THE UNIVERSITY OF
SYDNEY

Assignment 3

COMP2123: Data Structures and Algorithms
510415022

Problem 1. (20 points)

We are required to prove that the following algorithm TEST-BST correctly tests whether a given binary tree T has the binary search tree (BST) property. This property states that in a BST:

> Each internal node v stores an element e such that the elements stored in the left subtree of v are less than or equal to e, and the elements stored in the right subtree of v are greater than or equal to e.

```
1: function TEST-BST(T):                          ▷ Assumes T has n distinct keys
2:     for u ∈ T do
3:         if u.left ≠ null and u.key < RIGHTMOST-DESC(u.left).key then
4:             return False
5:         if u.right ≠ null and u.key > LEFTMOST-DESC(u.right).key then
6:             return False
7:     return True
```

We will prove the algorithm's correctness by inducting on the traversal of the binary tree T. Without loss of generality, we will consider that each node $u \in T$ is visited through a post-order traversal (the proof can similarly be conducted by considering a pre-order or in-order traversal). In particular, we will show that the contents of the `for`-loop on lines 3 to 6 of the function will either `pass` or return `False` correctly at each of the n iterations.

Base case $n = 1$**:** The first iteration of the post-order traversal will assign u to the leftmost leaf of T. As u is a leaf, it has no children, and so both $u.left = null$ and $u.right = null$. Neither `if`-statements are executed and thus the algorithm passes to the next iteration. This is the correct result as any single node follows the BST property trivially.

Inductive step: We assume for our inductive hypothesis that the algorithm holds correct until the k-th iteration of the post-order traversal, for some $k \geq 1$. Now we show the algorithm to be true for the $(k+1)$-th iteration.

The $(k+1)$-th node of the postorder traversal is dependent on the k-th node, which we will call v here for ease of notation. There are three cases to be considered here[1]:

1. If v does not have a parent (i.e. v is the root), then v is the last node in the traversal and there is no next node. The algorithm is correct in this k-th iteration as per our inductive hypothesis.

2. If v is a left child with a sibling, then the next node is the node returned from POSTORDER($v.parent.right$). This node will be a leaf node, which we have already proven the algorithm correct for in our base case.

3. If v is a right child, or v is a left child with no sibling, then v's parent u is the next node in the postorder traversal.

[1] These cases were explained in greater detail in Problem 7 of Tutorial 3.

The algorithm reaching node u from v must mean that the algorithm did not return `False` in its k-th iteration. This implies that the binary tree rooted at $u.left$, T_L, is a valid BST, and that the binary tree rooted at $u.right$, T_R, is also a valid BST, as per the inductive hypothesis. This in turn implies that RIGHTMOST-DESC($u.left$) will return the maximum value of T_L and that LEFTMOST-DESC($u.right$) will return the minimum value of T_R (this follows from the BST property).

Now, as the algorithm returns `False` if $u.key < \max\{T_L\}$ or returns `False` if $u.key > \min\{T_R\}$, essentially checking if the BST property is maintained, then the algorithm either correctly returns `False` or the binary tree rooted at u is a BST and the algorithm continues to the next iteration.

In all three cases, the $(k+1)$-th iteration correctly returns either `False` or passes to the next iteration. Hence, by the principle of mathematical induction, the algorithm is correct for all n iterations while traversing the nodes of T.

Problem 2. (40 points)

The question asks us to consider having k sorted arrays $A_1, A_2, \ldots, A_k$ storing n distinct keys. Let $S = \{A_1 \cup A_2 \cup \ldots \cup A_k\}$ and $x, y \in S$. Then, we want to design a data structure which can maintain a pinning pair $P = (x, y) : x < y$ as well as support the following operations in $O(k)$ additional space:

- INITALISE(): $x \leftarrow \min\{S\}$, $y \leftarrow \max\{S\}$; takes $O(k)$ time
- LEFT-FORWARD(): $x \leftarrow$ key after x in $\mathsf{sorted}(S)$; takes $O(\log k)$ time
- RIGHT-BACKWARD(): $y \leftarrow$ key before y in $\mathsf{sorted}(S)$; takes $O(\log k)$ time

a) The high level idea here is:

i. Construct one min-heap $H_{\min}$ and one max-heap $H_{\max}$ which each have a capacity of k elements.

ii. $H_{\min}$ will be filled with the smallest k elements in S, while $H_{\max}$ will be filled with the largest k elements in S.

iii. Calling INITIALISE() at the start of the algorithm will perform the above two steps, as well as set the pinning pair $P = (H_{\min}.\mathsf{remove\_min}(), H_{\max}.\mathsf{remove\_max}())$.

iv. Calling LEFT-FORWARD() will check if $x' := H_{\min}.\mathsf{remove\_min}() < y$ and if so, update the pinning pair to $P = (x', y)$. Otherwise, it will not update P and return -1.

v. Calling RIGHT-BACKWARD() will check if $y' := H_{\max}.\mathsf{remove\_max}() > x$ and if so, update the pinning pair to $P = (x, y')$. Otherwise, it will not update P and return -1.

vi. After any of the above three operations are called, $H_{\min}$ and $H_{\max}$ will refill themselves by taking the next smallest or largest element from S respectively.

The part of the algorithm needing the most explanation is how $H_{\min}$ and $H_{\max}$ will be filled with the first k smallest and largest elements of S. Furthermore, we must explain how each heap will be refilled after calling either INITIALISE(), LEFT-FORWARD() OR RIGHT-BACKWARD(). This following solution was inspired by Tutorial 5 Question 6.

We will maintain two pointers in each array, $p_{\min}$ and $p_{\max}$, for the first and last elements which have not been added to either of the heaps. Initially, these pointers will be set to the first and last indices of the array. Note that if the arrays are found to be empty, we will simply throw an error and return -1 (as no pinning pair is possible). If not, then in the INITIALISE() operation we will iterate over each of the k arrays and use $p_{\min}$ and $p_{\max}$ to add the first and last entry of each array to $H_{\min}$ and $H_{\max}$ respectively. The pointers will then be updated: $p_{\min} \leftarrow p_{\min} + 1$ and $p_{\max} \leftarrow p_{\max} - 1$. We now have two heaps of k elements each.

When the root of one of the heaps is removed through calling either $\mathsf{remove\_min}()$ or $\mathsf{remove\_max}()$, we refill that heap using this principle: if the min/max value from the root came from array A_i, then the heap will be refilled with the next min/max value from array A_i. To

check which array the root element came from, we will store the pointer to the array alongside the element itself in the heap. Once again, the next value will be accessed using $p_{\min}$ or $p_{\max}$, the pointer then being updated in the same way as before.

b) The main question of correctness is: *how do the heaps storing only k elements output the sorted values of n keys?* Firstly, observe that at any moment of the algorithm, we maintain each heap with exactly one element from each of the k arrays – this is why we refill the heap with an element from the same array as the newly-removed root. This ensures that the sorting which occurs in the heaps takes into account the contents of all arrays. Furthermore, note that each of the k arrays are *sorted*. This means that the node which refills the heap can never be smaller than the previous root element. Hence, at any moment, the roots of $H_{\min}$ and $H_{\max}$ store the minimum and maximum of the "next" k elements of the set S. This ensures that the output of the heaps is in sorted order, so that when we call LEFT-FORWARD or RIGHT-BACKWARD, x and y are guaranteed to be set to the next smallest and next largest elements in S respectively. For further details, please see Solution 6 of Tutorial 5 from which the proof here has been derived.

c) The space complexity of this data structure fits in the requirements of the question. We store a total of $2k$ pointers in the arrays as well as two heaps, each of capacity k. In total this is $O(k)$ space. The time complexity will require some further analysis.

   i. INITIALISE(): This operation will iterate over all k arrays and set up two pointers $p_{\min}$ and $p_{\max}$ for each array. Alongside this, the operation will add the element at each of the pointers to $H_{\min}$ and $H_{\max}$ respectively. Recalling that building a heap of n elements using the bottom-up approach takes $O(n)$ time, and including the two calls to remove_min() and remove_max() which take $O(\log k)$ time each, the total running time of this operation can be bounded by $O(2k + k + 2\log k) \approx O(k)$. Note that here we are including the refill operation inside the remove_min()/remove_max() time complexity of $O(\log k)$ – adding one element to a heap will trigger up-heap bubbling, an operation which also takes $O(\log k)$ time.

   ii. LEFT-FORWARD(), RIGHT-BACKWARD(): Each of these operations simply calls remove_min() or remove_max() and then updates the pinning pair. As previously explained, these operations (including their subsequent refills) can be undertaken in $O(\log k)$ time, as required.

Problem 3. (40 points)

The problem requires us to design an algorithm to calculate the risk factor of a given vertex u in a connected undirected graph G, where the risk factor is given by

$$\text{risk factor of u} = |\{(x, y) \in V - u \times V - u : \nexists\ x\text{-}y \text{ path in } G[V - u]\}|\,. \tag{1}$$

The algorithm should run in $O(n + m)$ time where $n = |V|$ and $m = |E|$. The answer below assumes the graph is given in an adjacency-list representation.

a) We will break down the algorithm into a few main parts: checking whether u is a cut vertex, calculating the number of nodes in each connected component in the graph $G[V - u]$, and finally calculating the risk factor of u by checking the number of pairs of vertices which cannot reach one another in the induced graph.

I. **Finding if the given vertex u is a cut vertex**
This problem was largely defined in the Week 7 lecture as well as question 7 of Tutorial 7, however we will briefly recap it here as it is a key part of our algorithm.
In a DFS tree, a vertex u is a *cut vertex* if one of the two following conditions is true:

i. u is the root of the DFS tree and has at least two children
ii. u is not the root of the tree and has at least one child v such that no vertex in the subtree of v is connected to an ancestor of v using a backedge

If we were required to find all cut vertices of the graph, we would need to compute and store the level of each node and the `down_and_up` value for each node in order to perform the algorithm explored in the lectures which relies on condition ii. (see Appendix A for the full details). However, as we are simply testing if a *given* node u is a cut vertex, we can just run a depth-first search rooted at u and use condition i. as our cut vertex condition.

II. **Removing the vertex u from G**
If the root of the DFS tree u has less than two children (and thus is not a cut vertex), the algorithm will return 0. Otherwise, we must remove u to continue further with our algorithm. As we are assuming the graph G has been provided in an adjacency list data structure, removal of a vertex along with its connected edges can be done in $O(\deg(v))$ time. Let this graph $G[V - u]$ be denoted G'.

III. **Calculating the number of nodes in each connected component of G'**
This step can also be undertaken using a variation of the depth-first search algorithm. The high-level idea here is that the first recursive DFS call on a node n will cause all the nodes in the connected component containing n to be visited. Then, the next DFS call will either be on a node in the first connected component (which has already been visited in the first iteration and therefore will be skipped), or it will be on a node in the second connected component of the graph. In the latter case, the DFS algorithm will

run and cause all the nodes in this second connected component to be marked as visited. The algorithm continues as such until it has traversed all the connected components of the graph. We can therefore modify the DFS algorithm to keep a count of:

i. How many connected components the graph G' has

ii. The number of nodes in each of the connected components

Our main function, COUNT-COMPONENTS will initiate the DFS helper function in each connected component of the graph and will keep track of the number of connected components in G'. The DFS helper function DFS-COUNT will return the number of nodes inside each connected component. Pseudocode has been provided below for both these functions.

```
1:  function COUNT-COMPONENTS(G)
2:      components ← ∅          ▷ stores the size of each connected component in G
3:      for u ∈ G do
4:          visited[u] ← False
5:
6:      for u ∈ G do
7:          if not visited[u] then
8:              count = DFS-COUNT(G, u, visited)
9:              components.append(count)
10:     return components, length(components)
```

```
1:  function DFS-COUNT(G, u, visited)
2:      visited[u] ← True
3:      count ← 1               ▷ stores number of nodes in this connected component
4:      for each vertex v adjacent to u do
5:          if not visited[v] then
6:              count ← DFS-COUNT(G, v, visited)
7:      return count
```

IV. **Calculating the risk factor**

The above step returns a list of the form $[v_1, v_2, \ldots, v_k]$ where v_k is the number of nodes in the k-th connected component of G'. Now all that is left to do is to calculate the risk factor.

To simplify this calculation, note that each vertex u in some connected component of G' fails to reach *exactly* the same number of vertices as its neighbours. For example, if one connected component C_1 contains 4 nodes, another component C_2 contains 6, and another component C_3 contains 8, then all 4 nodes in C_1 equally cannot reach $6+8=14$ nodes. The risk factor in this case would be $\frac{1}{2}[4 \times 14 + 6 \times 12 + 8 \times 10] = 56$. The formula to calculate the risk factor of a node u based on the above observation is

$$\text{risk factor} = \frac{1}{2} \sum_{i=1}^{k} v_i \cdot (n - v_i) \tag{2}$$

where n is the number of nodes in the graph G. This can simply be calculated through one traversal over each v_i in the list.

b) i. Step I, detecting if the given vertex u is a cut vertex, is correct based on the properties of the DFS traversal. The depth-first search algorithm, as the name suggests, produces a tree rooted at the input node u by first traversing deeply along a single path p_1 in the graph. It will visit one of u's neighbours v through the edge (u, v), and then one of v's neighbours z, and so on, until all the nodes connected to u by the edge (u, v) have been visited. These will all appear in the DFS tree in a single chain of descendants from u.

Once this path p_1 has been deeply searched, the algorithm may return to u to initiate a search on a different path p_2. By the very way DFS runs, p_1 must not be connected to p_2 in any other way besides through u, as otherwise those nodes would have been searched in the first path. The vertices on p_2 will appear in the DFS tree in another single chain of descendants from u. Hence, if u were to be removed, the vertices in p_1 and p_2 would be disjoint from one another. In general, if the root of the DFS tree has two or more children then the root is a cut vertex.

ii. Step II will either remove the node from the graph if u is found to a be a cut vertex in the result of step I, or it will return 0 as there are no pairs of vertices being disconnected from each other.

iii. Step III's correctness stems from the ideas behind the depth-first search as well. As was elaborated on in Lecture 7, the DFS algorithm will visit every node of a graph exactly once. If there are k connected components, the algorithm will require k calls to DFS-COUNT from the main function to completely traverse the tree. Here we are simply storing this value k, as well as keeping track of how many internal recursive calls were made in each of these k calls to store the number of nodes in each connected component.

iv. Finally, the formula in step IV is just a simplification of the discrete mathematics formula of finding pairings between sets. Given a set of cardinalities $S = \{v_1, v_2, \ldots, v_k\}$, the total sum of pairings between the sets is

$$\begin{aligned}
\text{pairings} &= v_1v_2 + \ldots + v_1v_k + v_2v_1 + \ldots + v_2v_k + \ldots + v_{k-1}v_k && (3)\\
&= v_1(v_2 + v_3 + \ldots + v_k) + v_2(v_1 + v_3 + \ldots + v_k) + \ldots && (4)\\
&= v_1(|S| - v_1) + v_2(|S| - v_2) + \ldots && (5)\\
&= \sum_{i=1}^{k} v_i \cdot (|S| - v_i) && (6)
\end{aligned}$$

As the pairings are unordered, we must divide by 2 to account for $(v_i, v_j) = (v_j, v_i)\,\forall\, i \neq j$.

$$\text{pairings} = \frac{1}{2}\sum_{i=1}^{k} v_i \cdot (|S| - v_i) \quad (7)$$

c) Step I only requires one traversal of the graph G, which by using DFS takes $O(n + m)$ time. Step II's time complexity was noted above, and is $O(\deg(v))$ time. Step III requires one

further DFS traversal of the graph G' which is upperbounded by $O(n+m)$. Finally, step IV requires $|S|$ constant-time calculations which in its worst case will be upperbounded by $O(n)$. Therefore, the total time complexity of the algorithm is $O((n+m)+n+\deg(v)) \approx O(n+m)$, as $\deg(v) \leq n-1$.

A Finding all cut vertices in a graph

In general, we can detect the cut vertices of a graph $G(V, E)$ by computing and storing two values for each node $u \in V$: the level of the node, and `down_and_up`, defined as the highest level the node can reach by taking DFS edges down the tree and then one back edge up. Then, a node u is a cut vertex if and only if for all children v of u, the lowest level v can reach using downwards DFS edges and one back edge is still greater than its own level.

Lemma 1. *For a simple connected undirected graph G, a node u is a cut vertex if and only if for all children v of u in the DFS tree of G, `down_and_up[v]` $\geq$ `level[u]`.*

Proof. $\implies$: Running the DFS algorithm on a graph G produces a tree T. If u is a cut vertex on level ℓ in the DFS tree, there must be no edges in T which a descendant of u can use to travel to a lower level than ℓ. Therefore, for each child v of u, `down_and_up`$[v]$ must be greater than `level`$[u]$.
$\impliedby$: In the DFS tree T, if for all children v of u, `down_and_up[v]` $\geq$ `level[u]`, there must be no paths in G which connect the subgraphs induced by $G[V - u]$. Therefore, u is a cut vertex. □

The following algorithm, a variant of Tarjan's Algorithm for strongly connected components, is explained thoroughly in Question 7 of Tutorial 7, and its correctness relies on the proof of Lemma 1. Some pseudocode is provided for this algorithm below. Note that in Tarjan's algorithm, rather than storing the `level` of each node, a simple time counter is used which stores the `discovery_time` of the node. These are equivalent for the purposes of finding cut vertices.

```
1:  function CUT-VERTICES(G)
2:      result ← ∅
3:      for u ∈ G do
4:          disc[u] ← −1                                  ▷ discovery_time
5:          down_up[u] ← −1
6:          parent[u] ← −1
7:      time ← 0
8:
9:      for u ∈ G do
10:         if disc[u] = −1 then                          ▷ if u is not visited
11:             DFS-TARJAN(u, time, result)
12:     return result
```

```
1:  function DFS-TARJAN(G, u, time, result)
2:      disc[u] = down_up[u] = time
3:      time ← time + 1
4:      children ← 0
5:
6:      for each vertex v adjacent to u do
```

```
7:        if disc[v] = −1 then                                  ▷ if v is not visited
8:            children ← children + 1
9:            parent[v] ← u
10:           DFS-TARJAN(v, time)
11:           down_up[u] ← min{down_up[u], down_up[v]}
12:
13:           if parent[u] = −1 and children > 1 then
14:               result.append(u)
15:           if parent[u] ≠ 1 and down_up[v] ≥ disc[u] then
16:               result.append(u)
17:       else if v ≠ parent[u] then                            ▷ ∃ a back edge (u, v)
18:           down_up[u] = min{down_up[u], disc[v]}
19:   return
```