Geometric Processing Coursework 1

Yu Wang (201292704)

1 Introduction.

This paper analyses the time complexity of the face2faceindex.cpp and faceindex2directededge.cpp respectively. Then provides the entire time complexity of the coursework.

2 *face2faceindex* time complexity.

2.1 In function *ReadFileTriangleSoup*.

This function is used for extracting unique vertices from the .tri file, and recording non-unique vertex IDs. Figure 1 shows the code , a double loop , affecting the complexity most. Clearly, we can see that the worst situation is we need to traverse the whole array *vertices* with n elements. And considering the outer loop, we have to loop n times. Hence this function has the $O(n^2)$ time complexity.

```
for(int vertex=0; vertex<nVertices; vertex++){
    // traverse all of the vertices and calculate the unique ID of them.
    inFile >> currentVertex.x >> currentVertex.y >> currentVertex.z; // read the coordinate in
    existIndex = notFound; // reset the no found value when the new loop starts
    for (int i = 0; i < vertices.size(); ++i) {
        //from the already existing vertices, look for if the coordinate is in the vector.
        //if so, record the index i to the var existIndex
        if (currentVertex == vertices[i]){
            existIndex = i;
            break;
        }
    }
```

Figure 1, the double loop in function *ReadFileTriangleSoup*.

2.2 Other functions.

In face2faceindex.cpp, except for ReadFileTriangleSoup function, these functions *writeVertices* and writeFaces only have one layer loop. Therefore, these functions have O(n) time complexity. The remaining functions can not affect the time complexity of this class.

3 *faceindex2directededge* time complexity.

3.1 In function *ReadFaceIndexFile*.

Function ReadFaceIndexFile shoulders reading the .face file. It has two one layer loop, and one double layer loop(shown in figure 2). The inner loop, marked in figure 2, has a constant level loop which is three. The outer do-while loop has O(n) time complexity. Hence, we can tell that the complexity of this dual loop in figure 2 is O(3n), which is

still O(n). The other two loops have one layer loop. So they have O(n) complexity as well. Consequently, this function has O(n) complexity.

```
do {//in this loop, we are reading the face section
    this->facesInfo.append(currentLine).append( s: "\n"); // the first line of face info has been r
    stringstream lineStream(currentLine);// create a stream to read tokens of a line
    lineStream >> currentToken; //read "Face"
    if ("Face" != currentToken){
        //here, we have to confirm the first token is "Face", if not the data structure is wrong
        break;
    }
    lineStream >> currentToken; //read face index, but we do not need it
    for (int i = 0; i < 3; ++i) {
        //then the next three tokens is the vertices IDs of a face, so read three times
        long currentVerId = 0;
        lineStream >> currentVerId;
        this->vertexIDs.push_back(currentVerId);
    }
    if (inFile.eof()){
        //here, if we have reach the end of the file, then break the loop, it means we have alread
        break;
    }

} while (getline( &: inFile,  &: currentLine)); //get next line
```

Figure 2, the double loop in function *ReadFaceIndexFile*.

3.2 the function *lookForPinchPoint*.

In this function, one loop (in figure 3) is for calculating the degree of every vertex, which is easy to tell its complexity O(n).

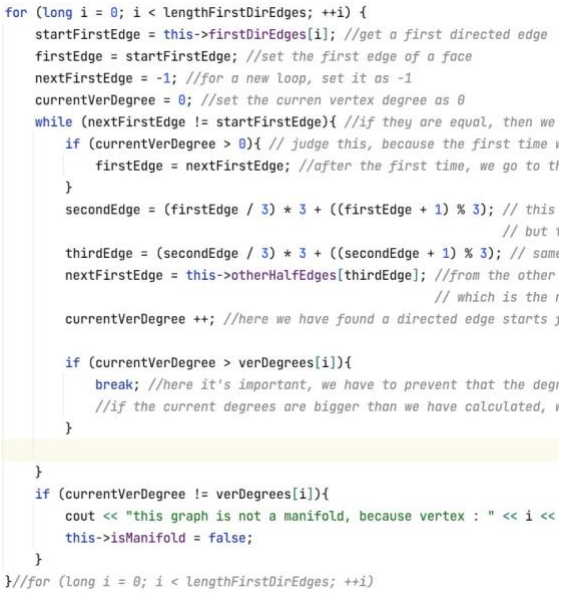
```
for (int i = 0; i < lengthVerIDs; ++i) {
    int currentVer = this->vertexIDs[i]; //find ev
    verDegrees[currentVer] ++; //calculate how man
                               //we just care how
                               //it means an edge
}
```

Figure 3, the loop for calculating degrees in function *lookForPinchPoint*.

Another double layer loop (shown in figure 4) is for calculating the degree of every vertex by counting the directed edges starting for the vertex. If the degree is not equal as the loop in figure 3 calculates, we can tell that there is a pinch point.

The two layer loop in figure 4, as the inner while loop is also can reach n times, considering the worst condition. So we can tell that the two layer loop has $O(n^2)$ complexity.

In sum, the function *lookForPinchPoint* has $O(n^2)$ complexity.

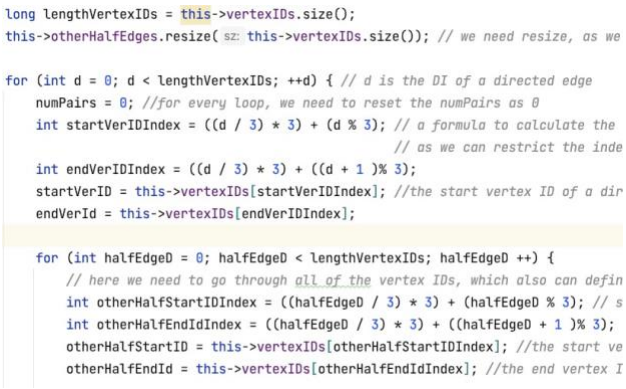


```
for (long i = 0; i < lengthFirstDirEdges; ++i) {
    startFirstEdge = this->firstDirEdges[i]; //get a first directed edge
    firstEdge = startFirstEdge; //set the first edge of a face
    nextFirstEdge = -1; //for a new loop, set it as -1
    currentVerDegree = 0; //set the curren vertex degree as 0
    while (nextFirstEdge != startFirstEdge){ //if they are equal, then we
        if (currentVerDegree > 0){ // judge this, because the first time
            firstEdge = nextFirstEdge; //after the first time, we go to th
        }
        secondEdge = (firstEdge / 3) * 3 + ((firstEdge + 1) % 3); // this
                                                                  // but
        thirdEdge = (secondEdge / 3) * 3 + ((secondEdge + 1) % 3); // same
        nextFirstEdge = this->otherHalfEdges[thirdEdge]; //from the other
                                                         // which is the
        currentVerDegree ++; //here we have found a directed edge starts
        if (currentVerDegree > verDegrees[i]){
            break; //here it's important, we have to prevent that the degr
            //if the current degrees are bigger than we have calculated,
        }
    }
    if (currentVerDegree != verDegrees[i]){
        cout << "this graph is not a manifold, because vertex : " << i <<
        this->isManifold = false;
    }
}//for (long i = 0; i < lengthFirstDirEdges; ++i)
```

Figure 4, two layer loop in function *lookForPinchPoint*.

3.3 the fucntion generateOtherHalfEdges.

In this function, as the double layer loop (in figure 5) has to traverse all of the vertex IDs, the complexity is $O(n^2)$.



```
long lengthVertexIDs = this->vertexIDs.size();
this->otherHalfEdges.resize( sz: this->vertexIDs.size()); // we need resize, as we

for (int d = 0; d < lengthVertexIDs; ++d) { // d is the DI of a directed edge
    numPairs = 0; //for every loop, we need to reset the numPairs as 0
    int startVerIDIndex = ((d / 3) * 3) + (d % 3); // a formula to calculate the
                                                   // as we can restrict the inde
    int endVerIDIndex = ((d / 3) * 3) + ((d + 1 )% 3);
    startVerID = this->vertexIDs[startVerIDIndex]; //the start vertex ID of a dir
    endVerId = this->vertexIDs[endVerIDIndex];

    for (int halfEdgeD = 0; halfEdgeD < lengthVertexIDs; halfEdgeD ++) {
        // here we need to go through all of the vertex IDs, which also can defin
        int otherHalfStartIDIndex = ((halfEdgeD / 3) * 3) + (halfEdgeD % 3); // s
        int otherHalfEndIdIndex = ((halfEdgeD / 3) * 3) + ((halfEdgeD + 1 )% 3);
        otherHalfStartID = this->vertexIDs[otherHalfStartIDIndex]; //the start ve
        otherHalfEndId = this->vertexIDs[otherHalfEndIdIndex]; //the end vertex I
```

Figure 5, the double layer loop in function *generateOtherHalfEdges*.

3.4 the remaining fucntions.

The remaining functions, such as writeOtherHalfEdges, generateFirstDirectedEdges writeFirstDirectedEdges and writeVertices, only have one layer loop. Hence, their time complexities are O(n).

4 Consulsion

In sum, the maximum order of magnitude of time complexity in both *faceindex2directededge* and *face2faceindex* is $O(n^2)$. Therefore, the complexity of my program is $O(n^2)$.