CS 6150: Assignment 4
Due: Dec 12, 2014

This assignment has 5 questions, for a total of 100 points. Unless otherwise specified, complete and reasoned arguments will be expected for all answers.

Question	Points	Score
Cycle covers	20	
Disconnecting a railroad	20	
Targeted Ad Placement	20	
Covering Marked Cells	20	
Nested Boxes	20	
Total:	100	

In the questions that follow, the goal in each case is to solve the problem using some kind of flow construction. A valid answer will specify how the input flow network is constructed, what problem is solved, and how the solution is used to obtain the desired answer. For full marks, you must explain the procedure, and prove that it yields the desired answer. Unlike in other assignments we will not be concerned with running time in these questions, since you will merely invoke a standard algorithm from one of the flow variants we've studied in class. Of course, the transformation should run in polynomial time.

Question 1: Cycle covers . . . . . **[20]**

Solve Question 4.

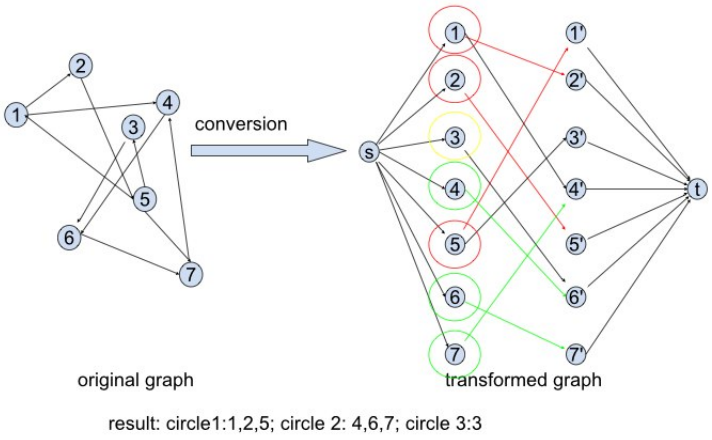

Figure 1

Solution:

Algorithm 1: Cycle Cover

Require: A given directed graph $G = (V, E)$.
Ensure: A cycle cover $List\ ans[1..m]$ that cover all the vertices and the cycles are vertex-disjoint.

List $Header[0..2n+1]$;
$C[0..2n+1][0..2n+1]$;
$memset(C, 0, sizeof(C))$;
for each edge $i \to j$ in E **do**
    add $n+j$ to List $Header[i]$;
    $C[i][n+j] = 1$;
end for
for $i = 1$ to n **do**
    add i to List $Header[0]$;
    $C[0][i] = 1$;
    add $n+i$ to List $Header[2n+1]$;
    $C[n+i][2n+1] = 1$;
end for

14: $match[1..n]$;
15: $f = \text{FIND\_A\_FLOW}(Header[0..2n+1], C[0..2n+1][0..2n+1])$;
16: $G' = \text{CREATE\_RESIDUAL\_GRAPH}(Header[0..2n+1], C[0..2n+1][0..2n+1])$;
17: **while** $Search\_Augmenting\_Path(G')$ **do**
18: add the augmented flow f' to f;
19: modify $match[1..n]$;
20: create the new residual graph G';
21: **end while**
22: **while** $!isEmpty(V)$ **do**
23: **if** vertex i is not visited **then**
24: $j = i$;
25: **while** $j \neq i$ and $j \neq -1$ **do**
26: $j = match[j]$;
27: **end while**
28: **if** $j == -1$ **then**
29: return *no cycle cover*!;
30: **else**
31: all vertexes in this cycle form a list in a right order and delete them from V, adding the list to $ans[1..m]$;
32: **end if**
33: **end if**
34: **end while**
35: return $set[1..m]$;

EXPLANATION: For the algorithm, Line 1 to Line 13 creates a new graph with source s and the target t. $Header[0...2n+1]$ is the adjacent list storeing the new graph. $Header[0]$ represents s and $Header[2n+1]$ represents t. For each vertex i in G, separate it into two vertexes i,i'. $Header[1..n]$ represent vertexes 1,..,n. $Header[n+1..1n]$ represent 1',...,n'. If edge ij exists, then connext i and j'. Connect s to 1,...,n and connect 1',..,n' to t. C stores the edges' capacities in the new graph, for each edge, the capacity is 1. As the example in Figure 1 shows, there are seven nodes in the original graph and it's transformed to a new graph. The nodes 1,2,3,4,5,6,7 are separated into two nodes, i.e., 1,1',2,2',3,3',4,4',5,5',6,6',7,7'. The edges on the new graph all have capacity 1. The problem is reduced to find a maximum matching in a bipartite graph. Line 15 to Line 21 is a max-flow algorithm on the new graph. $match[1..n]$ stores the the succeed vertexes of 1,...,n in the final max flow, i.e., edge $j \rightarrow match[i]$ is in the max flow f. $match[1..n]$ illustrates the matching situation between 1,...,n and 1',...,n'. For example, in Figure 1, the red and green edges are in the max flow and $match[1] = 5$ and so on. Line 22 to Line 34 determines whether a cycle cover exists and find the cycle cover using DFS. $ans[1..m]$ stores these cycles. For example, in Figure 1, starting from 5, node 2 can be found since $match[5] = 2$ and node 1 can be found since $match[2] = 1$ and then since $match[1]p = 5$, the circle $1, 2, 5$ can be found.

Proof. This algorithm converts this cycle cover problem into bipartite matching problem. Separate each vertex i to i and i'. Then 1,..,n forms the left-side vertex set U representing the starting points of edges in the original graph and 1',...,n' forms the right-side vertex set W representing the ending points. As the notes shows, the max-flow algorithm on the new graph could find the max-matching between 1,...,n and 1',...,n'. Now just to show the max-matching has found the cycle cover.
Firstly, it's obviously that each edge in the max-matching is in the original graph. If the cycle cover exists, then all the vertexes are in cycles and for each vertex, there exists an edge starting from it. Hence the edges size of the cycle cover is n. It implies that the size of max-matching should be n(each vertex corresponds to an edge). Hence if the size of max-matching is smaller than n, it implies there is no cycle cover as the algorithm runs. If the max-matching size is n, the matching

is just the cycle cover. Given the matching, for each vertex, it has at most one out degree. It's obviously that each vertex has a degree now. If there are vertexes not in a cycle, at least a vertex has no incoming edge or outgoing edge. If there exists a vertex i having no incoming edge, then i' has no matching. It implies that 1,..,$i'-1$,$i'+1$,..,n' have n matchings and there is a vertex has at least two matchings which is impossible. If vertex i has no outgoing edge, then the matching size can't be n as well. So all the vertexes are in cycles. If there are two cycles not disjoint, there should be at least one vertex in two cycles and its out degree and in degree should be more than one. As the matching shows, the degree has to be one, so the assumption is impossible. Hence if the size of the max-matching is n, then it has built the cycle cover and the edges in the matching are all the edges of the cycle cover. Obviously, using DFS on the max-matching could find the cycle cover out. □

*Analysis.*For the first part of the algorithm, it has to check each edge in E and edge the new edges from s and to t. So the time complexity is O(E). For the second part, we could compute the maximum flow in O(VE) time using either Orlin's algorithm or off-the-shelf Ford-Fulkerson. For the third part, the running time of DFS is O(V). So the running time of the whole algorithm is O(VE) and computing the maximum flow is the major part. Give the space cost, there are majorally an adjacent list $Header$ with space cost $O(E+2V)$, the two-dimension array C with cost $O(V^2)$ and $ans[1..m]$ with $O(V)$. So the space cost is $O(E+3V+V^2)=O(V^2)$.

Question 2: Disconnecting a railroad..... **[20]**

Solve Question 5. Remember that the goal to find a minimum set of *vertices* whose removal separate the two stations.

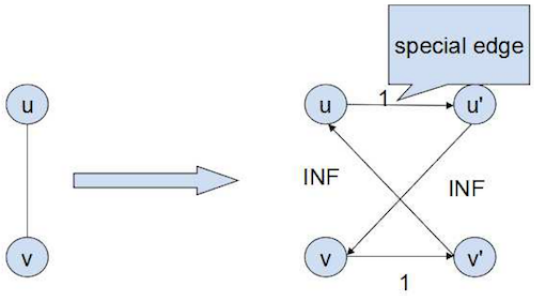

Figure 2

Solution:

Algorithm 2: Disconnecting a railroad

Require: An undirected graph with a source vertex s(vertex 1) representing station Ffarquhar and a target vertex t (vertex n)representing station Tidmouth.

Ensure: A minimum set ans of vertices whose removal separate the two stations.

1: $C[1..2n][1..2n]$;
2: $memset(C, 0, sizeof(C))$, $C[i][j]=0$ represents there is no edge between vertex i and vertex j;
3: **for** each vertex i in V **do**
4: separate i into two vertices i and i' and add an edge $i \to i'$ with capacity 1:$C[i][n+i]=1$;
5: **for** each vertex j connecting with i **do**

6: add an edge $i' \to j$ with capacity ∞:$C[n+i][j] = INF$;
7: **end for**
8: **end for**
9: a new graph $G = (E, V)$ has been created with capacities $C[1..2n][1..2n]$, vertex s' as a new source and vertex t still as the target;
10: create $f[1..2n][1..2n]$;
11: find a flow f in G with DFS or BFS or Shortest Path Algorithm;
12: create the residual graph $G' = (E', V')$ of G: $C'[u][v] = C[u][v] - f[u][v]$ if $u \to v \in E$,$C'[u][v] = f[v][u]$ if $v \to u \in E$, otherwise $C'[u][v] = 0$;
13: **while** an augment flow $f' : v_0 \to v_1 \to \ldots \to v_r$ can be found in G' **do**
14: let $F = min_i C[v_i][v_{i+1}]$;
15: add f' to f: If $u \to v$ in f', $f[u][v] = f[u][v] + F$, if $v \to u$ in f', $f[u][v] = f[u][v] - F$, otherwise, $f[u][v] = f[u][v]$;
16: continually create the residual graph G' of G with the new flow f;
17: **end while**
18: find the vertices in the final residual graph G' reachable from the source s using DFS and add them to set S;
19: add the remaining vertices to T;
20: **for** each $u \in S$ **do**
21: **for** each $v \in T$ **do**
22: **if** edge uv exists in G **then**
23: add u to set ans;
24: **end if**
25: **end for**
26: **end for**
27: return ans;

EXPLANATION: Line 1 to Line 9 creates a new graph G. For each vertex i in the input graph, separate it into i and i' and add them to G. Then add edge $i \to i'$ to G. If undirected edge ij exists, then add new edges $i' \to j$ and $j' \to i$ to G. Figure 2 shows the process. Line 10 to Line 19 is the process of finding the min-cut of the new graph. Then the cut edges' starting vertices are just the minimum set of stations that needed to be closed. The last part returns the result. $C[1..2n][1..2n]$ stores the edges' capacities of the new graph and $C'[1..2n][1..2n]$ stores those of the residual graph. $f[1..2n][1..2n]$ stores the flow units of graph' edges. f' represents the augmented flow in the residual graph.INF means ∞. And I will use the definition *special edge* to represent edges like $u \to u'$.

Proof. Firstly, prove the input undirected graph and the created new graph G is equivalent. DEFINE u could reach v in G as u and u' could both reach v and v'. If u and v in the input graph are reachable from each other(since it's an undirected graph, once u and v are connected, they could reach each other), i.e., $u \leftrightarrow r_1 \leftrightarrow\leftrightarrow r_2 \leftrightarrow \ldots \leftrightarrow r_k \leftrightarrow v$, then in G, since $u \to u' \to r_1 \to r_1' \to r_2 \to r_2' \to \ldots \to r_k \to r_k' \to v \to v'$, so v is still reachable from u and similarly, u is still reachable from v. Hence the two graphs are equivalent.
Secondly, find the equivalent operations of the two graphs. If we delete the *special edges*, then all the vertices can't reach u as defined, and the path that passes through u has been blocked. It's equivalent to that we delete the vertex(station) u in the input graph. So the problem is equivalent to delete the fewest *special edges* in G that s' and t can't reach each other. We have proved that if s' can't reach t, then t can't reach s'. So the problem is simplified - s' can't reach t. If s' can't reach t, then all vertices s' can reach and can't reach could form two sets - S and T. And the deleted edges are the only edges from S to T. Our problem now is to find such a division then the edges from S to T should all be *special edges*, and the size is minimum. It's obviously a min-cut and max-flow problem. It can be proved.

The min-cut problem is to find a division of G - S and T such that the total capacity $C[S,T]$ of edges from S to T is minimum. Since $C[S,T] = |f_{max}|$, it is finite. Since the edges other than *spacial edges* have INF capacities, so the edges from S to T in the min-cut are all *special edges*. Since they all have capacity 1, the total capacity is just the size of *special edges* from S to T. In other words, use min-cut and max-flow algorithm on G can help find fewest *special edges* connecting S and T. *special edge* $s \to s'$ is either from T to S or in S and $t \to t'$ is either from T to S or in T, so they won't be cut, which implies that the source station and target station won't be closed. Set S is just the vertices that can be reached from s' in the final residual graph and the others form T, this has been shown in the notes.
Finally, each *special edge* corresponds to a vertex(station) in the original input graph. So the minimum set can be found.
The algorithm creates the new graph G, runs max-flow and min-cut on G , uses DFS to find S and T and finally finds the *special edges* from S to T, so it can correctly find the minimum set of stations. □

*Analysis.*Creating G costs $O(V^2)$. The major part is the min-cut and max-flow algorithm. It's obviously the max flow $\|f*\|$ in this case is $O(V)$, since $\|f*\| = C[S,T]$ is the size of *special edges*. Ford-Fulkerson algorithm searches the augmented path with one unit capacity every iteration, so it's suitable for this case and its cost in this case is $O(EV)$. In addition, forming the cut S and T on the final residual graph costs $O(V)$ and finding the *special edges* from S to T costs $O(V^2)$. So the total time cost is $O(VE + V^2)$. For the space cost, the major cost is on $C[1..2n][1..2n]$ and $f[1..2n][1..2n]$, separately representing the edges' capacities and the max flow. So the space cost is $O(n^2)$.

Question 3: Targeted Ad Placement ..... **[20]**
Solve Question 13.

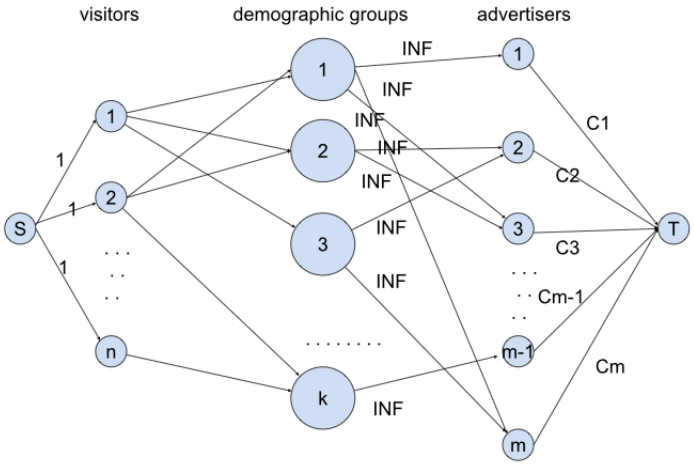

Figure 3

Solution:

Algorithm 3: Targeted Ad Placement

Require: n visitors and their memberships in demographic groups, k demographic groups, m advertisers and their target ad groups and their desired number of ads displayed - $C_1, C_2, ..., C_m$.

Ensure: To determine whether every advertiser has its desired number of ads displayed and every ad is seen by someone in suitable group if each visiter can seen only one ad per day.

1: create a graph G with a source vertex s(1) and a target vertex t($n+k+m+2$):$C[1..n+k+m+2][1..n+k+m+2]$;
2: **for** $i = 1$ to n **do**
3: $C[1][1+i] = 1$
4: **end for**
5: **for** $i = 1$ to n **do**
6: **for** $j = 1$ to k **do**
7: $C[1+i][1+n+j] = 1$ if visitor i claims membership in group j;
8: **end for**
9: **end for**
10: **for** $i = 1$ to k **do**
11: **for** $j = 1$ to m **do**
12: $C[1+n+i][1+n+k+j] = \infty$ if advertiser j said group i should see its ad;
13: **end for**
14: **end for**
15: **for** $i = 1$ to m **do**
16: $C[1+n+k+i][n+k+m+2] = C_i$;
17: **end for**
18: a graph $G = (E, V)$ has been created with edges $C[1..n+k+m+2][1..n+k+m+2]$;
19: create $f[1..n+k+m+2][1..n+k+m+2]$;
20: find a flow f in G using DFS or BFS or Shortest Path Algorithm;
21: create the residual graph $G' = (E', V')$ of G: $C'[u][v] = C[u][v] - f[u][v]$ if $u \to v \in E$,$C'[u][v] = f[v][u]$ if $v \to u \in E$, otherwise $C'[u][v] = 0$;
22: **while** an augment flow $f' : v_0 \to v_1 \to \ldots \to v_r$ can be found in G' **do**
23: let $F = min_i C[v_i][v_{i+1}]$;
24: add f' to f: If $u \to v$ in f', $f[u][v] = f[u][v] + F$, if $v \to u$ in f', $f[u][v] = f[u][v] - F$, otherwise, $f[u][v] = f[u][v]$;
25: continually create the residual graph G' of G with the new flow f;
26: **end while**
27: **for** $i = 1$ to m **do**
28: **if** $f[1+n+k+i][n+k+m+2] < C_i$ **then**
29: return false;
30: **end if**
31: **end for**
32: return true;

EXPLANATION: Line 1 - Line 17 creates a directed graph G with source vertex s(1) and target vertex t($n+k+m+2$). Figure 3 is an example. Line 18 - Line 26 runs max-flow algorithm on G. And Line 27 - Line 32 makes the judgement by judging whether each edge connecting target vertex t is saturated. $C[1..n+k+m+2][1..n+k+m+2]$ are the edges' capacities. $C[i][j]$ is the capacity of edge $i \to j$ in G. If $C[i][j] = 0$, then there is no edge from i to j. $C'[1..n+k+m+2][1..n+k+m+2]$ are those of the residual graph G'. $f[1..n+k+m+2][1..n+k+m+2]$ could store the flow.$f[i][j] = 0$ if there is no flow from i to j. f' represents the augmented path and F is the bottleneck capacity of this augmented flow. Vertex 1 is the source vertex and t is the target vertex. Vertexes 2 - $n+1$ represent the n visitors. Vertexes $n+2$ - $n+k+1$ represent the k demographic groups. Vertexes $n+k+2$ - $n+k+m+1$ represent the m advertisers and $n+m+k+2$ represents the target vertex.C_i is the number of ad i expected to be displayed by advertiser i.

Proof. An unit flow in G created by this algorithm can be transformed into an event that an ad is seen by a suitable visitor. The flow going through visitor i can only flow into vertexes representing groups visitor i belongs to and then flows into the vertexes representing the ads that only these groups can see and finally flows into t. Since the capacities of the graph are integers, the maximum flow produced by augmented-path algorithm is integer. And the capacities of edges from s are all 1. So each visitor can only see one ad. Hence the number of such events is equal to the flow size $|f*|$. The event that an ad j is seen by a suitable visitor i can be transformed into an unit feasible flow in G conversely: $s \to$ visitor i $\to$ group j $\to$ ad k $\to t$. So the max flow size $|f*|$ is equal to all such events happened per day. The capacities of edges going target t are just the numbers the advertisers expect their ads to be displayed for. So if the maximum flow can't saturate any edges from vertices representing the ads to the target t, it implies that all the visitors have been displayed one ad on that day and still can't satisfy some advertiser. So running max-flow algorithm on G can solve the problem and the algorithm is correct. □

Analysis. For the time cost, creating G costs $O((n+m)(1+k)) = O((n+m)k)$ as the algorithm shows. Running max-flow algorithm on G is the major part. Each augmented path in this case is bound to has one unit capacity using whatever augmenting-path algorithm on G since each edge from s is one unit. Using Ford and Fulkerson's algorithm on G costs $O(E\|f*\|) = O(En) = O((n+nk+km+m)n)$ time since $|f*\|$ is obviously $O(n)$. Finally, the last judging part only costs $O(m)$. So the total time cost is $O((n+m)kn)$. For the space complexity, the major cost is $C[1..n+k+m+2][1..n+k+m+2]$ and $f[1..n+k+m+2][1..n+k+m+2]$, so it's $O((n+k+m)^2)$.

Question 4: Covering Marked Cells ..... **[20]**

Solve all parts of Question 2. The breakdown by part is $5+5+10$.

Part(a)

Solution:

Algorithm 4: Cover the largest number of marked cells

Require: $M[1..n, 1..n]$, which represents the $n \times n$ grid.
Ensure: A monotone path that covers the largest number of marked cells.

1: $DP[1..n][1..n] \leftarrow 0$;
2: $PATH[1..2n-2] \leftarrow 0$;
3: $DP[n][n] \leftarrow 1$ if $M[n][n] = True$, 0 if $M[n][n] = False$;
4: **for** $i \leftarrow n-1$ to 1 **do**
5: $DP[n][i] \leftarrow DP[n][i+1] + 1$ if $M[n][i] = True$, $DP[n][i+1]$ if $M[n][i] = False$;
6: $DP[i][n] \leftarrow DP[i+1][n] + 1$ if $M[i][n] = Ture$, $DP[i+1][n]$ if $M[i][n] = False$;
7: **end for**
8: **for** $i \leftarrow n-1$ to 1 **do**
9: **for** $j \leftarrow n-1$ to 1 **do**
10: $DP[i][j] \leftarrow max(DP[i+1][j], DP[i][j+1])$;
11: **if** $M[i][j] == 1$ **then**
12: $DP[i][j] \leftarrow DP[i][j] + 1$;
13: **end if**
14: **end for**
15: **end for**
16: $i \leftarrow 1$, $j \leftarrow 1$;
17: **for** $step \leftarrow 1$ to $2n-2$ **do**
18: **if** $DP[i+1][j] > DP[i][j+1]$ or $j == n$ **then**

19: $i \leftarrow i + 1$;
20: $PATH[step] \leftarrow DOWN$
21: **end if**
22: **if** $DP[i+1][j] <= DP[i][j+1]$ or $i == n$ **then**
23: $j \leftarrow j + 1$;
24: $PATH[step] \leftarrow RIGHT$;
25: **end if**
26: **end for**

EXPLANATION: This algorithm uses dynamic programming to find the monotone path that covers the largest number of marked cells. $DP[i][j]$ stores the largest number of marked cells starting from grid(i, j). $PATH[1..2n-2]$ stores the path starting from the left top of the $n \times n$ grid. $PATH[i] == DOWN$ means the next step of the monotone path should choose to go down from the current position. Line 3 - Line 15 is the dynamic programming process from the right bottom of the grid to the left top position. And Line 17 to Line 26 finds the monotone path that covered the largest number of marked cells.

Proof. The dynamic programming part of this algorithm can be proved by induction.
Base Case: If $M[n][n] = True$, only one marked cell can be cover starting from (n, n). Hence $DP[n][n] = 1$ is true and trivially, $DP[n][n] = 0$ otherwise;
Hypothesis:$DP[i][j+1]$ and $DP[i+1][j]$ correctly store the largest number of marked cells can cover separately starting from $(i, j+1)$ and from $(i+1, j)$;
Induction: Consider $DP[i][j]$ now. If $i = n$ and starting from this position, then there is only one direction to go - right. And it's similarly true when $j = n$. If $i < n$ and $j < n$, starting from the position (i, j), there are two chooses - go right or go down. Suppose $M[i][j] = True$ and it doesn't influence the judgement. If choosing go down, then there is at most $DP[i+1][j] + 1$ marked cells can be covered since this a monotone path. Similarly, at most $DP[i][j+1] + 1$ marked cells can be covered if choosing going right. So it's correct that $DP[i][j] = max(DP[i+1][j], DP[i][j+1]) + 1$. In conclusion, $DP[i][j]$ can correctly stores the largest number of marked cells. Hence it's obviously true for the process of finding the target path. □

*Analysis.*The algorithm has at most two-level loops. Hence the time cost is $O(n)$). The extra space cost is $DP[1..n][1..n]$ and $PATH[1..2n-1]$, so the space cost is $O(n^2)$.

Part(b)

Solution: This greedy algorithm is wrong. Implement the greedy algorithm on the grid in Figure 4, then the first monotone path is as the red line shows since this path can cover the largest number of marked cells. Then the blue path and the green path are produced. There are three paths produced by the greedy algorithm in total. However, it needs only two path to cover all the marked cells - the green path and the black path. Hence this algorithm is wrong.

part(c)

Solution:

Algorithm 5: finding the smallest set of monotone paths using max-flow algorithm
Require: $M[1..n, 1..n]$, which represents the $n \times n$ grid.
Ensure: A monotone path that covers the largest number of marked cells.

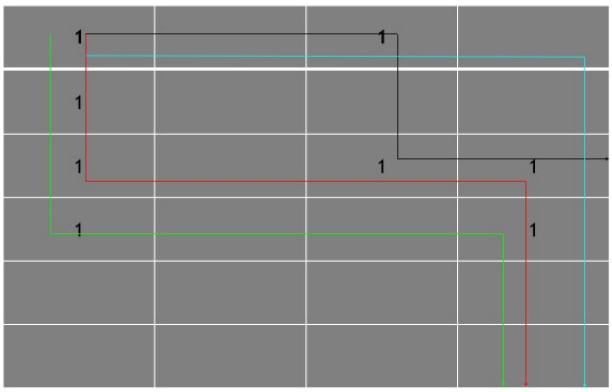

Figure 4

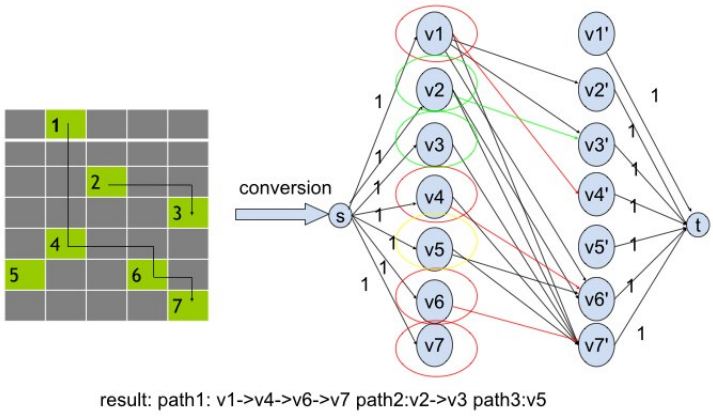

Figure 5

go through the $n \times n$ grid and consider the marked cells as vertices set $V = \{v_1, v_2, ..., v_m\}$;
divide each vertex into two vertices and add a source vertex s and a target vertex t:$V = \{v_1, v_2, ..., v_m\} \rightarrow V\{v_1, v_2, ..., v_m\} + V'\{v'_1, v'_2, ..., v'_m\} + s + t$;
create edges set E:$C[0..2m+1][0..2m+1] \leftarrow 0$;
for $i \leftarrow 1$ to m **do**
  for $j \leftarrow 1$ to m **do**
    if v_j's corresponding cell is in the bottom right region of v_i's corresponding cell **then**
      add an edge from v_i to v'_j with capacity 1 to E:$C[i][m+j] \leftarrow 1$;
    end if

```
    end for
end for
for i ← 1 to m do
    add edge from s to v_i with capacity 1 to E:C[0][i] ← 1;
    add edge from v_i' to t with capacity 1 to E:C[m + i][2m + 1] ← 1;
end for
f[0..2m + 1][0..2m + 1] ← 0;
find a feasible flow f in the graph G = (E, V) using DFS, BFS ,shortest-path or MST algorithm
and update f[0..2m + 1][0..2m + 1];
create the residual graph G' = (E', V') of G: C'[u][v] ← C[u][v] − f[u][v] if u → v ∈ E,C'[u][v] ←
f[v][u] if v → u ∈ E, otherwise C'[u][v] = 0;
while an augment flow f' : s → r_1 → ... → r_k → t can be found in G' using DFS, BFS
,shortest-path or MST algorithm do
    let F ← min_i C[r_i][r_{i+1}];
    add f' to f: If u → v in f', f[u][v] ← f[u][v] + F, if v → u in f', f[u][v] ← f[u][v] − F,
    otherwise, f[u][v] ← f[u][v];
    continually create the residual graph G' of G with the new flow f;
end while
match[1..m] ← 0;
unmatch[count ← 1] ← 0
for i ← 1 to m do
    for j ← 1 to m do
        if f[i][m + j] == 1 then
            macth[j] ← i;
            break;
        end if
    end for
    if no flow crosses v_i then
        unmatch[count + +] ← i;
    end if
end for
for k ← 1 to count do
    i ← unmatch[k];
    path ← empty;
    while match[j ← i]! = 0 do
        add v_j to path and j ← match[j];
    end while
    add path to ans
end for
return ans;
```

EXPLANATION: This algorithm reduces this problem into a max flow problem. Line 1 to Line 14 creates the graph $G = (E, V)$ where V is the set of vertices (each marked cell corresponds to two vertices v_i and v_i') and E is the set of edges representing a marked cell could reach another. $v_1, v_2, ..., v_m$ represent the starting marked cells and $v_1', v_2', ..., v_m'$ represent the ending marked cells, i.e., if marked cell i can reach marked cell j, then there is an edge from v_i to v_j' in the graph. As the example in Figure 5 shows, the marked cells 1,2,3,4,5,6,7 are separated into nodes $v_1, v_2, v_3, v_4, v_5, v_6, v_7$ and $v_1', v_2', v_3', v_4', v_5', v_6', v_7'$. Since marked cell 1 in the grid can reach cells 2,3,4,6,7, the algorithm adds edges $v_1 \rightarrow v_2', v_1 \rightarrow v_3'$ and so on. $C[0..2m + 1][0..2m + 1]$ stores the capacities of these edges. Here, the subscript 0 represents the source vertex, 1,2,..,m represent the vertices $v_1, ..., v_m$, $m + 1, m + 2, ..., 2m$ represent vertices $v_1', v_2', ..., v_m'$ and $2m + 1$ represents the target vertex.$C[i][j] = 0$ represents there is no edge between them. Line 15 to Line

22 runs the max-flow algorithm on G, and $f[0..2m+1][0..2m+1]$ stores the flow. $f[i][j] = 0$ represents there is no flow from i to j. Actually, the max-flow algorithm help find the maximum matching between $v_1, v_2, ..., v_m$ and $v'_1, v_2, ..., v'_m$. Line 23 to Line 44 finds these monotone paths. $match[1..m]$ stores the vertices' matching vertices. $match[i] = j$ means v_i' matches v_j, it implies there is a flow from v_j to v_i' is in the max flow and $v_j \to v_i$ is in one of the target monotone paths of the grid. As Figure 5 shows, the red and green edges represent the max-flow. Since there is a flow from v_1 to v'_4in the max flow,$match[4] = 1$ and so on. $unmatch[]$ stores vertices that has no flow goes through. If there is no flow from v_i to $v'_1, v'_2, ..., v'_m$, it will be added to $unmatch[]$. As Figure 5 shows, v_7, v_5, v_3 are the unmatched vertices. In fact, they are the ending vertices of paths. Line 25 - Line 34 finds these matchings between $v_1, v_2, ..., v_m$ and $v'_1, v'_2, ..., v'_m$ and finds these ending vertices. Line 36 to Line 43 finds these target monotone paths by searching from the ending vertices to their matching ones iteratively. As Figure 3 shows, the vertices bounded by red circles, the vertices bounded by green circles, the vertices bounded by yellow circle will form the minimum set of paths and they will be found by the algorithm.

Proof. The algorithm reduces this problem to a maximum matchings in Bipartite Graph problem and uses the max-flow algorithm to solve the maximum matchings problem. Each monotone path in the grid can be transformed into several matchings in the Bipartite Graph. It creates the Bipartite Graph, where vertices on the left are $v_1, v_2, ..., v_m$ and vertices on the right are $v'_1, v'_2, ..., v'_m$. For path $r_1 \to r_2 \to ... \to r_{k-1} \to r_k$, there are matchings $r_1 - r'_2$, $r_2 - r'_3$,...,$r_{k-1} - r'_k$.Once a path is found in the grid, the covered marked cells will be uncovered, hence no covered marked cell will appear in two paths. Hence no edges in the Bipartite Graph will share a vertex. So the set of paths can be transformed to the matchings of Bipartite Graph. Conversely, if there are matchings $r_i - r'_{i+1}$, $r_{i+1} - r'_{i+2}$,...,$r_k - r'_{k+1}$, where r'_i and r_{k+1} have no matchings, it can be transformed into a monotone path of the grid. $r_i - r'_{i+1}$ exists means that vertex's corresponding marked cell r_i can reach r_{i+1}. And similarly, r_{i+1} can reach r_{i+2} and so on. Hence the path $r_i \to r_{i+1} \to r_{i+2} \to ... \to r_k \to r_{k+1}$ can be formed in the grid. Moreover, r_i represents the first marked cell the path goes through and r_{k+1} represents the last marked cell the path goes through. In addition, since no edges share one vertex, the paths will not intersect. Hence the feasible matchings of the Bipartite Graph are just a set of the monotone paths of the grid.
Moreover, the vertices unmatched on the left side of the Bipartite Graph are the ending marked cells of a specific path. Suppose the unmatched vertex v_i is not the ending point of paths, then its corresponding marked cell has successive cell v_j' in the path, which implies that there should be edge between v_i and v_j'. It's a contradiction. In addition, the endpoints of paths can't be the matched vertices of the left side. Since the endpoints have no successive points in paths and it implies that no flow will pass through these vertices in the Graph. Moreover, each endpoint in paths corresponds to a specific path since no path will intersect. So the number of unmatched vertices equals to the number of paths in the grid. The problem is to find the minimum set of paths, and it can be reduced to find a matchings of Bipartite Graph that makes the unmatched vertices' number is smallest. Then it's just the maximum matchings in Bipartite Graphs problem. This problem can be reduced to a max-flow problem as the notes have shown.
After the maximum matching is found, goes from the unmatch vertex v_i, we can find its "father" vertex by searching v_i' matching vertex. Then again and again, the path can be found. In conclusion, the algorithm is correct. □

*Analysis.*For the time cost, the algorithm first reduce the problem into a maximum matchings in Bipartite Graph problem by mapping the marked cells to the vertices. Its time cost is $O(n^2)$. And then reduce the Bipartite Graph problem into the max-flow problem. The time needed to create the graph is $O(E)$. Computing the maximum flow is $O(VE)$ using off-the-shelf Ford-Fulkerson algorithm. Finally, finding the paths just goes through the vertices, so it costs $O(V)$. Since $O(V) = O(n^2)$, the total time complexity is $O(VE + E + n^2 + V) = O(VE)$. For the space complexity, the major cost is

on the Graph $G = (E, V)$. It should be $O(E + V)$. Here in the algorithm, it's represented by $C[][]$, hence the space cost is $O(V^2)$.

Question 5: Nested Boxes ..... **[20]**

Solve Question 3.

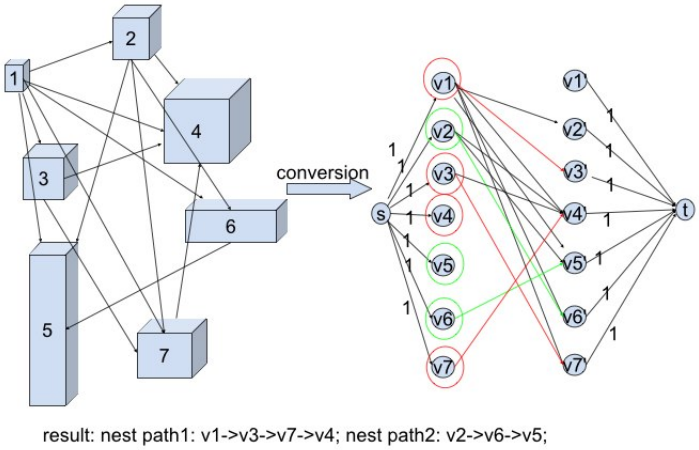

Figure 6

Solution:

Algorithm 6: Best Way to Nest Boxes

Require: $Height[1..n]$, $Width[1..n]$ and $Depth[1..n]$. $Height[\,]$ stores the boxes' heights, $Width[\,]$ stores the boxes' widths and $Depth[\,]$ stores the boxes' depths;

Ensure: A strategy to nest the boxes s.t., the visible boxes number is smallest;

%create graph $G = (E, V)$;
consider each boxes as vertexes $v_1, v_2, ..., v_n \in V$;
divide each vertex v_i into two vertices - v_i and v_i': $V \leftarrow V + \{v_1', v_2', ..., v_n'\}$;
for $i = 1$ to n **do**
  for $j = 1$ to n **do**
    if $Height[i] < Height[j]$ and $Width[i] < Width[j]$ and $Depth[i] < Depth[j]$ **then**
      add edge from v_i to v_j' with capacity 1 to E;
    end if
  end for
end for
add a source vertex s to V and add edges from s to $v_1, v_2, ..., v_n$ with capacity 1 to E;
add a target vertex t to V and add edges from $v_1', v_2', ..., v_n'$ to t with capacity 1 to E;
% run augmented-path algorithm on G
if fail to find a feasible flow f in $G = (E, V)$ **then**
  return "all boxes are visible!";
end if

```
if find a feasible flow f in G then
   create the residual graph G' of G = (E, V, f) ;
   while success to find a feasible augmented path f' = s → r_1 → r_2 → ... → r_k → t in G' do
      G ← (E, V, f + f');
      create the residual graph G' of G;
   end while
end if
% find the strategy to nest boxes
for i = 1 to n do
   if there is no flow leaving v_i then
      while there is a flow leaving v_j and flowing into v'_i do
         output "put box j into box i";
      end while
      output "box i is visible";
   end if
end for
```

EXPLANATION: As the example in Figure 6 shows, the algorithm converts the problem to a bipartite matching problem. All the seven boxes correspond to two vertices in the new graph. For example, box 1 corresponds to vertices v_1 and vertices v'_1. If one box can be nested into another, then, the algorithm adds an edge between them. For example, box 1 can be nested in all others, so there are edges from v_1 to $v'_2, v'_3, .., v'_7$. All the edges have capacity 1. Then the algorithm runs max-flow algorithm on the new graph and figures out the nest strategies using the same method as problem 4.p

Proof. Firstly, the boxes can be correctly nested if and only if there is a feasible flow in G. A feasible flow in G can be transformed into a correct strategy of nesting boxes. If there is a flow $s \to v_i \to v'_j \to t$, it implies that box i can be nested into box j. And we just place box i into box j. Since the edges' capacities are integers, the flows of the maximum flow produced by augmented-path algorithm are all integers. Since the edges leaving s have capacities 1, they can only be saturated or be empty. Hence vertices $v_1, v_2, ..., v_n$ can only have flow leaving them. It implies that the box can't be placed in more than one boxes(if box i is placed in box j and box j is placed in k, it doesn't mean box i is placed in box k) In addition, the edges to the target t have capacities 1 as well, so $v'_1, v'_2, ..., v'_n$ can have at most one flow entering them and it implies that the box can't contain more than two boxes at one time, hence this strategy is correct. Conversely, a correct strategy of placing boxes can be transformed into a feasible unit flow in G. If box i is placed in box j, then it can be transformed to a flow from v_i to v'_j. After that, box i can't be placed in other boxes, so no other flows will leave v_i. And box j can't directly contain other boxes(since the edges of boxes are between 10cm - 20cm), so there is no other flow entering v'_j. Hence the flow is feasible.
Then, the vertices that no flow passes through correspond to the visible boxes. Suppose there is a vertex v_i that no flow leaves it and its corresponding box i isn't visible. Since a flow represents a strategy of placing boxes as shown above, box i isn't visible means box i is placed in other boxes. So there is a flow leaves v_i and it's a contradiction. Hence the number of visible boxes m equals to the the number of such vertices and it equals to the number of unsaturated edges leaving s. It's obviously that the flow size $|f|$ is the number of saturated edges leaving s. Hence $m = n - |f|$. Our goal is to minimize m, i.e., figure out the max flow size $|f*|$.
After the the maximum flow algorithm, the flows between $v_1, v_2, .., v_n$ and $v'_1, v'_2, ..., v'_n$ is just the answer we need. If there is a flow from v_i to v'_j, it means box i should placed in box j in the best strategy. In conclusion, our algorithm is true. □

*Analysis.*The algorithm reduces the problem into a max flow problem and it costs $O(E + V)$ to create the Graph. Since the capacities are all 1, Ford-Fulkerson Algorithm is fast enough and its time cost is $O(E|f*|) = O(EV)$. Finally, the algorithm finds the strategy of placing boxes using

DFS and it costs $O(V)$. Hence the total time cost is $O(E + V + EV + V) = O(EV) = O(En)$. For the space cost, the major cost is Graph $G = (E, V)$ and its residual Graph G'. Hence space cost is $O(E + V) = O(n + E)$.