COMP 360: Algorithm Design Instructor: Yang Cai

Assignment 1 Solution

General Rules: In solving these questions you may consult books but you may not search on the web for solutions. You must write up your final solution yourself. You should drop your solutions in the assignment drop-off box located in the Trottier Building. No late assignments accepted.

1. (15 pts) **Ford-Fulkerson Algorithm:**
   Consider the following maximum flow problem. (Edge capacities are shown.)

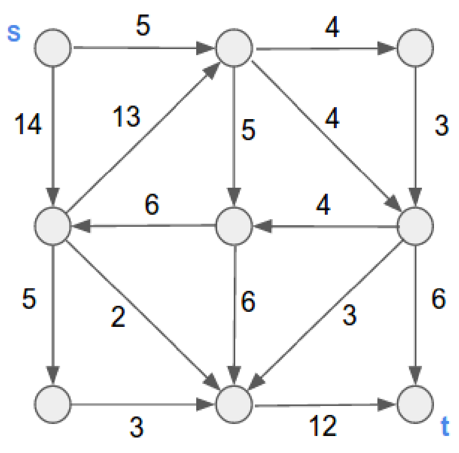


Figure 1: Flow Network for Question 1.

   (a) (5 pts) Apply the Ford-Fulkerson algorithm to find a maximum s-t flow. Use the **shortest augmenting path method** to choose the augmenting path in each iteration.

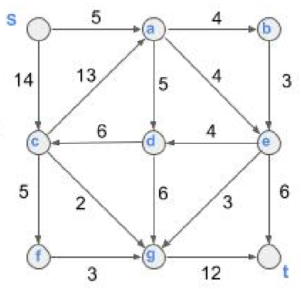
   (b) (5 pts) Repeat (a), but now use the **maximum capacity augmenting path method** to choose the augmenting path in each iteration.

   (c) (5 pts) Prove that your solutions give maximum s-t flows by giving a certificate that shows it is impossible to find a larger flow.

Solution

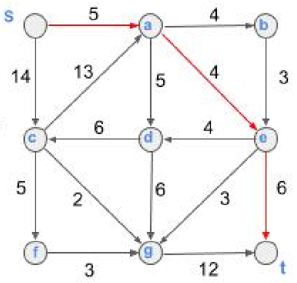
1(a) Use the shortest augmenting path method. There are more than 1 possible shortest augmenting path and you can start with any path.

Original graph: G

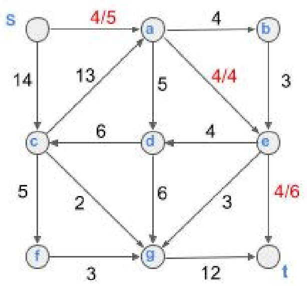


Take a 3-length path
$s \rightarrow a \rightarrow e \rightarrow t$

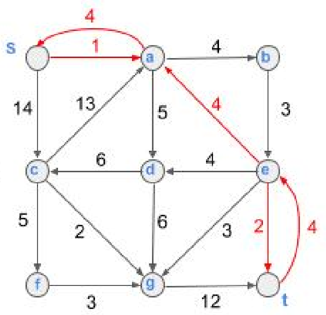
Residual graph: G_f



Original graph: G

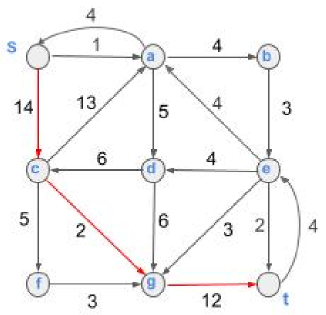


Update the residual graph

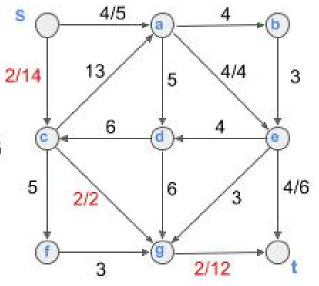


Take a 3-length path
$s \rightarrow c \rightarrow g \rightarrow t$

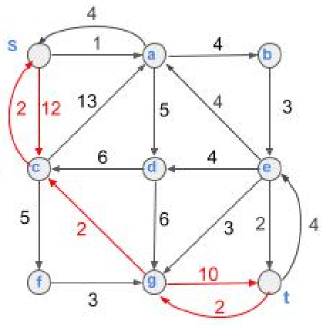
Residual graph: G_f



Original graph: G

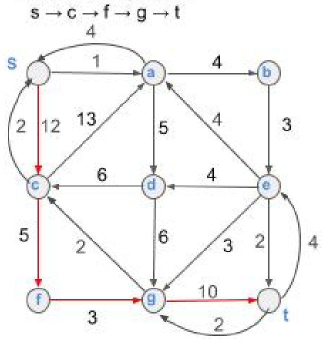


Update the residual graph

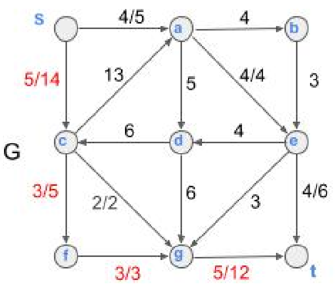


Take a 4-length path
$s \rightarrow c \rightarrow f \rightarrow g \rightarrow t$

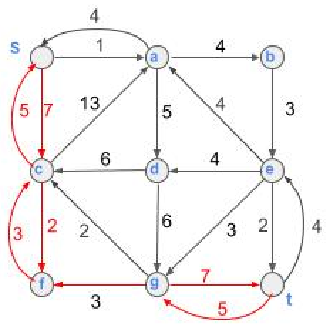
Residual graph: G_f



Original graph: G

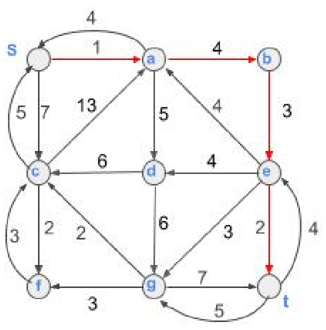


Update the residual graph

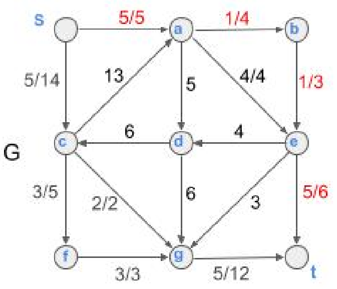


Take a 4-length path
$s \rightarrow a \rightarrow b \rightarrow e \rightarrow t$

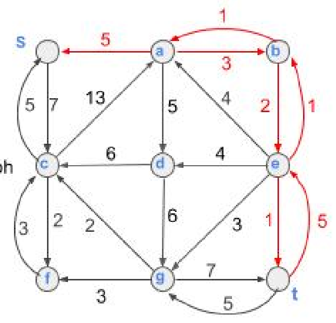
Residual graph: G_f



Original graph: G

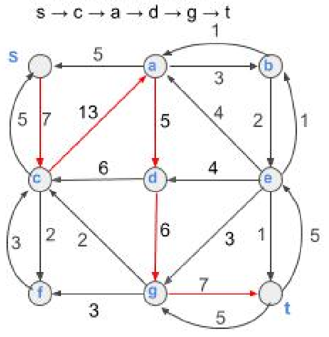


Update the residual graph

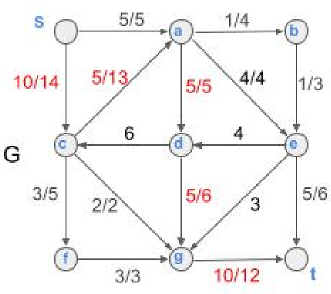


Take a 5-length path
$s \rightarrow c \rightarrow a \rightarrow d \rightarrow g \rightarrow t$

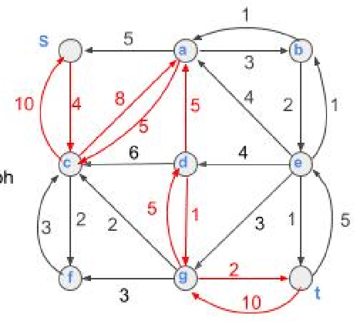
Residual graph: G_f



Original graph: G

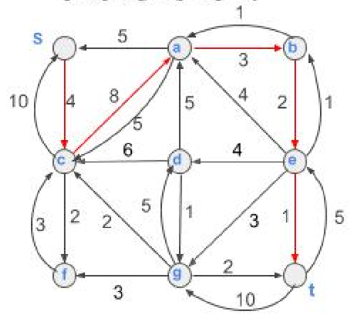


Update the residual graph

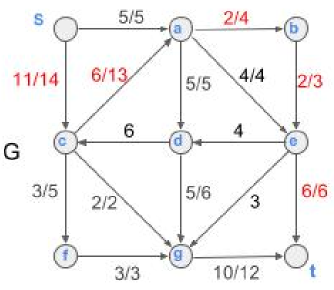


Take a 5-length path
$s \to c \to a \to b \to e \to t$

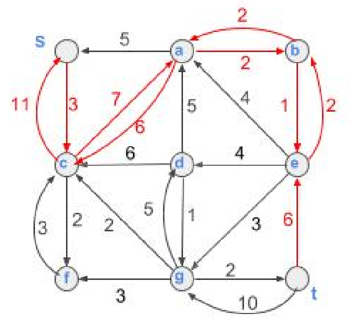
Residual graph: G_f



Original graph: G

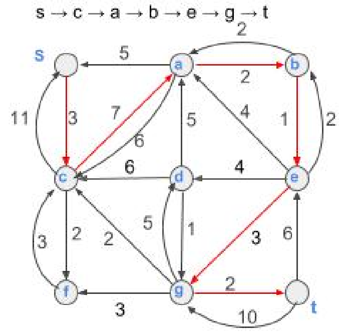


Update the residual graph

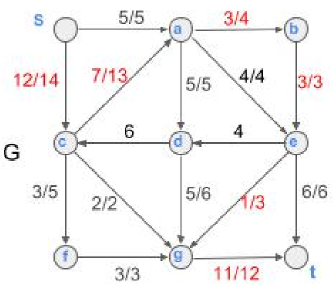


Take a 6-length path
$s \to c \to a \to b \to e \to g \to t$

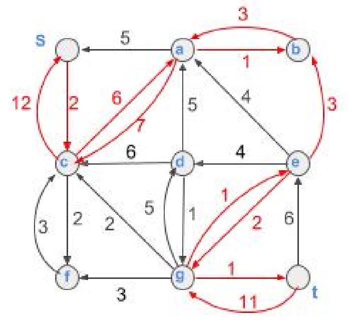
Residual graph: G_f



Original graph: G

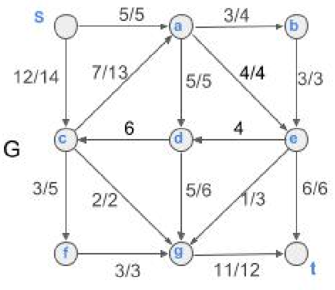


Update the residual graph



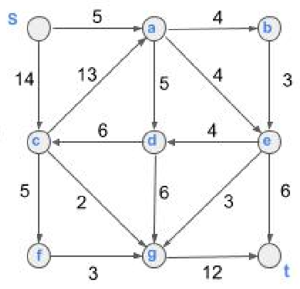
There are no paths from s to t anymore.
So the algorithm stops outputting the following flow with the value of 17:

Original graph: G



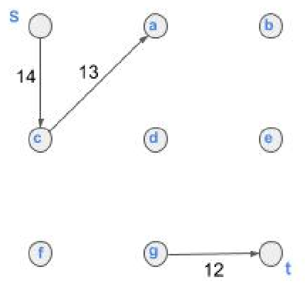
1(b) Use the maximum capacity augmenting path method (a.k.a. the capacity scaling algorithm). Since the max capacity is 14, the maximum degree of 2 smaller than 14 is 8 so start with initializing $\Delta = 8$

Original graph: G



Δ=8

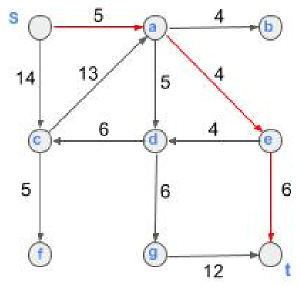
$G_f(8)$



No path from s to t, update Δ=8/2=4

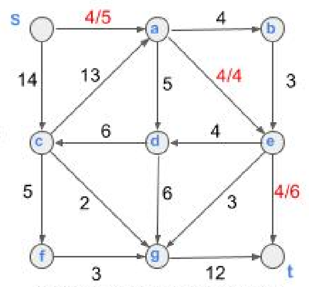
Δ=4

$G_f(4)$



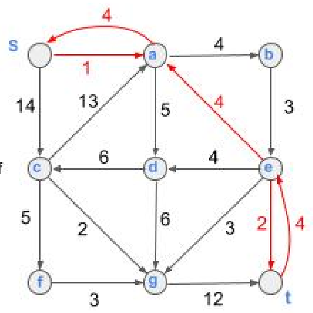
Take the path s→a→e→t

Original graph: G

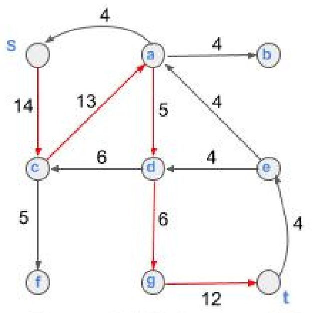


Update the flow in original graph

Residual graph: G_f

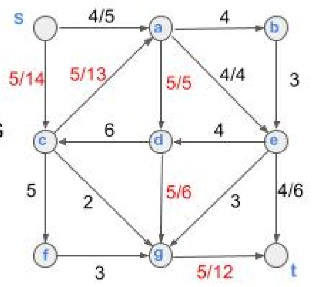


$G_f(4)$



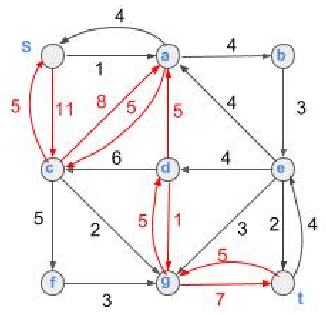
Construct $G_f(4)$ for the new residual graph,
Then take the path s→c→a→d→g→t

Original graph: G

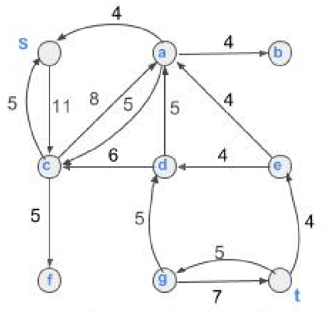


Update the flow in original graph

Residual graph: G_f

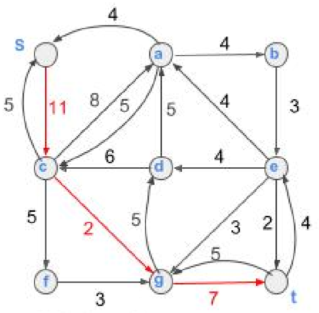


$G_f(4)$



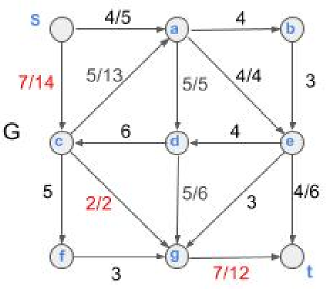
No path from s to t. Update Δ=4/2=2

$G_f(2)$



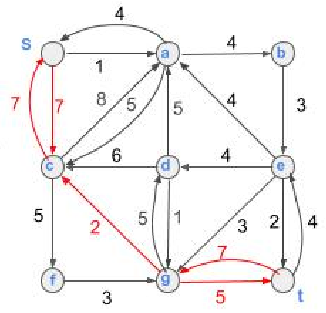
Take the path s→c→g→t

Original graph: G

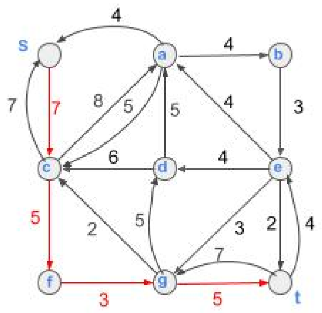


Update the flow in original graph

Residual graph: G_f

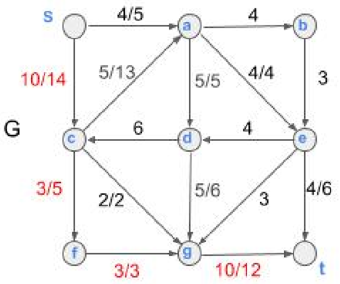


$G_f(2)$



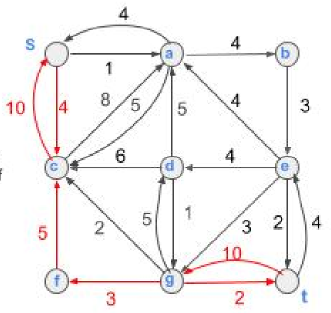
Take the path s→c→f→g→t

Original graph: G

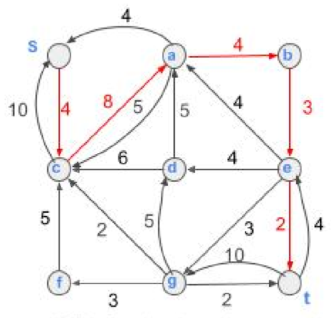


Update the flow in original graph

Residual graph: G_f

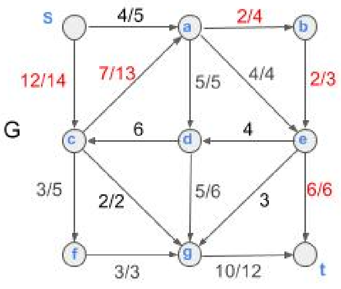


$G_f(2)$



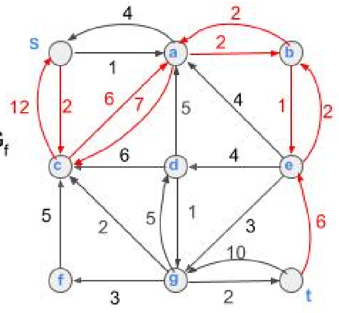
$G_f(2)=G_f$, take the path s→c→a→b→e→t

Original graph: G

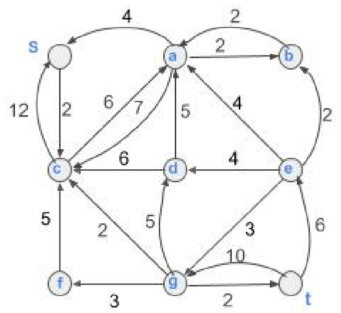


Update the flow in original graph

Residual graph: G_f

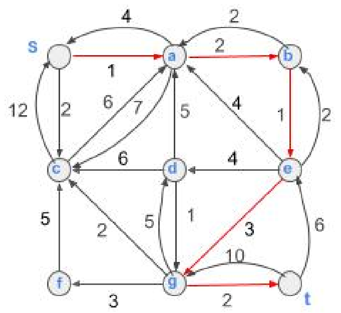


$G_f(2)$



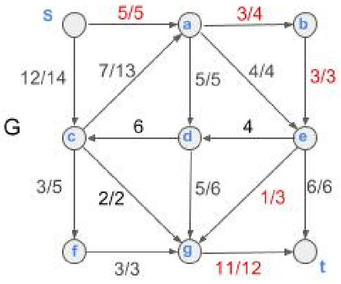
No path from s to t. Update Δ=2/2=1

$G_f(1)$



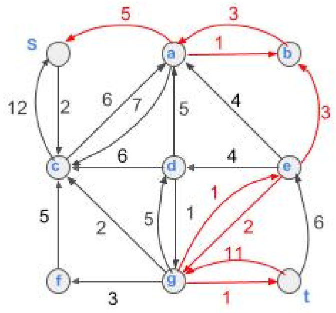
Take the path s→a→b→e→g→t

Original graph: G



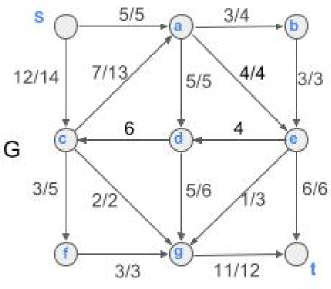
Update the flow in original graph

Residual graph: G_f = $G_f(1)$



There are no paths from s to t anymore.
So the algorithm stops outputting the following flow with the value of 17:

Original graph: G



1(c) The capacity of cut (A, B) with $A = \{s, a, b, c, f\}$ and $B = \{d, g, e, t\}$ which is the minimum cut of the graph is 17. Thus, by the min cut max flow theorem the maximum capacity of the graph is also 17 as shown in the parts a) and b).

2. (15 pts) Given an $N \times M$ matrix with a number of gold coins in each entry, start at the top right corner. Solve the following two problems in $O(N \cdot M)$ and $O\left((N \cdot M)^2\right)$ respectively using dynamic programming.

   (a) (10 pts) You have to reach the bottom left corner by moving only left and down. You collect all gold coins on your path. Find a path that will allow you to collect the maximum possible number of gold coins.

   (b) (5 pts) You have to reach the bottom left corner by moving only left and down. From there you need to go back up to the top right corner moving only right and up. You collect all gold coins on your path. Find a path that will allow you to collect the maximum possible total number of gold coins.

   (c) Bonus Question (5 pts): Design an algorithm to solve question (b) in time $O\left(\max\{N, M\} \cdot \min\{N, M\}^2\right)$.

Solution:

Bonus Question
First, the problem is equivalent to finding two paths starting from the top right corner, ending at the bottom left corner, moving only left and down, containing the greatest possible amount of gold coins. For paths A, B denote by $f(A, B)$ the amount of gold coins collected. Let V be the matrix, it's (i, j)-th entry denoted by $V[i, j]$. Moreover, note that for any such paths A, B, there exist paths A', B' which do not intersect except at the top right corner and bottom left corner and $f(A, B) \leq f(A', B')$, assuming the matrix is not a vector and entries of V are in $\mathbb{N}$. Hence there is an "upper" path U closer to the upper left corner and "lower" path L closer to the bottom right corner.
Let $M[i, j, k]$ be the maximum number of gold coins achieved by some U, L counting only the gold gathered in lines i and above, where U goes through $V[i, j]$ and L through $V[i, k]$. We only compute M for all i and pairs (j, k) where $j < k$. Clearly $M[1, j, k]$ can be computed in $O(M)$. Assume we have computed M up to row i. Consider row $i + 1$.

```
| $M[i+1, m-1, m] = M[i, m-1, m] + V[i+1, m-1] + V[i+1, m]$
| For $j = (m-2)$ to 1:
|     $M[i+1, j, m] = V[i+1, j] + max\{M[i, j, m], M[i+1, j-1, m]\}$
| For $k = m-1$ to 2:
|     For $j = m-2$ to 1:
|         $M[i+1, j, k] =$
|         $max\{M[i, j, k] + V[i+1, j] + V[i+1, k],$
|         $M[i+1, j-1, k] + V[i+1, j],$
|         $M[i+1, j, k-1] + V[i+1, k],$
|         $M[i+1, j-1, k-1] + V[i+1, j] + V[i+1, k]\}$
```

The order finding the values of new entries of M for a given row is important so that we don't make calls to cells of M that haven't been computed yet. For every entry in M we only need constant time. Thus the runtime is $O(M^2N)$. We could similarly fill M by columns, which would give runtime $O(N^2M)$.

To find the corresponding path, modify M slightly. Add extra variables d_1 and d_2 which will take values in {up, right}, indicating which direction the max path came from. Then once we computed all of M, we can start at the bottom left corner and backtrack using d_1 and d_2 in order to recover the paths. This does not change the time complexity.

3. (15 pts) McGill has hired you to write an algorithm to schedule the final exams. Each semester, McGill offers n different classes. There are r different rooms on campus. You are given two arrays $E[1 \ldots n]$ and $S[1 \ldots r]$, where $E[i]$ is the number of students enrolled in the i-th class, and $S[j]$ is the number of seats in the j-th room. At most one final exam can be held in each room. Class i can hold its final exam in room j only if $E[i] \leq S[j]$.

   Describe a polynomial time algorithm to assign a room to each class (or report correctly that no such assignment is possible). Prove the correctness of your algorithm and analyze its running time.

Solution: Construct the following bipartite graph $G = (A \cup B, E)$: $A = \{a_1, ..., a_n\}, B = \{b_1, ..., b_r\}$. For each i, j, $(a_i, b_j) \in E$ if and only if $E[i] \leq S[j]$. Then assigning a room to each class is equivalent to matching each vertex in A to a vertex in B. Thus the problem is equivalent to checking if the maximum matching in G has size n. This can be solved by the Ford-Fulkerson algorithm in polynomial time.

4. (15 pts) Suppose you have n computers, $I = \{1, 2, ..., n\}$. Now you can update the system of these computers. For computer i, you can get a benefit $b_i \geq 0$ by updating its system. However, there will be interaction problems between the old and new system. For each pair of computer i and j $(i < j)$, if one of them has an old system while the other one has a new system, there is an expense $x_{ij} \geq 0$. If you can update any computer, you can just update all of them so that you get $\sum_i b_i$ benefit without any expense.

   However, now there is a set of unchangeable computers $S \subseteq I$. You can not update the system of unchangeable computers. They have to use an old system. Your goal is to choose a subset of those changeable computers $I \backslash S$ to do system updating, so that the sum of benefits minus expenses is maximized.

   Design a polynomial-time algorithm to solve this problem. Prove the correctness of your algorithm and analyze its running time.

   Hint: use min-cut.

Solution:

The problem asks to find a set $M \subseteq I \backslash S$ to maximize $\sum_{i \in M} b_i - \sum_{i \in M, j \in I \backslash M} x_{ij}$. We have

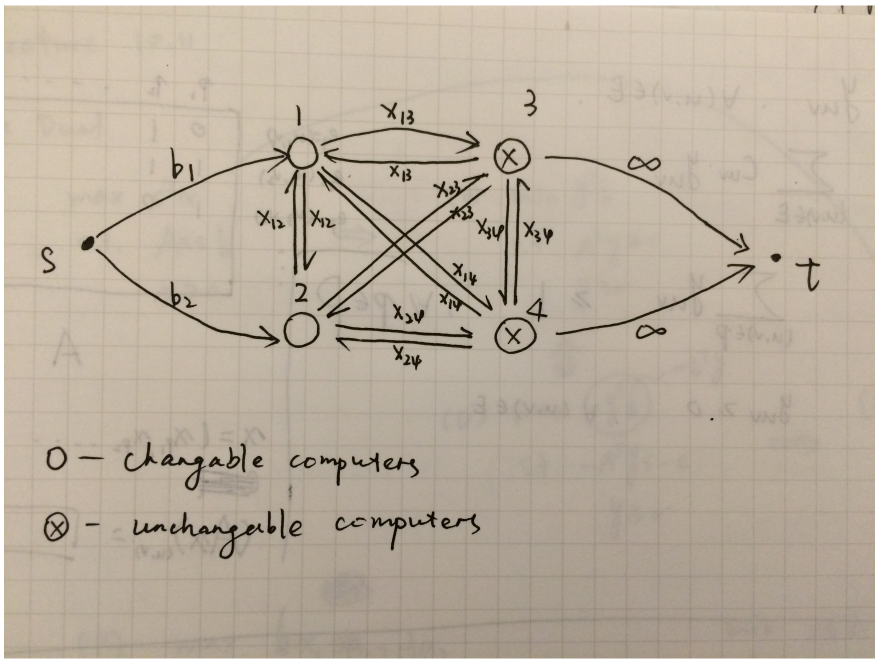
$$\sum_{i \in M} b_i - \sum_{i \in M, j \in I \backslash M} x_{ij} = -\left(\sum_{i \in M, j \in I \backslash M} x_{ij} + \sum_{i \in I \backslash (S \cup M)} b_i - \sum_{i \in I \backslash S} b_i \right)$$

Notice that $\sum_{i \in I \backslash S} b_i$ is a fixed value. Thus the problem is equivalent to

$$\min_{M \subseteq I \backslash S} \left(\sum_{i \in M, j \in I \backslash M} x_{ij} + \sum_{i \in I \backslash (S \cup M)} b_i \right)$$

Now consider the following network:

- For each computer i, add a vertex i to the network. For each pair (i, j), $i < j$, add two direct edges $(i, j), (j, i)$, both with capacity x_{ij}.
- Add a source s. Add edge (s, i) for each changeable computer $i \in I \backslash S$ with capacity b_i.
- Add a sink t. Add edge (i, t) for each unchangeable computer $i \in S$ with capacity ∞.



Let the minimum cut of this network be $(s \cup M, t \cup (I \backslash M))$. Firstly $M \subseteq I \backslash S$. That's because if there exists some $i \in M \cap S$, then edge (i, t) is in the cut with infinite capacity, contradiction. Secondly, the size of the cut is $\sum_{i \in M, j \in I \backslash M} x_{ij} + \sum_{i \in I \backslash (S \cup M)} b_i$. Thus the problem is equivalent to finding the minimum cut of the above network, which can be solved by the Ford-Fulkerson algorithm in polynomial time.