Lab08-Graph Exploration

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

∗ If there is any problem, please contact TA Yihao Xie.
∗ Name:Yanjie Ze Student ID:519021910706 Email: zeyanjie@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.

   Proof.
   We prove the first proposition **by construction**.
   Denote a minimum spanning tree as T.
   If T does not include e, this satisfies our requirement.
   If T includes e: for the cycle including e, there must exist an edge not included in the minimum spanning tree T, we denote the edge as e_{cycle}.Since e is a maximum-weight edge, e_{cycle} may have a smaller weight than e or the same weight as e.

   - If e_{cycle} has a smaller weight, T must include e_{cycle} and exclude e. Otherwise, T will not be a MST. Therefore, this situation is not possible. This means that if T includes e, e can't be the unique maximum-weight edge.
   - If e_{cycle} has the same weight, we can replace e with e_{cycle}. Thus we construct a minimum spanning tree not including e, which satisfies our requirement.

   Finally, we prove that there is a minimum spanning tree that does not include e.

   We prove the second proposition **by contradiction**.
   Assume the minimum spanning tree T of the graph G includes the edge e, which is an unique maximum-weight edge on some cycle of connected graph $G = (V, E)$.
   Since T is a minimum spanning tree, it will have cycles if we add one more edge of the graph G.
   Since G already has at least one cycle, which we have assumed as the basic setting, we know that: **for this cycle in the graph, it has at least one edge not included in T.** Otherwise, T will have cycles. Let's denote this edge not included as e_{cycle}(in the same cycle with e).
   By the assumption, e is the unique maximum-weight edge, so e_{cycle} has a smaller weight.
   Therefore, **T is not a minimum spanning tree**. Because T can remove e and select e_{cycle}, which both ensures the connectivity and diminishes the sum of the weights.
   However, this contradicts the assumption, which means the assumption is wrong. So there is no minimum spanning tree including e.
   By contradiction, we finish proving the second proposition. □

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

Proof.
First we introduce **Lemma 1** and prove it.

Lemma 1: *For two different minimum spanning tree T, T' of G, $\forall e \in T \backslash T'$, $\exists e' \in T' \backslash T$, $weight(e) = weight(e')$ and e, e' are in the same cycle of G.*

Proof. The relationship of T and T' is shown in Fig. 1. The purple part is the intersection of T and T', which we don't care. The white parts are $T \backslash T'$ and $T' \backslash T$, which must exist because T and T' are different.

We randomly pick one edge in $T \backslash T'$ as e, and our goal is to find e' in $T' \backslash T$.

i. e is in a cycle of G. Otherwise e will be the necessary edge for both T and T', then e will be in $T \cap T'$.

ii. T' also has an edge in this cycle, different from e, denoted as e'. If e' is the same with e, then e will be in $T \cap T'$. If T' doesn't have an edge in this cycle, T' will lose its connectivity. Therefore, we prove the existence of $e' \in T' \backslash T$.

iii. In this cycle, both e and e' can't be the unique maximum-edge, proved by the proposition:*there is no minimum spanning tree of G that includes e if e is the unique maximum-weight edge on the cycle.*

iv. Further, in this cycle, e and e' have the same weight. Otherwise, if $e \in T$ has a smaller weight, T' will certainly select e and discard e'.

Finally, we prove that **Lemma 1** holds.

□

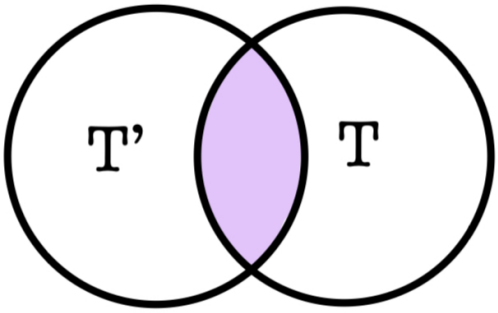

Figure 1: Venn Diagram for MSTs

Since **Lemma 1** holds for both T and T' mutually, this means $|T \backslash T'| = |T' \backslash T|$.

Based on **Lemma 1**, we are able to repeatedly find an edge e in $T \backslash T'$ and replace it with corresponding e' in $T' \backslash T$, until T is equal to T'. □

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.
Define an array $visit_{vertex}$. If $visit_{vertex}(v) = 1$, vertex v has been visited.

Define a matrix $visit_{edge}$. If $visit_{edge}(u, v) = 1$, edge (u, v) has been traversed. For convenience, the two variables are set to be global variables.

The **idea** of the whole algorithm is that **we modify Depth First Search to not only explore the not visited vertexes, but also explore the not visited edges**.

The whole algorithm consists of Alg. 1 and Alg. 2, as shown below.

Algorithm 1: EXPLORE(G,v)

Input: $G = (V, E)$; $v \in V$; $visit_{vertex}, visit_{edge}$;
if *not* $visit_{vertex}(v)$ **then**
    return;
else
    $visit_{vertex}(v) = 1$;
for *edge* $(v, u) \in E$ **do**
    if *not* $visit_{vertex}(v, u)$ **then**
        $visit_{edge}(v, u) = 1$;
        EXPLORE(G, u);

Explanation for Alg. 1: We do exploration based on whether the edge of one vertex has been visited, since our purpose is to traverse all the edges.

There exists one case that the old DFS can't solve: Once a vertex has been visited, there may still exist edges not visited. Therefore, we use $visit_{edge}(u, v)$ as our metric to judge whether to explore.

Algorithm 2: DFS(G)

Input: $G = (V, E)$ is a graph
Ouput: $visit_{edge}(u, v) = 1, \forall edge(u, v) \in E$
for $v \in V$ **do**
    $visit_{vertex}(v) = 0$;
for $edge(u, v) \in E$ **do**
    $visit_{edge}(u, v) = 0$;
for $v \in V$ **do**
    if *not* $visit_{vertex}(v)$ **then**
        EXPLORE(G, v);

□

Then we embed Alg. 1 into **Depth First Search**, to get Alg. 2.

How can we find a way out of maze given enough coins?

We can utilize the idea of Alg. 2.

(a) We use the number of coins to represent the times of points we visit.

(b) While we haven't reached the final:

    i. We now are at a point.

    ii. If the number of coins at this point is equal to the number of available paths, we walk back to the path we come to this point, and go to **iv**.

iii. Else, Choose one available path that has no coins dropped on it. While we are walking on it, we drop coins to mark it.

iv. Then we reach a new point, and drop a coin on the new point.

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:

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

Solution.
The sequence of exploring blocks is :
(A, A)(**Start**) $\rightarrow (B, A) \rightarrow (B, B) \rightarrow (B, C) \rightarrow (C, C)(back\ to\ (B, C)) \rightarrow (A, C) \rightarrow (A, D) \rightarrow (A, E) \rightarrow (B, E) \rightarrow (C, E) \rightarrow (D, E) \rightarrow (D, D)$(**Finish**)

Then the path from the start to the end is the following sequence, which is shown in Fig. 2:
(A, A)(**Start**) $\rightarrow (B, A) \rightarrow (B, B) \rightarrow (B, C) \rightarrow (A, C) \rightarrow (A, D) \rightarrow (A, E) \rightarrow (B, E) \rightarrow (C, E) \rightarrow (D, E) \rightarrow (D, D)$(**Finish**)

□

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

Solution.
The sequence of exploring blocks is:
(A, A)(**Start**) $\rightarrow (B, A) \rightarrow (B, B) \rightarrow (C, A) \rightarrow (B, C) \rightarrow (D, A) \rightarrow (C, C) \rightarrow (A, C) \rightarrow (D, B) \rightarrow (A, D) \rightarrow (E, B) \rightarrow (A, E) \rightarrow (E, C) \rightarrow (B, E) \rightarrow (E, D) \rightarrow (C, E) \rightarrow (D, D)$(**1st path reaches the final**.) $\rightarrow (D, E) \rightarrow (D, D)$(**2nd path reaches the final**)

There exist two paths in total, and the shortest path is the following sequence, as shown in Fig. 3:
(A, A)(**Start**) $\rightarrow (B, A) \rightarrow (C, A) \rightarrow (D, A) \rightarrow (D, B) \rightarrow (E, B) \rightarrow (E, C) \rightarrow (E, D) \rightarrow (D, D)$(**Finish**)

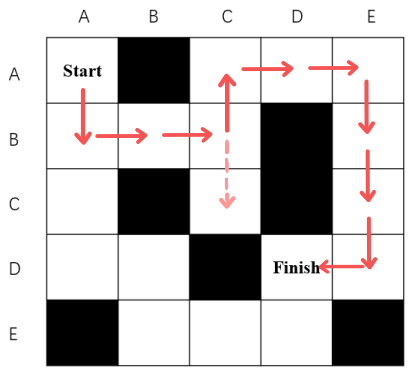

Figure 2: Maze Path by DFS

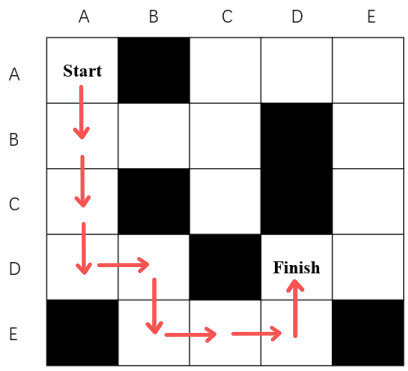

Figure 3: Shortest Path by BFS

□

(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 is used to find one path.
DFS will go deeper and deeper when there still exists an available path in front of it. Once DFS has reached the destination, it will stop and not explore other possible path(in this problem). Although in the worst case DFS will go through all the paths and finally get one available path, using DFS to find a path is still a gooc choice.
BFS is used to find the shortest path. BFS will go through all possible paths. In the end we compare all possible paths and get the shortest path. □

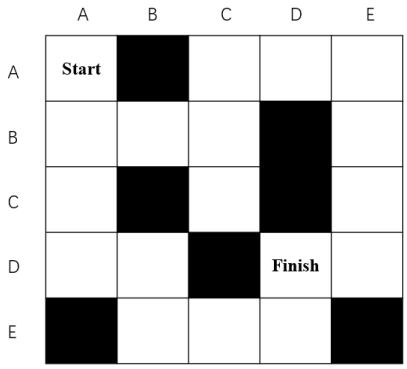

Figure 4: An example of making room for one new element in the set of arrays.

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.

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

Solution.
The full code is in **SCC.cpp**. And the number of Strongly Connected Components is **202**.
In the program, we implement the algorithm following these steps:

Program. SCC Counting

i. Construct an adjacent matrix 10000×10000 based on edges, to represent the graph.
ii. Reverse the matrix.
iii. Run **Depth-first Search** on the matrix and get the post number of each vertex.
iv. Reverse the matrix.
v. Initialize $num_{components} = 0$.
vi. While the graph is not empty:
   A. Find the vertex with the largest post number.
   B. Run DFS starting from this vertex to delete the vertexes in this SCC.
   C. $num_{components} = num_{components} + 1$

vii. Return the number of SCC $num_{components}$.

□

(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 of Strongly Connected Components is shown in Fig. 5, supported by *Gephi*. The larger the component in the graph is, the more vertexes it includes.

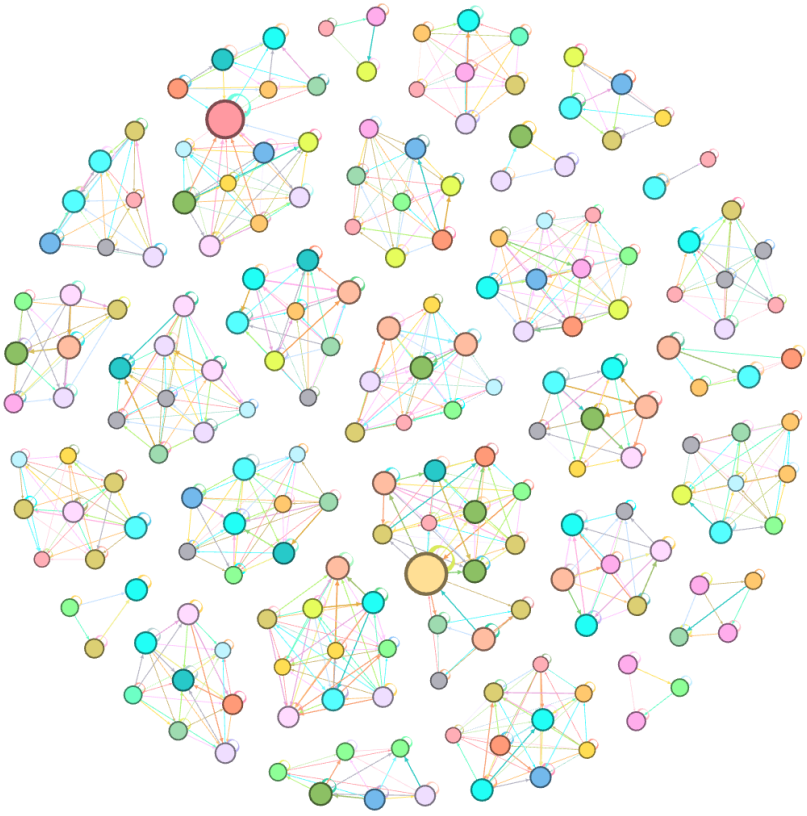

Figure 5: SCC Graph

□

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