ECE 592 Project 2

Kudiyar Orazymbetov (korazym@ncsu.edu)
Nico Casale (ncasale@ncsu.edu)

October 25, 2017

Note that the entries are links.

Contents

List of Tables

List of Figures

Listings

1 Introduction

This project covers an implementation and study of shortest path algorithms; namely, Dijkstra's algorithm and the A* algorithm were explored for their ability to find the shortest distance between any two cities in North Carolina. In this project, we show the results of our implementations and compare the time and memory performance of each algorithm.

The code used to execute this project can be found in Appendix A.

2 Dijkstra's Algorithm - Results

In implementing Dijkstra's algorithm, we used a data set from mileage-charts.com. To achieve closer results to the Google-recommended paths, we used a threshold of 100 miles for distances. In that way, hops greater than 100 miles were not considered by Dijkstra's Algorithm. We found that lower thresholds ($\sim$ 20 miles) would either not find a path or be a little convoluted, especially for cities with few neighbors. To illustrate the results, the figures below show the Google-recommended paths and the paths revealed by Dijkstra's algorithm.

Note that we included the data file (`mileageChart.xlxs`) so that our results can be duplicated. We fixed at least one error in the labels for the cities- 'White Plains' was listed as a city twice, but the first entry was actually the data for 'White Pines'.

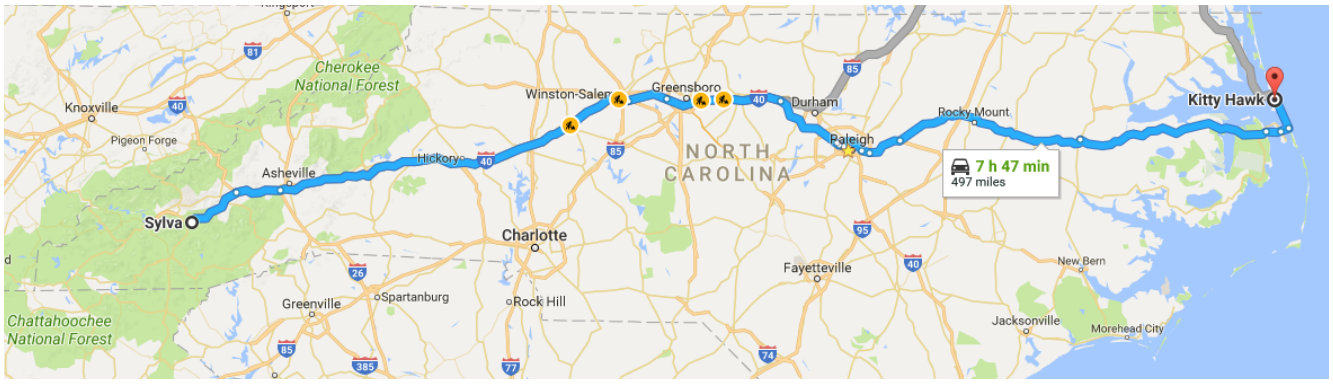

Figure 2.1: Google's recommended path from Sylva to Kitty Hawk (497 miles).

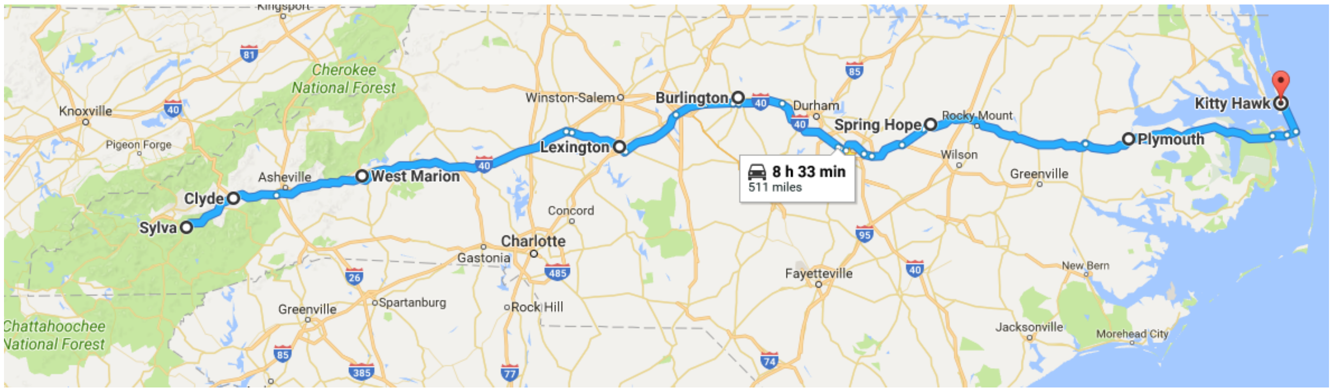

Figure 2.2: Dijkstra's algorithm's recommended path from Sylva to Kitty Hawk (511 miles).

Note that the difference in distance and path is due to the way Google Maps directs you to individual cities. When each hop is plotted, it takes the driver off of the highway, which adds distance that wouldn't be considered if you were driving the whole way without stopping. But generally, the path given by Dijkstra's algorithm does appear similar to Google's. Note that Dijkstra's algorithm is optimizing distance while Google considers time in addition to distance. The next example illustrates this trade-off clearly.

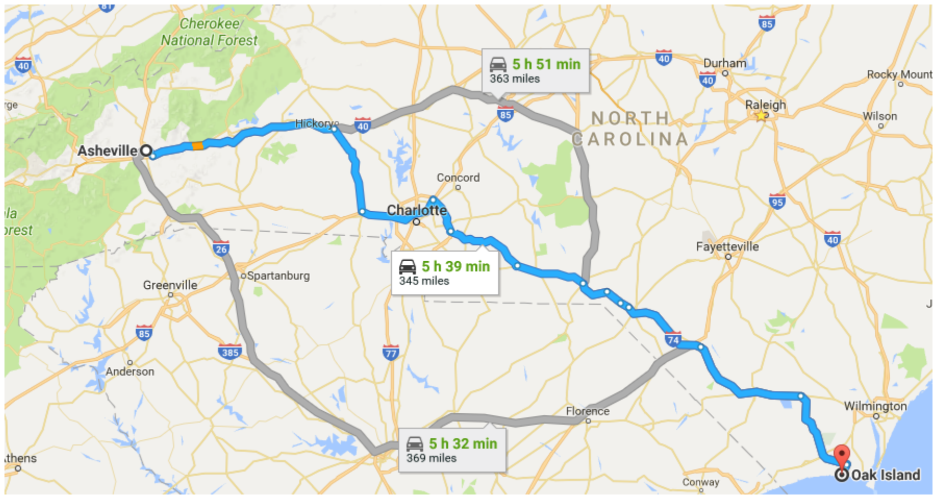

Figure 2.3: Google's recommended path from Asheville to Oak Island (345 miles).

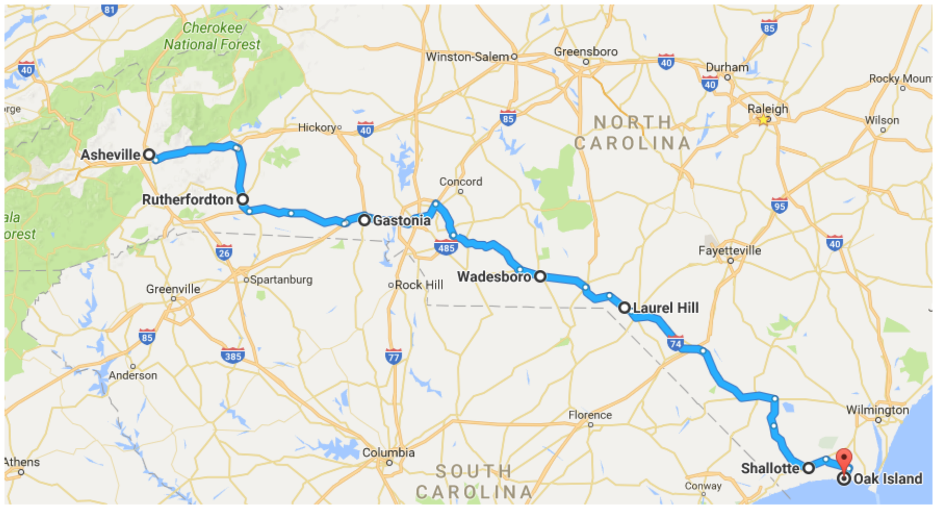

Figure 2.4: Dijkstra's algorithm's path from Asheville to Oak Island (349 miles).

Note that the paths are quite different at first. This can be attributed to the way Google optimizes for distance as well as time. In Google's recommended path, the driver would stay on Interstate 40 for a while before branching down to the Charlotte area. This is primarily to take advantage of the higher speed limits on the highway. Dijkstra's algorithm, only optimizing for distance, shows a higher total distance. This is only because Google Maps can't be finagled into mapping a very direct route without manual interference. So the indirect hop between Asheville and Rutherfordton is still present in the Dijkstra mapping. The table below summarizes these results.

Start	**Finish**	**Dijkstra (hops)**	**Google**
Sylva	Kitty Hawk	511 (10)	497
Asheville	Oak Island	349 (8)	345

Table 1: Comparison of the results of Dijkstra's algorithm and Google's recommended path.

3 Dijkstra's Algorithm - Analysis

To verify the time complexity for Dijkstra's algorithm empirically, we wrote a script to test a random collection of trips across NC. Since the running time is dependent on the number of nodes traversed, we decreased our threshold to 20 miles. This encourages the algorithm to use a larger number of hops in the paths it develops. Empirically, the plot below shows that the maximum number of hops required for most trips in NC, using 20 mile hops, is around 45. The plot also shows these results with a linear regression. The regression indicates that the trend of the runtime is dependent on the number of nodes traversed, as our algorithm halts when the destination node has been evaluated. Once the algorithm finds the destination, no other combination of nodes can yield a shorter distance to the destination by virtue of the order in which nodes are evaluated in Dijkstra's algorithm. The next node is chosen as the node with the shortest distance from the last node, and so on, starting from the initial location.

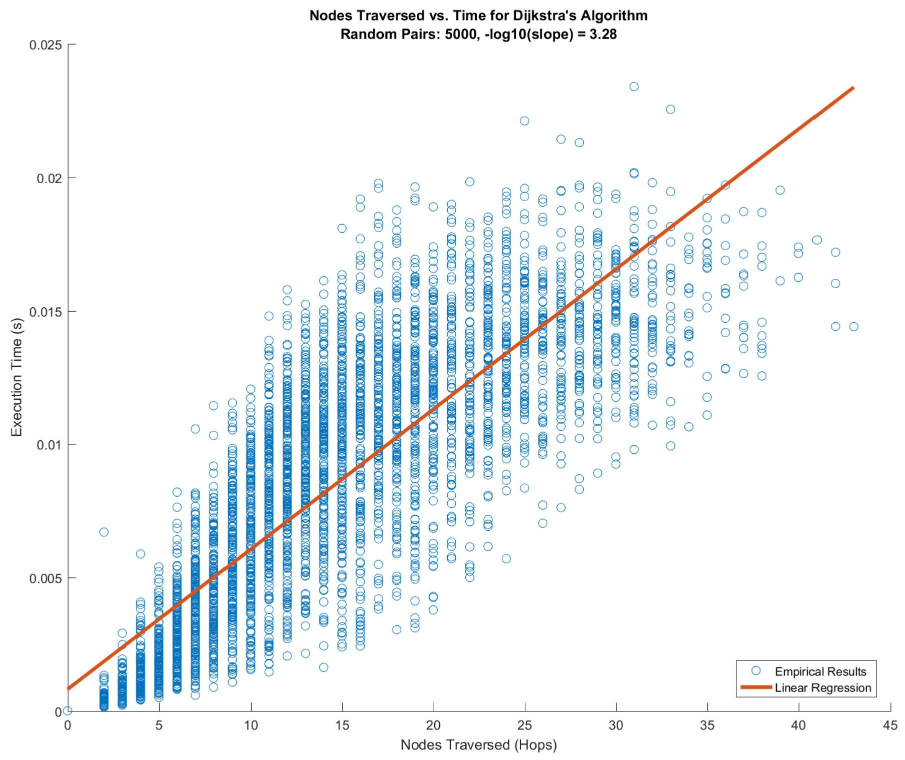

Figure 3.1: Empirical results of Dijkstra's Algorithm.

Note that the theoretical time complexity is estimated by our linear regression, which has a slope of 3.28. The precise theoretical complexity is on the order of the number of vertices in the graph squared, *i.e.* $O(|V|^2)$. There are 445 cities on our map, so the algorithm has a worst-case complexity of $O(445^2)$. This is a constant value as the big-O notation is simply a worst-case measure. As the algorithm stops early a majority of the time, it does not need to step through each node- especially if it is well connected as our map is.

4 A* Algorithm - Analysis

After implementing the A* algorithm and verifying that it yields the same result as Dijkstra's algorithm, we ran the same experiment as before with some modifications. Being that the A* algorithm requires a pre-calculated heuristic function, we used the same destination and calculated a random assortment of distances to the destination. The heuristic function depends only on the destination, so we saved some simulation time by considering only a variety of paths to the same destination. The heuristic function is used by the A* algorithm to guide the iterations toward the goal in an optimal manner. When considering nodes, the heuristic function gives an estimate for the distance from a given node to the goal node (destination.) This, ideally, should allow the algorithm to find the best path more quickly. We found that our algorithm didn't perform very well, but has potential to. Firstly, it's not really faster than our Dijkstra implementation. The Wikipedia article for the A* algorithm indicates that the heuristic function is the main factor that determines the order of complexity. If the search tree is not organized in an effective manner, the algorithm's runtime suffers. The heuristic function generates an implicit search tree based on its biases, and the article mentions that the optimal heuristic function is monotonic and admissible.

As we expect, keeping track of the heuristic data requires $|V|$ extra memory locations to store the distance of each node to the destination. MATLAB uses double-precision, but lower-precisions could be tried, as the heuristic function is meant mainly to point in the general direction. In addition, the pre-calculation is $O(|V|^2)$ as we are using Dijkstra's Algorithm to obtain the heuristic function.

The figure below shows the results of our experiment over many cities with final destinations of Raleigh and Charlotte. 5000 distances were found to each destination. Note that the runtime appears linear in the number of nodes traversed. However, the linear regression is skewed, because it's sensitive to the outliers. We also expect that the higher run-time of our implementation of the A* algorithm is due to a naïve coding scheme that doesn't maximize processing efficiency.

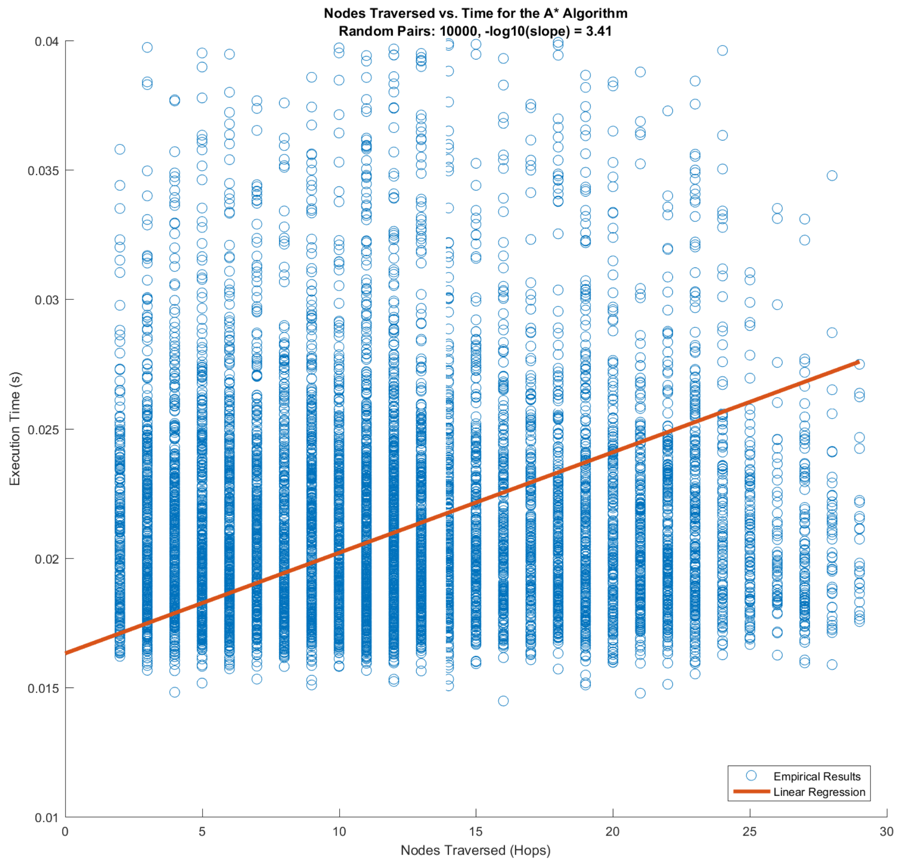

Figure 4.1: Empirical results of the A* Algorithm.

5 Conclusion

In conclusion, we have shown that at least two algorithms are feasibly implemented in MATLAB to find the shortest distance between two cities in North Carolina- Dijkstra's Algorithm and the A* Algorithm. Dijkstra's Algorithm, while slower, has a lower memory overhead than A*, which requires a pre-calculated heuristic function to achieve it's speed improvement over Dijkstra's algorithm.

6 Appendix A: Code Listings

6.1 Main Code: `project2.m`

Listing 1: Main code that simulates the project.

```
%{
ECE 592 Project 2
Kudiyar (Cody) Orazymbetov (korazym@ncsu.edu)
Nico Casale (ncasale@ncsu.edu)
%}

clear; close all;
addpath('..');

global seed
seed = 475859; rng(seed);

global imagesFolder
imagesFolder = '../images/';
addpath(imagesFolder);
overwriteImage = 1;

fprintf('ECE 592 Project 2\n');
fprintf(strcat(datestr(now),'\n'));

%% data retrieval and preprocessing
[distances, labels] = retrieveData();

%% Dijkstra's Algorithm
% choose two cities
a = 'Sylva';
b = 'Kitty Hawk';

%a = 'Asheville';
%b = 'Oak Island';

%a = 'Cullowhee';
%b = 'Moyock';

fprintf('\nStart: %s\nFinish: %s\n', a, b);

% get indexes of cities
a = find(labels == a);
b = find(labels == b);

[path, totalDistance] = dijkstra(a, b, distances, labels);

% print path
fprintf('\nThe Tao:\n');
for i = 1:length(path)
   fprintf('\t%s, index: %d\n', labels(path(i)), path(i));
end

fprintf('\nHops: %d\nTotal Distance: %3.2f\n', length(path), totalDistance);

%% Time Complexity of Dijkstra's Algorithm
```

```matlab
samples = 5000;
hops = zeros(1, samples);
times = zeros(1, samples);

for i = 1:samples
   a = randi(length(labels));
   b = randi(length(labels));

   fprintf('\nSample: %d', i);
   %fprintf(', Start: %s, Finish: %s\n', labels(a), labels(b));

   tic;
   [path, totalDistance] = dijkstra(a, b, distances, labels);
   if (isempty(path) || length(path) == 1)
      continue;
   end
   times(i) = toc;

   hops(i) = length(path);
   %fprintf('Hops: %d\nTime: %2.2f\nTotal Distance: %3.2f\n', length(path), times(i),
       totalDistance);
end

%% plot results
figure(1); clf; hold on;
h(1) = scatter(hops, times);

% theoretical results - slope 2
[hops, I] = sort(hops);
times = times(I);
X = [ones(length(hops),1) hops'];
b = X\times'; % b(2) - m b(1) - b
yTh = X*b;
h(2) = plot(hops, yTh, 'LineWidth', 3);
hold off;

axis([0 45 0 0.025]);
%set(gca,'xscale','log', 'yscale', 'log')
xlabel('Nodes Traversed (Hops)');
ylabel('Execution Time (s)');
title(sprintf('Nodes Traversed vs. Time for Dijkstra''s Algorithm\nRandom Pairs: %d, -log10(slope
    ) = %2.2f', samples, -log10(b(2))));
legend('Empirical Results', 'Linear Regression', 'Location', 'Southeast');

%% save images
if (overwriteImage)
   file = sprintf('dijkstraEmpirical');
   file = strcat(imagesFolder, file);
   print(file, '-dpng');
end

%% A* Algorithm
% choose two cities
a = 'Sylva';
b = 'Kitty Hawk';

```

```
%a = 'Asheville';
%b = 'Oak Island';

%a = 'Cullowhee';
%b = 'Moyock';

fprintf('\nStart: %s\nFinish: %s\n', a, b);

% get indexes of cities
a = find(labels == a);
b = find(labels == b);

% find the distance from every node to b using Dijkstra's Algo.
heuristics = heuristicCostFunction(b, distances, labels);
[path_A, totalDistance_A] = Astar(a, b, heuristics, distances, labels);

% print path
fprintf('\nThe Tao:\n');
for i = 1:length(path_A)
   fprintf('\t%s, index: %d\n', labels(path_A(i)), path_A(i));
end

fprintf('\nHops: %d\nTotal Distance: %3.2f\n', length(path_A), totalDistance_A);

%% Time Complexity of the A* Algorithm
samples = 10000;
hops = zeros(1, samples);
times = zeros(1, samples);

b = 'Charlotte';
b = find(labels == b);
heuristics = heuristicCostFunction(b, distances, labels);

for i = 1:samples
   a = randi(length(labels));

   fprintf('\nSample: %d', i);
   %fprintf(', Start: %s, Finish: %s\n', labels(a), labels(b));

   tic;
   [path, totalDistance] = Astar(a, b, heuristics, distances, labels);
   if (isempty(path) || length(path) == 1)
      continue;
   end
   times(i) = toc;

   hops(i) = length(path);
   %fprintf('Hops: %d\nTime: %2.2f\nTotal Distance: %3.2f\n', length(path), times(i),
       totalDistance);

   if i == samples/2
      b = 'Raleigh';
      b = find(labels == b);
      heuristics = heuristicCostFunction(b, distances, labels);
   end
end
```

```

%% plot results
figure(2); clf; hold on;
h(1) = scatter(hops, times);

% theoretical results - slope 2
[hops, I] = sort(hops);
times = times(I);
X = [ones(length(hops),1) hops'];
b = X\times'; % b(2) - m b(1) - b
yTh = X*b;
h(2) = plot(hops, yTh, 'LineWidth', 3);
hold off;

axis([0 30 0.01 0.04]);
%set(gca,'xscale','log', 'yscale', 'log')
xlabel('Nodes Traversed (Hops)');
ylabel('Execution Time (s)');
title(sprintf('Nodes Traversed vs. Time for the A* Algorithm\nRandom Pairs: %d, -log10(slope) =
    %2.2f', samples, -log10(b(2))));
legend('Empirical Results', 'Linear Regression', 'Location', 'Southeast');

%% save images
if (overwriteImage)
   file = sprintf('AstarEmpirical');
   file = strcat(imagesFolder, file);
   print(file, '-dpng');
end
```

6.2 Helper Function: `retrieveData.m`

Listing 2: Function to read the data file.

```
%{
ECE 592 Project 2
Kudiyar (Cody) Orazymbetov (korazym@ncsu.edu)
Nico Casale (ncasale@ncsu.edu)
%}

function [distances, labels] = retrieveData()

   % distances greater than minDist will be truncated to inf
   minDist = 20;

   % (b) import into matlab
   [distances, labelsCell, ~] = xlsread('mileageChart.xlsx');

   % (c) zero diagonal entries of distances
   for i = 1:size(distances,1)
      distances(i,i) = 0;
   end

   % (d) replace distances above threshold with inf
   numinf = 0;
   for i = 1:size(distances,1)
      for j = 1:size(distances,2)
```

```
            if distances(i,j) >= minDist
               distances(i,j) = inf;
               numinf = numinf + 1;
            end
         end
      end

      % print number of real valued matrix entries
      fprintf('Number of real valued distances: %d\n', numel(distances) — numinf);

      % process labels into vector
      labelsarr = string(labelsCell);
      labels = labelsarr(1,2:end);

end
```

6.3 Dijkstra's Algorithm: `dijkstra.m`

Listing 3: Function to execute Dijkstra's Algorithm.

```
%{
ECE 592 Project 2
Kudiyar (Cody) Orazymbetov (korazym@ncsu.edu)
Nico Casale (ncasale@ncsu.edu)
%}

function [path, totalDistance] = dijkstra(a, b, origDists, labels)

   % assign to every node a tentative distance
   % initial position gets 0, all other nodes get inf
   dists = inf*ones(1,size(origDists,1));
   dists(a) = 0;

   % the unvisited set
   visited = false(1,size(origDists,1));

   % keep track of parents
   parents = zeros(1,size(origDists,1),1);

   %{
   ————3————
   for current node: calculate tentative distances of all
   neighbors. Compare the tentative distances to the current
   distance that the neighbors hold and assign the smaller one.
   i.e. if current node is A and marked with a distance of 6, and the edge
   connecting it with a neighbor B has length 2, then the distance to B
   (through A) will be 6 + 2 = 8. If be was previously marked with a
   distance greater than 8 then change it to 8. Otherwise keep the current
   value.
   %}
   curr = 0;
   while(any(visited == 0))

      % choose unvisited node with the smallest distance
      tempDists = dists;
      tempDists(visited) = inf;
      last = curr;
      [~,curr] = min(tempDists);

      if (last == curr)
         break;
      end

      visited(curr) = 1;
      %nnz(visited)

      %{
      If the destination node has been marked visited
      or if the smallest tentative distance among the nodes in the unvisited
      set is infinity (when planning a complete traversal; occurs when there
      is no connection between the initial node and the remaining unvisited
      nodes), then stop the algo.
      %}
      if visited(b)
         %fprintf('Destination found.\n');
         break;
      end

      % find tentative distances to all neighbors
      tentativeDists = origDists(curr,:) + dists(curr);
      % among nonzero and noninf neighbors
      neighbors = ~isinf(tentativeDists);
      neighbors(curr) = 0;

      % don't assign neighbors' parents to current if current's parent has
      % a nieghbor... find locations where assigned dist is greater
      compareDists = dists > tentativeDists;
      compareDists(~neighbors) = 0;
      % set these locations to tentativeDists
      dists(compareDists) = tentativeDists(compareDists);

      parents(compareDists) = curr;

      % temporary print label
      %fprintf('Loc: %s\n', labels(current));

      if min(tentativeDists) == inf
         %fprintf('Warning: destination not found.\n');
      end

   end

   % walk backwards to find the path
   done = 0;
   path = [];
   trace = b;
   while(~done)
      if trace == a
         done = 1;
      end
      try
         path = [parents(trace) path];
         trace = parents(trace);
      catch
         totalDistance = 0;
         path = 0;
         return;
      end
   end
   path = [path b];
   path = path(2:end);

   totalDistance = dists(b);

end
```

6.4 A* Algorithm: `heuristicCostFunction.m` and `Astar.m`

Listing 4: Helper function to A*.

```
%{
ECE 592 Project 2
Kudiyar (Cody) Orazymbetov (korazym@ncsu.edu)
Nico Casale (ncasale@ncsu.edu)
%}

function heuristics = heuristicCostFunction(b, distances, labels)

   heuristics = zeros(1,length(labels));

   % find the distance from every node to b using Dijkstra's Algo.
   for i = 1:length(labels)
      a = i;
      if a == b
         heuristics(i) = inf;
         continue;
      end

      [~, heuristics(i)] = dijkstra(a, b, distances, labels);

      if heuristics(i) == 0
         heuristics(i) = inf;
      end

   end

end
```

Listing 5: Function to execute the A* Algorithm.

```
%{
ECE 592 Project 2
Kudiyar (Cody) Orazymbetov (korazym@ncsu.edu)
Nico Casale (ncasale@ncsu.edu)
%}

function [path, totalDistance] = Astar(a, b, heuristics, distances, labels)

```

```
    % the set of nodes already evaluated
    closedSet = false(1,size(distances,1));

    % the set of discovered nodes that are not evaluated yet
    % initially, only the start is known
    openSet = false(1,size(distances,1));
    openSet(a) = 1;

    % keep track of parents
    cameFrom = zeros(1,size(distances,1));

    % for each node, the cost of getting from a to that node
    gScore = inf*ones(1,size(distances,1));
    gScore(a) = 0;

    % for each node, the total cost of getting from the start node
    % to the goal by passing that node. partly known, partly heuristic
    fScore = inf*ones(1,size(distances,1));
    fScore(a) = heuristics(a);
    curr = 0;
    while any(openSet)
       tfScores = fScore;
       tfScores(~openSet) = inf;
       [mv, curr] = min(tfScores);

       if curr == b
          break;
       elseif mv == inf
          break;
       end

       openSet(curr) = 0;
       closedSet(curr) = 1;

       currentDists = distances(curr, :);
       % among nonzero and noninf neighbors
       neighbors = ~isinf(currentDists);
       neighbors(curr) = 0;
       neighbors(closedSet) = 0;

       openSet = openSet | neighbors;

       tentative_gScores = gScore(curr) + currentDists;
       compareDists = tentative_gScores < gScore;
       compareDists(~neighbors) = 0;

       cameFrom(compareDists) = curr;
       gScore(compareDists) = tentative_gScores(compareDists);
       fScore(compareDists) = gScore(compareDists) + heuristics(compareDists);

    end

    % walk backwards to find the path
    done = 0;
    path = [];
    trace = b;
```

```
   while(~done)
      if trace == a
         done = 1;
      end
      try
         path = [cameFrom(trace) path];
         trace = cameFrom(trace);
      catch
         totalDistance = 0;
         path = 0;
         return;
      end
   end
   path = [path b];
   path = path(2:end);

   totalDistance = gScore(b);

end
```