CS 621 - Assignment 1

Ali Hassani

October 11, 2022

1 Chapter 1, Exercise 6

Peripatetic Shipping Lines, Inc., is a shipping company that owns n ships and provides service to n ports. Each of its ships has a schedule that says, for each day of the month, which of the ports it's currently visiting, or whether it's out at sea. (You can assume the "month" here has m days, for some $m > n$.) Each ship visits each port for exactly one day during the month. For safety reasons, PSL Inc. has the following strict requirement:

(†) No two ships can be in the same port on the same day.

The company wants to perform maintenance on all the ships this month, via the following scheme. They want to truncate each ship's schedule: for each ship S_i, there will be some day when it arrives in its scheduled port and simply remains there for the rest of the month (for maintenance). This means that S_i will not visit the remaining ports on its schedule (if any) that month, but this is okay. So the truncation of S_i's schedule will simply consist of its original schedule up to a certain specified day on which it is in a port P; the remainder of the truncated schedule simply has it remain in port P.

Now the company's question to you is the following: Given the schedule for each ship, find a truncation of each so that condition (†) continues to hold: no two ships are ever in the same port on the same day. Show that such a set of truncations can always be found, and give an algorithm to find them.

Answer:

This problem can be solved with Gale-Shapley, as it can be expressed as a stable matching problem. Ships will be matched to ports in which they will remain for the rest of the month. What is needed is to define preferences between ports and ships so that an unstable pair becomes equivalent to the problem condition (no two ships at the same port on the same day).
Assume we define preferences in the following manner:

Ships prefer to remain in the first port they arrive at, and ports prefer to be provided maximum service, which means ports prefer to keep the last ship that visits them.

No two ships can be at the same port on the same day, and we know this condition is already met in the initial schedule.
Now we prove that assigning ships to ports with a stable matching algorithm, such as Gale-Shapley, will not break that condition. In other words, an unstable match based on this definition of the problem, will be equivalent to breaking that condition.
Assume the very first assignment is ship a being assigned to some port P, and this breaks the condition. This means that some ship b is going to visit port P after a has already visited it and stopped there for maintenance. Because b visits P after a, based on the definition above, P prefers b over a. This is contradictory, since if P preferred b, it would have proposed to b and have been matched with b in the first place.
Now assume that ℓ assignments have been made without breaking the condition, and assignment $\ell + 1$ is made. Assume it assigns ship c to port Q, and this breaks the condition. This means that some ship d is going to visit port Q after c has already visited it and stopped there for maintenance. If that is true, because d visits Q after c, Q prefers d over c. At the same time, if d is already assigned to some port R, it must come after Q on its schedule, which means d prefers Q to its current match R as well, and this is an unstable match. If d is not assigned to any port, then Q must have proposed to it before c, as it prefers d and d is unmatched, therefore this is a contradiction.
As a result, we have proven that by the definition above, an unstable match would be equivalent to breaking the condition, and therefore Gale-Sahpley can always find a solution.

2 Chapter 2, Exercise 8

You're doing some stress-testing on various models of glass jars to determine the height from which they can be dropped and still not break. The setup for this experiment, on a particular type of jar, is as follows. You have a ladder with n rungs, and you want to find the highest rung from which you can drop a copy of the jar and not have it break. We call this the highest safe rung. It might be natural to try binary search: drop a jar from the middle rung, see if it breaks, and then recursively try from rung $n/4$ or $3n/4$ depending on the outcome. But this has the drawback that you could break a lot of jars in finding the answer. If your primary goal were to conserve jars, on the other hand, you could try the following strategy. Start by dropping a jar from the first rung, then the second rung, and so forth, climbing one higher each time until the jar breaks. In this way, you only need a single jar—at the moment it breaks, you have the correct answer—but you may have to drop it n times (rather than $\log n$ as in the binary search solution). So here is the trade-off: it seems you can perform fewer drops if you're willing to break more jars. To understand better how this trade- off works at a quantitative level, let's consider how to run this experiment given a fixed "budget" of $k \geq 1$ jars. In other words, you have to determine the correct answer—the highest safe rung—and can use at most k jars in doing so.

2.a

Suppose you are given a budget of $k = 2$ jars. Describe a strategy for finding the highest safe rung that requires you to drop a jar at most $f(n)$ times, for some function $f(n)$ that grows slower than linearly. (In other words, it should be the case that $\lim_{n\to\infty} f(n)/n = 0$.)

2.b

Now suppose you have a budget of $k > 2$ jars, for some given k. Describe a strategy for finding the highest safe rung using at most k jars. If $f_k(n)$ denotes the number of times you need to drop a jar according to your strategy, then the functions f_1, f_2, f_3, . . . should have the property that each grows asymptotically slower than the previous one: $\lim_{n\to\infty} f_k(n)/f_{k-1}(n) = 0$ for each k.

Answer:

For $k = 2$:

Let's assume x is an integer and $x = f(n)$. The best strategy that would ensure at most $f(n)$ or x drops is as follows:

1. Drop a jar from rung x. If it breaks, start at rung 1 and work up to $x - 1$. With the initial drop that's exactly x drops.

2. If it did not break, go up $x - 1$ rungs and drop. If it breaks, start at rung $x + 1$ and work up to $2x - 1$. With the two drops, that's exactly x drops.

3. If it did not break, go up $x - 2$ rungs and drop. If it breaks, start at rung $2x$ and work up to $3x - 3$. With the three drops, that's exactly x drops.

4. Continue this, while moving up one rung less each time.

When and if we get to a point where we move up only 1 rung, we should hope we've reached n (or exceeded it and stopped). This means that the sum of these steps should be greater than or equal to n:

$$x + (x - 1) + (x - 2) + ... + 2 + 1 = \frac{x(x + 1)}{2} \geq n.$$

This can be simplified into the following quadratic inequality:

$$x^2 + x - 2n \geq 0.$$

We can solve it by letting it be equal to 0 and solving it as an equation:

$$x = \frac{-1 \pm \sqrt{1 + 8n}}{2},$$

and because it's a positive integer, we can conclude that:

$$f(n) = \lceil \frac{\sqrt{1 + 8n} - 1}{2} \rceil,$$

and

$$\lim_{n \to \infty} \frac{f(n)}{n} = \lim_{n \to \infty} \frac{\sqrt{1 + 8n} - 1}{2n} = 0,$$

which is bound by $\mathcal{O}(\sqrt{n})$.

For $k > 2$:

Let's start with $k = 3$. Suppose our strategy is to drop the first jar at multiples of $\lfloor n^{2/3} \rfloor$. Whenever the first jar breaks, we've reduced the problem to size $\lfloor n^{2/3} \rfloor$ with $k = 2$, and we know from the previous question that the maximum number of drops with our strategy is bound by $\mathcal{O}(\sqrt{\lfloor n^{2/3} \rfloor})$. Therefore, we drop the first jar $\lceil \frac{n}{\lfloor n^{2/3} \rfloor} \rceil$ times at most. As a result, $f_3(n)$ can be expressed as follows:

$$f_3(n) = \lceil \frac{n}{\lfloor n^{2/3} \rfloor} \rceil + \mathcal{O}(\sqrt{\lfloor n^{2/3} \rfloor}).$$

We can remove the floors, because we're estimating an upper bound, and we can remove the ceiling by adding a constant to serve as the remainder of the division, or by simply replacing n with $2n$, an upper bound:

$$f_3(n) = \frac{2n}{n^{2/3}} + \mathcal{O}(n^{2/3^{1/2}}) = 2n^{1/3} + \mathcal{O}(n^{1/3}) = \mathcal{O}(n^{1/3}).$$

And it's obvious that the asymptotically slower growth is maintained for $k = 3$:

$$\lim_{n\to\infty} \frac{f_3(n)}{f_2(n)} = \lim_{n\to\infty} \frac{n^{1/3}}{n^{1/2}} = \lim_{n\to\infty} n^{2/6-3/6} = \lim_{n\to\infty} n^{-1/6} = \lim_{n\to\infty} \frac{1}{n^{1/6}} = 0.$$

Now assume the same strategy holds for some k, and f_k is bound by $\mathcal{O}(n^{1/k})$ as a result. If we had an additional jar, or $k+1$, and follow the same strategy and drop the first jar at multiples of $\lfloor n^{k/(k+1)} \rfloor$, and recursively look for the target rung, we could express f_{k+1} as:

$$f_{k+1}(n) = \frac{2n}{n^{k/(k+1)}} + \mathcal{O}(n^{k/(k+1)^{1/k}}) = 2n^{1/(k+1)} + \mathcal{O}(n^{1/(k+1)}) = \mathcal{O}(n^{1/(k+1)}).$$

And

$$\lim_{n\to\infty} \frac{f_{k+1}(n)}{f_k(n)} = \lim_{n\to\infty} \frac{n^{1/(k+1)}}{n^{1/k}} = \lim_{n\to\infty} n^{\frac{k}{k(k+1)} - \frac{k+1}{k(k+1)}} = \lim_{n\to\infty} \frac{1}{n^{1/k(k+1)}} = 0.$$

(Note: this limit approaches 0 as long as $k < n$, which would pretty much have to be an assumption.)

3 Chapter 3, Exercise 4

Inspired by the example of that great Cornellian, Vladimir Nabokov, some of your friends have become amateur lepidopterists (they study butterflies). Often when they return from a trip with specimens of butterflies, it is very difficult for them to tell how many distinct species they've caught—thanks to the fact that many species look very similar to one another.

One day they return with n butterflies, and they believe that each belongs to one of two different species, which we'll call A and B for purposes of this discussion. They'd like to divide the n specimens into two groups – those that belong to A and those that belong to B – but it's very hard for them to directly label any one specimen. So they decide to adopt the following approach.

For each pair of specimens i and j, they study them carefully side by side. If they're confident enough in their judgment, then they label the pair (i, j) either "same" (meaning they believe them both to come from the same species) or "different" (meaning they believe them to come from different species). They also have the option of rendering no judgment on a given pair, in which case we'll call the pair *ambiguous*.

So now they have the collection of n specimens, as well as a collection of m judgments (either "same" or "different") for the pairs that were not declared to be ambiguous. They'd like to know if this data is consistent with the idea that each butterfly is from one of species A or B. So more concretely, we'll declare the m judgments to be consistent if it is possible to label each specimen either A or B in such a way that for each pair (i, j) labeled "same", it is the case that i and j have the same label; and for each pair (i, j) labeled "different", it is the case that i and j have different labels. They're in the middle of tediously working out whether their judgments are consistent, when one of them realizes that you probably have an algorithm that would answer this question right away.

Give an algorithm with running time $\mathcal{O}(m + n)$ that determines whether the m judgments are consistent.

Answer:

We start by constructing an undirected graph, with n nodes representing the butterflies. We simply draw an edge between two nodes if there exists a confident judgement made between their corresponding butterflies, and draw no edge between ambiguous pairs. We can differentiate between "same" and "different" judgements in a number of ways, but for simplicity we can just maintain two different sets of node addresses within each node. We also maintain an additional bit for a "temporary" specimen assignment.

```
struct Node{
    bool visited = False;
    bool tmp_assign;
    Node* same_connections;
    Node* different_connections;
}
```

Constructing this graph (as an array) is of order $\mathcal{O}(m+n)$, because we initialize a Node for each butterfly (n), and draw two edges between every pair in the set of judgements ($2m$).

We can then perform a breadth-first search (BFS) on the graph (or each component given that it's disconnected due to ambiguous pairs), starting at some node s representing a butterfly. Our BFS queue can be modified a bit to store pointers to nodes, as well as their parent node:

```
struct QueueItem{
    Node* node_ptr;
    Node* parent_ptr;
    bool same_as_parent;
}

Queue<QueueItem> queue;
```

We can start off with the starting node `s`, temporarily assign it to species 1 (`s->tmp_assign = 1`), add its "same" connections to the BFS queue as with their `same_as_parent` bit set to 1, and add its "different" connections to the BFS queue with their `same_as_parent` bit set to 0. We continue running BFS by visiting children nodes by dequeuing them, temporarily assigning them to the same species as their parent if `same_as_parent`, otherwise assigning them to the opposite species. BFS is bound by $\mathcal{O}(m+n)$.

We then check the results in order to see if the judgements are consistent. For every judgement of m judgements

1. If they are labeled "same" and their `tmp_assign` bits do not match, we terminate because this is an inconsistency.

2. If they are labeled "different" and their `tmp_assign` bits match, we terminate because this is an inconsistency.

The loop above is of order $\mathcal{O}(m)$.

Therefore, the entire procedure is still bound by $\mathcal{O}(m + n)$.

4 Chapter 3, Exercise 11

You're helping some security analysts monitor a collection of networked computers, tracking the spread of an online virus. There are n computers in the system, labeled $C_1, C_2, ..., C_n$, and as input you're given a collection of trace data indicating the times at which pairs of computers communicated. Thus the data is a sequence of ordered triples (C_i, C_j, t_k); such a triple indicates that C_i and C_j exchanged bits at time t_k. There are m triples total.

We'll assume that the triples are presented to you in sorted order of time. For purposes of simplicity, we'll assume that each pair of computers communicates at most once during the interval you're observing.

The security analysts you're working with would like to be able to answer questions of the following form: If the virus was inserted into computer C_a at time x, could it possibly have infected computer C_b by time y? The mechanics of infection are simple: if an infected computer C_i communicates with an uninfected computer C_j at time t_k (in other words, if one of the triples (C_i, C_j, t_k) or (C_j, C_i, t_k) appears in the trace data), then computer C_j becomes infected as well, starting at time t_k. Infection can thus spread from one machine to another across a sequence of communications, provided that no step in this sequence involves a move backward in time. Thus, for example, if C_i is infected by time t_k, and the trace data contains triples (C_i, C_j, t_k) and (C_j, C_q, t_r), where $t_k \leq tr$, then C_q will become infected via C_j. (Note that it is okay for t_k to be equal to t_r; this would mean that C_j had open connections to both C_i and C_q at the same time, and so a virus could move from C_i to C_q.)

For example, suppose $n = 4$, the trace data consists of the triples

$$(C_1, C_2, 4), \quad (C_2, C_4, 8), \quad (C_3, C_4, 8), \quad (C_1, C_4, 12),$$

and the virus was inserted into computer C_1 at time 2. Then C_3 would be infected at time 8 by a sequence of three steps: first C_2 becomes infected at time 4, then C_4 gets the virus from C_2 at time 8, and then C_3 gets the virus from C_4 at time 8. On the other hand, if the trace data were

$$(C_2, C_3, 8), \quad (C_1, C_4, 12), \quad (C_1, C_2, 14),$$

and again the virus was inserted into computer C_1 at time 2, then C_3 would not become infected during the period of observation: although C_2 becomes infected at time 14, we see that C_3 only communicates with C_2 before C_2 was infected. There is no sequence of communications moving forward in time by which the virus could get from C_1 to C_3 in this second example.

Design an algorithm that answers questions of this type: given a collection of trace data, the algorithm should decide whether a virus introduced at computer C_a at time x could have infected computer C_b by time y. The algorithm should run in time $\mathcal{O}(m + n)$.

Answer:

To solve this problem, we're going to construct a directed graph and traverse it with a breadth-fist search. Visited nodes in the end will tell us whether or not a specific computer was infected by a specific time, given a virus that was introduced at another computer at an earlier time. The nodes in this graph will have a (computer, time) structure, maintaining the factor of time together with the directions in the graph (edges from t to times greater than or equal to t only).

We define each node with both the computer index and a time value, as well as a pointer to another node.

```
class Node{
    bool visited = False;
    int time = -1, index = -1;
    Node* connections;
    void construct(int i, int t);
}
```

We then define a linked list with a complexity of $\mathcal{O}(1)$ for insert:

```
class<T> LinkedListItem{
    T* self = null,  ptr_to_next = null;
    void construct(T* x) {this->self = x};
}
class LinkedList{
    LinkedListItem* start = null, end = null;
    void insert(Node* n){
        LinkedListItem x;
        x->self = n;
        if (start == null && end == null){
            start = &x;
            end = &x;
        }
        else{
            this->end->ptr_to_next = &x;
            this->end = &x;
        }
    }
}
```

Given the number of computers, n, we initialize an array of size n of `LinkedLists`, one for each computer, called A ($\mathcal{O}(n)$). We then construct the graph by going through the trace data.
For every triple (C_i, C_j, t):

1. ```
   if (A[i]->end != null && A[i]->self->time == t)
       Node* a = A[i]->self
   else
       Node* a = &(new Node(i, t))
   ```

2. ```
   if (A[j]->end != null && A[j]->self->time == t)
       Node* b = A[j]->self
   else
       Node* b = &(new Node(j, t))
   ```

3. `a->connections->insert(b)`

4. `b->connections->insert(a)`

5. `A[i]->insert(a)`

6. `A[j]->insert(b)`

The $\mathcal{O}(1)$ insertion is important here to ensure this loop is bound by $\mathcal{O}(m)$, where m is the number of iterations or trace data points. It should be noted that the if-else statements ensure there won't be any duplicates **because the trace data is sorted by time**.

Now assuming a virus was introduced at computer C_a at time x, we simply search the linked list `A[a]`:
`LinkedListItem* q = A[a]->start`
While `q != null` and `q->self != null` and `q->self->time <= x`:

1. `q = q->ptr_to_next`

If `q == null`, then the virus will not infect any other computer. (This loop is $\mathcal{O}(m)$.)
Otherwise, we can do a BFS on the graph with `q` as the starting node. We can complete the search in $\mathcal{O}(m)$. That still remains true because even with the bidirectional connections, we still have $2m$ edges, and $\mathcal{O}(2m)$ nodes (for every triple, we add at most two new nodes to the graph, and at most two new edges).

We finally search linked list $A[b]$ to figure out whether or not C_b was infected by the virus by time y, by checking if it was visited:
`bool infected = False`
`LinkedListItem* q = A[b]->start`
While `q != null` and `q->self != null` and `q->self->time <= y`:

1. `q = q->ptr_to_next`
2. `infected = q->self->visited`

`infected` will reveal whether or not C_b was infected by time y. (This loop is $\mathcal{O}(m)$.)
Because all operations in this algorithm were bound by $\mathcal{O}(m)$, and the initialization of the array was $\mathcal{O}(n)$, the algorithm will run in $\mathcal{O}(m+n)$.