算法设计与分析第四次作业 - 网络流及其应用

201628013229058 洪鑫

1 Load balance

问题描述

> 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. Give the number of jobs and two computer ID for each job. You task is to minimize the max load.
> (hint: binary search)

算法描述

可以构造一个网络来解决这个问题。

构建一个四层有向图：第一层只有一个起点，第二层的每个节点表示一个任务，第三层的每个节点表示一台计算机，第四层为一个汇点。所有边都是从上一层指向下一层，具体连接情况如下：

起点与第二层的每个节点都有一条边相连，并且这条边的权重为1，限制一个任务只能完成一次。第二层每个任务节点与其相对应的第三层的计算机节点相连，边的权重也为1。第三层所有节点都和汇点相连。

我们的任务是找到第三层到汇点所有边可满足的最小权值，使得所有的任务都可以被执行，也即网络的最大流为任务数。通过在0~n之间使用二分查找的办法，可以找到可满足的最小权值。

伪代码

n为任务数，list有n行，每行有两个元素，分别表示第i个任务可执行的两个计算机编号。max_flow为求得的最大流量。

```
LoadBalance(n, list):
V={s,t}
E={}
G = <V,E>
for i = 1 to n
```

```
    c0 = list[i][0]
    c1 = list[i][1]
    V = V U {v2i, v3c0, v3c1}
    E = E U {(s,v2i,1),(v2i,v3c0,1),(v2i,v3c1,1),(v3c0,t,k),(v3c1,t,k)}
end
x = 0
y = n
while True:
    k = (x+y)/2
    max_flow = Ford-Fulkerson(G, s, t)
    if max_flow == n
        y = k
    else if y == x+1
        return y
    else
        x = k
end
```

正确性证明

分两部分证明:

1.算法构造的网络满足问题对任务和计算机的约束。
当第三层节点（计算机）到汇点边的权值足够大时，该网络的最大流一定满足：第二层每个节点（任务）的入流量和出流量都为1。也即满足：所有任务都被可选的两台计算机中的一台执行了。

2.二分查找后得到的返回值是最小权值： 首先，该网络的最大流满足：第三层所有节点的入流量等于出流量，而根据第二层节点（任务）出流量的含义，可以得出第三层节点（计算机）出流量即为该计算机执行的任务总数。
其次，第三层节点到汇点边的权值统一赋值k，k即为选定的所有计算机最大负载量，因为该网络的最大流一定满足：第三层节点到汇点的流量小于k。 最后，折半查找得到的返回值y一定是最小的计算机最大负载量。因为如果不是，那么存在一个最大负载量$y' < y$使得 `max_flow = Ford-Fulkerson(G, s, t)` 。但是，算法返回的y满足:当$k = y - 1$， `max_flow` 不等于 `Ford-Fulkerson(G, s, t)` ，相互矛盾，所以返回的是最小的计算机最大负载量。

算法复杂度

算法的复杂度为$O(n^2 log_2 n)$，其中n为任务数。

构造网络的复杂度为$O(n)$。Ford-Fulkerson算法的复杂度为$O(mC)$，其中m为边数，C为起点的最大出流量，在该问题中，$E \leq 5n$，$C \leq n$，在本问题中运用Ford-Fulkerson算法的复杂度为$O(n^2)$。折半查找的复杂度为$O(log_2 n)$，所以总查找复杂度为$O(n^2 log_2 n)$。最后的算法复杂度为$O(n + n^2 log_2 n)$，也即$O(n^2 log_2 n)$

2 Matrix

问题描述

> 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 satisfys the conditions.

算法描述

可以构造一个网络来解决这个问题。

构建一个五层有向图，第一层为一个起点，第二层节点数等于矩阵的行数，第三层节点数等于矩阵的元素数，第四层节点数等于矩阵的列数，第五层为一个汇点。

起点和所有的第二层节点分别相连，边的权值为对应的行和。第二层节点的每个节点和其对应的第三层节点相连（行元素），比如第二层第一个节点和第三层中所有对应矩阵第一行元素的节点相连，权值都为1。同样的第三层某一列的所有元素和对应的第四层中的列节点相连，边的权值为1。第四层所有节点和汇点相连，边的权值为列节点所对应矩阵列的列和。

求该网络的最大流，通过最大流中第二层到第三层节点边的值或第三层到第四层节点边的值即可得到一个解，应为节点的入流量等于出流量，两种方式均可。值为1的表示该节点对应的矩阵元素为1。

伪代码

b[i]为矩阵A第i行的行和，c[j]为矩阵A第j列的列和。max_flow为求得的最大流，max_flow(e)表示边e的流量。

```
Matrix(b,c):
V={s,t}
E={}
G = <V,E>
m = |b|
c = |c|
for i = 1 to m
    V = V U {bi}
    E = E U {(s,bi,b[i])}
end
for j = 1 to n
    V = V U {cj}
    E = E U {(cj,t,c[j])}
end
for i = 1 to m
    for j = 1 to m
        V = V U {aij}
        E = E U {(bi,aij,1),(aij,cj,1)}
    end
end
max_flow=Push-Relabel(G)
for i = 1 to m
    for j = 1 to m
        Aij = max_flow((ci,aij))
    end
end
return A
```

正确性证明

分为两部分证明:

1.满足矩阵元素为0，1的约束
通过约束第二层到第三层边和第三层到第四层边的权值为1，使得第三层所有的节点在最大流中，入流量等于出流量，可取的值为0和1。而该值即为对应矩阵元素的值，满足了问题的0，1约束。

2.返回的结果满足行和，列和的约束
首先，行和的总和等于列和的总和。其次对于第二层的节点（行和）而言，最大总流出量大于等于最大总流出量，同样对于第四层节点（列和）而言最大总流入量大于等于最大总流出量。所以最大流必然满足：起点到第二层所有边以及第四层到汇点所有边都被占满。结合1可以得出返回的矩阵满足了行和、列和约束。

综上所述，算法得出的矩阵满足问题要求，算法正确。

算法复杂度

算法的时间复杂度为$O((mn)^3)$，其中m为矩阵的行数，n为矩阵的列数。构造网络的时间复杂度为$O(mn)$。Push-Relabel算法的复杂度为$O(V^2E)$，其中V为顶点数，E为边数。在该问题中，
$V=mn+m+n+2, E=m+n+2mn$，用Push-Relabel算法求构造网络最大流的时间复杂度为$O((mn)^3)$。总时间复杂度为$O(mn+(mn)^3)$，也即$O((mn)^3)$

3 Unique Cut 3

问题描述

Let $G=(V,E)$ be a directed graph,with source $s \in V$, sink $t \in V$,and non-negative edge capacities c_e . Give a polynomial-time algorithm to decide whether G has a unique minimum st cut.

算法描述

可以使用Ford-Fulkerson算法得到有向图的一个最小割。

首先找到有向图的一个最小割。然后分别将最小割所对应边的集合中的每一条边的容量加1，然后计算新网络的最小割，如果存在一个新的网络所求得的最小割与最先得到的最小割总容量相等，那么该有向图的最小割不是唯一的，否则该有向图的最小割是唯一的。

伪代码

C表示最小割对应的边的集合，c(C)表示最小割的大小。

```
UniqueCut(G):
C = MinCut(G)
for e in C:
    ce += 1
    C' = Ford-Fulkerson(G)
    if c(C) == c(C')
        return false
    end
    ce -= 1
end
return true
```

正确性证明

分为两部分证明：

1.如果新图的最小割大小和原始图大小相等，最小割不是唯一的
因为$c(C) = c(C')$，而因增加了原始图中最小割边集合中一条边的权值，所以新的最小割边集合必然和原来最小割集合不一致，也因此显然原始图中存在多个最小割。

2.如果所有新图的最小割大小和原始图最小割大小都不想等，最小割是唯一的
假设在这种情况下，存在另一个不同的最小割C'，有$c(C) = c(C')$。那么必然存在一条边$e \in C$并且$e \notin C'$，增加e的大小不会改变最小割的大小，这与2的条件相违背，所以在该条件下最小割是唯一的。

算法复杂度

算法的复杂度为$O(m^2C)$，其中m为有向图边的总数，C为起点的最大流出量。Ford-Fulkerson算法的复杂度为$O(mC)$。最小割边集合中最少有一条边，最多有$m-1$条边，所以循环中最多执行$m-1$次Ford-Fulkerson算法，总共执行m次Ford-Fulkerson算法。

4 Problem Reduction 4

问题描述

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.

算法描述

通过构造网络，利用最小费用流求解。

对于矩阵中每个元素用两个节点b_{ij}，c_{ij}表示，b_{ij}向c_{ij}有一条边，费用为对应元素在矩阵中的值。除了b_{ij}，c_{ij}之间的边之外，所有连接b_{ij}的边都指向b_{ij}，所有连接c_{ij}的边都从c_{ij}出发。按照矩阵中的位置连接相邻的节点，比如c_{ij}连接$b_{i(j+1)}$和$b_{(i+1)j}$，右侧和底部的节点情况特殊，只有一个可行方向。除了(b_{00}, c_{00})和$(b_{(m-1)(n-1)}, c_{(m-1)(n-1)})$之外所有边的最大流量为1。

通过求解流量为2的最小费用流可以得到问题的解。

伪代码

(b,c,x,y)表示从b指向a的边，流量为x，费用为y。flow((b,c))表示最小费用流中(b,c)边的流量。

```
MatrixWalk(A):
m,n = size(A)
V={bij:i∈[0,m),j∈[0,n),i∈Z,j∈Z} U {cij:i∈[0,m),j∈[0,n),i∈Z,j∈Z}
E={(bij,cij,1,Aij):i∈[0,m),j∈[0,n),i∈Z,j∈Z,i+j∉{0,m+n-2}}
E = E U {(b00,c00,2,A00) and (b(m-1)(n-1)c(m-1)(n-1),2,A(m-1)(n-1))}
G=<V,E>
for i=0 to m-1
    for j=0 to n-1
        if i != m-1
            E = E U {(cij,b(i+1)j,1,0)}
        end
        if j != n-1
            E = E U {(cij,bi(j+1),1,0)}
        end
    end
end
flow=Klein(G)
for i=0 to m-1
    for j=0 to n-1
        Sij = flow((bij,cij))
    end
end
return S
```

正确性证明

通过以下几点证明:

1.满足所有节点只经过一次。
构造的网络中，边(b_{ij}, c_{ij})可以代表矩阵中的一个元素，因为除了左上角和右下角元素的最大流量为2之外，其余所有元素的最大流量为1，保证了该元素只经过一次。

2.满足一去一回和方向性要求 通过寻找流量为2的费用流，因为除了起始和结束的一条边之外，所有边的流量为1，将会导致有两条线路从c_{00} 到达$b_{(m-1)(n-1)}$ 。因为所有的c_{ij} 连接$b_{i(j+1)}$ 和$b_{(i+1)j}$ ，保证了方向性。两条线路中具体哪条线路去，哪条线路回均可，不影响结果。

3.费用最小 在满足1，2的基础上，求解最小费用流问题，得到的网络流显然费用最小，也即对应的行走方式付出的费用最低。

综上所述，算法正确。

算法复杂度

时间复杂度为$O(mn)$，其中m是行数，n是列数。构造网络的复杂度为$O(mn)$。可以使用Ford-Fulkerson算法结合Klein算法求解最小费用流问题，Ford-Fulkerson算法的时间复杂度为$O(VC)$，在本题中，
$V = 3mn - m - n, C = 2$，复杂度为$O(mn)$。

5 Ford-Fulkerson algorithm 8

问题描述

Implement Ford-Fulkerson algorithm to find the maximum flow of the fol- lowing network, and list your intermediate steps. Use you implementation to solve problem 1 and show your answers.

INPUT: (N, M) means number of jobs and computers. Next N line, each line has two computer ID for a job. see more detial in the file problem1.data.

OUTPUT: the minimum number of the max load.

Ford-Fulkerson 算法实现 （*Python*）

Python

```python
#!/usr/bin/python3
# Ford-Fulkerson Algorithm
# Author: HongXin
# 2016.12.13

import networkx as nx


class FordFulkerson:
    def __init__(self, graph, s, t):
        self.__source = s
        self.__target = t
        if type(graph) is nx.classes.digraph.DiGraph:
            self.__graph = graph
        else:
            self.__graph = nx.DiGraph()
            self.__graph.add_weighted_edges_from(graph)
        self.flow = nx.DiGraph()
        self.f = 0
        self.__step = 0

    def get_max_flow(self, mid=True):
        # mid - True for print intermediate process of flow, False for not.
        while True:
            path = self.__get_a_path()
            if path is None:
                break
            bottleneck = self.__bottleneck(path)
            self.__construct_residual_graph(path, bottleneck)
            self.__update_flow(path, bottleneck)
            if mid:
                self.__print_flow()
        return self.f

    def __get_a_path(self):
        # Return a random path from source to target, None for no available pat
        # also can use dijkstra algorithm here to get a shortest path
        # the method which I used here really get a random path from source to
        paths = list(nx.all_simple_paths(self.__graph, self.__source, self.__ta
        return None if len(paths) == 0 else paths[0]

    def __bottleneck(self, path_nodes):
        # Find the minimum edge of given path
        return min(map(lambda i: self.__graph.get_edge_data(path_nodes[i], path
                       range(len(path_nodes) - 1)))

    def __construct_residual_graph(self, path_nodes, bottleneck):
        # Construct residual graph.
        # Update(decrease) capacity first and update(increase or add new) backw
        for i in range(len(path_nodes) - 1):
            start, end = path_nodes[i], path_nodes[i + 1]
```

```python
                self.__graph[start][end]['weight'] -= bottleneck
                if self.__graph[start][end]['weight'] == 0:
                    self.__graph.remove_edge(start, end)
                if self.__graph.has_edge(end, start):
                    self.__graph[end][start]['weight'] += bottleneck
                else:
                    self.__graph.add_edge(end, start, weight=bottleneck)

    def __update_flow(self, path_nodes, bottleneck):
        self.f += bottleneck
        # Update flow.
        for i in range(len(path_nodes) - 1):
            start, end = path_nodes[i], path_nodes[i + 1]
            if self.flow.has_edge(start, end):
                self.flow[start][end]['weight'] += bottleneck
            elif self.flow.has_edge(end, start):
                remain = self.flow[end][start]['weight'] - bottleneck
                if remain > 0:
                    self.flow[end][start]['weight'] = remain
                else:
                    self.flow.remove_edge(end, start)
                    if remain < 0:
                        self.flow.add_edge(start, end, weight=-remain)
            else:
                self.flow.add_edge(start, end, weight=bottleneck)

    def __print_flow(self):
        # Example: Step 0 : 1 -> [('s', 'u', {'weight': 1}), ('u', 'v', {'weigh
        if self.__step == 0:
            print('Intermediate Process: (max flow & detail):')
        print('Step', self.__step, ':', self.f, '->', self.flow.edges(data=True
        self.__step += 1


if __name__ == '__main__':
    edge_list = [('s', 'u', 1), ('s', 'v', 1), ('u', 'v', 1), ('u', 't', 1), ('
    ford = FordFulkerson(edge_list, 's', 't')
    print('Max Flow:', ford.get_max_flow())

```

Min load 算法实现

Python

```python
def file2problems(path):
    f = open(path, 'r')
    problems, job_count, job = [], 0, 0
    for line in f:
        line = line.strip()
        if not(line.startswith('#') or line == ''):
            if job > job_count or job == 0:
                if job != 0:
                    problems.append(graph)
                job_count = int(line.split(' ')[0])
                job = 1
                graph = nx.DiGraph()
            else:
                computers = line.split(' ')
                graph.add_edge('s', 'j'+str(job), weight=1)
                graph.add_edge('j'+str(job), 'c'+computers[0], weight=1)
                graph.add_edge('j'+str(job), 'c'+computers[1], weight=1)
                graph.add_edge('c'+computers[0], 't', weight=job_count)
                graph.add_edge('c'+computers[1], 't', weight=job_count)
                job += 1
    problems.append(graph)
    return problems


def min_load(graph):
    job_count = len(graph.edges('s'))
    x, y = 0, job_count
    while True:
        load = (x+y)//2
        for edge in graph.in_edges('t', data=True):
            edge[2]['weight'] = load
        ford = FordFulkerson(graph.copy(), 's', 't')
        max_flow = ford.get_max_flow(False)
        if max_flow == job_count:
            y = load
        elif y == x + 1:
            return y
        else:
            x = load

if __name__ == '__main__':
    _list = file2problems('problem1.data')
    for problem in _list:
        print('Min Load: ', min_load(problem))
```

运行结果

```
Min Load:  2
Min Load:  1
Min Load:  2
Min Load:  3
Min Load:  1
Min Load:  1
Min Load:  3
Min Load:  1
Min Load:  1
Min Load:  1
```

中间过程示例（问题一）

```
Intermediate Process: (max flow & detail):
Step 0 : 1 -> [('j3', 'c2', {'weight': 1}), ('c2', 't', {'weight': 1}), ('s', '
Step 1 : 2 -> [('j3', 'c2', {'weight': 1}), ('c2', 't', {'weight': 2}), ('j2',
Step 2 : 3 -> [('j3', 'c1', {'weight': 1}), ('c1', 't', {'weight': 1}), ('c2',
Step 3 : 4 -> [('j3', 'c1', {'weight': 1}), ('c1', 't', {'weight': 2}), ('c2',
Intermediate Process: (max flow & detail):
Step 0 : 1 -> [('j3', 'c2', {'weight': 1}), ('c2', 't', {'weight': 1}), ('s', '
Step 1 : 2 -> [('j3', 'c1', {'weight': 1}), ('c2', 't', {'weight': 1}), ('j2',
Intermediate Process: (max flow & detail):
Step 0 : 1 -> [('j3', 'c2', {'weight': 1}), ('c2', 't', {'weight': 1}), ('s', '
Step 1 : 2 -> [('j3', 'c1', {'weight': 1}), ('c2', 't', {'weight': 1}), ('j2',
Min Load:  2
```

6 Push-relabel

问题描述

Implement push-relabel algorithm to find the maximum flow of a network, and list your intermediate steps. Use your implementation to solve problem 2 and write a check problem to see if your answer is right.

INPUT: Numbers of rows and columns. And the sum of them. See more detial in the file problem2.data.

OUTPUT: The matrix you get. Any one satisfy the conditions will be accept.

Push-Relabel 算法实现 （*Python*）

Python

```python
#!/usr/bin/python3
# Push-relabel Algorithm
# Author: HongXin
# 2016.12.14

import networkx as nx


class PushRelabel:
    def __init__(self, graph, s, t):
        self.__source = s
        self.__target = t
        if type(graph) is nx.classes.digraph.DiGraph:
            self.__graph = graph
        else:
            self.__graph = nx.DiGraph()
            self.__graph.add_weighted_edges_from(graph)
        self.pre_flow = nx.DiGraph()
        self.f = 0
        self.__step = 0

    def get_max_flow(self, mid=True):
        self.__init_node()
        self.__init_pre_flow()
        while True:
            node = self.__get_non_zero_excess_node()
            if node is None:
                break
            edge = self.__get_high_start_edge(node)
            if edge is None:
                self.__relabel(node)
                if mid:
                    self.__print_label()
            else:
                bottleneck = self.__push(edge)
                self.__construct_residual_graph(edge, bottleneck)
                if mid:
                    self.__print_push()

        self.f = self.__excess(self.__target)
        return self.f

    def __init_node(self):
        # Init height. Assign source with n and others with 0.
        for node, data in self.__graph.nodes_iter(True):
            data['height'] = 0
        self.__graph.node[self.__source]['height'] = len(self.__graph.node)

    def __init_pre_flow(self):
        # Init pre_flow. Assign edges start from source with its capacity.
        self.pre_flow.add_edges_from(self.__graph.edges(self.__source, data=Tru
        # Construct residual graph
        for start, end, data in self.__graph.edges(self.__source, data=True):
            self.__graph.add_edge(end, start, weight=data['weight'])
            self.__graph.add_edge(start, end, weight=0)

    def __get_non_zero_excess_node(self):
        result = None
        for node in self.pre_flow.nodes():
            if self.__excess(node) > 0 and node != self.__source and node != se
                # find the lowest non zero excess node
                if result is None or self.__graph.node[node]['height'] < self._
                    result = node
        return result

    def __excess(self, node):
        into, out, = 0, 0
        for edge in self.pre_flow.in_edges(node, data=True):
            into += edge[2]['weight']
        for edge in self.pre_flow.out_edges(node, data=True):
            out += edge[2]['weight']
        return into - out

    def __get_high_start_edge(self, node):
        for start, end, data in self.__graph.edges(node, data=True):
            if data['weight'] > 0 and self.__graph.node[start]['height'] > self
                return start, end, data
        return None

    def __push(self, edge):
        start, end, weight = edge[0], edge[1], edge[2]['weight']
        excess = self.__excess(start)
        if self.pre_flow.has_edge(start, end):
            bottleneck = min(weight, excess)
            self.pre_flow.edge[start][end]['weight'] += bottleneck
        elif self.pre_flow.has_edge(end, start):
            bottleneck = min(self.pre_flow[end][start]['weight'], excess)
            if bottleneck == self.pre_flow[end][start]['weight']:
                self.pre_flow.remove_edge(end, start)
            else:
                self.pre_flow.edge[end][start]['weight'] -= bottleneck
        else:
            bottleneck = min(weight, excess)
            self.pre_flow.add_edge(start, end, weight=bottleneck)
        return bottleneck

    def __construct_residual_graph(self, edge, bottleneck):
        start, end = edge[0], edge[1]
        self.__graph.edge[start][end]['weight'] -= bottleneck
        if self.__graph.has_edge(end, start):
            self.__graph.edge[end][start]['weight'] += bottleneck
        else:
            self.__graph.add_edge(end, start, weight = bottleneck)

    def __relabel(self, node):
        self.__graph.node[node]['height'] += 1

    def __print_label(self):
        print('Step', self.__step, '(Relabel):', self.__graph.node)
        self.__step += 1

    def __print_push(self):
        print('Step', self.__step, '(Push):', self.pre_flow.edge)
        self.__step += 1

if __name__ == '__main__':
    # edge_list = [('s', 'u', 1), ('s', 'v', 1), ('u', 'v', 1), ('u', 't', 1),
    edge_list = [('s','u',3),('u','v',1),('v','t',2)]
    ford = PushRelabel(edge_list, 's', 't')
    print('Max Flow:', ford.get_max_flow())
```

Get Matrix 算法实现

Python

```
def file2problems(path):
    f = open(path, 'r')
    problems, line_num = [], 0
    for line in f:
        line = line.strip()
        if not (line.startswith('#') or line == ''):
            if line_num == 0:
                m, n = map(int, line.split(' '))

            if line_num > 2:
                line_num = 0
                graph = nx.DiGraph()
                for i in range(m):
                    graph.add_edge('s', 'b' + str(i), weight=b[i])
                for j in range(n):
                    graph.add_edge('c' + str(j), 't', weight=c[j])
                for i in range(m):
                    for j in range(n):
                        graph.add_edge('b' + str(i), 'a' + str(i) + str(j), wei
                        graph.add_edge('a' + str(i) + str(j), 'c' + str(j), wei
                problems.append((graph, m, n))
                m, n = map(int, line.split(' '))
```

```python
            elif line_num == 1:
                b = list(map(int, line.split(' ')))
            else:
                c = list(map(int, line.split(' ')))

            line_num += 1

    problems.append(graph)
    return problems


def flow2matrix(flow, m, n):
    matrix = np.zeros((m, n))
    for i in range(m):
        for j in range(n):
            if flow.has_node('a' + str(i) + str(j)) \
                    and flow.has_edge('a' + str(i) + str(j), 'c' + str(j))\
                    and flow.edge['a' + str(i) + str(j)][ 'c' + str(j)]['weight
                matrix[i][j] = 1
    return matrix


if __name__ == '__main__':
    _list = file2problems('problem2.data')
    for _graph, _m, _n in _list:
        push = PushRelabel(_graph, 's', 't')
        push.get_max_flow(False)
        print(flow2matrix(push.pre_flow, _m, _n))
```

部分结果 (第一组、第二组)

```
[[ 1.  0.  1.  1.  0.  0.  0.  0.  1.  1.]
 [ 0.  1.  0.  0.  1.  0.  1.  0.  1.  1.]
 [ 0.  0.  1.  1.  1.  1.  1.  0.  1.  1.]
 [ 0.  0.  1.  0.  1.  1.  1.  0.  1.  1.]
 [ 0.  0.  1.  1.  0.  1.  1.  0.  1.  1.]
 [ 0.  0.  1.  0.  0.  0.  1.  0.  0.  1.]
 [ 1.  0.  1.  0.  0.  0.  1.  0.  1.  1.]
 [ 0.  0.  1.  1.  1.  1.  1.  1.  1.  0.]
 [ 0.  0.  1.  0.  1.  1.  1.  0.  1.  1.]
 [ 0.  0.  0.  0.  1.  0.  1.  0.  1.  0.]]
```

```
[[ 0.  0.  1.  1.  1.  0.  1.  0.  1.  1.  1.  1.  1.  1.]
 [ 0.  0.  1.  0.  1.  0.  1.  0.  1.  0.  1.  1.  0.  0.]
 [ 0.  0.  1.  1.  0.  0.  1.  0.  1.  1.  0.  0.  1.  1.]
 [ 0.  0.  0.  1.  1.  1.  0.  0.  1.  0.  0.  1.  0.  1.]
 [ 0.  0.  0.  0.  0.  1.  1.  0.  0.  1.  0.  0.  0.  1.]
 [ 0.  0.  1.  0.  0.  0.  0.  0.  0.  1.  0.  0.  1.  1.]
 [ 0.  0.  1.  0.  0.  0.  1.  0.  0.  1.  0.  0.  1.  1.]
 [ 0.  1.  1.  0.  1.  1.  1.  1.  1.  0.  1.  1.  0.  0.]
 [ 1.  0.  1.  0.  1.  1.  1.  0.  1.  1.  1.  1.  0.  0.]
 [ 0.  0.  1.  0.  1.  0.  1.  0.  1.  1.  1.  1.  1.  0.]
 [ 0.  0.  0.  0.  0.  1.  1.  0.  0.  1.  1.  0.  1.  1.]
 [ 0.  0.  1.  1.  1.  0.  0.  0.  1.  1.  1.  1.  1.  1.]
 [ 0.  0.  0.  0.  0.  0.  1.  0.  0.  0.  1.  1.  1.  1.]]
```