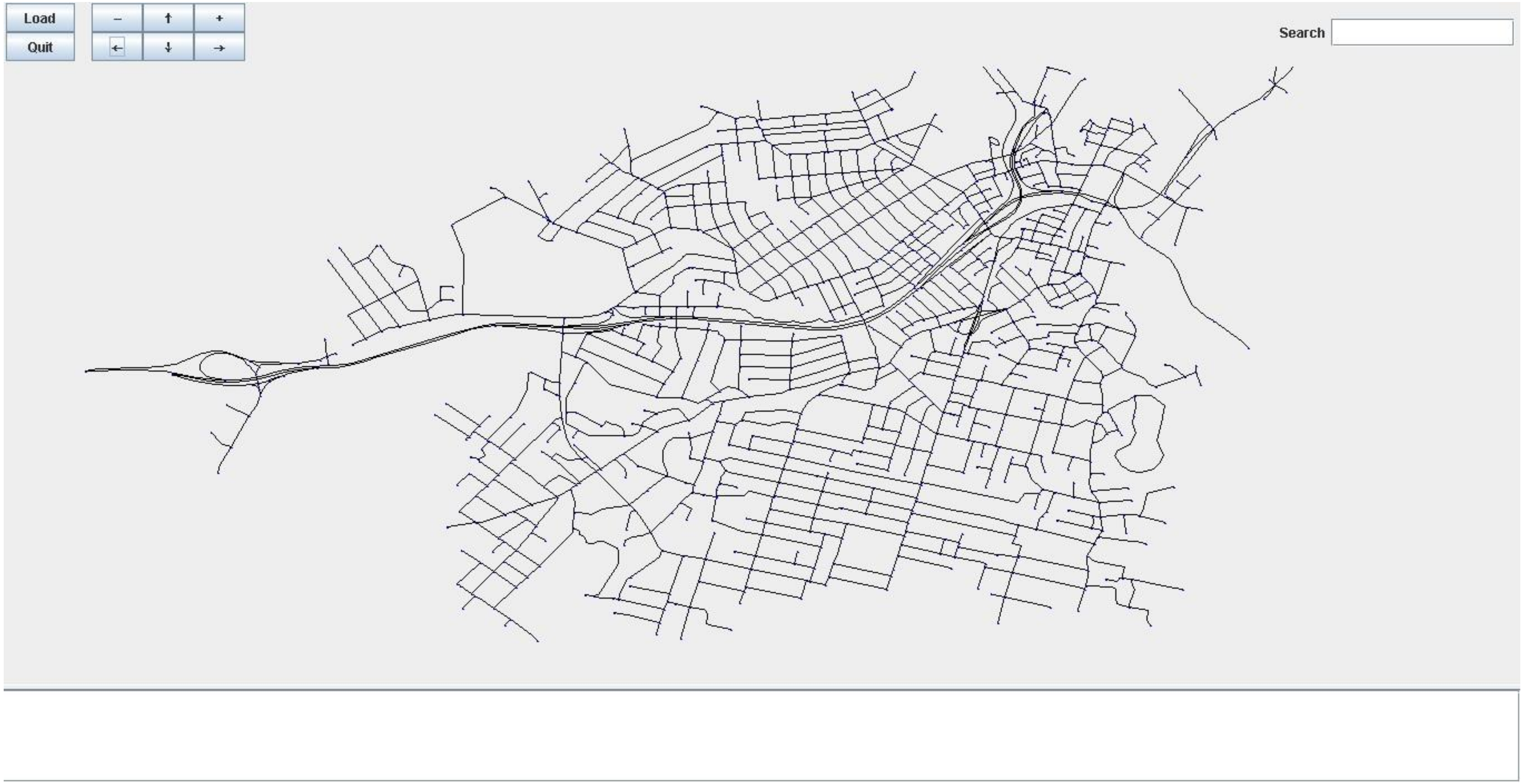
COMP261 Tutorial 1

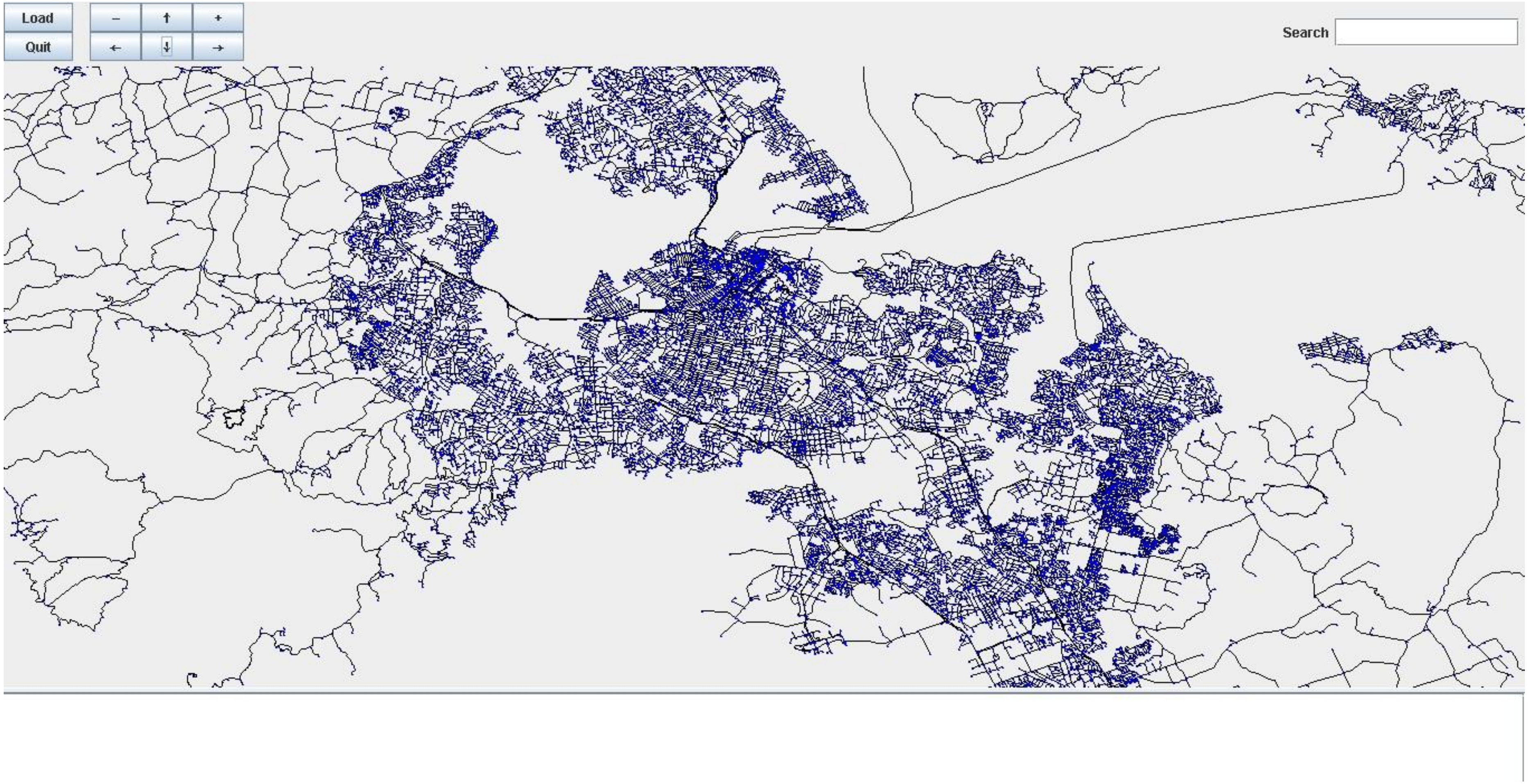
2017, Trimester 1

Aaron Craig

The Assignment

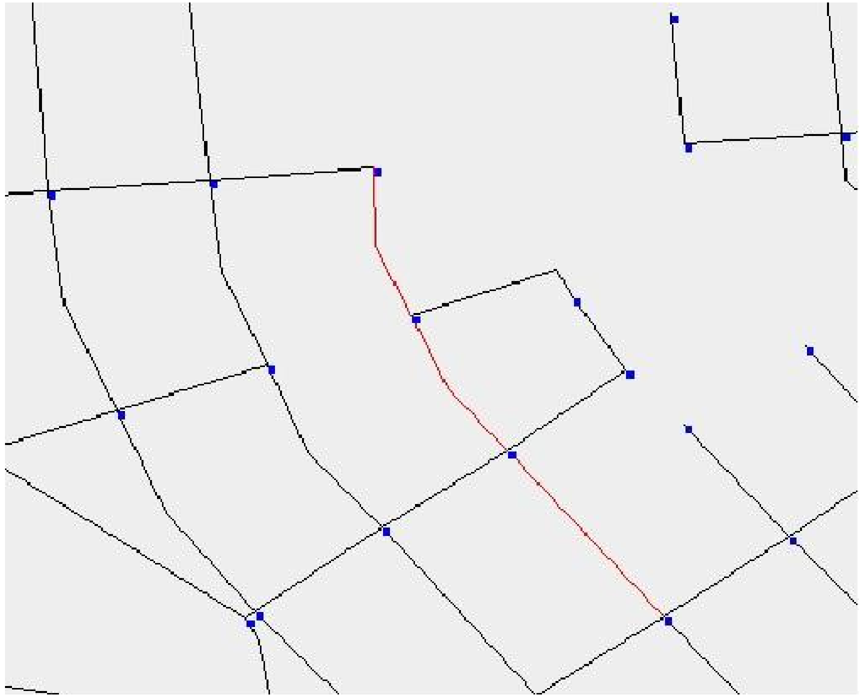
- GoogleMaps for Auckland
  - Load data from files into a graph data structure
  - Draw the graph data structure in a GUI
  - Look up roads by name, click on intersections
- Code for the GUI is given to you
  - Need to subclass `GUI` and implement the functionality
  - Check out the squares example in the assignment handout
  - Read `GUI.java`
- Two data sets (small and large)


Load
Quit
–
↑
+
←
↓
→
Search


Load
Quit
–
↑
+
←
↓
→
Search

Roads as Graphs

- Nodes = intersections
- Edges = segments of road
- Roads impose restrictions on the speed-limit, whether its segments are one-way etc.
- Schofield street in red
  - 3 segments
  - Segments themselves composed of several lines



Representing Points

- Latitude/Longitude
    - Represents a position on a sphere (earth)
    - Coordinates in files are stored as latitude/longitude
- Location
    - Represent (x,y) coordinates on a flat plane with (0,0) = Auckland centre
    - Stored as a pair of doubles
- `java.awt.Point`
    - Represents an (x,y) coordinate on computer screen (y-axis inverted)
    - Stored as a pair of doubles, rounded to the nearest integer
- `Location.java` has methods for converting between representations

Loading From Files

- Look at the data files with a text editor
    - Each line describes one intersection/segment/road
- Focus on getting Minimum working first, ignore unnecessary parts of file
    - E.g. extra columns in the roads file
- Some of the data may be incorrect or malformed
    - E.g. segments that refer to intersection IDs that don't exist
- Not always clear what is the right way to handle this
    - Choose a sensible solution, justify it

Scanner vs. BufferedReader

- `new Scanner(file);`
    - Tokenises input data, presents it as a sequence of tokens
    - Will be slow on the big data set
- `new BufferedReader(new FileReader(file));`
    - Does not tokenise input data
    - Need to manually implement tokenising logic
        - Loop over each line in the file and tokenise/parse it
        - `String.split` may help you

Initialising the Graph

- `Intersection` and `Segment` classes depend on each other
- Need to separately instantiate segments and intersections, then pass them to each other via method call e.g. an `addSegment` method
    - 1. Load data from intersections file and instantiate intersections
    - 2. Load data from segments file and instantiate segments
    - 3. Add to each intersection its neighbours (and the segment leading to that neighbour)
- Need intersection IDs to do step 3, probably don’t need them afterwards
- Can do step 3 as you’re doing step 2

Asymptotic Analysis

- Measures how time or space of an algorithm scales with input size/s
  - Time: how the number of operations scales with input size/s
  - Space: how the amount of memory needed scales with input size/s
- Constant factors not relevant to asymptotic complexity
  - 1000000000000000000000000000000000000*n = O(n)
  - $10*n^2 = O(n^2)$
- Unless we say otherwise, O(....) refers to the worst-case
  - Correct, formal notation that you might see on Wikipedia: big-oh O, big-omega Ω, big-theta Θ

Graph Asymptotics

- Asymptotic complexity of a graph data structure (or algorithm):
  - Can be measured as a function of number of nodes (N) and edges (E)
  - In a simple graph, E is at most N * (N-1), so worst case E = $O(N^2)$
- Two main representations
  - Adjacency matrix: a 2d array where entries represent edges
  - Collections of objects that store neighbours in a list or map
- Probably use the second for the assignment

Objects w/ Adjacency Lists

- Store a collection of nodes = O(N)
- Let M = number of neighbours. Worst case: M = O(N)
- Each node stores its neighbours in a list/map/set
    - Best case: O(M) space. Worst case: O(N) space
    - Finding neighbours is best case O(M) time, worst case O(N) time
    - Better in sparse graphs (small number of edges)
- Finding neighbour from an edge can be fast
    - O(1) time if storing neighbours as a map from edges to neighbours

Adjacency Matrix

- Store edges in an N x N matrix (2d array)
- Always requires O(N^2) memory (best and worst case)
  - A lot of wasted space if small number of edges in graph
- Checking if an edge exists from node i to node j is O(1)
  - Just look up `matrix[i][j]`
- Looking up neighbours of node i is O(N)
  - Need to look down all of `matrix[i]`, which is length N
- Cache locality improves constant factor
  - Data structure in one contiguous part of memory
  - Doesn’t affect asymptotic behaviour, but reduces the constant factor