CS180 Homework 3 Solution

Question 1

We have a connected graph $G = (V, E)$, and a specific vertex $u \in V$. Suppose we compute a depth-first search tree rooted at u, and obtain a tree T that includes all nodes of G. Suppose we then compute a breadth-first search tree rooted at u, and obtain the same tree T. Prove that $G = T$. (In other words, if T is both a depth-first search tree and a breadth-first search tree rooted at u, then G cannot contain any edges that do not belong to T.)

Solution. Let T be the common spanning tree produced by both a DFS and a BFS of G, each rooted at u. For a vertex v, let $\ell(v)$ denote its depth in T (that is, its distance from u within T).
Fact 1 (BFS level property). In any BFS tree rooted at u, for every edge $(x, y) \in E$ (whether or not it is a tree edge), one has

$$|\ell(x) - \ell(y)| \leq 1.$$

Reason. BFS assigns to each vertex its shortest-path distance from u in G. Along any original edge (x, y), these distances can differ by at most 1.
Fact 2 (DFS non-tree edges in undirected graphs). In a DFS tree of an undirected graph, every non-tree edge joins a vertex to an ancestor in the DFS tree. In particular, if $(x, y) \in E \setminus E(T)$, then one of x, y is an ancestor of the other in T. Moreover, it cannot be a parent–child pair, because that edge is already in T. Hence, if x is an ancestor of y and $(x, y) \notin E(T)$, then

$$\ell(y) \geq \ell(x) + 2.$$

Reason. When an undirected edge (x, y) is first examined during the recursive exploration of x:

- If y is undiscovered at that moment, the algorithm recurses to y, so (x, y) becomes a tree edge.
- If y has been discovered but not finished, then y lies on the current recursion stack; hence y is an ancestor of x in the DFS tree, and (x, y) is a back edge to an ancestor.
- If y is already finished, then when y was explored earlier the same edge (x, y) was considered from y. At that time, x was either undiscovered (which would have made (x, y) a tree edge) or on the recursion stack above y (so x was an ancestor of y).

Thus every non-tree edge in an undirected DFS connects a vertex to an non-parent ancestor.
Now assume for contradiction that G has an edge $(a, b) \notin E(T)$. By Fact 2, without loss of generality, a is an ancestor of b in T and $\ell(b) \geq \ell(a) + 2$. But then, by the BFS level property (Fact 1) applied to the same edge (a, b), we must have $|\ell(a) - \ell(b)| \leq 1$, which contradicts $\ell(b) \geq \ell(a) + 2$.
Therefore no such edge can exist, and so $E = E(T)$. Since T is spanning, we conclude $G = T$. $\square$

Question 2

Let us consider a long, quiet country road with houses scattered very sparsely along it. (We can picture the road as a long line segment, with an eastern endpoint and a western endpoint.) Further, let us suppose that despite the bucolic setting, the residents of all these houses are avid cell phone users. You want to place cell phone base stations at certain points along the road, so that every house is at most 4 miles from one of the base stations.

Give an efficient algorithm that achieves this goal, using as few base stations as possible. That is, given a list of n houses with distances $d_1, d_2, \ldots, d_n$ from the western endpoint of the road, return a list of cell towers with the minimal length possible such that every house has cell service.

Example. If the houses' distances from the western endpoint are $[7, 1, 4, 10, 15]$, an optimal list of cell towers could be $[4, 13]$. The first tower covers $[0, 8]$ which includes the houses at distances $[1, 4, 7]$, and the second covers $[9, 17]$ which includes the houses at distances $[10, 15]$. There is no way to cover all houses with a single tower, so this is optimal (but not the only optimal arrangement).

Solution

Algorithm Given the house positions $d_1, \ldots, d_n$ along the line, proceed as follows.

1. Sort the positions so that $x_1 \leq x_2 \leq \cdots \leq x_n$.
2. Initialize an empty list T of tower positions and set $i \leftarrow 1$.
3. While $i \leq n$:
   3.1. Let $p \leftarrow x_i + 4$. Place a tower at position p and append p to T.
   3.2. Increase i to be the first index with $x_i > p + 4$ (skip all houses that are within 4 miles of p).
4. Output T.

Proof of correctness Each tower placed at position p covers the interval $[p - 4,\ p + 4]$. In step 3.1, we place the tower at $x_i + 4$, which is the rightmost position that still covers house x_i. This choice maximizes coverage to the east while still serving x_i, so it covers every house in $[x_i,\ x_i + 8]$. Step 3.2 then advances i past all covered houses. Hence the algorithm always returns a set of towers that covers all houses.
It remains to show optimality, i.e., that the algorithm uses the minimum possible number of towers. We will show this by an exchange argument. Let the list T is produced by our algorithm, and the list T' is optimal. Let $p_1 \leq p_2 \leq \cdots \leq p_m$ be positions in the list T, and $p'_1 \leq p'_2 \leq \cdots \leq p'_{m'}$ be positions in the list T'. Suppose j is smallest index such that $p_j \neq p'_j$. Let x_i be the first house uncovered by the $p_1 - p_{j-1}$. Then, in list T', we can shift the p'_j to $x_i + 4$ (which is p_j) and still covers all houses. By continuing this operation, we transform T' to be the first m' elements of T. Since the first m' elements of T have already covered all houses, our algorithm would not build additional base stations. Thus, we have $m = m'$ and T is also optimal.

Time complexity Sorting the positions takes $O(n \log n)$ time. The single pass that places towers and advances i runs in $O(n)$ time. The total running time is $O(n \log n)$ and the additional space is $O(1)$ beyond the output list.

Question 3

The wildly popular Spanish-language search engine El Goog needs to do a serious amount of computation every time it recompiles its index. Fortunately, the company has at its disposal a single large supercomputer, together with an essentially unlimited supply of high-end PCs.
They have broken the overall computation into n distinct jobs, labeled $J_1, J_2, \ldots, J_n$, which can be performed completely independently of one another. Each job consists of two stages: first it needs to be preprocessed on the supercomputer, and then it needs to be finished on one of the PCs. Let us say that job J_i needs p_i seconds of time on the supercomputer, followed by f_i seconds of time on a PC.
Since there are at least n PCs available on the premises, the finishing of the jobs can be performed fully in parallel: all the jobs can be processed at the same time. However, the supercomputer can only work on a single job at a time, so the system managers need to work out an order in which to feed the jobs to the supercomputer. As soon as the first job in order is done on the supercomputer, it can be handed off to a

PC for finishing; at that point in time a second job can be fed to the supercomputer; when the second job is done on the supercomputer, it can proceed to a PC regardless of whether or not the first job is done (since the PCs work in parallel); and so on.
Let us say that a *schedule* is an ordering of the jobs for the supercomputer, and the *completion time* of the schedule is the earliest time at which all jobs will have finished processing on the PCs. Give a polynomial-time algorithm that finds a schedule with as small a completion time as possible.

Solution

Algorithm Order the jobs by nonincreasing finishing times f_i (that is, sort so that $f_1 \geq f_2 \geq \cdots \geq f_n$), breaking ties arbitrarily. Process the jobs on the supercomputer in this order. As soon as a job finishes its supercomputer stage, immediately start its PC stage on some available PC.

Proof of correctness Fix any schedule σ, and write P_k for the total supercomputer time of the first k jobs in σ: $P_k = \sum_{\ell=1}^{k} p_{\sigma(\ell)}$. If job i appears in position k of σ, its PC stage can start only after its supercomputer stage completes, at time P_k, and it then needs an additional f_i seconds on a PC. Because PCs run in parallel without contention, the time at which all jobs are finished (the completion time or makespan) is

$$T(\sigma) = \max_{1 \leq k \leq n} \big(P_k + f_{\sigma(k)}\big).$$

We will show that for any schedule that has an adjacent pair i then j with $f_i < f_j$, swapping this pair does not increase the value of the maximum above.
Consider two adjacent jobs i then j appearing at σ. Let S be the supercomputer time accumulated before i starts . Then the two jobs contribute the following two candidates to the maximum:

$$\text{Before swap:} \quad S + p_i + f_i, \qquad S + p_i + p_j + f_j.$$

If we swap i and j, these two candidates become

$$\text{After swap:} \quad S + p_j + f_j, \qquad S + p_j + p_i + f_i.$$

The rest of the terms in the maximum are identical in the two schedules. Therefore it suffices to compare the pairwise maxima over these two local terms. When $f_i < f_j$, we have

$$\max\{\, S{+}p_i{+}f_i,\ S{+}p_i{+}p_j{+}f_j \,\} \ \geq\ \max\{\, S{+}p_j{+}f_j,\ S{+}p_j{+}p_i{+}f_i \,\},$$

because $S{+}p_i{+}p_j{+}f_j \geq S{+}p_j{+}f_j$ and $S{+}p_i{+}p_j{+}f_j \geq S{+}p_j{+}p_i{+}f_i$. Thus swapping a violating adjacent pair (i, j) with $f_i < f_j$ does not increase the makespan.
By repeatedly applying such swaps, any schedule can be transformed into one in which f values are in non-increasing order. Since each swap does not increase the completion time, a schedule sorted by nonincreasing f_i is optimal.

Time complexity Sorting the n jobs by f_i takes $O(n \log n)$ time. Once sorted, the supercomputer runs them in that order, and PCs start their tasks immediately when available. Computing the makespan if desired can be done in a single pass by maintaining the running sum P_k, which is $O(n)$. Therefore the algorithm runs in $O(n \log n)$ time and $O(1)$ extra space beyond the input and output order.

Question 4

One of the basic motivations behind the Minimum Spanning Tree Problem is the goal of designing a spanning network for a set of nodes with minimum total cost. Here we explore another type of objective: designing a spanning network for which the most expensive edge is as cheap as possible.
Specifically, let $G = (V, E)$ be a connected graph with n vertices, m edges, and positive edge costs that you may assume are all distinct. Let $T = (V, E')$ be a spanning tree of G; we define the *bottleneck edge* of T to be the edge of T with the greatest cost. A spanning tree T of G is a *minimum-bottleneck spanning tree* if there is no spanning tree T' of G with a cheaper bottleneck edge.

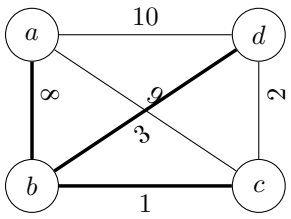
(a) Is every minimum-bottleneck tree of G a minimum spanning tree of G? Prove or give a counterexample.

(b) Is every minimum spanning tree of G a minimum-bottleneck tree of G? Prove or give a counterexample.

Solution (a) Not every minimum-bottleneck tree is a minimum spanning tree

Counterexample. Let the vertex set be $\{a, b, c, d\}$. Consider the edge weights (all distinct):

$$w(ab) = 8, \quad w(ac) = 9, \quad w(ad) = 10, \quad w(bc) = 1, \quad w(cd) = 2, \quad w(bd) = 3.$$



The graph is connected. Any spanning tree must contain at least one edge incident to a. Among edges from a, the cheapest is ab with cost 8, and all other edges from a have cost strictly greater than 8. Therefore, in every spanning tree the bottleneck cost is at least 8.
There are many spanning trees whose bottleneck cost is exactly 8. Two examples are

$$T_1 = \{ab, bc, cd\} \quad \text{with total cost } 8 + 1 + 2 = 11 \text{ and bottleneck } 8,$$
$$T_2 = \{ab, bc, bd\} \quad \text{with total cost } 8 + 1 + 3 = 12 \text{ and bottleneck } 8.$$

Since no spanning tree can have bottleneck less than 8, both T_1 and T_2 are minimum-bottleneck spanning trees. However, T_2 has strictly larger total weight than T_1, so T_2 is not a minimum spanning tree. Hence, not every minimum-bottleneck tree is an MST.

(b) Every minimum spanning tree is a minimum-bottleneck tree

Proof. Let T be an MST and let $e^\star$ be the unique heaviest edge of T (edge weights are distinct). Remove $e^\star$ from T; this splits T into two components A and B. By the cut property, among all edges crossing the cut (A, B), $e^\star$ has minimum possible weight among edges that can appear in any MST. In particular, any spanning tree S must contain at least one edge f that crosses (A, B). Since $e^\star$ is the lightest edge across this cut that can connect A and B without violating minimality of T, the weight of f satisfies $w(f) \geq w(e^\star)$. Thus the bottleneck edge of any spanning tree S has weight at least $w(e^\star)$. Therefore the maximum edge weight in T is less than or equal to the maximum edge weight in any other spanning tree, which proves that T is a minimum-bottleneck spanning tree. □

Question 5

A small business—for example, a photocopying service with a single large machine—faces the following scheduling problem. Each morning they get a set of jobs from customers. They want to do the jobs on their single machine in an order that keeps their customers happiest. Customer i's job will take t_i time to complete. Given a schedule (i.e., an ordering of the jobs), let C_i denote the finishing time of job i. For example, if job j is the first to be done, we would have $C_j = t_j$; and if job j is done right after job i, we would have $C_j = C_i + t_j$. Each customer i also has a given weight w_i that represents his or her importance to the business. The happiness of customer i is expected to be dependent on the finishing time of i's job. So the company decides that they want to order the jobs to minimize the weighted sum of the completion times:

$$\sum_{i=1}^{n} w_i C_i.$$

Design an efficient algorithm to solve this problem. That is, you are given a set of n jobs with a processing time t_i and a weight w_i for each job. You want to order the jobs so as to minimize the weighted sum of the completion times:

$$\sum_{i=1}^{n} w_i C_i.$$

Example: Suppose there are two jobs: the first takes time $t_1 = 1$ and has weight $w_1 = 10$, while the second job takes time $t_2 = 3$ and has weight $w_2 = 2$. Then doing job 1 first would yield a weighted completion time of $10 \cdot 1 + 2 \cdot 4 = 18$, while doing the second job first would yield the larger weighted completion time of $10 \cdot 4 + 2 \cdot 3 = 46$.

Solution

Algorithm Sort the jobs by nonincreasing *weight–per–time* ratio w_i/t_i (equivalently, by nondecreasing t_i/w_i). Then process the jobs in that sorted order on the single machine. Ties can be broken arbitrarily.

Proof of correctness We use an exchange argument. Consider any adjacent pair of jobs i and j in a schedule, where job i is done before job j. Let T be the total processing time of all jobs scheduled before i. The contribution of i and j to the weighted sum $\sum_k w_k C_k$ is

$$w_i(T + t_i) + w_j(T + t_i + t_j) \;=\; (w_i + w_j)T \;+\; w_i t_i \;+\; w_j(t_i + t_j).$$

If we swap i and j, the contribution becomes

$$w_j(T + t_j) + w_i(T + t_j + t_i) \;=\; (w_i + w_j)T \;+\; w_j t_j \;+\; w_i(t_j + t_i).$$

The difference (old minus new) is

$$\big[w_i t_i + w_j t_i + w_j t_j\big] - \big[w_j t_j + w_i t_j + w_i t_i\big] \;=\; t_i w_j - t_j w_i.$$

Thus the schedule with i before j is no worse than the schedule with j before i if and only if $t_i w_j - t_j w_i \le 0$, which is equivalent to

$$\frac{w_i}{t_i} \;\ge\; \frac{w_j}{t_j}.$$

Therefore, in any optimal schedule, every adjacent pair must be ordered so that w/t is nonincreasing from left to right. This means that sorting by nonincreasing w_i/t_i yields an optimal schedule.

Complexity Computing the ratios and sorting the n jobs takes $O(n \log n)$ time. The schedule is produced directly from the sorted order. The space usage is $O(1)$ beyond the array of jobs (or $O(n)$ if we output a new list).

Question 6

Problem. Given an $n \times m$ grid consisting of W (Water) and L (Land) cells, the task is to count the number of islands. An island is a group of adjacent L cells connected horizontally, vertically, or diagonally, and is surrounded by water or the grid boundary. The goal is to determine how many distinct islands exist in the grid.

Example. For the following grid:

$$\begin{array}{ccccc} L & L & W & W & W \\ W & L & W & W & L \\ L & W & W & L & L \\ W & W & W & W & W \\ L & W & L & L & W \end{array}$$

There are 4 islands.

Solution

Algorithm

We model the grid as a graph where each land cell is a vertex, and there is an edge between two land cells if they are adjacent horizontally, vertically, or diagonally (8-neighborhood). We scan the grid. When we see an L cell that has not been visited, we start a search from it (either Depth-First Search or Breadth-First Search) over the 8 directions, marking every reachable L cell as visited. Each time we start such a search, we increase a counter by one. At the end of the scan the counter is the number of islands.

Pseudocode (iterative BFS).

```
count_islands(G):
  n, m = number of rows, number of columns of G
  visited = array[n][m] initialized to false
  islands = 0
  directions = [(-1,-1), (-1,0), (-1,1),
                ( 0,-1),         ( 0,1),
                ( 1,-1), ( 1,0), ( 1,1)]
  for i in 0..n-1:
    for j in 0..m-1:
      if G[i][j] == 'L' and not visited[i][j]:
        islands = islands + 1
        queue = [(i, j)]
        visited[i][j] = true
        while queue not empty:
          (x, y) = queue.pop()
          for (dx, dy) in directions:
            u = x + dx; v = y + dy
            if 0 <= u < n and 0 <= v < m:
              if G[u][v] == 'L' and not visited[u][v]:
                visited[u][v] = true
                queue.push((u, v))
  return islands
```

Proof of correctness We prove that the algorithm returns the number of islands.
First, consider any time the algorithm increments the counter and starts a BFS from an unvisited land cell c. By construction, the BFS explores exactly the set S of land cells that are connected to c through a sequence of horizontal, vertical, or diagonal moves. Every neighbor that is an L cell is enqueued and marked visited, and this process repeats, so every cell in S is marked visited. A water cell is never enqueued, and a land cell that is not connected to c is not reachable by such moves, so it is not marked visited during this BFS. Therefore this BFS marks exactly one connected component of land cells.
Second, after this BFS finishes, no later BFS will start inside S, because all cells of S are visited. Thus each connected component of land cells causes exactly one increment of the counter.
Finally, since the outer scan visits every grid cell, if there is a connected component S' of land cells, the first time the scan encounters a cell of S' that is still unvisited, the algorithm starts a BFS and increments the counter. Hence every island is counted once, and only once. Therefore the returned counter equals the number of islands.

Time complexity Each cell is enqueued and dequeued at most once. In processing a cell, we check at most 8 neighbors. The running time is $O(nm)$, and the auxiliary space is $O(nm)$ for the visited array and the queue in the worst case.