深 圳 大 学 实 验 报 告

课程名称：算法设计与分析

实验项目名称：实验六 最大流应用问题

学院：计算机与软件学院

专业：计算机科学与技术

指导教师：杨烜

报告人：沈晨珩 学号：2019092121 班级：19 计科国际

实验时间：2021.6.15

实验报告提交时间：2021.6.15

教务部制

实验六 最大流应用问题

一、实验目的：

（1）掌握最大流算法思想。
（2）学会用最大流算法求解应用问题。

二、内容：论文评审问题

1. 有m篇论文和n个评审，每篇论文需要安排a个评审，每个评审最多评b篇论文。请设计一个论文分配方案。
2. 要求应用最大流解决上述问题，画出m=10，n=3的流网络图并解释说明流网络图与论文评审问题的关系。
3. 编程实现所设计算法，计算a和b取不同值情况下的分配方案，如果没有可行方案则输出无解。

三、实验要求

1. 在blackboard提交电子版实验报告，注意实验报告的书写，整体排版。
2. 实验报告的实验步骤部分需详细给出算法思想与实现代码之间的关系解释，不可直接粘贴代码（直接粘贴代码者视为该部分内容缺失）。
3. 实验报告中要求证明该算法的关键定理，并说明这些定理所起的作用。
3. 实验报告样式可从http://192.168.2.3/guide.aspx 表格下载－学生适用－在校生管理－实践教学－实验：深圳大学学生实验报告）
4. 源代码作为实验报告附件上传。
5. 在实验课需要现场运行验证并讲解PPT。

四、实验过程及结果

一：实验结果正确性展示

对于数据a = 2，b = 8，m = 10，n = 3，进行测试
三个算法均正确。

```
I:\大二下\算法设计与分析\实验\实验六 最大流应用问题>cd "i:\大二下\算法设计与分析\实验
+ Dinic.cpp -o Dinic && "i:\大二下\算法设计与分析\实验\实验六 最大流应用问题\"Dinic
Max Flow = 20
时间为：3
```

```
I:\大二下\算法设计与分析\实验\实验六 最大流应用问题>cd "i:\大二下\算法设计与
+ FF.cpp -o FF && "i:\大二下\算法设计与分析\实验\实验六 最大流应用问题\"FF
Max Flow = 20
时间为：3
```

```
I:\大二下\算法设计与分析\实验\实验六 最大流应用问题>cd "i:\大二下\算法设计与
+ EK.cpp -o EK && "i:\大二下\算法设计与分析\实验\实验六 最大流应用问题\"EK
Max Flow = 20
时间为：3
```

二：模型建立

有m篇论文和n个评审，每篇论文需要安排a个评审，每个评审最多评b篇论文。

对于论文评审问题，总共需要评审的论文数为a*m，评委能够评审的论文数为b*n，因此若能够分配所有论文，需要满足的第一个条件就是a*$m \geq b*n$。

下面进行详细解读：

1. 第一层：源点

   该有向图的起点是源点，我们用节点0来表示

2. 第二层：论文

   源点连接着m篇论文，由于每篇论文需要被评审a次，因此源点->论文的容量值为a。源点到论文一共发出a*m的流量，表示所有论文需要被评审的总次数。

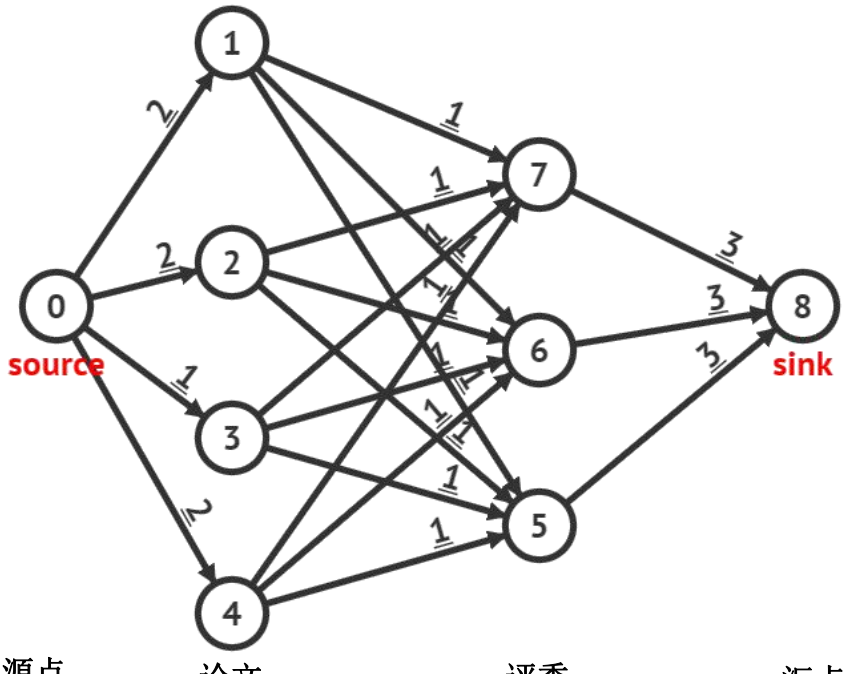
3. 第三层：评委

   对于每一篇论文，连线表示某一篇论文被某一位评审评阅了一次，因此从论文->评委的容量值为1。每篇论文发出1流量，总共有m*n条连线，表示总共有m*n种分配情况。

4. 第四层：汇点

   一位评委能够评阅b篇论文，因此评委->汇点的容量值为b。评委->汇点的总流量值为n*b。

获取最终网络流图后，若论文I流入评委J，则说明评委J评阅了论文I。



如果所有论文都能够被评审，需要满足两个条件：

1. 源点->论文的流量（最大流） ≤ 论文->评委的流量 （a * m ≤ m * n,即a≤n）
2. 源点->论文的流量（最大流） ≤ 评委->汇点的流量 （a * m ≤ b * n）

三：最大流计算——Ford-Fulkerson 增广路方法

该方法通过寻找增广路径来更新最大流，有EK,dinic等主流算法，主要区别在于对于增广路径的获取方法。

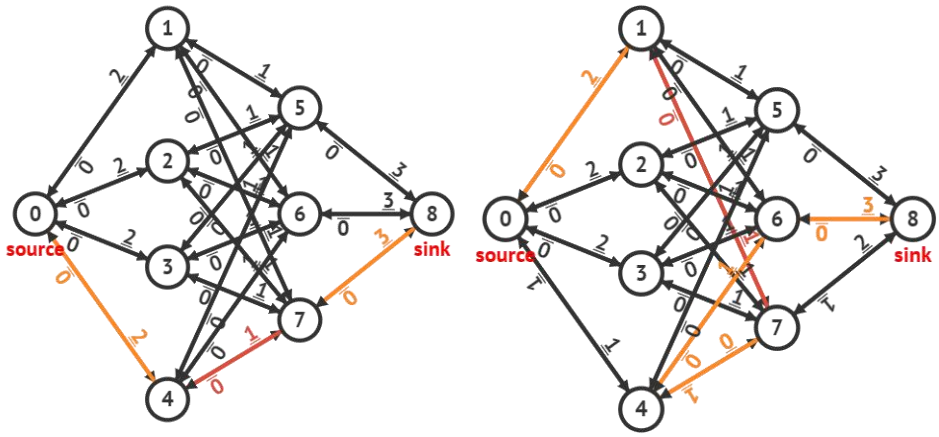
算法原理：

① 解决最大流问题的Ford-Fulkerson方法是一种迭代方法。开始时，对所有 $u, v \in V$，有 $f(u, v) = 0$，即初始状态时流的值为0。

② 在每次迭代中，可通过“增广路径”来增加流值。增广路径可以看作是从源点s到汇点t之间的一条路径。

③ 沿该路径可以压入更多的流，从而增加流的值。

④ 反复进行这一过程，直到增广路径都被找出为止。最大流最小割定理将说明在算法终止时，这一过程可产生出最大流。

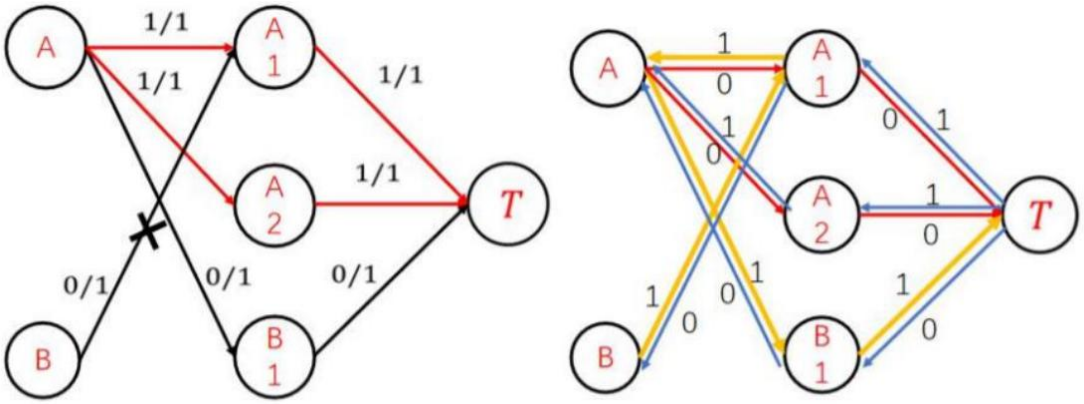


伪代码：

```
initMaxFlow
while there is an augmenting path
        find an augmenting path
                increase maxflow
return maxflow
```

由于增广路的搜索需要一定的顺序，不同的顺序会导致不同的结果。因此可能会因为搜索顺序不合适导致最大流求解错误。因此，可以引入残留网络，完成对错误搜索增广路产生的流量退回。在残留网络中对每一条边增加一条反向边，其容量与原边相同，流量为原边最大限制容量与原边已使用流量之差。

如下图，不引入残留网络时可能先找到两条通路 $(A, A1, T)$ 与 $(A, A2, T)$，这将导致流经后面B1 时已经流量满，而无法连接节点造成最大流求解错误。引入残留网络后就可以将 $(A1, T)$ 这条通路给B，最终将得到正确的最大流结果。



每一次增广的过程中，更新残留网络。正向边减去流量，反向边加上流量。
最大流最小割定理告诉我们：一个流是最大流，当且仅当它的残留网络中不包含增广路径。由此可以得到网络最大流。

伪代码：

```
initMaxFlow
while there is an augmenting path
    find an augmenting path
        for each edge u- > v in the path
            decrease capacity u- > v by bottleneck
            increase capacity v- > u by bottleneck
 increase maxflow by bottleneck
```

以下展示的三种算法均基于以上方法，区别在于对于增广路径的寻找方式。

三：最大流算法实现

1．DFS搜索 —— 基本的Ford-Fulkerson算法

(1) 算法原理

① 利用DFS算法，在图中搜索出一条增广路径。

② 基于这条增广路径，使用Ford-Fulkerson方法

(2) 伪代码

```
Ford - Fulkerson :
initMaxFlow
while there is an augmenting path
    find an augmenting path using DFS
        increase maxflow by bottleneck
return maxflow
```

DFS (v_1, maxflow):

if $v_1 = \text{end}$:

    return maxflow;

vertex[v] is visited

for each connected edge $v_1 \rightarrow v_2$ (unvisited & remain capacity) :

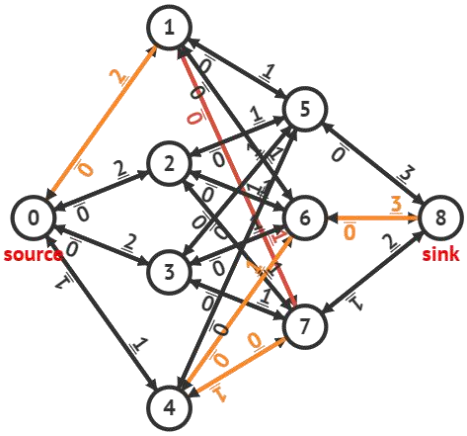
    bottleneck = DFS(v_2 , min(maxflow, remain capacity));

    decrease capacity $v_1 \rightarrow v_2$ by bottleneck

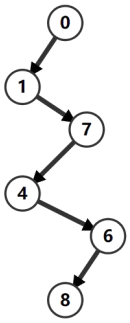
    increase capacity $v_2 \rightarrow v_1$ by bottleneck

    return bottleneck

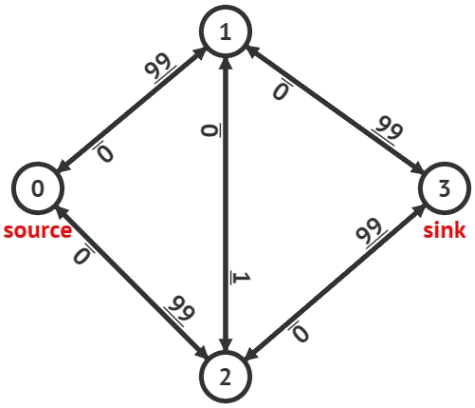
(3) 算法效率分析



以左图为例，DFS搜索树为下图，显然不是最短路径，可以进行优化。第二、三种算法对搜索路径长度进行了优化。



(4) 时间复杂度分析



在左图的特殊情况中，DFS过程为0-1-2-3，0-2-1-3，0-1-2-3，0-2-1-3.....完成最大流计算需要198次。

在DFS过程中，每一次迭代最大流至少增大1，因此最大流的求解迭代次数至多为f^*。

又每一轮DFS的复杂度为$O(E)$

因此最坏情况的时间复杂度为$O(E|f^*|)$

2．BFS搜索 —— Edmond-Karp算法

(1) 算法原理

① 利用BFS算法，在图中搜索出一条增广路径。

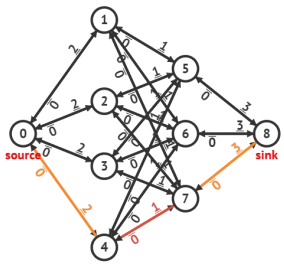
② 基于这条增广路径，使用Ford-Fulkerson方法

(2) 伪代码

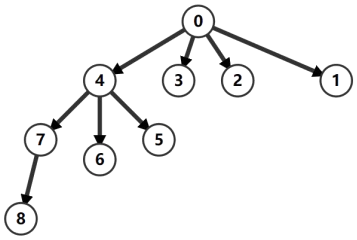
```
Edmonds_Karp:
initMaxFlow
while there is an augmenting path
    find an augmenting path using BFS
        bottleneck = BFS()
        decrease capacity by bottleneck
        increase capacity by bottleneck
return maxflow

BFS：
memset(pre, -1, sizeof(pre))
pre[start] = 0 // 记录前驱节点
flow[start] = 0  // 记录瓶颈流量
q.push(start)
while q is not empty
    v1 = q.front();
    q.pop();
    if v1 = end
        break;
    for each connected edge v1 → v2 (unvisited & remain capacity):
        pre[v2] = v1
        flow[v2] = min(flow[v1], capacity)
        q.push(v2)
if pre[end] == -1
    return -1; //判断是否连断是否连
return flow[end];
```

(3) 算法效率分析



以左图为例，BFS搜索树为下图，显然可以得到由源点到汇点的最短路径，是一条效率较高的增广路径。



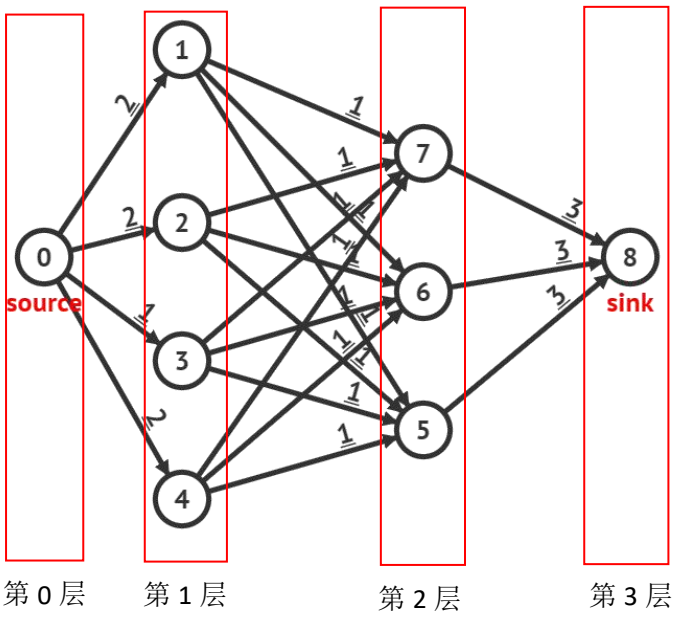
(4) 时间复杂度分析

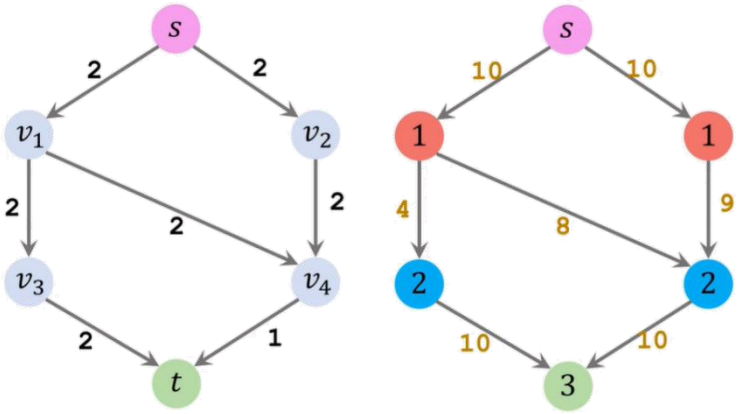
Edmonds and Karp在1972年，以及Dinic在1970年都独立的证明了如果每步增广路径都是最短的话，那么整个算法将会执行 $O(VE)$ 步。又每一轮DFS的复杂度为 $O(E)$。
因此最坏情况的时间复杂度为 $O(VE^2)$

3．层次搜索 —— Dinic算法

(1) 算法原理

① 对流网络采用BFS给每个节点标号，标号的内容为到源点的经过的最短边数，这样就把原图构建成了一个分层图，其中每条边均从第i层连向第i+1层。





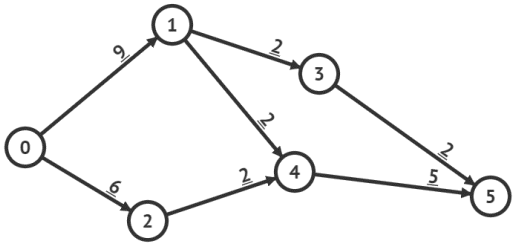
② 在分层图上进行DFS搜索，获取增广路径。

③ 基于这条增广路径，使用Ford-Fulkerson方法

(2) 算法优化

① 多路增广：每次找到一条增广路的时候，如果残余流量没有用完，可以利用残余部分流量，再找出一条增广路。这样就可以在一次BFS中找出多条增广路，大大提高了算法的效率。

② 当前弧优化：如果一条边已经被增广过，那么它就没有可能被增广第二次。那么，我们下一次进行增广的时候，就可以不必再走那些已经被增广过的边。



(3) 伪代码

```
Dinic :
initMaxFlow
while t is reachable from s in the residual graph
    find the level graph (BFS)
    for each blocking flow in the level graph (DFS)
        update the capacity in the blocking flow
        increase maxflow by bottleneck

BFS :
    memset(level, -1, sizeof(level));
    level[0] = 0;
    q.push(0);
    while q is not empty :
        v1 = q.front();
        q.pop();
      for each connected edge v1 → v2 (unvisited & remain capacity) :
            level[v2] = level[v1] + 1;
            q.push(v2);
```

DFS(v_1, maxflow):

if v_1 = end:

return maxflow

for each connected edge $v_1 \to v_2$ (unvisited & remain capacity & level[v_2] > level[v_1]):

bottleneck = DFS(v_2, min(maxflow, remain capacity));

decrease capacity $v_1 \to v_2$ by bottleneck

increase capacity $v_2 \to v_1$ by bottleneck

return bottleneckd

(4) 时间复杂度分析

Dinic算法分为两部分，分层以及增广。

每一次分层，需要执行一次BFS操作，时间复杂度为$O(V+E)$。

BFS之后还包含若干次增广。根据增广路的性质，每一次增广都至少会产生一个零边，所以增广次数上界为$O(E)$ 。每一次增广都会搜寻到一条从源点到汇点的路径，路径的长度上界为$O(V)$，所以每一次分层的时间复杂度为$O(VE)+O(V+E)=O(VE)$。

因为最多会进行$O(V)$次分层，所以总的时间上界为$O(V^2E)$。在本题题意下，经过测试，通常情况下分层次数在5次以下，因此时间复杂度为$O(VE)$。

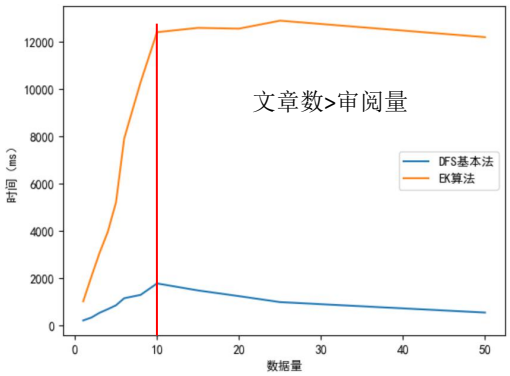
五：数据测试

由于DFS基本法 / EK 算法与Dinic算法差距较大，故分开测试。

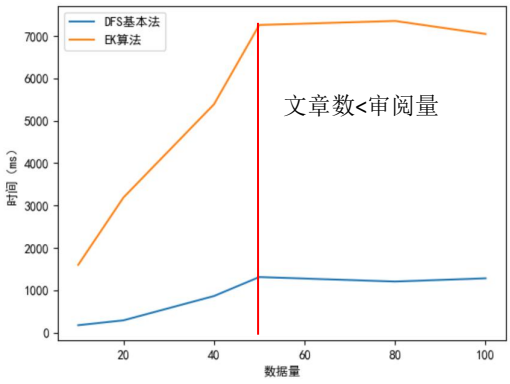
有m篇论文和n个评审，每篇论文需要安排a个评审，每个评审最多评b篇论文。

DFS基本法 / EK 算法:

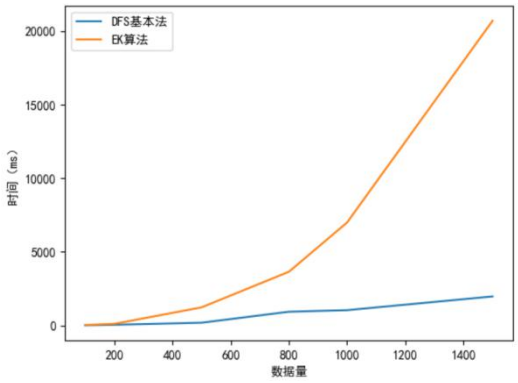
b=100, m=1000, n=100, a为变量



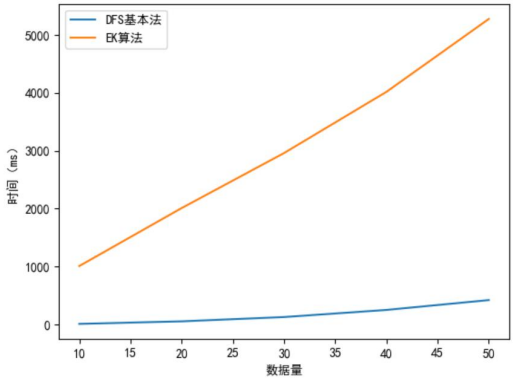
a=5, m=1000, n=100, b为变量



a=5, b=100, n=100, m为变量

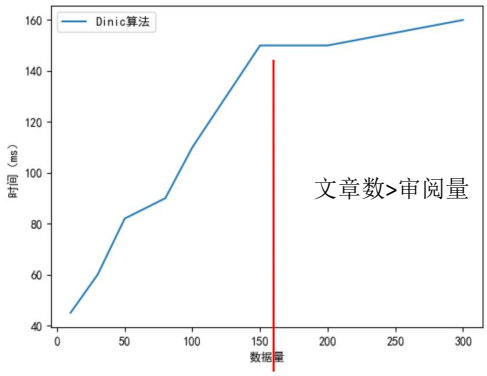


a=5, b=100, m=1000, n为变量

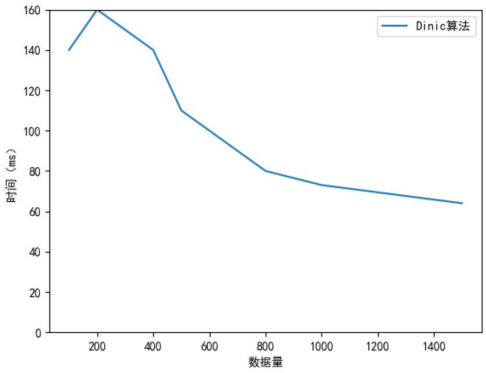


Dinic算法：

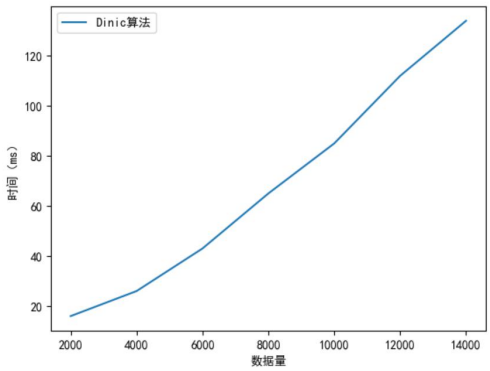
b=1000, m=10000, n=1000, a为变量



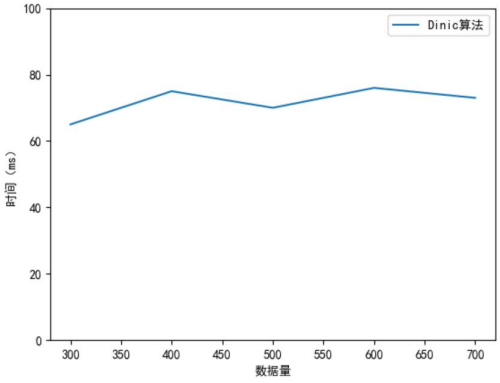
a=50, m=10000, n=1000, b为变量



a=50, b=1000, n=1000, m为变量



a=50, b=1000, m=10000, n为变量



五、经验总结

本次实验完成了图论中对于最大流的求解。三种算法都是基于同一种方法，但是对于时间效率却截然不同。可以看出不同实现方式对于算法效率有着很大的影响。