GUÍA DE APRENDIZAJE PARA CONCURSANTES ICPC Y IOI: COMPONENTES FUERTEMENTE CONEXA (*SCC*)

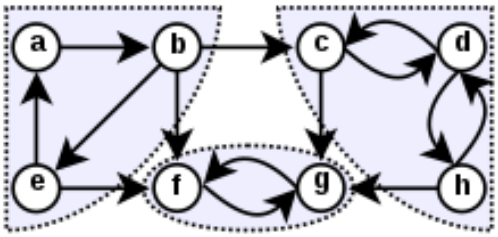
1. Introducción

Tienes un gráfico dirigido G con vértices V y aristas E. Es posible que haya bucles y múltiples aristas. Denotemos n como número de vértices y m como número de aristas en G.

Una **componente fuertemente conexa (*strongly connected component,scc*)** es un subconjunto máximo de vértices C tal que dos vértices cualquiera de este subconjunto son accesibles entre sí, es decir, para cualquier $u, v \in C$:

$$u \mapsto v, v \mapsto u$$

donde $\mapsto$ significa accesibilidad, es decir, existencia del camino desde el primer vértice hasta el segundo.



En grafo se muestra arriba podemos ver que cuenta con 8 nodos y 14 aristas. Dentro del grafo se hallan tres componentes fuertemente conexa las cuales como se observa en la figura están encerradas dentro de lineas discontinuas. De como hallarlas dado el grafo dirigido veremos en la siguiente guía.

2. Conocimientos previos

2.1. Recorrido en profundidad

La búsqueda en profundidad (DFS) es un algoritmo para atravesar o buscar estructuras de datos de árboles o grafos. El algoritmo comienza en el nodo raíz (seleccionando algún nodo arbitrario como nodo raíz en el caso de un grafo) y explora lo más lejos posible a lo largo de cada rama antes de retroceder. Se necesita memoria adicional, generalmente una pila, para realizar un seguimiento de los nodos descubiertos hasta el momento a lo largo de una rama específica, lo que ayuda a retroceder en el grafo.

2.2. Ordenación topológica

Una ordenación topológica (topological sort, topological ordering, topsort o toposort en inglés) de un grafo acíclico dirigido G es una ordenación lineal de todos los nodos de G que satisface que si G contiene la arista dirigida uv entonces el nodo u aparece antes del nodo v. La condición que el grafo no contenga ciclos es importante, ya que no se puede obtener ordenación topológica de grafos que contengan ciclos.

3. Desarrollo

Es obvio que las componentes fuertemente conexa no se cruzan entre sí, es decir, esta es una partición de todos los vértices del grafo. Por lo tanto, podemos dar una definición de **grafo de condensación** G^{SCC} como un grafo que contiene todos los componentes fuertemente conectados como un vértice. Cada vértice del grafo de condensación corresponde a la componente fuertemente conexa del grafo G. Hay una arista orientada entre dos vértices C_i y C_j del grafo de condensación si y solo si hay dos vértices $u \in C_i, v \in C_j$ tales que hay una arista en el grafo inicial, es decir, $(u, v) \in E$.

La propiedad más importante del grafo de condensación es que es acíclico. De hecho, supongamos que hay una arista entre C y C', demostremos que no hay arista entre C' y C. Supongamos que $C' \mapsto C$. Entonces hay dos vértices $u' \in C$ y $v' \in C'$ tal que $v' \mapsto u'$. Pero desde u y u' están en el mismo componente fuertemente conectado, entonces hay un camino entre ellos; lo mismo par v y v'. Como resultado, si unimos estos caminos tenemos que $v \mapsto u$ y al mismo tiempo $u \mapsto v$. Por lo tanto u y v debe estar en la misma componente conectados, por lo que esto es una contradicción. Esto completa la prueba.

3.1. Algoritmo de Tarjan

Es un algoritmo en la teoría de grafos para encontrar los componentes fuertemente conexa (SCC) de un grafo dirigido. Incluyen el algoritmo de Kosaraju y el algoritmo de componente fuerte basado en la ruta. El algoritmo lleva el nombre de su inventor, Robert Tarjan.

La idea básica del algoritmo es esta: una búsqueda de profundidad primero (DFS) comienza desde un nodo de inicio arbitrario (y las búsquedas posteriores de profundidad primero se realizan en cualquier nodo que aún no se haya encontrado). Como de costumbre con la búsqueda de profundidad, la búsqueda visita cada nodo del grafo exactamente una vez, declinando revisar cualquier nodo que ya haya sido visitado. Por lo tanto, la colección de árboles de búsqueda es un bosque de expansión del grafo. Las componentes fuertemente conexas se recuperarán como ciertos subárboles de este bosque. Las raíces de estos subárboles se llaman *raíces* de los componentes fuertemente conexas. Cualquier nodo de una componente fuertemente conexa podría servir como raíz, si es el primer nodo de un componente que descubre la búsqueda.

Los nodos se colocan en una pila en el orden en que se visitan. Cuando la búsqueda de profundidad primero visita un nodo v y sus descendientes, esos nodos no están necesariamente surgidos de la pila cuando esta llamada recursiva regresa. La propiedad invariante crucial es que un nodo permanece en la pila después de que se haya visitado si y solo si existe una ruta en el grafo de entrada a algún nodo antes en la pila. En otras palabras, significa que en el DFS un nodo solo se eliminaría de la pila después de que todas sus rutas conectadas se hayan recorrido. Cuando el DFS retrocede, eliminaría los nodos en una sola ruta y volvería a la raíz para iniciar una nueva ruta.

Al final de la llamada que visita v y sus descendientes, sabemos si v tiene un camino hacia cualquier nodo antes en la pila. Si es así, la llamada regresa, dejando v en la pila para preservar el invariante. Si no, entonces v debe ser la raíz de su componente fuertemente conectado, que consiste en v junto con cualquier nodo más adelante en la pila que v (tales nodos tienen rutas

de regreso a v pero no a ningún nodo anterior, porque si tenían rutas a los nodos anteriores, v también tendría caminos a nodos anteriores que es falso). El componente conectado enraizado en v se mueve desde la pila y se devuelve, preservando nuevamente el invariante.

A cada nodo v se le asigna un entero único $v.index$, que numera los nodos consecutivamente en el orden en que se descubren. También mantiene un valor $v.lowlink$ que representa el índice más pequeño de cualquier nodo en la pila que se sabe que es accesible desde v hasta el subárbol DFS de v, incluido v mismo. Por lo tanto, v debe dejarse en la pila si $v.lowlink < v.index$, mientras que v debe eliminarse como la raíz de un componente fuertemente conectado si $v.lowlink == v.index$. El valor $v.lowlink$ se calcula durante la búsqueda en profundidad desde v, ya que encuentra los nodos a los que se puede acceder desde v.

Si bien no hay nada especial en el orden de los nodos dentro de cada componente fuertemente conxa, una propiedad útil del algoritmo es que no se identificará ningún componente fuertemente conexa antes que cualquiera de sus sucesores. Por lo tanto, el orden en el que se identifican las componentes fuertemente conexas constituye un tipo topológico inverso del DAG formado por las componentes fuertemente conxexas.

3.2. Algoritmo de Kosaraju

El algoritmo descrito fue sugerido de forma independiente por Kosaraju y Sharir en 1979. Este es un algoritmo fácil de implementar basado en dos series de búsqueda en profundidad.

En el primer paso del algoritmo estamos haciendo una secuencia de primeras búsquedas en profundidad, visitando todo el grafo. Comenzamos en cada vértice del grafo y ejecutamos una búsqueda en profundidad desde cada vértice no visitado. Para cada vértice estamos haciendo un seguimiento del tiempo de salida $tout[v]$. Estos tiempos de salida tienen un papel clave en un algoritmo y este papel se expresa en el siguiente teorema.

Primero, hagamos anotaciones: definamos el tiempo de salida $tout[C]$ de la componente fuertemente conexa C como máximo de valores $tout[v]$ Por todos $v \in C$. Además, durante la demostración del teorema mencionaremos los tiempos de entrada $in[v]$ en cada vértice y de la misma manera considerar $tin[C]$ para cada componente fuertemente conectado C como mínimo de valores $tin[v]$ Por todos $v \in C$.

teorema: Dejar C y C' son dos componentes diferentes fuertemente conexa y hay una arista (C, C') en un grafo de condensación entre estos dos vértices. Entonces $out[C] > out[C']$.

Hay dos casos diferentes principales en la prueba, dependiendo de qué componente visitará primero la búsqueda en profundidad primero, es decir, dependiendo de la diferencia entre $tin[C]$ y $tin[C']$:

- El componente C se alcanzó primero. Significa que la búsqueda en profundidad primero viene en algún vértice v de componente C en algún momento, pero todos los demás vértices de los componentes C y C' no fueron visitados todavía. Por condición hay una arista (C, C') en un grafo de condensación, por lo que no solo el componente completo C es accesible desde v pero todo el componente C' es accesible también. Significa que la primera búsqueda en profundidad se ejecuta desde el vértice v visitará todos los vértices de los componentes

C y C', así serán descendientes para v en un primer árbol de búsqueda en profundidad, es decir, para cada vértice $u \in C \cup C', u \neq v$ tenemos eso $tout[v] > tout[u]$, como decíamos.

- Suponga que ese componente C' fue visitado primero. De manera similar, la primera búsqueda en profundidad llega en algún vértice v de componente C' en algún momento, pero todos los demás vértices de los componentes C y C' no fueron visitados todavía. Pero por condición hay una ventaja. (C, C') en el grafo de condensación, por lo tanto, debido a la propiedad acíclica del grafo de condensación, no hay un camino de regreso desde C' a C , es decir, búsqueda en profundidad desde el vértice v no llegará a los vértices de C. Significa que los vértices de C será visitado por profundidad primera búsqueda más tarde, por lo que $tout[C] > tout[C']$. Esto completa la prueba.

El teorema probado es la base del algoritmo para encontrar componentes fuertemente conectados. De ello se deduce que cualquier arista (C, C') en el gráfico de condensación proviene de un componente con un valor mayor de $tout$ al componente con un valor menor.

Si ordenamos todos los vértices $v \in V$ en orden decreciente de su tiempo de salida $tout[v]$ entonces el primer vértice u va a ser un vértice que pertenece al componente *raiz* fuertemente conectado, es decir, un vértice que no tiene aristas entrantes en el gráfico de condensación. Ahora queremos ejecutar dicha búsqueda desde este vértice u de modo que visitará todos los vértices en este componente fuertemente conectado, pero no en otros; al hacerlo, podemos seleccionar gradualmente todos los componentes fuertemente conectados: eliminemos todos los vértices correspondientes al primer componente seleccionado, y luego encontremos un vértice con el mayor valor de $tout$ y ejecutar esta búsqueda desde allí, y así sucesivamente.

Consideremos el grafo transpuesto G^T , es decir, grafo recibido de G invirtiendo la dirección de cada arista. Obviamente, este grafo tendrá las mismas componentes fuertemente conexa que el grafo inicial. Además, el grafo de condensación G^{SCC} también se transpondrá. Significa que no habrá aristas desde nuestro componente *raíz* a otros componentes.

Por lo tanto, para visitar todo el componente *raíz* fuertemente conexo, que contiene el vértice v, es suficiente para ejecutar la búsqueda desde el vértice v en grafo G^T. Esta búsqueda visitará todos los vértices de esta componente fuertemente conexa y solo ellos. Como se mencionó anteriormente, podemos eliminar estos vértices del grafo y encontrar el siguiente vértice con un valor máximo de $tout[v]$ y ejecutar la búsqueda en el grafo transpuesto desde él, y así sucesivamente.

Por lo tanto, construimos el siguiente algoritmo para seleccionar componentes fuertemente conexa:

1. Ejecutar secuencia de profundidad primera búsqueda de grafo G que devolverá vértices con el aumento del tiempo de salida $tout$, es decir, alguna lista $order$.
2. Construir grafo transpuesto G^T . Ejecute una serie de primeras búsquedas en profundidad (amplitud) en el orden determinado por la lista $order$ (para ser exactos en orden inverso, es decir, en orden decreciente de tiempos de salida). Cada conjunto de vértices, alcanzado después de la siguiente búsqueda, será el próximo componente fuertemente conectado.

Finalmente, es apropiado mencionar aquí la ordenación topológica. En primer lugar, el paso 1

del algoritmo representa un tipo de grafo topológico inverso G (en realidad, esto es exactamente lo que significa ordenar los vértices por tiempo de salida). En segundo lugar, el esquema del algoritmo genera componentes fuertemente conectados por orden decreciente de sus tiempos de salida, por lo que genera componentes (vértices del grafo de condensación) en orden de clasificación topológico.

3.3. Algoritmo Gabow

En la teoría de grafos, los componentes fuertemente conexa de un grafo dirigido se pueden encontrar utilizando un algoritmo que usa la búsqueda de profundidad primero en combinación con dos pilas, una para realizar un seguimiento de los vértices en el componente actual y el segundo para realizar un seguimiento de la actual ruta de búsqueda. Purdom (1970), Munro (1971), Dijkstra (1976), Cheriyan y Mehlhorn (1996) y Gabow (2000) han propuesto versiones de este algoritmo (1970), Munro (1971), Dijkstra (1976), Cheriyan y Mehlhorn (1996) y Gabow (2000); De estos, la versión de Dijkstra fue la primera en lograr el tiempo lineal.

El algoritmo realiza una búsqueda de profundidad primero del grafo G dado, manteniendo como hace dos pilas S y P (además de la pila de llamadas normal para una función recursiva). La pila S contiene todos los vértices que aún no se han asignado a una componente fuertemente conexa, en el orden en que la búsqueda de profundidad primero llega a los vértices. La pila P contiene vértices que aún no se ha determinado que pertenezcan a diferentes componentes fuertemente conexa entre sí. También utiliza un contador C del número de vértices alcanzados hasta ahora, que utiliza para calcular los números de pedido de los vértices.

Cuando la búsqueda de profundidad primero llega a un vértice v, el algoritmo realiza los siguientes pasos:

1. Establezca el número de orden de C a v, e incremento C.
2. Empuje v en S y también en P.
3. Para cada arista de v a un vértice vecino w:
   - Si aún no se ha asignado el número de preorden de w (la aristas es una arista de árbol), busque recursivamente w;
   - De lo contrario, si w aún no se ha asignado a una componente fuertemente conexa (la arista es una aristas hacia adelante/posterior/transversal):
     - Elimine vertices repetidamente desde P hasta que el elemento superior de P tenga un número de orden por adelantado menor o igual al número de orden de w.
4. Si v es el elemento superior de P:
   - Elimine vertices de S hasta que v halla sido eleminado y asigne los vértices eliminados a un nuevo componente.
   - Elimine v de P

El algoritmo general consiste en un bucle a través de los vértices del grafo, llamando a esta búsqueda recursiva en cada vértice que aún no tiene un número de pedido asignado.

Al igual que este algoritmo, el algoritmo de componentes fuertemente conectados de Tarjan también usa la primera búsqueda de profundidad junto con una pila para realizar un seguimiento de los vértices que aún no se han asignado a un componente, y mueve estos vértices a un nuevo componente cuando termina expandiendo el vértice final de su componente. Sin embargo, en lugar de la pila P, el algoritmo de Tarjan utiliza un arreglo de números de orden por vértice, asignado en el orden en que los vértices se visitan por primera vez en la búsqueda de profundidad primero. El arreglo de orden se utiliza para realizar un seguimiento de cuándo formar un nuevo componente.

4. Implementación

4.1. C++

4.1.1. Algoritmo de Tarjan

```
struct graph {
   int n; vector<vector<int>> adj;
   graph(int n) : n(n), adj(n) { }
   void add_edge(int src, int dst) { adj[src].push_back(dst);}

   vector<vector<int>> strongly_connected_components() {
      vector<int> open, id(n); vector<vector<int>> scc;
      int t = -n - 1;
      auto argmin = [&](int u, int v){ return id[u] < id[v] ?u:v;};
      function<int(int)> dfs = [&](int u) {
         open.push_back(u); id[u] = t++; int w = u;
         for (int v : adj[u]) {
            if (id[v] == 0) w = argmin(w, dfs(v));
            else if (id[v] < 0) w = argmin(w, v);
         }
         if (w == u) {
            scc.push_back( { });
            while (1) {
               int v = open.back(); open.pop_back();
               id[v] = scc.size(); scc.back().push_back(v);
               if (u == v) break;
            }
         }
         return w;
      };
      for (int u = 0; u < n; ++u) if (id[u] == 0) dfs(u);
      return scc;
        }
};
```

4.1.2. Algoritmo de Kosaraju

```
vector<vector<int>> adj, adj_rev;// G y G'
vector<bool> used;
vector<int> order, component;
int n; //cantidad de nodos

void dfs1(int v) {
   used[v] = true;
   for (auto u : adj[v]) if (!used[u]) dfs1(u);
   order.push_back(v);
}

void dfs2(int v) {
   used[v] = true; component.push_back(v);
   for (auto u : adj_rev[v]) if (!used[u]) dfs2(u);
}

void sccKosaraju() {
   used.assign(n, false);
   for (int i = 0; i < n; i++) if (!used[i]) dfs1(i);
   used.assign(n, false);
   reverse(order.begin(), order.end());

   vector<int> roots(n, 0); vector<int> root_nodes;
   vector<vector<int>> adj_scc(n);
   for (auto v : order)
      if(!used[v]) {
         dfs2(v);
         int root = component.front();
         for (auto u : component)
            roots[u] = root;
         root_nodes.push_back(root); component.clear();
      }

   for (int v = 0; v < n; v++)
      for (auto u : adj[v]) {
         int root_v = roots[v], root_u = roots[u];
         if (root_u != root_v)
            adj_scc[root_v].push_back(root_u);
      }
}
```

Aquí, adj es grafo, adj_rev es grafo transpuesto. La función $dfs1$ implementa la primera búsqueda en profundidad en el grafo G , la función $dfs2$ en el grafo transpuesto G^T. La función $dfs1$ llena la lista $order$ con vértices en orden creciente de sus tiempos de salida (en realidad, se está haciendo un ordenamiento topológico). La función $dfs2$ almacena todos los vértices alcanzados en la lista $component$, que almacenará el siguiente componente fuertemente conectado después

de cada ejecución.

Hemos seleccionado la raíz de cada componente como el primer nodo de su lista. Este nodo representará todo su SCC en el grafo de condensación $roots[v]$ indica el nodo raíz del SCC al que v pertenece el nodo. $root\_nodeses$ la lista de todos los nodos raíz (uno por componente) en el grafo de condensación.

adj_scces la lista de adyacencia del $root\_nodes$. Ahora podemos atravesar adj_scc como nuestro grafo de condensación, usando solo aquellos nodos que pertenecen a $root\_nodes$.

4.1.3. Algoritmo Gabow

```
struct graph {
   int n;
   vector<vector<int>> adj;
   graph(int n) : n(n), adj(n) { }
   void add_edge(int src, int dst) { adj[src].push_back(dst);}

   vector<vector<int>> strongly_connected_components() {
      vector<vector<int>> scc; vector<int> S, B, I(n);
      function<void(int)> dfs = [&](int u) {
         B.push_back(I[u] = S.size()); S.push_back(u);
         for (int v: adj[u]) {
            if (!I[v]) dfs(v);
            else while (I[v] < B.back()) B.pop_back();
         }
         if (I[u] == B.back()) {
            scc.push_back({});
            B.pop_back();
            for (; I[u] < S.size(); S.pop_back()) {
               scc.back().push_back(S.back()); I[S.back()] = n + scc.size();
            }
         }
      };
      for (int u = 0; u < n; ++u) if (!I[u]) dfs(u);
      return scc;
   }
};
```

4.2. Java

4.2.1. Algoritmo de Tarjan

```
private class Graph {
   List<Integer>[] adj;
   List<List<Integer>> components ;
   int[] lowlink; boolean[] used;
   List<Integer> stack; int time;
```

```
   public Graph(int nnodes) {
      adj = new List[nnodes];
      for (int i = 0; i < nnodes; i++) adj[i] = new ArrayList<>();
   }

   private void dfs(int u) {
      lowlink[u] = time++; used[u] = true;
      stack.add(u);
      boolean isComponentRoot = true;

      for (int v : adj[u]) {
         if (!used[v]) dfs(v);
         if (lowlink[u] > lowlink[v]) {
            lowlink[u] = lowlink[v];
            isComponentRoot = false;
         }
      }

      if (isComponentRoot) {
         List<Integer> component = new ArrayList<>();
         while (true) {
            int k = stack.remove(stack.size() - 1);
            component.add(k); lowlink[k] = Integer.MAX_VALUE;
            if (k == u) break;
         }
         components.add(component);
      }
   }

   public List<List<Integer>> scc() {
      int n = adj.length; lowlink = new int[n];
      used = new boolean[n]; stack = new ArrayList<>();
      components = new ArrayList<>();

      for (int u = 0; u < n; u++) if (!used[u]) dfs(u);
      return components;
   }

   public void addEdge(int i, int j) { adj[i].add(j);}
}
```

Una versión del mismo algoritmo pero eliminado la recursividad

```
private class Graph {
   List<Integer>[] adj;
   List<List<Integer>> components ;
   int[] lowlink; int time;
```

```
public Graph(int nnodes) {
   adj = new List[nnodes];
   for (int i = 0; i < nnodes; i++) adj[i] = new ArrayList<>();
}

public List<List<Integer>> scc() {
   int n = adj.length; int[] stack = new int[n];
   int st = 0; int[] stack_cur = new int[n];
   int[] stack_pos = new int[n]; int[] stack_prev = new int[n];
   int[] index = new int[n]; Arrays.fill(index, -1);
   lowlink = new int[n]; time = 0;
   components = new ArrayList<>();

   for (int u = 0; u < n; u++) {
      if (index[u] == -1) {
         int top = 0; stack_cur[top] = u;
         stack_pos[top] = 0; stack_prev[top] = -1;
         while (top >= 0) {
            int cur = stack_cur[top]; int pos = stack_pos[top];
            if (index[cur] == -1) {
               index[cur] = time;lowlink[cur] = time;
               ++time; stack[st++] = cur;
            }
            if (pos < adj[cur].size()) {
               int v = adj[cur].get(pos); ++stack_pos[top];
               if (index[v] == -1) {
                  ++top; stack_cur[top] = v;
                  stack_pos[top] = 0; stack_prev[top] = cur;
               }
               else
                  lowlink[cur] = Math.min(lowlink[cur], lowlink[v]);
            }
            else {
               int prev = stack_prev[top];
               if (prev != -1)
                  lowlink[prev] = Math.min(lowlink[prev], lowlink[cur]);
               if (lowlink[cur] == index[cur]) {
                  List<Integer> component = new ArrayList<>();
                  while (true) {
                     int v = stack[--st]; lowlink[v] = Integer.MAX_VALUE;
                     component.add(v); if (v == cur) break;
                  }
                  components.add(component);
               }
               --top;
            }
         }
      }
   }
       return components;
```

```
    }

    public void addEdge(int i, int j) { adj[i].add(j);}
}
```

4.2.2. Algoritmo de Kosaraju

```
private class Graph {
    List<Integer>[] adj;
    List<List<Integer>> components ;

    public Graph(int nnodes) {
        adj = new List[nnodes];
        for (int i = 0; i < nnodes; i++) adj[i] = new ArrayList<>();
    }

    public List<Integer>[] sccGraph() {
        int[] comp = new int[adj.length];
        for (int i = 0; i < components.size(); i++)
        for (int u : components.get(i)) comp[u] = i;
        List<Integer>[] g =Stream.generate(ArrayList::new).limit(components.size
            ()).toArray(List[]::new);
        Set<Long> edges = new HashSet<>();
        for (int u = 0; u < adj.length; u++)
            for (int v : adj[u])
                if (comp[u] != comp[v] && edges.add(((long) comp[u] << 32) + comp[
                    v]))
                    g[comp[u]].add(comp[v]);
        return g;
    }

    public List<List<Integer>> scc() {
        int n = adj.length; boolean[] used = new boolean[n];
        List<Integer> order = new ArrayList<>();
        for (int i = 0; i < n; i++) if(!used[i])dfs(adj,used,order,i);
        List<Integer>[] reverseGraph = Stream.generate(ArrayList::new).limit(n).
            toArray(List[]::new);
        for (int i = 0; i < n; i++)
            for (int j : adj[i]) reverseGraph[j].add(i);
        components = new ArrayList<>();
        Arrays.fill(used, false); Collections.reverse(order);
        for (int u : order)
        if (!used[u]) {
            List<Integer> component = new ArrayList<>();
            dfs(reverseGraph, used, component, u); components.add(component);
        }
        return components;
    }
```

```
   private void dfs(List<Integer>[] graph, boolean[] used, List<Integer> res,
       int u) {
      used[u] = true;
      for (int v : graph[u]) if (!used[v]) dfs(graph, used, res, v);
      res.add(u);
   }

   public void addEdge(int i, int j) { adj[i].add(j);}
}
```

4.2.3. Algoritmo Gabow

```
private class Graph {
   int nnodes;
   List<Integer>[] adj;
   List<List<Integer>> components;
   int[] I;
   Stack<Integer> S, B;

   public Graph(int nnodes) {
      adj = new List[nnodes]; this.nnodes =nnodes;
      for (int i = 0; i < nnodes; i++) adj[i] = new ArrayList<>();
   }

   public List<List<Integer>> scc() {
      components = new ArrayList<>();
      int n = adj.length; I = new int[n];
      S = new Stack<Integer>(); B = new Stack<Integer>();
      for (int u = 0; u < n; ++u) if (I[u] == 0) dfs(u);
      return components;
   }

   private void dfs(int u) {
      B.add(I[u] = S.size()); S.add(u);
      for (int v: adj[u]) {
         if (I[v]==0) dfs(v);
         else while (I[v] < B.peek()) B.pop();
      }
      if (I[u] == B.peek()) {
         components.add(new ArrayList<Integer>());
         B.pop();
         for (; I[u] < S.size(); S.pop()) {
            components.get(components.size()-1).add(S.peek());
            I[S.peek()] = this.nnodes + components.size();
         }
      }
   }
```

```
    public void addEdge(int i, int j) { adj[i].add(j);}
}
```

5. Complejidad

La complejidad de los tres algoritmos es la misma O($v + e$) siendo v la cantidad de vértices y e la cantidad de aristas pero cuando se analiza la implementación de los tres destaca que Kosaraju y Gabow son más simple que Tarjan. Por lo general Kosaraju y Gabow son los utilizados para hallar las componentes fuertemente conexa.

6. Aplicaciones

Existen un buen número de problemas o ejercicios que su solución radica a partir de saber determinar las componentes fuertementes conexa de grafo dirigido o el grafo de condensación que se forma a partir de estas y trabajar con dicho grafo.

7. Ejercicios propuestos

A continuación una lista de los ejercicios que se puede resolver a partir se ser capaces de determinar las componentes fuertemente conexa en un grafo.

- MOG - Tesoros escondidos
- CSES -Coin Collector
- CSES - Flight Routes Check
- CSES - Planets and Kingdoms
- SPOJ - Ada and Panels
- SPOJ - Capital City
- SPOJ - True Friends
- SPOJ - Good Travels
- SPOJ - Lego
- UVA - 11838 - Come and Go
- UVA 247 - Calling Circles
- UVA 13057 - Prove Them All
- UVA 12645 - Water Supply
- UVA 11770 - Lighting Away

- UVA 12926 - Trouble in Terrorist Town
- UVA 11324 - The Largest Clique
- UVA 11709 - Trust groups
- UVA 12745 - Wishmaster
- Codeforces - Scheme
- Codeforces - Checkposts
- Codechef - Chef and Round Run
- Dev Skills - A Song of Fire and Ice