Geometric Processing Coursework 1

Jonathan Alderson

University of Leeds

October 21st 2020

Intro

The time complexity will be analyzed for coursework 1. Split up into the face to face index section and the face index to directed edge section. With each algorithm being looked at individually.

1 Face to Face Index

1.1 Calculate Unique Faces

This is done as we are reading in the file. For each vertex, loop through array of unique vertices, if not present, add to list.

```
for (int vertex = 0; vertex < nVertices; vertex++)
{
    // reading code
    // ...
    // check if we have already seen this vertex
    for(unsigned int i = 0; i < vertices.size(); i++)
    {
      if(thisVertex == vertices[i]){ foundIndex = i; break; }
    }
    // ...
    // more reading code
}
```

Here we have an exterior loop of n and an interior loop of up to n. Giving a time complexity of $O\ \mathrm{n}^2$. This is acceptable, as vertices will start small, and grow slowly as we only add unique vertices, so is better than a worst case $O\ \mathrm{n}^2$.

1.2 Calculate Midpoint

```
for (int vertex = 0; vertex < nVertices; vertex++)
{
    // reading code
    // ...
     midPoint = midPoint + thisVertex;
    // ...
    // more reading code
}
```

Midpoint is also calculated as the file is initially read. Although there are 6 if statements checking the x, y, z components, this is all $O\ 1$. Giving the whole operation $O\ n$, this is very reasonable. And could only be improved by only taking the midpoint of the unique vertices rather than all, to save some time.

1.3 Saving File

The save file section of Face to Face Index has been created so that functions can be reused by inherited classes, so has been separated out into smaller functions. Saving the header and changing the file name happen in constant time, although the file changing involves two loops for string splitting:

```cpp
// Tokenize first by /
while(getline(splitter, intermediate, '/'))
{
    tokens.push_back(intermediate);
}
// then tokenise again by .
std::stringstream splitter2(tokens[2]);
tokens.clear();
while(getline(splitter2, intermediate, '.'))
{
    tokens.push_back(intermediate);
}
```

The file names will always be a short length and are determined by the file system, this will only be 3, 4 or 5. So can be counted as constant time
$O\ 1$.
Saving vertices and saving faces both involve a loop of n. This happens twice, these loops could be adjoined to only loop once instead of twice, but this has been avoided to improve readability.
This gives the saving of files a time complexity of $O\ n$.

2 Face Index to Directed Edge

2.1 Reading File

When reading the file, we again calculate the midpoint, and add to our vertices and vertexID arrays. All operations and vector push backs happen in $O\ 1$, which happens for every vertex we read, giving $O\ n$ time. Unfortunately, there are three passes, for reading, calculating the midpoint, and applying the changes. This is since we only read the unique vertices but are using all to calculate the midpoint. This could be made more efficient by only using the unique vertices to calculate the midpoint.

2.2 Calculate First Directed Edge

This is straightforward, we loop through all our edges once, then perform some constant time operations, giving the final time complexity of $O\ n$.

2.3 Calculate Other Half

This is one of the longest algorithms, we must step through each edge, calculate the start and end point, then once more loop over each edge, checking if we have any other edge matches. This cannot be sped up with break statements when the matching pair is found, as some non-manifold meshes may have more than two shared edges, which would not be discovered if break statements were implemented. This section could be improved with data structures and lookup tables to improve time complexity.

```
// vertexA vertexB
for(unsigned long vA = 0; vA < vertexID.size(); vA++)
{
  // calculate the start and end point
  matches = 0;
  for(unsigned long vB = 0; vB < vertexID.size(); vB++)
  {
    // if this has the right end and start
       matches++;
  }

  // check for errors
  if(matches < 1 || matches > 1)
  {
    // printout error
    manifold = 0;
  }
}
```

Since we have two nested for loops of length n. We have a time complexity of $O\ \mathrm{n}^2$.

2.4 Check Pinch Point

To check for pinch points, we first need to calculate the order of each vertex. This for loop has been taken outside so all the orders are calculated once, saving time, instead of looping through looking for the individual vertex for each vertex, saving some time complexity. We then loop through each vertex and loop around its one ring. It could be argued that the one ring will be such a small number compared to n that it could be counted as constant time. But for worst case, we can loop for up to n times, even with a break statement if we go beyond the order for that vertex. So, we have one loop of $O\ n$, followed by a for loop and a nested while loop of the one ring, giving the total time complexity of $O\ \mathrm{n}^2$.

2.5 Calculate Genus

This is the simplest algorithm; we have already calculated the necessary components and can rearrange the equation beforehand. Giving the time complexity of $O\ 1$.

```cpp
void FaceIndex2DirectedEdge::calculateGenus()
{
  long v = vertices.size();
  long e = vertexID.size() / 2;
  long f = vertexID.size() / 3;
  long g = (e + 2 - f - v) / 2;
  std::cout << "Genus: " << g << '\n';
}
```

2.6 Save File

Saving file is done once everything has already been calculated, it just involves a series of for loops all of length n. As none of these are nested and just sequential, we get the final time complexity of $O\ n$.

3 Conclusion

Overall, the time complexity for these algorithms is good enough for the coursework. It can be noted that a lot of the algorithms do not need to be run in any specific order and are reusing the same for loops, for example, looping over all the vertices in the mesh. To improve efficiency, we could group all or for loops together, this would make the code slightly messier so was avoided for this coursework. I also imaging there are more efficient ways of doing the other half algorithm as we have an $\mathrm{O}(\mathrm{n}^2)$ time complexity.