Assignment 5 **Wen Han Xu**
CS711008Z ID: 201818018670073
Date: 2018/1/10

1 Load Balance

You have some different computers and jobs. For each job, it can only be done on one of two specified computers. The load of a computer is the number of jobs which have been done on the computer. Given the number of jobs and two computer ID for each job. Your task is to minimize the max load.

1.1 Algorithm Description

The *Load Balance* problem can be modeled as a typical **Network MaxFlow** problem. Suppose there are n jobs (denoted as J_i) and m computers (denoted as PC_i). The problem can be graphed in the following way.

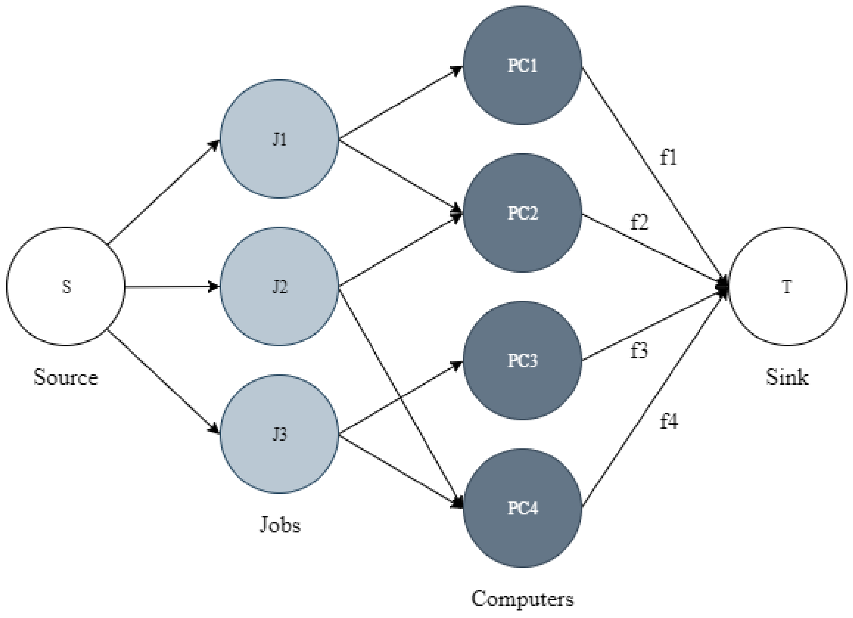

Figure 1: Job Assigment Graph

1. Add 2 virtual node S and T as the source and sink of the graph.
2. Add n edges (S, J_i) with flow capacity 1.
3. Add 2n edges (J_i, PC_{k1}), (J_i, PC_{k2}) with flow capacity 1.
4. Add m edges (PC_i, T) with flow capacity Q.

We note that the flow on edges (PC_i, T) f_i is exactly the load of PC_i, as the conservation constraint requires $f^{in}(v) = f^{out}(v)$. Thus the Load Balance problem can be converted to finding **the minimal Q such that the maximum flow of the network is** n. By observation Q is not larger than n, as at most n flow (jobs) can be assigned in the network. Thus, we can utilize binary search starting from $[0, n]$ to find the optimal Q.

Algorithm 1 Load Balance

```
procedure LoadBalance($G, J, PC$)
    Construct Job Assignemnt Graph
    $Left \leftarrow 1$
    $Right \leftarrow n + 1$
    while $Left < Right$ do
        $Q \leftarrow \frac{Left+Right}{2}$
        Set $C(PC_i, T) = Q$
        if $MaxFlow(S, T) == n$ then
            $Right \leftarrow Q$
        else
            $Left \leftarrow Q + 1$
    return $Left$
```

1.2 Proof of Correctness

The proof will be divided into 2 parts.

Firstly we will prove that the **LoadBalance** problem can be modeled as a **MaxFlow** problem. The flow on edges can be viewed as job assignments. Initially, the source node S activates at most n jobs by sending flows along edge (S, J_i). Then its assigned to one of two specified computers along edges (J_i, PC_{k1}) and (J_i, PC_{k2}). The flow from PC_i to T represents the work load of PC_i. Finding the maximum flow is equally finding the maximum amount of jobs that can be scheduled.

Secondly we will prove that by utilizing **BinarySearch**, the minimalest Q can be found. In 1, when $MaxFlow(S, T) == n$, it means that all n jobs can be arranged in the network, thus the capacity Q is sufficiently large, we need to scale it down to find a smaller Q. When $MaxFlow(S, T) < n$, the capacity Q is too small for all jobs to be scheduled, thus we need to scale it up. At the end of the loop, we will locate the smallest Q that allows all the jobs to be sent into the network.

1.3 Time Complexity

The binary search will take $O(\log n)$ loops in the worst case. There are $n + m + 2$ nodes and $n + 2n + m$ edges in the network. When implemented using Dinitz's algorithm, $MaxFlow$ takes $O(m'n'^2)$ time ,where $m' = n + 2n + m = 3n + m$ and $n' = n + m + 2$. Thus the overall time complexity is $O(\log n \cdot (m^3 + n^3))$.

2 Matrix

For a matrix filled with 0 and 1, you know the sum of every row and column. You are asked to give such a matrix which satisfies the conditions.

2.1 Algorithm Description

We first model this problem as a **NetworkFlow**. Suppose the matrix is $M \times N$. Let m_i denote the $i-$th row and n_j denote the $j - th$ column. We use $sum(m_i)$ or $sum(n_j)$ to represent the sum of row i or column j. The network graph is constructed in the following way:

1. Add 2 virtual node S and T as the source and sink of the graph.

2. Add M edges (S, m_i) with flow capacity $sum(m_i)$.
3. For each m_i, add m edges (m_i, n_j) (with flow capacity 1) to connect it with n column nodes.
4. Add N edges (n_j, T) with flow capacity $sum(n_j)$.

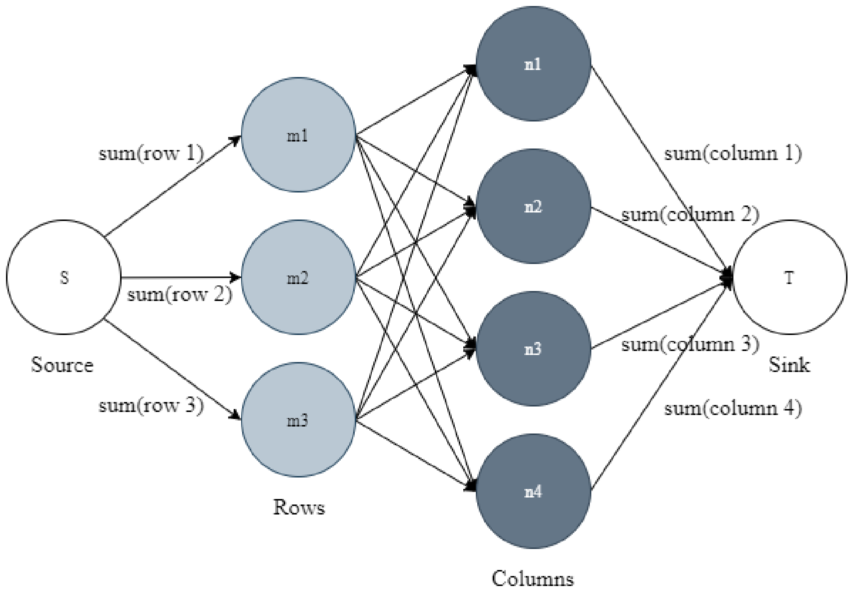

Figure 2: Matrix Network Graph

Algorithm 2 Matrix

```
1: procedure MATRIX($G, J, PC$)
2:     Construct Matrix Network Graph
3:     $f \leftarrow MaxFlow(S, T)$
4:     if $f = \sum sum(m_i) = \sum sum(n_j)$ then
5:         Such matrix $Mat$ exists
6:         $Mat(i, j) \leftarrow f(m_i, n_j)$
7:         return Matrix $Mat$
8:     else
9:         return "No such matrix"
```

2.2 Proof of Correctness

In figure 2 the flow on edge (S, m_i) represents how many "1"s are on row m_i, thus $C(S, m_i) = sum(m_i)$. For each row, its "1"s need to be redirected to several columns, thus we use $f(m_i, n_j)$ to represent that a "1" goes from row m_i to column n_j. By solving the **Matrix Network Graph**, we'll get the matrix construction when the maximum flow f value satisfies $f = \sum sum(m_i) = \sum sum(n_j)$. If the equality doesn't hold, then there exists no matrix that can satisfies the given conditions.

2.3 Time Complexity

There are $n+m+2$ nodes and $n+m+n\times m$ edges in the graph. Solving a **MaxFlow** problem using Dinitz's algorithm takes $(O(n+m+n\times m)(n+m+2)^2)$ time.

3 Problem Reduction

There is a matrix with numbers which means the cost when you walk through this point. you are asked to walk through the matrix from the top left point to the right bottom point and then return to the top left point with the minimal cost. Note that when you walk from the top to the bottom you can just walk to the right or bottom point and when you return, you can just walk to the top or left point. And each point CAN NOT be walked through more than once.

3.1 Algorithm Description

The problem can be modeled as a $MinCostFlow$ problem. The construction is as follows:

1. Make each unit in the matrix a unique node in the network graph.
2. Connect node with its right and bottom neighbours. The added edge has $capacity = 1$ and $cost = 0$.
3. Split each node into 2 *internal nodes*, connected by edge with $capacity = 1$ and $cost = Matrix(i,j)$.
4. Note that for nodes in the top-right and bottom-left corner, its internal edge has $capacity = 2$

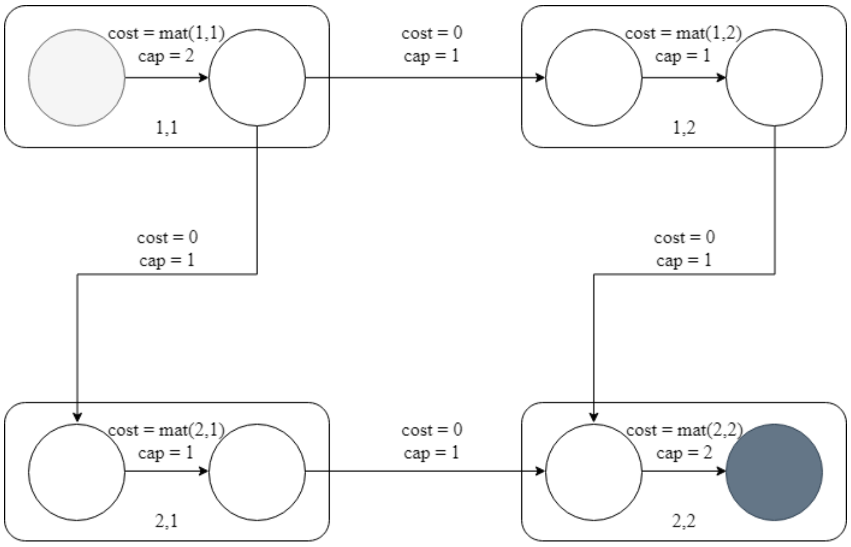

Figure 3: Problem Reduction

The problem has now been converted into a $MinCostFlow$ problem. By solving $MinCostFlow(2)$, we can found the walk path with minimal cost.

Algorithm 3 Problem Reduction

```
1: procedure ProblemReduction($G, J, PC$)
2:     Construct Problem Reduction Graph
3:     $MinCostFlow(2)$
```

3.2 Proof of Correctness

We will illustrate why we can use $MinCostFlow$ to solve the problem. In figure 3, the capacity on edges between nodes is set to 1, which means that a path(node) can only be taken once. The internal edges are weighted. For nodes in the top-right and bottom-left corner, its internal edges' capacities are assigned to 2, so that walks from both directions are considered. When a $MaxFlow$ solution is found with $flow = 2$, we say a walk is possible in the matrix, thus by utilizing Klein's algorithm we can yield the optimal solution with minimal cost.

3.3 Time Complexity

We use **Klein's** algorithm to solve the $MinCostFlow$ problem. The time complexity is $O(fmn)$, where f is the maximum flow, $O(mn)$ is the time used to find a negative cycle using bellman-ford's algorithm.