# CMSC 37000: Homework 1

Caleb Derrickson

January 25, 2024

## Contents

# Problem 1

We are given $2n$ equally spaced points on a line. Half the points are black, and half are white. The goal is to connect every black point to a distinct white point with a wire so as to minimize the total wire length.

Below are two suggestions for greedy algorithms, which may or may not solve the problem correctly. For each suggested algorithm, if it is correct, prove its correctness. If it in incorrect, prove that by providing an input on which the algorithm does not find an optimal solution.

Each of the suggested algorithms performs $n$ iterations. In every iteration, it selects a single pair of points to connect and the deletes these two points from the line. In order to define each algorithm, it is now enough to specify the greedy rule for selecting the pair of points to connect.

1. Greedy Rule 1: Find any pair $(a, b)$ of points, where $a$ is black and $b$ is white, and no other point lies between the two. Connect $a$ to $b$ and delete both points from the line.

2. Greedy Rule 2: Let $a$ be the leftmost black point and let $b$ be the leftmost white point. Connect $a$ to $b$ and delete both points from the line.

---

**Solution:**

Greedy Rule number 1 mentions that $(a, b)$ can be *any* pair of adjacent points. If this is the case, we can choose the pairings such that each successive pair tracks the distance of the previous. Figure 1 showcases this, where the numbers according to each line correspond to the order in which they were chosen and subsequently deleted. For each iteration, the pairing scheme below does obey Greedy Rule number 1, since the prior iterations will be deleted before picking the next two points. It is also a feasible one, since each point is connected to one and only one other point. The length of wire used for this pairing is 16. We can compare this to Figure 2, this pairing is feasible, and uses less wire, 8, than the one in Figure 1. Thus, the first pairing does not provide the optimal solution. Note that the pairing schema in Figure 2 just so happens to be Greedy Rule 2. This does not prove the optimality of Greedy Rule 2, I am just using it as a point of comparison.

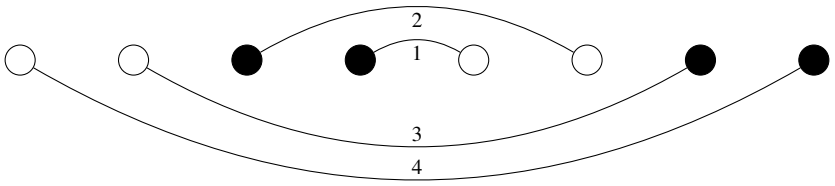



Figure 1: A pairing of adjacent points which does not give an optimal solution.

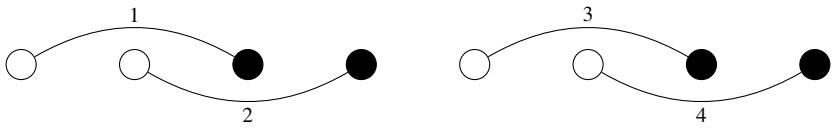



Figure 2: A more optimal pairing than above.

We will now prove the optimality of the pairing set $S$ given by Greedy Rule number 2. Suppose $I_k$ is the $k$-th iteration, which has removed $k < n$ pairings on the interval. Then $|I_k| = 2n - 2k = 2(n-k)$. Define $a_k, b_k$ as the black and white points chosen on the $k$-th iteration, and $d : I \times I \to \mathbb{N}$ be some distance metric. Note that measuring distances will be a proxy for the length of the wire used to connect two points. We will prove via induction on $k$ that $S$ is an optimal pairing set.

- Base Case: $k = 1$

  Suppose that $b_j$ is some other white point in $I_1$. Note that $b_1$ is chosen to be the leftmost white point on the interval. Then $d(a_1, b_j) = d(a_1, b_1) + d(b_1, b_j)$ by the triangle inequality.[1] This implies that $d(a_1, b_j) \geq d(a_1, b_1)$ for any point $b_j \in I_1$. Then Greedy Rule number 2 gives an optimal pairing.

- Induction step

  Suppose that Greedy Rule 2 gives an optimal pairing for iterations up to $k < n$. We will show that $k+1$ iteration also gives an optimal pairing. Note that $d(a_k, b_k) \leq d(a_k, b_j)$ for the iteration $k$ and interval $I_k$. Note that $a_i, b_i$ points have been removed from $I_k$ for all $i < k$. Then the points $a_k$ and $b_k$ have been removed from $I_k$ to produce $I_{k+1}$. Choose $a_{k+1}$ and $b_{k+1}$ to then be the leftmost black and white points of $I_{k+1}$, respectively. Then, for any $b'_{k+1} \in I_{k+1}$, $d(a_{k+1}, b'_{k+1}) = d(a_{k+1}, b_{k+1}) + d(b_{k+1}, b'_{k+1})$. This then implies that $d(a_{k+1}, b_{k+1}) \leq d(a_{k+1}, b'_{k+1})$, so the pairing $(a_{k+1}, b_{k+1})$ provides the optimal pairing on $I_{k+1}$.

Since each step provides an optimal pairing of points on the respective iterated interval $I_i$, then their summation, $\sum_{i=1}^{n} d(a_i, b_i)$ is less than (or equal to) any other pairing schema, therefore Greedy Rule 2 provides an optimal pairing sheme.

[1] The triangle inequality gives an equality in this point since the interval in consideration is only one dimensional, and $b_j$ lies to the right of $b_1$.

# Problem 2

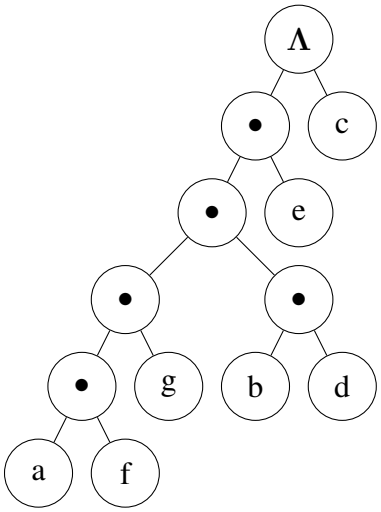
Construct the Huffman code for alphabet $\Sigma$ with 7 characters $\Sigma = \{a, b, c, d, e, f, g\}$ that have frequencies $p(a) = 0.02,\ p(b) = 0.1,\ p(c) = 0.5,\ p(d) = 0.07,\ p(e) = 0.21,\ p(f) = 0.04$, and $p(g) = 0.06$. Specifically, do the following:

- Draw the Huffman tree for $\Sigma$.
- Label the leaves of the tree with characters form $\Sigma$.
- Write the codeword for each character.

---

**Solution:**

The Problem mentions my answer need no explanations, so I will just give the tree and the codewords. Here the "•" marker denotes an empty node.




Code Words

- **a**: 00000
- **b**: 0010
- **c**: 1
- **d**: 0011
- **e**: 01
- **f**: 00001
- **g**: 0001

# Problem 3

A freelance photographer prepares her schedule for the next $n$ days. She may work or rest on each of the days.

- If she works on day $i$, she gets paid $p_i > 0$ dollars.
- However, she is not paid on those days she rests.
- She is not willing to work 3 days in a row.

Subject to these constraints, the photographer wants to maximize her pay.

Design a polynomial-time dynamic programming algorithm that given a sequence of $p_1, ..., p_n$ finds an optimal schedule. Describe your algorithm in detail. Prove its correctness. Specifically, do the following:

1. Define a dynamic-programming table and explain the meaning of its entries.
2. Write the initialization step of your algorithm.
3. Write the recurrence formula for computing entries of the table.
4. Explain the formula.
5. Find the running time of your algorithm.

---

**Solution:**

1. I will define and keep track of two tables: T and W. Table T will keep track of the max amount of money for the iteration and Table W is a binary list that will keep track of the days to work. Note that the way T is defined, it is monotonic. Since the question asks for a schedule that returns the maximum amount of money, we return only W. If you want the maximum amount of money as well, we can return T[end].
2. We initialize the two arrays with locally optimal values, where the first 2 days are set to working. Thus T[0] = p[0] + p[1] and W[0] = 1, W[1] = 1. All other entries in W are set to zero.
3. The recurrence formulas for T and W are different, and should be handled separately. For each iterated window, we need to optimize over that window. If there are not already two working days in a three day window, we accept days until this is true, modifying T and W accordingly. When we have that the past two days are considered working, we then need to compare the incoming day's profit with the previous two day's profits. Then T[k+1] = T[k] + max {0, p[k+1] - p[k], p[k+1] - p[k-1]}. If p[k+1] is less than both p[k] and p[k-1], then we add zero and set W[k+1] = 0. Else we swap the working day according to the maximum.

4. I will prove the feasibility and optimality here. We will prove both of these via induction on the iteration of the window.

   - Feasibility
     - Base Case: k = 1

       Here we are considering the feasibility of the initialization. Since we are not working more than two days in a row in W, then this holds.

     - Induction hypothesis

       Next we assume that for cases up to j, the algorithm has provided a feasible scheduling of days. We will now show the next case, j+1, is also feasible[2]. We will consider only the three days associated with j-1, j, and j+1. If either the two previous days, j-1 and j, are set to zero in W, then we can set W[j+1] = 1 and T[j+1] = T[j] + p[k+1], as discussed above. If both j-1 and j have their indices set to 1 in W, we then consider the value max {0, p[j+1] - p[j], p[j+1] - p[j-1]}. If this value is zero, then we set W[j+1] = 0 and T[k+1] = T[k], since it is not considered worth working on that day. If this value is not zero, then we work the j+1 day over the day which takes the least from p[j+1]. Thus, the algorithm will select the best two days to work within the window centered at j, implying the j+1 case gives a feasible schedule.

   - Optimality

     Note that by the way I wrote the above proof, it also argues that the algorithm produces an optimal schedule. Since T is a monotone array, we would argue that if the algorithm didn't produce an optimal schedule, then there would be another array T' such that T'[end] > T[end]. We would then have two cases to decide: if T' has more elements than T, i.e. we are working more days, or T' has chosen to work more lucrative days. Note that the first case cannot be true, since adding another day to our schedule would make it unfeasible. Thus, for the second case, suppose W and W', the associated working days, are identical up to two contiguous days. Thus, there is some window centered at j such that T' has chosen a more optimal solution than our T, and each window prior has been chosen identically (both are optimal). By our algorithm, we are choosing to work over the most lucrative two days, thus this window should be chosen identically for T and T', giving a contradiction.

5. Since we are only looping over p once, and for each iteration of the for loop we are only doing O(1) operations, the overall running time of the algorithm is O(n). The memory complexity of our algorithm is slightly higher: since we are allocating memory for two arrays of length n, the memory complexity is O(2n), which in all intensive purposes, is O(n).

[2] Note that the j+1 case window has its center on the j-th day.