Greedy Algorithms

Ish Shah

November 25, 2024

1 Introduction

Recently, in this course, you have been discussing greedy algorithms as solutions to certain problems:

- minimum spanning tree construction with Kruskal/Prim,
- Huffman encoding,
- greedy set cover.

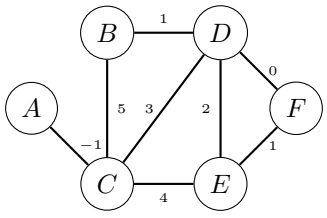
1.1 Kruskal's algorithm

Let's consider one of these algorithms as an example: Kruskal's algorithm, before we consider greedy algorithms more generally. Recall that Kruskal's algorithm follows a procedure as outlined below (expressed in pseudocode) to construct a minimum spanning tree of a weighted graph; that is, the subtree with least total weight:

Algorithm 1 Kruskal's algorithm to find a minimum spanning tree

```
procedure KRUSKAL(G)                    ▷ G = (V, E) is a graph with vertices V, edges E
    sort all edges in E by weight
    added ← 0
    i ← 1
    while added < |V| − 1 do
        if the i-th edge in E does not introduce a cycle then    ▷ check efficiently using union-find
            add the edge to the MST
        i ← i + 1
```

As an example, if we consider the graph



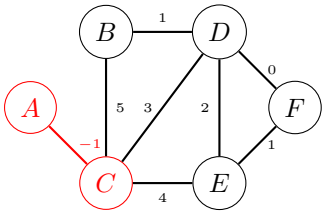
and would like to construct an MST, we can first sort the edges by weight, giving the following:

$$[(AC, -1), (DF, 0), (BD, 1), (EF, 1), (DE, 2), (CD, 3), (CE, 4), (BC, 5)].$$

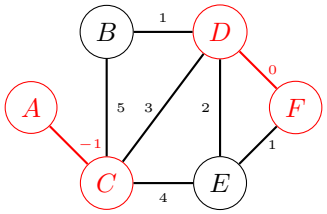
Performing this step-by-step, we have the following as seen on the next page (with added edges in red, unused edges in blue), we obtain the following result:

- Step 1: $[(AC,-1),(DF,0),(BD,1),(EF,1),(DE,2),(CD,3),(CE,4),(BC,5)]$
- Step 2: $[(AC,-1),(DF,0),(BD,1),(EF,1),(DE,2),(CD,3),(CE,4),(BC,5)]$
- Step 3: $[(AC,-1),(DF,0),(BD,1),(EF,1),(DE,2),(CD,3),(CE,4),(BC,5)]$
- Step 4: $[(AC,-1),(DF,0),(BD,1),(EF,1),(DE,2),(CD,3),(CE,4),(BC,5)]$
- Step 5: $[(AC,-1),(DF,0),(BD,1),(EF,1),(DE,2),(CD,3),(CE,4),(BC,5)]$
- Step 6: $[(AC,-1),(DF,0),(BD,1),(EF,1),(DE,2),(CD,3),(CE,4),(BC,5)]$ **(end)**

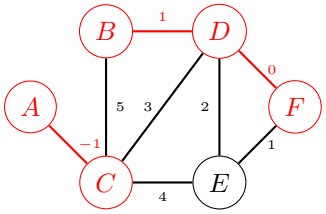
(1)



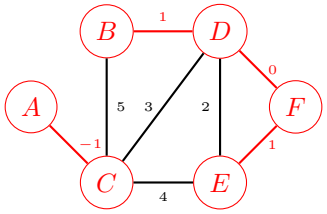
(2)



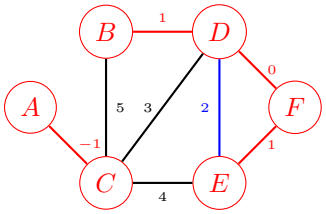
(3)



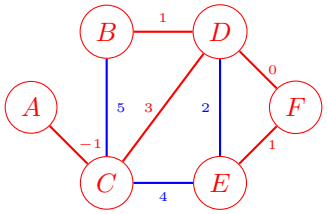
(4)



(5)



(6)



In red from graph (6), we have our minimum spanning tree.

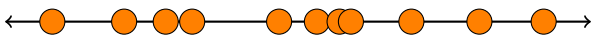
Kruskal's algorithm works out quite nicely: we identify some trick (namely, sorting the edges by weight), and then we add all the edges in sorted order while we can do so without introducing a cycle.

We had a **heuristic** of making optimal choices at every step, by choosing the remaining edge of minimum weight that did not introduce a cycle. We can apply such ideas to other problems as well–this is the common thread which unifies various greedy algorithms. Now, we consider some other such algorithms.

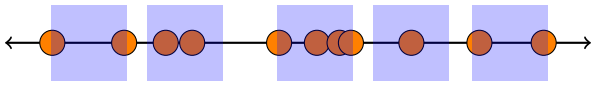
2 Other greedy algorithms: example 1

What we will see here is another problem where a greedy heuristic/strategy is needed–and we will see an example of a greedy strategy that fails, too.

Suppose we are given a set of n real numbers X, so $X = \{x_1, x_2, \ldots, x_n\}$. You can image them as a set of points plotted on a number line:



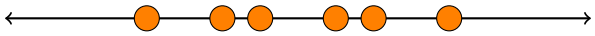
We can cover these points using **closed unit intervals**–intervals of the form $[a, a+1]$ (which contain endpoints):



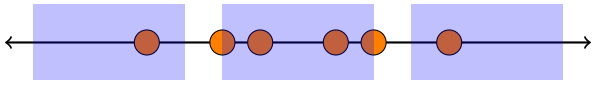
Your objective is to find the minimum number of closed unit intervals that are needed to cover all points of X, and construct a set Y of intervals which covers X while attaining this minimum. There are multiple ideas for a greedy strategy we can consider here.

2.1 A failed attempt

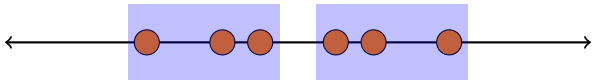
Consider this idea: First, use a unit interval that covers the maximum number of points possible. Then, use a unit interval that covers the maximum number of yet-uncovered points possible. Repeat until every point is covered.
For now, set aside the question of implementation details–this algotithm still faces issues regardless. Consider $X = \{-1, -0.5, -0.25, 0.25, 0.5, 1\}$:



The closed unit interval which covers the most points is $[-0.5, 0.5]$, which contains a total of 4 points: $\{-0.5, -0.25, 0.25, 0.5\}$. Then, we can cover the remaining two points with one interval for each one:



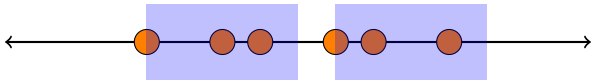
However, it's possible to do better, using only two intervals. For example, consider $[-1.125, -0.125]$ and $[0.125, 1.125]$:



So, this greedy strategy clearly fails.

2.2 A correct approach

How about this idea: we sort the points from least to greatest. Starting from the least, we place an interval with left endpoint at this point. Then, do the same for the least point that is not covered, and repeat this until every point is covered. This seems to work on our previous example, creating the cover of X given by $Y = \{[-1, 0], [0.25, 1.25]\}$:



It's also clear that this solution runs in $O(n \log n)$ time, with the sorting taking $O(n \log n)$ and the placement of intervals taking $O(n)$. Again, this is still a greedy algorithm, even if the strategy we are using is different this time; our heuristic is to choose the next available point after putting them in sorted order.

2.2.1 A justification for this algorithm

Is this algorithm correct? Yes, and I will *sketch* (not fully detail) a proof below.

Proof sketch. We know that the smallest point (which is the leftmost on the number line) must be covered by one interval. It's best for the interval covering this point to have said smallest point as its lower endpoint, as that maximizes the number of points that can fall in this interval (since nothing is to the left of this minimum point, but there can be more points to its right).

After this first step, the same idea applies. After we added the first interval, there must be some new smallest uncovered point. Of course, we want to cover this point using an interval with lower endpoint equal to this point for the same reason we did it previously. Apply the same argument again to all the following steps. □

That is essentially the idea! This is not written very formally, but the idea outlined above can be used to write a **proof of correctness** for our algorithm. This sketch should, at a minimum, convince you that it is at least reasonable to expect that such an algorithm works.

Exercise: Write a formal proof for this algorithm. *Hint: it may be helpful to use induction.*

3 Other greedy algorithms: example 2

Suppose you have n customers, each of which has requested a job be done. The amount of time each job takes is known–person i's job takes time T_i. You can only do one job at a time, and so people have to wait. This wait time is the sum of all the known times up to the current task:

$$\text{Person } j\text{'s time spent waiting} = T_j + \sum_{k \in \mathcal{C}} T_k$$

where $\mathcal{C}$ is the set of completed jobs up to the time where you decide to take on person j's job. Your goal is to minimize the average waiting time faced (or equivalently, the sum of the waiting times each person faces).

3.1 A solution and proof

The solution to this problem is simple: we sort the jobs by time required in ascending order and complete them exactly in this order, for an $O(n \log n)$ algorithm. Showing correctness, however, is a bit more difficult. To prove correctness formally, we will pursue what is known as an **exchange argument**.

Proof. Suppose that the optimal order is not in sorted order, so at some point we have a job followed right after by a shorter one (this is known as an **inversion**). Let's say the longer job is the k-th job, and the shorter one is the $k+1$-th job, so $T_k > T_{k+1}$ (remember this inequality!). The number of jobs after the k-th job remaining are $n-k$. Notice that the current arrangement adds

$$t_{\text{orig}} = (n-k)T_k + (n-k-1)T_{k+1}$$

across jobs k and $k+1$ to the total waiting time. If we are to swap the order in which we perform the two tasks, making the shorter one go first, the amount of time the two jobs contribute is now

$$t_{\text{after}} = (n-k)T_k + (n-k-1)T_{k+1}.$$

Taking the difference of the two times gives

$$\begin{aligned} t_{\text{orig}} - t_{\text{after}} &= (n-k)T_k + (n-k-1)T_{k+1} - [(n-k)T_k + (n-k-1)T_{k+1}] \\ &= [(n-k) - (n-k-1)](T_k - T_{k+1}) = T_k - T_{k+1} > 0, \end{aligned}$$

so $t_{\text{orig}} > t_{\text{after}}$. Thus, fixing an inversion always decreases the total wait time faced by each person, so it is optimal to have no inversions. Thus, it is optimal to never have inversions present–this is exactly what the greedy algorithm gives. □

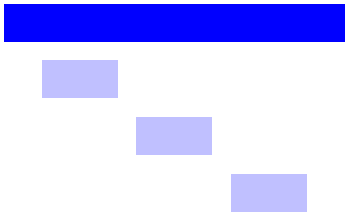
4 Further reading I suggest

For simply practical purposes (if nothing else), I believe it is good to be familiar with examples of greedy scheduling-related algorithms–knowledge of such algorithms may be important in certain situations such as technical interviews! I will state an example of such a problem and provide you with a reference for further reading (which includes a more detailed solution and proof).

The example of such a problem is as follows. Suppose we have a collection of jobs, each with start time s_i and end time f_i, for $i = 1, \ldots, n$. Your goal is to find the maximum number of jobs which do not have overlap (except possibly at the starting/ending times). To do this, certain ideas may seem promising but not work:

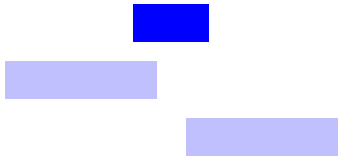
- Sort jobs by starting time, then add the first one available which does not conflict with any previously-chosen jobs.

  Counterexample:



- Sort jobs by duration $f_i - s_i$ and do the same.

  Counterexample:



- Sort jobs by number of conflicts, then do the same.

  Counterexample: exercise. (See linked notes for a valid counterexample.)

The correct solution turns out to be a similar idea with sorting: but the correct strategy is to sort by earliest *end times* and then add them in ascending order as long as they don't conflict with the set of previously-chosen jobs.

For more information on this solution as well as a proof of correctness, I suggest that you consult these slides, from which I have just summarized some content in section 4.1, or these notes. The former also contains some other similar problems that would be good to have a look at as well–I would suggest seeing the content in sections 4.1 and 4.2.