GABRIEL MOISE

DESIGN AND ANALYSIS OF ALGORITHMS

HT 2019

PROBLEM SHEET 4

DFS and connected components, cont'd

Question 1

To determine whether an undirected graph G has a cycle containing a particular dge e on not, we start the DFS algorithm, which is linear, from one of the two vertices that are connected by e, we ignore e as an edge and we see if we can reach the other vertex from the starting point.

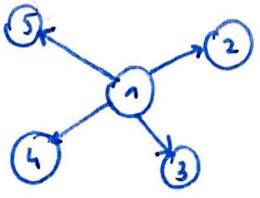
Let A be the starting vertex and B the vertex such that $(A,B) = e$. After the DFS we check if $d[A] < d[B] < f[B] < f[A]$. If so, then we can get from A to B in 2 separate ways (one is simply e and the other way is a path not containing e), thus we have a cycle that contains e in G. If not, we cannot have a cycle containing e in G as there is only one path from A to B.

Shortest paths

* Question 3

(a) n-vertex graph for which the depth of the recursion of DFS is n-1, whereas the queue of BFS has at most one vertex at any given time:

(b) n-vertex graph for which the queue of BFS will have n-1 vertices at one time, whereas the depth of the recursion of DFS is at most one:



Question 4

Linear-time algorithm for:

Input: An undirected graph $G=(V,E)$ with unit edge lengths; vertices $u, v \in V$

Output: The number of distinct shortest paths from u to v

We will use the BFS algorithm, but we will add to it the array paths such that:

paths [v] = number of distinct shortest paths from s (source) to v. (I renamed the source)

Pseudocode:

```
1.  for each v ∈ V \ {s}
2.      d[v] = ∞
3.      π[v] = nil
4.      paths[v] = 0        // No shortest paths found initially
5.  d[s] = 0
6.  paths[s] = 1            // One shortest path from s to itself
7.  π[s] = NIL
8.  ENQUEUE (Q, s)
9.  while Q ≠ ∅
10.     u = DEQUEUE (Q)
11.     for each v ∈ Adj[u]
12.         if d[v] = ∞
13.             d[v] = d[u] + 1   // Shortest path from s to v (Theorem 1)
14.             π[v] = u
15.             paths[v] = 1      // We discovered the first shortest path to v
16.             ENQUEUE (Q, v)
17.         else if (d[v] == d[u] + 1)  // We have found a new path to v, with the predecessor u
18.             paths[v] = paths[v] + paths[u]   // Thus, we add the number of distinct shortest paths to u
                                                 to paths[v]
19. return paths[v]
```

Running time: $O(|V|+|E|)$ $\Rightarrow$ linear-time algorithm

Question 5

To determine the length of the shortest cycle in the graph, we use Dijkstra's algorithm for every vertex of $G=(V,E)$ and we therefore determine $d[u,v]$ = the length of the shortest path from u to v, for all $u, v \in V$.

After this, for all pairs of vertices (u,v) we add the values of $d[u,v]$ and $d[v,u]$ to get the shortest length of a cycle containing vertices u and v. Then, we calculate the minimum of this. If it is ∞, there are no cycles in G, otherwise we return it.

Pseudocode:

```
1. for each s ∈ V        // We use Dijkstra's algorithm for every possible source
2.     for each v ∈ V
3.         d[s,v] = ∞
4.         π[s,v] = NIL
5.     d[s,s] = 0
6.     Q = MAKE-QUEUE(V) with d[s,v] as keys
7.     while Q ≠ ∅
8.       u = EXTRACT-MIN(Q)
9.       for each vertex v ∈ Adj[u]
10.          if d[s,u] + w(u,v) < d[s,v]
11.             d[s,v] = d[s,u] + w(u,v)
12.             π[s,v] = u
13.             DECREASE-KEY(Q, v, d[s,v])
14. min = ∞      // The shortest length of a cycle
15. for each u ∈ V                    // Calculating the minimum length
16.     for each v ∈ V
17.         if d[u,] + d[v,u] < min
18.             min = d[u,v] + d[v,u]
19. if min < ∞
20.     return min
21. else return "Acyclic graph!"
```

Running time

We used Dijkstra's algorithm for each vertex ⇒ lines 1-13 = $O((|V|^2 + |V| \cdot |E|) \log |V|)$

As lines 14.-21. need $O(|V|^2)$, the total running time is $O((|V|^2 + |V| \cdot |E|) \log |V|)$, as required.

Greedy algorithms

* Question 6

(a) The greedy approach does not find the optimal solution for:

Denominations: 20, 5, 1

Amount: 25

Greedy approach: 20, 1, 1, 1, 1, 1 ⇒ 6 coins

Optimal solution: 5, 5, 5, 5, 5 ⇒ 5 coins

(b) Available denominations: $p^k, p^{k-1}, \dots, p, 1$, where $p>1$ and $k \geq 0$.

First, we can have at most $(p-1)$ coins out of each $p^{k-1}, p^{k-2}, \dots, p, 1$ denominations. By supposing that we can have at least p of p^i, we can replace p of them with p^{i+1} and so on, thus obtaining a better solution.

If the optimal solution does not get at each step the maximum possible number of coins of denomination p^k, then we cannot get a better solution with the smaller denomination without needing at least p of one of them. This comes from:

$$(p-1)(p^{k-1}+p^{k-2}+\dots+p+1) < p^k$$

After this, we reduced the problem to a subproblem with denominations $p^{k-1}, p^{k-2}, \dots, p, 1$ and we use the same reasoning. Thus, the Greedy approach always gives back the correct result in this case.

Question 7

(a) Let G be an undirected graph and T a spanning tree of G.

We add $e \notin E(T)$, $e \in E(G)$ to T. Let's suppose that $e=(u,v)$ with $u, v \in T$. From the definition of a (spanning)tree, T is connected, so, before we added e to T, there existed a path from u to v, which obviously didn't have e in it. After adding e to T, we now have two distinct paths from u to v, therefore we have formed a cycle. Let's suppose that by adding e to T, we formed two distinct cycles. As both of them contain e, we have 3 ways to get from u to v, one is directly though e, one is via the first cycle and one via the second one. Before adding e, we then had two ways of getting from u to v, thus we had a cycle in T, which must be acyclic by definition, therefore we reached a contradiction.

(b) Before adding e, T was connected, acyclic and contained all the vertices of G.

After adding e, let's say that we formed the unique cycle $\langle u, a_1, a_2, \dots, a_k, v, u \rangle$. As before, every two vertices are connected. Furthermore, as we have a cycle, we can get from each node of the cycle to each other node of the cycle in two distinct ways.

After removing an edge e' from the cycle, we consider 2 cases, based on whether or not two vertices had an edge from the cycle or not (initially):

I The path between a and b did not contain any edge from the cycle. Therefore, this path was not affected in any way by adding or removing an edge from the cycle, so a and b have the same unique path between them.

II The path between them contained at least an edge e' from the cycle, with $e' \neq e$. If e' is not removed, then the path from a to b is the same (and it's unique since after we remove a different edge from the cycle, there will no longer be cycles in T). If e is removed, then the path from a to b changes in the way that e' is replaced by the other way to get

from one vertex to the other (if $e' = (a_i, a_{i+1})$, then we replace e' by $\langle a_i, a_{i-1}, \dots, a_1, u, v, a_k, \dots a_{i+1} \rangle$. Therefore, we form a path from a to b (unique as before as we already proved above).

This reasoning is applied to all pairs of vertices, thus T' is connected, acyclic and contains all the vertices of $G \Rightarrow T'$ is a spanning tree of G. ↑ the new tree formed

Question 8

(a) Let G be an undirected, connected, weighted graph, where the weight of its edges are all distinct. We want to prove that G has a unique minimum spanning tree (MST). We will show this by supposing there are two distinct MST of G, T_1 and T_2. As they are distinct, let E = the set of edges that are either in T_1, or in T_2 and we know that $E \neq \emptyset$. Let e be the edge with the minimum weight from E. We'll assume that $e \in T_1$, and since $e \in E \Rightarrow e \notin T_2$. By adding e to T_2, we will form a cycle (Q7, (a)) that is unique, let's call it C. As T_1 has no cycles, then at least one of the edges from C must not appear in T_1 (otherwise T_1 would have C). Let e' be that edge and since $e' \in T_2$ and $e' \notin T_1 \Rightarrow e' \in E$. We chose $e \in E$ to be the edge with the minimum weight from E, so $w(e) < w(e')$ (all weights are distinct). Now, if we remove e' from T_2' (the old T_2 + the edge e), from Q7, (b) we know that T_2' is a distinct MST of G, with $w(T_2') < w(T_2)$, which contradicts the fact that T_2 is an MST for G. Thus, the MST of G must be unique.

(b) To find a minimum spanning tree for G we use Kruskal's algorithm: we start from $A = \emptyset$, pick the edge with the smallest weight and add it to A, as long as it does not create cycles. In order to find the maximum spanning tree of G, we simply choose the edge with the biggest weight and add it to A, as long as it does not form a cycle.

```
1. A = ∅
2. for each v ∈ V
3.     MAKE-SET(v)
4. Sort E into decreasing order by weight w
5. for each edge (u,v) taken from the sorted list
6.     if FIND-SET(u) ≠ FIND-SET(v)
7.         A = A ∪ {(u,v)}
8.         UNION(u,v)
9. return A
```

Question 9

Let a group of n deep-sea divers $d_1, d_2, \dots, d_n$, each of them needing $t_1, t_2, \dots, t_n$ time to be lifted to the surface, respectively. Only one can be lifted at a time.

(a) We want to find an order so that the total time spent by each diver is minimised. At each step we lift a diver to the surface, thus the time added to the result is equal to $t \cdot d'$, where t is the time needed for the lift and d' is the number of remaining divers to be lifted plus one, the diver who got lifted.

Therefore, the sum is:

$T = t_{i_1} \cdot n + t_{i_2} \cdot (n-1) + \dots + t_{i_{n-1}} \cdot 2 + t_{i_n}$, where at step k we lift the diver d_{i_k}, $k \in \{1, \dots, n\}$.

To minimize T, we need $t_{i_1} \leq t_{i_2} \leq \dots \leq t_{i_n}$, thus we will lift at each step the diver who needs the least amount of time.

(b) Let's suppose without loss of generality that $t_1 < t_2 < \dots < t_n$ (for equal times, all the divers that have the same times can be lifted in any order without affecting the sum). Our solution was to lift $d_1, d_2, \dots, d_n$ in this order. Let's suppose there exists a distinct ordering such that T' (the time for that ordering) is smaller than T. Then, we must have an inversion (i, j) in the order of which we lift the divers; we encode our method with the permutation $(1, 2, \dots, n)$ and in the other method there must be j before i with $t_i < t_j$. Therefore $T' = A + t_j \cdot (n - \alpha) + B + t_i (n - \beta) + C$, where A, B, C are the times needed to lift divers up to the step where we lifted d_j, then to when we lifted d_i and until we finish. Also, as j comes before i, we have $\alpha < \beta$. Now, if we swap i with j, we get $T'' = A + t_i \cdot (n - \alpha) + B + t_j \cdot (n - \beta) + C$. Then,

$$T' - T'' = (t_j - t_i)(n - \alpha) + (t_i - t_j)(n - \beta) = -(t_j - t_i)\alpha + (t_j - t_i)\beta = \underbrace{(t_j - t_i)}_{>0}\underbrace{(\beta - \alpha)}_{>0} > 0 \Rightarrow$$

$\Rightarrow T'' < T'$, so we get a better solution.

Therefore, the best solution is when the permutation has no inversion, which is our solution from (a), $T = (1, 2, \dots, n)$.

(c) Now we suppose that each diver d_i can stay under the water only for time a_i. We want to find a "safe" order of lifting them so they don't run out of time in the water.

(i) We assume that $a_1 \leq a_2 \leq \dots \leq a_n$ and also that there exists a "safe" order to lift them to surface. We will then prove that the ordering $d_1, d_2, \dots, d_n$ is also "safe". We will do that recursively: let's think about the last diver, if it was d_n, we have nothing to do here, if it wasn't, then we could swap him with the last diver in

the order in which they are lifted up. This is because if the last diver was d_k, then he waited for the maximum amount of time under water T_n and he survived, then $a_k \geq T_n$ and as $a_n \geq a_k \Rightarrow a_n \geq T_n$, so d_n can be the last, too. As d_k is lifted earlier now, this swap keeps the order "safe". We then proceed in the same way for $d_{n-1}, d_{n-2}, \dots, d_2, d_1$ and thus we get that the order $d_1, d_2, \dots, d_n$ is safe.

(ii) To check if there is a safe order for the divers, given $t_1, t_2, \dots, t_n$ and $a_1, a_2, \dots, a_n$, we do the following:

- we sort $a_1, a_2, \dots, a_n$
- we check if the order we get from the sorting is "safe" or not
- if it is, then we found a "safe" order
- if not, then there is no "safe" order

The last part is the only one that needs explanation. Suppose that there exists an order which is "safe", but the order we chose is not. As we got $a_{i_1} \leq a_{i_2} \leq \dots \leq a_{i_n}$, with $\{i_1, i_2, \dots, i_n\} = \{1, 2, \dots, n\}$, from (i) we know that as long as there is a "safe" order, then $i_1, i_2, \dots, i_n$ is also "safe". However, it is not in this case, so there couldn't have been any "safe" order in the first place!

DFS and connected components, cont'd 2

Question 2

We want to design a linear-time algorithm that, given a directed graph G, detects if there is an odd-length cycle or not.

First, we will prove that a graph G does not contain an odd-length cycle iff it is Bipartite, i.e. it can be coloured with only to colours such that every edge in the graph has different colours at the ends.

Proof:

(i) BIPARTITE $\Rightarrow$ NO ODD-LENGTH CYCLE

Let's suppose that a graph G is bipartite and let A and B be the two sets of vertices. If we have a cycle $C = \langle v_1, v_2, \dots, v_k, v_1 \rangle$ in G, and we arbitrarily choose $v_1 \in A$, then we have $v_2 \in B, v_3 \in A, \dots, v_{2p} \in B, v_{2p+1} \in A \dots$ As v_k and v_1 have different colours, k must be a multiple of $2 \Rightarrow k$ is even $\Rightarrow$ the cycle C is even. Therefore, G can only contain even-length cycles $\Rightarrow$ G has no odd-length cycles.

(ii) NO ODD-LENGTH CYCLE ⇒ BIPARTITE

We will prove this by induction on the number of edges of G, $|E| = a$. For $a=0$ and $a=1$ this trivially holds for any graph G.

Now, let's suppose that every graph with at most a edges, this property holds and we'll prove that it also holds for every graph G with $(a+1)$ edges and with no odd-length cycles.

Let's consider an edge e of G, where $e=(u,v)$. The graph $G \setminus \{e\}$ is, by induction, bipartite, so we consider its bipartition to be $\{A, B\}$.

If u and v are from different sets, then the edge e maintains the fact that $\{A,B\}$ is a bipartition for G, so G is Bipartite.

If u and v are from the same set, let's say $u, v \in A$, there are two cases:

I. There existed a path from u to v. As the only edges that are in $G \setminus \{e\}$ go from A to B or vice-versa, to get from one element of a set to another element of the same set you need a path with an odd-number of vertices. After adding e, the graph contains an odd-length cycle ⇒ G is not bipartite (from (i)). Therefore by violating the no odd-length cycle of the graph, we obtain that it can't be Bipartite anymore.

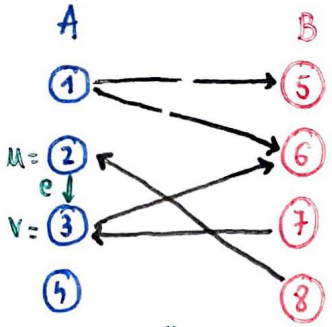
II There wasn't any path from u to v before adding e. Then we will create a new partition for G, (A', B'). We start from $u \in A'$, $v \in B'$. We set all the vertices w s.t. $(u,w) \in E \cup \{e\}$ to be from B'. Also, we set all the vertices x s.t. $(x,v) \in E \cup \{e\}$ to be from A'. Since the was no path from u to v before adding e, we must have $w \neq x$. By continuing with the same reasoning, we can colour the graph just with two colours, because we constantly use the fact that there was no path from u to v before adding e and also all the vertices that do not have any path to u or v (or vice-versa) stay in the same place, as initially the graph was Bipartite

Thus, G is bipartite.

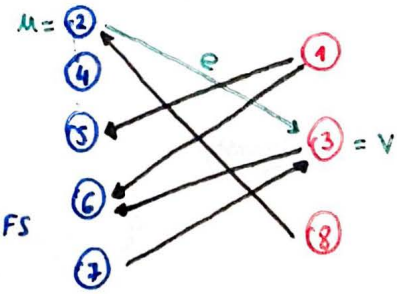
The algorithm now needs to determine if the graph is bipartite or not:

- if it is ⇒ no odd-length cycle
- if it isn't ⇒ has an odd-length cycle

To check if a graph is Bipartite or not, we first use the linear-time algorithm to determine its strongly-connected components SCC_1, SCC_2, ... SCC_k and then we use an adapted version of BFS for each of them (to keep track of the colours used):



- 2 is blue ⇒ 3 is red
- 3 is red ⇒ 6 is blue, 7 is blue ⇒ 1 is red ⇒ 5 is blue
- 4 remains blue



PSEUDOCODE

1. Call DFS(G) to compute finishing times $f[u]$ for all u
2. Compute G^T
3. Call DFS(G^T), visiting the vertices in order of decreasing $f[u]$
4. Output the vertices in each tree of the DFS forest formed in the second DFS as a separate SCC

// So far we needed $\Theta(|V|+|E|)$ time

5. for each SCC
6. choose a random source s
7. for each $u \in SCC$
8. colour[u] = -1 // It can be -1, meaning no colour and 0 or 1 for the two colours
9. colour[s] = 1
10. Q = $\emptyset$
11. ENQUEUE(Q, s)
12. while Q $\neq \emptyset$
13. u = DEQUEUE(Q)
14. for each $v \in Adj[u]$
15. if colour[v] == -1
16. colour[v] = 1 - colour[u] // colour v differently from u
17. ENQUEUE(Q, v)
18. else if colour[v] == colour[u] // then the graph is not bipartite
19. return TRUE // then it must have an odd-length cycle
20. return FALSE // All the vertices can be coloured correctly => no odd-length cycle

The running time for each SCC is $\Theta(|V_{scc}|+|E_{scc}|)$ => the total running time for lines 5-20 is $\Theta\left(\sum_{scc}(|V_{scc}|+|E_{scc}|)\right) = \Theta(|V|+|E|)$

Therefore, this algorithm needs linear time.