Belgian Olympiad in Informatics

Weekend Training 28-29 March 2015 — Day 2 and 3

Minimum Spanning Tree, Topological sorting, Eulerian paths and Strongly connected components

Trainer:
Guillaume Derval
Floris Kint
Notes:
François Aubry
Guillaume Derval
Floris Kint

March 28, 2015

Contents

1 Minimum spanning tree

Finding a minimum spanning tree is a problem that frequently occurs in programming contests.

1.1 Introduction

1.1.1 Formal definition

Given a graph G with edges E and vertices V. A spanning tree T of G is a tree that connects all vertices in V. The cost of the spanning tree is the sum of its edge weights. A minimum spanning tree (MST) of G is a spanning tree of G that has a minimal cost. That is, no spanning tree of G exists with a cost lower than the cost of T.

1.1.2 Intuition

Consider the following situation: Given a set of cities and no roads. You want all cities to be reachable from any other city. You don't care about the traveling time, but you do care about the total price of constructing those roads. The price of constructing a road is proportional to the distance between the cities it connects. Determine which roads you need to construct.

The problem above is a typical minimum spanning tree problem. The original graph G has edges between every pair of cities (the roads). The weights of the edges are the costs of the roads they represent. Your task is to select a subset of edges (roads) such that all cities are connected (reachable from every other city). The subset you're trying to find should have a minimal total edge weight (road cost).

1.1.3 Properties

We consider graphs with only positive edge weights.

A minimum spanning tree of a graph G with n vertices, consists of $n - 1$ edges (like any tree).

1.2 Prim's algorithm

A first algorithm that can find a minimum spanning tree of a graph, is called Prim's algorithm. The algorithm is very similar to Dijkstra's algorithm.

1.2.1 The algorithm

The algorithm first takes an arbitrary vertex of the original graph as the current tree. It then repeatedly selects the closest vertex of the graph to the tree. An easy way to implement this is by using a priority queue. The code can be found in Listing 1.

Listing 1: Prim's algorithm

```
1  set<pair<int, int> > prim(int start_node){
2    priority_queue<node> pq;
3    pq.push(node(start_node, 0, start_node));
4    set<int> tree;
5    set<pair<int, int> > edges;
6    while(!pq.empty()){
7      node t = pq.top();
8      pq.pop();
9      if(tree.find(t.index) != tree.end())
10       continue;
11     tree.insert(t.index);
12     if(t.index != t.edge_start)
13       edges.insert(make_pair(t.index, t.edge_start));
14     for(map<int, int>::iterator it = neighbours[t.index].
         begin(); it != neighbours[t.index].end(); ++it){
15       node n(it->first, it->second, t.index);
16       pq.push(n);
17     }
18   }
19   return edges;
20 }
```

1.2.2 Complexity

The algorithm iterates over all neighbours of every node in the graph. This is equivalent to iterating over all edges twice. In each of those iterations, it adds one node to the priority queue. The priority queue can contain $O(V^2)$ nodes, so this operation takes $O(\log V^2) = O(\log V)$ time. Thus, the algorithm runs in $O(E \log V)$.

1.3 Kruskal's algorithm

A second algorithm that can construct a minimum spanning tree, was written by Joseph Kruskal.

1.3.1 The algorithm

Kruskal's algorithm builds a forest (set of trees) which it merges by selecting the shortest available edges.

In the beginning, every node is a separate tree. Next, the algorithm keeps finding the edge with the lowest weight which has not been considered yet. If this edge connects two unlinked trees, the edge is added to the forest. If not, it is discarded. The code can be found in Listing 2.

Listing 2: Kruskal's algorithm

```
vector<edge> kruskal(){
  for(int i = 0; i < N; ++i){
    rank[i] = 1;
    parent[i] = i;
  }
  sort(edges.begin(), edges.end());
  vector<edge> mst_edges;
  for(vector<edge>::iterator it = edges.begin(); it !=
      edges.end(); ++it){
    if(is_same_tree(it->from, it->to)){
      continue;
    }
    merge(it->from, it->to);
    mst_edges.push_back(*it);
  }
  return mst_edges;
}
```

1.3.2 Union-find

Merging two trees and checking whether two nodes are in the same three can easily be done using a Union-Find data structure. The code can be found in Listing 3.

1.3.3 Complexity

The algorithm first sorts all edges ($O(E \log E)$). The algorithm iterates over all edges. For each of them, it checks whether the two endpoints are in the same tree and possibly merges the two trees. The tree a node belongs to can only change $n - 1$ times during the execution of the algorithm. The function *find_parent* runs in $O(1)$ except for when the structure of its tree has changed, in which case the function cannot take more than $O(\log V)$.

Overall this means that the algorithm runs in $O(E \log E) + O(E \log V) = O(E \log V)$.

1.4 Exercises

1. http://uva.onlinejudge.org/external/5/544.html
2. http://uva.onlinejudge.org/external/117/p11710.pdf (Try to use both algorithms)
3. http://uva.onlinejudge.org/external/101/p10147.pdf

Listing 3: Union find

```
1  int find_parent(int a){
2    if(parent[a] == a)
3      return a;
4    parent[a] = find_parent(parent[a]);
5    return parent[a];
6  }
7  bool is_same_tree(int a, int b){
8    return find_parent(a) == find_parent(b);
9  }
10 void merge(int a, int b){
11   int parent_a = find_parent(a);
12   int parent_b = find_parent(b);
13   if(rank[parent_a] < rank[parent_b]){
14     parent[parent_a] = parent_b;
15   }else{
16     parent[parent_b] = parent_a;
17     if(rank[parent_a] == rank[parent_b])
18       rank[parent_a]++;
19   }
20 }
```

2 Topological sorting

2.1 Introduction

2.1.1 An introductory problem

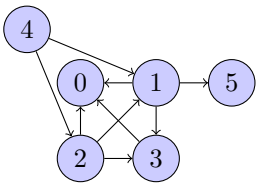
You have a number n of tasks to do, and each task have some dependencies over some other tasks that has to be done before.

If I give you these tasks and the list of their dependencies, can you give me the order in which I have to do the tasks?

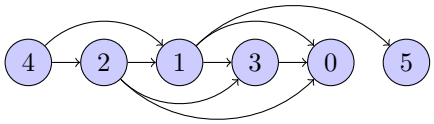
Task	Dependencies
0	1,2,3
1	2,4
2	4
3	1,2
4	-
5	1

This can be viewed as a directed[1] graph:

[1] In this problem, it is also acyclic. Why?



The solution can be derived from a reordering of the vertices:



2.1.2 Definition

A **topological sort** (sometimes called *toposort*) is an ordering of the vertices of a directed graph, such that, if there is an edge linking vertex u to vertex v, we have that u comes before v in the ordering.

2.1.3 Properties

- A topological sort exists for a given graph if and only if there is no directed cycles == if it is a Directed Acyclic Graph (DAG)
- Every DAG has a valid topological sort;
- A given DAG may have more than one toposort (for example, our earlier examples accepts three toposorts);
- If there is a path linking vertices u and v, you have that u comes before v in the toposort.

2.2 Algorithm

The topological sort is in fact the inverse order followed by the DFS **when "closing" the vertices**.

Listing 4: DFS algorithm

```
1  int UNVISITED = 0, OPEN = 1, CLOSED = 2;
2
3  void dfsVisit(LinkedList<Integer>[] adj_list, int node,
       int[] labels)
4  {
5    labels[node] = OPEN;
6    for(int new_node : adj_list[node])
7      if(labels[new_node] == UNVISITED)
8        dfsVisit(adj_list, new_node, labels);
9    labels[node] = CLOSED;
10 }
11
```

```
12 void dfs(LinkedList<Integer>[] adj_list)
13 {
14   int[] labels = new int[adj_list.length];
15   Arrays.fill(labels, UNVISITED);
16   for(int node = 0; node < adj_list.length; node++)
17     if(labels[node] == UNVISITED)
18       dfsVisit(adj_list, node, labels);
19 }
```

When a node u is marked as `CLOSED`, you have that all nodes to which there is a path from u have already been closed; that means that they lies **after** u in the toposort.

When then only have to remember the order in which the nodes are closed, and reverse it. We do that using a stack:

Listing 5: Toposort algorithm

```
1  int UNVISITED = 0, OPEN = 1, CLOSED = 2;
2
3  void dfsVisit(LinkedList<Integer>[] adj_list, int node,
       int[] labels, Stack<Integer> stack)
4  {
5    labels[node] = OPEN;
6    for(int new_node : adj_list[node])
7      if(labels[new_node] == UNVISITED)
8        dfsVisit(adj_list, new_node, labels, stack);
9    labels[node] = CLOSED;
10     stack.push(node);
11 }
12
13 Stack<Integer> toposort(LinkedList<Integer>[] adj_list)
14 {
15   int[] labels = new int[adj_list.length];
16     Stack<Integer> stack = new Stack<Integer>();
17
18   Arrays.fill(labels, UNVISITED);
19   for(int node = 0; node < adj_list.length; node++)
20     if(labels[node] == UNVISITED)
21       dfsVisit(adj_list, node, labels, stack);
22     return stack;
23 }
```

2.3 Exercises

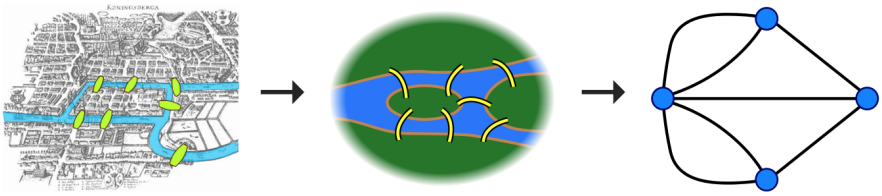
1. Do the problem 200 from UVA: `http://uva.onlinejudge.org/index.php?option=com_onlinejudge&Itemid=8&page=show_problem&category=24&problem=136`
2. Bonus: do the problem 872 from UVA: `http://uva.onlinejudge.org/external/8/872.html`

3 Eulerian paths and tours

3.1 The seven bridges of Königsberg

Königsberg, in Prussia[2], was traversed by a river, creating two small island. They were seven gates that allowed to go from one part of Königsberg to the other.

As the bridges of Königsberg were probably very beautiful, citizens wanted to create a path that crossed each bridge once. An unknown guy, named Leonard Euler, decided to tackle this problem. The figure below[3] shows the bridges of Königsberg.



Euler worked a bit on the problem, and saw that, in fact, he simply could invent graph theory to prove that there was no such path.

3.2 Definitions

- The degree of a node v, noted $\#v$, is the number of edges that are connected to this node. Edges that are loops (that connect a node to itself) count double.
- An Eulerian path[4] in a graph is a path that cross each edge of the graph once.
- An Eulerian tour[5] is an Eulerian path that is also a cycle.

Euler then searched an Eulerian[6] path in Königsberg.

3.3 Properties

After some hours thinking on the problem, he wrote some theorems:

- The sum of the degree for all nodes equals to two times the number of edges (all edges are linked to two nodes).
- A graph contains an Eulerian tour if and only if all nodes have an even degree. [7]

[2] now called Kaliningrad, Russia
[3] Taken from Wikipedia...
[4] Also called Eulerian trail
[5] Also called Eulerian cycle
[6] yep, he was a bit self-centered
[7] Here is the intuition for the demo: when you enter in a node, you "consume" one degree related to the inbound edge, and when you exit a node, you "consume" another degree related to the outbound edge. Thus, you consume two "part of degree" each time you go throughout a node.

- A graph contains an Eulerian path if and only if there is at most two nodes with an odd degree.

3.4 Hierholzer's Algorithm

The Hierholzer's algorithm is used to find Eulerian tours[8] in graph.

It is very simple: you simply start from a random vertex u, then follow one edge to another vertex (mark the edge as used), etc, until you reach again u. You have then formed a cycle that "uses" some of the edges. If there are still edges available on u, repeat the operation, and then merge the cycles, etc. If there are no longer unused edges starting from u, then go to the next vertex in the cycle, and repeat everything.

Listing 6: Hierholzer's Algorithm

```
1  LinkedList<Integer> eulerianCycle(LinkedList<Integer>[]
       adj_list)
2  {
3    //For simplicity, we assume that the graph is connected
         ...
4
5    LinkedList<Integer> cycle = new LinkedList<Integer>();
6    cycle.add(0); //add an initial node
7
8    //Iterate throught the cycle
9    ListIterator<Integer> listIterator = cycle.listIterator
         ();
10     while (listIterator.hasNext())
11     {
12       int new_start = listIterator.next();
13       //If we can begin a new cycle...
14       if(adj_list[new_start].size() != 0)
15       {
16         int current = new_start;
17         int size_added = 0;
18         do
19         {
20           int next_node = adj_list[current].poll();
21
22         //delete the edge in the other direction
23         //this is slow and can be improved using a
               separate Set
24         adj_list[next_node].remove(new Integer(current));
25
26         current = next_node;
27         listIterator.add(current);
28         size_added++;
```

[8]It can be easily expanded to find Eulerian paths: you just have to add an edge between the two nodes that have an odd degree, run the algorithm to produce the Eulerian tour, then delete the added edge!

```
29          } while(current != new_start);
30
31          //go back to the initial node, to verify there is
                no other edges
32          for (int i = 0; i < size_added; i++)
33            listIterator.previous();
34        }
35      }
36      return cycle;
37  }
```

This algorithm runs in linear time, depending only of the number of vertices and edges: $O(|V| + |E|)$.

3.5 Exercises

1. Do the problem 10054 from UVA: `http://uva.onlinejudge.org/external/100/p10054.pdf`. **Warning**: there is lot of input/output, so, in Java, use BufferedReader and BufferedWriter!

2. Bonus: do the problem 10596 from UVA: `http://uva.onlinejudge.org/index.php?option=com_onlinejudge&Itemid=8&category=24&page=show_problem&problem=1537`

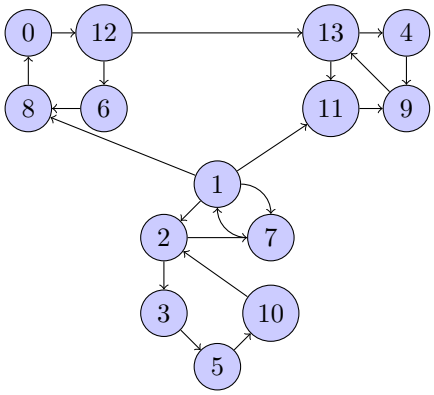
4 Strongly Connected Components

4.1 Introduction

Strongly Connected Components (SCC) are an extension of the concept of connexity for directed graphs.

A strongly connected component is a sub-graph in which there is a (directed) path from each vertex of the component to each vertex of the component.

An example is given in the graph below. There are three strongly connected components.



4.2 Properties

- Inside a SCC, you are sure that there is a path from everywhere to everywhere;
- If you contract each SCCs to a single vertex, the resulting graph is a DAG;
- A transposed graph has the same SCCs as the original one.

4.3 Algorithm

4.3.1 Using topological sort

After having performed the topological sort of the graph, you can use DFS **using topological order on the transposed graph** to find the SCCs.

The intuition is rather simple: the toposort algorithm on the complete graph will more or less give you the toposort of the DAG of the SCCs; transposing the graph will then allow you to only access the current SCC; each DFS call on a node will give you a new SCC.

Listing 7: Toposort+DFS

```
1  void dfsVisit(LinkedList<Integer>[] adj_list, int node,
       int start_node, int[] labels, Stack<Integer> stack)
2  {
3    labels[node] = OPEN;
4    for(int new_node : adj_list[node])
5      if(labels[new_node] == UNVISITED)
6        dfsVisit(adj_list, new_node, start_node, labels,
             stack);
7    //small change to original DFS: instead of CLOSED, mark
         nodes as the "start node"
8    labels[node] = start_node;
9    stack.push(node);
10 }
11
12 LinkedList<Integer>[] transpose(LinkedList<Integer>[] a)
13 {
14   LinkedList<Integer>[] b = new LinkedList[a.length];
15   for (int i = 0; i < b.length; i++)
16     b[i] = new LinkedList<Integer>();
17   for (int i = 0; i < a.length; i++)
18     for (int j: a[i])
19       b[j].add(i);
20   return b;
21 }
22
23 int[] scc(LinkedList<Integer>[] adj_list)
24 {
25   //first, run toposort
26   int[] labels = new int[adj_list.length];
27   Stack<Integer> stack = new Stack<Integer>();
```

```
28
29    Arrays.fill(labels, UNVISITED);
30    for(int node = 0; node < adj_list.length; node++)
31      if(labels[node] == UNVISITED)
32        dfsVisit(adj_list, node, node, labels, stack);
33
34    //run dfsVisit following the toposort order
35    labels = new int[adj_list.length];
36    LinkedList<Integer>[] adj_list_transposed = transpose(
          adj_list);
37    Arrays.fill(labels, UNVISITED);
38    while(!stack.isEmpty())
39    {
40      int node = stack.pop();
41      if(labels[node] == UNVISITED)
42        dfsVisit(adj_list_transposed, node, node, labels,
              stack);
43    }
44
45    //Labels now contains the ids of the SCCs.
46    return labels;
47  }
```

4.4 Tarjan's algorithm

Tarjan's algorithm use a completely different approach, and uses only a single DFS. It is however less simple to explain, to understand and to use in contests.

Listing 8: Toposort+DFS

```
1  int UNVISITED = -1;
2  int[] dfs_low;
3  int[] dfs_num;
4  boolean[] is_in_stack;
5  Stack<Integer> stack;
6  int num;
7  LinkedList< LinkedList<Integer> > sccs;
8
9  void tarjanVisit(LinkedList<Integer>[] adj_list, int node
       )
10 {
11   dfs_low[node] = dfs_num[node] = num;
12   num++;
13   stack.push(node);
14   is_in_stack[node] = true;
15
16   for(int s: adj_list[node])
17   {
18     if(dfs_num[s] == UNVISITED)
19     {
```

```
20          tarjanVisit(adj_list, s);
21          dfs_low[node] = Math.min(dfs_low[node], dfs_low[s])
                ;
22        }
23        else if(is_in_stack[s])
24        {
25          dfs_low[node] = Math.min(dfs_low[node], dfs_num[s])
                ;
26        }
27      }
28
29      if(dfs_low[node] == dfs_num[node])
30      {
31        LinkedList<Integer> scc = new LinkedList<Integer>();
32        int t;
33        do
34        {
35          t = stack.pop();
36          is_in_stack[t] = false;
37          scc.add(t);
38        } while(t != node);
39        sccs.add(scc);
40      }
41    }
42
43    LinkedList< LinkedList<Integer> > tarjan(LinkedList<
        Integer>[] adj_list)
44    {
45      dfs_low = new int[adj_list.length];
46      dfs_num = new int[adj_list.length];
47      is_in_stack = new boolean[adj_list.length];
48      stack = new Stack<Integer>();
49      num = 0;
50      sccs = new LinkedList< LinkedList<Integer> >();
51
52      Arrays.fill(dfs_num, UNVISITED);
53      Arrays.fill(dfs_low, 0);
54      Arrays.fill(is_in_stack, false);
55
56      for (int i = 0; i < dfs_num.length; i++) {
57        if(dfs_num[i] == UNVISITED)
58          tarjanVisit(adj_list, i);
59      }
60      return sccs;
61    }
```

4.5 Exercises

Do the problem 247 from UVA: `http://uva.onlinejudge.org/index.php?option=com_onlinejudge&Itemid=8&category=24&page=show_problem&problem=183`