Informatics 2 – Introduction to Algorithms and Data Structures

Tutorial 3: Sorting and Graph Algorithms

This sheet covers material from Lectures 13, 14 and 15.

1. Assume that you are given an array A containing n integer numbers from the set $\{0, 1, \ldots, k\}$ for some $k \leq n$. Search on the internet or the textbooks for an algorithm that on input A produces a sorted array containing those n numbers in non-decreasing order in time $O(n)$. Present the pseudocode for the algorithm and provide a justification for its running time.

   This algorithms sorts the array in $O(n)$ time. Does this contradict the lower bound of $\Omega(n \lg n)$ on the running time of any general sorting algorithm presented in the lectures?

   Solution:

   We first describe the CountingSort algorithm, which is also in CLRS 8.2. The algorithm first creates a new array $C[0, 1, \ldots, k]$ which will store the number of occurrences of each number $i \in \{0, 1, \ldots, k\}$ in the array, and initialises $C[i] = 0$ for all $i \in \{0, 1, \ldots, k\}$. Then, it runs through the given array A and for each element $A[j]$, it increases the number of occurrences of that element in the array C by 1. At the end of this step, they array C contains the number of occurrences of each number of the array A, with $C[i]$ being the number of occurrences of number i.

   From the array C, we can compute the modified array which stores the positions at which each the sorted array should switch from the previous element to the next and then we can place the elements in the correct positions by counting down from the length of the array A, which is n, until we reach position 1. The pseudocode for the algorithm is given in Algorithm 1.

   Lines 1-3 of the algorithm can be done in $O(k)$ time, as we have to initialise an array of length k with zeros. Lines 4-6 can be done in $O(n)$, as we run through the array A of length n and then perform a constant number of computations in each iteration of the loop. Lines 7-8 can be done in $O(k)$ time, since again we run over an array of length k and perform a constant number of computations each time. Finally, lines $9 - 12$ can be done in time $O(n)$ for the same reasons. Overall, since $k \leq n$, the asymptotic running time of the algorithm is $O(n)$.

   The running time of the algorithm does not contradict the lower bound of $\Omega(n \lg n)$ for general sorting algorithm. A general sorting algorithm should be able to sort any type of array that contains *comparable* elements, i.e., elements for which there is a comparison operation which indicates which one should go before which. CountingSort's performance is clearly contigent on these elements being numbers from a certain set,

Algorithm 1 CountingSort(A, B, k)

```
1: Initialise array C[0, ..., k].
2: for i = 0 to k do
3:     C[i] = 0
                                  ▷ The array C now contains 0 in all entries.
4: for j = 1 to n do
5:     h = A[j]
6:     C[h] = C[h] + 1
              ▷ The array C now contains the number of occurrences of each element of A.
7: for i = 1 to k do
8:     C[i] = C[i] + C[i − 1]
   ▷ The array C now contains in position i, the number of elements which have value at
   most i.
9: for j = n to 1 do
10:    h = C[A[j]]
11:    B[h] = A[j]
12:    C[A[j]] = C[A[j]] − 1
```

the size of which is known in advance. If we do not know this size in advance, we do not know how large the arrray C would have to be.

2. Consider the graph G of Figure 1.

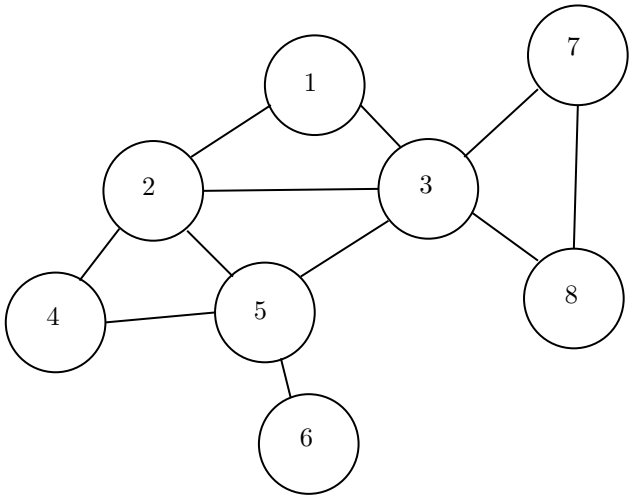

Figure 1: The graph G (part of the graph in Fig 3.2. in KT)

(a) Write the adjacency matrix and the adjacency list representation for the graph. For the adjecency list, for consistency, consider the neighbours of a vertex appearing in ascending numerical order in the list for each node.

(b) Run Depth-First Search (DFS) on G starting from node 1. Explain the steps of DFS and note the order in which the nodes will be explored by DFS. Write down the spanning tree produced by DFS in adjacency list representation.

0	1	1	0	0	0	0	0
1	0	1	1	1	0	0	0
1	1	0	0	1	0	1	1
0	1	0	0	1	0	0	0
0	1	1	1	0	1	0	0
0	0	0	0	1	0	0	0
0	0	1	0	0	0	0	1
0	0	1	0	0	0	1	0

Table 1: The adjacency matrix representation of G.

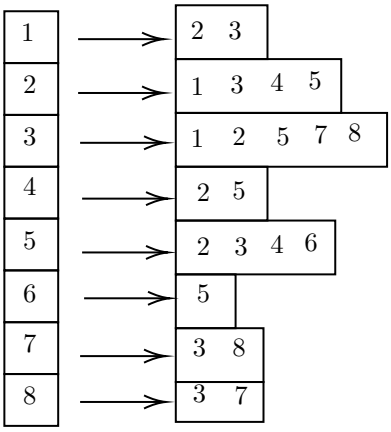

Figure 2: The adjacency list representation of G.

(c) Run Breadth-First Search (BFS) on G starting from node 1. Explain the steps of BFS and note the level that each node will be assigned to during the execution of BFS. Write down the spanning tree produced by BFS in adjacency list representation.

(d) Compare the two spanning trees produced by DFS and BFS starting from node 1 respectively. What do you observe?

Solution:

(a) The adjacency matrix representation of the graph is given in Table 1. The adjacency list representation of the graph is given in Figure 2.

(b) We first run DFS$(G, 1)$ and label node 1 as *visited*. All of the edges incident to node 1 are unexplored and all of the neighbours are unvisited. We take the next neighbour in the adjacency list, which is node 2. We label edge $\{1, 2\}$ as a *discovery edge* and call DFS$(G, 2)$. We label node 2 as visited. Next, we consider the edges incident to node 2, all of which are unexplored. We take the next neighbour of node 2 in the adjacency list, which is node 1. Node 1 has is visited, so we label edge $\{2, 1\}$ as a *back edge* and do *not* call DFS on node 1. We continue with the next node in the adjacency list of node 2, which is node 3. The edge $\{1, 3\}$ is unexplored, so we label it as a *discovery edge*, and we call DFS$(G, 3)$.

We continue in this manner with the rest of DFS through the graph (in class the whole process will be shown, see Lecture 14 slides and Section 3.2. on KT (also

Figure 3.5. for exactly the same example). The nodes will be explored in the order $1, 2, 3, 5, 4, 6, 7, 8$, where "explored" here means that DFS will be called on input G and those nodes. The discovery edges are

$$\{1,2\}, \{2,3\}, \{3,5\}, \{4,5\}, \{5,6\}, \{3,7\}, \{7,8\},$$

which form the DFS spanning tree. Writing those in adjacency list representation is easy.

(c) Our BFS run starts from node 1 and places it in L_0. Then, it explores the neighbours of node 1, namely node 2 and node 3 and places them in L_1. These nodes were not visited before, so the edges $\{1,2\}$ and $\{1,3\}$ are labelled as *discovery edges*. Then, all the neighbours of all nodes in L_1 (namely node 2 and node 3) are considered and places in L_2. These are the nodes $4, 5, 7$ and 8. From the edges reaching those nodes $\{2,4\}, \{2,5\}, \{3,7\}$ and $\{3,8\}$ reach nodes that were not visited before, so they are *discovery edges*. The edges $\{2,3\}$ and $\{3,5\}$ reach previously visited nodes and hence they are *cross edges*. Finally, the algorithm will consider the neighbours of the nodes in L_2 and put them in L_3. There is only a single such node, node 6. The edge that reaches node 6 is $\{5,6\}$ which is labelled as a *discovery edge*, and all of the other edges considered in this step, namely $\{4,5\}, \{7,8\}$ are labelled as cross edges. The BFS spaning tree is formed by the discovery edges $\{1,2\}, \{1,3\}, \{2,4\}, \{2,5\}, \{3,7\}, \{3,8\}, \{5,6\}$. Writing those in adjacency list representation is easy.

(d) We observe that the spanning trees produced by DFS and BFS are different, even though they started at the same node. The next question shows that if they produce the same spanning tree T, then it must be the case that $T = G$.

3. (a) Prove the following property for the layers produced by BFS: For any edge (u, v), either u and v are in the same layer, or $|L(u) - L(v)| = 1$, where $L(x)$ is the layer of node x.

Solution:

We will use proof by contradiction. Suppose that there exists any edge $e = (u, v)$ such that u is in some layer i and v is in some layer j, such that $j > i+1$. Since u is in a layer with a smaller index, it was obviously explored first in the operation of BFS, at step i. In the next step, BFS explores all the neighbours of u that have not been explored in previous steps. Since v is a neighbour of u and it was not explored in step $i+1$, it must have been explored before step $i+1$. However, since v has a label $j > i+1$, it must have been explored in step j, and we obtain a contradiction.

(b) $^{(*)}$ Let $G = (V, E)$ be a connected graph and let $s \in V$ be a node of G. Suppose that we run DFS(G, s) and obtain a DFS spanning tree T and that we also run BFS(G, s) and obtain the same BFS spanning tree T. Prove that $G = T$.

Solution:

We will use proof by contradiction. Suppose that $G \neq T$. This means that there exists an edge $e = (u, v)$ in G, which is not an edge of T. Since e is not part of the DFS tree, it must be a *back edge*. This means that one of the two endpoints (u or v) must be an ancestor of the other in the way the DFS tree is generated (see also Statement 3.7., page 85 of KT). Since $e = (u, v)$ does not belong to T, this implies that in T, $|d(s, v) - d(s, u)| \geq 2$. In turn, this means that v and u are in layers of the BFS tree that are neither consecutive, nor the same. However, this is not possible by statement (a) of the question, a contradiction.

4. (a) Assume that you are given access to a graph $G = (V, E)$ and a node $u \in V$, and for every node $v \in V$, the distances $d(u, v)$ between u and v in G. Present an algorithm that produces a sorted list of the nodes in terms of non-decreasing distance from u which runs in time $O(|V|)$. Recall that the distance between two nodes is the number of edges in the shortest path between the nodes in G.

   Solution:

   The possible distances in the graph are elements of the set $\{0, 1, \ldots, n-1\}$. We can use the CountingSort algorithm to sort these distances, with $k = n$, and the running time will be $O(n)$. To produce a sorted list of nodes rather than the distances, all we need is to keep track of which node has each distance from u, which we can recover from list A containing the distances. In particular, in Line 11 of the algorithm, in the array B we can store the index j of the node that has distance $A[j]$ from u rather than the distance itself.

   (b) Assume that you are given access a the graph $G = (V, E)$ and a node $u \in V$ as above, but not the distances $d(u, v)$ as above. Present an algorithm that produces a sorted list of the nodes in terms of non-decreasing distance from u which runs in time $O(|V| + |E|)$.

   Solution:

   This can be done by simply using BFS starting from u. We know from the way that DFS works, that each node i in layer L_i has distance i from u. Instead of using different lists L_i we can use a single list L to which we append always the new nodes that BFS adds. At the end of the execution of the algorithm, L will contain the nodes of G in non-decreasing order of distance from u.