Graph algorithms: Exploring connectivity

1 The problem formulation

In the previous lectures we already mentioned the following computational problem. Assume that we are given a graph G and two of its vertices s and t. We would like to know if s is connected to t. This is known as *s-t-connectivity problem* for graph G. One can view s-t-connectivity problem as a *maze problem*. Imagine G as a maze; vertices represent gates and edges represent paths. Starting at gate s, we would like to find a path through the gates to the gate t.

For graphs with small number of vertices, s-t-connectivity can be solved easily. For instance, one draws the graph, visually inspects it, and then decides if s and t are connected. How do we decide if s and t are connected when G has a large (say more than million) number of vertices? Here we provide a general method that solves the s-t-connectivity problem. Later, we will refine this method by presenting several algorithms and then provide their implementations. The general method for solving the s-t-connectivity problem is the following:

1. Construct the component of s, that is, the set $C(s)$ consisting of all vertices connected to s.
2. Check if t is in $C(s)$. If t is in $C(s)$ then s and t are connected; otherwise, not.

Obviously, constructing $C(s)$ is crucial. Conceptually, building $C(s)$ is easy. Initially, start with $R = \{s\}$. Each time, when an edge $\{u, v\}$ is found such that $u \in R$ and $v \notin R$, add v to R. Here is a high level description of the algorithm that constructs $C(s)$:

Algorithm 1 BuildComponent(s):

Initially $R = \{s\}$
While there is an edge $\{u, v\}$ such that u is in R and v is not in R
− Add v to R
Endwhile
Output R.

We claim that R, the output of the algorithm, coincides with the component $C(s)$. Clearly, every vertex in R is in $C(s)$. We want to show that every vertex of $C(s)$ belongs to R. So, let us take x from $C(s)$. There is a path P from s to x. The path P is of the form $v_0, v_1, \ldots, v_n$ with $s = v_0$ and $x = v_n$. If x is not in R, then take the first v_i in the path that does not belong to R (in particular, v_i might be equal to x). Then v_{i-1} is in R. Then (v_{i-1}, v_i) is such that $v_{i-1} \in R$ and v_i is not in R. Hence, the algorithm can not output R because of the while loop condition.

The argument above shows that any implementation of Algorithm 1 must output $C(s)$. In the next sections we describe two methods, known as *breadth-first search* (BFS) and *depth-first search* (DFS), that implement Algorithm 1. These two methods produce the component $C(s)$.

2 BFS-algorithm

Let G be a graph and s, t be its vertices. One method of generating the component of s is to implement Algorithm 1 as follows. Build *layers* of vertices L_0, L_1, L_2, ... such that all vertices of layer L_i are at distance exactly i from s. Here is the process of building these layers. The process is called *breadth-first search* or *BFS*-algorithm:

1. The initial layer L_0 consist of only one vertex s, that is, $L_0 = \{s\}$.
2. The layer L_1 consists of *all* vertices v such that $\{s, v\}$ is an edge. Thus, L_1 consist of all vertices at distance exactly 1 from s.
3. Assuming that the layers L_0, L_1, ..., L_i are constructed, we build the layer L_{i+1} as follows. Put a vertex v into L_{i+1} if and only if v has not been put into any of the previous layers and there exists $u \in L_i$ such that $\{u, v\}$ is an edge of the graph.

Observation 1 *A vertex v belongs to one of the layers L_0, L_1, ..., L_i, ... if and only if v is in the component of s. Hence, s and t are connected if and only if t appears in one of L_is.*

Indeed, note that the BFS-algorithm implements Algorithm 1. We already argued that the output of Algorithm 1 is the the component of s. Thus, the observation is clearly true.

Recall that the distance between vertices x to y in G is the length of the shortest path from x to y. We use this concept in the following observation about the layers:

Observation 2 *A vertex x belongs to L_i if and only if the distance from s to x is i.*

Proof. For L_0 and L_1 the observation is true. Now, x is in L_2 if and only if x belongs to neither L_0 nor L_1 and there exists a vertex $u \in L_1$ such that $\{u, x\}$ is an edge. This implies x belongs to L_2 if and only if the distance from s to x is 2. Now consider L_i. Then x is in L_i if and only if x does not belong to any of the previous L_0, L_1, ..., L_{i-1} and there exists a vertex $u \in L_{i-1}$ such that $\{u, x\}$ is an edge. We already know that $u \in L_{i-1}$ if and only if the distance from s to u is $i - 1$. This implies that x belongs to L_i if and only if the distance from s to x is i. This is exactly what we wanted.

A beautiful property of the BFS-algorithm is that it naturally produces a tree T, called a *BFS-tree* of the graph. Here is an informal description of the tree T. The root of the tree is s. Now consider the stage when a vertex v has just been seen by the BFS-algorithm. This happens when some node u in a layer L_i exists such that u has an edge to v and v has not been seen before this stage. We add v to the tree T. So, u becomes a parent of v since u is responsible for the appearance of v; in this way the new edge $\{u, v\}$ is added to the tree T. This process generates a tree because it prohibits cycles.

We now explain, in more detail, one way of generating a BFS-tree. In the first two items below we explain the first two steps, and then a method generating a BFS-tree:

1. The root of the tree is the start node s.

2. Consider all vertices adjacent to s. Let us enumerate all them in some fixed order, say

$$x_{1,1}, x_{1,2}, \ldots, x_{1,\ell_1}.$$

   Declare that these all are the children of the root s. The number of elements at distance 1 from s in G are exactly those at distance 1 from the root of the tree that is being constructed.
3. Now suppose we have constructed the tree such that all the nodes of the tree at distance i from the root s are exactly those from the layer L_i built above. Also, these are the leaves of the tree built so far. We assume we have an enumeration of all these leaves:

$$x_{i,1}, x_{i,2}, \ldots, x_{i,\ell_i}.$$

   For each of $x_{i,1}, x_{i,2}, \ldots, x_{i,\ell_i}$ proceed as follows. Take $x_{i,t}$ (so, t ranges from 1 to ℓ_i). Consider all nodes y that are adjacent to $x_{i,t}$ in the graph G. For each of these y, if y has not appeared in the tree yet, then declare that y is a child of $x_{i,t}$ in the tree.

Exercise 1. Construct two distinct BFS-trees for each of the following graphs: the wheel graph W_5 and the cube graph $Cube_3$.

The BFS-tree satisfies the following properties: (1) Its set of nodes coincides with the component of vertex s; (2) Every edge of the tree is an edge of the graph. In this sense, the BFS-tree *spans* the component of s. In particular, if G is connected than the generated BFS-tree is such that all nodes of the tree coincide with the set of all vertices of G and edges of the tree are also edges of the graph; if G has cycle then the BFS-tree omits at least one edge of the cycle.

Exercise 2. Let x and y be vertices of G such that x is in the layer L_i and y is in L_j. Show that if there is an edge in the graph G between x and y then i and j differ at most by 1.

3 DFS-algorithm

Similar to the breadth-first search algorithm, the DFS-algorithm refines the Algorithm 1. The idea is somewhat orthogonal to the idea of BFS-algorithm. Namely, we start at vertex s, and begin walking along the edges as far as possible until a "dead end"; the "dead end" is a vertex whose neighbours have already been visited during the walk. Once a "dead end" is reached we backtrack to the latest vertex that has an adjacent vertex not visited yet, and we move to that vertex continuing on the process. This is a *depth-first search* algorithm because it visits vertices in the graph as deeply as possible backtracking only after a "dead end" vertex. This idea contrasts with the idea of the BFS-algorithm where vertices of G are visited layers by layers. Here is the DFS-algorithm:

Observation 3 *A vertex v belongs to R if and only if v is in the component of s. Hence, s and t are connected if and only if t is in R.*

Indeed, note that depth-first search implements Algorithm 1. We already argued that the output of the Algorithm 1 is the the component of s. Thus, the observation is true.

Algorithm 2 DFS(u)-algorithm:

Mark vertex u *visited* and add it to R
For each edge $\{u, v\}$
– If v is not marked *visited* then call $DFS(v)$
– Endif
Endfor

The DFS-algorithm visits vertices of the graph in different order than the BFS-algorithm. The DFS-algorithm might go visiting vertices of the graph, sometimes going far away from s before it backtracks. A good example here is the way the BFS-algorithm differs from the DFS-algorithm when applied to complete graphs K_n. The BFS-algorithm applied to K_n produces a BFS-tree of height 1: the root s and its $n-1$ children. In contrast, DFS-algorithm starts with s, visits a vertex, say v_1, then visits the next vertex, say v_2, until it reaches a "dead end", and then goes back to s and stops. Since the graph is complete, the DFS-algorithm produces a tree of height $n-1$; a totally different tree from the BFS-tree.

Similar to the BFS-algorithm, the DFS-algorithm also produces a tree, called a DFS-tree, in a natural way. Here is a construction of a DFS-tree:

Algorithm 3 DFS(s)-tree:

Create a tree with root s
While there is an unvisited vertex v adjacent to s
– Put the edge $\{s, v\}$ into the tree
– Call DFS (v)-tree algorithm
EndWhile

Exercise 3. What are the heights of the BFS-trees for the wheel graphs W_n.

Let $d(x, y)$ be the distance between vertices x and y in graph G. Consider a vertex f such that f and s are connected and $d(s, v) \leq d(s, f)$ for all vertices v connected to s. As we saw in the previous section, the distance $d(s, f)$ equals to the number of layers in the BFS-algorithm. Hence, the height of every BFS-tree of a graph G equals $d(s, f)$. In contrast, the height of any DFS-trees is at least $d(s, f)$.

Exercise 4. Prove that the height of any BFS-tree for graph G starting at x is smaller or equal to the heights of all DFS-trees for G starting at x.

Observation 4 *If x, y are adjacent in G, then in any BFS-tree of G either x is a descendant of y or y is descendant of x.*

Proof. Take a vertex v in G. Consider the time t_1 when DFS-algorithm is invoked on v, and consider the time t_2 when DFS-algorithm finishes processing the vertex v. Between times t_1 and

t_2 all the vertices u that are marked visited are descendants of v in the DFS-tree. Now, assume that x is encountered first by the DFS-algorithm. Then, since $\{x, y\}$ is an edge, the vertex y must be visited between the times t_1 and t_2. Hence, y is a descendant of x in the BFS-tree.

Next our goal is to implement the BFS and DFS algorithms and analyse their running times.

4 Representations of graphs

To analyse running time of graph algorithms, such as BFS and DFS algorithms, we need to represent graphs. These representations will be inputs to graph algorithms, and as such we also need to say a few few words about the input sizes.

For graph algorithms, there are some typical operations that may be performed on graphs. These might include inserting and deleting vertices, as well as inserting and deleting edges to and from various sets. For instance, if a social networking service has a new user, a new vertex must be inserted into the graph, or if Amy does not want to be Bob's friend, then the edge between Amy and Bob must be deleted. Other examples can be taken from the algorithms we have already considered. For instance, we might want to add a vertex to the list of visited vertices in BFS or DFS algorithms; or we might want to know if a vertex has already been processed. In addition, we would like to support these operations efficiently. To do all these, we must carefully consider how we represent graphs depending on types of operations we would like to use.

There are two natural parameters for any graph G: the first parameter is the number of vertices, typically denoted by n; the second parameter is the number of edges denoted by m. Graph representation depend on these n and m. So, we assume the set of vertices of graphs with n elements coincides with the set $\{0, 1, 2, \ldots, n-1\}$.

A simple way to represent a graph G is the *adjacency matrix*. The matrix is $n \times n$ matrix so that the entry $\{u, v\}$ of this matrix is 1 if and only if $\{u, v\}$ is an edge of the graph. The matrix is symmetric in the sense that $A(u, v) = A(v, u)$ for all vertices u and v (since G is undirected). For instance, consider the graph presented in Figure 1.

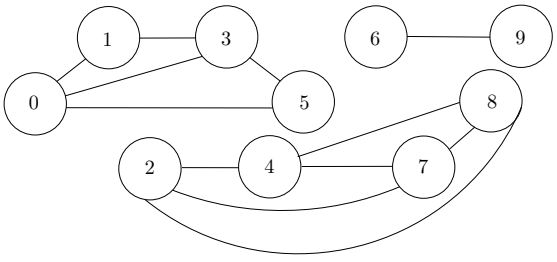

Fig. 1. An example of a graph

The matrix representation of this graph is the following:

$$\begin{pmatrix} 0 & 1 & 0 & 1 & 0 & 1 & 0 & 0 & 0 & 0 \\ 1 & 0 & 0 & 1 & 0 & 0 & 0 & 0 & 0 & 0 \\ 0 & 0 & 0 & 0 & 1 & 0 & 0 & 1 & 1 & 0 \\ 1 & 1 & 0 & 0 & 0 & 1 & 0 & 0 & 0 & 0 \\ 0 & 0 & 1 & 0 & 0 & 0 & 0 & 1 & 1 & 0 \\ 1 & 0 & 0 & 1 & 0 & 0 & 0 & 0 & 0 & 0 \\ 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 1 \\ 0 & 0 & 1 & 0 & 0 & 1 & 0 & 0 & 1 & 0 \\ 0 & 0 & 1 & 0 & 0 & 1 & 0 & 1 & 0 & 0 \\ 0 & 0 & 0 & 0 & 0 & 0 & 1 & 0 & 0 & 0 \end{pmatrix}$$

Exercise 5. Describe adjacency matrix representations for complete graphs K_n, bipartite graphs $K_{n,m}$, wheel graphs W_n, and cycle graphs C_n.

The second way to represent graphs G is known as the *adjacency list*. In this representation for each node u we create a list $Adj(u)$ containing all vertices adjacent to u. So, the length of this list equals the number of all vertices u such that $\{u, v\}$ is an edge of the graph. So, if $\{u, v\}$ is an edge of G then it appears in two lists: in $Adj(u)$ and in $Adj(v)$. So, the adjacency list representation of the graph in Figure 1 is the following:
0 : 1, 3, 5.
1 : 0, 3.
2 : 4, 7, 8.
3 : 0, 5.
4 : 2, 7, 8.
5 : 0, 3.
6 : 9.
7 : 2, 4, 8.
8 : 4, 7.
9 : 6.

Natural question is which of these representations is good? The answer depends on the problem at hand and the tools that are being used for the implementation. However, we point out several differences between these two representations. These differences should be taken into account when implementing graph algorithms.

1. The adjacency matrix representation takes n^2 amount of space. So, the size of the input in the matrix representation equals n^2 when G has n vertices.
2. In the adjacency matrix representation, if we want to know all vertices adjacent to u, then we need to go through the u's row in the matrix, and collect all vertices v connected to u, that is $A(u, v) = 1$. This takes $\Theta(n)$ time.
3. The adjacency list representation requires only $O(n + m)$ space. Indeed, we need to have an array of size n of pointers, each pointer pointing to a list $Adj(v)$. Note that the lengths of these lists varies from vertex to vertex. In adjacency list representation, every edge $\{u, v\}$ appears exactly twice: in the lists $Adj(u)$ and $Adj(v)$. Hence, the total length of all lists equals $2m$. Hence, $(n + m)$ space is needed for adjacency list representation. Therefore the size of the input in this representation is $n + m$.

4. The adjacency list representation is more compact.
5. In adjacency matrix representation, it takes a constant time to find out if $\{u, v\}$ is an edge.
6. In adjacency list representation in takes it $O(degree(u))$ to find out if $\{u, v\}$ is an edge. Recall that the degree of u is the number of vertices adjacent to u.
7. In adjacency list representation, given u, finding a vertex adjacent to u takes a constant time. In adjacency matrix it takes $O(n)$ time.

Exercise 6. Prove that the sum of the degrees of the vertices of any graph G equals $2m$, where m is the number of edges of G.

5 Implementing the BFS-algorithm

The adjacency list representation of graphs is suited for implementing the BFS-algorithm. Let G be a graph given with its adjacency list representation. Our implementation needs to examine edges leaving vertex u. When an edge $\{u, v\}$ leaving u is examined, we need to check if v has been added into a layer before. In order to efficiently do that we need to create an array *Added* of length n (the number of vertices of G), and as soon as v is put into a layer we should set $Added(v) = true$. Initially all values of the array are set to $False$. Once the array is created, deciding if v belongs to the array *Added* is a constant time operation. As in Section 3, our implementation creates layers L_0, L_1, ..., L_i, ... The layers are maintained as lists.

Algorithm 4 BFS(s)-implementation:

```
INPUT graph G in adjacency list, vertex s
OUTPUT BFS tree T with root s
  Create an array Added of length n
  Set Added(s) ← true and Added(v) ← False for all v ≠ s
  Set L_0 to be the list consisting of s
  Set the layer counter i = 0
  Set the current BFS-tree T to consist of the root s
  while L_i is not empty do
     Set L_{i+1} to be empty
     for each vertex u ∈ L_i and each edge {u, v} do
        if Added(v) = False then
           set Added(v) ← True,
           add edge {u, v} to T and
           add u to L_{i+1}
        end if
     end for
     Increment the counter i by one
  end while
  return T
```

Here is now time analysis of the algorithm. We will show that the running time of this BFS-implementation is linear on the size of G:

Fact 1 *The BFS(s)-implementation above, given graph G, vertex s, runs in time proportional to $(n+m)$ with respect to the adjacency list representation of G.*

Proof. Setting up the array *Added* takes time proportional to n. The algorithm creates at most n lists L_0, L_1, ... This takes time proportional to n. For each u and each v executing *the if statement* in the while loop takes a constant time. For each u *the if statement is executed* in time proportional to the degree d_u of u. The total length of lists in the adjacency list representation is $2m$. Hence the total running time over all nodes is in $O(m)$. So, the total time spent for executing the algorithm is $O(n+m)$. We have proved the fact.

Exercise 7. A *queue* data structure maintains a lists of elements. This list is implemented as either linked or doubly linked list data structure. In queue data structure we extract elements in first-in, first-out (FIFO) order: we select elements in the same order in which they were added to the list. Above, we implement the BFS-algorithm using the list of layers L_0, L_1, ..., L_i, ... Implement the BFS-algorithm by using a single list L maintained as a queue. Your implementation should also build a BFS-tree. In addition, explain why the running time of your implementation is $O(n+m)$.

6 Implementing the DFS-algorithm

We will use a *stack* data structure to implement the DFS-algorithm. This data structure maintains a list of elements implemented as a linked or doubly linked list. When we select (delete) an element from this list we take the first element. When we add (insert) an element we also put the element into the first position of the list. Thus, stack is a linked data structure in which we extract elements in last-in, first-out (LIFO) order: every time we select an element, we take the one that was added most recently. Extracting and adding operations are constant time.

To explain our implementation of the DFS-algorithm and the use of the stack data structure, lets assume that the DFS-algorithm is applied to vertex u of graph G. How does DFS-proceed? Well, the DFS-algorithm will examine an adjacent vertex to u. Assume that u has 4 adjacent vertices listed in adjacency list order a_1, a_2, a_3, a_4 and all these vertices have not been visited yet. The algorithm puts this list on stack: first a_1 is put, then a_2, a_3 and finally a_4. So, the the content of stack becomes: a_4, a_3, a_2, a_1. Note that the adjacency list of u is put on stack in reverse order. The algorithm selects a_4, and the DFS-algorithm is now applied to a_4. Assume that a_4 has 5 adjacent vertices all distinct from the previously seen vertices and listed in the adjacency list order: b_1, b_2, b_3, b_4, b_5. So, we put these on stack, and the content of stack now becomes:

$$b_5, b_4, b_3, b_2, b_1, a_3, a_2, a_1.$$

Now the DFS-algorithm runs on b_5. Assume that b_5 has 2 adjacent vertices all distinct from the previously seen nodes and listed the adjacency list order: c_1, c_2. We put these on stack, and the content of stack becomes:

$$c_2, c_1, b_4, b_3, b_2, b_1, a_4, a_3, a_2, a_1.$$

Now the DFS-algorithm is applied to c_2. This continues on. The point is that, in the current state of stack, for the DFS-algorithm to finish its execution at vertex u (from which we started our discussion), the algorithm needs to process the vertices in the sequence:

$$c_2, c_1, b_4, b_3, b_2, b_1, a_3, a_2, a_1.$$

from left-to-right order one-by-one.

The example above gives us a hint on implementation of the DFS-algorithm when it is applied to u. We use stack data structure. We add all vertices adjacent to u to our stack. Then we proceed to explore a new vertex a adjacent to u. As we explore a, we add all vertices adjacent to a to the stack. These vertices that we put on stack will be explored before we return to explore other the vertices adjacent to u.

In the implementation of the BFS-algorithm, we used the Boolean valued array *Added* that codes the component $C(s)$: a vertex u is in $C(s)$ if and only if $Added(u) = True$. A point is that $Added(u)$ were made true as soon as the algorithm visits u. In our DFS-implementation, we also need to code vertices that appear in the component $C(s)$. For this we also use a Boolean array that we call *Explored*. The Boolean value $Explored(u)$ is made true as soon as all adjacent vertices of u are put in the stack. So, the arrays *Added* and *Explored* act differently: $Added(u)$ is made true as soon as DFS-algorithm sees it; $Explored(u)$ is made true as soon as all adjacent vertices of u are put into stack. Here is an implementation of the DFS-algorithm:

Algorithm 5 DFS(s)-Implementation:

```
INPUT  A graph G, vertex s
OUTPUT  DFS tree T
  Create an empty stack S
  S.push(s)
  Set Added(s) ← True, and Added(u) ← False for all u ≠ s
  Set Explored(u) ← False for all vertices u
  while S is not empty do
     u ← S.pop()
     for each edge {u, v} adjacent to u do
        if not Added(v) then
           Add {u, v} to T
           S.push(v)
        end if
     end for
     Explored(u) ← True
  end while
  return T
```

We would now like to analysis the running time of this implementation. The main operations of this implementation are select and add vertices to the stack S. Each of these operations takes a constant time. Therefore, it suffices to bound the number of these select and add operations.

Again, it suffices to bound the number of times the algorithm added vertices to the stack S because each vertex needed to be added once for each time it is deleted.

Take a vertex v. It is added to the stack S every time when a vertex u adjacent to v is explored by the implementation. Hence, the number of time v is added to to S equals its degree. Also, note that once a vertex u is explored, it will never be explored again. Hence, at most $n+m$ elements are added to S. So, we have the following observation:

Observation 5 *The running time of DFS-implementation above is $O(n + m)$.*

Exercise 8. Implement an algorithm, using BFS or DFS, that given graph G, outputs all of its components.