Python_ADS

python 二维数组初始化

参考博客:python3 初始二维数组

Python3中初始化一个多维数组，通过`for range`方法。以初始化二维数组举例：

```
arr = [[] for i in range(5)]
>>> [[], [], [], [], []]
arr = [[0, 0] for i in range(5)]
arr[2].append(2)
>>> [[0, 0], [0, 0], [0, 0, 2], [0, 0], [0, 0]]
12345
```

初始一个一维数组，可以使用*或者`for range`

```
arr1 = [None for i in range(5)]
>>> [None, None, None, None, None]
arr2 = [None]*5
>>> [None, None, None, None, None]
1234
```

但是用*初始化二维数组则会在修改数组内容时出现错误，例如：

```
arr = [[0, 0]]*5
arr[2] = 2
>>> [[0, 0], [0, 0], 2, [0, 0], [0, 0]] # 直接复制不会出现错误
arr[2].append(2)
>>> [[0, 0, 2], [0, 0, 2], [0, 0, 2], [0, 0, 2], [0, 0, 2]]
arr[2][1] = 5
>>> [[0, 5], [0, 5], [0, 5], [0, 5], [0, 5]]
1234567
```

而使用`for range`初始化不会产生该问题，range会另外开辟一个新的内存地址；*会指向同一个内存地址，改变值会其内存地址指向的值，从而改变所有的值。

堆排序

```
import heapq


class Test():
    def __init__(self, a, b):
```

```
        self.a = a
        self.b = b

    def __lt__(self, other):
        if self.a == other.a:
            return self.b < other.b
        else:
            return self.a < other.a

    def __str__(self):
        return str(self.a) + " " + str(self.b)

    def __repr__(self):
        return "(" + str(self.a) + " , " + str(self.b) + ")"


heap = []
heapq.heappush(heap, Test(1, 5))
heapq.heappush(heap, Test(1, 3))
heapq.heappush(heap, Test(2, 2))
heapq.heappush(heap, Test(2, 7))
heapq.heappush(heap, Test(2, 3))
heapq.heappush(heap, Test(4, 3))
heapq.heappush(heap, Test(10, 1))

while heap:
    print(heap)
```

堆区间第 k 大

```
import heapq

nums = [14, 20, 5, 28, 1, 21, 16, 22, 17, 28]
heapq.nlargest(3, nums)
# [28, 28, 22]
heapq.nsmallest(3, nums)
# [1, 5, 14]
```

堆实现优先队列

```
import heapq

class PriorityQueue:

  def __init__(self):
    self._queue = []
    self._index =0
```

```
  def push(self, item, priority):
    # 传入两个参数，一个是存放元素的数组，另一个是要存储的元素，这里是一个元组。
    # 由于heap内部默认有小到大排，所以对priority取负数
    heapq.heappush(self._queue, (-priority, self._index, item))
    self._index += 1

  def pop(self):
    return heapq.heappop(self._queue)[-1]
q = PriorityQueue()

q.push('lenovo', 1)
q.push('Mac', 5)
q.push('ThinkPad', 2)
q.push('Surface', 3)

q.pop()
# Mac
q.pop()
# Surface
```

线段树区间和_lazy

```
# 线段树的节点类
class TreeNode(object):
    def __init__(self):
        self.left = -1
        self.right = -1
        self.sum_num = 0
        self.lazy_tag = 0

    # 打印函数
    def __str__(self):
        return '[%s,%s,%s,%s]' % (self.left, self.right,
                                  self.sum_num, self.lazy_tag)

    # 打印函数
    def __repr__(self):
        return '[%s,%s,%s,%s]' % (self.left, self.right,
                                  self.sum_num, self.lazy_tag)


# 线段树类
# 以_开头的是递归实现
class Tree(object):
    def __init__(self, n, arr):
```

```
        self.n = n
        self.max_size = 4 * n
        self.tree = [TreeNode() for i in range(self.max_size)]  # 维护一个TreeNode数组
        self.arr = arr

    # index从1开始
    def _build(self, index, left, right):
        self.tree[index].left = left
        self.tree[index].right = right
        if left == right:
            self.tree[index].sum_num = self.arr[left - 1]
        else:
            mid = (left + right) // 2
            self._build(index * 2, left, mid)
            self._build(index * 2 + 1, mid + 1, right)
            self.pushup_sum(index)

    # 构建线段树
    def build(self):
        self._build(1, 1, self.n)

    def _update2(self, ql, qr, val, i, l, r, ):
        mid = (l + r) // 2
        if l >= ql and r <= qr:
            self.tree[i].sum_num += (r - l + 1) * val  # 更新和
            self.tree[i].lazy_tag += val  # 更新懒惰标记
        else:
            self.pushdown_sum(i)
            if mid >= ql:
                self._update2(ql, qr, val, i * 2, l, mid)
            if qr > mid:
                self._update2(ql, qr, val, i * 2 + 1, mid + 1, r)
            self.pushup_sum(i)

    # 区间修改
    def update2(self, ql, qr, val, ):
        self._update2(ql, qr, val, 1, 1, self.n)

    def _query2(self, ql, qr, i, l, r, ):
        if l >= ql and r <= qr:  # 若当前范围包含于要查询的范围
            return self.tree[i].sum_num
        else:
            self.pushdown_sum(i)  # modify
            mid = (l + r) // 2
            res_l = 0
            res_r = 0
            if ql <= mid:  # 左子树最大的值大于了查询范围最小的值-->左子树和需要查询的区间交集非空
```

```
                res_l = self._query2(ql, qr, i * 2, l, mid, )
            if qr > mid:  # 右子树最小的值小于了查询范围最大的值-->右子树和需要查询的区间交集非空
                res_r = self._query2(ql, qr, i * 2 + 1, mid + 1, r, )
            return res_l + res_r

    def query2(self, ql, qr):
        return self._query2(ql, qr, 1, 1, self.n)

    # 求和,向上更新
    def pushup_sum(self, k):
        self.tree[k].sum_num = self.tree[k * 2].sum_num + self.tree[k * 2 + 1].sum_num

    # 向下更新lazy_tag
    def pushdown_sum(self, i):
        lazy_tag = self.tree[i].lazy_tag
        if lazy_tag != 0:  # 如果有lazy_tag
            self.tree[i * 2].lazy_tag += lazy_tag  # 左子树加上lazy_tag
            self.tree[i * 2].sum_num += (self.tree[i * 2].right - self.tree[i * 2].left + 1) * lazy_tag  # 左子树更新和
            self.tree[i * 2 + 1].lazy_tag += lazy_tag  # 右子树加上lazy_tag
            self.tree[i * 2 + 1].sum_num += (self.tree[i * 2 + 1].right - self.tree[
                i * 2 + 1].left + 1) * lazy_tag  # 右子树更新和
            self.tree[i].lazy_tag = 0  # 将lazy_tag 归0

    # 深度遍历
    def _show_arr(self, i):
        if self.tree[i].left == self.tree[i].right and self.tree[i].left != -1:
            print(self.tree[i].sum_num, end=" ")
        if 2 * i < len(self.tree):
            self._show_arr(i * 2)
            self._show_arr(i * 2 + 1)

    # 显示更新后的数组的样子
    def show_arr(self, ):
        self._show_arr(1)

    def __str__(self):
        return str(self.tree)

# 落谷测试用例1
def test():
    n = 5  # 1 5 4 2 3
    arr = [1, 5, 4, 2, 3]
    tree = Tree(n, arr)
```

```
    tree.build()
    tree.update2(2, 4, 2)
    # # print(tree)
    res = tree.query2(3, 3)
    # print(tree)
    print(res)
    tree.update2(1, 5, -1)
    tree.update2(3, 5, 7)
    res = tree.query2(4, 4)
    print(res)


if __name__ == '__main__':
    # 样例输出
    line1 = [int(x) for x in input().strip().split(" ")]
    n = line1[0]  # 数字的个数
    m = line1[1]  # 操作的个数
    arr = [int(x) for x in input().strip().split(" ")]
    tree = Tree(n, arr)
    tree.build()
    for i in range(m):
        line = [int(x) for x in input().split(" ")]
        op = line[0]
        if op == 1:
            tree.update2(line[1], line[2], line[3])#区间更新
        elif op == 2:
            res = tree.query2(line[1], line[2])#区间查询
            print(res)


```

线段树区间最大_lazy

> 给定一个非负整数数组，你最初位于数组的第一个位置。
>
> 数组中的每个元素代表你在该位置可以跳跃的最大长度。
>
> 你的目标是使用最少的跳跃次数到达数组的最后一个位置。
>
> 示例:
>
> > 输入: [2,3,1,1,4] 输出: 2 解释: 跳到最后一个位置的最小跳跃数是 2。 从下标为 0 跳到下标为 1 的位置，跳 1 步，然后跳 3 步到达数组的最后一个位置。 说明:
>
> 假设你总是可以到达数组的最后一个位置。

```
#include <iostream>
#include <cstdio>
#include <cstring>
#include <algorithm>
```

```cpp

using namespace std;
typedef long long ll;
const int maxn = 200005;
int n, m;

struct node{
    int l, r;
    int MAX_VALUE; //表示区间最大值
    int mid(){
        return (l + r) >> 1;
    }
};
node tree[maxn * 4];
int value[maxn];

void init(int root, int l, int r){
    tree[root].l = l;
    tree[root].r = r;

    if(l == r){
         tree[root].MAX_VALUE = value[l];
         return;
    }

    int m = (l + r) >> 1;

    init(root<<1, l, m);
    init(root<<1|1, m+1, r);

    tree[root].MAX_VALUE = max(tree[root<<1].MAX_VALUE,
tree[root<<1|1].MAX_VALUE);
}

void update(int root, int idx, int v){
    if(tree[root].l == tree[root].r){
        tree[root].MAX_VALUE = v;
        return;
    }

    if(idx <= tree[root].mid()) update(root<<1, idx, v);
    else update(root<<1|1, idx, v);

    tree[root].MAX_VALUE = max(tree[root<<1].MAX_VALUE,
tree[root<<1|1].MAX_VALUE);
}

int query(int root, int l, int r){

```

```
    if(l == tree[root].l && r == tree[root].r) return tree[root].MAX_VALUE;

    int m = tree[root].mid();

    if(l > m) return query(root<<1|1, l, r);
    else if(r <= m) return query(root<<1, l, r);
    else return max(query(root<<1, l, m), query(root<<1|1, m+1, r));
}
```

树状数组_区间和

```python
class NumArray:
    def __init__(self, nums: List[int]):
        ''' 初始化sum 数组, 从一计数, 0 号不用
        '''
        self.tree = [0 for _ in range(len(nums)+1)]     # 从第一个数计算下标和
        for k in range(1, len(self.tree)):
            self.tree[k] = sum(nums[k-(k&-k):k])  # 原来的nums从0计数, tree从1计数

    def update(self, i: int, val: int) -> None:
        diff = val - self.sumRange(i,i)  # 计算更新的值和原数的差值, i,j从0 计数
        k = i+1
        while k<=len(self.tree)-1:
            self.tree[k] += diff
            k += k&-k

    def sumRange(self, i: int, j: int) -> int:
        # i, j 是从 0计数
        if i+1 == 1:
            return self.sum1k(j+1)
        else:
            return self.sum1k(j+1) - self.sum1k(i)

    def sum1k(self, k: int) -> int:
        res = 0
        while k>=1:
            res += self.tree[k]
            k -= k&-k
        return res
```

主席树区间第k小/大

```python
import bisect
import copy


class TreeNode(object):
    def __init__(self):
        self.left_node = None
        self.right_node = None
        self.num = 0
        self.l = -1
        self.r = -1

    # 打印函数
    def __str__(self):
        # return '[%s,%s,] num:%s, %s' % (self.l, self.r, self.num, id(self)) # 查看地址,确实新建了部分节点
        return '[%s,%s,] num:%s,' % (self.l, self.r, self.num)

    # 打印当前树形结构
    def _show_arr(self, node, ):
        print(node)
        if node.l == node.r:
            return
        else:
            self._show_arr(node.left_node)
            self._show_arr(node.right_node)

    def show_arr(self, ):
        self._show_arr(self)

    # 打印区间求差之后的树形结构
    def show_diff(self, node2):
        self._show_diff(self, node2)

    def _show_diff(self, node, node2):
        print(node.l, node.r, node.num - node2.num)
        if node.l == node.r:
            return
        else:
            self._show_diff(node.left_node, node2.left_node)
            self._show_diff(node.right_node, node2.right_node)


# sum数组：记录节点权值
# p：记录离散化后序列长度，也是线段树的区间最大长度

```

```python
# 递归建一棵空树
def build(l, r):
    node = TreeNode()
    node.l = l
    node.r = r
    if l == r:
        return node
    else:
        m = (l + r) >> 1
        node_left = build(l, m)
        node_right = build(m + 1, r)
        node.left_node = node_left
        node.right_node = node_right
        return node


def insert(x, node: TreeNode):
    node.num += 1
    if node.l == node.r:  # 已经到了子节点了
        return
    m = (node.l + node.r) >> 1
    if m >= x:  # 左子树的最大值大于了该值,搜索左子树
        left_node = copy.copy(node.left_node)  # 复制一份节点
        node.left_node = left_node
        insert(x, node.left_node)
    if m < x:  # 右子树的最小值小于该值
        right_node = copy.copy(node.right_node)  # 复制一份节点
        node.right_node = right_node
        insert(x, node.right_node)


def find_k(nl: TreeNode, nr: TreeNode, k):
    if nr.l == nr.r:
        return nr.l
    left_num_diff = nr.left_node.num - nl.left_node.num
    if k <= left_num_diff:
        return find_k(nl.left_node, nr.left_node, k)
    else:
        return find_k(nl.right_node, nr.right_node, k - left_num_diff)


# 落谷用例
def test():
    arr = [25957, 6405, 15770, 26287, 26465, ]
    arr2 = sorted(arr)  # 排序 [6405, 15770,25957 ,26287,26465]
    z = list(map(lambda x: bisect.bisect(arr2, x), arr))  # [3, 1, 2, 4,
5]
    n = build(1, len(z))
    rt = []
```

```
    rt.append(n)
    for x in z:
        n2 = copy.copy(rt[-1])  # 复制最后一个版本的树
        insert(x, n2)  # 将值添加进去
        rt.append(n2)
        # n2.show_arr()
        # print()
    # 2 2 1
    res = find_k(rt[1], rt[2], 1)
    print(res)
    print(arr2[res - 1])
    # 1 2 2
    res = find_k(rt[0], rt[2], 2)
    print(res)
    print(arr2[res - 1])
    # 4 4 1
    res = find_k(rt[3], rt[5], 1)
    print(res)
    print(arr2[res-1])


if __name__ == '__main__':
    # test()
    line1 = [int(x) for x in input().strip().split(" ")]
    n = line1[0]  # 数字的个数
    m = line1[1]  # 查询的个数
    arr = [int(x) for x in input().strip().split(" ")]
    # 离散化
    arr2 = sorted(arr)  # 排序
    z = list(map(lambda x: bisect.bisect(arr2, x), arr))

    rt = build(1, len(z))
    rt_arr = [rt]
    for x in z:
        rt_temp = copy.copy(rt_arr[-1])  # 复制最后一个版本的树
        insert(x, rt_temp)  # 将值添加进去
        rt_arr.append(rt_temp)
    for i in range(m):
        line = [int(x) for x in input().split(" ")]
        res = find_k(rt_arr[line[0] - 1], rt_arr[line[1]], line[2])
        print(arr2[res-1])


```

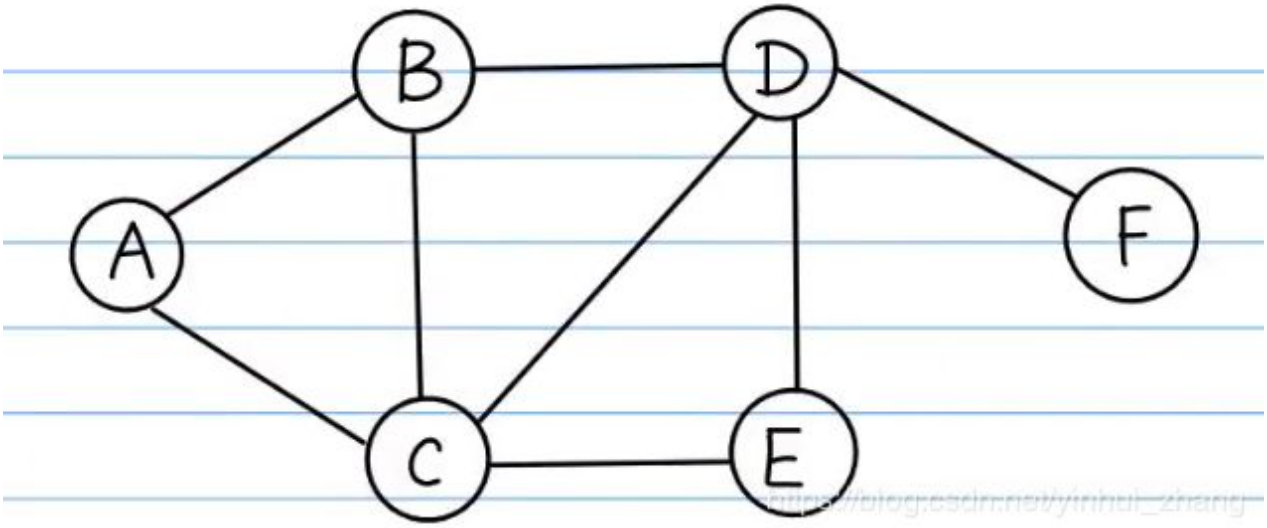
最长公共子串

```
def lcs(x,y):
    d = [0] * (len(x) + 1)
```

```
    for i in range(0,len(d)):
        d[i] = [0] * (len(y) + 1)

    for i in range(1,len(x) + 1):
        for j in range(1, len(y) + 1):
            if x[i-1] == y[j-1]:
                d[i][j] = d[i-1][j-1] + 1
            else:
                d[i][j] = max(d[i-1][j],d[i][j-1])
    print d

def lcs_extend(x,y):
    d = [0] * (len(x) + 1)
    p = [0] * (len(x) + 1)
    for i in range(0,len(d)):
        d[i] = [0] * (len(y) + 1)
        p[i] = [0] * (len(y) + 1)

    for i in range(1,len(x) + 1):
        for j in range(1, len(y) + 1):
            if x[i-1] == y[j-1]:
                d[i][j] = d[i-1][j-1] + 1
                p[i][j] = 1
            elif d[i-1][j] > d[i][j-1]:
                d[i][j] = d[i-1][j]
                p[i][j] = 2
            else:
                d[i][j] = d[i][j-1]
                p[i][j] = 3
    print d
    print p
    lcs_print(x,y,len(x),len(y),p)

def lcs_print(x,y,lenX,lenY,p):
    if lenX == 0 or lenY == 0:
        return
    if p[lenX][lenY] == 1:
        lcs_print(x,y,lenX-1,lenY-1,p)
        print x[lenX-1],
    elif p[lenX][lenY] == 2:
        lcs_print(x,y,lenX-1,lenY,p)
    else:
        lcs_print(x,y,lenX,lenY-1,p)

x = 'abcdf'
y = 'facefff'
lcs_extend(x,y)
```

Python_Graph

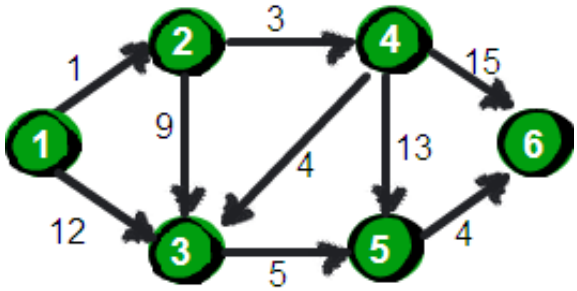
DFS、BFS




```
graph = {
        'a' : ['b', 'c'],
        'b' : ['a', 'c', 'd'],
        'c' : ['a','b', 'd','e'],
        'd' : ['b' , 'c', 'e', 'f'],
        'e' : ['c', 'd'],
        'f' : ['d']
        }


def BFS(graph, s):
    queue = []
    queue.append(s)
    seen = set()
    seen.add(s)
    while len(queue) > 0:
        vertex = queue.pop(0)
        nodes = graph[vertex]
        for node in nodes:
            if node not in seen:
                queue.append(node)
                seen.add(node)
        print(vertex)

BFS(graph, 'a')


def DFS(graph, s):
    stack = []
```

```
    stack.append(s)
    seen = set()
    seen.add(s)
    while len(stack) > 0:
        vertex = stack.pop()
        nodes  = graph[vertex]
        for node in nodes:
            if node not in seen:
                stack.append(node)
                seen.add(node)
        print(vertex)
DFS(graph, 'a')


def DFS1(graph, s, queue=[]):
    queue.append(s)
    for i in graph[s]:
        if i not in queue:
            DFS1(graph, i, queue)
    return queue
print(DFS1(graph, 'a'))
```

Dijkstra、Floyd算法




```
inf = float('inf')
matrix_distance = [[0,1,12,inf,inf,inf],
                   [inf,0,9,3,inf,inf],
                   [inf,inf,0,inf,5,inf],
                   [inf,inf,4,0,13,15],
                   [inf,inf,inf,inf,0,4],
                   [inf,inf,inf,inf,inf,0]]

def dijkstra(matrix_distance, source_node):
    inf = float('inf')
    # init the source node distance to others
    dis = matrix_distance[source_node]
    node_nums = len(dis)

    flag = [0 for i in range(node_nums)]
    flag[source_node] = 1
```

```
17
18      for i in range(node_nums-1):
19          min = inf
20          #find the min node from the source node
21          for j in range(node_nums):
22              if flag[j] == 0 and dis[j] < min:
23                  min = dis[j]
24                  u = j
25          flag[u] = 1
26          #update the dis
27          for v in range(node_nums):
28              if flag[v] == 0 and matrix_distance[u][v] < inf:
29                  if dis[v] > dis[u] + matrix_distance[u][v]:
30                      dis[v] = dis[u] + matrix_distance[u][v]
31
32      return dis
33
34  print(dijkstra(matrix_distance, 0))
35
36
37  def Floyd(dis):
38      #min (Dis(i,j) , Dis(i,k) + Dis(k,j) )
39      nums_vertex = len(dis[0])
40      for k in range(nums_vertex):
41          for i in range(nums_vertex):
42              for j in range(nums_vertex):
43                  if dis[i][j] > dis[i][k] + dis[k][j]:
44                      dis[i][j] = dis[i][k] + dis[k][j]
45      return dis
46  print(Floyd(matrix_distance))
```

Prim、Kruskal算法

```
1   """
2   代码来源:
    https://github.com/qiwsir/algorithm/blob/master/kruskal_algorithm.md
3           https://github.com/qiwsir/algorithm/blob/master/prim_algorithm.md
4   做了几个细节的小改动
5   """
6
7
8   from collections import defaultdict
9   from heapq import *
10
11  def Prim(vertexs, edges, start_node):
12      adjacent_vertex = defaultdict(list)
13      for v1, v2, length in edges:
```

```
        adjacent_vertex[v1].append((length, v1, v2))
        adjacent_vertex[v2].append((length, v2, v1))

    mst = []
    closed = set(start_node)

    adjacent_vertexs_edges = adjacent_vertex[start_node]
    heapify(adjacent_vertexs_edges)

    while adjacent_vertexs_edges:
        w, v1, v2 = heappop(adjacent_vertexs_edges)
        if v2 not in closed:
            closed.add(v2)
            mst.append((v1, v2, w))

            for next_vertex in adjacent_vertex[v2]:
                if next_vertex[2] not in closed:
                    heappush(adjacent_vertexs_edges, next_vertex)

    return mst


vertexs = list("ABCDEFG")
edges = [ ("A", "B", 7), ("A", "D", 5),
          ("B", "C", 8), ("B", "D", 9),
          ("B", "E", 7), ("C", "E", 5),
          ("D", "E", 15), ("D", "F", 6),
          ("E", "F", 8), ("E", "G", 9),
          ("F", "G", 11)]

print('prim:', Prim(vertexs, edges, 'A'))

#*************************************************


node = dict()
rank = dict()

def make_set(point):
    node[point] = point
    rank[point] = 0

def find(point):
    if node[point] != point:
        node[point] = find(node[point])
    return node[point]

def merge(point1, point2):
    root1 = find(point1)
```

```
    root2 = find(point2)
    if root1 != root2:
        if rank[root1] > rank[root2]:
            node[root2] = root1
        else:
            node[root1] = root2
            if rank[root1] == rank[root2] : rank[root2] += 1


def Kruskal(graph):
    for vertice in graph['vertices']:
        make_set(vertice)

    mst = set()

    edges = list(graph['edges'])
    edges.sort()
    for edge in edges:
        weight, v1, v2 = edge
        if find(v1) != find(v2):
            merge(v1 , v2)
            mst.add(edge)
    return mst

graph = {
    'vertices': ['A', 'B', 'C', 'D'],
    'edges': set([
        (1, 'A', 'B'),
        (5, 'A', 'C'),
        (3, 'A', 'D'),
        (4, 'B', 'C'),
        (2, 'B', 'D'),
        (1, 'C', 'D'),
        ])
    }

print(Kruskal(graph))
```

最大流Push-relabel

```
class Arc(object):
    def __init__(self):
        self.src = -1
        self.dst = -1
        self.cap = -1


s, t = -1, -1
```

```
with open('sample.dimacs') as f:
    for line in f.readlines():
        line = line.strip()
        if line.startswith('p'):
            tokens = line.split(' ')
            nodeNum = int(tokens[2])
            edgeNum = tokens[3]
        if line.startswith('n'):
            tokens = line.split(' ')
            if tokens[2] == 's':
                s = int(tokens[1])
            if tokens[2] == 't':
                t = int(tokens[1])
        if line.startswith('a'):
            tokens = line.split(' ')
            arc = Arc()
            arc.src = int(tokens[1])
            arc.dst = int(tokens[2])
            arc.cap = int(tokens[3])
            arcs.append(arc)

nodes = [-1] * nodeNum
for i in range(s, t + 1):
    nodes[i - s] = i
adjacent_matrix = [[0 for i in range(nodeNum)] for j in range(nodeNum)]
forward_matrix = [[0 for i in range(nodeNum)] for j in range(nodeNum)]
for arc in arcs:
    adjacent_matrix[arc.src - s][arc.dst - s] = arc.cap
    forward_matrix[arc.src - s][arc.dst - s] = arc.cap
flow_matrix = [[0 for i in range(nodeNum)] for j in range(nodeNum)]

height = [0] * nodeNum
height[0] = nodeNum
for i in range(len(adjacent_matrix)):
    flow_matrix[0][i] = adjacent_matrix[0][i]
    adjacent_matrix[0][i] = 0
    adjacent_matrix[i][0] = flow_matrix[0][i]


def excess(v):
    in_flow, out_flow = 0, 0
    for i in range(len(flow_matrix)):
        in_flow += flow_matrix[i][v]
        out_flow += flow_matrix[v][i]
    return in_flow - out_flow


def exist_excess():
    for v in range(len(flow_matrix)):
```

```
        if excess(v) > 0 and v != t - s:
            return v
    return None


v = exist_excess()
while v:
    has_lower_height = False
    for j in range(len(adjacent_matrix)):
        if adjacent_matrix[v][j] != 0 and height[v] > height[j]:
            has_lower_height = True
            if forward_matrix[v][j] != 0:
                bottleneck = min([excess(v), adjacent_matrix[v][j]])
                flow_matrix[v][j] += bottleneck
                adjacent_matrix[v][j] -= bottleneck
                adjacent_matrix[j][v] += bottleneck
            else:
                bottleneck = min([excess(v), flow_matrix[j][v]])
                flow_matrix[j][v] -= bottleneck
                adjacent_matrix[v][j] -= bottleneck
                adjacent_matrix[j][v] += bottleneck
    if not has_lower_height:
        height[v] += 1
    v = exist_excess()
for arc in arcs:
    print 'f %d %d %d' % (arc.src, arc.dst, flow_matrix[arc.src - s]
[arc.dst - s])
```