

# 3805ICT

## Advanced Algorithms

### Assignment 1

Candice Smith  
S5168666

#### Question 1

##### Problem:

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

##### Solution:

The algorithm used to solve this problem is the sweep line algorithm. The sweep line algorithm works first by ordering the lines by their leftmost x-coordinate. This is achieved by using a priority queue. The algorithm works based on 3 possible events; the sweep line reaches a vertical line, it reaches the beginning of a horizontal line, or it reaches the end of a horizontal line. If it is the first event the number of intersections with that line are counted. If it is the second event the number of intersections with that line are counted, then it is inserted into the active lines set. If it is the third event it is simply removed from the active lines set.

##### Complexity:

The outer for loop runs  $n$  times where  $n$  is the number of lines making it  $O(n)$ . Counting intersections is  $O(k)$  where  $k$  is the number of intersections. Both inserting and deleting a line from the active lines set is  $O(\log n)$ . This makes the overall complexity  $O(n) * (O(\log n) + O(k)) = O(n(\log n + k))$ . The space complexity is  $O(n)$  for PQ and  $O(n)$  for activeLines making it  $O(n) + O(n) = O(n)$ .

##### Pseudocode:

$PQ \leftarrow$  Priority queue of lines sorted by leftmost x-coordinate.

$intersections \leftarrow 0$

$activeLines \leftarrow$  set of lines sorted by y-coordinate

**procedure** SWEEP-LINE( $PQ$ )

**for each**  $line$  in  $PQ$  **do**..... $O(n)$

**if**  $line$  is vertical **then**

$intersections \leftarrow$  number of  $activeLines$  that intersect with  $line$ ..... $O(k)$

**else if**  $line$  is vertical and it's the first coordinate **then**

$intersections \leftarrow$  number of  $activeLines$  that intersect with  $line$ ..... $O(k)$

            insert  $line$  into  $activeLines$ ..... $O(\log n)$

**else**

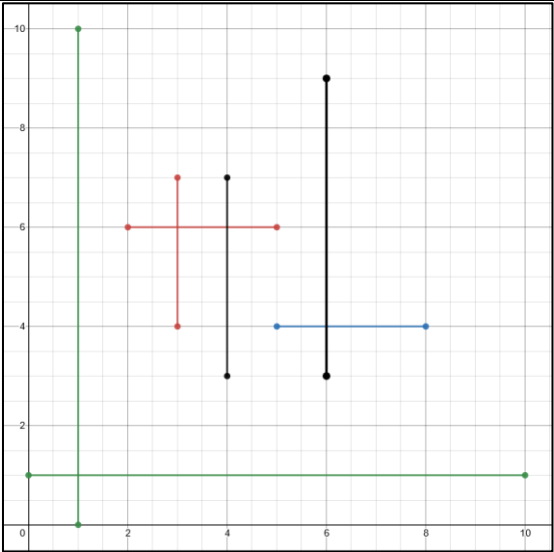
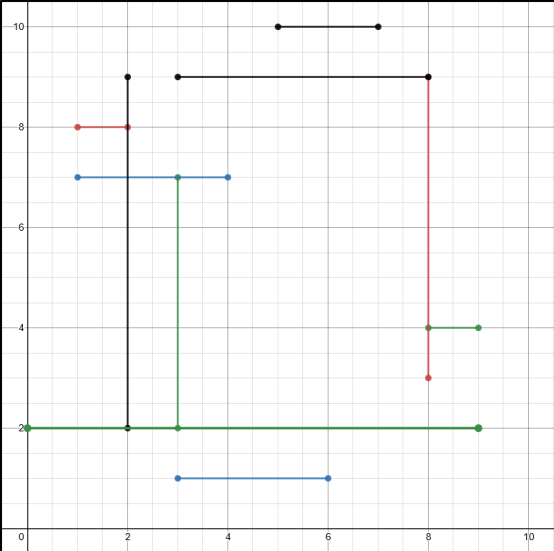
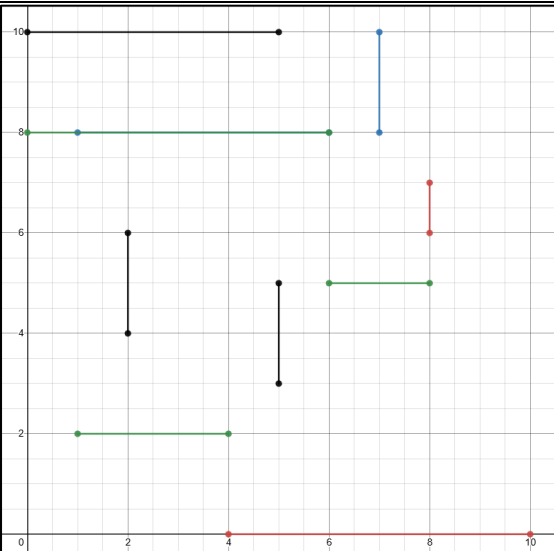
            remove line from  $activeLines$ ..... $O(\log n)$

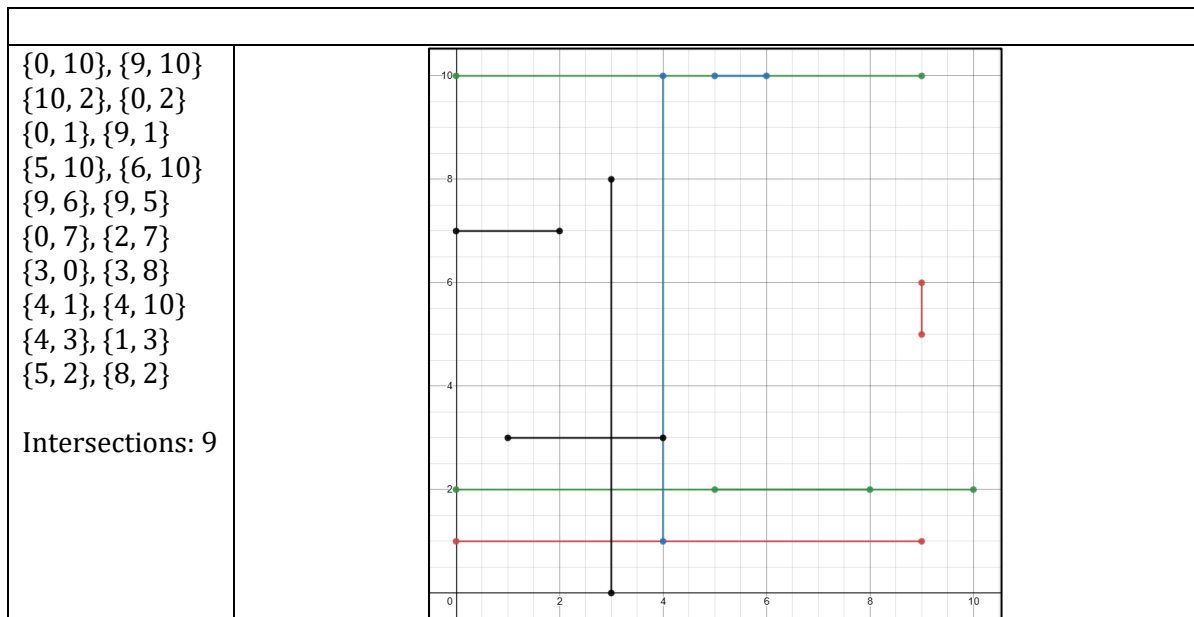
**end if**

**end for**

**end procedure**

# Results:

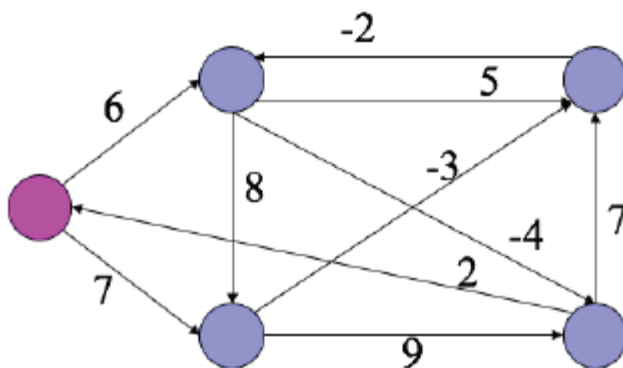
Lines	Visual
$\{0, 1\}, \{10, 1\}$ $\{1, 0\}, \{1, 10\}$ $\{3, 4\}, \{3, 7\}$ $\{5, 4\}, \{8, 4\}$ $\{2, 6\}, \{5, 6\}$ $\{4, 3\}, \{4, 7\}$ $\{6, 3\}, \{6, 9\}$  Intersections: 4	
$\{8, 4\}, \{9, 4\}$ $\{3, 2\}, \{3, 7\}$ $\{8, 9\}, \{8, 3\}$ $\{1, 7\}, \{4, 7\}$ $\{1, 8\}, \{2, 8\}$ $\{7, 10\}, \{5, 10\}$ $\{2, 2\}, \{2, 9\}$ $\{3, 1\}, \{6, 1\}$ $\{8, 9\}, \{3, 9\}$ $\{9, 2\}, \{0, 2\}$  Intersections: 7	
$\{1, 2\}, \{4, 2\}$ $\{8, 5\}, \{6, 5\}$ $\{10, 0\}, \{4, 0\}$ $\{1, 8\}, \{6, 8\}$ $\{8, 6\}, \{8, 7\}$ $\{5, 10\}, \{0, 10\}$ $\{2, 6\}, \{2, 4\}$ $\{7, 8\}, \{7, 10\}$ $\{5, 5\}, \{5, 3\}$ $\{6, 8\}, \{0, 8\}$  Intersections: 1	



## Question 2

### Problem:

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



### Solution:

The Bellman-Ford algorithm was implemented to find the shortest path from the pink node to all other nodes. This algorithm also searches for negative weighted paths and will return a warning if it finds one.

### Complexity:

The time complexity of Bellman-Ford is  $O(VE)$ . The first for loop is  $O(V)$  where  $V$  = number of vertices. The second for loop is  $O(V)$ . The nested for loop is  $O(E)$  where  $E$  = number of edges. The last for loop is  $O(E)$ . The overall complexity is  $O(V)+O(V)+O(E)+O(E) = O(VE)$ .

### Pseudocode:

```

G ← graph
V ← list of vertices in G
E ← list of edges in G

```

```

numV ← number of vertices in G
numE ← number of edges in G
s ← start vertex
d ← list of distances from each vertex to start vertex
w ← list of weights of each edge

procedure BELLMAN-FORD(G, V, E, s, d, w)
    for each vertex v in V do .....O(V)
        d[v] ← ∞
    end for
    d[s] ← 0
    for each vertex v in V do .....O(V)
        for each edge (u, v) in E do .....O(E)
            if d[v] > d[u] + w(u, v) then
                d[v] ← d[u] + w(u, v)
            end if
        end for
    end for
    for each edge (u, v) in E do .....O(E)
        if d[v] > d[u] + w(u, v) then
            Negative cycle found
        end if
    end for
end procedure

```

### Results:

```

Shortest path to node 1:
0 2 3 1
Shortest path to node 2:
0 2
Shortest path to node 3:
0 2 3
Shortest path to node 4:
0 2 3 1 4
Warning: Graph contains negative weighted path(s).

```

The program returned the shortest path for all node from the start node and threw a warning for node 4 that the shortest path found has a negative weight.

### Question 3

#### Problem:

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 ( $> 50,000,000$  vertices and  $> 100,000,000$ ) 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 can effectively perform the 2 operations listed above.

#### Solution:

Data structure: `vector<set<int>>> G`

To answer the question *is vertex  $v$  connected to vertex  $u$ ?* the following was used:

```
set<int>::const_iterator found1 = G.at(u).find(v);
set<int>::const_iterator found2 = G.at(v).find(u);
if (found1 == G.at(u).end() && found2 == G.at(v).end())
    v is not connected to u
else
    v is connected to u
```

To produce a list of all vertices connected to  $v$  the following was used:

```
for (set<int>::const_iterator it = G.at(u).begin(); it != G.at(u).end(); it++)
    cout << *it << endl;
```

#### Complexity:

The access element operation `vector.at()` is  $O(1)$  and the find operation of set is  $O(1)$  due to the built in hashing. This makes the overall operation  $O(1)$ . To iterate through a set is  $O(n)$  where  $n$  is the number of vertices connected to vertex  $v$ .

## Question 5

#### Problem:

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.

#### Solution:

The given data (`dictionary.txt`) combined with the given problem (ladder-gram) implies the use of a graph like structure. Every possible path in the graph from source to target will need to be produced. This is because each node (word) can have many adjacent nodes (words one character different). To ensure that the result produced is the minimum number of steps possible from the source word to the target word a brute force algorithm is required. This problem was solved using breadth-first search which finds all possible paths and the shortest one is returned.

#### Complexity:

Given the nature of the data and the problem the graph would not be very densely connected making the number of edges not close to the maximal number of edges ( $n*(n-1)/2$ ) making the algorithm time complexity determined by the number of vertices rather than the number of

edges. This implies an overall complexity of  $O(V)$  where  $V$  = number of words (vertices) in the graph. This would be the number of words of a given word length.

The while loop has a complexity of  $O(E)$  where  $E$  = number of edges. The for loop for checking if a word is in the dictionary is  $O(V)$ . The for loop for each character in a word is  $O(c)$  where  $c$  = some constant (word length). The for loop for each letter in the alphabet is  $O(c)$  where  $c$  = numbers of letter in the alphabet. Combining these gives the total time complexity of  $O(VE)$ . The space complexity would be  $O(V+E)$ . This is due to all edges  $E$  are stored in the queue  $Q$  and all vertices  $V$  are stored in the dictionary  $d$ .

#### Pseudocode:

```

s ← start string
t ← target string
d ← word dictionary containing words of the same size as s and t
visited ← list of visited words
Q ← queue of paths
currPath ← path of words from s to t
allPaths ← list of all paths found
depth ← 1
depthLimit ← ∞

```

**procedure** BFS( $Q, s, t, d, currPath, allPaths, depth, depthLimit$ )

```

while Q not empty do .....O(E)
    currPath ← Q.pop
    currWord ← currPath[-1]
    if currPath size > depth do
        for each word w in visited do .....O(V)
            erase w from d
        end for
        if size of currPath > depthLimit do
            break
        else
            depth ← size of currPath
        end if
    end if
    for each character c in currWord do .....O(C)
        nextWord ← currWord
        for each letter l in alphabet do .....O(C)
            nextWord[c] ← l
            if nextWord in d do
                tempPath ← currPath
                insert nextWord in tempPath
                insert nextWord in visited
                if nextWord = t do
                    insert tempPath in allPaths
                    depthLimit ← depth
                end if
            end if
        end for
    end for

```

```

        insert tempPath in Q
    end if
end if
end for
end for
end while
return allPaths
end procedure

```

#### Results:

Source	Target	Output	Steps	CPU
lead	gold	lead → load → goad → gold	3	0.002
hide	seek	hide → bide → bids → beds → bees → sees → seek	6	0.123
arcus	dolce	arcus → argus → argue → argle → angle → anile → anise → arise → prise → poise → hoise → house → douse → douce → dolce	14	0.037
number	redded	number → cumber → curber → curbed → curded → carded → warded → wadded → radded → redded	9	0.157

### Question 6

#### Problem:

The file *dictionary.txt* contains one word per line. Subsets of these words can be ordered such that, except for 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 algorithms and implement C++ programs that find such sequences separately for words of length 4 through to words of length 15 characters.

#### Solution:

The initial idea was to use breadth-first search to find the solution. However, BFS is extremely slow when dealing with large amounts of data such as *dictionary.txt*. To speed up the program the *dictionary.txt* was shortened to only contain words of length equal to the length chosen by the user. The dictionary was then sorted into “buckets” of every word that has the same second and third last characters i.e., bucket “ea” of word length 4 would contain words such as *bead*, *heat*, *lead*, *mean*, *team*, etc. These buckets were used to decide on the best possible starting word to maximise the length of the chain without having to search every possible word in the dictionary. This process was also used when deciding on the next word in the chain.

#### Complexity:

The outer while loop will run  $n$  times where  $n$  is the number of words of the selected length. The inner while loop runs until it finds a bucket with words in it which is represented as some constant. However, running the program shows that it runs a total of roughly  $n$  times. The erase and insert operations take  $\log n$  time. The *findNextWord()* function takes  $k$  time where  $k$  = number of words in a bucket. This gives an overall complexity of  $O(n) + O(n) * O(\log n) + O(k) = O(n \log n + k)$ .

**Pseudocode:**

```

wordLen ← user input from 4 to 15
sortedDict ← dictionary of words of length wordLen sorted in buckets by their second and third
last letters
currWord ← starting word (first word from the largest bucket in sortedDict)
currSeq ← sequence of words found
maxLenSeq ← largest sequence
p1 ← wordLen - 3
p2 ← wordLen - 2
endChars ← currWord[p1] + currWord[p2]
findNextWord(): finds word with most words in its bucket in sortedDict

```

```

procedure WORD-CHAIN(sortedDict, currWord, wordLen)

```

```

    while true do.....O(n)
        if endchars not in sortedDict do
            if size of currSeq > 1 do
                currWord ← currSeq[-1]
                erase currWord from currSeq.....O(logn)
                endChars ← currWord[p1] + currWord[p2]
            else
                break
        end if
        while size of sortedDict[endchars] = 0 do.....O(c)
            if size of currSeq > 1 do
                if size of currSeq > size of maxLenSeq do
                    maxLenSeq ← currSeq
                end if
                currWord ← currSeq[-1]
                erase currWord from currSeq.....O(logn)
                endChars ← currWord[p1] + currWord[p2]
            else
                break
            end if
        end while
        if endchars in sortedDict and size of sortedDict[endChars] > 0 do.....O(n)
            nextWord ← findNextWord().....O(k)
            insert nextWord in currSeq.....O(logn)
            currWord ← nextWord
        else
            break
        end if
    end while
    return maxLenSeq
end procedure

```



**Results:**

<b>Word Length</b>	<b>Num. Words</b>	<b>Seq. Length</b>	<b>CPU Found</b>	<b>CPU Total</b>	<b>Word Length</b>	<b>Num. Words</b>	<b>Seq. Length</b>	<b>CPU Found</b>	<b>CPU Total</b>
<b>4</b>	3,862	92	0	0.001	<b>10</b>	12,308	2,140	0.052	0.054
<b>5</b>	8,548	2,951	0.028	0.031	<b>11</b>	7,850	1,461	0.025	0.027
<b>6</b>	14,383	5,081	0.084	0.087	<b>12</b>	5,194	862	0.016	0.014
<b>7</b>	21,729	7,784	0.203	0.209	<b>13</b>	3,275	466	0.005	0.006
<b>8</b>	26,448	9,477	0.291	0.296	<b>14</b>	1,775	215	0.001	0.002
<b>9</b>	18,844	4,779	0.128	0.132	<b>15</b>	954	86	0	0.001

The results produced are drastically faster than the proposed results. They do not produce as long of a sequence for all words lengths except for 4.