Lab06-Graph Exploration

CS214-Algorithm and Complexity, Xiaofeng Gao, Spring 2019.

* If there is any problem, please contact TA Mingran Peng.
* Name:KylinChen Student ID:517030910155 Email: k1017856853@icloud.com

1. Given a graph, find the number of Strongly Connected Components in the graph.

   (a) Complete the implementation in the provided C/C++ source code. Notice that in the source code there will be more detailed explanation.(The source code *SCC.cpp* is attached on the course webpage.)

   (b) Use the *Gephi* to draw the graph. If you think the data provided is not beautiful, you can generate your own data. Notice that result of *Gephi* will be taken into consideration of Best Lab.

   Solution.

   - **(a)** Code file *SCC.cpp* is attached in the .zip file.
   - **(b)** In this problem, we use python to extract data from *scc.in* to generate data.xlsx. Then we can import the .xlsx into *Gephi* to draw the graph. Data-Operation file(main.py), Model file(prob1.gephi), Data file(data.xlsx) is attached in */materials* content.

     One of generated graph can be as Graph. 1.

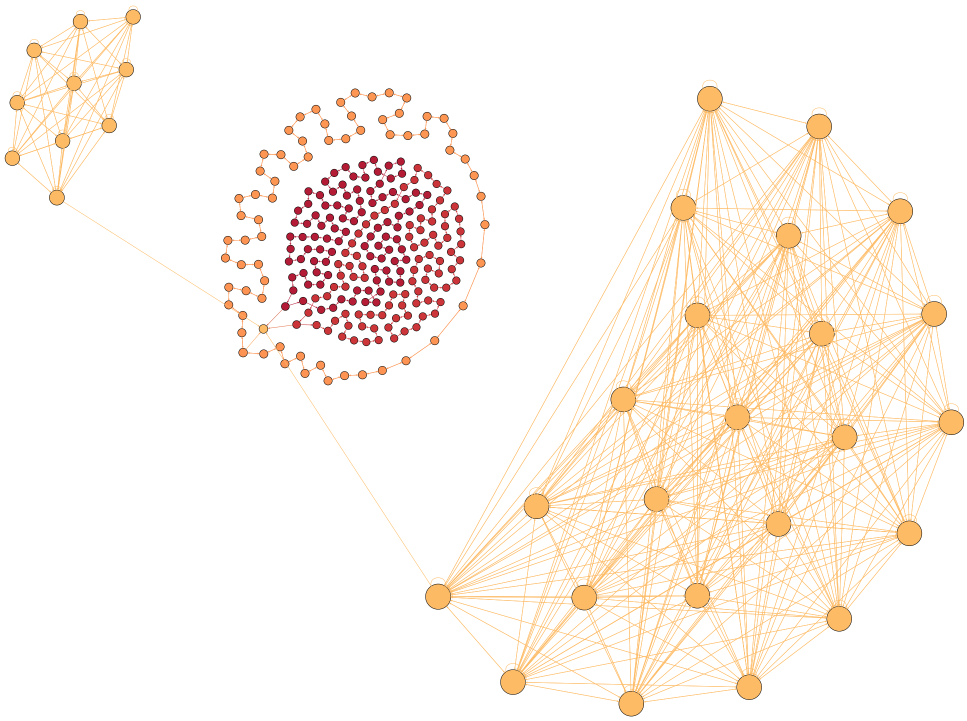

Figure 1: **Generated Graph by Gephi**

2. Remember the lemma introduced in the course: : $\forall u, v \in V$, intervals $[PRE(u), POST(u)]$, $[PRE(v), POST(v)]$ are either disjoint or one is contained within the other.

   Prove the lemma.

Proof. This lemma and its proof relies on the Algorithm. 1 and Algorithm. 2 below:

Algorithm 1: $EXPLORE(G, v)$

Input: $G = (V, E)$ is a graph; $v \in V$
Output: $VISITED(u) = true$ for all nodes u reachable from v

$VISITED(v) = true$;
$PREVISIT(v)$;
for *each* $edge(v, u) \in E$ **do**
  if *not* $VISITED(u)$ **then**
    $EXPLORE(G, u)$;
$POSTVISIT(v)$;

Algorithm 2: $DFS(G)$

Input: $G = (V, E)$ is a graph
Output: $VISITED(v)$ is set to true for all nodes $v \in V$

for *each* $v \in V$ **do**
  $VISITED(v) = false$;
for *each* $v \in V$ **do**
  if *not* $VISITED(v)$ **then**
    $EXPLORE(G, v)$;

We can easily simplify the relationship between point v and u as three model in Graph. 2

Model Explanation:

- According to symmetry, we can define DFS Algorithm access ordering is: $p \rightarrow u \rightarrow v$ or $u \rightarrow v$ (if p doesn't exist).
- Dash line means there exist only one disjoint path (don't have same points in another path or available paths are accessed before) between these two points.
- Full line means there exist path between these two points (a stronger constrain than last one).
- No line means there doesn't exist any path or all available paths are accessed before between these two points.
- p is some arbitrary point besides u and v, if there exist p' have full line from p to u, it means these model(2) is model(1) actually.
- We can easily find that all the relationship between point u and v can be simplified into these three formats and they are matually exclusive.

Now, we will give interval relationship between $[PRE(u), POST(u)]$ and $[PRE(v), POST(v)]$ for every model:

- In **model(1)**, refering to Algorithm. 1, once we call EXPLORE(G, u) it will recurrsively call EXPLORE(G, v), and EXPLORE(G, u) start before EXPLORE(G, v), EXPLORE(G, v) terminate before EXPLORE(G, u), it means $[PRE(v), POST(v)]$ is contained by $[PRE(u), POST(u)]$.

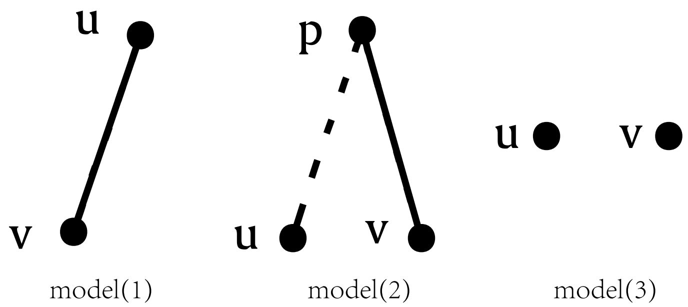

Figure 2: **Simplified Models**

- In **model(2)**, once we call EXPLORE(G, u), it can't recurrsively call EXPLORE(G, v) because there are no available path for it. In this case, refering to Algorithm. 2, we will call DFS(G) to find another explore beginning point. Therefore, EXPLORE(G, u) terminate before we call EXPLORE(G, v), it means [$PRE(v)$, $POST(v)$] is disjoint with [$PRE(u)$, $POST(u)$].
- In **model(3)**, similarly, once we call EXPLORE(G, u), it can't recurrsively call EXPLORE(G, v) because there are no available path for it. In this case, refering to Algorithm. 2, we will call DFS(G) to find another explore beginning point in another connected component. Therefore, EXPLORE(G, u) terminate before we call EXPLORE(G, v), it means [$PRE(v)$, $POST(v)$] is disjoint with [$PRE(u)$, $POST(u)$].

As is mentioned above, $\forall u, v \in V$, intervals $[PRE(u), POST(u)]$, $[PRE(v), POST(v)]$ are either disjoint or one is contained within the other.

3. Consider there is a network consists n computers. For some pairs of computers, a wire exists in the pair, which means these two computers can communicate with delay t.

   Assume that computer s wants to issue a message to computer t, we want to know the minimum time needed to send this message.

   You need to provide the pseudo code and analyze the time complexity.

   Solution. We can use Improved-BFS-Algorithm or Dijkstra-Algorithm to solve this problem, their pseudo code and analyze are follows:

   (1) Improved-BFS-Algorithm:

Algorithm 3: Improved-BFS-Algorithm

Input: $G = (V, E)$ is a graph; $s, t \in V$;
Output: Minimal weighted route from s to t;

```
Res[|V|];
for each i ∈ |V| do
    Res[i] = ∞;
Res[t] ← 0;
Queue → [ ];
enqueue(Queue, s);
while Not empty(Queue) do
    v = dequeue(Queue);
    for each p connect v do
        if Res[v] + t < Res[p] then
            Res[p] = Res[v] + t;
            enqueue(Queue, p);
return Res[t];
```

(2) Dijkstra-Algorithm:

Algorithm 4: Dijkstra-Algorithm

Input: $G = (V, E)$ is a graph; $s, t \in V$;
Output: Minimal weighted route from s to t;

```
C ← {s};
Res[|V|];
for each i ∈ |V| do
    Res[i] = ∞;
Res[t] ← 0;
while V and V − C are connected do
    Tmp ← {};
    for each p ∈ |V − C| which links w in C do
        Tmp ← Tmp ∪ {p};
        Res[p] = MinRes[p], Res[w] + t;
        k = Min(Res[∀ i in Tmp]);
        C ← C ∪ {k};
return Res[t];
```

Complexity Analysis:

We first define the unity time cost as we access an edge or a point.

- **Improved-BFS-Algorithm:** Now that BFS access all the points in the graph and access all the edges(include check unavailable edges), we can find the Time Complexity is $O(|V| + |E|)$.

- **Dijkstra-Algorithm:** Time Complexity relies on the data structure we use. As for row 6 and row 11 in Algorithm. 4. How we find the minimal-distance point determine the time complexity.

  (1) If we use matrix to store the vertex relationship and use brute travel method, for each vertex we will check all the $|V| - 1$ vertices, it means the Time Complexity is $O(|V|^2)$.

  (2) If we use a binary heap to find the minimal-distance vertex for each point in graph, we can transmute the algorithm into BFS and binary heap insertion in each step, which means the Time Complexity is $O((E + N)logN)$.

  (3) If we use fibonacci heap to find the minimal-distance vertex for each point in graph, we can get a optimized Time Complexity $O(NlogN + E)$.