   The base case ($i = 0$) is to give $n = \lfloor n/d_0 \rfloor$ coins of denomination $d_0 = 1$. So assume that $i > 0$. If $n < d_i$, then we can only use coins $d_0, d_1, \ldots, d_{i-1}$ to give change for n cents. Hence, our claim holds that we give $0 = \lfloor n/d_i \rfloor$ coins of denomination d_i. If $n \ge d_i$, we observe that optimal change has to include at least one coin of denomination d_i. Otherwise, the induction hypothesis implies that optimal change includes $\lfloor n/d_{i-1} \lfloor \ge d_i/d_{i-1} = c$ coins of denomination d_{i-1}; c of them can be replaced with a single coin of denomination d_i. Since optimal change contains at least one coin of denomination d_i if $n \ge d_i$, we conclude from the optimal substructure property that we can obtain optimal change for n cents by adding this coin of denomination d_i to optimal change for $n - d_i$ cents. By the induction hypothesis, optimal change for $n - d_i$ cents includes $\lfloor (n - d_i)/d_i \rfloor = \lfloor n/d_i \rfloor - 1$ coins of denomination d_i. Hence, optimal change for n cents includes $\lfloor n/d_i \rfloor$ coins of denomination d_i, as claimed.

c. Here's an example: $d_0 = 1$, $d_1 = 7$, and $d_2 = 10$. For 14 cents, our algorithm would produce 5 coins $10 + 1 + 1 + 1 + 1$. Optimal change is $7 + 7$.

d. We have shown in the answer to Question a that the problem has optimal substructure. This proof was independent of the coin denominations. Hence, we can use a dynamic programming algorithm based on the following equation, where $N(i)$ denotes the number of coins in an optimal solution for i cents:

$$N(i) = \begin{cases} 0 & \text{if } i = 0 \\ \min\{1 + N(i - d_j) : 1 \le j \le k \text{ and } d_j \le i\} & \text{if } i > 0 \end{cases}$$

   The algorithm is the following:

Optimal-Change($\boldsymbol{n, k, d}$)

1 $N[0] \leftarrow 0$
2 **for** $i = 1..n$
3 **do** $N[i] \leftarrow +\infty$
4 **for** $j = k, k-1, \ldots, 0$
5 **do if** $d[j] \leq i$
6 **then** $q \leftarrow N[i - d[j]] + 1$
7 **if** $q < N[i]$
8 **then** $N[i] \leftarrow q$
9 $\triangleright$ $G[i]$ is the largest coin denomination used in optimal change for i cents.
10 $G[i] \leftarrow d[j]$
11 **for** $i = 0..k$
12 **do** $C[i] \leftarrow 0$
13 **while** $n > 0$
14 **do** $C[G[n]] \leftarrow C[G[n]] + 1$
15 $n \leftarrow n - G[n]$
16 **return** C

The correctness of this algorithm follows from our above discussion. The running time is $\mathcal{O}(kn)$: To see this, we have to count how often every loop is executed because, inside each loop, we perform only constant work. The loop in Lines 2–10 is executed n time; the loop in Lines 4–10 is executed $k+1$ times per iteration of the outer loop; hence, Lines 1–10 take $\mathcal{O}(kn)$ time. The loop in Lines 11–12 is executed $k+1$ times. The loop in Lines 13–15 is executed at most n times because it is executed until $n = 0$ and n decreases by at least one in every iteration of the loop. Hence, Lines 11–16 take $\mathcal{O}(n+k)$ time, and the total running time is indeed $\mathcal{O}(kn)$.