Course: Computer Engineering
Subject: Discrete Mathematics (MDISC)

US19 - Sprint 3

Selection Sort Algorithm
Kruskal's Algorithm
Dijkstra's Algorithm
Algorithm Complexity

G141; Class 1DM/1DN
Igor Coutinho - 1230543
Rafael Barbosa - 1230544
Daniel Silva - 1231046
Samara Miranda - 1230432

June 9, 2024

Contents

1 Introduction **3**

2 Theoretical Background **4**
2.1 Selection Sort . . . 4
2.2 Kruskal's Algorithm . . . 5
2.3 Dijkstra's Algorithm . . . 6
2.4 Algorithm Complexity . . . 7
2.5 Big O Notation in Algorithm Complexity Analysis . . . 8

3 US13 **9**
3.1 Pseudo Code for US13 . . . 9
3.2 Complexity Analysis of the Pseudo Code for US13 . . . 10
3.2.1 Analysis Pseudo Code Find Minimum Spanning Tree . . . 10
3.2.2 Analysis Pseudo Code findRoot . . . 10
3.2.3 Analysis code for the modifiedUnion . . . 11

4 US17 **12**
4.1 Pseudo Code for US17 . . . 12
4.2 Complexity Analysis of the Pseudo Code for US17 . . . 13
4.2.1 Analysis Pseudo Code computeShortestPathsUS17 . . . 13
4.2.2 Analysis Pseudo Code minDistance() . . . 13

5 US18 **14**
5.1 Pseudo Code for US18 . . . 14
5.2 Complexity Analysis of the Pseudo Code for US18 . . . 15
5.2.1 Analysis Pseudo Code computeShortestPathsUS18 . . . 15
5.2.2 Analysis Pseudo Code minDistance() . . . 15

1 Introduction

The present file was developed within the scope of the Discrete Mathematics course (MDISC), during Sprint 3, as part of the Integrative Project, in the second semester of the Computer Engineering program.
Throughout the file, it is possible to verify and understand the different topics that are addressed (Selection Sort Algorithm, Kruskal's Algorithm, Dijkstra's Algorithm, and Algorithm Complexity), with pseudo code provided for each of the topics.
Finally, an analysis of US13, US17, and US18 is presented, based on the number of iterations and the accumulated complexity.

2 Theoretical Background

2.1 Selection Sort

The Selection Sort algorithm is a simple and efficient method for sorting a list of elements. It belongs to the category of comparison-based sorting algorithms and is known for its ease of implementation.

Algorithm:

1. **Selection of the Smallest Element:** The algorithm begins by finding the smallest element in the unsorted list.
2. **Swap:** This smallest element is then swapped with the first element of the list.
3. **Updating the Unsorted List:** After the swap, the first element of the list is considered sorted, and the unsorted list is reduced by one element, as we now know that the first element is the smallest.
4. **Repetition:** We repeat steps 1 to 3 for the remaining unsorted list, finding the next smallest element and swapping it with the second element of the list, then with the third, and so on, until the entire list is sorted.

Complexity:

The Selection Sort algorithm has a time complexity of $O(n^2)$, where n is the number of elements in the list. This is because, in each iteration, we need to traverse the unsorted list to find the smallest element. Despite its simplicity, Selection Sort is not efficient for large datasets compared to more sophisticated algorithms like Merge Sort or Quick Sort. However, it is useful in situations where implementation simplicity is more important than performance.

Conclusion:

Selection Sort is a simple and easy-to-understand sorting algorithm. Although not the most efficient choice for large datasets, its simplicity makes it useful in situations where performance is not critical and quick implementation is required.

2.2 Kruskal's Algorithm

Kruskal's algorithm is a greedy algorithm used to find the minimum spanning tree (MST) of a weighted undirected graph. A minimum spanning tree of a graph is a subset of its edges that connects all the vertices with the minimum possible total weight, without forming any cycles.

Algorithm:

1. **Initialization:** Begin with an empty set of edges and a forest of disjoint trees, where each vertex is a separate tree.
2. **Edge Sorting:** Sort all the edges of the graph in non-decreasing order of their weights.
3. **Edge Selection:** Iterate through the sorted edges and select each edge one by one, considering the sorted order. If adding the edge does not create a cycle in the current forest, add it to the set of selected edges. Otherwise, discard it.
4. **Tree Connection:** Repeat step 3 until all the trees in the forest are connected into a single tree.

Conclusion:

Kruskal's algorithm is efficient and guarantees obtaining a minimum spanning tree. Its time complexity is determined by the sorting of edges, which is typically $O(E \log E)$, where E is the number of edges in the graph.

2.3 Dijkstra's Algorithm

Dijkstra's Algorithm is a classic shortest path algorithm used to find the shortest path between a source node and all other nodes in a weighted directed or undirected graph. It is widely used in network routing problems, such as determining the shortest path in navigation systems and packet routing in computer networks.

Algorithm:

1. **Initialization:** Start with a graph, where each node initially has an associated cost of infinity, except the source node, which has a cost of zero. Maintain a list of unvisited nodes.
2. **Selection of the Nearest Node:** Select the unvisited node with the lowest current cost. This will be the nearest node to the source node.
3. **Updating Costs:** For each unvisited neighbor of the selected node, calculate the total cost to reach that neighbor through the selected node. If this total cost is less than the cost currently associated with the neighbor, update the cost.
4. **Marking the Selected Node as Visited:** After calculating the costs for all neighbors of the selected node, mark it as visited.
5. **Repetition:** Repeat steps 2 to 4 until all nodes have been visited.
6. **Result:** After visiting all nodes, the cost associated with each node will be the cost of the shortest path from the source node to that node.

Complexity:

Dijkstra's algorithm has a time complexity of $O((V+E)\log V)$, where V is the number of nodes and E is the number of edges in the graph. This is due to the use of a priority queue (e.g., a binary heap) to select the next nearest node. If implemented with an adjacency list and a binary heap for the priority queue, Dijkstra's algorithm is efficient even for large graphs.

Conclusion:

Dijkstra's algorithm is a valuable tool for finding the shortest path in a weighted graph.

2.4 Algorithm Complexity

Algorithm complexity refers to the measure of resources, such as time and space, required by an algorithm to execute a task. It serves as a yardstick for comparing different algorithms and determining their suitability for solving specific problems. By understanding algorithm complexity, we can make informed decisions about algorithm selection, ultimately optimizing software performance.

Time Complexity

Time complexity quantifies the number of computational steps or operations performed by an algorithm relative to the input size. It provides insights into how the algorithm's execution time scales with input growth. The Big O notation, denoted as $O(f(n))$, is commonly used to express time complexity, where $f(n)$ represents an upper bound on the number of operations as a function of input size n. For example, $O(n^2)$ denotes quadratic time complexity, indicating that the number of operations grows quadratically with input size.

Space Complexity

Space complexity measures the amount of memory or storage space required by an algorithm to solve a problem. It encompasses factors such as variable storage, data structures, and auxiliary space. Similar to time complexity, space complexity is also expressed using Big O notation, reflecting the maximum amount of space used by the algorithm as a function of input size.

Analysis Techniques

Various techniques are employed to analyze algorithm complexity, including:

- **Asymptotic Analysis:** Evaluating the behavior of an algorithm as the input size approaches infinity. It focuses on the dominant term in the time or space complexity function, providing an overall understanding of algorithm efficiency.
- **Worst-case, Best-case, and Average-case Analysis:** Assessing algorithm performance under different scenarios, such as the maximum, minimum, and average input conditions. While worst-case analysis ensures algorithm reliability, best-case and average-case analysis offer insights into typical performance.
- **Mathematical Modeling:** Formulating mathematical equations to represent algorithm complexity, facilitating precise analysis and comparison.

2.5 Big O Notation in Algorithm Complexity Analysis

Big O notation is a fundamental tool in algorithm complexity analysis. It describes the asymptotic behavior of an algorithm concerning the size of its input, providing a way to quantify the growth of execution time or space usage as the input size increases.

Definition

Formally, we say an algorithm has complexity $O(g(n))$ if, for sufficiently large inputs, the number of operations performed by the algorithm is upper-bounded by a constant times the function $g(n)$. In other words, the algorithm's execution time or space usage does not grow faster than the function $g(n)$ as n (the input size) tends to infinity.

Examples

Big O notation is commonly used to describe three common types of complexity:

- **Constant ($O(1)$):** An algorithm with constant complexity performs a constant number of operations, regardless of the input size. For example, accessing the first element of a list has complexity $O(1)$.
- **Linear ($O(n)$):** An algorithm with linear complexity performs a number of operations proportional to the input size. For example, a linear search in an unsorted list has complexity $O(n)$, where n is the number of elements in the list.
- **Quadratic ($O(n^2)$):** An algorithm with quadratic complexity performs a number of operations proportional to the square of the input size. For example, a quadratic search in a two-dimensional array has complexity $O(n^2)$, where n is the size of one side of the array.
- **Logarithmic ($O(\log n)$):** An algorithm with logarithmic complexity performs a number of operations proportional to the logarithm of the input size. For example, a binary search in a sorted list has complexity $O(\log n)$.

3 US13

3.1 Pseudo Code for US13

Algorithm 1 Find Minimum Spanning Tree, findRoot and modifiedUnion

```
1: procedure FINDMINIMUMSPANNINGTREE(edges, mstEdges)
2:     sort edges by weight in ascending order ▷ Sort edges by weight in ascending order
3:     initialize parent ▷ Initialize the parent array
4:     initialize rank ▷ Initialize the rank array
5:     initialize set of vertices ▷ Initialize the set of vertices
6:     for all edge in edges do ▷ For each edge in edges
7:         add edge's origin to the set of vertices ▷ Add edge's origin to the set of vertices
8:         add edge's destination to the set of vertices ▷ Add edge's destination to the set of vertices
9:     end for
10:    for all edge in edges do ▷ For each edge in edges
11:        rootX = findRoot(parent, edge's origin) ▷ Find the root of the edge's origin
12:        rootY = findRoot(parent, edge's destination) ▷ Find the root of the edge's destination
13:        if rootX ≠ rootY then ▷ If the roots are different
14:            add edge to mstEdges ▷ Add the edge to mstEdges
15:            modifiedUnion(parent, rank, rootX, rootY) ▷ Perform the union of the two sets
16:            if size of mstEdges == size of vertices - 1 then ▷ nº of edges in mstEdges equals nº of vertices minus one
17:                break ▷ Break the loop
18:            end if
19:        end if
20:    end forreturn mstEdges ▷ Return the set of edges in the minimum spanning tree
21: end procedure

22: procedure FINDROOT(parent, node)
23:    if node is not in parent then
24:        add node to parent with itself as its own parent ▷ Add the node to the parent if not present, with itself as parent
25:    end if
26:    while parent of node is not node do ▷ While the parent of the node is not the node itself
27:        node ← parent of node ▷ Update the node to the parent of the current node
28:    end while
29:    return node ▷ Return the root node
30: end procedure

31: procedure MODIFIEDUNION(parent, rank, x, y)
32:    rootX ← findRoot(parent, x) ▷ Find the root of node x
33:    rootY ← findRoot(parent, y) ▷ Find the root of node y
34:    if rootX = rootY then ▷ If both nodes have the same root, they are already in the same set
35:        return ▷ Return without performing any modification
36:    end if
37:    if rank of rootX < rank of rootY then ▷ Compare the ranks of the roots
38:        set parent of rootX to rootY ▷ Attach the roots of the trees with lower and higher rank
39:    else if rank of rootX > rank of rootY then
40:        set parent of rootY to rootX
41:    else ▷ If both roots have the same rank
42:        set parent of rootY to rootX ▷ Attach the root of y to the root of x
43:        increment rank of rootX by 1 ▷ Increment the rank of x
44:    end if
45: end procedure
```

3.2 Complexity Analysis of the Pseudo Code for US13

3.2.1 Analysis Pseudo Code Find Minimum Spanning Tree

Lines of Code	Number of Iterations	Cumulative Complexity
sort edges by weight in ascending order	$\|n\| \log \|n\|$	$O(\|n\| \log \|n\|)$
initialize parent	1	$O(1)$
initialize rank	1	$O(1)$
initialize set of vertices	1	$O(1)$
for each edge in edges	$\|n\|$	$O(\|n\|)$
add edge's origin to the set of vertices	$\|n\|A$	$O(\|n\|)$
add edge's destination to the set of vertices	$\|n\|A$	$O(\|n\|)$
for each edge in edges	$\|n\|$	$O(\|n\|)$
rootX = findRoot(parent, edge's origin)	$\|n\|C$	$O(\|n\|)$
rootY = findRoot(parent, edge's destination)	$\|n\|C$	$O(\|n\|)$
if rootX $\neq$ rootY	$\|n\|C$	$O(\|n\|)$
add edge to mstEdges	$\|n\|A$	$O(\|n\|)$
modifiedUnion(parent, rank, rootX, rootY)	$\|n\|C$	$O(\|n\|)$
if size of mstEdges == size of vertices - 1	$\|n\|C$	$O(\|n\|)$
break	$\|n\|C$	$O(\|n\|)$
return mstEdges	$1R$	$O(1)$

Table 1: Analysis code for the Minimum Spanning Tree

3.2.2 Analysis Pseudo Code findRoot

Lines of Code	Number of Iterations	Cumulative Complexity
add *node* **to** *parent* **with itself as its own parent**	1	$O(1)$
while *parent* **of** *node* **is not** *node*	n	$O(n)$
update *node* **to the parent of the current** *node*	n	$O(n)$
return *node*	$1R$	$O(1)$

Table 2: Analysis code for the findRoot

3.2.3 Analysis code for the modifiedUnion

Lines od Code	Number of Iterations	Cumulative Complexity
sort edges by weight in ascending order	$\|n\| \log \|n\|$	$O(\|n\| \log \|n\|)$
initialize parent	1	$O(1)$
initialize rank	1	$O(1)$
initialize set of vertices	1	$O(1)$
for each edge in edges	$\|n\|$	$O(\|n\|)$
add edge's origin to the set of vertices	$\|n\|A$	$O(\|n\|)$
add edge's destination to the set of vertices	$\|n\|A$	$O(\|n\|)$
for each edge in edges	$\|n\|$	$O(\|n\|)$
rootX = findRoot(parent, edge's origin)	$\|n\|C$	$O(\|n\|)$
rootY = findRoot(parent, edge's destination)	$\|n\|C$	$O(\|n\|)$
if rootX $\neq$ rootY	$\|n\|C$	$O(\|n\|)$
add edge to mstEdges	$\|n\|A$	$O(\|n\|)$
modifiedUnion(parent, rank, rootX, rootY)	$\|n\|C$	$O(\|n\|)$
if size of mstEdges == size of vertices - 1	$\|n\|C$	$O(\|n\|)$
break	$\|n\|C$	$O(\|n\|)$
return mstEdges	$1R$	$O(1)$

Table 3: Analysis code for the modifiedUnion

4 US17

4.1 Pseudo Code for US17

Algorithm 2 computeShortestPathsUs17 and minDistance()

```
1: procedure computeShortestPathsUs17(graph, startVertex)
2:     initialize distances array with infinity for all vertices ▷ Initialize the distances array with infinity for all vertices
3:     initialize predecessors array with -1 for all vertices ▷ Initialize the predecessors array with -1 for all vertices
4:     initialize visited array with false for all vertices ▷ Initialize the visited array with false for all vertices
5:     set distance of startVertex to 0 ▷ Set the distance of the startVertex to 0
6:     for i from 0 to numVertices - 1 do
7:         u ← minDistance() ▷ Find the vertex with the minimum distance
8:         mark u as visited ▷ Mark vertex u as visited
9:         for v from 0 to numVertices do
10:            if v is not visited and there is an edge from u to v then ▷ Check if vertex v is not visited and if there is an edge from u to v
11:                if distance to u is not infinity and distance to u + edge from u to v is less than distance to v then
12:                    update distance to v ▷ Update the distance to vertex v
13:                    set predecessor of v to u ▷ Set the predecessor of v to u
14:                end if
15:            end if
16:        end for
17:    end for
18: end procedure

19: procedure minDistance
20:    min ← Integer.MAX_VALUE ▷ Initialize minimum distance to a very large value
21:    minIndex ← −1 ▷ Initialize index of minimum distance vertex to -1
22:    for each vertex v from 0 to numVertices − 1 do ▷ Iterate over all vertices
23:        if vertex v is not visited and distance to v is less than or equal to min then ▷ Check if vertex is not visited and its distance is less than current minimum
24:            min ← distance to v ▷ Update minimum distance
25:            minIndex ← v ▷ Update index of minimum distance vertex
26:        end if
27:    end for
28:    return minIndex ▷ Return index of vertex with minimum distance
29: end procedure
```

4.2 Complexity Analysis of the Pseudo Code for US17

4.2.1 Analysis Pseudo Code computeShortestPathsUS17

Lines of Code	Number of Iterations	Cumulative Complexity
initialize distances array with infinity for all vertices	$1A$	$O(1)$
initialize predecessors array with -1 for all vertices	$1A$	$O(1)$
initialize visited array with false for all vertices	$1A$	$O(1)$
set distance of startVertex to 0	$1A$	$O(1)$
for (i from 0 to numVertices - 1)	n	$O(n)$
$u \leftarrow$ minDistance()	$(n)C$	$O(n)$
mark u as visited	$(n)A$	$O(n)$
for (v from 0 to numVertices)	n^2	$O(n^2)$
if (v is not visited and there is an edge from u to v)	n^2C	$O(n^2)$
if distance to u is not infinity and distance to u + edge from u to v is less than distance to v	n^2C	$O(n^2)$
update distance to v	n^2A	$O(n^2)$
set predecessor of v to u	n^2A	$O(n^2)$
minDistance()	nC	$O(n)$

Table 4: Analysis code for the computeShortestPathsUs17

4.2.2 Analysis Pseudo Code minDistance()

Lines of Code	Number of Iterations	Cumulative Complexity
min $\leftarrow$ Integer.MAX_VALUE	$1A$	$O(1)$
minIndex $\leftarrow -1$	$1A$	$O(1)$
For each vertex v from 0 to numVertices - 1 do	n	$O(n)$
If vertex v is not visited and distance to v is less than or equal to min	$(n)C$	$O(n)$
min $\leftarrow$ distance to v	$(n)A$	$O(n)$
minIndex $\leftarrow v$	$(n)A$	$O(n)$
End If	$(n)C$	$O(n)$
End For	n	$O(n)$
Return minIndex	$1R$	$O(1)$

Table 5: Analysis code for the minDistance()

5 US18

5.1 Pseudo Code for US18

Algorithm 3 computeShortestPathsUs18 and minDistance()

```
1: procedure ComputeShortestPathsUS18(matrix graph, array startVertices)
2:     initialize distances[] and predecessors[] arrays
3:     for i ← 0 to numVertices −1 do
4:         distances[i] ← ∞                                   ▷ Initialize distances to infinity
5:         predecessors[i] ← −1                               ▷ Initialize predecessors to -1
6:     end for
7:     for each startVertex in startVertices do
8:         distances[startVertex] ← 0                         ▷ Set distance to start vertex as 0
9:     end for
10:    for i ← 0 to numVertices −1 do
11:        u ← minDistance()                                  ▷ Find vertex with minimum distance
12:        if u = −1 then
13:            break                                          ▷ If no vertex found, exit loop
14:        end if
15:        visited[u] ← true                                  ▷ Mark vertex as visited
16:        for v ← 0 to numVertices −1 do
17:            if not visited[v] and graph[u][v] ≠ 0 and distances[u] ≠ ∞ and distances[u] + graph[u][v] < distances[v] then
18:                distances[v] ← distances[u] + graph[u][v]  ▷ Update distance
19:                predecessors[v] ← u                        ▷ Update predecessor
20:            end if
21:        end for
22:    end for
23: end procedure

24: procedure minDistance
25:    min ← Integer.MAX_VALUE                                ▷ Initialize minimum distance to a very large value
26:    minIndex ← −1                                          ▷ Initialize index of minimum distance vertex to -1
27:    for each vertex v from 0 to numVertices − 1 do         ▷ Iterate over all vertices
28:        if vertex v is not visited and distance to v is less than or equal to min then ▷ Check if vertex is not visited and
       its distance is less than current minimum
29:            min ← distance to v                            ▷ Update minimum distance
30:            minIndex ← v                                   ▷ Update index of minimum distance vertex
31:        end if
32:    end for
33:    return minIndex                                        ▷ Return index of vertex with minimum distance
34: end procedure
```

5.2 Complexity Analysis of the Pseudo Code for US18

5.2.1 Analysis Pseudo Code computeShortestPathsUS18

Lines of Code	Number of Iterations	Cumulative Complexity
For i from 0 to numVertices -1 do	n	$O(n)$
distances[i] $\leftarrow \infty$	$(n)A$	$O(n)$
predecessors[i] $\leftarrow -1$	$(n)A$	$O(n)$
For each startVertex in startVertices do	n	$O(n)$
distances[startVertex] $\leftarrow 0$	$(n)A$	$O(n)$
For i from 0 to numVertices -1 do	n	$O(n)$
$u \leftarrow$ minDistance()	$(n \cdot n)C + (n \cdot n)A$	$O(n^2)$
If $u = -1$ then	$(n)C$	$O(n)$
visited[u] $\leftarrow$ true	$(n)A$	$O(n)$
For v from 0 to numVertices -1 do	$n \cdot n$	$O(n^2)$
If not visited[v] and graph[u][v] $\neq 0$ and distances[u] $\neq \infty$ and distances[u] + graph[u][v] $<$ distances[v] then	$(n \cdot n)C$	$O(n^2)$
distances[v] $\leftarrow$ distances[u] + graph[u][v]	$(n \cdot n)A + (n \cdot n)Op$	$O(n^2)$
predecessors[v] $\leftarrow u$	$(n \cdot n)A$	$O(n^2)$

Table 6: Analysis code for the computeShortestPathUs18

5.2.2 Analysis Pseudo Code minDistance()

Lines of Code	Number of Iterations	Cumulative Complexity
min $\leftarrow$ Integer.MAX_VALUE	$1A$	$O(1)$
minIndex $\leftarrow -1$	$1A$	$O(1)$
For each vertex v from 0 to numVertices - 1 do	n	$O(n)$
If vertex v is not visited and distance to v is less than or equal to min	$(n)C$	$O(n)$
min $\leftarrow$ distance to v	$(n)A$	$O(n)$
minIndex $\leftarrow v$	$(n)A$	$O(n)$
End If	$(n)C$	$O(n)$
End For	n	$O(n)$
Return minIndex	$1R$	$O(1)$

Table 7: Analysis code for the minDistance()