Assignment 2:
A* and informed heuristic searches on maps for wheelchair accessibility

Submission Deadline: [6 May 2023]

Cameron Malone
218344989

Task Objective

The objective of this task was to investigate informed or *heuristic* search strategies on a search space modelled after a real-life road network with the specific caveat that the agent be represented as someone inhibited by a wheelchair in their ability to traverse said network. The task is aimed at exploring various forms of heuristics and how their properties can affect the implementation of the common A* heuristic search algorithm.

As an extension, we were tasked with researching alternative methods for network-style pathfinding solutions; in particular, those which find common deployment in real-world scenarios for their marked improvement in runtime query costs for end users. To this end, I decided to implement and gauge the performance of the *Contraction Hierarchies* algorithm.

1. Abstract

1.1 Informed / Heuristic Search

In the context of search algorithms, *informed* refers to the ability for the problem solving agent to derive some notion of its *distance* from the goal. This distance measure is at the core of every informed search algorithm and is given by a *heuristic function* $h(n)$

Heuristic functions come in a large variety of forms, the most recognisable of which may be the Manhattan (city-block) and Euclidean distances, known as the L1 and L2 norms respectively which describe the distance between points . [1]

1.2 A (Greedy-Best-First search)*

A star is an extremely simple algorithm that builds on the foundations of Greedy-Best-First search: Given knowledge on the location of the goal state and a means by which to measure distance to it, we should at every stage seek to expand nodes which minimise this distance.

For A*, the *evaluation function* $f(n)$, by which we assign priority to the expansion of some node n is given by $f(n) = g(n) + h(n)$. Where $g(n)$ is the cumulative cost of actions taken to reach the current state from the initial state, and $h(n)$ is the distance to the goal state, provided by some heuristic function. [1]

1.3 Contraction-Hierarchies

In essence, Contraction Hierarchies is a method for quantifying some measure of a node's *importance* to traversal and then *ordering* the nodes in a network based on that metric. In theory, if an optimal ordering of nodes can be found, then, despite the somewhat overwhelming cost of pre-processing, subsequent queries to the graph can leverage the order to drastically reduce the average search-space required to find optimal-cost routes between points.

In his excellent project paper on the subject, John Lazarsfeld [2] equates the procedure as being akin to zooming out on Google maps. As you decrease the fidelity of the picture, smaller, less relevant streets begin to fade out of view in favour of large junctions, highways, and interstate roads. These larger streets often form a large portion of the driving during an average trip and including them is usually more favourable than travelling the same distance over many much smaller roads (less capacity, lower speed limit etc.).

In practice, categorising the importance of nodes in a network and creating Google's 'Big yellow roads' comes down artificially reducing the search space of a network by adding additional edges between nodes in a process known as *contraction.*

1.4 Node Contraction

Given a graph $G = (V, E)$ and a vertex $v \in V$, we can *contract v by:*

1. Denoting the sets U, the set of vertices adjacent to v which have outgoing edges to v, and W, the set of vertices which v has outgoing edges to.
2. For all u in U do:
   a. Find the maximum cost of a path $u \to v \to w$ for all w in W.
   b. Run a Dijkstra's search on the localised subgraph starting a u, which finds the cost of paths to all w in W excluding v, which terminates upon reaching a node with a path score higher than the maximum found earlier.
3. If a path $u \to w$ excluding v exists such that the cost $d(u, w)$ is less than the cost of travelling $u \to v \to w$, then we can add a shortcut edge $u \to w$ with weight $d(u, w)$. (i.e. there is a better option than including v.)
4. Remove v and all its incident edges from the graph.

1.5 Querying

This process is repeated until all nodes in the graph have been contracted and removed. The leftover product is then a set of shortcuts known as an overlay graph, which can be used in the *search process* (modified bi-directional Dijkstra's) for more efficient querying.

The *order* in which the nodes were contracted dictates their relative *importance* and is calculated by *simulating* contraction without directly modifying the graph, then noting the number of shortcuts that would have been added (edge difference). Generally, we aim to prioritise and thus contract early in the process nodes which have a lower edge difference.

Complete node contraction on the graph concludes the pre-processing segment of CH. Querying the resulting graph (union of the original graph G and overlay graph G' containing the shortcuts) is then a matter of running two *complete* Dijkstra's on following modified sub-graphs of G': [2]

- G'_D: Downward graph from vertex v containing only nodes and edges towards nodes where v contracted earlier.
- G'_U: Upward graph from vertex v containing only nodes and edges towards nodes which were contracted before v

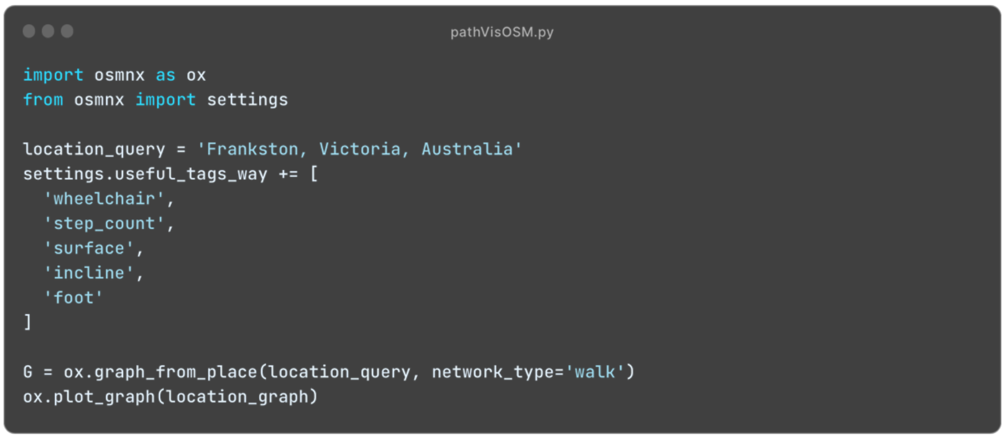
Note: unique ordering of nodes means that $G'_D \cup G'_U$ contains the whole search space G' [2]

Note: For a fully bi-directional / symmetric G' starting at vertex v, $G'_D = G'_U$

Once the complete searches are finished, the solution is found by taking the minimum sum of path costs across every node in the intersection of nodes reached in both searches. This node can then be used to back-trace paths in the respective search sub-graphs using parent-edge and parent-node pointers in the node objects.

2. Approach, Challenges and Discussion

My initial approach to modelling my local suburb as a network graph used OSM (Open Street Maps) data downloaded and translated to a graph through the python NetworkX interface layer OSMNX. Wheelchair accessibility da2ta is well-supported by the OSM architecture. With custom filters applied when fetching the map, way data (ways being the OSM equivalent to edges) can be configured to include any of a wide list of 'tags' (fields) (**Figure 1**).



```
import osmnx as ox
from osmnx import settings

location_query = 'Frankston, Victoria, Australia'
settings.useful_tags_way += [
  'wheelchair',
  'step_count',
  'surface',
  'incline',
  'foot'
]

G = ox.graph_from_place(location_query, network_type='walk')
ox.plot_graph(location_graph)
```

Figure. 1 OSMNX data import and graph creation with custom way filters applied according to OSM wheelchair accessibility guidelines: https://wiki.openstreetmap.org/wiki/Wheelchair_routing

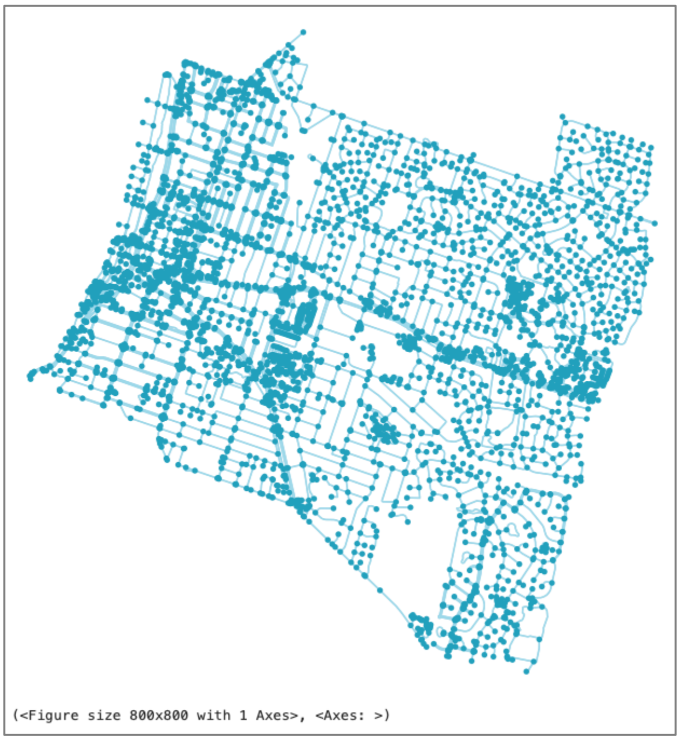
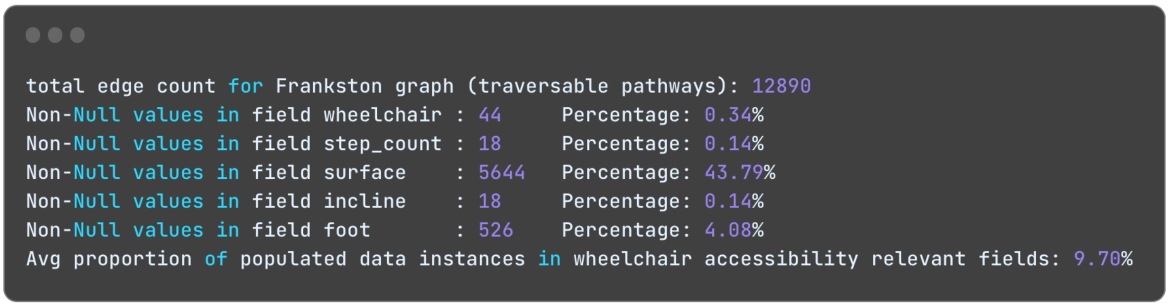


Figure 2. OSMNX network graph output for query "Frankston, Victoria" shown in Fig. 1

While visually attractive and initially very promising, the implementation encountered some impassable bottlenecks when it came time to perform edge queries and reconstruct wheelchair accessible paths. This becomes immediately clear when we take a closer look at the data.



```
total edge count for Frankston graph (traversable pathways): 12890
Non-Null values in field wheelchair : 44     Percentage: 0.34%
Non-Null values in field step_count : 18     Percentage: 0.14%
Non-Null values in field surface    : 5644   Percentage: 43.79%
Non-Null values in field incline    : 18     Percentage: 0.14%
Non-Null values in field foot       : 526    Percentage: 4.08%
Avg proportion of populated data instances in wheelchair accessibility relevant fields: 9.70%
```

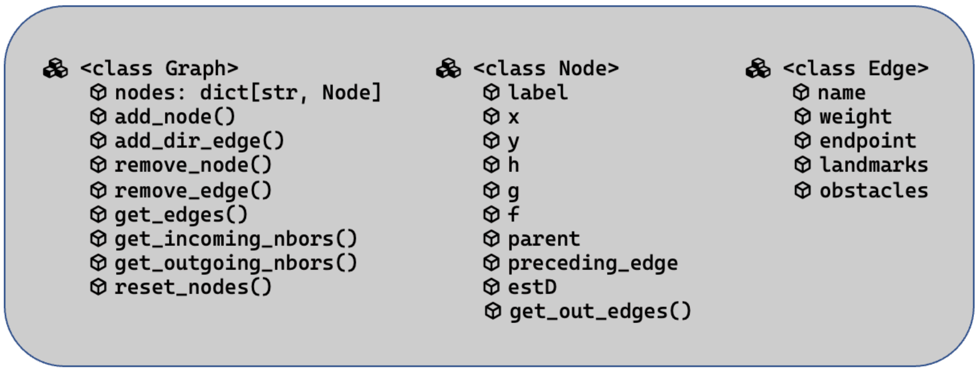
Figure 3. Summary statistics on edge Data Frame output by OSMNX

At 12,890 ways, the resulting graph is extremely dense, and yet simultaneously devoid of any meaningful data. Without considering road surface, which is likely the least impactful metric requested in the filters, the mean percentage of populated records was 1.67%.

I was unable to resolve this issue; the sparsity of the data made it difficult to produce meaningful pathways, while the density of the overall graph made manual edge property insertion infeasible.

2.1 Implementation: Custom Graph Class

After careful consideration, I decided to implement the graph, node, and edge data structures from scratch, ensuring consistency in style and behaviour.



```
<class Graph>                    <class Node>             <class Edge>
    nodes: dict[str, Node]           label                    name
    add_node()                       x                        weight
    add_dir_edge()                   y                        endpoint
    remove_node()                    h                        landmarks
    remove_edge()                    g                        obstacles
    get_edges()                      f
    get_incoming_nbors()             parent
    get_outgoing_nbors()             preceding_edge
    reset_nodes()                    estD
                                     get_out_edges()
```

Figure 4. Custom python graph, node, and edge classes attribute and method overview

The inspiration for the design of the classes was given by inspection of the source code behind the natively supported OSMNX graph and node implementations. The lightweight nature of these custom classes made modification and extension during algorithm tuning a simple task.

Between the attributes and methods shown above, all operations and features necessary to complete both A* and Contraction Hierarchies are available.

2.2 A Heuristic Search:*

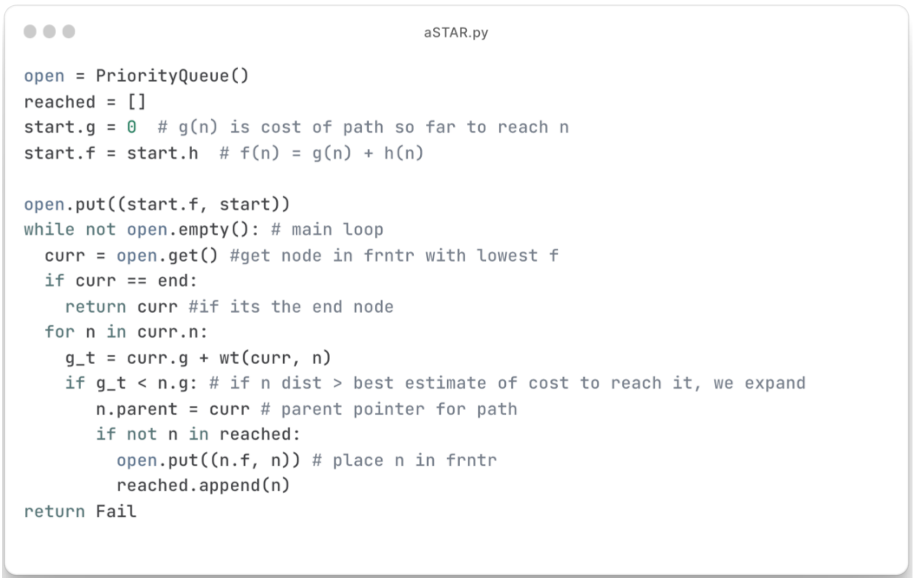
After encountering some difficulty in constructing a working and bug-free graph data-structure, particularly in handling the deletion of nodes and edges in the graph class, the A* algorithm itself was surprisingly easy to implement.

Using [1]'s excellent breakdown of Greedy-Best-First search [ch. 3.5.1, pp. 103-107] as a foundation, A* can be defined as a derived GBFS with an evaluation function:

$$f(n) = g(n) + h(n).$$

2.3 Putting it together:

Python's built-in min-heap Priority Queue allows for simple prioritisation of appending to the frontier those nodes which minimise $f(n)$. With that, the complete algorithm can be (only slightly) simplified and expressed as:


aSTAR.py

```python
open = PriorityQueue()
reached = []
start.g = 0  # g(n) is cost of path so far to reach n
start.f = start.h  # f(n) = g(n) + h(n)

open.put((start.f, start))
while not open.empty(): # main loop
  curr = open.get() #get node in frntr with lowest f
  if curr == end:
    return curr #if its the end node
  for n in curr.n:
    g_t = curr.g + wt(curr, n)
    if g_t < n.g: # if n dist > best estimate of cost to reach it, we expand
       n.parent = curr # parent pointer for path
       if not n in reached:
         open.put((n.f, n)) # place n in frntr
         reached.append(n)
return Fail
```

2.4 Heuristics: How is A affected by choice of heuristic?*

Russell and Norvig [1] explain in their chapter on informed search algorithms that A* is complete[1], but only conditionally optimal. Complete-ness of the algorithm is evident seeing the above code – given initialisation of $g_{initial}(n) = \infty, \forall n \in V$, all nodes in a graph connected from the source node will pass the $g_{temp} < g(n)$ test at least once and be appended to the frontier. Cost-optimality is determined by the heuristic, particularly its *admissibility*,

"an admissible heuristic is one that *never* over-estimates the cost to reach the goal" [1]

[1] *given existing solution or finite search space, and $\epsilon > 0$ (ϵ is lower bound for action cost [1]*

For the purposes of this assignment, I evaluated the performance of A* using three different heuristics:

- *Manhattan Distance (L1 norm)*: $\Sigma_1^n |p_i - q_i|$
- *Euclidean Distance (L2 norm):* $\sqrt{\sum_{i=1}^{n}(q_i - p_i)^2}$
- *Haversine Distance[2](Great Circle Distance):* $hav(\theta) = \sin^2(\frac{\theta}{2})$

[2] *Great circle distance gives the shortest distance between two points on a sphere, gives <0.5% error on latitude and <0.2% error on longitude [4]*

2.5 Graph representation of local suburb:

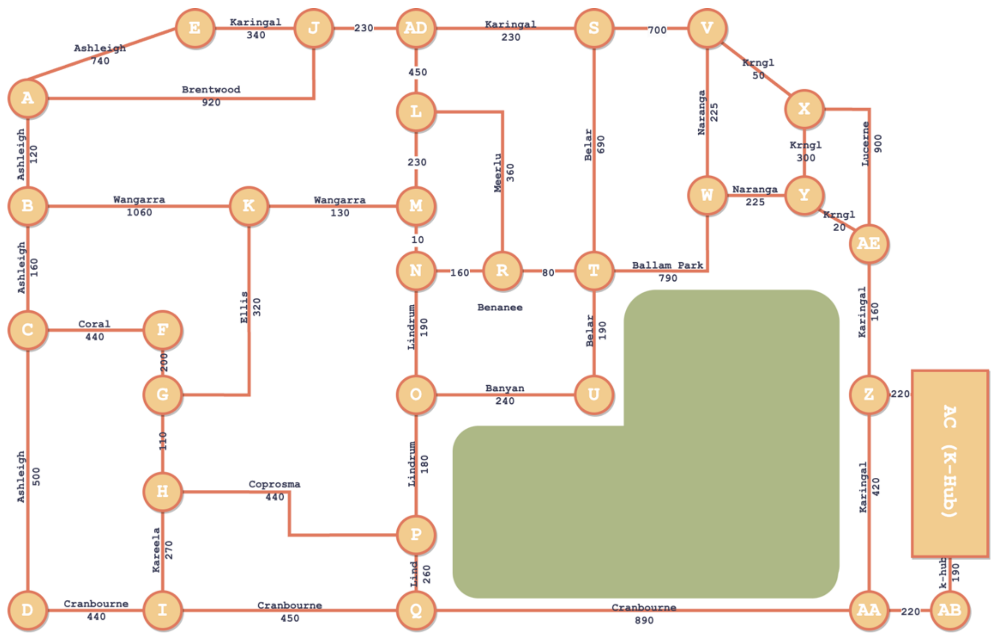
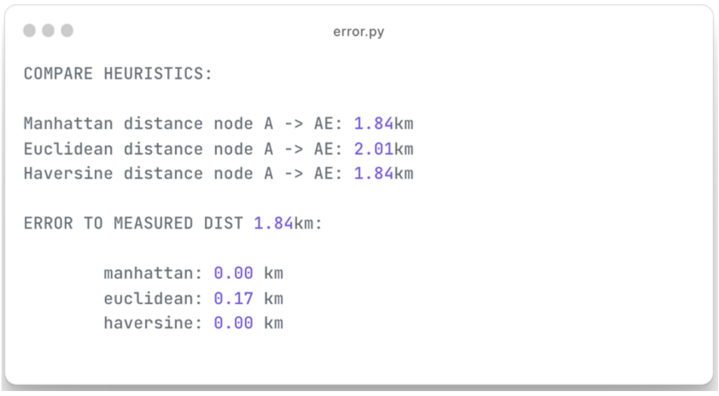


Figure 5. *Abstract representation of the graph data structure used for A* search algorithm. Latitude and Longitude coordinates in node objects reflect real positions of the intersections, and edge weights are calculated using real distances (measured on Google Maps) scaled by a weighting factor whose value is determined by the level of observed wheelchair accessibility (according to OSM guidelines (see Fig. 1)*

2.5 Evaluating Heuristic Error:

Heuristic error was evaluated from nodes "A to AE" on the above graph with a real observed distance of 1.84km

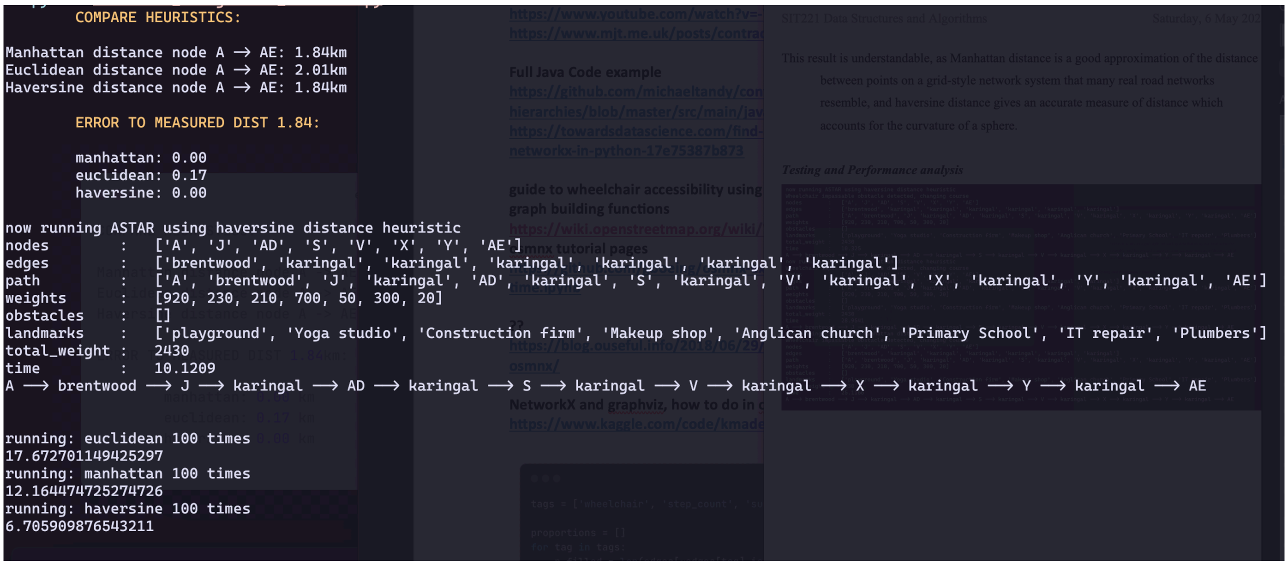


```
COMPARE HEURISTICS:

Manhattan distance node A -> AE: 1.84km
Euclidean distance node A -> AE: 2.01km
Haversine distance node A -> AE: 1.84km

ERROR TO MEASURED DIST 1.84km:

        manhattan: 0.00 km
        euclidean: 0.17 km
        haversine: 0.00 km
```

Figure 6. Error between heuristic distance predictions and measured values

This result is not unexpected. Manhattan distance provides good approximations for the distance between points on grid-like networks that resemble many real-world traffic networks, and Haversine distance gives accurate measures of distance which account for the curvature of the earth.

2.6 Testing and Performance analysis



```
        COMPARE HEURISTICS:

Manhattan distance node A → AE: 1.84km
Euclidean distance node A → AE: 2.01km
Haversine distance node A → AE: 1.84km

        ERROR TO MEASURED DIST 1.84:

        manhattan: 0.00
        euclidean: 0.17
        haversine: 0.00

now running ASTAR using haversine distance heuristic
nodes        :   ['A', 'J', 'AD', 'S', 'V', 'X', 'Y', 'AE']
edges        :   ['brentwood', 'karingal', 'karingal', 'karingal', 'karingal', 'karingal', 'karingal']
path         :   ['A', 'brentwood', 'J', 'karingal', 'AD', 'karingal', 'S', 'karingal', 'V', 'karingal', 'X', 'karingal', 'Y', 'karingal', 'AE']
weights      :   [920, 230, 210, 700, 50, 300, 20]
obstacles    :   []
landmarks    :   ['playground', 'Yoga studio', 'Construction firm', 'Makeup shop', 'Anglican church', 'Primary School', 'IT repair', 'Plumbers']
total_weight :   2430
time         :   10.1209
A ⟶ brentwood ⟶ J ⟶ karingal ⟶ AD ⟶ karingal ⟶ S ⟶ karingal ⟶ V ⟶ karingal ⟶ X ⟶ karingal ⟶ Y ⟶ karingal ⟶ AE


running: euclidean 100 times
17.672701149425297
running: manhattan 100 times
12.164474725274726
running: haversine 100 times
6.705909876543211
```

Figure 7. *Upper: example of complete path reconstruction on search graph following A* using the haversine heuristic function. Lower: mean runtime (ms) over 100 iterations of A* using Euclidean, Manhattan and Haversine heuristics.*

Tests involved first proving the correctness of the path building functionality of the A* implementation, followed by averaging the performance of the three metrics over 100 iterations of A*. The clear performance winner in this case was the haversine metric. Explanations for this could include:

- Lower observed error between measured and predicted goal distance could provide for a more optimised node expansion order. (Fig. 6)

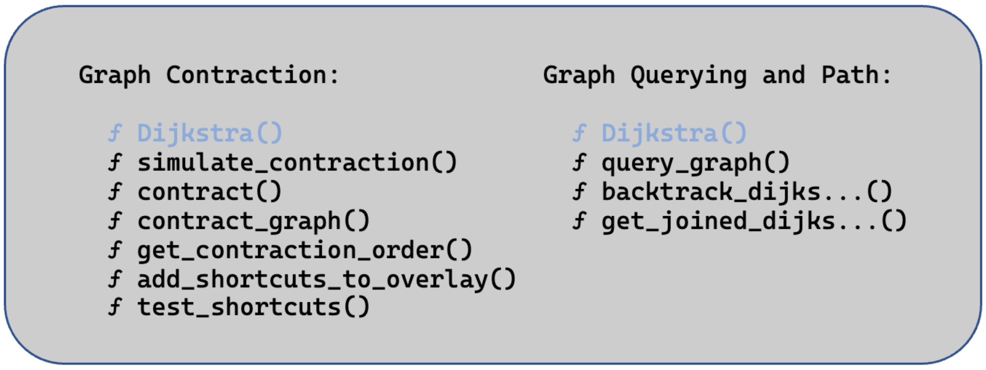
- The function is imported as a third-party dependency and could be implemented using native C/C++ code which can leverage low-level compilation optimisation, type safety and/or memory manipulation for drastically improved observed performance.

2.7 Contraction Hierarchies

As explained in sections **1.3-1.5,** the CH (Contraction Hierarchies) algorithm, applied on some graph $G = \{V, E\}$ can be separated into two distinct sub-processes:

1. **Graph contraction**: contract each node $v \in V$ by order of *importance*, given by the simulated contraction of G, such that an overlay graph G' is produced which persists said order through artificial subgraphs G'_D and G'_U.
2. **Graph query:** Execute consecutive complete Dijkstra's searches: one from source node s on G'_U in the *upwards* direction with respect to the contraction order; and two, from target node t on G'_D in the *downwards* direction with respect to the contraction order. The path is reconstructed via back-tracing from the optimal (lowest summed cost) node found in the intersection of the set of nodes reached by both searches.

2.8 Implementation:



```
Graph Contraction:                Graph Querying and Path:

  ƒ Dijkstra()                      ƒ Dijkstra()
  ƒ simulate_contraction()          ƒ query_graph()
  ƒ contract()                      ƒ backtrack_dijks...()
  ƒ contract_graph()                ƒ get_joined_dijks...()
  ƒ get_contraction_order()
  ƒ add_shortcuts_to_overlay()
  ƒ test_shortcuts()
```

Figure 8. *CH graph contraction and querying function overview for 218344989_assignment2_solution.py*

Please refer to the above as a guide when searching through the submission. The source code is well-documented with frequent but concise logging to std. output.

2.9 Challenges

I encountered severe difficulties implementing the contraction hierarchies algorithm. After the relative success of the custom graph implementation for use with A*, every line of the contraction hierarchies process seemed to hit bizarre edge cases in my code that I had not previously accounted for. While I do believe this resulted in a more robust data structure, this was not the point of the assignment. Although I reached what I believe is a working solution, far too many hours were spent fixing the graph for the search problem instead of the inverse.

2.10 Lessons Learned

If I were to repeat an attempt at the Contraction Hierarchies algorithm, I would utilise existing, and more importantly *working* graph implementations such as those provided by the NetworkX library so that more time can be spent devising solutions to problems that are actually within the scope of the task.

2.11 Performance analysis and comparison