Warsaw University of Technology Faculty of Geodesy and Cartography

Project

Adam Korejwo

Yagmur Soydan Huner

Sandrine Uhmuruza

Aim of task

Implement routing algorithms for spatial data of the road network

Data used

OSM data:

The OpenStreetMap Foundation is a non-profit foundation whose aim is to support and enable the development of freely-reusable geospatial data. It is closely connected with the OpenStreetMap project, although its constitution does not prevent it supporting other projects.

Software used

ArcMap:

ArcMap is the main component of Esri's ArcGIS suite of geospatial processing programs, and is used primarily to view, edit, create, and analyze geospatial data. ArcMap allows the user to explore data within a data set, symbolize features accordingly, and create maps.

The software language used

Python:

Python is a high-level, interpreted, general-purpose programming language. Its design philosophy emphasizes code readability with the use of significant indentation. Python is dynamically-typed and garbage-collected. It supports multiple programming paradigms, including structured (particularly procedural), object-oriented and functional programming. It is often described as a "batteries included" language due to its comprehensive standard library.

ArcPy Library:

ArcPy is a Python site package that provides a useful and productive way to perform geographic data analysis, data conversion, data management, and map automation with Python. This package provides a rich and native Python experience offering code completion (type a keyword and a dot to get a pop-up list of properties and methods supported by that keyword; select one to insert it) and reference documentation for each function, module, and class. The additional power of using ArcPy is that Python is a general-purpose programming language. It is interpreted and dynamically typed and is suited for interactive work and quick prototyping of one-off programs known as scripts while being powerful enough to write large applications in. ArcGIS applications written with ArcPy benefit from the development of additional modules in numerous niches of Python by GIS professionals and programmers from many different disciplines.

Queue Library:

The queue module implements multi-producer, multi-consumer queues. It is especially useful in threaded programming when information must be exchanged safely between multiple threads. The Queue class in this module implements all the required locking semantics.

class queue.PriorityQueue(maxsize=0):

Constructor for a priority queue. maxsize is an integer that sets the upperbound limit on the number of items that can be placed in the queue. Insertion will block once this size has been reached, until queue items are consumed. If maxsize is less than or equal to zero, the queue size is infinite.

Graph theory

In mathematics, graph theory is the study of graphs, which are mathematical structures used to model pairwise relations between objects. A graph in this context is made up of vertices (also called nodes or points) which are connected by edges (also called links or lines). A distinction is made between undirected graphs, where edges link two vertices symmetrically, and directed graphs, where edges link two vertices asymmetrically. Graphs are one of the principal objects of study in discrete mathematics.

Steps:

Graph construction

Read external dataset and create a graph representing road network.

Read data:

Cursors:

A cursor is a data access object that can be used to either iterate over the set of rows in a table or insert new rows into a table. Cursors have three forms: search, insert, and update. Cursors are commonly used to read existing geometries and write new geometries. Each type of cursor is created by a corresponding ArcPy function (SearchCursor, InsertCursor, or UpdateCursor) on a table, table view, feature class, or feature layer. A search cursor can be used to retrieve rows. An update cursor can be used to update and delete rows, while an insert cursor is used to insert rows into a table or feature class.

Cursor	Explanation
`arcpy.da.InsertCursor(in_table, field_names)`	Inserts rows
`arcpy.da.SearchCursor(in_table, field_names, {where_clause}, {spatial_reference}, {explode_to_points}, {sql_clause})`	Read-only access
`arcpy.da.UpdateCursor(in_table, field_names, {where_clause}, {spatial_reference}, {explode_to_points}, {sql_clause})`	Updates or deletes rows

cursor = arcpy.SearchCursor(layer)

Access the geometry field of objects and points of geometry:

SHAPE:

Accessing full geometry with SHAPE@ is an expensive operation. If only simple geometry information is required, such as the x,y coordinates of a point, use tokens such as SHAPE@XY, SHAPE@Z, and SHAPE@M for faster, more efficient access.

row_id = row.OBJECTID

start_point = row.Shape.firstPoint.X, row.Shape.firstPoint.Y

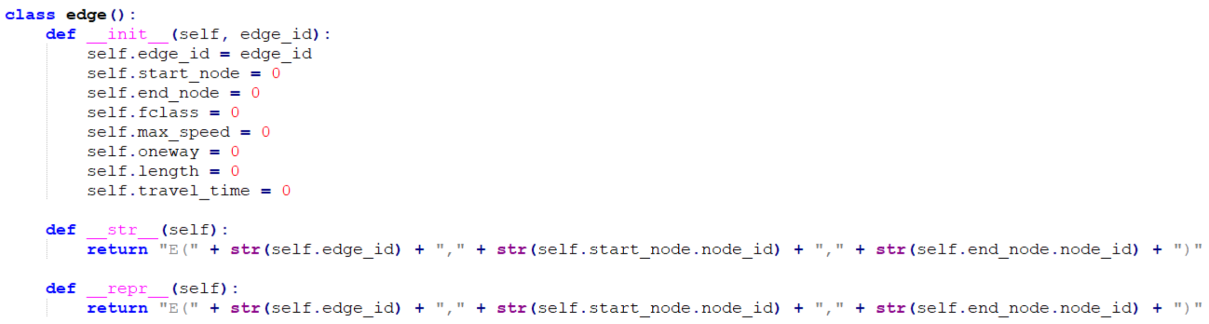
end_point = row.Shape.lastPoint.X, row.Shape.lastPoint.Y

Generate edges and nodes

Create a class for edges:

In geometry, an edge is a particular type of line segment joining two vertices in a polygon, polyhedron, or higher-dimensional polytope. In a polygon, an edge is a line segment on the boundary, and is often called a polygon side.

In our code, the connection between nodes which is include; id, x,y of the first point of linear geometry, x,y of the last point of linear geometry



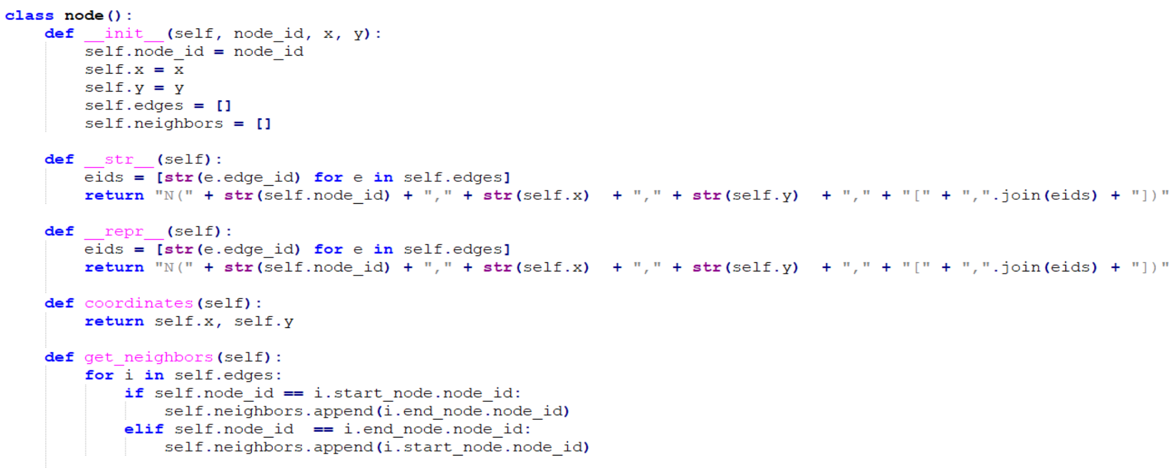
```
class edge():
    def __init__(self, edge_id):
        self.edge_id = edge_id
        self.start_node = 0
        self.end_node = 0
        self.fclass = 0
        self.max_speed = 0
        self.oneway = 0
        self.length = 0
        self.travel_time = 0

    def __str__(self):
        return "E(" + str(self.edge_id) + "," + str(self.start_node.node_id) + "," + str(self.end_node.node_id) + ")"

    def __repr__(self):
        return "E(" + str(self.edge_id) + "," + str(self.start_node.node_id) + "," + str(self.end_node.node_id) + ")"
```

Figure 1- edge class

Create a class for nodes:

In mathematics, and more specifically in graph theory, a vertex (plural vertices) or node is the fundamental unit of which graphs are formed: an undirected graph consists of a set of vertices and a set of edges (unordered pairs of vertices), while a directed graph consists of a set of vertices and a set of arcs (ordered pairs of vertices). In a diagram of a graph, a vertex is usually represented by a circle with a label, and an edge is represented by a line or arrow extending from one vertex to another.

In our code, vertices which is include; id, x,y of a corresponding junction



```
class node():
    def __init__(self, node_id, x, y):
        self.node_id = node_id
        self.x = x
        self.y = y
        self.edges = []
        self.neighbors = []

    def __str__(self):
        eids = [str(e.edge_id) for e in self.edges]
        return "N(" + str(self.node_id) + "," + str(self.x)  + "," + str(self.y)  + "," + "[" + ",".join(eids) + "])"

    def __repr__(self):
        eids = [str(e.edge_id) for e in self.edges]
        return "N(" + str(self.node_id) + "," + str(self.x)  + "," + str(self.y)  + "," + "[" + ",".join(eids) + "])"

    def coordinates(self):
        return self.x, self.y

    def get_neighbors(self):
        for i in self.edges:
            if self.node_id == i.start_node.node_id:
                self.neighbors.append(i.end_node.node_id)
            elif self.node_id  == i.end_node.node_id:
                self.neighbors.append(i.start_node.node_id)
```

Figure 2- node class

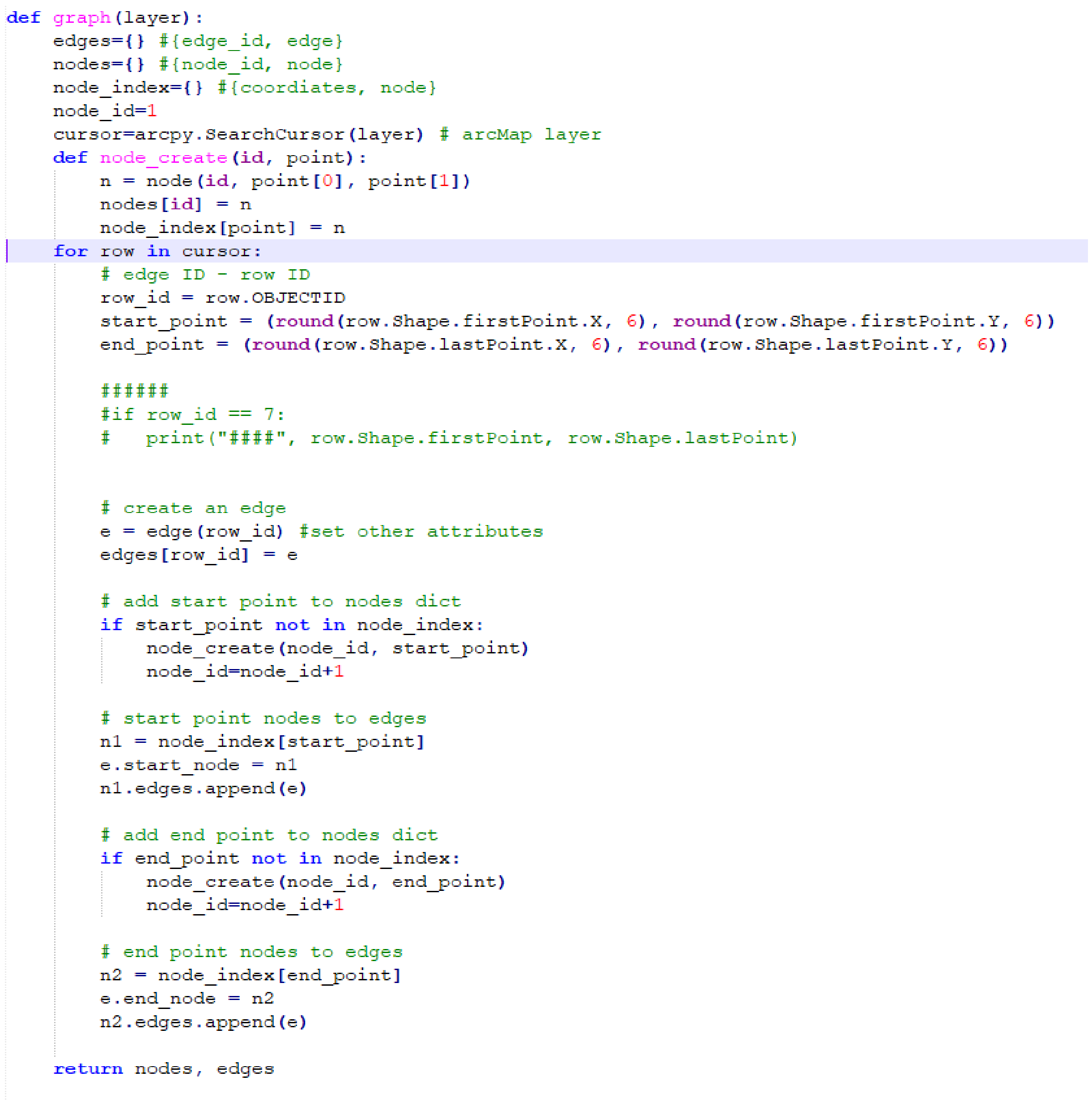
Create graph function

Graph

In one restricted but very common sense of the term, a graph is an ordered pair $G = (V, E)$ comprising:

- V , a set of vertices (also called nodes or points);
- a set of edges (also called links or lines), which are unordered pairs of vertices (that is, an edge is associated with two distinct vertices).

$$E \subseteq \{\{x, y\} \mid x, y \in V \text{ and } x \neq y\}$$



```
def graph(layer):
    edges={} #{edge_id, edge}
    nodes={} #{node_id, node}
    node_index={} #{coordiates, node}
    node_id=1
    cursor=arcpy.SearchCursor(layer) # arcMap layer
    def node_create(id, point):
        n = node(id, point[0], point[1])
        nodes[id] = n
        node_index[point] = n
    for row in cursor:
        # edge ID - row ID
        row_id = row.OBJECTID
        start_point = (round(row.Shape.firstPoint.X, 6), round(row.Shape.firstPoint.Y, 6))
        end_point = (round(row.Shape.lastPoint.X, 6), round(row.Shape.lastPoint.Y, 6))

        ######
        #if row_id == 7:
        #   print("####", row.Shape.firstPoint, row.Shape.lastPoint)


        # create an edge
        e = edge(row_id) #set other attributes
        edges[row_id] = e

        # add start point to nodes dict
        if start_point not in node_index:
            node_create(node_id, start_point)
            node_id=node_id+1

        # start point nodes to edges
        n1 = node_index[start_point]
        e.start_node = n1
        n1.edges.append(e)

        # add end point to nodes dict
        if end_point not in node_index:
            node_create(node_id, end_point)
            node_id=node_id+1

        # end point nodes to edges
        n2 = node_index[end_point]
        e.end_node = n2
        n2.edges.append(e)

    return nodes, edges
```

Figure 3- Graph code

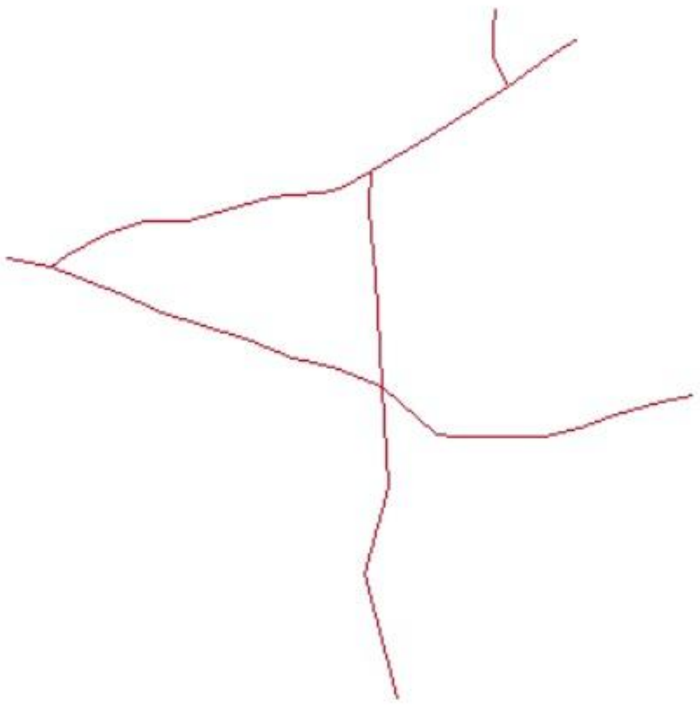


Figure 4- testing data set

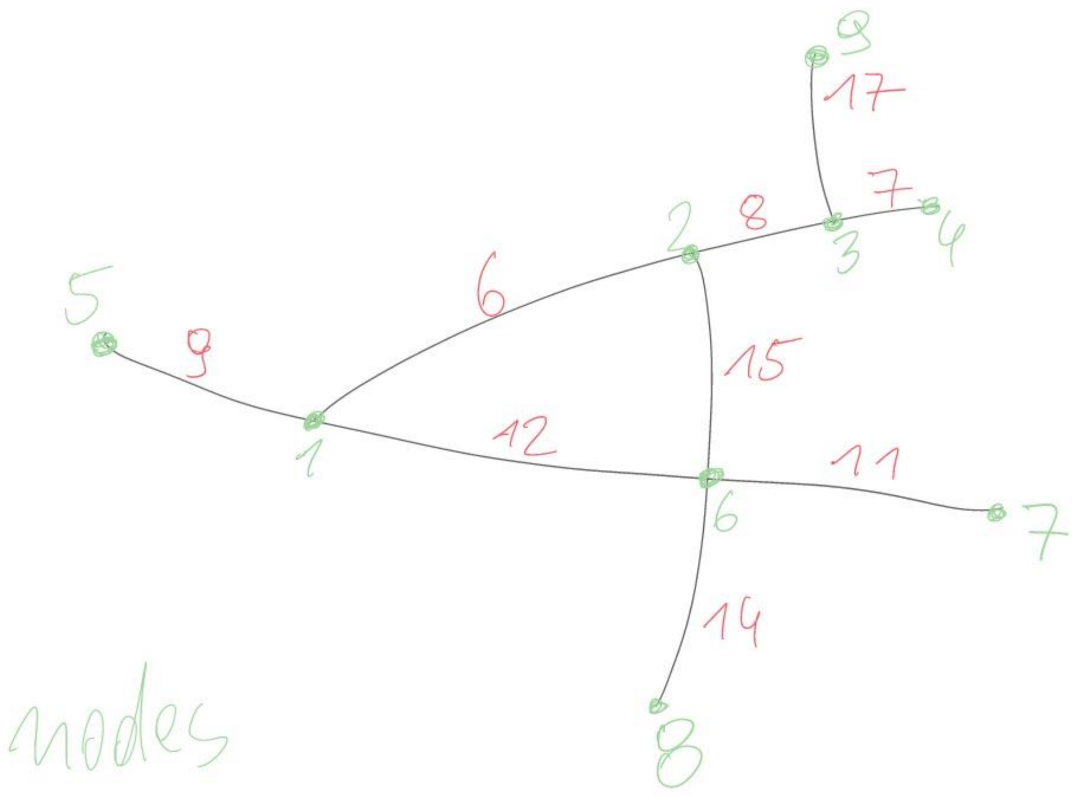
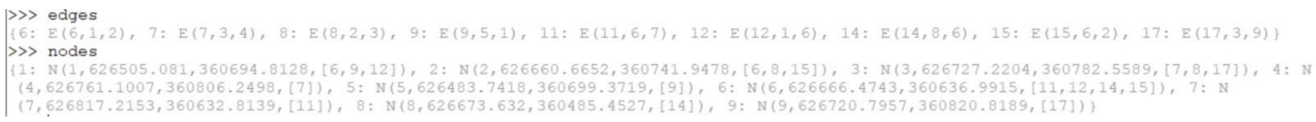


Figure 5-id of edges and nodes



```
>>> edges
{6: E(6,1,2), 7: E(7,3,4), 8: E(8,2,3), 9: E(9,5,1), 11: E(11,6,7), 12: E(12,1,6), 14: E(14,8,6), 15: E(15,6,2), 17: E(17,3,9)}
>>> nodes
{1: N(1,626505.081,360694.8128,[6,9,12]), 2: N(2,626660.6652,360741.9478,[6,8,15]), 3: N(3,626727.2204,360782.5589,[7,8,17]), 4: N
 (4,626761.1007,360806.2498,[7]), 5: N(5,626483.7418,360699.3719,[9]), 6: N(6,626666.4743,360636.9915,[11,12,14,15]), 7: N
 (7,626817.2153,360632.8139,[11]), 8: N(8,626673.632,360485.4527,[14]), 9: N(9,626720.7957,360820.8189,[17])}
```

Figure 6- Graph code result

Implementation of graph algorithms

A* Algorithms:

A* (pronounced "A-star") is a graph traversal and path search algorithm, which is often used in many fields of computer science due to its completeness, optimality, and optimal efficiency. One major practical drawback is its $O(b^d)$ space complexity, as it stores all generated nodes in memory. Thus, in practical travel-routing systems, it is generally outperformed by algorithms which can pre-process the graph to attain better performance, as well as memory-bounded approaches; however, A* is still the best solution in many cases.

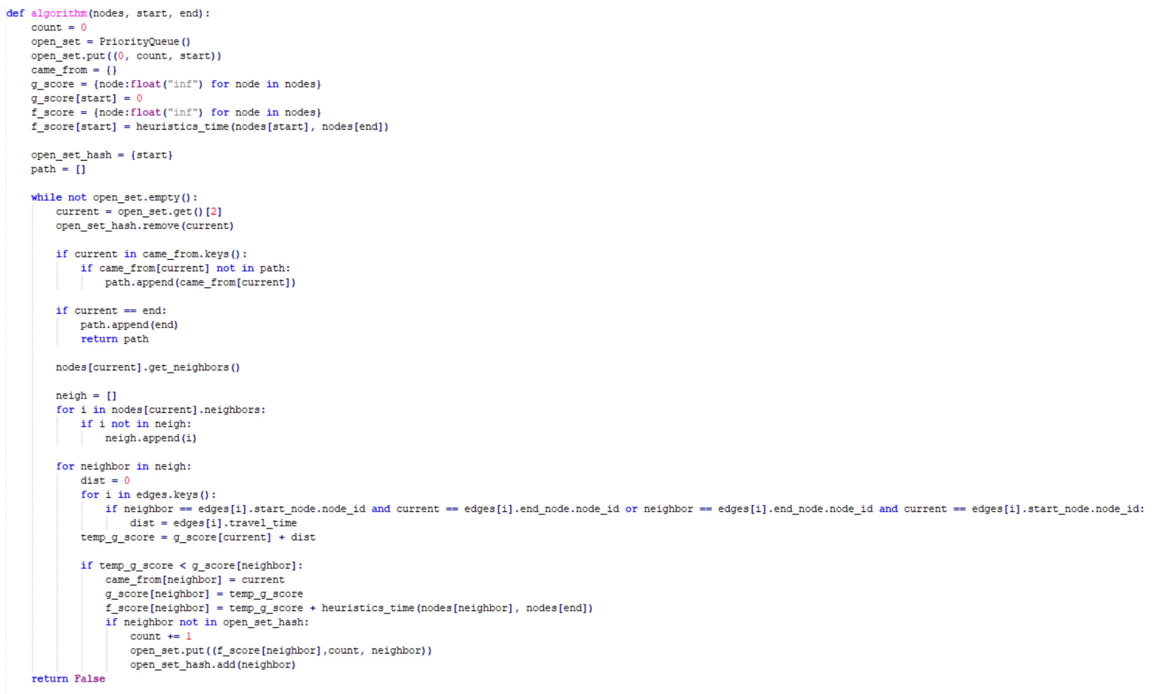
In our code; as input we need starting and ending nodes. Algorithms recognizes node's neighbors, reads distance between them, and calculate heuristics.

For the shortest path; Heuristics are calculated as Euclidian distance to the end point

For the fastest path; Heuristics are calculated as Euclidian distance to the end point divided by speed limit.

According to OSM data; many feature don't have speed limit so for that reason we have benefited from the following website

https://wiki.openstreetmap.org/wiki/Key:maxspeed



```
def algorithm(nodes, start, end):
    count = 0
    open_set = PriorityQueue()
    open_set.put((0, count, start))
    came_from = {}
    g_score = {node:float("inf") for node in nodes}
    g_score[start] = 0
    f_score = {node:float("inf") for node in nodes}
    f_score[start] = heuristics_time(nodes[start], nodes[end])

    open_set_hash = {start}
    path = []

    while not open_set.empty():
        current = open_set.get()[2]
        open_set_hash.remove(current)

        if current in came_from.keys():
            if came_from[current] not in path:
                path.append(came_from[current])

        if current == end:
            path.append(end)
            return path

        nodes[current].get_neighbors()

        neigh = []
        for i in nodes[current].neighbors:
            if i not in neigh:
                neigh.append(i)

        for neighbor in neigh:
            dist = 0
            for i in edges.keys():
                if neighbor == edges[i].start_node.node_id and current == edges[i].end_node.node_id or neighbor == edges[i].end_node.node_id and current == edges[i].start_node.node_id:
                    dist = edges[i].travel_time
            temp_g_score = g_score[current] + dist

            if temp_g_score < g_score[neighbor]:
                came_from[neighbor] = current
                g_score[neighbor] = temp_g_score
                f_score[neighbor] = temp_g_score + heuristics_time(nodes[neighbor], nodes[end])
                if neighbor not in open_set_hash:
                    count += 1
                    open_set.put((f_score[neighbor],count, neighbor))
                    open_set_hash.add(neighbor)
    return False
```

Figure 7- A Algorithm*

```python
def main(layer, start, end):
    nodes, edges, road_classes = graph(layer)

    max_speeds = {'path':20000, 'track':50000}
    top_speed = max(max_speeds.values())

    path = algorithm(nodes, start, end)

    return path
```

Figure 8- A fastest road*

```python
def main(layer, start, end):
    nodes, edges, road_classes = graph(layer)

    top_speed = 1

    path = algorithm(nodes, start, end)

    return path
```

Figure 9- A shortest road*

```
('=== current node ', 4, ' ===')
('f values: ', {1: inf, 2: inf, 3: 0.001747073182526061, 4: 0.0018228784858561943, 5: inf, 6: inf, 7: inf, 8: inf, 9: inf})
('=== current node ', 3, ' ===')
('f values: ', {1: inf, 2: 0.0019083537916021016, 3: 0.001747073182526061, 4: 0.0018228784858561943, 5: inf, 6: inf, 7: inf, 8: inf,
 9: 0.0021128800081681443})
('=== current node ', 2, ' ===')
('f values: ', {1: 0.0031823212414481693, 2: 0.0019083537916021016, 3: 0.001747073182526061, 4: 0.0018228784858561943, 5: inf, 6:
 0.0015079887739227\86, 7: inf, 8: inf, 9: 0.0021128800081681443})
('=== current node ', 6, ' ===')
('f values: ', {1: 0.0031823212414481693, 2: 0.0019083537916021016, 3: 0.001747073182526061, 4: 0.0018228784858561943, 5: inf, 6:
 0.001507988773922786, 7: 0.0, 8: 0.0020574617202840442, 9: 0.0021128800081681443})
```

Figure 10- Steps of A algorithm*

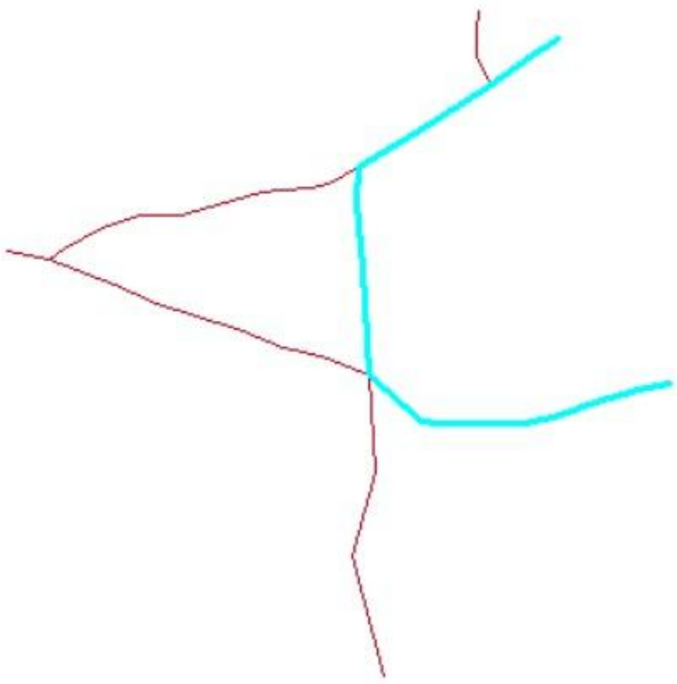


Figure 11 – A Result of shortest path(4-7)*

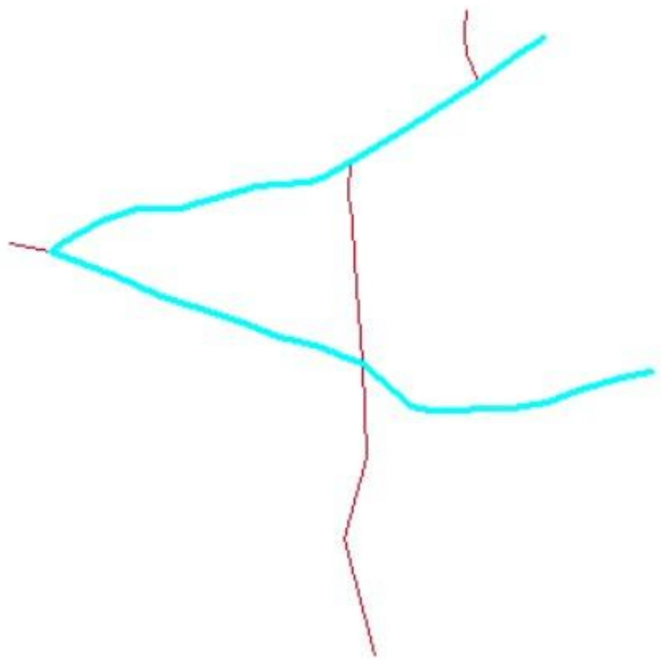


Figure 12 -A result of fastest road(4-7)*

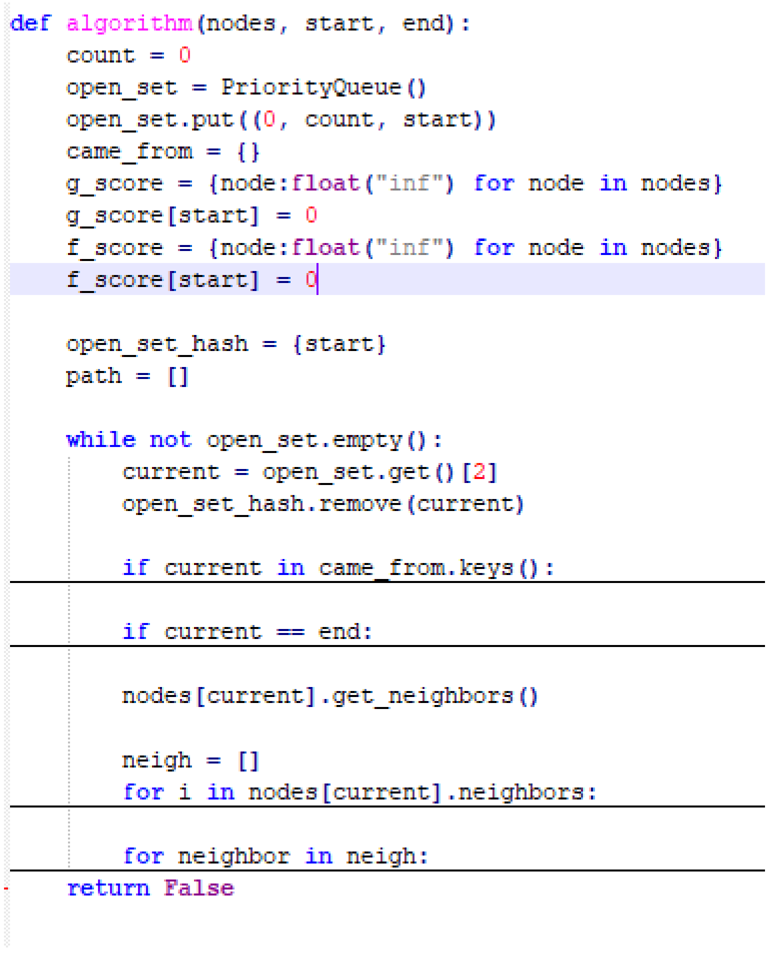
Dijkstra Algorithms:

Dijkstra's algorithm is an algorithm for finding the shortest paths between nodes in a graph, which may represent, for example, road networks. It was conceived by computer scientist Edsger W. Dijkstra in 1956 and published three years later.

The algorithm exists in many variants. Dijkstra's original algorithm found the shortest path between two given nodes,[6] but a more common variant fixes a single node as the "source" node and finds shortest paths from the source to all other nodes in the graph, producing a shortest-path tree.

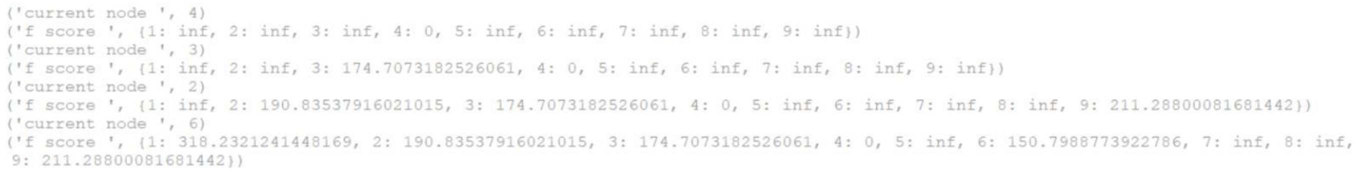
In our code; as input we need starting and ending nodes. Algorithms recognizes node’s neighbors, reads distance between them.

We set heuristics as ‘0’ in our script.



```python
def algorithm(nodes, start, end):
    count = 0
    open_set = PriorityQueue()
    open_set.put((0, count, start))
    came_from = {}
    g_score = {node:float("inf") for node in nodes}
    g_score[start] = 0
    f_score = {node:float("inf") for node in nodes}
    f_score[start] = 0

    open_set_hash = {start}
    path = []

    while not open_set.empty():
        current = open_set.get()[2]
        open_set_hash.remove(current)

        if current in came_from.keys():

        if current == end:

        nodes[current].get_neighbors()

        neigh = []
        for i in nodes[current].neighbors:

        for neighbor in neigh:
    return False
```

Figure 13- Dijkstra Algorithms codes



```
('current node ', 4)
('f score ', {1: inf, 2: inf, 3: inf, 4: 0, 5: inf, 6: inf, 7: inf, 8: inf, 9: inf})
('current node ', 3)
('f score ', {1: inf, 2: inf, 3: 174.7073182526061, 4: 0, 5: inf, 6: inf, 7: inf, 8: inf, 9: inf})
('current node ', 2)
('f score ', {1: inf, 2: 190.83537916021015, 3: 174.7073182526061, 4: 0, 5: inf, 6: inf, 7: inf, 8: inf, 9: 211.28800081681442})
('current node ', 6)
('f score ', {1: 318.2321241448169, 2: 190.83537916021015, 3: 174.7073182526061, 4: 0, 5: inf, 6: 150.7988773922786, 7: inf, 8: inf,
 9: 211.28800081681442})
```

Figure 14-Steps of Dijkstra algorithm

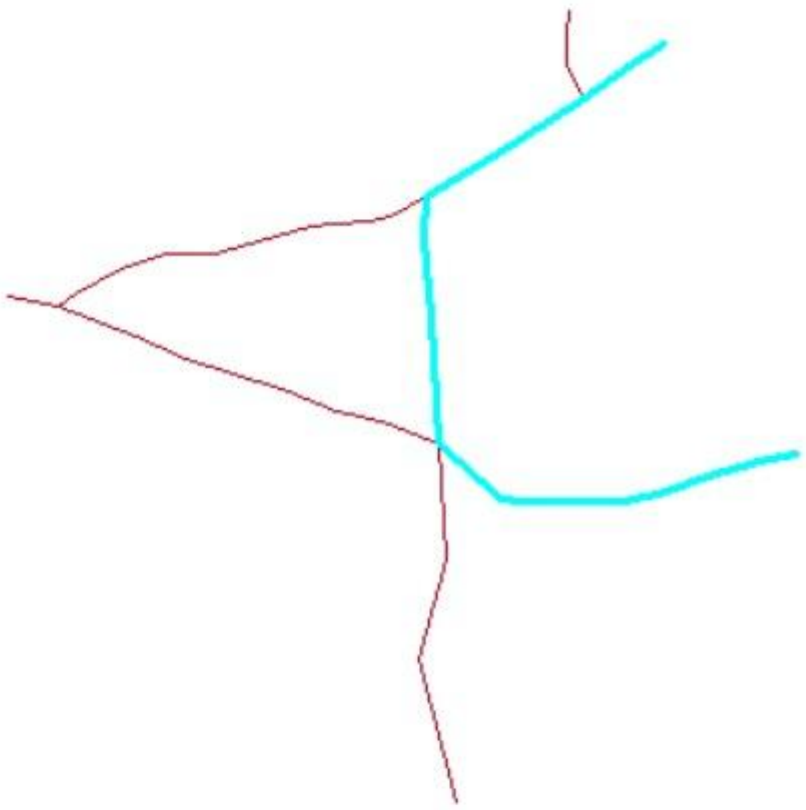


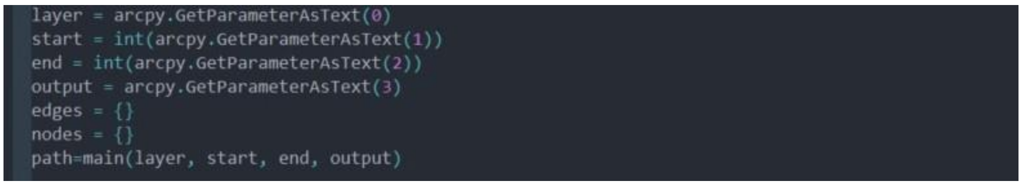
Figure 15- Dijkstra Algorithm

Integrate with ArcMap

Goal:

To enable a user to run routing algorithms in ESRI ArcMap and display the results

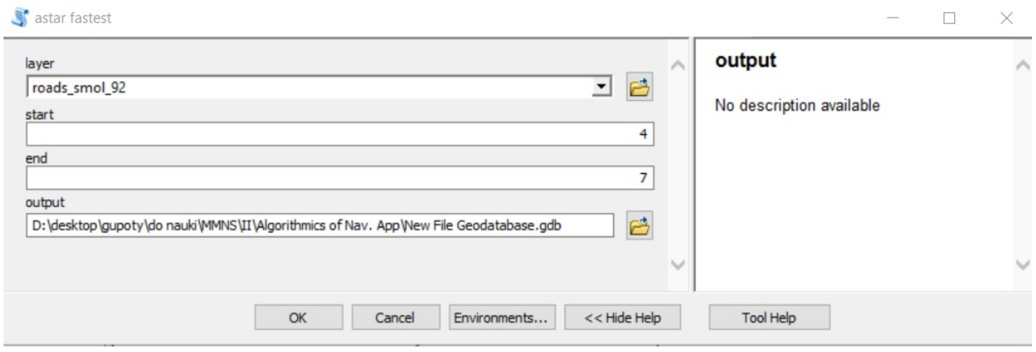
Choosing starting and destination points of the route:



```
layer = arcpy.GetParameterAsText(0)
start = int(arcpy.GetParameterAsText(1))
end = int(arcpy.GetParameterAsText(2))
output = arcpy.GetParameterAsText(3)
edges = {}
nodes = {}
path=main(layer, start, end, output)
```

GetParameterAsText (index)

- Gets the specified parameter as a text string by its index position from the list of parameters



astar fastest

layer: roads_smol_92

start: 4

end: 7

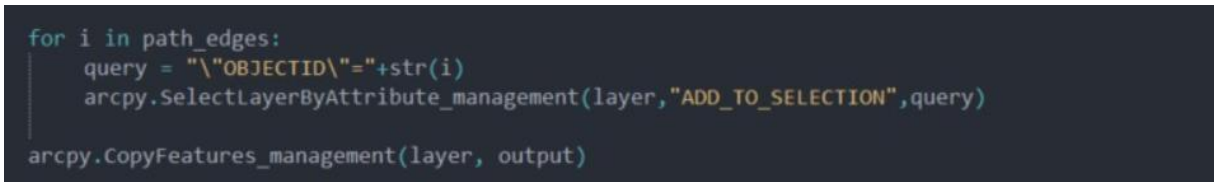
output: D:\desktop\gupoty\do nauki\MMNS\II\Algorithmics of Nav. App\New File Geodatabase.gdb

output

No description available

We implement algorithm using nodes id. This is not useful way but we can improve by coordinates.

Display calculated route

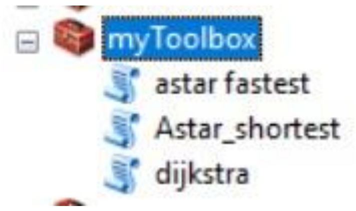


```
for i in path_edges:
    query = "\"OBJECTID\"="+str(i)
    arcpy.SelectLayerByAttribute_management(layer,"ADD_TO_SELECTION",query)

arcpy.CopyFeatures_management(layer, output)
```

We can display calculated route with two ways.

- With Select Layer By Attribute (it will select calculated road)
    - Adds, updates, or removes a selection based on an attribute query arcpy.management.SelectLayerByAttribute(in_layer_or_view, {selection_type}, {where_clause}, {invert_where_clause})
- With Copy Features (it will save copy of calculated road to selected workspacew)
    - Copies features from the input feature class or layer to a new feature class arcpy.management.CopyFeatures(in_features, out_feature_class, {config_keyword}, {spatial_grid_1}, {spatial_grid_2}, {spatial_grid_3})

Add your script to Toolbox or create an Add-in



We create three tools for A* shortest and fastest path and Dijksta.

Conclusion

We implemented certain algorithms using the Arcpy and Queue libraries. Firstly, we created the graph based on graph theory rules. This graph consists of edges and nodes. With input data prepared this way we were able to apply the A* and Dijkstra algorithms returning paths between start and end point of our choice. The algorithm scripts were later integrated into ArcMap software by creation of a toolbox. We implement algorithm using nodes id created in data preprocessing. This is not the most handy way of carrying the data but can be improved.

Resources

https://en.wikipedia.org/wiki/OpenStreetMap_Foundation

https://en.wikipedia.org/wiki/ArcMap

https://en.wikipedia.org/wiki/Python_(programming_language)

https://pro.arcgis.com/en/pro-app/2.8/arcpy/get-started/what-is-arcpy-.htm

https://pro.arcgis.com/en/pro-app/2.8/arcpy/get-started/data-access-using-cursors.htm

https://docs.python.org/3/library/queue.html

https://en.wikipedia.org/wiki/A*_search_algorithm