Data Structures and Algorithms: The formal notes

Josh Felmeden

October 16, 2019

Contents

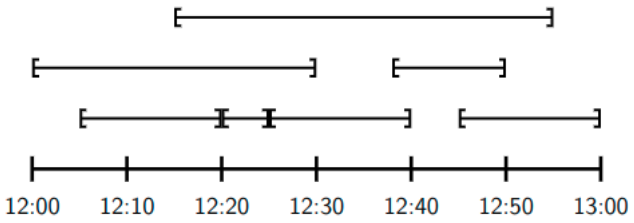
1 Greedy algorithms

1.1 Interval scheduling

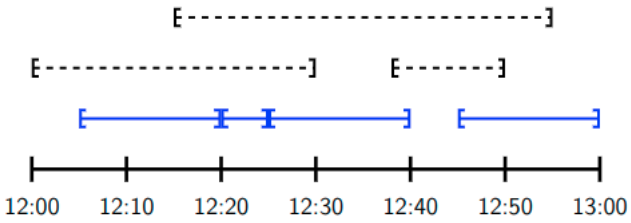
Let's suppose that you're running a satellite imaging service. Taking a satellite picture of an area isn't instant and can take some time. It can also only be done on the day where the satellite's orbit is lined up correctly. Say you want to request some images to be taken from said satellite, each of which can only be taken at certain times and you can only take one picture at a time. How do we satisfy as many requests as possible?

The requested satellite times that we have to deal with are: 12:00-12:30, 12:05-12:20, 12:15-12:55, 12:20-12:25, 12:38-12:50, and 12:45-13:00.

If we visualise this in a graph, we get:



If we take a greedy algorithm approach to assigning these slots, we could do something like assign the slot that finishes earliest, and then repeat doing this until we have reached the end. For example, the slot that finishes fastest is 12:05-12:20, so we assign this. This now removes the ability for both 12:10-12:30 and 12:15-12:55 to be assigned, so we remove these. This continues until we end up with something looking like this:



This means that we satisfy four requests (which is actually the maximum possible, so well done us).

We can formalise this by saying that a **request** is a pair of integers (s, f) with $0 \leq s \leq f$.

The algorithm that we're left with is this:

```
public sub greedySchedule
    sort R
    for each i in {1 ... n} do
        if s_i >= lastf then
            A.append(s_i, f_i)
            lastf = f_i
        end if
    next
end sub
```

Now, we need to prove that the output is actually a **compatible subset** of R. This is sort of intuitive because the set we added doesn't break compatibility, since $s \geq$ `lastf` and `lastf` is the latest finish time that's already in A.

We can formalise this with a loop invariant. At the start of the ith iteration, we see that

- A contains a compatible subset $\{(S_1, F_1), \ldots, (S_t, F_t)\}$ of R.
- `lastf` = $\max(\{0\} \cup \{F_j : j \leq t\})$

The base case ($i = 0$) is immediate because $A = []$. The induction step is that A was compatible at the start of the iteration, and therefore if we append a pair s_i, f_i to A then $s \geq$ `lastf` $\geq F_j$ for all $f \leq t$. This means that (s_i, f_i) is compatible with A.

Greedy algorithms definition

Greedy algorithms are actually an informal term and people have different definitions. The definition that we're going to use is:

- They start with a sub-optimal solution.
- They look over all the possible improvements and pick the one that looks the best at the time.
- They never backtrack in 'quality'.

Greedy algorithms might fail; it's not enough to just do the obvious thing at each stage. While the algorithms might fail initially, we can use the knowledge that we gained from the results of the algorithm to design a more correct one.

2 Graphs

Graph Definition

A **graph** is a pair $G = (V, E)$ where $V = V(G)$ is a set of **vertices** $E = E(G)$ is a set of **edges** contained in $\{\{u, v\} : u, v \in V, u \neq\}$

Walk Definition

A **walk** in a graph $G = (V, E)$ is a sequence of vertices such that $\{v_i, v_{i+1} \in E$ for all $i \leq k - 1$ We say that the walk is from v_0 to v_k and call k the length of the walk.

2.1 Euler walks

An **Euler walk** is one that contains every edge in G exactly once.

Two graphs might be **equal**. This is the case when two graphs $G_1 = (V_1, E_1)$ and $G_2 = (V_2, E_2)$ are equal and written $G_1 = G_2$ if $V_1 = V_2$ and $E_1 = E_2$. This does present some issues, however, because sometimes graphs look like they should be equal, when they're not because the edges are labelled differently.

This is where **isomorphism** comes in. G_1 and G_2 are **isomorphic** if there's a bijection $f : V_1 \to V_2$ such that $\{f(u), f(v)\} \in E_2$ if and only if $\{u, v\} \in E_1$.

Intuitively, this means that $G_1 \xrightarrow{\sim} G_2$ if they are the same graph but the vertices are relabelled.

In a graph $G = (V, E)$, the **neighbourhood** of a vertex v is the set of vertices joined to v by an edge. Formally, $N_G(v) = \{w \in V : \{v, w\} \in E$. Also, for all sets of vertices $X \subseteq V = \cup_{v \in X} N_G(v)$

The **degree** of a vertex v is the **number** of vertices joined to v. Formally: $d_G(v) = |N_G(v)|$

Theorem: If G has an Euler walk, then either:

- Every vertex of G has even degree or
- All but two vertices v_0 and v_k have even degree, and any Euler walk must have v_0 and v_k as endpoints.

Does every single graph that satisfies both of these conditions have an Euler walk? No, because the graphs need to be **connected**.

Within a graph, we also have subgraphs and induced subgraphs. A **subgraph** $H = (V_H, E_H)$ of G is a graph with $V_H \subseteq$ and $E_H \subseteq E$. H is an **induced subgraph** if $V_H \subseteq V$ and $E_H = \{e \in E : e \subseteq V_H\}$.

For all vertex sets $X \subseteq V$, the graph is **induced** by X is:

$$G[X] = (X, \{e \in E : e \subseteq X\})$$

A **component** H of G is the maximal connected induced subgraph of G, so $H = G[V_H]$ is connected, but $G[V_H \cup \{v\}]$ is disconnected for all $v \in V \backslash V_H$.

Theorem: Let $G = (V, E)$ be a **connected** graph, and let $u, v \in V$. Then, G has an Euler walk from u to v if and only if either:

1. $u = v$ and every vertex of G has even degree, or
2. $u \neq v$ and every vertex of G has even degree except u and v.