

Project Report  
City of Vancouver Navigation Toolbox Development  
Chao Zhang  
August 8, 2018

**CS 890EQ - Topics in GIS:  
Spatial Analysis and Geoprocessing**

**Final Project Report**

**City of Vancouver Navigation Toolbox Development**

Chao Zhang 200383834

University of Regina

Department of Computer Science

August 8, 2018

## **Abstract**

As the demand for navigation access increases so does the improvement of software development, more and more navigation tools have been created for efficient life among cities.

This project is to create a new navigation toolbox based on ArcGIS by python scripts to provide users shortest route information among two arbitrary places among the city of Vancouver.

The input are citizens addresses ID or names of different places with their types and results can offer users detailed navigation including street names, travelling distances and positions of street intersections.

***Key word: navigation; shortest route; Dijkstra's algorithm.***

## Contents

<b>1.0 Introduction:</b>	<b>1</b>
<b>2.0 Literature Review</b>	<b>2</b>
2.1 Previous Work Review	2
2.2 General steps of Dijkstra's Algorithm	2
2.3 Similarities and Differences with Approaches	3
2.4 Apply Dijkstra's Algorithm into City Network	3
2.5 Improvement of Dijkstra's Algorithm	4
<b>3.0 Study Area Description</b>	<b>5</b>
<b>4.0 Description of Datasets &amp; Data Preprocessing</b>	<b>5</b>
4.1 Listing Table of Input Data	6
4.2 Listing Table of Data for Spatial Analysis	7
4.3 Listing Table of Attributes for Spatial Analysis	7
4.4 Data Collecting and Geodatabase Development	7
4.5 Data Checking and Preprocessing	8
<b>5.0 Methodology</b>	<b>8</b>
5.1 Flowchart of Solution	9
5.2 List of Created Functions	11
5.3 Detailed Description of Methods	12
<b>6.0 Results and Discussion</b>	<b>24</b>
6.1 Created Map	25
6.2 Results of Navigation Information	26
<b>7.0 Assumptions and Limitations</b>	<b>34</b>
<b>8.0 Conclusion and Future Work</b>	<b>35</b>
<b>References:</b>	<b>36</b>
<b>Appendix</b>	<b>37</b>

## **1.0 Introduction:**

Nowadays, navigation plays a more and more significant role in urban city life especially in some big cities. It has totally replaced traditional maps for users to find a location or select an optimal route among cities in our daily life.

In this project, a navigation toolbox has been created containing four functions in city of Vancouver based on geographic information system(GIS). Users can use this toolbox to easily find nearest or specified school, library or school from home among Vancouver, get shortest a route between two properties and obtain a shortest path between two given different places. It will show users detailed information in navigation which contains street names, street lengths and positions of intersections.

The motivation of undertaking the research is from daily life. Every time when I use Google map, I notice it is an essential software among people and wonder if there is possible to make a navigation software like that by myself.

GIS provides XY coordinates and some attributes of every object in Vancouver, relationships among different objects can be found and spatial positions with geographic references are reliable to analyze to create new toolbox. The general solution for navigation in this project is to locate start and end points first, then choose network area covering these two points and last analyze this network area to provide an optimal path. In the rest of this report, it will illustrate data collection, flowchart of solution, details in network area analysis which is how to find an optimal path and Dijkstra's algorithm applied in shortest route calculation.

## **2.0 Literature Review**

In this section of report, it provides previous research with new ideas and changes to current research, core algorithm used in project, how algorithm be applied into project and algorithm improvements.

### **2.1 Previous Work Review**

In previous work, UNA toolbox from MIT City Form Lab is used to solve the shortest route problem and it can get the shortest path easily just by input start and end data. However, a new idea came to my mind is to create a navigation toolbox by myself which can calculate the shortest path among the whole Vancouver city network based on Dijkstra's algorithm.

### **2.2 General steps of Dijkstra's Algorithm**

Dijkstra's algorithm is an advanced version of greedy algorithm for finding the shortest path between nodes in a graph. Dijkstra, E. W. (1959) first found there was an approach to construct the tree of minimum total length between the n nodes and find the path of minimum total length between two given nodes to get a shortest path. The general steps of this approach can be concluded in six step: 1. Mark all unvisited nodes and create a set containing all unvisited nodes. 2. Set initial node value 0 as current node and assign every node a tentative distance value. 3. Calculate the distance between current node and all its neighbour nodes to get the shortest tentative distance and mark it. 4. Once checking all neighbour nodes of current node, mark

current node which we have visited and remove it from the set we generated in step one. 5. When the shortest tentative distance is infinity or it has come to the final unvisited node, Dijkstra's algorithm has finished. 6. If it does not come to the end, select the unvisited node with shortest distance as current node and go back to step three.

## **2.3 Similarities and Differences with Approaches**

For similarities, Dijkstra's algorithm provides a mathematical solution for finding a shortest path among a weighted graph and the project purpose is also to yield a shortest route among the city network. However, Dijkstra's algorithm can not be used directly among city network and there is a model need to be created for application.

## **2.4 Apply Dijkstra's Algorithm into City Network**

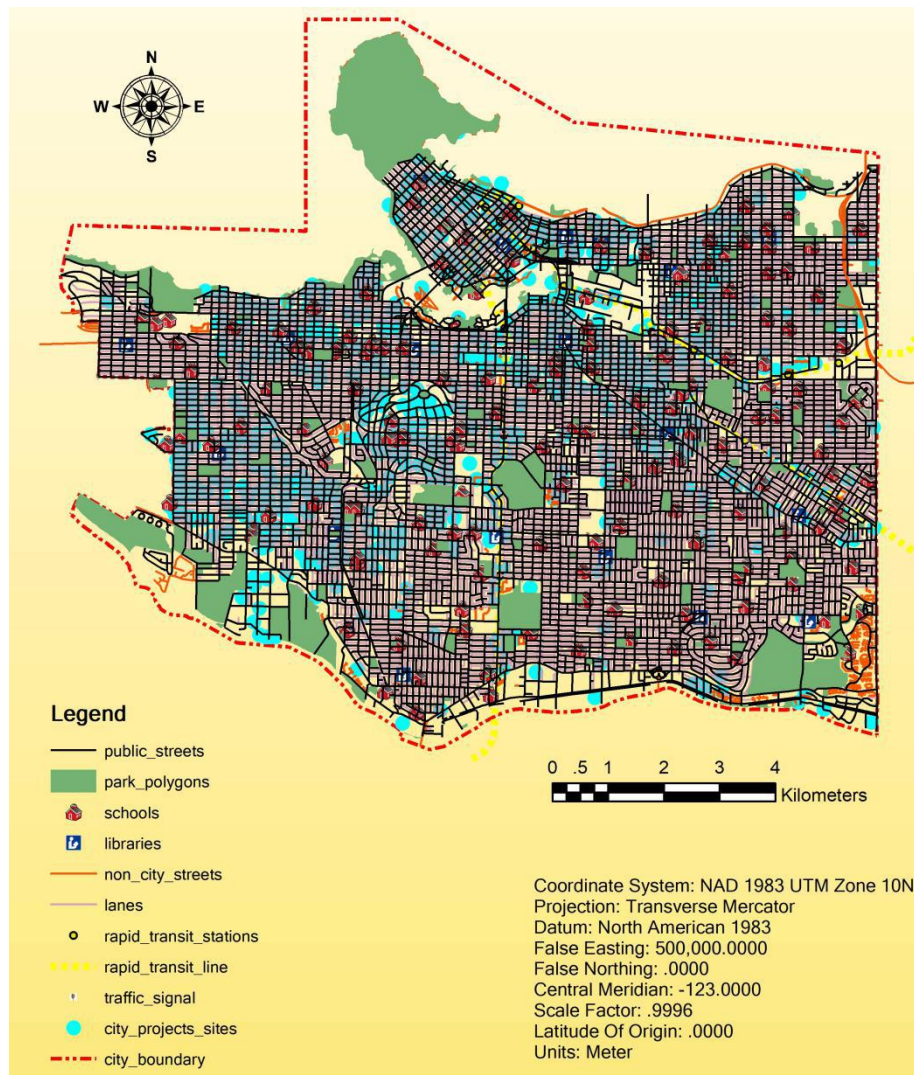
Building city network for Dijkstra's algorithm to use is to create a weighted graph based on street intersections where street intersections represent for nodes in graph and different street lengths between two nodes represent for corresponding distances between two intersections. Due to the large data set consisted of whole intersections and streets in Vancouver should be processed, there is impossible to store all data and use Dijkstra's algorithm to calculate a shortest path among the whole city. What has been done is to find a particular area covering start and end points and then perform calculation in this specific area which can save plenty of time and storage space.

## **2.5 Improvement of Dijkstra's Algorithm**

In DongKai Fan and Ping Shi (2010) research, they improved storage structure and searching area algorithm which are two drawbacks of classical Dijkstra's algorithm. Their algorithm appended analysis of space complexity and analysis of time complexity for storage structure and set restricted search area when finding next node during route calculation. It is possible to create an advanced version of Dijkstra's Algorithm in this project to select searching area in finding network regions among the city and store processed topological relationships by deleting some unnecessary intersections.



### 3.0 Study Area Description



Study area in this project is the city of Vancouver. It contains 17013 public streets, 6092 street intersections, 100851 property addresses, 264 parks, 194 schools and 21 libraries and all of these data mentioned are analyzed in the project. For other data such as non city streets, lines, city projects sites, traffic signals and so on are added to consist of the final map but these data are not processed.

### 4.0 Description of Datasets & Data Preprocessing

This section will outline the table listing of input data sets and attributes of data

set used in analysis, data collecting, development of geodatabase, data checking and preprocessing.

#### 4.1 Listing Table of Input Data

Name of data	Type of data
property addresses	shape file
public streets	shape file
street intersections	shape file
park polygons	shape file
schools	shape file
libraries	shape file
non city streets	shape file
lanes	shape file
rapid transit stations	shape file
rapid transit line	shape file
property parcel polygons	shape file
traffic_signal	shape file
city projects sites	shape file
city boundary	shape file
property cadastral boundaries	shape file
picture of vancouver	jpeg format image

## 4.2 Listing Table of Data for Spatial Analysis

Name of data	Type of data
property addresses	shape file
public streets	shape file
street intersections	shape file
park polygons	shape file
schools	shape file
libraries	shape file

## 4.3 Listing Table of Attributes for Spatial Analysis

Name of data	Name of attributes used in analysis
property addresses	FID, Shape*, STREETNAME
public streets	FID, Shape*, Shape_Leng
street intersections	FID, Shape*, XSTREET
park polygons	FID, Shape*, PARK_NAME
schools	FID, Shape*, NAME
libraries	FID, Shape*, NAME

## 4.4 Data Collecting and Geodatabase Development

All data mentioned are downloaded directly from the website established by the government of Vancouver. Once these data are ready to use, a new layer is created in ArcMap to project these data. Then a file geodatabase is created to store selected

shape files.

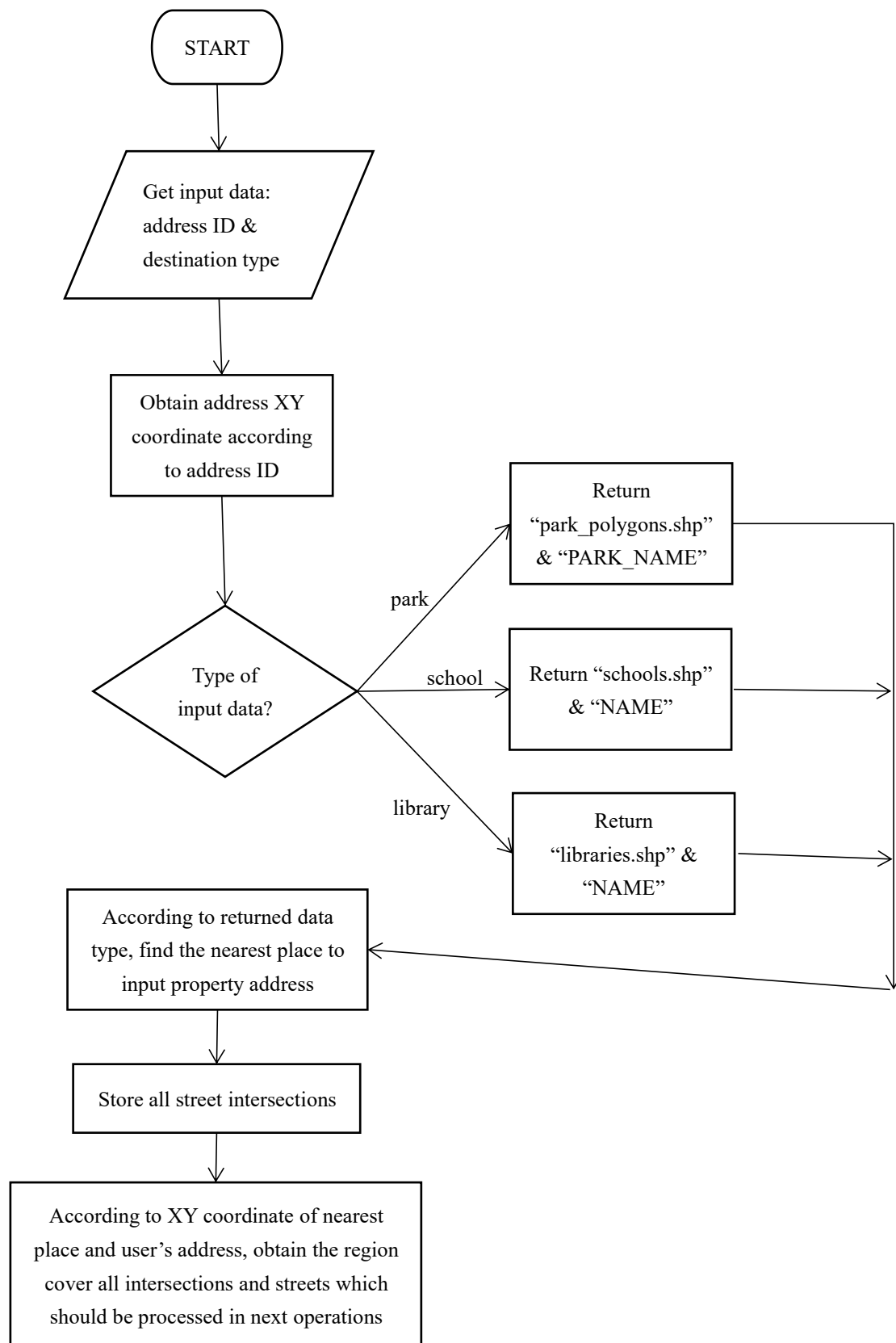
## **4.5 Data Checking and Preprocessing**

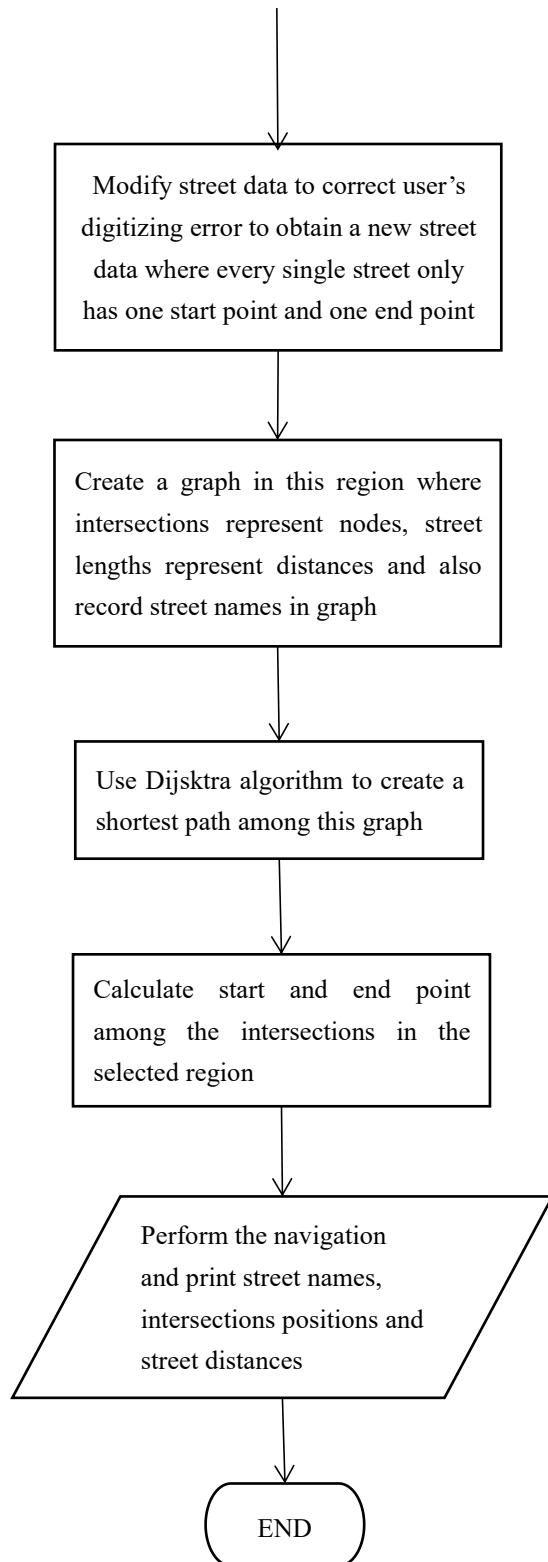
After finishing building map, checking attribute data of these shape files makes sure data ready to use are reliable. For error and blank in data attribute, the best way is to find information online then fill and correct them. After checking data attribute, it shows data are almost good except for one problem that the designer got some errors when digitizing public\_street shape file. Some of streets are separated in this shape file which means some streets may contain more than two points (two points refer to start point and end point consisted of a whole street). As for data preprocessing, it is necessary to join these separated streets with the same street name into a whole street then store these street information into one list for next operations. What has been solved is to store all streets with street name and XY coordinates into a list first, then process these streets with same street names, join XY coordinates of these streets having the same street name together to update a new street connecting separated information and then delete repeating elements.

## **5.0 Methodology**

This section covers a flowchart of solution, an introduction to created functions, a detailed description including seven steps of spatial analysis methods and explanations of core python scripts.

## 5.1 Flowchart of Solution





## 5.2 List of Created Functions

Name of function	Usage
dis(a,b)	calculate the distance between two points a & b
getposition_citizen(id_num)	get XY coordinate according to input address ID
destination_info(type)	return destination shape file and attribute name according to input type
location_distance_calc(c_x,c_y,fc,a_name)	get the shortest place XY coordinate and distance
store_street_intersections()	store all intersections into a list
get_max_and_min(x,y,park_x,park_y)	get the region four vertexs
get_region(start_x,end_x,start_y,end_y,storedata)	get the region of intersections
get_street(start_x,end_x,start_y,end_y,storedata)	get the region of streets
create_gragh(region,region_num,street)	greate the graph in the region
dijkstra(graph, initial, end)	perform dijsktra algorithm to output the shortest route
shortest(region,x,y,l_x,l_y)	assign start and end point
navigation(path,region,street,type,l_name)	print navigation information

## 5.3 Detailed Description of Methods

### *Step 1: Store street intersections*

Street intersections are consisted of the whole city network so the first thing is to store intersections attributes from shape file into an array. A function is created and the python scripts is below:

```
def store_street_intersections():  
  
    data = []  
  
    for i in xrange(6970):  
  
        data.append([])  
  
        for j in xrange(3):  
  
            data[i].append([])  
  
    fc = "street_intersections.shp"  
  
    cursor=arcpy.da.SearchCursor(fc,["FID","SHAPE@XY"])  
  
    for row in cursor:  
  
        num = row[0]  
  
        x,y = row[1]  
  
        data[num][0] = num  
  
        data[num][1] = x  
  
        data[num][2] = y  
  
    del row  
  
    del cursor  
  
    return data
```



A three dimensional list called 'data' has been created containing three attributes which are feature ID of intersections and their X Y coordinates separately. From the for loop, the range is 6970 which means there are 6970 intersections in Vancouver and these intersections information have been stored in list 'data'. For spatial data analysis, it is not wise to process all intersections because it may cost a long time. Next section will illustrate how to select specific region containing few intersections according to users input data to reduce unnecessary process, the selected region of streets and details of operations.

### *Step 2: Collect necessary spatial data from input*

According to input user address ID, it is efficient to use a search cursor in address shape file and easily get XY coordinate from FID. A part of property\_addresses attribute table and function are below:

property_addresses						
	FID	Shape *	CIVIC_NO	STREETNAME	TAX_COORD	SITE_ID
▶	0	Point	2150	MAPLE ST	09564266	015197778
	1	Point	2828	W 33RD AV	73006066	029209714
	2	Point	2766	W 38TH AV	74206066	029189900
	3	Point	1221	BIDWELL ST	61211098	028587413
	4	Point	8555	GRANVILLE ST	13082663	028800273
	5	Point	1735	W 4TH AV	64012065	029191360
	6	Point	5906	MAIN ST	19075512	029199450
	7	Point	1099	RICHARDS ST	13860695	029182115
	8	Point	1192	W 59TH AV	81314516	029012899

(Figure 1.1: a part of attribute table of property\_addresses)

```
def destination_info(type):
```

```
    if type == "park":
```

```
        return ("park_polygons.shp","PARK_NAME")
```

```
    if type == "library":
```

```
return ("libraries.shp","NAME")
```

```
if type == "school":
```

```
return ("schools.shp","NAME")
```

Due to the name domain in park is different from other two so we must use ‘if’ to select feature names. After this process, it can get the nearest place to user address. For example, if the user want to go to nearest park, for every park in our map, it contains its own XY coordinate. A part of of park\_polygons attribute table is below:

FID	Shape *	PARK_NAME	PARK_ID	
15	Polygon	Arbutus Village Park	1	<a href="http://covapp.vancouver.ca/parkfinder">http://covapp.vancouver.ca/parkfinder</a>
16	Polygon	Art Phillips Park	19	<a href="http://covapp.vancouver.ca/parkfinder">http://covapp.vancouver.ca/parkfinder</a>
17	Polygon	Ash Park	122	<a href="http://covapp.vancouver.ca/parkfinder">http://covapp.vancouver.ca/parkfinder</a>
18	Polygon	Balaclava Park	34	<a href="http://covapp.vancouver.ca/parkfinder">http://covapp.vancouver.ca/parkfinder</a>
19	Polygon	Barclay Heritage Square Park	200	<a href="http://covapp.vancouver.ca/parkfinder">http://covapp.vancouver.ca/parkfinder</a>
20	Polygon	Barclay Heritage Square Park	200	<a href="http://covapp.vancouver.ca/parkfinder">http://covapp.vancouver.ca/parkfinder</a>
21	Polygon	Barclay Heritage Square Park	200	<a href="http://covapp.vancouver.ca/parkfinder">http://covapp.vancouver.ca/parkfinder</a>
22	Polygon	Bates Park	66	<a href="http://covapp.vancouver.ca/parkfinder">http://covapp.vancouver.ca/parkfinder</a>
23	Polygon	Bates Park	66	<a href="http://covapp.vancouver.ca/parkfinder">http://covapp.vancouver.ca/parkfinder</a>
24	Polygon	Beaconsfield Park	148	<a href="http://covapp.vancouver.ca/parkfinder">http://covapp.vancouver.ca/parkfinder</a>
25	Polygon	Bobolink Park	191	<a href="http://covapp.vancouver.ca/parkfinder">http://covapp.vancouver.ca/parkfinder</a>
26	Polygon	Braemar Park	174	<a href="http://covapp.vancouver.ca/parkfinder">http://covapp.vancouver.ca/parkfinder</a>
27	Polygon	Brewers Park	79	<a href="http://covapp.vancouver.ca/parkfinder">http://covapp.vancouver.ca/parkfinder</a>
28	Polygon	Burrard View Park	67	<a href="http://covapp.vancouver.ca/parkfinder">http://covapp.vancouver.ca/parkfinder</a>

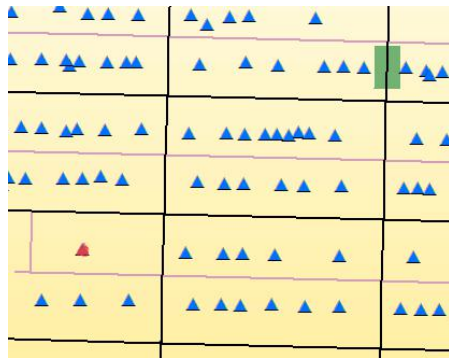
(Figure 1.2: a part of attribute table of park\_polygons)

Then a search cursor can be used for every single park to calculate the absolute distance  $\sqrt{(x_1 - x_2)^2 + (y_1 - y_2)^2}$  between every park (x1,x2) and user’s home (x2,y2) and then output the nearest one’s XY coordinate and name. For school and library, there are the same as park. So far, we have start point XY coordinate and destination XY coordinate with names.

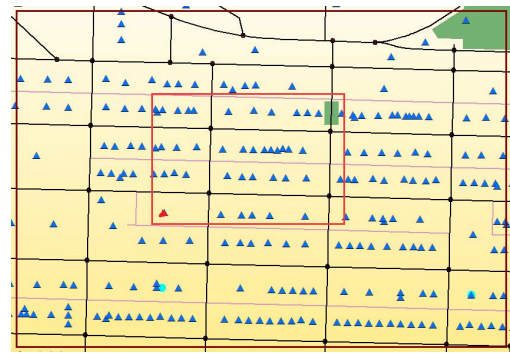
### *Step 3: Select region which should be analyzed from the map*

This step is to avoid unnecessary processes because it need not always compute all intersections and streets data in Vancouver which may causes few minutes even

few hours. Therefore, a selected region should be assigned. For example, there are start point which is the red triangle and end point which is green rectangle in Figure 3. The original region is light red rectangle and processed region is dark rectangle after adding boundary for original region to make sure all intersections which may affect route calculation are covered. The minimum distance for boundary is 600 meters which has been tested for this number covering routes between all addresses and selected places.



(Figure 2.1: start point and end point)



(Figure 2.2: selected region)

The next process is to store all intersections in this region into one list and all streets in this region into another list. Because there already exists a list containing all 6970 intersections in data preprocessing, so this process just need to clip some of them from the whole intersections data set into a new list and the function below can do it:

```
def get_region(start_x,end_x,start_y,end_y,storedata):
```

```
    data = []
```

```
    t = 0
```

```
    for i in xrange(6970):
```

```
        x = storedata[i][1]
```

```
y = storedata[i][2]

if x>start_x and x<end_x and y>start_y and y<end_y:

    data.append([])

    for j in xrange(3):

        data[t].append([])

        data[t][0] = i

        data[t][1] = x

        data[t][2] = y

        t+=1

    return data,t

region,region_num=get_region(min_x-600,max_x+600,min_y-600,max_y+600,
storedata)
```

In this function, start\_x, end\_x, start\_y, end\_y are two pair of XY coordinate systems and the boundary distance is set to 600 as input parameters when using this function. Finally, it can output a region list containing all intersections need to be processed in next step.

When it comes to streets storing, it needs some operations to solve maker's error in public street shape file. The purpose is to connect separated streets into a whole street. Function get\_street below can both store streets data and process separated streets connections.

```
def get_street(start_x,end_x,start_y,end_y,storedata):
```

```
    data = []
```

```
t = 0

fc = "public_streets.shp"

cursor = arcpy.da.SearchCursor(fc,["SHAPE@", "SHAPE@XY", "HBLOCK"])

for row in cursor:

    x,y = row[1]

    if x>start_x and x<end_x and y>start_y and y<end_y:

        s = 0

        data.append([])

        data[t].append([])

        data[t][s] = row[2]

        s+=1

        for point in row[0].getPart(0):

            data[t].append([])

            data[t][s] = point.X,point.Y

            s+=1

        t+=1

for i in xrange(len(data)-1):

    for j in xrange(i+1,len(data)):

        if data[i][0] == data[j][0]:

            add = data[j][1:len(data[j])]

            for k in xrange(len(add)):

                data[i].append(add[k])
```

```
data[j][0] = "del"

newdata = []

for list in data:

    if list[0] != "del":

        newdata.append(list)

return newdata,len(newdata)
```

In this function, two elements streets name and XY coordinate are stored in street data list. Last two for loops are to detect if two different streets have the same name and then append all points in one street to another street and delete this ‘subset’ street later.

#### *Step 4: Create graph*

For every intersection in the region, distances between every single intersection need to be created and then output the graph which is a three-dimensional list containing one start node, one end node and the distance between them. Function create\_graph can yield the graph by input street intersections and streets in the selected region.

```
def create_graph(region,region_num,street):

    data = []

    t = 0

    for i in xrange(region_num-1):
```

```
for j in xrange(i+1,region_num):

    for list in street:

        if (region[i][1],region[i][2]) in list and (region[j][1],region[j][2]) in
list:

            data.append([])

            for k in xrange(3):

                data[t].append([])

                data[t][0] = region[i][0]

                data[t][1] = region[j][0]

                data[t][2]=dis(region[i][1]-region[j][1],region[i][2]-region[j][2])

            t+=1

return data
```

It needs to make sure two intersections are in one street then there will be a route between them and store this route into graph. `region[i][1]` and `region[i][2]` are XY coordinate of intersections. The first two loops are to detect every possible path between arbitrary two points in the region. `data[t][0]` and `data[t][1]` are start node, end node and `data[t][2]` is the distance between them.

#### *Step 5: Use Dijkstra algorithm to output the shortest route*

In this section, scripts are from open source code online and have been some changes to be adapt to the whole project. In `create_graph` function, the return data fromat has changed to `[node1, node2, distance]` to match input format in dijkstra

algorithm function. The result of this function is a shortest route containing all passing nodes which is a list consisted of different pairs of intersection to intersection.

*Step 6: Print navigation output*

After getting the shortest path, it can print navigation information for users.

Function navigation can show street names, street lengths and intersection positions.

```
def navigation(path,region,street,type,l_name):

    arcpy.AddMessage ("The nearest {0} near your address is:
{1}".format(type,l_name))

    arcpy.AddMessage ("Route calculation:")

    origin = []

    for count in xrange(len(path)-1):

        num1 = path[count]

        num2 = path[count+1]

        for list in region:

            if list[0] == num1:

                x1 = list[1]

                y1 = list[2]

            if list[0] == num2:

                x2 = list[1]

                y2 = list[2]
```



```
for list in street:

    if (x1,y1) in list and (x2,y2) in list:

        mark = list[0]

        distance = int(round(dis (x1-x2,y1-y2)))

    turn = re.split(' ',mark)

    turn.pop(0)

    if count == 0:

        origin = turn

    if turn != origin:

        arcpy.AddMessage ("Make a turn at the intersection")

        origin = turn

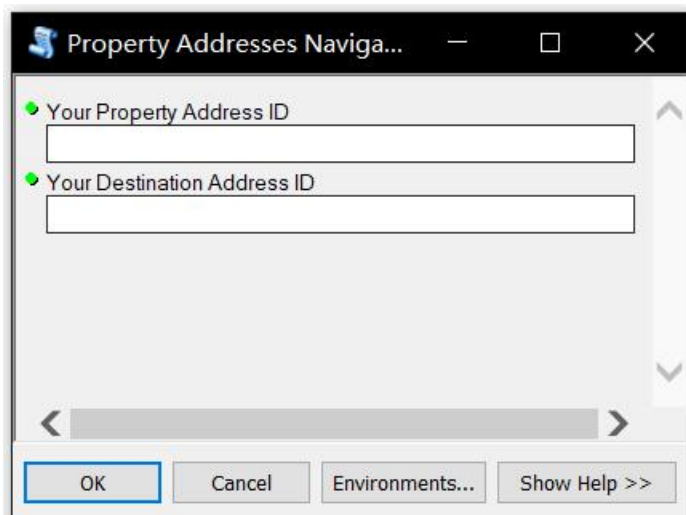
    arcpy.AddMessage ("Go straight along {0}: {1} {2}s".format(mark, distance,
units))

arcpy.AddMessage ("You have reached your destination, have a nice day!")
```

Due to the shortest path output does not contain street names only has intersections ID, the first two loops are to find XY coordinate of two intersections according to their ID. Then the next loop is to find the street which contains these two intersections. Then it can print street name and the distance how long users should go. In Canada, straight roads are classified into roads with same name without different numbers in front of them. So parameters 'turn' and 'origin' in this function are used to mark if the main street name has changed which means there is no longer straight road in there and then it can remind users may make a turn at the intersection.

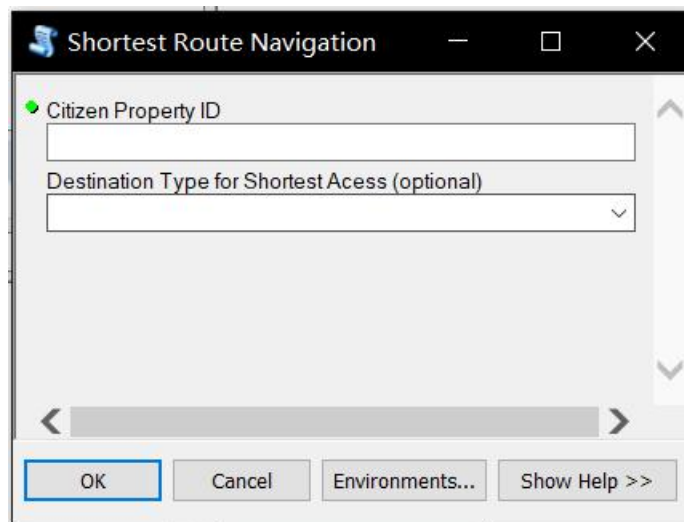
*Step 7: Create toolbox*

Based on this script, there are some changes have been added and the project finally yields four different tools for users: 1. property addresses navigation which can provide navigation information from one property address to another address; 2. shortest route navigation which is from one property address to nearest park, school or library; 3. specified destination navigation gives users a path from property address to specific park, school or library; 4. two given places navigation offers a shortest route from one place to another place among parks, libraries and schools. For various input data types, four interfaces are different and they are shown below:



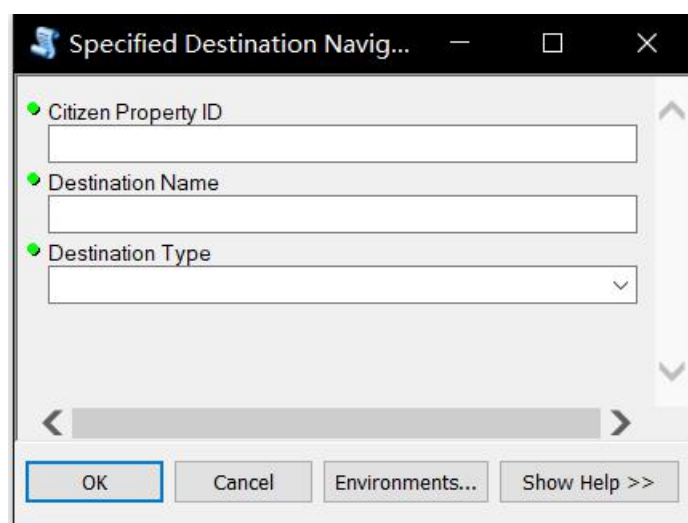
**(Figure 3.1: property addresses navigation)**

Input are two addresses IDs and output are shortest route navigation information between these two input addresses.



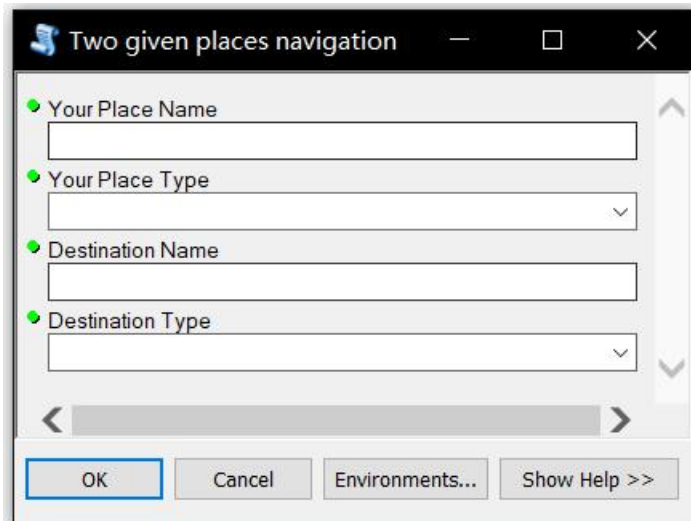
(Figure 3.2: shortest route navigation)

Input are address ID and destination type(park, school or library), output are shortest route navigation information to the nearest place.



(Figure 3.3: specified destination navigation)

Input are address ID, destination name and destination type(park, school or library), output are shortest route navigation information to this place.



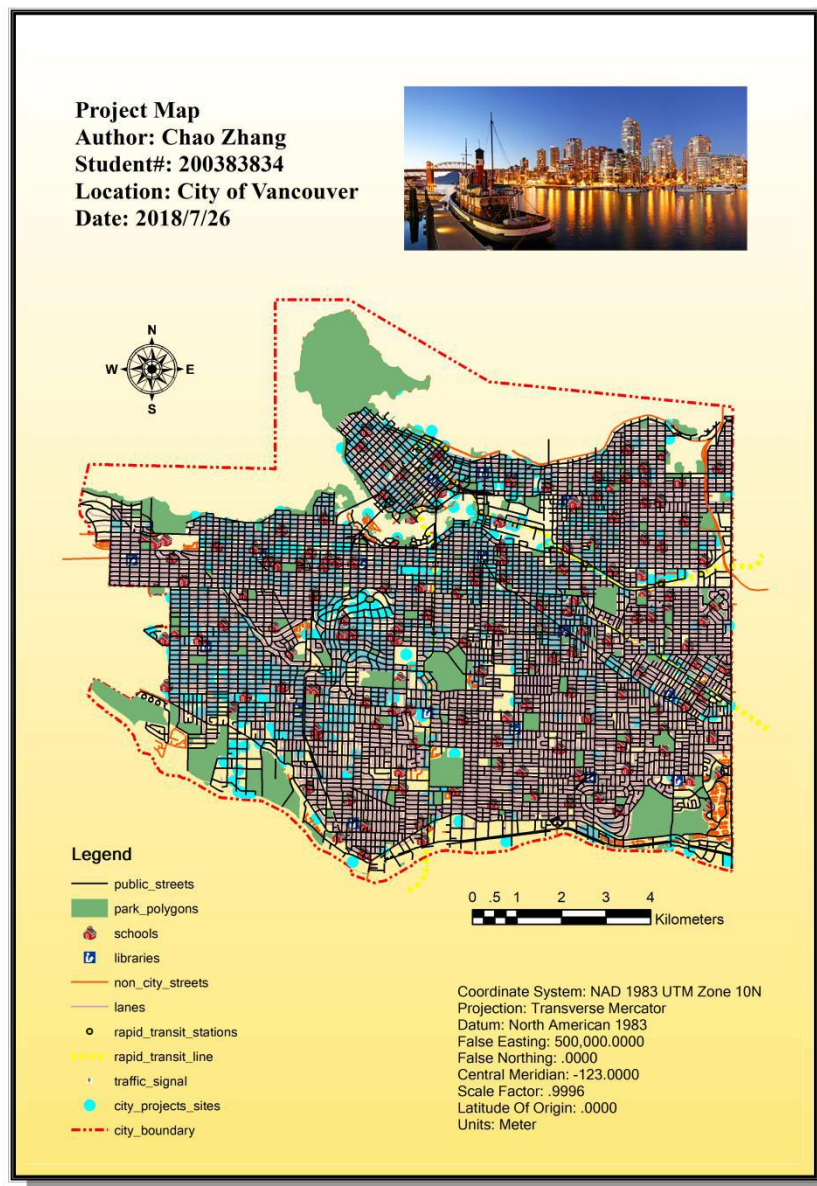
(Figure 3.4: two given places navigation)

Input are two places' IDs and types and places can be any one in parks, schools or libraries, output are shortest route navigation information between these two places.

## 6.0 Results and Discussion

This section will illustrate two parts: the map created by ArcMap and results of navigation information for four different tools.

## 6.1 Created Map



(Figure 4: created map)

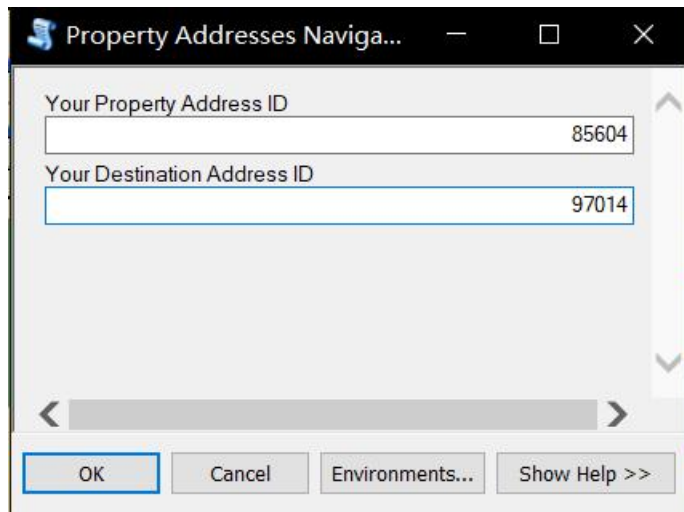
The created map contains some of shape files such as public streets, park polygons, schools, libraries and so on in order to illustrate the view of whole city. It also includes some other data like rapid transit line with its stations because it plays an important role in city traffic though they have not been analyzed. For street intersections and property addresses, they are not included in the map because the

large number of them may cause the whole map unclear to see and it is unnecessary for users to see these data but they are mainly processed by python scripts to create the navigation toolbox.

## 6.2 Results of Navigation Information

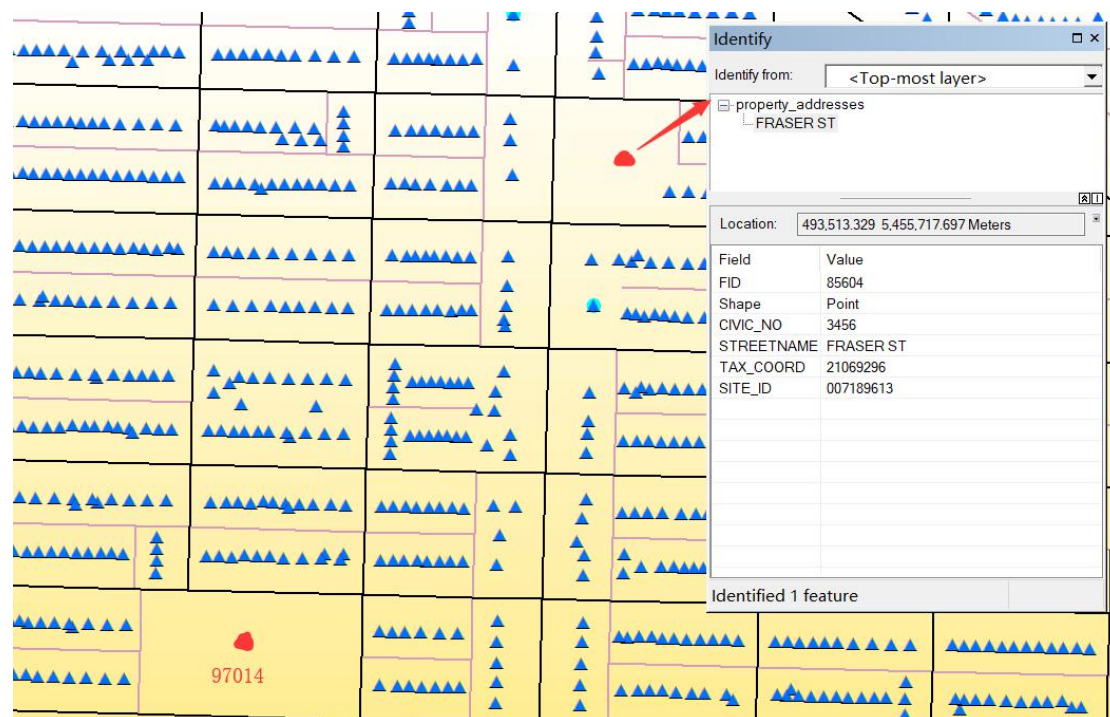
*Test 1: property address navigation*

Input:



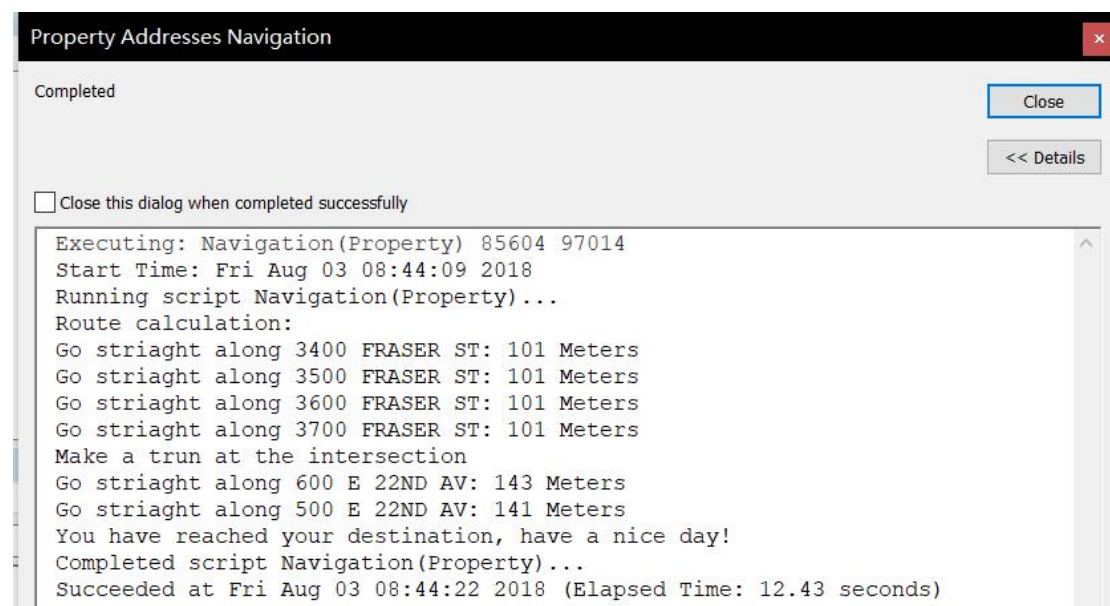
(Figure 5.1.1: Test 1 input)

Input are two property addresses ID, from 85604 to 971014 and these two addresses are red marks in the map where blue triangles are represent for other property addresses.



(Figure 5.1.2: Test 1 input on map)

Output:



(Figure 5.1.3: Test 1 output information)





(Figure 5.1.4: Test 1 output route)

*Test 2: property address navigation*

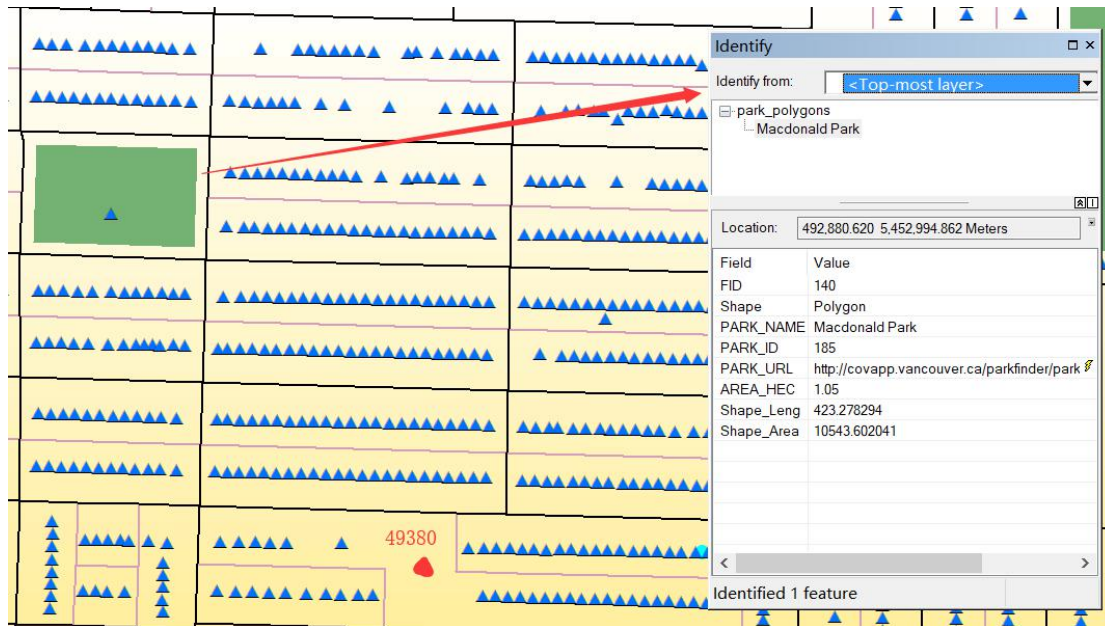
Input:

(Figure 5.2.1: Test 2 input)

Input are property ID and destination type, from 49380 address to nearest park.

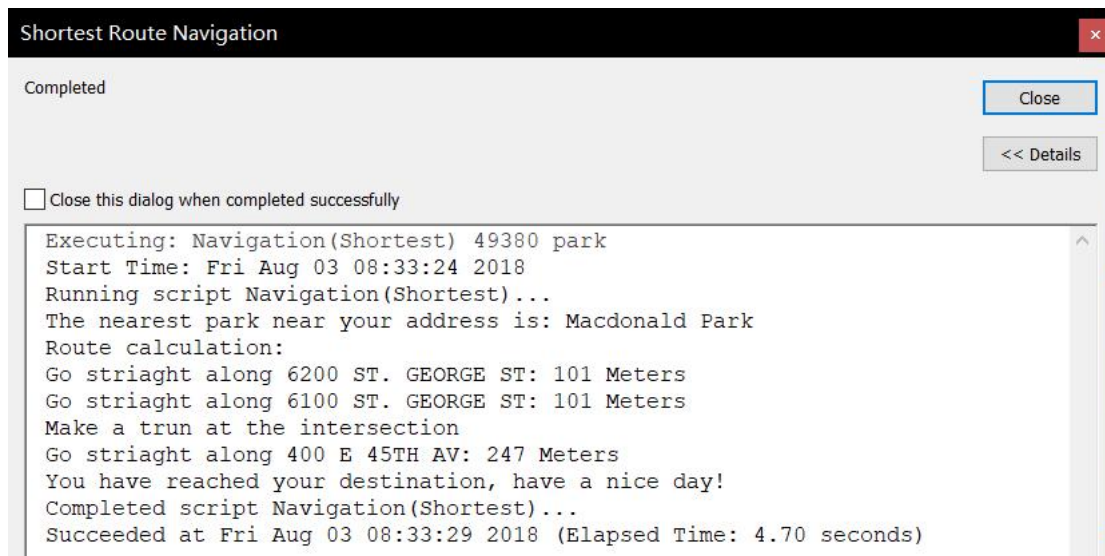
From the map, nearest park obviously is Macdonald Park.



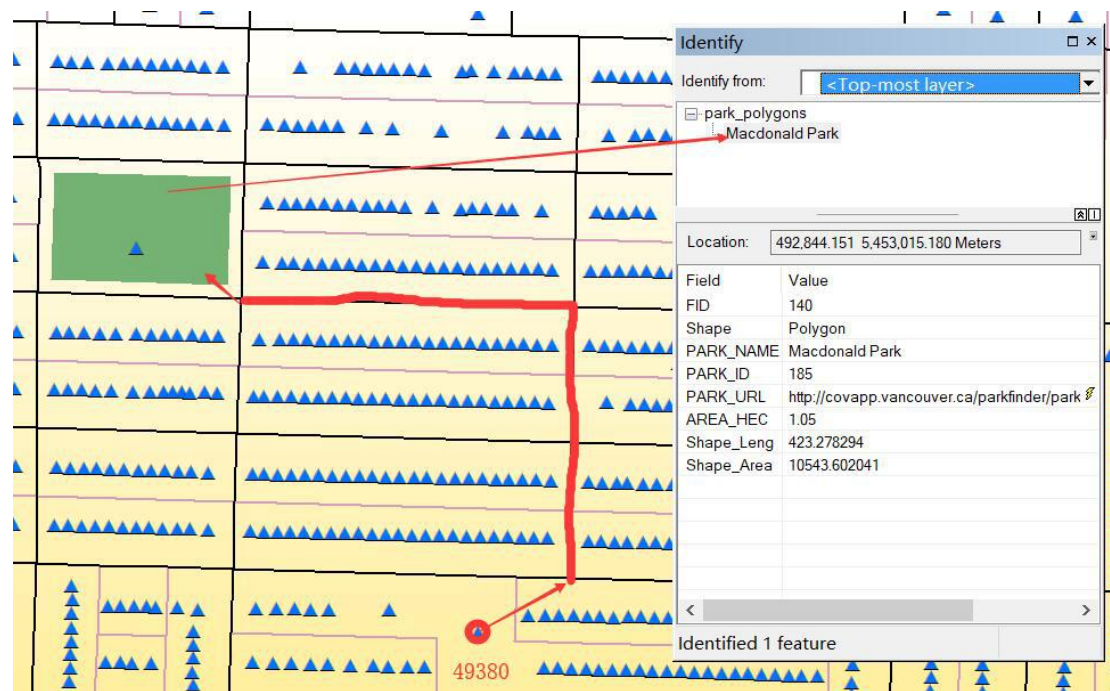


(Figure 5.2.2: Test 2 input on map)

Output:



(Figure 5.2.3: Test 2 output information)



(Figure 5.2.4: Test 2 output route)

### *Test 3: specified destination navigation*

Input:

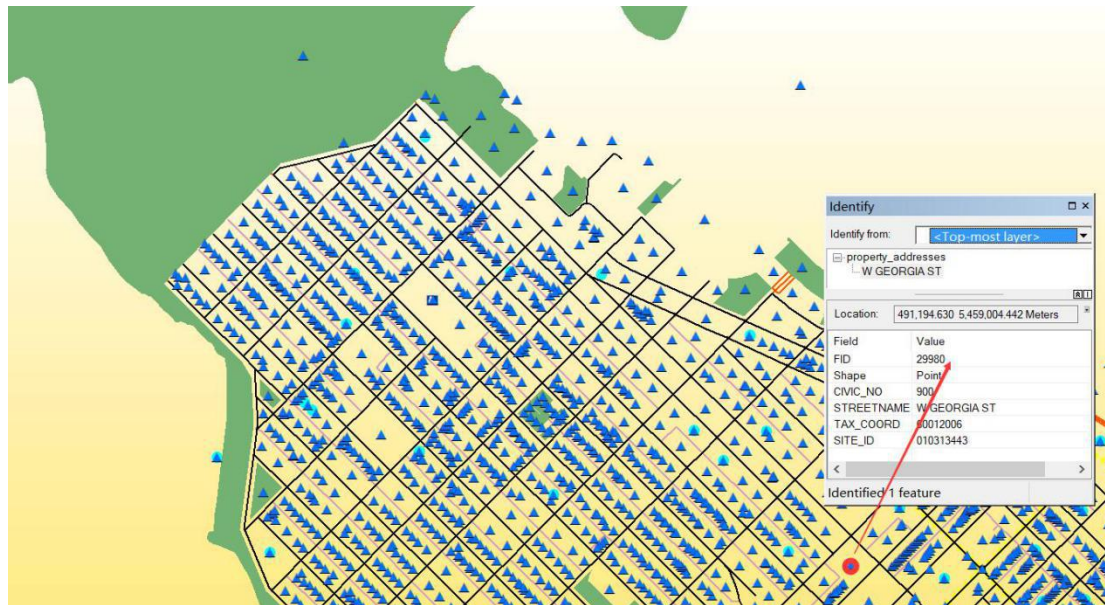
The screenshot shows a dialog box titled 'Specified Destination Navig...'. It contains the following input fields:

- Citizen Property ID: 29980
- Destination Name: Stanley Park
- Destination Type: park (selected from a dropdown menu)

At the bottom, there are four buttons: OK, Cancel, Environments..., and Show Help >>.

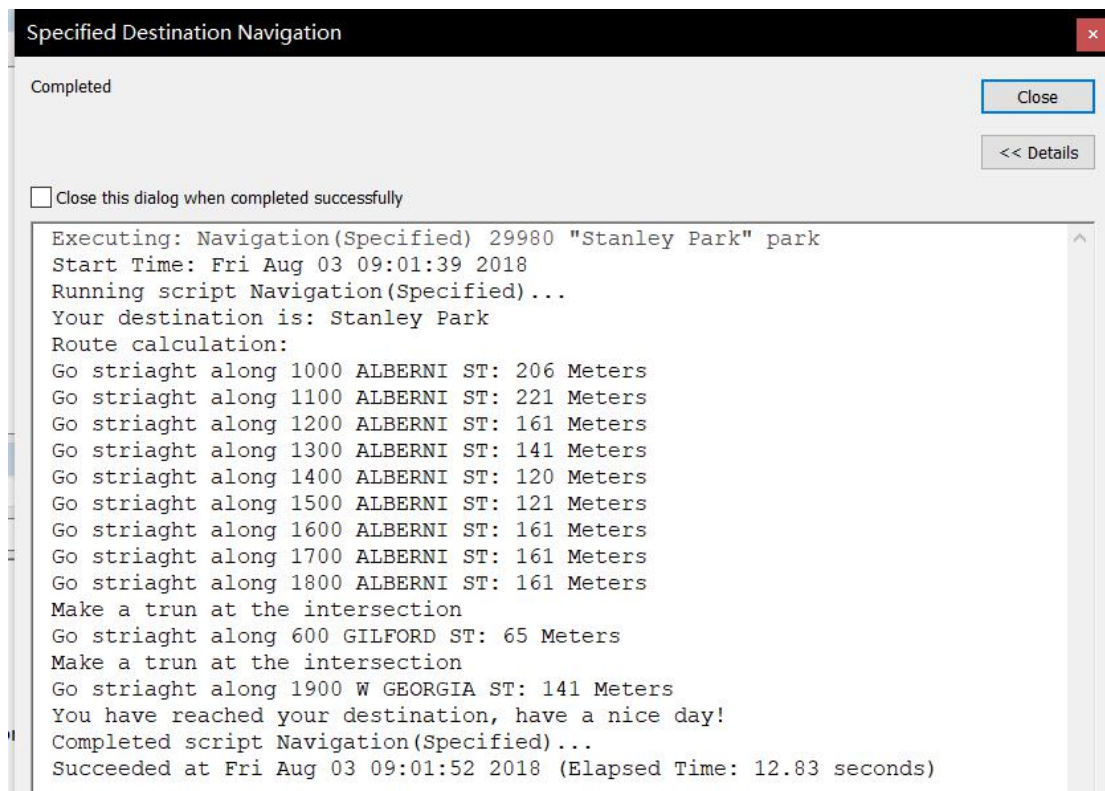
(Figure 5.3.1: Test 2 input)

Input are property address ID which is 29980 and the destination is Stanley Park.



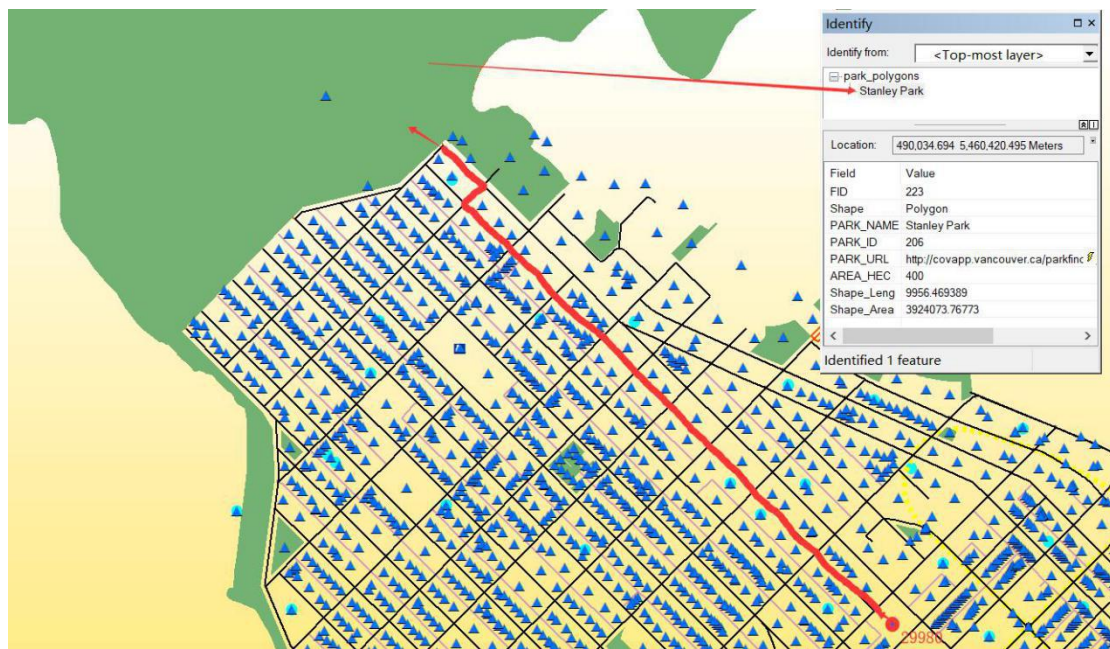
(Figure 5.3.2: Test 3 input on map)

Output:



(Figure 5.3.3: Test 3 input information)





(Figure 5.3.4: Test 3 output route)

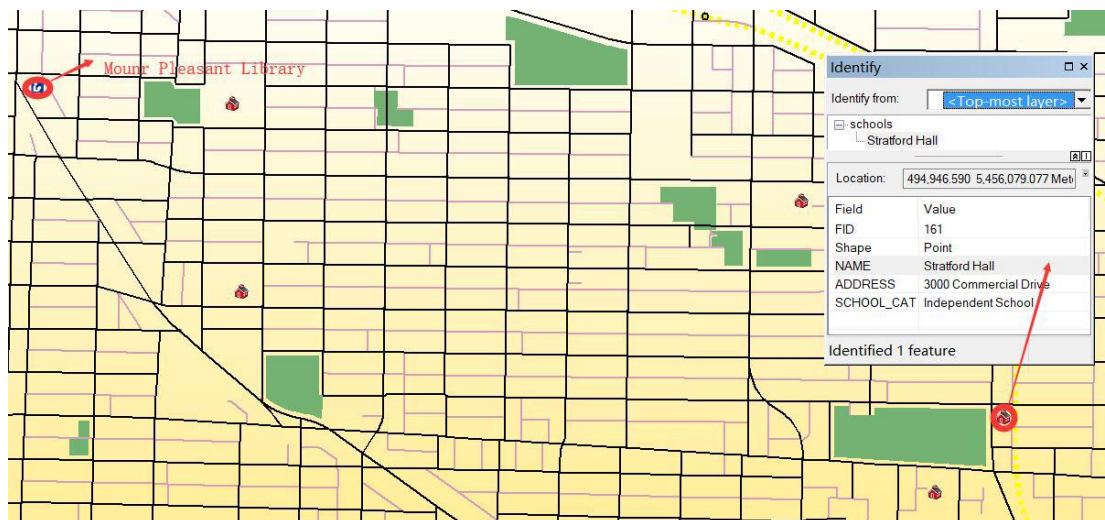
#### *Test 4: two given places navigation*

Input:

The screenshot shows a dialog box titled 'Two given places navigation'. It contains four input fields: 'Your Place Name' with the value 'Mount Pleasant', 'Your Place Type' with a dropdown menu showing 'library', 'Destination Name' with the value 'Stratford Hall', and 'Destination Type' with a dropdown menu showing 'school'. At the bottom, there are four buttons: 'OK', 'Cancel', 'Environments...', and 'Show Help >>'.

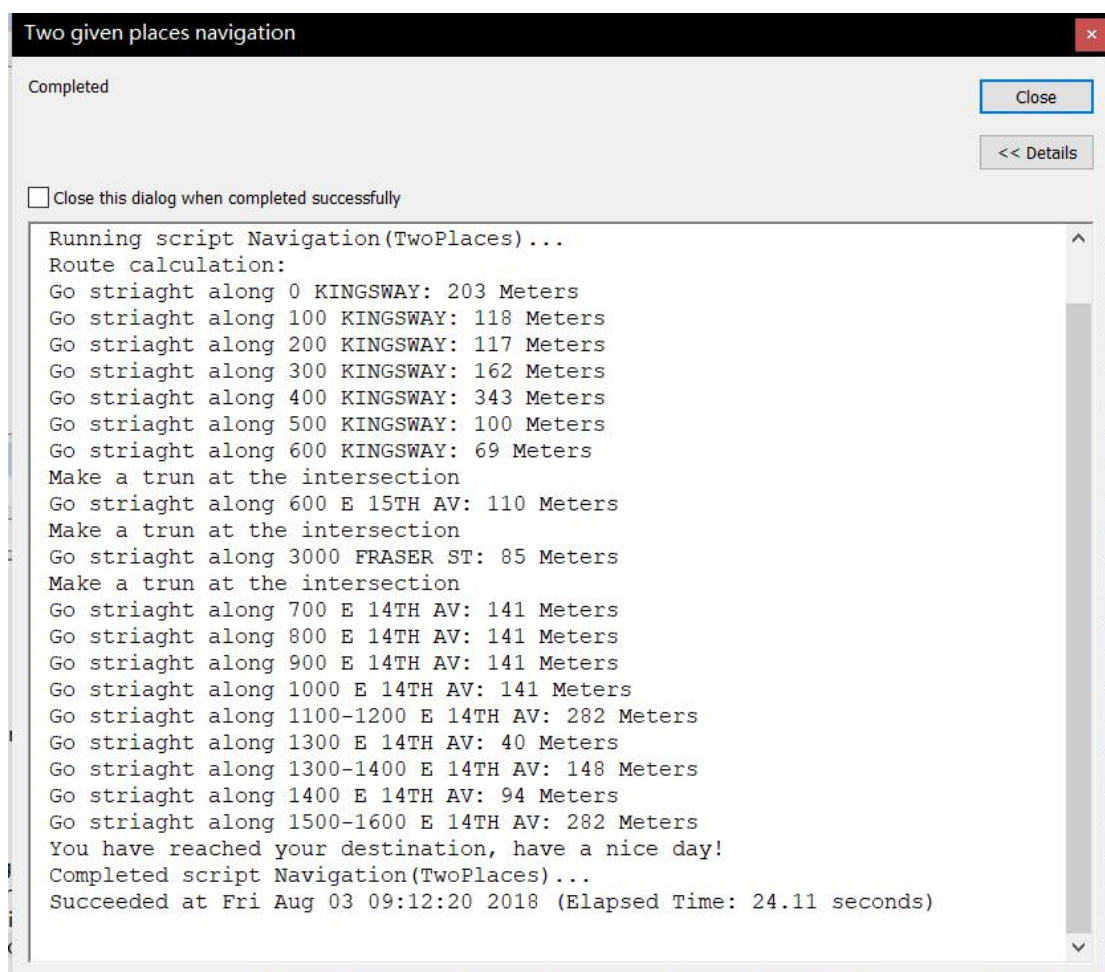
(Figure 5.4.1: Test 4 input)

Input are Mount Pleasant Library and Stratford Hall School which are the start and end places.

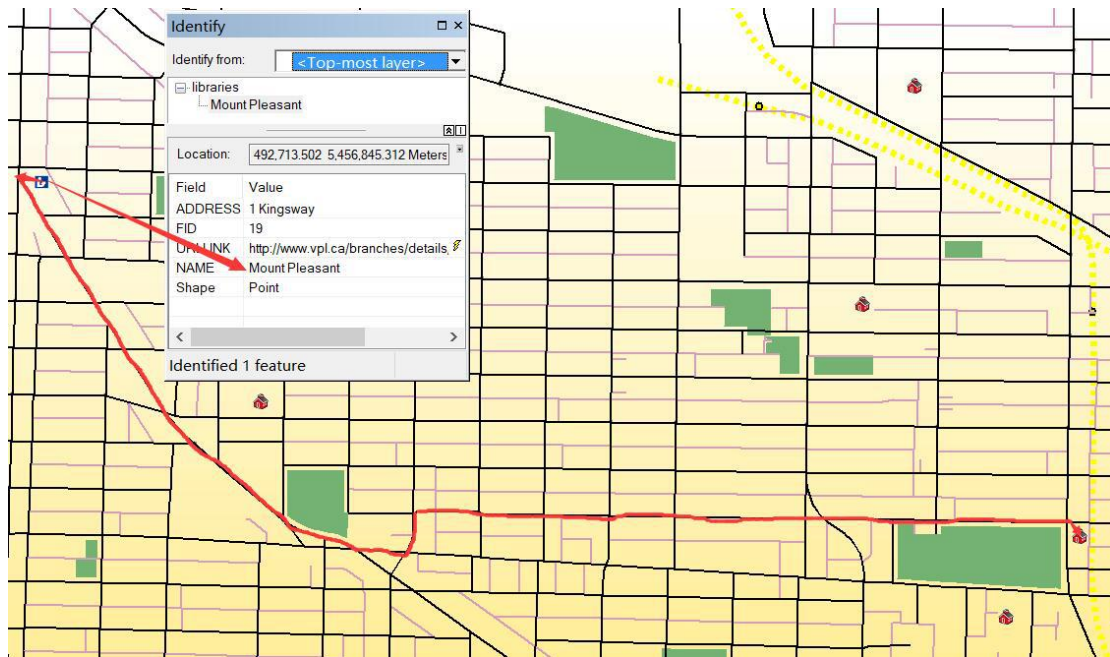


(Figure 5.4.2: Test 4 input on map)

Output:



(Figure 5.4.3: Test 4 output information)



(Figure 5.4.4: Test 4 output route)

## 7.0 Assumptions and Limitations

This toolbox is created to handle all navigation among Vancouver so it may take a long time for large areas route calculation. Due to the core mind is to make a graph among street intersections, once the covered area is bigger, it needs more time to deal with the created graph. A new method is to upgrade the shortest route calculation called arc-flag approach which is based on Dijkstra's algorithm to accelerate the route calculation. It contains a preprocessing of data in graph to divide into different regions and gather whether an arc is on a shortest path into a given region(Möhring, 2007). So the combination of appropriate partitioning and two direction search can make route calculation more efficient. In addition, the toolbox only can provide shortest path for drivers because it is based on public streets. For users who walk, skate or cycle, the process area should include non public streets and lanes and more than one graph

should be generated with various decisions between different types of roads.

## **8.0 Conclusion and Future Work**

The project provides route navigation from property addresses to nearest or specific schools, libraries and parks and also between two arbitrary given places among city of Vancouver. From attributes in different shape files, spatial data like XY coordinates of objects has been found to analyse and yield the toolbox. Also, the relationship between attributes in every shape file should be generated from object ID to its other attributes for getting access to all the information. From this project, I have gained knowledge in searching GIS data online, creating maps by ArcGIS, spatial data analysis, skills of Python Scripts programming especially in debugging public streets data sets where I found a digitizing error and it makes me understand GIS spatial data more clearly. In the future, I hope I could improve this navigation toolbox by adding lanes and non public street shape files to generate an advanced version of navigation toolbox both for drivers and pedestrians.

## References:

- Dijkstra, E. W. (1959). A note on two problems in connexion with graphs. *Numerische mathematik*, 1(1), 269-271.
- Fan, D., & Shi, P. (2010, August). Improvement of Dijkstra's algorithm and its application in route planning. In *Fuzzy Systems and Knowledge Discovery (FSKD), 2010 Seventh International Conference on* (Vol. 4, pp. 1901-1904). IEEE.
- Jasika, N., Alispahic, N., Elma, A., Ilvana, K., Elma, L., & Nosovic, N. (2012, May). Dijkstra's shortest path algorithm serial and parallel execution performance analysis. In *MIPRO, 2012 proceedings of the 35th international convention* (pp. 1811-1815). IEEE.
- Möhring, R. H., Schilling, H., Schütz, B., Wagner, D., & Willhalm, T. (2007). Partitioning graphs to speedup Dijkstra's algorithm. *Journal of Experimental Algorithmics (JEA)*, 11, 2-8.



## Appendix

### *Source Code of Python Scripts:*

```
import arcpy

import math

import re

from arcpy import env

from collections import defaultdict

env.overwriteOutput = True

env.workspace = "C:/Users/CHAO/Desktop/GIS_project/data/"

units = arcpy.Describe("park_polygons.shp").spatialReference.linearUnitName

input_id = arcpy.GetParameter(0)

input_type = arcpy.GetParameter(1)


def dis(a,b):

    result = math.sqrt(a*a + b*b)

    return result


def getposition_citizen(id_num):

    fc = "property_addresses.shp"

    cursor = arcpy.da.SearchCursor(fc,["SHAPE@XY","FID"])

    for row in cursor:

        if row[1] == id_num:
```

```
        x,y = row[0]

    del row

    del cursor

    return x,y


def destination_info(type):

    if type == "park":

        return ("park_polygons.shp","PARK_NAME")

    if type == "library":

        return ("libraries.shp","NAME")

    if type == "school":

        return ("schools.shp","NAME")


def location_distance_calc(c_x,c_y,fc,a_name):

    cursor = arcpy.da.SearchCursor(fc,["SHAPE@XY",a_name])

    s = 99**9

    for row in cursor:

        x,y = row[0]

        distance = dis(x-c_x,y-c_y)

        if distance<s:

            s = distance

            name = row[1]
```

```
l_x = x
```

```
l_y = y
```

```
del row
```

```
del cursor
```

```
return name,l_x,l_y
```

```
def store_street_intersections():
```

```
    data = []
```

```
    for i in xrange(6970):
```

```
        data.append([])
```

```
        for j in xrange(3):
```

```
            data[i].append([])
```

```
    fc = "street_intersections.shp"
```

```
    cursor = arcpy.da.SearchCursor(fc,["FID","SHAPE@XY","XSTREET"])
```

```
    for row in cursor:
```

```
        num = row[0]
```

```
        x,y = row[1]
```

```
        data[num][0] = num
```

```
        data[num][1] = x
```

```
        data[num][2] = y
```

```
    del row
```

```
    del cursor
```

```
return data
```

```
def get_max_and_min(x,y,park_x,park_y):
```

```
    if x>park_x:
```

```
        max_x = x
```

```
        min_x = park_x
```

```
    else:
```

```
        max_x = park_x
```

```
        min_x = x
```

```
    if y>park_y:
```

```
        max_y = y
```

```
        min_y = park_y
```

```
    else:
```

```
        max_y = park_y
```

```
        min_y = y
```

```
    return min_x,max_x,min_y,max_y
```

```
def get_region(start_x,end_x,start_y,end_y,storedata):
```

```
    data = []
```

```
    t = 0
```

```
    for i in xrange(6970):
```

```
        x = storedata[i][1]
```

```
y = storedata[i][2]

if x>start_x and x<end_x and y>start_y and y<end_y:

    data.append([])

    for j in xrange(3):

        data[t].append([])

        data[t][0] = i

        data[t][1] = x

        data[t][2] = y

        t+=1

return data,t


def get_street(start_x,end_x,start_y,end_y,storedata):

    data = []

    t = 0

    fc = "public_streets.shp"

    cursor = arcpy.da.SearchCursor(fc,["SHAPE@", "SHAPE@XY", "HBLOCK"])

    for row in cursor:

        x,y = row[1]

        if x>start_x and x<end_x and y>start_y and y<end_y:

            s = 0

            data.append([])

            data[t].append([])
```

```
data[t][s] = row[2]

s+=1

for point in row[0].getPart(0):

    data[t].append([])

    data[t][s] = point.X,point.Y

    s+=1

t+=1

for i in xrange(len(data)-1):

    for j in xrange(i+1,len(data)):

        if data[i][0] == data[j][0]:

            add = data[j][1:len(data[j])]

            for k in xrange(len(add)):

                data[i].append(add[k])

            data[j][0] = "del"

newdata = []

for list in data:

    if list[0] != "del":

        newdata.append(list)

return newdata,len(newdata)

def create_gragh(region,region_num,street):

    data = []
```

```
t = 0

for i in xrange(region_num-1):

    for j in xrange(i+1,region_num):

        for list in street:

            if (region[i][1],region[i][2]) in list and (region[j][1],region[j][2]) in

list:

                data.append([])

                for k in xrange(3):

                    data[t].append([])

                    data[t][0] = region[i][0]

                    data[t][1] = region[j][0]

                    data[t][2] = (region[i][1]-region[j][1],region[i][2]-region[j][2])

                    t+=1

        return data

def shortest(region,x,y,l_x,l_y):

    start = 99**9

    end = 99**9

    for list in region:

        num = list[0]

        xxx = list[1]

        yyy = list[2]
```

```
    if dis(x-xxx,y-yyy)<start:

        start = dis(x-xxx,y-yyy)

        start_num = num

    if dis(xxx-l_x,yyy-l_y)<end:

        end = dis(xxx-l_x,yyy-l_y)

        end_num = num

    return start_num,end_num


def navigation(path,region,street,type,l_name):

    arcpy.AddMessage ("The nearest {0} near your address is:

{1}".format(type,l_name))

    arcpy.AddMessage ("Route calculation:")

    origin = []

    for count in xrange(len(path)-1):

        num1 = path[count]

        num2 = path[count+1]

        for list in region:

            if list[0] == num1:

                x1 = list[1]

                y1 = list[2]

            if list[0] == num2:

                x2 = list[1]
```



```
y2 = list[2]

for list in street:

    if (x1,y1) in list and (x2,y2) in list:

        mark = list[0]

        distance = int(round(dis (x1-x2,y1-y2)))

        turn = re.split(' ',mark)

        turn.pop(0)

        if count == 0:

            origin = turn

        if turn != origin:

            arcpy.AddMessage ("Make a turn at the intersection")

            origin = turn

            arcpy.AddMessage      ("Go      straight      along      {0}:      {1}

{2}s".format(mark,distance,units))

            arcpy.AddMessage ("You have reached your destination, have a nice day!")

citizenID = input_id

destination_type = input_type

filename,attribute_name = destination_info(destination_type)

p_x,p_y = getposition_citizen(citizenID)

location_name,l_x,l_y = location_distance_calc(p_x,p_y,filename,attribute_name)

min_x,max_x,min_y,max_y = get_max_and_min(p_x,p_y,l_x,l_y)
```

```
storedata = store_street_intersections()
```

```
region,region_num =
```

```
get_region(min_x-600,max_x+600,min_y-600,max_y+600,storedata)
```

```
street,street_num =
```

```
get_street(min_x-600,max_x+600,min_y-600,max_y+600,storedata)
```

```
edges = create_graph(region,region_num,street)
```

```
graph = Graph()
```

```
for edge in edges:
```

```
    graph.add_edge(*edge)
```

```
start,end = shortest(region,p_x,p_y,l_x,l_y)
```

```
path = dijkstra(graph,start,end)
```

```
navigation(path,region,street,destination_type,location_name)
```

(dijkstra()) are reference source code and can be found in below link

<http://benalexkeen.com/implementing-djikstras-shortest-path-algorithm-with-python/>)