3805ICT Advanced Algorithms
Assignment Part One

Ethan Leet
s5223103
ethan.leet@griffithuni.edu.au

April 27, 2022

Contents

0 Introduction **4**
- 0.1 Overview . . . 4
- 0.2 Compilation and Runtime . . . 4

1 Geometric Problem - N Lines Intersections **5**
- 1.1 Problem Statement . . . 5
- 1.2 Algorithm Description . . . 5
- 1.3 Complexity . . . 6
  - 1.3.1 Space Complexity . . . 6
  - 1.3.2 Time Complexity . . . 6
- 1.4 Experiments and Results . . . 7

2 Bellman-Ford Algorithm **8**
- 2.1 Problem Statement . . . 8
- 2.2 Algorithm Description . . . 8
- 2.3 Complexity . . . 9
  - 2.3.1 Space Complexity . . . 9
  - 2.3.2 Time Complexity . . . 9
- 2.4 Experiments and Results . . . 9

3 Adjacency List & Adjacency Matrix Alternative For Large Graphs **10**
- 3.1 Problem Statement . . . 10
- 3.2 Algorithm Description . . . 10
- 3.3 Complexity . . . 12
  - 3.3.1 Space Complexity . . . 12
  - 3.3.2 Time Complexity . . . 12
- 3.4 Experiments and Results . . . 13

4 Custom Structure for Unordered Integer Lists **14**
- 4.1 Problem Statement . . . 14
- 4.2 Algorithm Description . . . 14
- 4.3 Complexity . . . 15
  - 4.3.1 Space Complexity . . . 15
  - 4.3.2 Time Complexity . . . 15
- 4.4 Experiments and Results . . . 16

5 Ladder-Gram **17**
- 5.1 Problem Statement . . . 17
- 5.2 Algorithm Description . . . 17
- 5.3 Complexity . . . 18
  - 5.3.1 Space Complexity . . . 18
  - 5.3.2 Time Complexity . . . 18
- 5.4 Experiments and Results . . . 19

6 Longest Circular Sequence **20**
6.1 Problem Statement 20
6.2 Introduction 20
6.3 Algorithm Description 21
6.4 Complexity 22
6.4.1 Space Complexity 22
6.4.2 Time Complexity 23
6.5 Experiments, Results and Comparisons 24
6.6 Conclusion 25

0 Introduction

0.1 Overview

This assignment involved writing c++ code to solve six Advanced Algorithm based problems. Test cases were then created to prove the validity of the algorithm used for each of the six problems. Each question is broken down into several sub sections throughout this report. The sub sections include; problem statement, algorithm description, cursory complexity analysis, and experiments and results. Question six has the extra requirement of adding an introduction, comparisons and a conclusion to the report.

0.2 Compilation and Runtime

All programs included in this submission compile without errors nor warnings using clang++ and the c++ standard library 14. Each program can be compiled using the following command from the question folder, using the question number in place of x.

clang++ main.cpp -std=c++14 -o questionx -Ofast

Alternatively, each question also contains a make file which runs the above command. Finally, there is also a make file in the parent folder which will make every program in this submission and place the binaries in the respective question folders. To use the make command, CMake must be installed and then the following command can be executed from either the parent or child folder.

make

After successful compilation programs can then be executed with the following command, using the question number in place of x. Note that in order to execute the following command you must be inside the question folder.

./questionx

Note, that certain questions require command line arguments to be passed at execution time. These arguments can be passed as follows:

./questionx arg1 arg2 argx

1 Geometric Problem - N Lines Intersections

1.1 Problem Statement

A common geometric problem is, given a set of N lines, how many intersect. The brute force algorithm is $O(n^2)$. Design an $O(n \log_n)$ algorithm and implement it in C++ for the case where there are only vertical or horizontal lines. Derive and demonstrate the efficiency of your algorithm.

1.2 Algorithm Description

In order to achieve the $O(n \log_n)$ complexity restriction for this problem a sweep line algorithm was chosen. This algorithm can be described below:

Algorithm 1 Sweep Line Algorithm

```
active lines ← all lines
count ← 0
while active lines exist do
    if Leftmost point of line then
        count ← count intersections
    else
        remove from active lines
    end if
end while
return count
```

There are three main data structures used in this program and are of type struct. The first is a struct of points which holds the x and y coordinate of a point in 2D space. The second is a struct of lines which represents a set of points to form a line in 2D space. The final main structure is a struct of events. Each line is treated as an event which contains the initial x and y coordinates of the line (sorted), the total coordinates of the line (as a line struct) and a boolean attribute which indicates if the line is currently active, meaning we have read the initial points of the line but have not reached the end of the line.

In addition to these structures four main functions were also implemented and are outlined below:

1. Read File:
   This function reads a text file from the command line which contains the coordinates of one line per line in the format x_1, y_1, x_2, y_2. It then determines the type of the line (vertical, horizontal or invalid) and orders each determined line from smallest to largest points (to ensure the lowest points are ordered first). It then adds each line as an event, sorts all events based on the sorting function below and finally returns the ordered and sorted structure of line events.

2. Event Sort:
   In order to sort a structure of line events the cpp std::sort algorithm was used with a custom comparator function. This function firstly sorts points based on x coordinate, if the x coordinates are equal it will sort based on y coordinate. If the y coordinate is also equal it will then sort based upon the boolean attribute which indicates if the line is active.

3. Intersections:
   This function calls the below Counter Clockwise algorithm with two lines to determine if there is an intersection present. Due to the events already being sorted, if the Counter Clockwise algorithm returns a result > 0 an intersection is not present. This functions returns that an intersection is found in all other cases.

4. Counter Clockwise:
   This algorithm takes three points and uses the below formula to determine whether they form a counter clockwise angle. The formula results in twice the signed area of a triangle, if the area is positive then it is counter clockwise, if it is negative then it is clockwise and an intersection is found.

$$\begin{vmatrix} p_x & p_y & 1 \\ q_x & q_y & 1 \\ r_x & r_y & 1 \end{vmatrix} = (q_x - p_x)(r_y - p_y) - (r_x - p_x)(q_y - p_y)$$

1.3 Complexity

1.3.1 Space Complexity

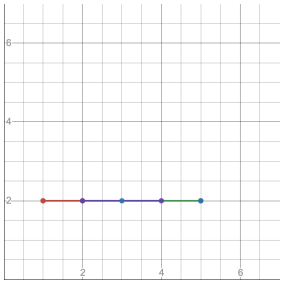
Each line that is read in from the text file creates a struct of points, a struct of lines and a struct of events. Each line is also represented in the form of a vector of all lines and a vector of active events. Initially, when these two vector are full they both hold n elements where n is the amount of lines. Each structure then holds n points. It is speculated that a total space complexity of $4n$ can be expected for this program and can be bounded by $O(n)$.

1.3.2 Time Complexity

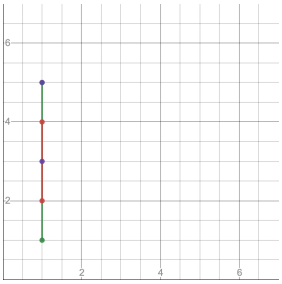
If we assume that there are n lines passed then each line is represented by $2n$ end points. This means that the sweep line algorithm evaluates the intersection at at most n points and has a run time of $2n$ or of order $O(n)$. At each point evaluated the sweep line algorithm uses the intersection and counter clockwise algorithms which simply return a result and therefore do not contribute to the sweep line algorithm complexity order. For the sweep line algorithm to operate each line must first be sorted. the std::sort operation was chosen with a comparator function that runs in $\Theta(1)$. Therefore, a total time complexity for this program is bounded by the std::sort library function which operates in $O(n \log_n)$.

1.4 Experiments and Results

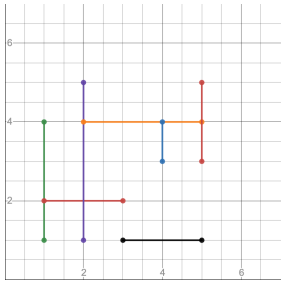
It was noted that there exist five possible test cases for this problem. When there are; overlaps, no overlaps, only horizontal overlaps, only vertical overlaps, and when a square or rectangle is formed. Each test case was implemented, a visual representation of each test case and tabular results of each test case can be seen below.



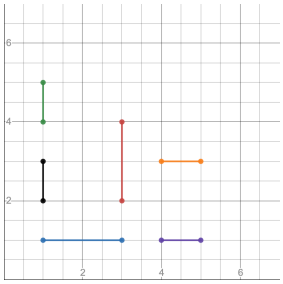
Horizontal Test



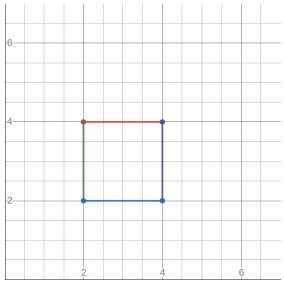
Vertical Test



Overlap Test



No Overlap Test



Square Test

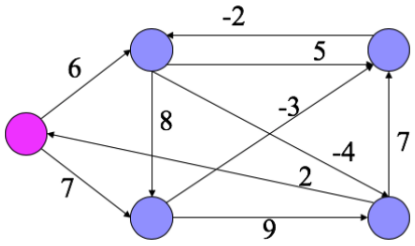
Test Case	Pass/Fail	CPU Time (microseconds)
Horizontal	Pass	111
Vertical	Pass	116
Overlap	Pass	110
No Overlap	Pass	113
Square	Pass	103

A milliseconds response was first used however this returned zero for all cases. As such, a microseconds counter was implemented. Note that using microseconds resulted in inconsistent output times and can not be used to accurately and consistently measure time.

2 Bellman-Ford Algorithm

2.1 Problem Statement

Write a C++ program that uses the Bellman-Ford algorithm to find the shortest paths from the pink node to all other nodes:



2.2 Algorithm Description

The Bellman-Ford algorithm can be seen below.

Algorithm 2 Bellman-Ford Algorithm

```
all distances ← infinity
for i ← 1 to vertices − 1 do
    all distances ← relaxed edges
end for
for i ← 0 to all edges do
    check for negative weight cycles
    if negative weight exists then
        return ERROR
    end if
end for
return all distances
```

In addition to this algorithm, a structure to hold all edges was implemented of type struct. This structure contains the source node, destination node and weight of that path.

A function to read the file containing the graph was also implemented. This function opens the graph that is passed to the program and adds each node and its paths to a vector of edges. The function returns this vector as well as the total edges and vertices present in the graph.

2.3 Complexity

2.3.1 Space Complexity

This program has two main structures which make up the total space complexity. The first is a vector of integers and the second is a vector of edges. The edges structs hold three integers, giving a total complexity of $3 \times n$ where n is the amount of edges in the graph and is bounded by $O(n)$.

2.3.2 Time Complexity

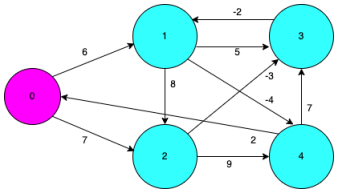
The read file function contains a single order loop which reads every line in the file, giving a complexity of $O(n)$ for this part of the program.

The Bellman-Ford algorithm is bounded by the relaxation section of the algorithm. This complexity is $O(VE)$ where V is the amount of vertices present and E is the amount of edges present in the graph.

As the file which holds the graph has at most $V + 1$ lines, the total complexity for this program is $O(VE)$.

2.4 Experiments and Results

As each node is unlabelled the following labels were provided to each:



The shortest paths from the pink node to all other nodes is listed below:

Vertex	Shortest Path
Source –> Source	0
Source –> 1	2
Source –> 2	7
Source –> 3	4
Source –> 4	-2

The total run time for this program was measured at 112 microseconds.

Other trivial and non trivial graphs were also created and tested to prove the validity of this algorithm.

3 Adjacency List & Adjacency Matrix Alternative For Large Graphs

3.1 Problem Statement

Most graph computing algorithms assume that the adjacency matrix and adjacency lists can be stored in computer memory so the following 2 operations will be fast:

- Is vertex v connected to vertex u?
- Produce a list of all vertices connected to v.

However, the advent of very large graphs (e.g. 100,000 vertices and > 1,000,000 edges) prevents the memory storage of the adjacency matrix and standard adjacency lists for these graphs. Design and implement in C++ a data structure for storing such graphs that is able to effectively perform the 2 operations listed above. Demonstrate the efficiency of your data structure.

3.2 Algorithm Description

The data structure created is implemented as a c++ class. This structure implements all graphs as an adjacency list where it stores vertices and edges differently depending on if the graph is sparse or dense. The following calculation was chosen to determine density:

$$D = \frac{edges}{vertices \times (vertices - 1)}$$

When a graph is sparse it is stored as a standard adjacency list. When the graph is dense, meaning the above calculation > 0.5, it will inverse the edges present in the graph and store the result as an adjacency list, meaning that it will store all edges that are not present in the graph. The resulting adjacency list is stored as a c++ map type where the key of the map is the vertex and the value is the set of all edges which relate to that vertex. A detailed list of the structure members and their functions are as follows:

1. Determine Graph:
   This method reads a graph presented as a text file. It simply reads the number of vertices and edges and performs the above density calculation. The function calls either the construct sparse or construct dense methods based on this calculation.
2. Construct Sparse Graph:
   This method simply reads the text file passed by the user and adds all vertices and its edges to the structure.
3. Construct Dense Graph:
   This method reads the file $vertices$ amount of times adding each vertex and its set of edges into the structure one at a time. After the set of all edges for that vertex is found the method will scan this set of edges and store the inverse set into the adjacency list structure.

Algorithm 3 Construct Dense Graph

```
initialise adjacency list
initialise set
for i ← 0 to vertices do
    for item in file do
        if item = i then
            add item to set
        end if
    end for
    for j ← 0 to vertices do
        if j not in set then
            adjacency list at i ← j
        end if
    end for
end for
return adjacency list
```

4. Connected:
   This method conducts a binary search on the vertex v in search of vertex u. For the sparse graph this simply returns true if u is connected to v and false otherwise. For a dense graph it returns the opposite, if the binary search finds u in v it will return false, and true otherwise.

5. Neighbours:
   The final method returns the set of all neighbours of v. For a sparse graph this simply returns the set (value) of vertex v in the adjacency list. For a dense graph, the inverse set is returned.

Algorithm 4 Neighbours

```
initialise neighbours
for item in adjacency list do
    if sparse then
        add item to neighbours
    else
        if not item then
            add item to neighbours
        end if
    end if
end for
return neighbours
```

3.3 Complexity

3.3.1 Space Complexity

Due to this structure adding the inverse edges present for a density > 0.5 it can be seen that the worst case space complexity appears when we have a graph with a density of $0.4 \leq density \leq 0.6$. As such there are two cases:

Examining the case where $0.4 \leq density < 0.5$. This case is the upper limit of when a graph is classified as sparse. In this case, there can be at maximum $\frac{1}{2}E - 1$ edges present where E is the amount of edges needed for a density calculation of 0.5 and therefore being a dense graph.

The final case is where $0.5 \leq density \leq 0.6$. In this case the density calculation is the lower limit of what classifies a graph as dense. In this case there will be at maximum $\frac{1}{2}E$ edges present in the data structure.

In all other cases where a graph is either very dense or very spares the amount of edges saved in the structure will always be $< \frac{1}{2}E$. As such, a total space complexity for this data structure can be derived as $\frac{1}{2}E$ which equates to order $O(E)$.

3.3.2 Time Complexity

The two main methods required by this structure are:

1. Is vertex v connected to vertex u?
   For this method, a binary search is performed which has at most $log_2(first - last) + O(1)$ comparisons and is of order $O(\log n)$.

2. Produce a list of all vertices connected to v.
   In order to to produce a list of neighbours connected to vertex v an iterator is used to traverse all edges of v and is of order $O(n)$ where n is the amount of edges connected to vertex v.

In addition to these methods, the structure also reads a file and decides if the resulting graph is dense or sparse. The worst case scenario is when the resulting graph is dense. In this case, the function iterates V times where V is the amount of vertices. Inside this iteration the file of size n is read as well as a second iterator running V times. This gives us a total complexity of $V \times n \times V$ and is of order $O(V^2 n)$.

Therefore, the total runtime complexity for this structure is $O(V^2 n)$.

3.4 Experiments and Results

In order to test this data structure two types of sample graphs were created, one for a sparse graph and the other for a dense graph. A variety of different graphs were tested for each type, ranging from very small graphs, to graphs with density of $0.4 \leq density \leq 0.6$ to very large graphs. In each case each type of graph returned the correct results for each method and behaved according to the complexities listed above.

It was theorised that for certain large graphs the space complexity could be reduced by using a range of pairs to represent the edges present instead of the inverse set that was used. An example is as follows:

Assume vertex v contains edges $\{1, 2, 3, 4, 7, 8, 9\}$. Then the range of pairs represented in the graph would be $\{(1, 4), (7, 9)\}$.

For dense graphs of $density < 0.6$ this method would greatly increase performance compared to the method chosen for this problem. However, when the density starts to increase above 0.6 the performance will decrease compared to the method implemented. As the problem states we are storing very large graphs, this was concluded to mean very dense. As such this theorized method was not chosen due to the fact of the space requirement needed for storing almost fully connected graphs will be greater than the method that was chosen. As an example:

Assume v contains edges $\{1, 3, 5, 7, 8, 9\}$. Then the range of pairs represented in the graph would be $\{(1, 1), (3, 3), (5, 5), (7, 9)\}$ compared to $\{2, 4, 6\}$ with the method that was implemented.

If further research was to be conducted regarding storing space efficient graphs the implemented method and this theorized method could be used in tandem to provide a more space efficient solution.

4 Custom Structure for Unordered Integer Lists

4.1 Problem Statement

Design and implement in C++ a data structure for storing unordered lists of integers that:

a. Can store integers in the range 0 .. n where n is some upper bound.

b. Duplicate integers are not allowed in the list.

c. Is O(1) for add, delete, test for being in the list and iterating through the list.

d. Is O(k) (where k is the number of integers in the list) for clearing the list.

4.2 Algorithm Description

The data structure created for this problem consisted of a c++ class which used two std::array containers to implement the required methods. The first container is initialised to negative one (meaning empty) and holds the index of the value which is present in the second container.

The structure holds the following private members which are used in all methods; index container, values container and the current size of the list. Using the above implementation of two lists to represent one list structure, the following methods could be implemented in the required complexity:

1. O(1) Add: This method first performs an error check to ensure the item to be added is valid. It then updates the index container with the new item as well as adds the item to the values container at the position $length + 1$.

Algorithm 5 O(1) Add(item)

$index[value] \leftarrow length$
$value[length + 1] \leftarrow add\ val$

2. O(1) Delete:
   Error checking is firstly performed by this method to ensure the item is present. Assuming the value is present the method then replaces the item with the last item in the value container. It then updates the index of the removed value followed by updating the last element in the index container and updating the index containers size.

Algorithm 6 O(1) Delete(item)

$temp \leftarrow last\ item\ in\ value$
$value[temp] \leftarrow value[size - 1]$
$index[value[temp]] \leftarrow temp$
$vale[size], index[item] \leftarrow -1$
$update\ size$

3. O(1) Test for being in the list:
   Due to the nature of how this structure is represented by two lists, checking if an item exists can be done simply by returning the index container indexed by the item.

4. O(1) Iterator:
   Due to the nature of how this structure is represented by two lists, returning an iterator to an item can be done by simply returning the value container indexed by the iterator.

5. O(k) Clear:
   This method traverses the list structure k times where k is the amount of elements present in the structure. Upon each traversal it updates the index container indexed with the k_{th} value of the value container to minus one.

Algorithm 7 O(k) Clear

for $i \leftarrow 0$ **to** $size$ **do**
    $index[value[i]] \leftarrow -1$
end for

4.3 Complexity

4.3.1 Space Complexity

This program is bounded by some upper bound n and is therefore the maximum size of the container. This program uses two containers to represent one giving a total space of $n + n$ and is of order $0(n)$.

4.3.2 Time Complexity

The time complexity for this program was broken down into five methods whose complexity is described below:

1. O(1) Add:
   In order to add an item into the list structure the index container and values container must both be updated. Each update happens in constant time giving an overall complexity of $O(1)$ for this method.

2. O(1) Delete:
   To delete an item the method performs several constant time operations; it replaces the item with the last item in the values container $(O(1))$, updates the index of the removed value $(O(1))$, reduces the size of the values container $(O(1))$, updates the index container $(O(1))$ and finally reduces the size of the structure $(O(1))$. This gives a total complexity of $O(1)$ for this method.

3. O(1) Test for being in the list:
   The contains method simply indexes the item we are testing with the index container and returns true if the item is present (not equal to minus one) and is therefore constant, and is of order $O(1)$.

4. O(1) Iterator:
   This method simply indexes the item we wish to iterate with the values container and returns that item. It therefore operates in constant time and is of order $O(1)$. Note, that traversing the container is still in linear time. This means that iterating the entire structure will do n $O(1)$ calls to this method, where n is the size of the structure.

5. O(k) Clear:
   This method traverses the list structure k times where k is the amount of elements present in the structure. Upon each traversal it updates the index container indexed with the k_{th} value of the value container to minus one. This update is performed in constant time k times, giving a total complexity of $O(k)$.

4.4 Experiments and Results

An upper bound of fifty was chosen to conduct experiments and test this structure. The below figures outline the test results used for the five required methods as well as further edge case testing such as adding duplicates and trying to add out of bounds.

```
Testing Add:
List after adding {4, 6, 2, 31, 22, 17}
Item 1 = 4
Item 2 = 6
Item 3 = 2
Item 4 = 31
Item 5 = 22
Item 6 = 17

Testing Add Duplicate:
ERROR! 4 is already in list

Testing Add Out of Bounds:
ERROR! 55 must be > 0 and < 50
```

```
Testing Delete:
List after deleting {22, 6}
Item 1 = 4
Item 2 = 17
Item 3 = 2
Item 4 = 31

Testing if item is in list:
Looking for 31... 31 is in the list
Looking for 6... 6 is not in the list

Testing Clear:
List contains the following items...
```

5 Ladder-Gram

5.1 Problem Statement

The goal of a ladder-gram is to transform a source word into the target word on the bottom rung in the least number of steps. During each step, you must replace one letter in the previous word so that a new word is formed, but without changing the positions of the other letters. All words must exist in the supplied dictionary (dictionary.txt). For example, we can achieve the alchemist's dream of changing LEAD to GOLD in 3 steps or HIDE to SEEK in 6 steps. Minimise the number of steps.

5.2 Algorithm Description

It was noticed that this problem is a graph problem where the goal is to find the shortest path between two nodes. Two algorithms were then implemented; the first handles the trivial case where letters can simply be swapped from $source$ to $target$, and the second algorithm is a Bi-directional Breadth First Search.

To find a solution this program firstly uses the letter swap algorithm to traverse the $source$ word. It scans each letter in the source word looking for a match in the $target$ word. If there is a match it will scan the next letter. If no match is found it will swap the letters from $source$ to $target$ assuming the new word is in the dictionary. The algorithm will finally return the count of swaps if a successful match is found. If no match is found the program will then try swapping letters from $target$ to $source$, and then finally try the Bi-directional Breadth First Search assuming no match is found.

The Bi-directional Breadth First Search algorithm performs similarly to a Breadth First Search Algorithm except it starts the search from the start and end nodes. The idea behind this algorithm is to keep track of visited nodes from both searches. When a node is found by either search that has been visited before we can quickly calculate the answer by adding the path costs from the start node search to the end node search (decreased by one as the start node search already includes the node in visited). This essentially reduces the computations needed by half. The BFS and BBFS algorithms are described below:

Algorithm 8 Breadth First Search(queue, visited)

```
visited ← root
cost ← 0
while queue not empty do
    node ← dequeue
    cost ← cost + 1
    if node is goal then
        return cost
    end if
    visited ← node
    queue ← node
end while
```

Algorithm 9 Bi-directional Breadth First Search

```
visited ← 0
start ← BFS(front of queue, visited)
end ← BFS(back of queue, visited)
if solution then
    count ← path cost start visited
    count + path cost end visited
    return count − 1
end if
```

5.3 Complexity

5.3.1 Space Complexity

This problem holds in memory the whole length n of a dictionary. The Bi-directional Breadth First Search algorithm holds all visited nodes. As we are searching for words of certain lengths, the length of visited nodes will always be less of the size, giving an upper bound space complexity of $O(n)$.

5.3.2 Time Complexity

The letter swap function traverses both the source and target words of length l. The worst case is when the whole length l of the word is traversed, giving $2l$ or an upper bound of $O(l)$.

The Bi-Directional Breadth First Search algorithm does not reduce complexity of the solution compared to a Breadth First Search approach but it does create a more accurate tight bound in most cases. The worst case scenario is that the algorithm will not find a solution and therefore traverse n words n times from both the start and end nodes, giving a complexity of $n^2 + n^2$ which gives an upper bound of $O(n^2)$. However for

most solutions, the amount of nodes traversed is halved due to the start and end queues meeting in the middle. A more accurate complexity when a solution exists can be seen as $\frac{n^2}{2}$.

The function that reads in the dictionary operates in linear time giving a complexity of $O(n)$ for this function.

As such, a total worst case complexity of $O(n^2)$ can be expected for this program. However, a tight worst case complexity of $\frac{n^2}{2}$ can be expected for most cases where a solution exists.

5.4 Experiments and Results

A multitude of valid and invalid words were chosen as well as edge cases implemented to test the validity of this program. Note that when a result of -1 is listed no solution was found. The results for some test cases can be seen below:

Source	Target	Result	CPU Time (milliseconds)
lead	gold	3	10
hide	seek	6	9
hide	sook	5	9
ladder	batter	4	9
tester	finder	4	10
plumber	plaster	7	10
abandoned	aberrants	-1	10
gold	finder	Error	-
seek	dont	Error	-

6 Longest Circular Sequence

6.1 Problem Statement

The file dictionary.txt contains one word per line. Subsets of these words can ordered such that, with the exception of the first word, the second and third letter of each word is identical to the third last and second last of the preceding word. Words may only be used once within a sequence.

Design an algorithm and implement C++ programs that find circular sequences separately for words of length 4 through to words of length 15 characters.

Analyse in detail the problem and the performance of each of your algorithms. You must produce a detailed Latex/Word paper containing an Introduction, Algorithm Description, Experimental Results and Comparisons and a Conclusion.

6.2 Introduction

Upon theoretical inspection of this problem it was noted that the problem is essentially a graph searching problem where the node we are searching for is the youngest child of the parent node, that also links back to the parent via the given criteria. Investigation into such algorithms were conducted and a Depth First Traversal approach was chosen.

The remainder of this report highlights; the description of the algorithms used, the complexity of each individual algorithm as well as the program complexity as a whole, all experiments, results and comparisons that were conducted either directly through implementation or indirectly through theoretical analysis, and a conclusion highlighting the key findings and overall results.

6.3 Algorithm Description

In order to find the longest Circular Sequence of words a Depth First Search was conducted on all valid words. Valid words in this context refers to the current word length we are operating on. The depth First Search algorithm can be seen on the following page.

Algorithm 10 Depth First Search Algorithm

```
longestSequence ← 0
sequence ← NIL
stack ← valid words
while stack ≠ empty do
    word ← stack.pop
    if word creates sequence then
        sequence ← word
    end if
    if word creates circular sequence then
        longest sequence ← sequence length
    end if
end while
return longest sequence
```

It was noticed that the ordering in which these valid words were traversed effected the length of the circular sequence. As such, two algorithms were created to test this hypothesis.

The first algorithm uses the order of the valid words to traverse the graph, where the order is indicated by simply reading the dictionary file and saving each word of appropriate length in the order it is read.

Algorithm 11 Order Based Algorithm

```
while read dictionary do
    word = word in dictionary
    valid words ← word
end while
Perform DFS on valid words
```

The second algorithm initially reads in valid words the same way. However the valid words are shuffled after each traversal, the best sequence is saved and the process repeats. After one minute the algorithm will finish its current traversal and return the best sequence found.

Algorithm 12 Randomised Shuffle Algorithm

$best \leftarrow 0$
while $read\ dictionary$ **do**
    $word$ = $word\ in\ dictionary$
    $valid\ words \leftarrow word$
end while
while $timer < 60\ seconds$ **do**
    $shuffled \leftarrow randomize\ valid\ words$
    $result \leftarrow Perform\ DFS\ on\ shuffled\ words$
    if $result > best$ **then**
        $best \leftarrow result$
    end if
end while
return $best\ sequence$

It is worth noting that the Depth First Search algorithm is only applicable for words of length five or greater. For words of length four, no sequencing checks are required. This is due to the fact that the second and third letters are the same as second last and third last letters of any word of length four. For this case, each word was added to an unordered map structure where the key is the first and second letters of the word and the value is the word itself. The highest key is then returned as the best sequence for words of length four.

6.4 Complexity

6.4.1 Space Complexity

This program uses two main data structures, an unordered map containing strings and a vector of strings, and a vector of strings. The DFS algorithm also uses a stack which is initially the size of the vector of words. It is expected that each of the aforementioned containers operate in linear space as per the c++ standard library 14 documentation. It is expected that the program is bounded by the unordered map structure which contains n keys and an n sized vector of words, where n is the size of the dictionary. As previously stated, a vector is expected to operate in n space and therefore the unordered map holds $n \times n$ values. As such, an expected total space complexity of $O(n^2)$ can be expected.

6.4.2 Time Complexity

The time complexity for this program can be derived by breaking down each individual function:

Depth First Search:
The DFS algorithm operates on a graph containing V vertices and E edges, and has a total complexity of $O(V + E)$.

Loading of the Dictionary:
This function scans every item in the dictionary giving a total complexity of $O(n)$.

Circular Sequencing of Length Four:
This function scans every valid word which gives a total time complexity of $O(k)$ where k is the amount of valid words and < size of the dictionary. As such $O(k) < O(n)$.

Ordered Algorithm:
This function is executed eleven times and calls the load words function and DFS algorithm each time. It calls the Sequence of length four once. This gives a complexity of $11 \times n + 11 \times (V + E) + k$ and is of order $O(V + E)$ where $(V + E) > n$.

Randomised Shuffle:
This function is executed eleven times and calls the load words function and the DFS algorithm each time. It calls the Sequence of length four once. As this function also operates under a time restraint, it calls these functions S times where S is the amount of executions within the time constraint. This gives a complexity of $S \times (11 \times n + 11 \times (V + E) + k)$ and is of order $O(S(V + E))$ where $(V + E) > n$.

It can be seen by the above analysis that the program is bounded by the Randomised Shuffle function in order $O(S(V + E))$.

6.5 Experiments, Results and Comparisons

Upon testing the hypothesis originally stated in section 6.2, that the DFS would produce different results based upon different word orders, the following results were produced by the two algorithms:

Word Length	Algorithm	Sequence Length	CPU Time (seconds)
4	-	92	0.22
5	Ordered	2910	0.142
5	Randomised	2970	41.36
6	Ordered	5022	0.526
6	Randomised	5095	4.16
7	Ordered	5696	4.187
7	Randomised	7791	29.573
8	Ordered	9473	8.343
8	Randomised	9483	25.09
9	Ordered	2517	1.933
9	Randomised	4781	43.102
10	Ordered	1614	0.548
10	Randomised	2212	51.34
11	Ordered	1462	0.236
11	Randomised	1474	36.746
12	Ordered	859	0.076
12	Randomised	865	0.143
13	Ordered	244	0.029
13	Randomised	470	6.382
14	Ordered	1	0.013
14	Randomised	218	0.423
15	Ordered	69	0.008
15	Randomised	89	0.034

It can be seen in the tabular results above that overall the ordered DFS method finds consistently high results in a very quick time. For certain word lengths such as seven, nine and fourteen, this method proved quite inaccurate compared to the Randomised Shuffle algorithm. Upon investigation into this decline in accuracy, it could be seen and proved by example, that the DFS algorithm would return different results based upon the order to which the words were passed, confirming the aforementioned hypothesis.

If further testing were to be conducted using the order based DFS algorithm, perhaps a baseline model or could be implemented to help improve its accuracy. This model would firstly find out how many possible words could form a sequence via an exhaustive search, followed by repeat executions until a sequence of near that length is found in respect to some tolerance.

In addition to the above tests, a heuristic based approach was also tried and tested. This heuristic is similar to how the words of length four calculation operates, it ranks words

based on the amount of times the second and third letters appeared. It then ordered these words in the valid words structure based on that heuristic and conducted a DFS on that order.

This heuristic proved strong for words of length less than seven however inaccurate for words of a greater length. This implies that these longer words with a higher heuristic form a circular sequence but not as strong as a circular sequence compared to those of a lower heuristic value. Perhaps further testing could be conducted using this heuristic as a baseline and using the lowest to highest ranks of the heuristic for words of length seven and beyond rather than highest to lowest as was tested.

Interestingly, this problem was originally solved in the inverse way. Where the third last and second last letters of a word were matched to the first and second of the next. In this case, the heuristic explained above outperformed both the ordered and randomised algorithms in terms of length of sequence and CPU time.

6.6 Conclusion

Overall, a Depth First Search approach proved to be a highly effective for finding the longest circular sequence for words of some length in respect to time and accuracy. Due to the nature of a DFS algorithm expanding each node to its youngest child first, it can be guaranteed that this algorithm will find the longest circular sequence for that node. However, as seen in the results above, this is not guaranteed to be the absolute longest path across all nodes for that word length.

If this approach was to be chosen as the primary tool in finding the longest circular sequence a decision will need to be made regarding if close enough to the longest is good enough. If this is not the case, other methods should be implemented such as the methods theorized in section 6.4.

To conclude, the order based DFS found quite good results for most cases in a very short time. The Randomized Shuffle approach found similar results for most cases but greatly increased results for other cases at the cost of time and computational power. If a DFS based approach was to be chosen for this problem a ordered based is recommended when quick execution time is important. If execution time is not as important the Randomized Shuffle should be considered. Perhaps, an approach involving both algorithms could be considered to capture the increased results of the Randomized Shuffle and the low execution time of the Order based approach.