COMP20007 - Assignment 1

Lucas Fern (1080613)

May 30, 2020

Question 1

```
begin
    Data: equation = a string comprised of parenthesis, arithmetic operators and any digits 0-9

    digits ⟵ CreateStack(), operators ⟵ CreateStack()
    last_stack ⟵ None

    while next = NextChar(equation) do
                /* NextChar() iteratively returns the next character in a string.  */
        if IsDigit(next) and last_stack = operators then
                /* IsDigit() returns True when called on an integer, False otherwise.  */
            digits.push(next)
            last_stack ⟵ digits
        else if next ∈ {(, +, −, ∗, /} then
            operators.push(next)
            last_stack ⟵ operators
        else if next ∈ {)} then
            try
                d_2 ⟵ digits.pop()
                d_1 ⟵ digits.pop()
                op ⟵ operators.pop()
            except Error
                return NotWellFormed
            end

            if op ∉ {(} then
                digits.push(Eval(d_1 + op + d_2))
                /* Eval() evaluates an arithmetic expression provided as a string, string
                addition is assumed to work as concatenation.  */
            else
                return NotWellFormed
            end

            try
                open_parenthesis ⟵ operators.pop()
            except Error
                return NotWellFormed
            end
            if open_parenthesis ∉ {(} then
                return NotWellFormed
            end
        else
            return NotWellFormed
        end
    end

    try
        answer ⟵ digits.pop()
    except Error
        return NotWellFormed
    end
    if answer and IsEmpty(digits) and IsEmpty(operators) then
        return answer
    else
        return NotWellFormed
    end
end
```

Algorithm 1: Determining Well Formed Arithmetic Expressions

Question 3b

A linearization of a Directed Acyclical Graph (DAG) is a linear arrangement of the nodes such that all pointers between elements in the graph are pointing to later elements in the linearization. In the tree trimming problem, we are looking for a graph where a linearization exists such that every vertex - or tree - can be walked to on a single walk of the graph. This implies that every vertex in the linearization must have an edge to its immediate successor, as otherwise traversing the linearization would involve skipping over vertices that could not be visited later due to the nature of directed acyclical graphs.

From this it can be concluded that a linearization of any one route where it is possible to trim all the trees is unique, as switching any two nodes in the linearization where every neighbour is connected by an edge would result in a pointer pointing backwards, invalidating the linearization.

Thus to solve the tree trimming problem, we must find *any* linearization of the given graph and check if it has the property where every vertex has an outgoing edge to its immediate successor. It is known that the topological sort algorithm can produce a linearization of a DAG in $O(V + E)$ time, where the basic operation is taken to be a step through an adjacency list, V is the number of nodes and E is the number of edges. This is done by implementing a depth-first search of the graph - in this case, starting at the top of the mountain, though this isn't strictly necessary - and whenever a node is popped off the depth first search, adding it to the start of the linearization (effectively building it in reverse order). In order to make this depth first search run in $O(V + E)$ time, it was run on an adjacency list instead of a matrix, as in the list we are only recording when edges do exist, rather than having to check whether an edge does exist for every combination of vertices.

Once the linearization is found, another function is called to check whether each node has an outgoing edge to its immediate successor. For this operation to run in $O(V)$ time, it was necessary to read the edges into an adjacency matrix as well as the previously used lists, since now for neighbouring nodes a and b in the linearization, $\textsc{AdjacencyMatrix}[a][b]$ can be used to check for an edge from a to b in linear $(O(1))$ time, with the basic operation being accessing an element of an array, where in the case of using lists, this would take $O(E)$ time, as a vertex may have up to E edges.

Once the linearization is deemed valid or invalid, the function can return. Thus the total time complexity for `is_single_run_possible()` is $O(V + E) + V \cdot O(1) = O(V + E)$.

NB: Checking whether a linearization is a valid run can be done in $O(E)$ time with adjacency lists as if a single node has $n \leq E$ edges, all the others must have at most $E - n$, meaning the total amount of nodes checked is at most E. This results in the same $O(V + E)$ time complexity and it was chosen to use the adjacency matrix approach which runs in $O(V)$ time under the assumption that V would be less than E in most cases.