Lab08-Graph Exploration

CS214-Algorithm and Complexity, Xiaofeng Gao & Lei Wang, Spring 2021.

$*$ If there is any problem, please contact TA Yihao Xie.

$*$ Name: Zilong Li Student ID: 518070910095 Email: logcreative-lzl@sjtu.edu.cn

1. Given an undirected graph $G = (V, E)$. Prove the following propositions.

   (a) Let e be a maximum-weight edge on some cycle of connected graph $G = (V, E)$. Then there is a minimum spanning tree of G that does not include e. Moreover, there is no minimum spanning tree of G that includes e if e is the unique maximum-weight edge on the cycle.

   Proposition 1. *There is a minimum spanning tree of G that does not include e.*

   Proof of Proposition 1. Suppose $e : v_0 \leftrightarrow v_1$ is on the cycle

$$v_0 \to v_1 \to \cdots \to v_n \to v_0$$

where $n \geq 1$ because there has at least one edge e. If there is a minimum spanning tree $T = (V, E_T)$ that include e, consider the graph $(V, E_T - \{e\})$, either v_0 or v_1 is an isolated vertex. Otherwise, v_0 and v_1 are connected in $G' = (V, E_T - \{e\})$ by a path l, with the addition of $\{e\}$ to form the original graph $(V, E_T - \{e\} + \{e\}) = T$, there must be a loop:

$$v_0 \xleftrightarrow{e} v_1 \xleftrightarrow{l} v_0$$

which conflicts the definition of the tree T.

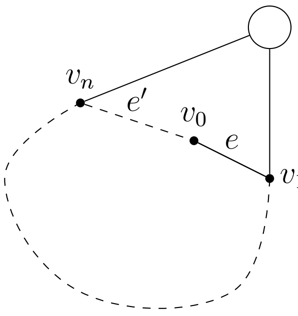

Figure 1: Diagram for Problem (a)

Without the loss of generality, suppose v_0 is the isolated vertex, shown in Figure 1. Then consider the edge $e' : v_n \leftrightarrow v_0$ on the loop, $T' = (V, E_T - \{e\} + \{e'\})$ is a spanning tree. Because v_n is connected in $G' = (V, E_T - \{e\})$ because T is a spanning tree and the loss of $\{e\}$ could only damage the connectivity of v_0. The operation will reconnect v_0. Due to the maximality of weight for e on the loop,

$$|e'| \leq |e|$$

Thus, the new spanning tree will also be a minimum spanning tree derived from

$$\sum_{e_i \in E_T - \{e\}} |e_i| + |e'| \leq |e| + \sum_{e_i \in E_T - \{e\}} |e_i|$$

where $T' = (V, E_T - \{e\} + \{e'\})$ does not include e.

$\square$

Proposition 2. *There is no minimum spanning tree that includes e if such weight on e is unique.*

Proof of Proposition 2. Proof by contradiction. Suppose the minimum spanning tree T contains e, then by the same deduction in Proposition 1, the weight of $e' : v_n \leftrightarrow v_0$ is strictly smaller than that of e.

$$|e'| < |e|$$

Thus, the new spanning tree $T' = (V, E - \{e\} + \{e'\})$ will get a smaller overall weight of edges.

$$\sum_{e_i \in E_T - \{e\}} |e_i| + |e'| < |e| + \sum_{e_i \in E_T - \{e\}} |e_i|$$

which conflicts the minimality of T. Therefore, the minimum spanning tree does not include e in this scenario. □

(b) Let T and T' are two different minimum spanning trees of G. Then T' can be obtained from T by repeatly substitute one edge in $T \backslash T'$ by one edge in $T' \backslash T$ and meanwhile the result after each subsitution is still a minimum spanning tree.

Proof. Because T and T' share the same set of vertices V of G, the property of spanning tree will guarantee that

$$|E_T| = |E_{T'}| = |V| - 1 \tag{1}$$

Because T and T' are different, $T \backslash T', T' \backslash T \neq \varnothing$. The quantity relationship

$$|E_T| = |E_T \cap E_{T'}| + |E_T \backslash E_{T'}|$$
$$|E_{T'}| = |E_T \cap E_{T'}| + |E_{T'} \backslash E_T|$$

with Equation (1) follows that

$$|E_T \backslash E_{T'}| = |E_{T'} \backslash E_T| \neq 0 \tag{2}$$

Thus, there always exists $e \in T \backslash T'$, where $e : v_0 \leftrightarrow v_1$. Because $e \notin T' \backslash T$ and there is no loop in T, suppose v_0 is isolated in $(V, E_T - \{e\})$. $T' \backslash T$ must come up with a path $l \subseteq E_{T'} \backslash E_T$ to connect $v_0 \xleftrightarrow{l} v_1$. As a consequence, there is a loop in G.

$$v_0 \xleftrightarrow{e} v_1 \xleftrightarrow{l} v_0$$

Similar to the construction in Proposition 1, $e'(v_n \leftrightarrow v_0) \in l$ could be selected.

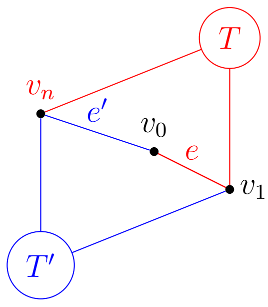

Figure 2: Diagram for Problem (b)

Shown in Figure 2, if $|e'| < |e|$, the new spanning tree $(V, E_T - \{e\} + \{e'\})$ will be even smaller than T, which violates the minimality of T. If $|e'| > |e|$, the spanning tree $(V, E_{T'} - \{e'\} + \{e\})$ will be even smaller than T', which also violates the minimality of T'. Therefore, $|e'| = |e|$. This procedure will not damage the minimality of the new spanning tree and equation (2) guarantees that the subsitution will always exists. □

2. Let $G = (V, E)$ be a connected, undirected graph. Give an $O(|V| + |E|)$-time algorithm to compute a path in G that traverses each edge in E exactly once in each direction. Describe how you can find your way out of a maze if you are given enough coins to apply your algorithm.

   Solution. The implementation is shown in Algorithm 1 to traverse each edge in E exactly once in each direction.

   Explanation. The modified DFS shown in Function `DFS` will record the previous point of the process. After starting from the edge, the starting vertex will be recorded as visited in Line **2**. After leaving the vertex, the ending vertex will also be recorded as visited in Line **14**. Because the process is trigged from e, the early turnback on edge e is not allowed in Line **5**. If the adjacent vertex is visited before but the edge is not, then the path will make a loop to traverse exactly both directions of the edge in Line **9**. If it is not visited, the DFS process will start from the unvisited adjacent vertex in Line **12**. An executing example is shown in Figure 3.

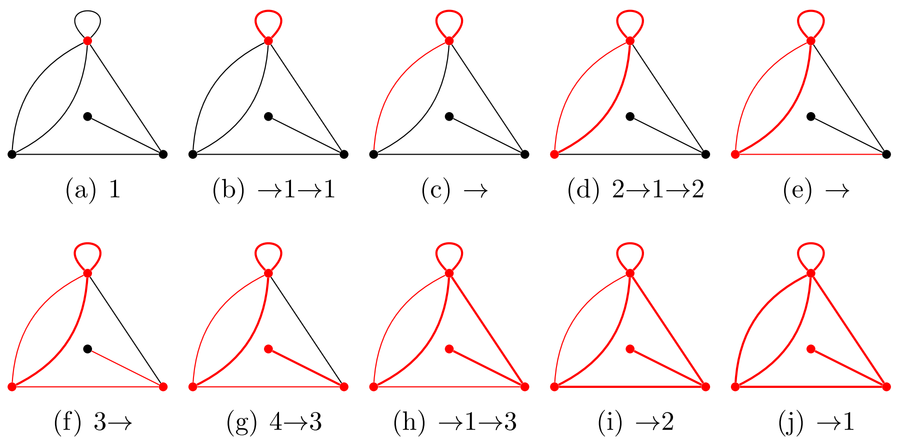

Figure 3: An excuting example.

   Correctness. It uses a *second* matrix to store the edge that has been visited, which makes sure that if the edge is visited twice, it can not be visited again. If it is in the procedure of visiting e, it can not be visited backwards until the local DFS is finished. DFS has been proved that every connected vertices will be visited in the connected G. If there is an edge e' that hasn't been visited after the traverse, where the edge e' connects v and v'. Then v must been visited due to a connected graph, the edge connect to it must be visited if $second[e]$ is false. And initially, all edge is set to false. A contradiction indicates that all edges are visited.

   Time Complexity. The time complexity is analyzed as follows:

   Initialization. Clearing the vertex visited array $first[v]$ costs $O(|V|)$. Clearing the edge visited matrix $second[e]$ costs $O(|E|)$.

   DFS. The depth-first search has a running time of $O(|V| + |E|)$, which is shown in Function `DFS`.

   As a result, the total running time is $O(|V| + |E|)$.

   To find a way out of a maze, the use of optional parameter f in `DFS` is necessary. When the path hits the finish point, the whole `DFS` process will stop and return the path to the finish point. The implementation is shown in Algorithm 2. □

```
Function DFS($G$,$s$,$t$,$e$=NULL,$f$ =NULL)
  Data: Graph $G$, the source vertex $s$, the target vertex $t$, the optional finsh point $f$
  Output: Path segement $l$ from and back to $s$, or the path to $f$
  $l \leftarrow (s)$;
  $first[s] \leftarrow true$;
  foreach $e_i$ in $adj(t)$ do
      $v \leftarrow$ the other end of $e$;
      if $e_i = e$ then continue;
      if $v = f$ then return $l \leftarrow (l \overset{e}{\to} t \overset{e_i}{\to} f)$;
      if $first[v] = true$ then
          if $second[e_i] = false$ then
              $l \leftarrow (l \overset{e_i}{\to} v \overset{e_i}{\to} t)$;
              $second[e_i] \leftarrow true$;
      else
          $l \leftarrow (l \overset{e}{\to}$DFS $(t,v,e_i))$;
          if end of $l$=$f$ then return $l$ ;
  $first[t] \leftarrow true$;
  $l \leftarrow (l \overset{e}{\to} s)$;
  return $l$;
```

```
Algorithm 1: Traverse each edge in both direction
  Input: Graph $G = (V, E)$
  Output: The path that traverses every edge in $G$ exactly once in each direction
  foreach $v \in V$ do
      $first[v] \leftarrow false$;
  foreach $e \in E$ do
      $second[e] \leftarrow false$;
  choose an edge $v_0 \in V$ randomly;
  return DFS $(G, v_0, v_0, NULL)$;
```

```
Algorithm 2: Find a way out of a maze
  Input: Maze $G = (V, E)$, Begin $b$, Finish $f$
  Output: The path $l$ from $b$ to $f$
  foreach $v \in V$ do
      $first[v] \leftarrow false$;
  foreach $e \in E$ do
      $second[e] \leftarrow false$;
  return DFS $(G, b, b, NULL, f)$;
```

3. Consider the maze shown in Figure 4. The black blocks in the figure are blocks that can not be passed through. Suppose the block are explored in the order of right, down, left and up. That is, to go to the next block from (X, Y), we always explore $(X, Y + 1)$ first, and then $(X + 1, Y)$,$(X, Y - 1)$ and$(X - 1, Y)$ at last. Answer the following subquestions:

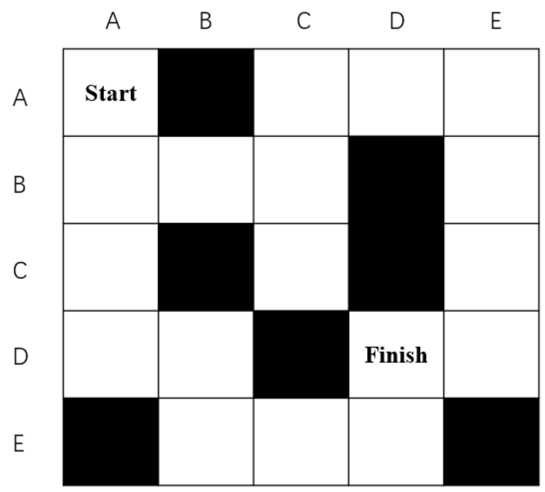

Figure 4: The blocks in the maze.

(a) Give the sequence of the blocks explored by using DFS to find a path from the "start" to the "finish".

Solution. The DFS order is:

AA, AB, BB, CB, CC, CA, DA, EA, EB, EC, ED, DD

which is shown in Figure 5.
And the path found is:

AA, AB, BB, CB, CA, DA, EA, EB, EC, ED, DD

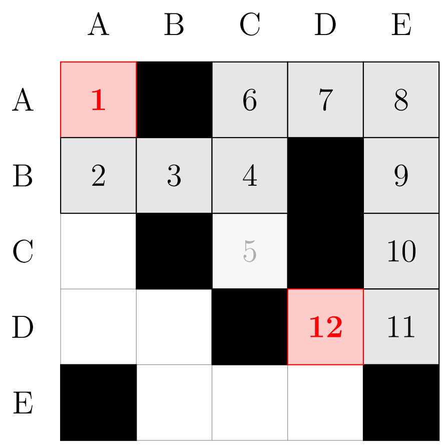

Figure 5: DFS Explored Sequence

□

(b) Give the sequence of the blocks explored by using BFS to find the shortest path from the “start” to the “finish”.

Solution. The BFS order is:

AA, AB, BB, AC, CB, AD, CC, CA, BD, DA, BE, EA, CE, EB, DE, EC, DD

which is shown in Figure 6.
And the shortest path is:

AA, AB, AC, AD, BD, BE, CE, DE, DD

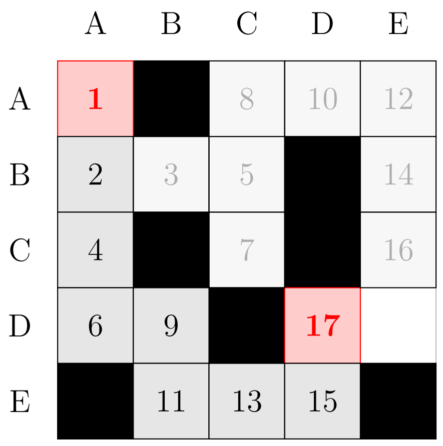

Figure 6: BFS Explored Sequence

□

(c) Consider a maze with a larger size. Discuss which of BFS and DFS will be used to find one path and which will be used to find the shortest path from the start block to the finish block.

Solution. DFS will be used to find one path. DFS could reach every connected node, which is proved in the lecture. Then, DFS will certainly find a path that reaches the **Finish** from **Start**.

BFS will be used to find the shortest path. When EJECT happens, BFS will INJECT the node unvisited and adjacent to the ejected node with 1 increment on the distance.
By mathematical induction, the node with distance d will be ejected and the following nodes with distance $d + 1$ will be injected, with the basic step starting from $d = 0$ where there is only the **Start** in the queue. Figure 7 shows the process.
Then nodes are ordered by the distance to INJECT to the queue and EJECT from the queue. Then before **Finish** is INJECT to the queue, there is no shorter path to **Finish**. And after **Finsh** is EJECT from the queue, the path is always farther than the path when injected.
Thus, BFS will find the shortest path from **Start** to **Finish**.

□

4. Given a directed graph G, whose vertices and edges information are introduced in data file “SCC.in”. Please find its number of Strongly Connected Components with respect to the following subquestions.

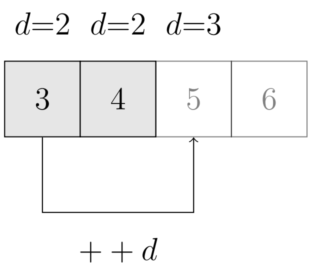

Figure 7: Inject

(a) Read the code and explanations of the provided C/C++ source code "SCC.cpp", and try to complete this implementation.

Solution. The count of Strong Connected Components is

202

□

(b) Visualize the above selected Strongly Connected Components for this graph G. Use the *Gephi* or other software you preferred to draw the graph. (If you feel that the data provided in "SCC.in" is not beautiful, you can also generate your own data with more vertices and edges than G and draw an additional graph. Notice that results of your visualization will be taken into the consideration of Best Lab.)

Solution. The visualization using Gephi is shown in Figure 8.

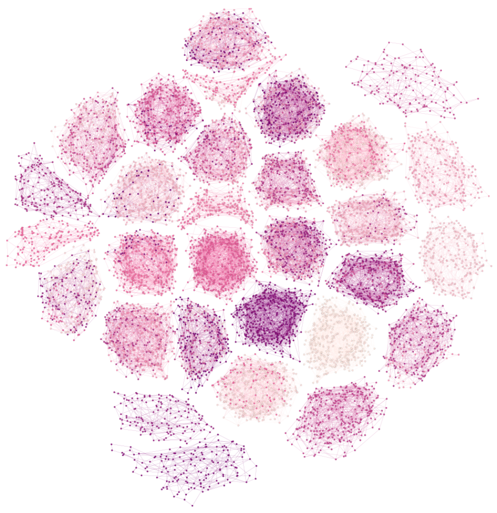

Figure 8: Using Gephi

Additionally, the project that I completed in the Principles and Practice of Problem Solving course is also used to visualize the graph. GraphGenDecomp program visualize the Strong Connected Components (SCC) and all edges in Figure 9. You can get more information about this program on GraphGenDecomp on GitHub, which is implemented in C++ and Fast Light Toolkit (FLTK). Basically, it uses spiral funciton to organize nodes into a centralized form for every sub-graph.

□

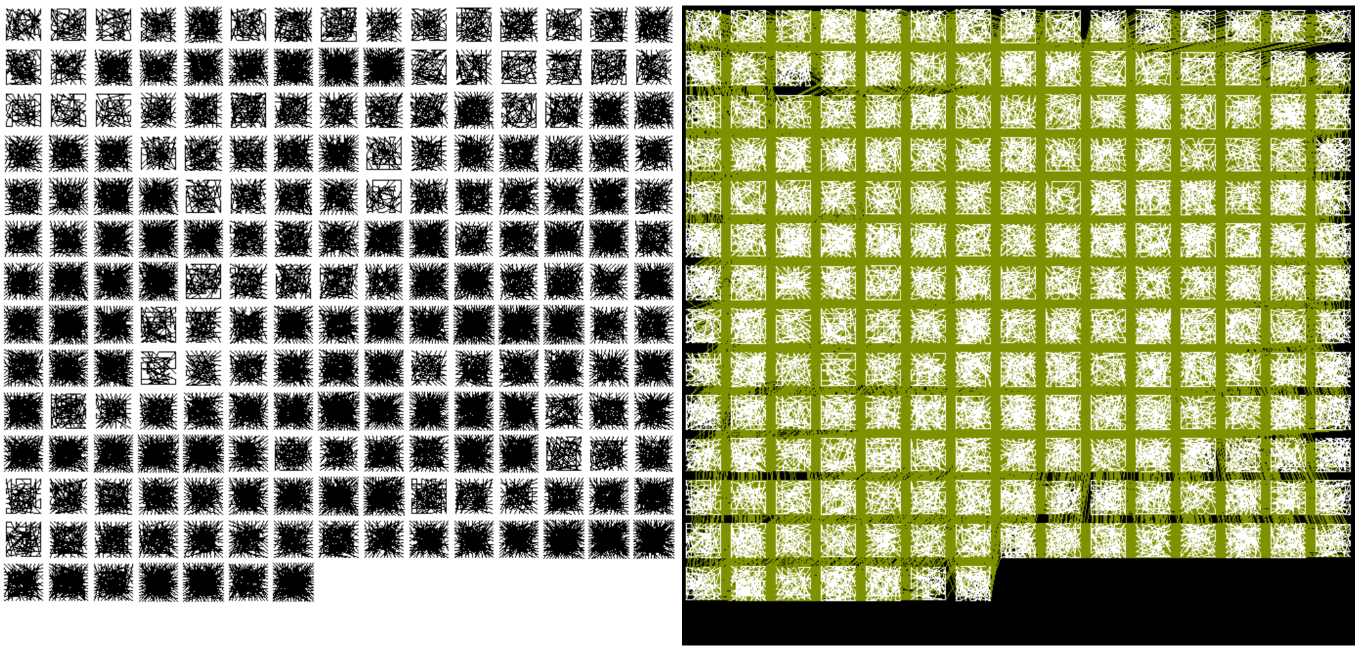
(a) SCC only (b) with crossing edges

Figure 9: Using GraphGenDecomp

A Code for Problem 4

SCC.cpp is the main code for counting and getting the strong connected components. The additional output to `partition.txt` is for the input to GraphGenDecomp.

```cpp
#include <vector>
#include <iostream>
#include <fstream>
using namespace std;
//Please put this source code in the same directory with
      SCC.in
//And do NOT change the file name.
/*
This function computes the number of Strongly Connected
     Components in a graph
Args:
    n: The number of nodes in the graph. The nodes are
         indexed as 0~n-1
    edge: The edges in the graph. For each element (a,b)
         in edge, it means
          there is a directed edge from a to b
          Notice that there may exists multiple edge and
               self-loop
Return:
    The number of strongly connected components in the
         graph.
*/

class Digraph {
public:
    Digraph(int _n) {
        n = _n;
        G = vector<vector<int>>(n);
    }
    void addEdge(pair<int, int> e) {
        G[e.first].push_back(e.second);
    }
    Digraph reverse() {
        Digraph GR(n);
        for (int i = 0; i < n; ++i)
            for (int end : G[i])
                GR.addEdge(make_pair(end, i));
        return GR;
    }
    vector<int> dfs() {
        visited = vector<bool>(n, false);
        visiting = vector<int>();
        for (int i = 0; i < n; ++i)
            if(!visited[i])
                dfs(i);
        return visiting;
    }
    int SCC(vector<int> order) {
        int count = 0;
        visited = vector<bool>(n, false);
        ofstream fscc;
        fscc.open("partition.txt");
        for (int node : order) {
            if (!visited[node]) {
                visiting = vector<int>();
                dfs(node);
                ++count;
                for (int v : visiting)
                    fscc << v << ' ';
                fscc << endl;
            }
        }
        fscc.close();
        return count;
    }
private:
    int n;
    vector<vector<int>> G;
    vector<bool> visited;
    vector<int> visiting;
    void dfs(int node) {
        visited[node] = true;
        for (int adj : G[node])
            if(!visited[adj])
                dfs(adj);
        visiting.push_back(node);
    }
};

int SCC(int n, vector<pair<int,int>>& edge) {
    Digraph G(n);
    for (auto e : edge)
        G.addEdge(e);
    Digraph GR = G.reverse();
    vector<int> visiting = GR.dfs();
    reverse(visiting.begin(), visiting.end());
    return G.SCC(visiting);
}
//Please do NOT modify anything in main(). Thanks!
int main()
{
    int m,n;
    vector<pair<int,int>> edge;
    ifstream fin;
    ofstream fout;
    fin.open("SCC.in");
    cout<<fin.is_open()<<endl;
    fin>>n>>m;
    cout<<n<<" "<<m<<endl;
    int tmp1,tmp2;
    for(int i=0;i<m;i++)
    {
        fin>>tmp1>>tmp2;
        edge.push_back(pair<int,int>(tmp1,tmp2));
    }
    fin.close();
    int ans=SCC(n,edge);
    fout.open("SCC.out");
    fout<<ans<<'\n';
    fout.close();
    return 0;
}
```

`datapre.py` is the data preprocessing code to adapt Gephi and GraphGenDecomp.

```python
import pandas as pd
import os
indata = pd.read_table(os.path.dirname(__file__) + "/
    SCC.in",sep=' ')
adapter=""
for l in indata.index.to_list():
    adapter += '<' + str(indata[indata.columns[0]][l]) +
        ',' + str(indata[indata.columns[1]][l]) + "
        ,1>\n"
with open("main.txt",'w') as file_object:
    file_object.write(adapter)
edges="source,target\n"
for l in indata.index.to_list():
    edges += str(indata[indata.columns[0]][l]) + ',' +
        str(indata[indata.columns[1]][l]) + "\n"
with open("edges.csv",'w') as file_object:
    file_object.write(edges)
nodes="id,label\n"
for i in range(0,int(indata.columns[0])):
    nodes+=str(i)+','+str(i)+'\n'
with open("nodes.csv",'w') as file_object:
    file_object.write(nodes)
```

Remark: Please include your .pdf, .tex, .cpp files for uploading with standard file names.