Project 2 – Topics in Data Science ECE 592

Name: Alhiet Orbegoso
ID: 200322491

For the development of the project it has been used MATLAB.

2. Data Retrieval and Pre-processing

A. Function Description

For the data retrieval, it has been created a function named ***data_retrieval***.

Inputs

- ***Filename:*** It is the name of the file containing the distances of NC cities. It is important to mention that the file provided by the link is in *.xlsx format. MATLAB processing time is large for *.xlsx format. To improve this, the file was saved as *.csv. Due to this, the file used for the project is in format *.csv.

- ***Upper_dist:*** It is the threshold in miles for the distance between cities. If the distance of any pair of cities is above the threshold, the edge connecting them is discarded.

Outputs:

- ***NC_city_names:*** It is a Nx1 vector containing the name of all the cities in NC. The value of N is the number of cities in the *.csv file. For this case N=444.

- ***NC_city_array:*** It is a NxN matrix containing the distances among all the cities in NC. The value of N is the number of cities in the *.csv file. For this case N=444. This array is also the result of applying the ***upper_dist*** threshold.

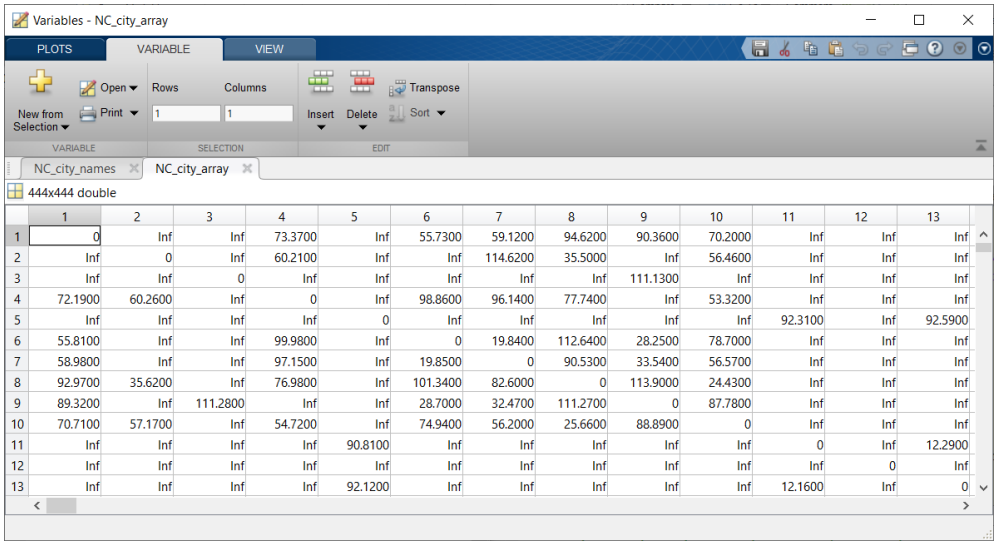
- ***NC_city_array_H:*** It is a copy of the ***NC_city_array*** output but without the ***upper_dist*** threshold. This matrix is used as a heuristic for the A* algorithm.

It is important to mention that rows and columns of outputs NC_city_names and NC_city_array are related. For example, the vector representing the distances of the city Asheboro (row 10 in NC_city_names) is located is the NC_city_array with the same row position (row 10). Moreover, in this vector each index represents the distance from Asheboro to all cities with the same index as in NC_city_names. In NC_city_array the

element in row=10 and column=8 represents the distance from Asheboro (row 10 in NC_city_names) to Archdale (row 8 in NC_city_names).

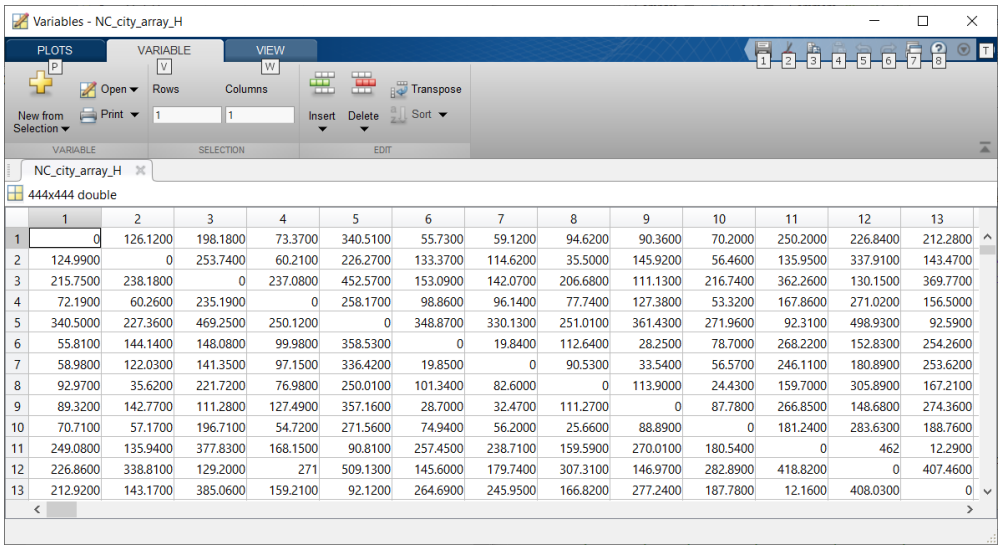
B. Results

Below is presented the outputs of the function with an upper limit of 120 miles.



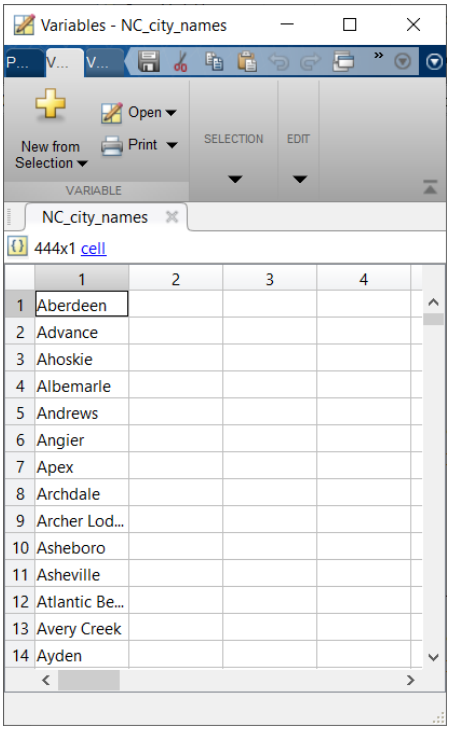
	1	2	3	4	5	6	7	8	9	10	11	12	13
1	0	Inf	Inf	73.3700	Inf	55.7300	59.1200	94.6200	90.3600	70.2000	Inf	Inf	Inf
2	Inf	0	Inf	60.2100	Inf	Inf	114.6200	35.5000	Inf	56.4600	Inf	Inf	Inf
3	Inf	Inf	0	Inf	Inf	Inf	Inf	Inf	111.1300	Inf	Inf	Inf	Inf
4	72.1900	60.2600	Inf	0	Inf	98.8600	96.1400	77.7400	Inf	53.3200	Inf	Inf	Inf
5	Inf	Inf	Inf	Inf	0	Inf	Inf	Inf	Inf	Inf	92.3100	Inf	92.5900
6	55.8100	Inf	Inf	99.9800	Inf	0	19.8400	112.6400	28.2500	78.7000	Inf	Inf	Inf
7	58.9800	Inf	Inf	97.1500	Inf	19.8500	0	90.5300	33.5400	56.5700	Inf	Inf	Inf
8	92.9700	35.6200	Inf	76.9800	Inf	101.3400	82.6000	0	113.9000	24.4300	Inf	Inf	Inf
9	89.3200	Inf	111.2800	Inf	Inf	28.7000	32.4700	111.2700	0	87.7800	Inf	Inf	Inf
10	70.7100	57.1700	Inf	54.7200	Inf	74.9400	56.2000	25.6600	88.8900	0	Inf	Inf	Inf
11	Inf	Inf	Inf	Inf	90.8100	Inf	Inf	Inf	Inf	Inf	0	Inf	12.2900
12	Inf	Inf	Inf	Inf	Inf	Inf	Inf	Inf	Inf	Inf	Inf	0	Inf
13	Inf	Inf	Inf	Inf	92.1200	Inf	Inf	Inf	Inf	Inf	12.1600	Inf	0

NC_city_array



	1	2	3	4	5	6	7	8	9	10	11	12	13
1	0	126.1200	198.1800	73.3700	340.5100	55.7300	59.1200	94.6200	90.3600	70.2000	250.2000	226.8400	212.2800
2	124.9900	0	253.7400	60.2100	226.2700	133.3700	114.6200	35.5000	145.9200	56.4600	135.9500	337.9100	143.4700
3	215.7500	238.1800	0	237.0800	452.5700	153.0900	142.0700	206.6800	111.1300	216.7400	362.2600	130.1500	369.7700
4	72.1900	60.2600	235.1900	0	258.1700	98.8600	96.1400	77.7400	127.3800	53.3200	167.8600	271.0200	156.5000
5	340.5000	227.3600	469.2500	250.1200	0	348.8700	330.1300	251.0100	361.4300	271.9600	92.3100	498.9300	92.5900
6	55.8100	144.1400	148.0800	99.9800	358.5300	0	19.8400	112.6400	28.2500	78.7000	268.2200	152.8300	254.2600
7	58.9800	122.0300	141.3500	97.1500	336.4200	19.8500	0	90.5300	33.5400	56.5700	246.1100	180.8900	253.6200
8	92.9700	35.6200	221.7200	76.9800	250.0100	101.3400	82.6000	0	113.9000	24.4300	159.7000	305.8900	167.2100
9	89.3200	142.7700	111.2800	127.4900	357.1600	28.7000	32.4700	111.2700	0	87.7800	266.8500	148.6800	274.3600
10	70.7100	57.1700	196.7100	54.7200	271.5600	74.9400	56.2000	25.6600	88.8900	0	181.2400	283.6300	188.7600
11	249.0800	135.9400	377.8300	168.1500	90.8100	257.4500	238.7100	159.5900	270.0100	180.5400	0	462	12.2900
12	226.8600	338.8100	129.2000	271	509.1300	145.6000	179.7400	307.3100	146.9700	282.8900	418.8200	0	407.4600
13	212.9200	143.1700	385.0600	159.2100	92.1200	264.6900	245.9500	166.8200	277.2400	187.7800	12.1600	408.0300	0

NC_city_array_H



NC_city_names

3. Dijkstra Algortihm

A. Function Description

The function implemented for Dijkstra algorithm is ***s_path_D***.

Inputs

- ***NC_city_names:*** It is a Nx1 vector containing the names of the cities. In this case is the output ***NC_city_names*** of the function ***data_retrieval***.

- ***NC_city_array:*** It is a NxN matrix containing the distances between all the cities. In this case is the output ***NC_city_array*** of the function ***data_retrieval***.

- ***Start_city:*** It is the name of the starting city.

- ***End_city:*** It is the name of the ending city.

Outputs:

- ***Miles:*** It is distance in miles of the shortest distance from ***start_city*** to ***end_city***.

- ***Route:*** It is the route of the shortest distance from start to end. It is a 1xN vector containing the name of each city visited in the way from ***start_city*** to ***end_city***. N is the quantity of cities visited.

B. Results

It is presented the shortest path for 4 pairs of cities, and then compared with Google Maps. Nevertheless, Google Maps provides the shortest path referred to time not to distance. Because of this, Google Maps uses the nearest available highways which not necessarily pass though the cities in the route calculated by Dijkstra algorithm. To verify which cities are being visited though the shortest path, it is also presented a second result visiting 8 cities of the route calculated by Dijkstra algorithm and its distance.

Remark: Google Maps has a limitation of 8 additional destinations, so it is not possible to use more than 8 cities.

Murphy – Elizabeth City (Upper_dist = 35 miles)

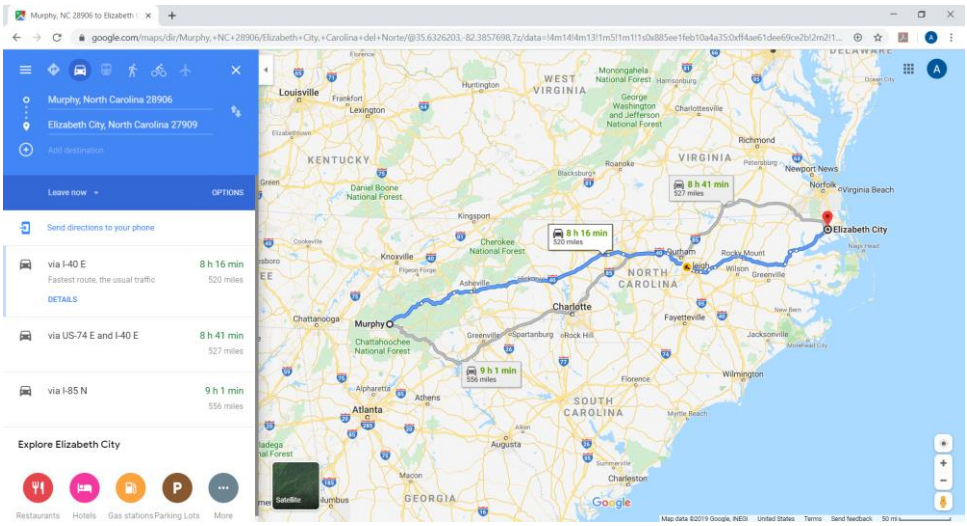
Dijkstra Algorithm

- Miles: 525.6
- Route: 23 cities visited.

  Start – Murphy – Andrews – Bryson City – Maggie Valley – Canton – Black Mountain – Marion – Connelly Springs – Hildebran – Stateville – Clemmons – Jamestown – Burlington – Mebane – Durham – Youngsville – Nashville – Rocky Mount – Princeville – Williamston – Windsor 1 – Hertford – Elizabeth City – **End**.

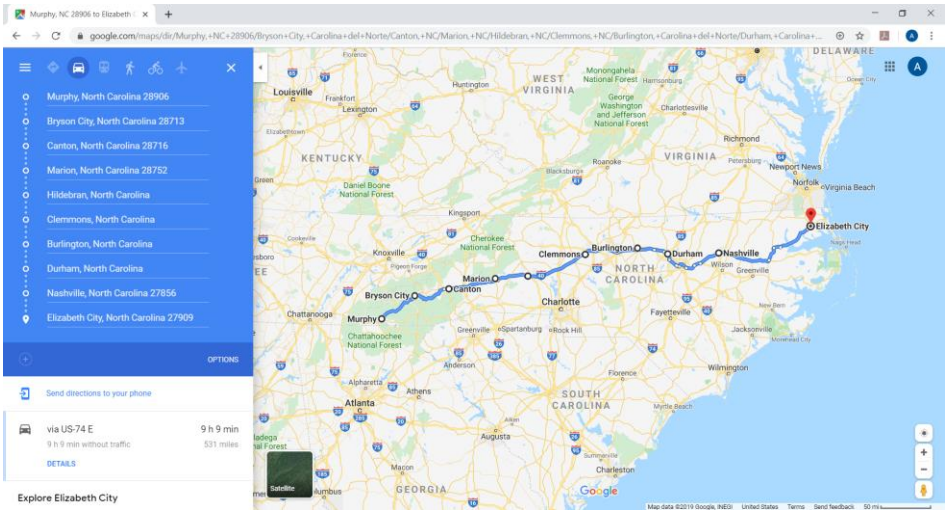
Google maps

- Miles: 520



Google Maps shortest path without cities

- Miles: 531



Google Maps shortest path with cities

Sparta – Nags Head (Upper_dist = 50 miles)

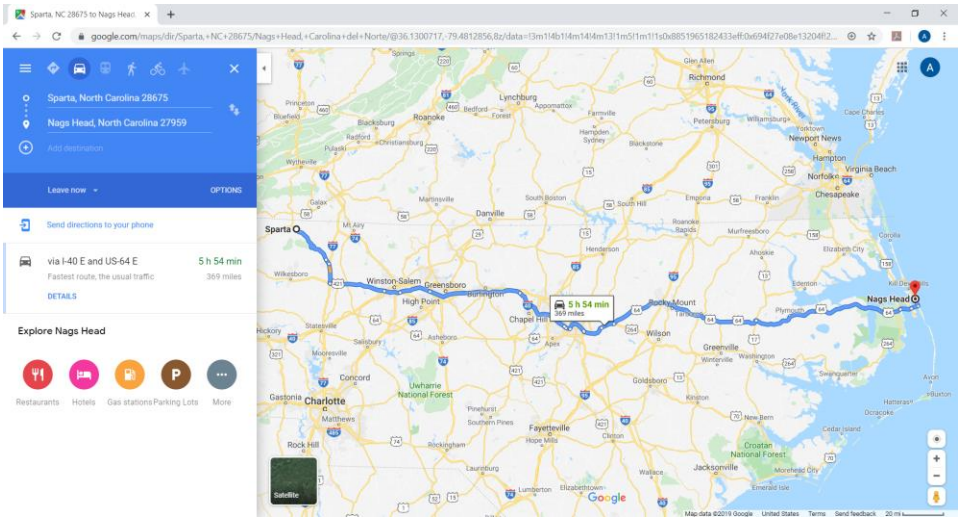
Dijkstra Algorithm

- Miles: 393.78
- Route: 12 cities visited.

  Start – Sparta – Yadkinville – Jamestown – Mebane – Durham – Youngville – Spring Hope – Princeville – Windor 1 – Elizabeth City – Kitty Hawk – Nags Head – **End**.

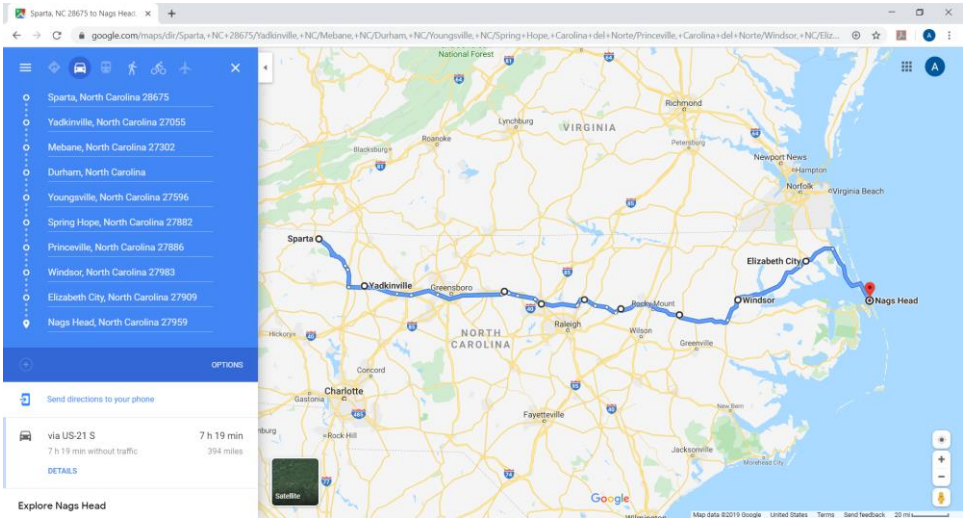
Google maps

- Miles: 369



Google Maps shortest path without cities

- Miles: 394



Google Maps shortest path with cities

Tabor City – Eden (Upper_dist = 40 miles)

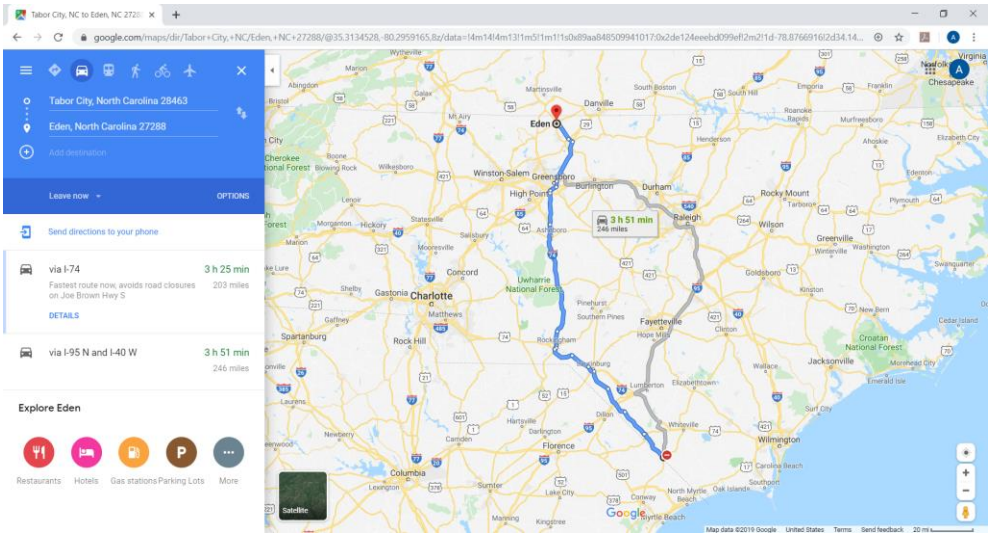
Dijkstra Algorithm

- Miles: 201.96
- Route: 8 cities visited.

  Start – Tabor City – Chadbourn – Lumberton – Fayetteville – Sanford – Pittsboro – Glen Raven – Eden – **End**.

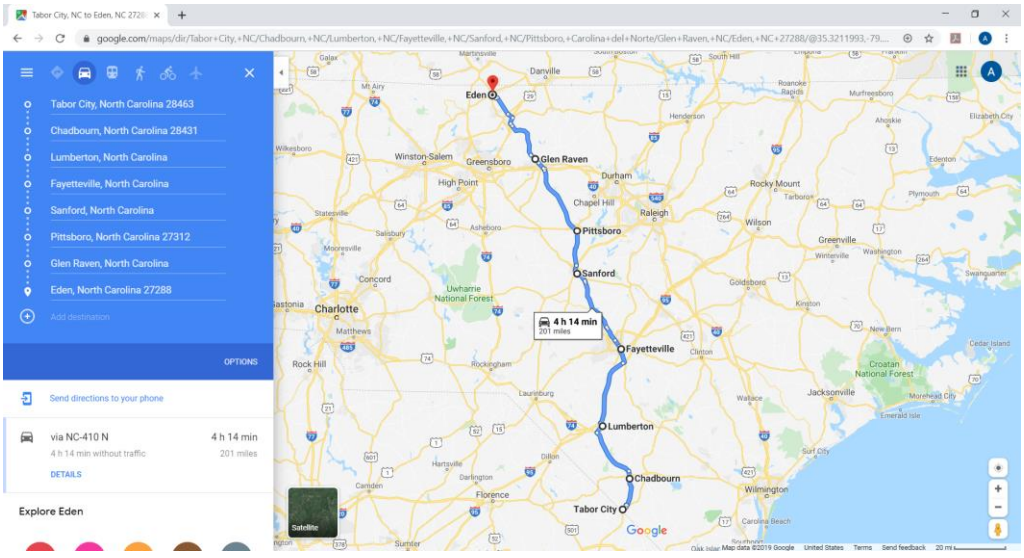
Google maps

- Miles: 203



Google Maps shortest path without cities

- Miles: 201



Google Maps shortest path with cities

Andrews – Surf City (Upper_dist = 70 miles)

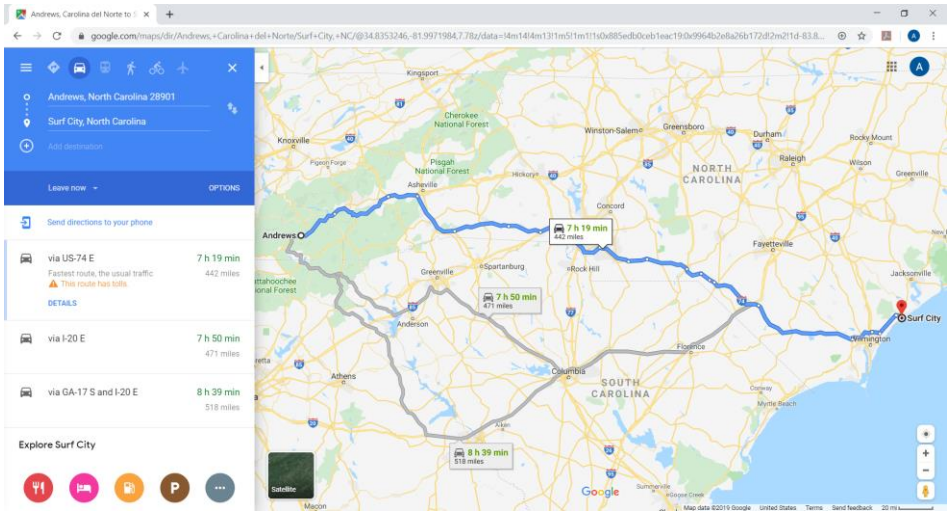
Dijkstra Algorithm

- Miles: 432.29
- Route: 13 cities visited.

  Start – Andrews – Maggie Valley – Fairview 1 – Rutherfordton – Forest City – Gastonia – Charlotte – Wadesboro – East Rockingham – Hamlet – Maxton – Lake Waccamaw – Surf City – **End**.

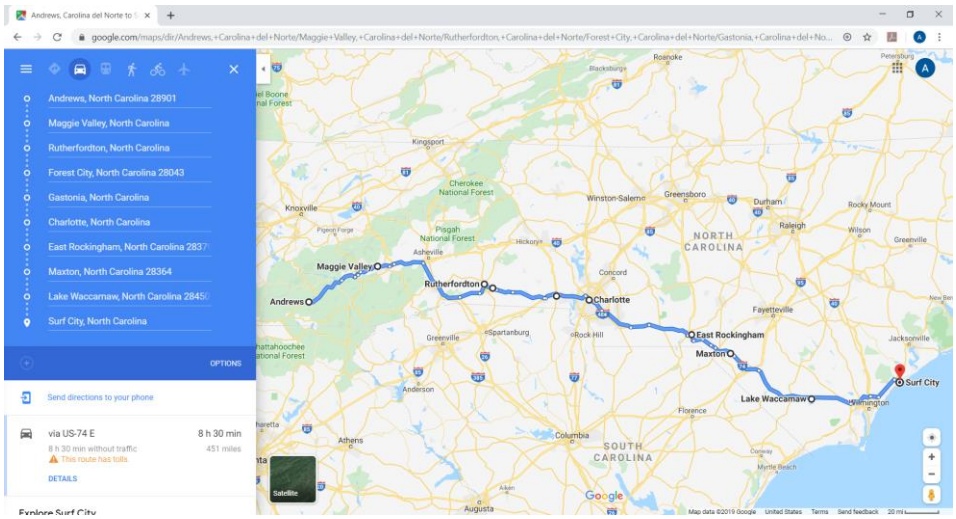
Google maps

- Miles: 442



Google Maps shortest path without cities

- Miles: 451



Google Maps shortest path with cities

Comments: From results is observed that if upper distance threshold is bigger, the algorithm solves the shortest path in less time. This fact is due to the connection between cities (vertex). If distance if big enough, the city graph will be more interconnected, and in some cases a direct connection between two cities is shorter than visiting cities between them.

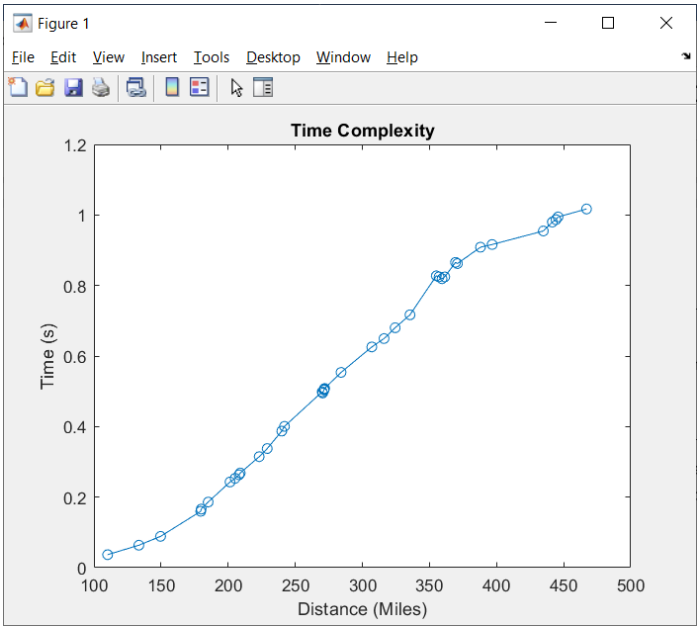
4. Analysis

A. Dijkstra Algorithm Complexity

To verify the Dijkstra algorithm complexity, it will be calculated the distance between 40 pairs of cities selected randomly, with the constrain that the start city will always be the same (star city is **Nags Head**). It was done in this way to ensure that graph is similar in all iterations. For all calculations, upper distance threshold has been set to 70.

Because cities are randomly selected, the result are not necessarily in increasing order. To visualize the increasing time complexity, distance vector is sorted and then repeated values are deleted, the same operations are applied in experimental running time vector. Then final vector is sorted without repeated elements.

The results are presented below:



Time vs distance Dijkstra Algortihm

List of cities visited (Randomly sorted):

1	Boiling Springs	**9**	Elkin	**17**	Pumpkin Center 1
2	Fairmont	**10**	Lake Waccamaw	**18**	Hoopers Creek
3	South Gastonia	**11**	Whiteville	**19**	Haw River
4	Avery Creek	**12**	Millers Creek	**20**	Woodfin
5	Waynesville	**13**	Marshville	**21**	Banner Elk
6	Rose Hill	**14**	Edneyville	**22**	Troy
7	Long Beach	**15**	Long Beach	**23**	Wallace
8	Mount Olive	**16**	Northlakes	**24**	Sneads Ferry

25	Broadway	**31**	Buies Creek	**37**	Tryon 2
26	Fairmont	**32**	Pinetops	**38**	Graham
27	Graham	**33**	Louisburg	**39**	Reidsville
28	Pumpkin Center 2	**34**	Sharpsburg	**40**	Dallas
29	Carrboro	**35**	Robersonville		
30	Rockwell	**36**	Wadesboro		

B. A* Algorithm Function Description

The function implemented for A* algorithm is ***s_path_A***. This function has the same inputs and outputs as Dijkstra algorithm, with the difference that it has one more input which is the heuristic matrix.

Inputs

- ***NC_city_names:*** Same as Dijkstra algorithm.

- ***NC_city_array:*** Same as Dijkstra algorithm.

- ***Start_city:*** It is the name of the starting city.

- ***End_city:*** It is the name of the ending city.

- ***NC_city_array_H:*** This is a NxN matrix containing the distances between all cities without the upper distance threshold. This matrix is used as heuristics for the A* algorithm. Given that A* algorithm uses a prior knowledge of the distances between the ***end_city*** and all other ones, the vector of distances corresponding to the end city is located with the same index as in ***NC_city_array***. Using it, is calculated the shortest distance. The value of N is the number of cities which is 444 in this case.

Outputs:

- ***Miles:*** Same as Dijkstra algorithm.

- ***Route:*** Same as Dijkstra algorithm.

C. A* Algorithm Tests

For tests will be presented the same cities and parameters as in Dijkstra Algorithm.

Murphy – Elizabeth City (Upper_dist = 35 miles)

A* Algorithm

- Miles: 541.93
- Route: 20 cities visited.

  Start – Murphy – Andrews – Bryson City – Clyde – Black Mountain – Glen Alpine – Connelly Springs – Stateville – Mocksville – Denton – Ramseur – Pitssboro – Apex – Archer Lodge – Nashville – Rocky Mount – Robersonville – Windsor 1 – Hertford – Elizabeth City – **End**.

Sparta – Nags Head (Upper_dist = 50 miles)

A* Algorithm

- Miles: 403.93
- Route: 12 cities visited.

  Start – Sparta – Yadkinville – Jamestown – Liberty – Pittsboro – Raleigh – Spring Hope – Princeville – Windor 1 – Elizabeth City – Kitty Hawk – Nags Head – **End**.

Tabor City – Eden (Upper_dist = 40 miles)

A* Algorithm

- Miles: 226.68
- Route: 9 cities visited.

Start – Tabor City – Chadbourn – Rowland – Hamlet – Biscoe – Franklinville – Pleasant Garden – Wentworth – Eden – **End**.

Andrews – Surf City (Upper_dist = 70 miles)

Dijkstra Algorithm

- Miles: 432.94
- Route: 8 cities visited.

  Start – Andrews – Clyde – Spindale – Charlotte – Wadesboro – Maxton – Lake Waccamaw – Surf City – **End**.

Comments: Results are summarized in the table below:

Test Parameters		Dijkstra		A*	
Start and End Cities	Upper_dist	Distance	# Cities	Distance	# Cities
Murphy – Elizabeth	35	525.6	23	541.93	20
Sparta – Nags Head	50	393.78	12	403.93	12
Tabor City – Eden	40	201.96	8	226.68	9
Andrews – Surf City	70	432.29	13	432.94	8

It is observed that A* algorithm shortest distance results are a quite bigger than Dijkstra algorithm. Moreover, the route calculated by A* is near to Dijkstra algorithm but not the same. These results differ because of the heuristics matrix. The prior knowledge assumed by the heuristic matrix is not accurate, in consequence the A* algorithm distances are above the Dijkstra ones. By using A* algorithm is important to select an accurate heuristic matrix.

Start and End Cities	Error (%)	Miles
Murphy – Elizabeth	3.11	16.33
Sparta – Nags Head	2.58	10.15
Tabor City – Eden	12.24	24.72
Andrews – Surf City	0.15	0.65

The percentage error presented is with respect of Dijkstra calculated distance. In the case of Andrews and Surf City, the upper distance threshold is 70, this is translated in a more interconnected graph. The distance error for these cities is the lowest one. From more experimentation (presented in section 4D) is verified that if upper distance threshold is increased the error is minimum for many city pairs. The heuristic matrix is, for this case, a set of edges of the graph (heuristic matrix is the distance between end city and all cities). Using this matrix is similar to consider all those edges, because of this the A* improve its accuracy if the upper distance is increased.

D. Time-Memory trade off Dijkstra and A* algorithms

First is presented the running times of Dijkstra and A* algorithms. For the A* running time, it will be used the pair of cities already selected in section 4A. Given that those pair cities were used for Dijkstra running time calculation, it can be affirmed that the graph used for both algorithms are the same.

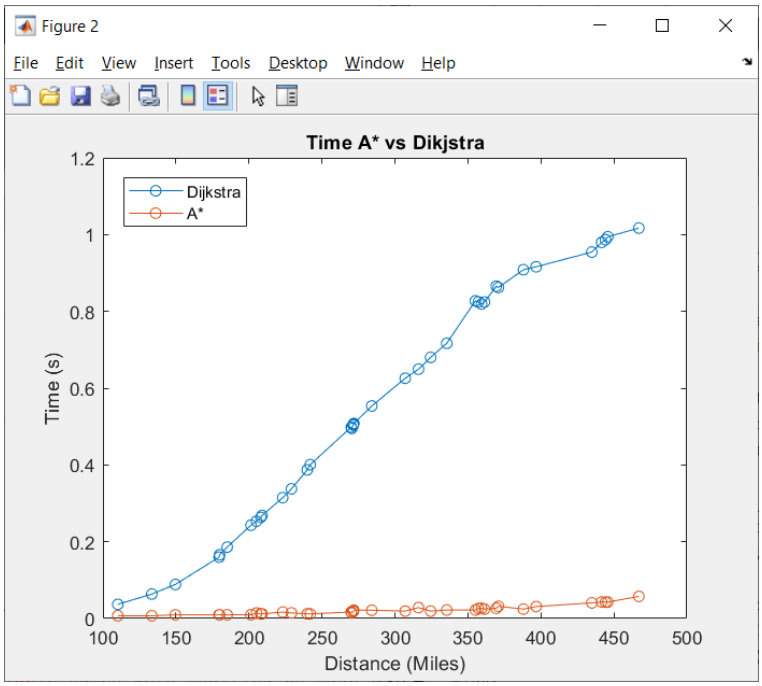
The same sort operations as in section 4A are applied here with the experimental A* algorithm running time vector.

Start city: Nags Head

The table below presents running time, miles and error results for 25 out of 40 cities. The cities are sorted randomly

End City	Dijkstra		A*		Error	Speed
Name	T (ms)	Miles	T (ms)	Miles	(%)	Ratio
Boiling Springs	909	388	24.45	398.95	2.82	37
Fairmont	508	271.85	21.61	274.29	0.9	24
South Gastonia	824	361.39	24.39	372.34	3.03	34
Avery Creek	994	445.82	42.78	452.47	1.49	23
Waynesville	1017	467.07	57.59	473.72	1.42	18
Rose Hill	243	201.37	9.72	201.93	0.28	25
Long Beach	499	270.08	16.83	272.53	0.91	30
Mount Olive	186	185.09	9.6	185.09	0	19
Elkin	717	335.48	22.05	342.39	2.06	33
Lake Waccamaw	506	271.61	19.21	271.77	0.06	26
Whiteville	504	271.45	18.95	284.21	4.7	27
Millers Creek	828	355.14	21.93	362.2	1.99	38
Marshville	680	324.46	19.25	332.89	2.6	35
Edneyville	955	434.85	40.7	445.74	2.5	23
Long Beach	497	270.08	16.69	272.53	0.91	30
Northlakes	862	370.86	31.78	377.51	1.79	27
Pumpkin Center 1	825	357.12	26.28	363.97	1.92	31
Hoopers Creek	987	444.22	42.82	452.56	1.88	23
Haw River	387	239.94	12.17	246.59	2.77	32
Woodfin	980	441.58	42.84	448.23	1.51	23
Banner Elk	916	396.71	31.15	403.62	1.74	29
Troy	554	284.12	21.57	289.63	1.94	26
Wallace	263	208.14	11.99	208.7	0.27	22
Sneads Ferry	253	205.2	14.39	205.2	0	18
Broadway	338	229.11	14.43	235.79	2.92	23

Running time plot for the 40 cities is presented below



Comments: As it was stated in section 4C, here is confirmed that with upper distance the accuracy of A* is near to Dijkstra algorithm. Therefore, considering that A* algorithm speed ratio is 20x in average, it can be selected for this type of application.

About the memory usage, the difference between both algorithms is the heuristic matrix. For this project the heuristic matrix size is 444x444, considering double values at each cell, the size is 444x444x64bits = 1.5Mbytes. In this case this size is tolerable given the speed and accuracy of the algorithm.

Name	Size	Bytes	Class
Miles_A	1x4	32	double
Miles_AE	1x40	320	double
Miles_D	1x4	32	double
Miles_DE	1x40	320	double
NC_city_array	444x444	1577088	double
NC_city_array_H	444x444	1577088	double
NC_city_names	444x1	58176	cell
Num	1x1	8	double
R	1x10	796	string

Heuristic matrix size

In general, the heuristic matrix if considered float values will have a size of $V^2 \times (4\ bytes)$, where V is the number of vertexes. For any application similar to this project, the heuristic matrix size for N cities is given by:

$$N^2.(4\ bytes)$$

If N =100000, the size is 4.10^{12} bytes which is 4TB approx. In this situation, the size is not tolerable. If it is the case, Dijkstra could be a possibility or other alternative algorithms.

MATLAB Code

Project 2 Script

```
%% 2) Data Retrieval
%% Parameters
filename = 'NC.csv';
upper_dist = 70;
%% Gets city table, city names, city distances
[NC_city_names, NC_city_array, NC_city_array_H] =
data_retrieval(filename,upper_dist);
%% 3) Dikjstra algorithm for 4 pairs of cities
% Parameters
start_cities = ["Murphy","Sparta","Tabor City","Andrews"];
end_cities = ["Elizabeth City","Nags Head","Eden","Surf City"];
% Variables
Miles_A = zeros(1,length(start_cities));
Routes_A = string(zeros(length(start_cities),25));
% Loop for each city
for i=1:length(start_cities)
    cityA = start_cities(i);
    cityB = end_cities(i);
    [Miles_A(i),R] = s_path_D(NC_city_names,NC_city_array,cityA,cityB);
    Routes_A(i,1:length(R))=R;
end
%% 4) Analysis
%% a) Verify Dikjstra Algorithm empirically
% Parameters
Num = 40; % Number random cities
start_cities_E = repmat("Nags Head",1,Num);
end_cities_E = NC_city_names(floor((length(NC_city_names)-1)*rand(1,Num))+1);
% Variables
Miles_DE = zeros(1,length(start_cities_E));
Routes_DE = string(zeros(length(start_cities_E),20));
time_D = zeros(1,length(start_cities_E));
% Loop for each city
for i=1:length(start_cities_E)
    cityA = start_cities_E(i);
    cityB = end_cities_E(i);
    tic % Start Time
    [Miles_DE(i),R] = s_path_D(NC_city_names,NC_city_array,cityA,cityB);
    time_D(i) = toc; % End Time
    Routes_DE(i,1:length(R))=R;
end
%% PLOT
% Reordering in increasing distance and eliminating repeated cities for
visualization
[ord_dist,ord] = sort(Miles_DE);
time_D_ord = time_D(ord);
[unq_dist,unqA,unqB] = unique(ord_dist);
time_D_unq = time_D_ord(unqA);
% PLOT
figure(1)
plot(unq_dist,time_D_unq,'o-');
title("Time Complexity")
xlabel("Distance (Miles)")
ylabel("Time (s)")
%% b) A* algorithm for 4 pairs of cities
% Parameters
start_cities = ["Murphy","Sparta","Tabor City","Andrews"];
end_cities = ["Elizabeth City","Nags Head","Eden","Surf City"];
% Variables
Miles_A = zeros(1,length(start_cities));
Routes_A = string(zeros(length(start_cities),25));
% Loop for each city
```

```
for i=1:length(start_cities)
    cityA = start_cities(i);
    cityB = end_cities(i);
    [Miles_A(i),R] =
s_path_A(NC_city_names,NC_city_array,NC_city_array_H,cityA,cityB);
    Routes_A(i,1:length(R))=R;
end
%% c) Verify A* and Dikjstra Algorithms Computational Time
% Parameters: Same Parameters as Dikjstra Algorithm (section a)
% Variables
Miles_AE = zeros(1,length(start_cities_E));
Routes_AE = string(zeros(length(start_cities_E),20));
time_A = zeros(1,length(start_cities_E));
% Loop for each city
for i=1:length(start_cities_E)
    cityA = start_cities_E(i);
    cityB = end_cities_E(i);
    tic % Start Time
    [Miles_AE(i),R] =
s_path_A(NC_city_names,NC_city_array,NC_city_array_H,cityA,cityB);
    time_A(i) = toc; % End Time
    Routes_AE(i,1:length(R))=R;
end
%% PLOT
% Reordering in increasing distance and eliminating repeated cities for
visualization
time_A_ord=time_A(ord);
time_A_unq=time_A_ord(unqA);
% PLOT
figure(2)
plot(unq_dist,time_D_unq,'o-');
hold on
plot(unq_dist,time_A_unq,'o-');
title("Time A* vs Dikjstra")
xlabel("Distance (Miles)")
ylabel("Time (s)")
legend('Dijkstra','A*')
```

Function s_path_D

```
function [miles, Route] = s_path_D(NC_city_names,NC_city_array,start_city,end_city)
%% Main Variables
NC_city_size = size(NC_city_array);
NC_city_cost = Inf*ones(NC_city_size(1),1); % Saves the minimum distance from a
given city to the start city
NC_city_unvisit = NC_city_names; % Saves the unvisited cities
NC_city_previous = strings(NC_city_size(1),1); % Saves the previous city from which
the distance to start city is minimum
No_path = false; % Verifies if there is a path between start and end cities
%% Algorithm Init
actual_city = start_city; % Start city is Actual city
NC_city_cost(strcmp(NC_city_names,actual_city)) = 0; % Actual city initialized with
0 cost
%% LOOP over all Table Cities, STOPS when end_city is reached
for idy = 1:length(NC_city_names)
    %% LOOP over all cities connected to actual city
    % Verifies if actual distance from start city is less than current one
    % in each city connected to actual city
    city_idx = find(strcmp(NC_city_names,actual_city)); % Actual city index
    city_vector = NC_city_array(:,city_idx); % Internal variable to store all
distances between vector of Actual city
    for idx = 1:length(city_vector)
        cityIsUnvisited = find(strcmp(NC_city_unvisit,NC_city_names{idx})); %
Verify if city is unvisited
        if (cityIsUnvisited ~= 0)
            if ((city_vector(idx)~=Inf)&&(idx~=city_idx))
                if (NC_city_cost(city_idx)+city_vector(idx)<NC_city_cost(idx))
```

```
                    NC_city_cost(idx) = NC_city_cost(city_idx)+city_vector(idx);
                    NC_city_previous(idx) = NC_city_names(city_idx);
                end
            end
        end
    end
    %% Removing actual city from unvisted array
    NC_city_unvisit{city_idx} = '';
    %% Finds unvisited city with lowest distance from start city and updates actual
city
    NC_city_cost_temp = NC_city_cost; % Auxiliary copy of city distances vector
    NC_city_cost_temp(not(~cellfun(@isempty,NC_city_unvisit)))=Inf; % All unvisited
cities get Inf value in auxiliary distance vector
    [~, dist_min_idx] = min(NC_city_cost_temp);
    actual_city = NC_city_names{dist_min_idx}; % City with lowest distance is
actual city
    %% STOP Conditions
    % If end_city is reached
    if (strcmp(actual_city,end_city))
        break
    end
    % If not possible to find shortest path
    No_path = sum(isinf(NC_city_cost_temp))==length(NC_city_cost_temp); % Verifies
if all elements of vector are Inf
    if (No_path)
        miles=-1;
        Route=-1;
        break
    end
end
%% Calculating ROUTE and MILES
% If exists a PATH
if (not(No_path))
    % Calculating Miles
    city_idx = strcmp(NC_city_names,end_city);
    miles = NC_city_cost(city_idx);
    % Calculating Route
    Route = [];
    city = end_city;
    while (not(strcmp(city,start_city)))
        city_idx = strcmp(NC_city_names,city);
        Route =[Route,city];
        city = NC_city_previous(city_idx);
    end
    Route =[Route,city];
    Route = fliplr(Route);
end
```

Function s_path_A

```
function [miles, Route] =
s_path_A(NC_city_names,NC_city_array,NC_city_array_H,start_city,end_city)
%% Parameters
%% Main Variables
NC_city_size = size(NC_city_array);
NC_city_cost = Inf*ones(NC_city_size(1),1); % Saves the minimum distance from a
given city to the start city
NC_city_unvisit = NC_city_names; % Saves the unvisited cities
NC_city_previous = strings(NC_city_size(1),1); % Saves the previous city from which
the distance to start city is minimum
No_path = false; % Verifies if there is a path between start and end cities
%% Algorithm Init
actual_city = start_city; % Start city is Actual city
NC_city_cost(strcmp(NC_city_names,actual_city)) = 0; % Actual city initialized with
0 cost
%% Calculating Heuristics
```

```
city_h_idx = strcmp(NC_city_names,end_city); % end city index (For Heuristics)
city_h = NC_city_array_H(:,city_h_idx); % Heuristics vector for end city
%% LOOP over all Table Cities, STOPS when end_city is reached
for idy = 1:length(NC_city_names)
    %% LOOP over all cities connected to actual city
    % Verifies if actual distance from start city is less than current one
    % in each city connected to actual city
    city_idx = find(strcmp(NC_city_names,actual_city)); % Actual city index
    city_g = NC_city_array(:,city_idx); % Internal variable to store all distances
between vector of Actual city
    for idx = 1:length(city_g)
        cityIsUnvisited = find(strcmp(NC_city_unvisit,NC_city_names{idx})); %
Verify if city is unvisited
        if (cityIsUnvisited ~= 0)
            if ((city_g(idx)~=Inf)&&(idx~=city_idx))
                if
(NC_city_cost(city_idx)+city_g(idx)+city_h(idx)<NC_city_cost(idx))
                    NC_city_cost(idx) = NC_city_cost(city_idx)+city_g(idx);
                    NC_city_previous(idx) = NC_city_names(city_idx);
                end
            end
        end
    end
    %% Removing actual city from unvisted array
    NC_city_unvisit{city_idx} = '';
    %% Finds unvisited city with lowest distance from start city and updates actual
city
    NC_city_cost_temp = NC_city_cost+city_h; % Auxiliary copy of city distances
vector plus heuristics distance
    NC_city_cost_temp(not(~cellfun(@isempty,NC_city_unvisit)))=Inf; % All unvisited
cities get Inf value in auxiliary distance vector
    [~, dist_min_idx] = min(NC_city_cost_temp);
    actual_city = NC_city_names{dist_min_idx}; % City with lowest distance is
actual city
    %% STOP Conditions
    % If end_city is reached
    if (strcmp(actual_city,end_city))
        break
    end
    % If not possible to find shortest path
    No_path = sum(isinf(NC_city_cost_temp))==length(NC_city_cost_temp); % Verifies
if all elements of vector are Inf
    if (No_path)
        miles=-1;
        Route=-1;
        break
    end
end
%% Calculating ROUTE and MILES
% If exists a PATH
if (not(No_path))
    % Calculating Miles
    city_idx = strcmp(NC_city_names,end_city);
    miles = NC_city_cost(city_idx);
    % Calculating Route
    Route = [];
    city = end_city;
    while (not(strcmp(city,start_city)))
        city_idx = strcmp(NC_city_names,city);
        Route =[Route,city];
        city = NC_city_previous(city_idx);
    end
    Route =[Route,city];
    Route = fliplr(Route);
end
```