Network Flow For Scheduling?

Muhammad Islam

December 11, 2022

1 Introduction

A graph $G(V, E)$ is comprised of a set of nodes or vertices, V, and a set of edges, E, which represents all possible pairs of distinct vertices, (u, v) in the graph. Specifically, a graph can either be directed or undirected. In a directed graph, each pair in E has to be an ordered pair such that (u, v) and (v, u) are distinct edges. In an undirected graph, however, such pairs aren't considered distinct edges. A network, G, is a directed graph with each edge $(u, v) \in E$ having a nonnegative capacity $c(u, v) \geq 0 \forall (u, v) \in E$. A flow in G is a real-valued function, $f : V \times V \to \mathbb{R}$. This function has to satisfy two conditions:

1. $\leq f(u, v) \leq c(u, v)\ \forall (u, v) \in E$ and $0 \leq f(u, v) \leq c(u, v)$
2. $\sum_{v \in V} f(v, u) = \sum_{v \in V} f(u, v)\ \forall u, v \in V - s, t$

where s, t are two distinct vertices in G (Lerson et al. 709). The first condition ensures that a flow doesn't exceed the capacity of any edge. The second condition ensures that the flow entering into any node besides s and t is the same as the flow leaving it.
Such networks are applicable to many domains such as transportation, communication, hydraulics, finance, etc (Wayne 2). As such, problems associated with this type of graph have been extensively over the past 68 years is a flow network such as the minimum mean cycle (Manber 238). One such problem is the maximum flow problem, which can be stated as follows:

> Given two distinct vertices, s, t, in a flow network G, find a flow f in E such that the total flow along any path from s to t is maximized.

Another such problem is called the minimum cut problem. A cut is the set of all edges $(u, v) \in E \mid u \in A, v \in B$ where $A \subset V \mid s \notin A$ and $B = V - A$. The capacity of a cut is the total sum of the capacities of its edges. The minimum cut problem, then, can be stated as follows:

> Given two distinct vertices, s, t, in a flow network G, find a cut C in E such that the capacity, c, of the cut is minimized.

While I will not provide a formal proof, the minimum cut problem is, in fact, the same problem as that of the maximum flow problem.
There are two main algorithms to solve the problem. The first algorithm is the Ford-Fulkerson algorithm, the pseudocode of which is provided below:

Algorithm 1 Ford-Fulkerson Algorithm

```
for (u, v) ∈ E do
    f(u, v) = 0
end for
while an path p exists in residual network G_f do
    c_f(p) = min c_f(u, v) : (v, v) ∈ p
    for (u, v) ∈ p do
        if (u, v) ∈ E then
            f(u, v) = f(u, v) + c_f(p)
        else
            f(u, v) = f(u, v) + c_f(p)
        end if
    end for
end while
```

The total running time of this algorithm is $\mathcal{O}(E * m)$ where m is the maximum flow. A more robust form of this algorithm is known as the Edmonds-Karp algorithm. This algorithm finds the p path using breadth first search, which allows the algorithm to run in $\mathcal{O}(VE^2)$ running time.

2 Background

The topic explored in this paper is inspired by the real-world problem of time management experienced by most ordinary people including students, employees, etc. Usually, an individual has several tasks or assignments which they wish to complete by a certain time called a deadline, while attending to several events such as sleep, lunch, dinner, classes/professional meetings, social gatherings, etc. Since events are unavoidable with fixed starting times and ending times, the time an individual has to complete tasks and assignments is limited to the time unoccupied by events called "free time". There are two main aspects of this problem that are worth noting:

- The time needed to complete each task, called its "duration", need not be the same.
- The deadlines of each task need not be the same.

These two aspects are important because if they didn't hold true, the solution to this problem would have been simple: divide the free time available before the deadline by the number of tasks and allocate the resulting time for each task, with the order in which the tasks are performed being irrelevant. However, because not all tasks take the same amount of time to complete, an equal amount of time wouldn't be ideal. Of course, the solution to the problem would still be trivial: assign a weight corresponding to the time needed for each task divided by the minimum amount of time needed for a task out of all tasks; divide the free time available before the deadline by the total weight of all tasks; then finally allocate time for each task corresponding to the product of the weight of the task and the previous result.The order in which the tasks are performed would also be irrelevant in this situation.

Yet, because not all tasks have the same deadline, order matters since, for instance, the task with the earliest deadline must be completed before any other task. In addition, this means that the times of each task may be split up into several blocks, as will be seen when I try to represent the problem. Not only that, there is no single fixed amount of free time. For tasks with earlier

deadlines, the amount of free time is less than tasks with later deadlines even if the task with the earlier deadline may need more time to complete.

I formalize the problem as follows:

> Define a task to be a record with a name, a duration, m, and a deadline f and an event to be a record with a name, start time, and end time. Given a set of tasks T of size n sorted by their deadlines, a set of events E, and a start date, d_0, for each task t_i, allocate at most i time blocks each of length l_j, such that $\frac{\sum_1^i l_j}{m}$ is maximized for all tasks.

It's important to note that the problem considers all tasks Before examining this problem, I will first demonstrate an equivalent representation of the problem, comparing it with that of a flow network. I will then explore a variant of the problem in which $\frac{\sum_1^{i-1} l_j}{m}$ is maximized rather than $\frac{l}{m}$. I will then finally propose an algorithm for the problem and generalize it to similar networks.

3 Findings

3.1 Representation of the Problem

To represent this problem, we can first calculate the amount of free time for each $t_i \in T$ between t_i and t_i other than the first task. For the first task, t_1, the amount free time is calculated between the start date and t_i. Let D represent the set of all of these amounts. We can make a square n by n table where the columns are for D and the rows are for T. An example of three tasks is shown in Table 1, with an extra task column, total column, and total row. Each entry, b_{ij}, where i is the row and j is the column of the corresponding entry, represents the length of a time block l_j devoted to doing a task t_i within D_j (the task need not be completed within this time block so l_j may be less than m_i, the corresponding m of task t_i). Entries must be nonzero real numbers. Observe that for each ith row, we can only fill all $b_{ij} \mid j \leq i$ with positive values. This is because after the deadline of the corresponding task, t_i, no time should be spent for the task (for this reason, the problem asks to allocate at most $i-1$ time blocks) This tabular representation, however, doesn't allow us

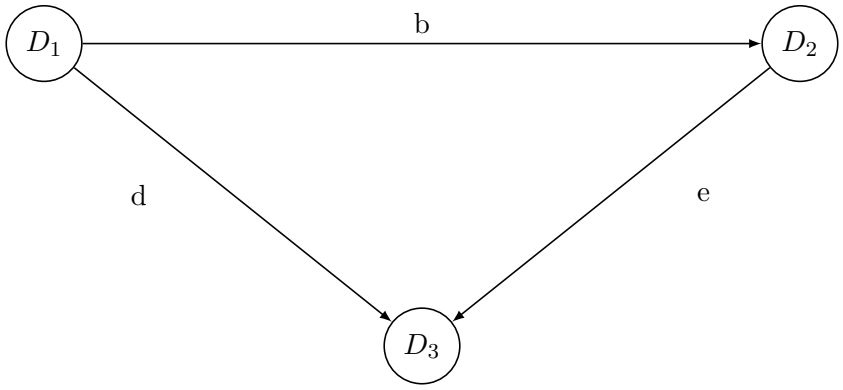
Task	Free Time Before Deadline			Total
t_1	a	0	0	a
t_2	b	c	0	b + c
t_3	d	e	f	d + e + f
Total	$D_1 = a + b + d$	$D_2 = c + e$	$D_3 = f$	$\sum_{i=1}^{3} D_i = a + b + c + d + e + f$

Table 1:

to understand how each free time amount and task are connected to each other.

Let's instead convert the table to a directed graph with n nodes. Each ith node is assigned the value D_i. Each edge from the node to any other jth node has a weight that represents the length of the time block within D_i devoted for task t_j. One such representation is shown in Figure 1 based off of Table 1.Observe that the in-degree of any ith node will be $i-1$ and that the out-degree will be $n-i$. This also indicates that the first node will have an indegree of 0 whereas the nth node will have an outdegree of 0. In addition, observe that for any task, t_i, $\sum_1^i l_j = D_i + \sum_{j=i+1}^{n} w_{ij} - \sum_{j=1}^{i-1} w_{ji}$ where w_{ij} represents the weight of the directed edge from node i to node j.

One important property that needs to be checked to ensure that this representation is equivalent to the tabular representation of the property is whether $\sum_{i=1}^{n} D_i = \sum_{i=1}^{n} \sum_{j=1}^{i} l_j$ (i.e whether the tabular property of the total sum of the sum of each row being the same as the total sum of the sum of each column is preserved). We can substitute $\sum_{j=1}^{i} l_j$ and obtain $\sum_{i=1}^{n} D_i = \sum_{i=1}^{n} (D_i + \sum_{j=i+1}^{n} w_{ij} - \sum_{j=1}^{i-1} w_{ji})$, showing that $\sum_{i=1}^{n} \sum_{j=i+1}^{n} w_{ij} = \sum_{i=1}^{n} \sum_{j=1}^{i-1} w_{ji}$ has to hold true. This is easily verifiable as any outgoing edge from a node is an ingoing edge to another node.



There are also other restrictions that I will uphold:

1. The total outgoing flow for the ith node must be no more than D_i.
2. The total ingoing flow for the ith node must be no more than m of task t_i.
3. All edges must have non-negative weights.
4. Either the total outgoing flow is 0 or the total ingoing flow is 0.

While the other restrictions stem directly from the problem, the last restriction simplifies the problem since there's a solution to the problem without the restriction if and only if there's a solution with the restriction. This can proved by contradiction: suppose that with the restriction, the problem doesn't have a solution. This implies that with the restriction, there isn't an equivalent $\frac{\sum_1^i l_j}{m}$ for all tasks $t_i \in T$ as that obtained without the restriction. However, this would mean that if $\sum_{j=i+1}^{n} w_{ij} = 0$ or $\sum_{j=i+1}^{n} w_{ij} = 0$, there isn't an equivalent $\sum_1^i l_j = D_i + \sum_{j=i+1}^{n} w_{ij} - \sum_{j=1}^{i-1} w_{ji}$, leading to a contradiction. As for the converse, the proof is trivial.

4 Solution

Similar to a flow network, this graph seems to have a certain source and a sink - node 1 and node n. In addition, for a node, we want to maximize the amount of "flow" to the sink. However, unlike the maximum flow problem, we want to maximize the amount of flow across all nodes. In addition, there's no requirement that the outgoing flow must be equal to the incoming flow.

5 References

1. Ford, L., & Fulkerson, D. (1956). Maximal Flow Through a Network. Canadian Journal of Mathematics, 8, 399-404. doi:10.4153/CJM-1956-045-5 Wayne, Kevin. "Generalized Maximum Flow Algorithms." Princeton University, 1999.

6 Appendix

1. https://github.com/1404mri/cmsc351hpaper/tree/master