Complexity Notation and Basic Data Structures

Solutions to the Practical Exercises

Departamento de Engenharia Informática (DEI)
Faculdade de Engenharia da Universidade do Porto (FEUP)

Spring 2023

A - Graphs

Exercise 1

a) The main idea of the algorithm is to first find the source vertex, and then iterate through its adjacent edges until the one going to the specified destination is found. This solution works even if there are multiple edges connecting the same pair of vertices (multigraphs).

Pseudo-code for removing an edge:

Input and auxiliary functions

- graph: input graph.
- source: vertex id of the source vertex.
- dest: vertex id of the destination vertex.
- delete(V, x): deletes element x from vector V.

Output: new graph, and a boolean return value indicating if the removal operation was successful.

```
remove_edge(graph, source, dest)
        // Find the source vertex
        srcVertex ← NULL
        foreach (v: graph.get_vertex_set())
            if(v.get_id() == source)
                srcVertex ← v
        if (srcVertex == NULL)
            return graph, false
        // Find the destination vertex
        adj ← srcVertex.get_adj()
        removedEdge ← false
```

```
foreach(edge: adj)
  if(edge.get_dest().get_id() == dest)
    // Remove the edge
     delete(adj, edge)
     removedEdge ← true
return graph, removedEdge
```

b) The algorithm can be divided into three main parts:
   1. Finding the source vertex can be performed in O(V) time, since, in the worst-case scenario, all vertices are processed and processing each one takes O(1) time since it only involves performing a numeric comparison.
   2. Find the destination vertex in the source vertex's list of adjacent edge and removing the corresponding each can be performed in O(E) time, since in the worst case, all of the graph's edges will need to be processed (if they are all incident to the source vertex) and processing each one takes O(1) time.
   3. Removing the actual edge can be performed in O(E) time since removing an element from a vector with E elements takes O(E) time. If a linked list is used, then this operation can be improved to O(1) time.

Thus, the overall algorithm takes $O(V + E + E) = O(V + 2 \times E) = O(V + E)$ time. In big-O notation, the constant scalars being multiplied can be omitted to simplify the expression.

Note: if a linked list is used for the vertices' adjacency list, the temporal complexity is the same: $O(V + E + 1) = O(V + E + 1) = O(V + E)$. In big-O notation, the constant scalars being added can be omitted to simplify the expression.

c) See source code.
d) The main idea of the algorithm is to first find the vertex, remove all edges from other vertices oriented towards this vertex and finally remove this vertex.

<u>Pseudo-code for removing a vertex</u>:

Input and auxiliary functions
- graph: input graph.
- id: id of the vertex to remove.
- delete(V, x): deletes element x from vector V.
- remove_vertex(graph, source, dest): removes an edge from a graph.

Output: new graph, boolean indicating if the removal operation was successful.

```
remove_vertex(graph, id)
  vs ← graph.get_vertex_set()
  foreach(v: vs)
    if(v.get_id() == source)
        foreach(u: graph.get_vertex_set())
            remove_edge(graph, v.get_id(), u.get_id())
        delete(vs, v)
        return graph, true
  return graph, false
```

e) The algorithm can be divided into three main parts:

1. determining if the vertex to be removed exists can be performed in O(V) time, since, in the worst-case scenario, all vertices are processed and processing each one takes O(1) time.
2. for each vertex, removing the edge that connects it to the vertex to remove can be performed in O(V×E) (O(E) for each vertex), since in the worst case, such an edge does not exist and thus all the vertice's adjacent edges need to be processed.
3. removing the actual vertex can be performed in O(V) time since removing an element from a vector with V elements takes O(V) time. If a linked list is used, then this operation can be improved to O(1) time.

Thus, the overall algorithm has a time complexity of O(V + V×E + V) = O(V×E).

f) See source code.

Exercise 2

a) The DFS algorithm keeps track of which vertices have already been visited to avoid visiting a vertex twice.

In each iteration, a new vertex is visited and one of its children is selected in the next iteration. Once a vertex is found without any unvisited children, DFS backtracks to the current vertex's father/ancestor. Therefore, other children of the current vertex are only visited once all the paths of the first child are explored (unless they are already visited in one of these paths).

One DFS order for the graph is: A, B, E, F, C, G, D.

Since there is no predefined order for selecting which child of a node should be visited first, different orders are possible. For instance, in vertex A, vertex C can be visited before reaching B. Thus, another valid DFS order is: A, C, F, B, E, G, D.

b) See source code.

Exercise 3

a) Analogous to DFS, the BFS algorithm keeps track of which vertices have already been visited.

At each iteration, a new vertex is visited and all of its children are added to a queue to be visited next. This queue determines which child is visited next. Therefore, a vertex's children are visited before reaching its grandchildren.

One BFS order for the graph is: A, B, C, D, E, F, G.

Just like with DFS, different orders are obtainable if there is no predefined order for selecting which child of a node should be added to the queue first. For instance, in vertex A, vertex C can be visited before reaching B. Thus, another valid BFS order is: A, C, B, D, E, F, G.

b) See source code.

Exercise 4

a) The topological sort of a graph can be computed by repeating the two following steps until there are no more vertices to process:
   - Select any vertex that has no incoming edges.
   - Remove the select vertex and its outgoing edges from the graph.

When there are no more vertices to process, if all the graph's vertices were processed, then it means that the graph has a valid topological sorting (in particular, it means the graph is connected and acyclic).

In this graph, the following vertices are obtainable with each iteration of the previously described algorithm: A, B, C, D, E, F, H, G, I.

Another valid topological sort is obtainable by swapping the order of vertices B and C, since C also becomes available once A is processed. The new order is: A, C, B, D, E, F, H, G, I.

b) See source code.

Exercise 5

See source code.

Exercise 6

a) The SCCs can be computed using the Kosaraju-Sharir algorithm.

First, the graph is traversed using a DFS and the visited vertices are added to a stack **after** exploring all the unvisited vertices of its adjacent edges. The DFS can start on any node. In this solution, the search will begin on vertex A.

The resulting stack (starting from the top) is: [A, D, C, G, B, E, F].

The stack's vertices are then processed and are all set as "not visited". The direction of all the graph's edges is temporarily reversed until the algorithm is done. For each unvisited vertex, a new DFS is run and all vertices that are reachable are marked as belonging to the same SCC.

In this example, by performing DFS starting on vertex A on the reverse graph, the following vertices are accessible: B, C, E and F.

The next unvisited vertex obtained from the stack is D, from which no new vertex is reachable.

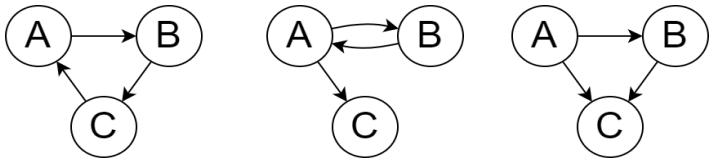
Finally, the last unvisited vertex is G, from which no new vertex is reachable either.

Therefore, the graph has three SCCs: {A, B, C, E, F}, {D} and {G}.

b) See source code.

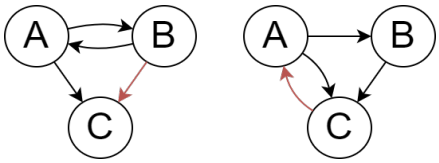
Exercise 7

a) In the three graphs below, the left one has 1 SCC ({A, B, C}), the middle one has 2 SCCs ({A, B} and{C}) and the right one has 3 SCCs ({A}, {B} and {C}).



b) Adding a new edge may trigger the union of two SCCs in the resting graph.

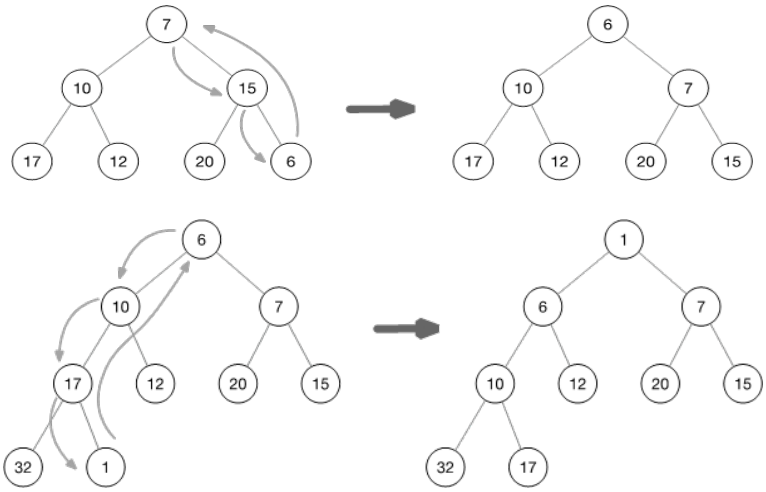
c) In the two graphs below, the adding the red edge to the left one does not modify the SCCs (which remain 3), while that adding the red edge to the right graph reduces the number of SCCs from 3 to 1.



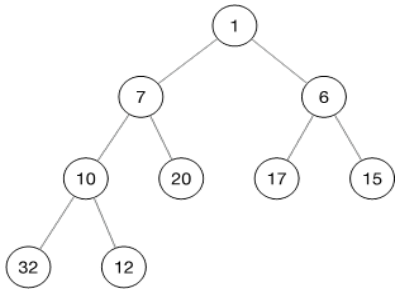
B - Heaps

Exercise 8

a) The figure below depicts the state during the insertion of the 6 and 1 elements for the first insertion order.



Inserting the elements in the order 1, 32, 6, 20, 12, 17, 15, 7, 10 would yield the min-heap shown below.



A basic conclusion is that as a partial-order data structure, there are many possible data organizations corresponding to different insertion order, all of which meet the partial order constraints between parent nodes and its children nodes.

b) See source code.

c) The algorithm has two main steps:

- building a heap using a n-element array takes O(n) time (link to a detailed justification).
- removing the minimum element of the heap n times takes O(n*log(n)) time since each individual extraction runs in logarithmic time and there at most n elements in the heap.

Thus, the overall algorithm time complexity is $O(n \times \log(n) + n \times \log(n)) = O(2 \times n \times \log(n)) = O(n v \log(n))$.

Exercise 9

a) See source code. The main idea of the algorithm is to build a min-heap with the set's elements and then extract the heap's minimum k times to get the k-th smallest element.
b) The algorithm has two main steps:
   1. building a heap of n elements takes O(n) time, as discussed in the solution of exercise 7.
   2. removing the minimum element of the heap k times takes O(k×log(n)) time since each individual extraction runs in logarithmic time and there are at most n elements in the heap.

Thus, the overall time complexity of the algorithm is O(n + k×log(n)).

Exercise 10

The number of elements in a heap of height h ranges from 2^h to 2^{h+1} -1. To prove this property by induction we will unse induction over the natural number h with `h ≥ 0`.
Base case: for h = 0, only one-element heap is possible, thus the number of elements ranges from 1 to 1.
$2^h = 2^0 = 1$ and $2^{h+1} - 1 = 2^{0+1} - 1 = 2^1 - 1 = 2 - 1 = 1$, so the property holds for h = 0.

Inductive step: let h > 0. Let us assume that the property is valid for heap of height (h - 1) and let us prove that this implies that the property also holds for h elements.

To make a heap of height h, at least one element needs to be added to a full heap of h levels, which according to the induction hypothesis, contains $2^h - 1$ elements, so a heap of height h has at least $2^h - 1 + 1 = 2^h$ elements.

The largest possible heap of height h is can be formed by joining the root of two full heaps of height (h - 1) with a new root element, thus resulting in $2 * (2^h - 1) + 1 = 2^{h+1} - 2 + 1 = 2^{h+1} - 1$ elements.

Therefore, the property holds for h.

Conclusion: As the property is true for h = 0 and, for any h > 0, if the property holds for h-1, then it holds for h, then the property is true for any h. Therefore, the number of elements in a heap of height h is contained in the interval $[2^h, 2^{h+1} - 1]$.

Exercise 11

The largest element can be in any position, as long as the corresponding node does not have any children. If the node has a child, then it means that there exists an element larger than the one being considered, which means that the latter cannot be the largest element of the min-heap.

C - Trees

Exercise 12

In the preorder traversal, the root of the tree is the first element, followed by the elements of the left subtree, followed by the elements of the right subtree.

First tree: D, B, A, C, F, E, G
Second tree: C, B, A, D, E
Third tree: E, C, B, A, D, H, F, G, I

In the inorder traversal, the elements of the left subtree come first, followed by the root, followed by the elements of the right subtree.

First tree: A, B, C, D, E, F, G
Second tree: A, B, C, D, E
Third tree: A, B, C, D, E, F, G, H, I

In the postorder traversal, the elements of the left subtree come first, followed by the elements of the right subtree, followed by the root.

First tree: A, C, B, E, G, F, D
Second tree: A, B, E, D, C
Third tree: A, B, D, C, G, F, I, H, E

In the level order traversal, the elements are sorted by increasing depth (i.e. the number of edges on the path between them and the root). Elements of the same depth are sorted from left to right. This corresponds to an order of the elements obtained during a breadth-first search traversal of the tree, starting at the root.

First tree: D, B, F, A, C, E, G
Second tree: C, B, D, A, E
Third tree: E, C, H, B, D, F, I, A, G