CS 621 - Assignment 2

Ali Hassani

October 23, 2022

1

The California Gold Rush (1848-1855) was one of the largest mass migrations in human history. Forty-niners came from across the globe – Oregon, China, Australia, Latin America, Hawaii, Europe, etc. – seeking to strike it rich mining for gold in the Sierra Nevada mountains.
For a combination of reasons (illiteracy, language difference, secrecy, loss), there is no written record of the years each mine has operated. As such, a group of local historians is trying to infer when each mine operated from alternate records. As a starting point, these historians are focusing on a specific set of n mines, and by cross-referencing various sources, these historians collected an initial set of records. Each record takes one of two possible forms, namely:

1. That two mines from this set of n operated simultaneously.
2. That one mine from the set of n closed before another mine (from the set of n) opened.

Assume that once a mine stopped operating, it remained closed forever.
Before considering additional mines beyond the n, they hired you to check whether their records have any internal contradictions and to determine a feasible ordering of their starting and closing dates (which are unknown). Provide an efficient algorithm that determines such an ordering of their starting and closing dates, or to say that the information is contradictory. Provide your algorithm's runtime.
Hint: Map the records to a directed graph, and represent each mine as two connected nodes. Use topological sort.

Answer:

We construct a directed graph with $2n$ nodes, where each mine m_i has a start node s_i and end node f_i, and we connect s_i to f_i for every i. We loop through the records, and for every record that states m_i closed before m_j opened:

- Connect f_i to s_j.

For every record that states m_i and m_j operated simultaneously, we simply create a new node $O_{i,j}$, and:

- Connect s_i to $O_{i,j}$,
- Connect s_j to $O_{i,j}$,
- Connect $O_{i,j}$ to f_i,
- Connect $O_{i,j}$ to f_j.

This does not create a cycle naturally, but would result in a cycle if there's a contradiction (say if we have both types of observations for the same two mines).

Then we simply perform a topological sort on the resulting graph. If it yields a topological ordering, then we can conclude there's no contradiction.

Time complexity: Running the topological sort is of $\mathcal{O}(|V| + |E|)$ for a given graph. Our graph has at least $2n$ nodes and at most $2n + p$ (p mines reported to have operated simultaneously.) It also has n edges between the start and finish nodes for each mine, and $q+4p$ edges added by observations (p simultaneous, q otherwise). Worst case scenario is having $n^2 - n$ observations (for every pair of two different mines), all of which are simultaneous, which would yield $2n + n^2 - n = n^2 + n$ nodes, and $n + 4n^2 - 4n = 4n^2 - 3n$ edges. Therefore the time complexity for this algorithm will be of $\mathcal{O}(n^2+n+4n^2-3n)$ or simply $\mathcal{O}(n^2)$.

2 Chapter 4, Exercise 5

Let's 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's 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 within four miles of one of the base stations.

Give an efficient algorithm that achieves this goal, using as few base stations as possible.

Answer:

Greedy choice: Place the first station 4 miles west of the east-most house, eliminate the houses within 4 miles of it, and repeat until no houses are left.

Correctness: Let $G = \{g_1, g_2, ..., g_\ell\}$ be the greedy solution. For a given solution set $A = \{a_1, a_2, ..., a_m\}$ that is not equal to G, we want to show that $|G| \leq |A|$ or $\ell \leq m$. Assuming both are sorted east-to-west, and starting with a_1 and g_1: if $a_1 > g_1$, this is a contradiction, because the first house will not be covered, since g_1 is the last point one can place a station while still covering the first house. Therefore $a_1 \leq g_1$. Now assume we've continued comparing elements in A and G up to some k, and we found that $a_k \leq g_k$ still holds. Because of our assumption that the two sets are sorted, and that both are correct (cover all houses), there can be no house left behind. Therefore, $g_1, g_2, ..., g_k$ should cover at least the same number of houses that $a_1, a_2, ..., a_k$ cover. Now if we look at a_{k+1}, because it should not leave any house between a_k and itself uncovered, and $a_k \leq g_k$, it would also not leave any house between g_k and itself uncovered. And the greedy choice, g_{k+1} would be the farthest point that we can go and not leave any house between g_k and itself uncovered by definition. Therefore $a_{k+1} \leq g_{k+1}$, and this is proof by induction that the greedy solutions stay ahead. By continuing this, eventually we reach ℓ where G has covered all houses, but A may still require more bases to cover all, which makes A a worse solution.

Algorithm: Assuming the houses are sorted by their distance from the east endpoint (west would work similarly):

- First station's location is exactly the first house's coordinate plus 4 miles west ($\mathcal{O}(1)$).
- Continue going through the houses until we reach a house x whose distance from the first station is more than 4 miles ($\mathcal{O}(n)$).
- Place the next station at x's coordinate 4 miles west ($\mathcal{O}(1)$).
- Repeat steps 1 through 3 until we run out of houses.

```
Inputs: int n, float* coordinates, int r = 4

Stack<int> stations;
stations->push(coordinates[0] + 4);
for (int i=1; i < n; ++i)
    if (abs(stations->peek() - coordinates[i]) > r)
        stations->push(coordinates[i] + 4);

return stations;
```

Time complexity:
Time complexity: $\mathcal{O}(n)$.
Time complexity with sort: $\mathcal{O}(n + n \log n) = \mathcal{O}(n \log n)$.

3 Chapter 4, Exercise 7

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've broken the overall computation into n distinct jobs, labeled J_1, J_2, . . . , 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's 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's 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. This is an important quantity to minimize, since it determines how rapidly El Goog can generate a new index.

Give a polynomial-time algorithm that finds a schedule with as small a completion time as possible.

Answer:

Greedy choice: Sort all jobs J_1, J_2, . . . , J_n by time on PC, or f_i, in descending order, so that jobs that take longer on PC get run on the supercomputer sooner.

Correctness: For a given solution set A that is not equal to the greedy solution G, and cannot be produced by the greedy algorithm: Firstly we know both are permutations of n jobs, so both will be of the same length (there cannot be any items in G that are not in A and vice versa). Therefore, there exists at least one instance of two back-to-back jobs J_b and J_a where $f_a > f_b$. In other words:

- J_b finishes at time $d_1 + p_b + f_b$, where d_1 is the sum of p_is before J_b.
- J_a finishes at time $d_1 + p_b + p_a + f_a$.

Now if we were to swap J_a and J_b's position in schedule A, leading to a new schedule A':

- J_a finishes at time $d_1 + p_a + f_a$, which is definitely less than $d_1 + p_b + d_2 + p_a + f_a$, so J_a now finishes faster.
- J_b finishes at time $d_1 + p_a + p_b + f_b$, which is slower than $d_1 + p_b + f_b$.

Now this does not affect the jobs that came before the two (they've already finished running on the supercomputer), or the jobs that come after them (they'll still have to wait for both jobs to finish on the supercomputer). This swap does however affect the two jobs themselves. For simplicity, we'll exclude d_1 from the notations, because it is a constant in both times.
In the original A, the time the two jobs J_a and J_b took was $p_b + p_a + f_a$ and $p_b + f_b$. Of the two, we know J_a finishes later because:

$$\begin{aligned} f_a > +f_b \quad &\implies \\ p_b + f_a > p_b + f_b \quad &\implies \\ p_b + p_a + f_a > p_b + f_a > p_b + f_b. & \end{aligned}$$

Therefore, the maximum of the two times is the time J_a takes, or $p_b + p_a + f_a$. We take the maximum, because the maximum of these two contributes to the global maximum of all jobs, therefore to the efficiency of the solution.
In the new A', the time the two jobs J_a and J_b took is $p_a + f_a$ and $p_a + p_b + f_b$. We already know that:

$$p_a + f_a \leq p_b + p_a + f_a.$$

And we can also see:

$$\begin{aligned} f_b \leq f_a \quad &\implies \\ p_a + p_b + f_b \leq p_a + p_b + f_a, & \end{aligned}$$

which means the maximum of the times in A' is less than or equal to the maximum in A, therefore this swap either does not affect A or improves it. Now if we consider A' to be our initial set, and find another pair of back-to-back jobs with a similar property and swap their order, we would find again that it either does not affect the solution's overall score or improves it. If we keep repeating this, it will act as a bubble sort and we will eventually end up with a solution that the greedy method could have produced (or exactly G if no to jobs have exactly the same f), and it will be in polynomial steps (n^2).

Algorithm:

- Sort all jobs by time on PC, or f_i in descending order.

Time complexity:
Time complexity: $\mathcal{O}(n \log n)$.

4 Chapter 4, Exercise 15

The manager of a large student union on campus comes to you with the following problem. She's in charge of a group of n students, each of whom is scheduled to work one shift during the week. There are different jobs associated with these shifts (tending the main desk, helping with package delivery, rebooting cranky information kiosks, etc.), but we can view each shift as a single contiguous interval of time. There can be multiple shifts going on at once.

She's trying to choose a subset of these n students to form a *supervising committee* that she can meet with once a week. She considers such a committee to be complete if, for every student not on the committee, that student's shift overlaps (at least partially) the shift of some student who is on the committee. In this way, each student's performance can be observed by at least one person who's serving on the committee.

Give an efficient algorithm that takes the schedule of n shifts and produces a complete supervising committee containing as few students as possible.

Example. Suppose $n = 3$, and the shifts are
Monday 4 P.M.–Monday 8 P.M.,
Monday 6 P.M.–Monday 10 P.M.,
Monday 9 P.M.–Monday 11 P.M..

Then the smallest complete supervising committee would consist of just the second student, since the second shift overlaps both the first and the third.

Answer:

Greedy choice: Sort all shifts by end time in ascending order. Until we have a complete supervising committee, for every remaining shift, select the latest ending shift it intersects with, add it to the supervising committee, and remove the shifts preceding it that it overlaps with.

Algorithm:

1. Sort all shifts by end time, and call the list U.
2. Initialize a committee list C.
3. While C is not complete:
4. (a) Select the first element in U, u_i.
   (b) Find the last $j \geq i$ where u_j overlaps with u_i.
   (c) Add student corresponding to u_j to the committee (C).
   (d) Remove u_i and u_j and elements in between from U (all of which should by definition overlap with u_j).

Note: The condition for the outer loop should be C's completeness, and not U being empty, despite the former being more expensive. The reason for that is that the remaining shifts in U could be covered by the last addition to C. On the other hand, if we remove every shift that any new addition to C intersects with, we could be removing a potential future committee member that would cover other shifts more optimally.
Note: C's completeness can be measured by simply adding a step to the loop that checks if u_j intersects with every remaining element in U.

Time complexity:

- The initial sort is $\mathcal{O}(n \log n)$.
- C can be initialized in constant time.
- The loop runs at worst exactly n times (no overlapping shifts). The operations inside it run at constant time. Checking completeness is $\mathcal{O}(n)$, therefore making the loop $\mathcal{O}(n^2)$.

Therefore the total time complexity is $\mathcal{O}(n^2)$.

Correctness: For a given solution set A that is not equal to the greedy solution G, and cannot be produced by the greedy algorithm: Assuming both are sorted in order of finish time, let a_i and g_i be the first element which A and G disagree on. One possibility is $a_i > g_i$, which would mean there is at least one shift that is not covered by a_1, a_2, ..., a_{i-1}, that a_i does not cover either. This is because g_i is the last shift that would cover the shifts not covered by earlier elements. Therefore $a_i < g_i$. There's two possible scenarios, either g_i intersects with more shifts, or it intersects with the same number of shifts. Therefore swapping the two would either improve the solution, by selecting a committee member with more overlaps, or it would not affect it negatively. By swapping the a_i with g_i, and repeating this process, we convert A to G while either improving it, or not affecting it at all.