CS5800 Final Project Report
Route Planning and Optimization using Greedy and DP approach

Vidhi Parekh, Dhairya Amin, Sai Rahul Ponnana

Introduction

When finding an optimal path between source and destination is in question, people usually talk about finding the shortest path. Considering real-life examples like food delivery, package/mail delivery, and public transport, a lot of elements such as cost, time, and convenience are factored into finding the best route. Due to the time constraint of the project and our group members' being students, we will only care about the distance of the route and will try to answer the question, "How can we find the shortest route between two locations?".
We will find the shortest path using Kruskal's and Dijkstra's algorithms followed by a dynamic programming approach using the Bellman equation. Using the analytic and problem-solving skills learned from this class, we will explore how those two approaches will perform on our chosen dataset and figure out if they help us find the shortest route between two locations. Also, we wish to provide a comparison of the performances of all the algorithms on our dataset so that we can answer the question, "How do Kruskal's algorithm, Dijkstra's algorithm and the Bellman equation perform on a real-world dataset?"

Problem Context

Vidhi Parekh:

Route planning is the process of computing an effective method of transportation or transfer of several goods through several stops. To a large extent, route planning is used to ascertain which route is the least time-consuming when moving from one place to another. With a background in Computer Science, I would like to dive deeper

into the concepts of various shortest path algorithms, and come up with faster and efficient solutions to optimize route planning which can be further explored and used in numerous real-time applications. For instance, finding a shortest path or minimum spanning tree using Kruskal's algorithm provides an optimal cost-cutting solution to land cables or TV networks across a city or a large area. Furthermore, MST also provides a shortest path for tour operations which covers all the nodes/cities using the minimum total weight or distance such that I am able to visit all the important places in the least amount of time. This project is useful to me in terms of understanding the underlying concepts of Greedy and Dynamic programming algorithms and thoroughly implementing them to solve a real-world problem graphically. So, in this project, I am hoping to present a methodology to optimize route planning and provide an efficient solution for this problem.

Dhairya Amin:

Every time I see an Amazon delivery truck or a food delivery car, I always wonder how they would pick their routes. I would like to explore how a company would pick the shortest path from the source to the destination. From the materials learned in this class, we see how real-life problems can use graph theory to convert them into problems that can be solved using algorithms. I am a data science student, and this project will help me learn how a greedy algorithm performs vs a dynamic programming one on a real-world dataset. It will provide me with the knowledge of converting locations and road data to a graph using locations as nodes and roads as edges, and I can apply this to building accurate models in my data science projects.

Sai Rahul Ponnana:

By minimizing the distance traveled and/or the amount of time required, route planning determines the most economical path between several nodes or places. The employment of models to represent the necessary transport network forms the basis for route planning and optimization. Kruskal's, Dijkstra's algorithm will be used in this project to develop a route for any real-time scenario using a real-world dataset. The work in this project is mostly based on applying Kruskal's algorithm to map route optimization. In this project, we use various estimators, such as Euclidean, Manhattan, and diagonal distance, that help us calculate the exact distance between two points

(nodes), which in real time can be used to land cables, TV network lines, shortest walking routes, etc. We will also use the Bellman equation, a dynamic programming technique, to discover the shortest path and examine the performance of this approach. The location data is assumed to be in a 2D plane (graph), assuming there is no terrain, because we believe the actual world data to be more complex than it is.

Project Scope

In this project, we intend to find the shortest route between two locations. We will be using a graph dataset from an udacity algorithms course which contains locations as coordinates and roads as edges between two locations. We will be utilizing Kruskal's and Dijkstra's algorithm, which are greedy algorithms, and then the Bellman equation, which is a dynamic programming approach, to explore the shortest path, which can be used by packages or food delivery companies to save money on logistics, and then analyze how it performs on a real-world dataset. We would like to explore multiple algorithms on our dataset, but given the short time frame for the project, we will not be taking that into consideration.

Dataset

Our goal was to use the dataset that we mentioned in our project proposal. Upon exploring the dataset from the udacity course, it was in a custom format that required custom code provided in the course. The course wasn't free, so we had to look for a new dataset. Our team has worked with multiple datasets so, we were well equipped to create a dataset that accurately portrayed a real dataset. We created a text file that contained the hypothetical distance in miles between two U.S. cities. Using Python, we created a custom Graph object of cities as nodes and the connection between two cities as edges with the distances as the weight of those edges. We extracted each line from the text file and added it to our graph object. We have used the same graph object throughout the project.

Methodology & Analysis

Kruskal's Algorithm

When looking for the shortest path in a graph, we come across a term known as minimum spanning tree or MST. MST is a subset of the original graph that contains the same vertices and V-1 number of edges (V being the number of vertices in the original graph). The sum of weights of edges of the MST is of the minimum value. One of the algorithms that is widely used to generate a minimum spanning tree is Kruskal's algorithm.

Implementation

In Kruskal's algorithm, we first sort the edges in an ascending order based on their weights, and we add nodes to the minimum spanning tree if it does not create a cycle. To check if an edge does not create a cycle, it uses a union-find algorithm. It uses a greedy approach by picking the edge with the least weight at each iteration so, at every turn it makes a local optimal choice and in the end we get a global optimal solution. The main advantage of using Kruskal's is that it will guarantee a path that has the least weight i.e. distance in our case.
The minimum spanning tree that we got from our data is:

```
print("The minimum spanning tree with Kruskal's algorithm is:")
g.kruskal()
```

```
The minimum spanning tree with Kruskal's algorithm is:
Chicago to Boston with distance 117.45 miles
Minneapolis to Chicago with distance 148.27 miles
SaltLakeCity to Miami with distance 183.43 miles
WashingtonDC to Houston with distance 185.34 miles
Houston to SaltLakeCity with distance 215.91 miles
Denver to Albuquerque with distance 224.50 miles
KansasCity to Minneapolis with distance 245.71 miles
Nashville to Minneapolis with distance 269.04 miles
Atlanta to Nashville with distance 301.76 miles
Dallas to LasVegas with distance 316.74 miles
Omaha to KansasCity with distance 317.31 miles
Chicago to Dallas with distance 327.20 miles
Denver to StLouis with distance 331.12 miles
Atlanta to Denver with distance 350.38 miles
Seattle to Sacramento with distance 359.56 miles
Albuquerque to Sacramento with distance 359.68 miles
Houston to Boston with distance 384.70 miles
Phoenix to Omaha with distance 402.34 miles
Charolette to Omaha with distance 604.25 miles
```

Algorithm

We start from the edges with the lowest weight and keep adding edges until we reach our goal.

The steps for implementing Kruskal's algorithm are as follows:

- Sort all the edges from low weight to high
- Take the edge with the lowest weight and add it to the spanning tree. If adding the edge created a cycle, then reject this edge.
- Keep adding edges until we reach all vertices.

Pseudocode

```
KRUSKAL(G):
A = ∅
For each vertex v ∈ G.V:
   MAKE-SET(v)
For each edge (u, v) ∈ G.E ordered by increasing order by weight(u, v):
   if FIND-SET(u) ≠ FIND-SET(v):
   A = A ∪ {(u, v)}
   UNION(u, v)
return A
```

Runtime Analysis

The algorithm takes O(ElogE) time where E is the number of edges.

Dijkstra's Algorithm

Now we try to extend the greedy approach to find the shortest path between a source city and destination city. Dijkstra's algorithm is a well-known algorithm that has broad applications in the industry concerning networks or GPS especially when the algorithm is customized to suit a company's needs. Since we only have the distance between two cities i.e. positive values, Dijkstra's will help us find the shortest path between a source city and all other cities in the dataset using the distances between them.

Implementation

In Dijkstra's algorithm, we keep track of the shortest distances of nodes from the source city and those distances are initially set as infinity except for the source city which is 0. For a city whose shortest distance is yet to be calculated, we iterate through the adjacent cities of that city and then update those values only if the current distance plus the edge weight is less than the current distance. In this manner, the shortest distance is kept track of greedily for each city. In the end, we can just return the shortest distance value of the destination city.

Algorithm

Let's refer to the starting node as the initial node. Let the distance from the starting node to node Y be the distance of node Y. The Dijkstra algorithm will initially begin with unlimited distances and attempt to gradually increase them.

1. Mark all nodes unvisited. Create a set of all the unvisited nodes called the unvisited set.
2. Give each node a rough distance estimate; for our starting node, set it to zero, and for all other nodes, set it to infinity. The length of the currently known shortest path between a node v and the starting node serves as the node's tentative distance during the algorithm's execution. All other tentative distances are first set to infinity because initially no path other than the source itself (which is a path of length zero) is known to connect to any other vertex. Initial node is set as current.
3. Consider all of the current node's unexplored neighbors and estimate the distances between them and the current node. In order to determine which distance is less, compare the newly calculated tentative distance to the neighbor's current assigned distance. The distance to neighbor B through node A, for instance, will be 6 + 2 = 8 if the present node A is designated with a distance of 6 and the edge linking it with a neighbor B has length 2. Change B to 8 if it was previously marked with a distance greater than 8. If not, the current value will be retained.

4. Mark the current node as visited and delete it from the unvisited list once we have finished taking into account all of the current node's neighbors. The behavior in step 6 is ideal in that the next nodes to visit will always be in the order of "smallest distance from initial node first," meaning that any visits after would have a higher distance. A visited node will never be checked again.
5. When planning a route between two specific nodes, if the destination node has been marked as visited or when planning a full traversal, if the smallest tentative distance between the nodes in the unvisited set is infinite (this happens when there is no connection between the initial node and the remaining unvisited nodes), then stop. The algorithm is complete.
6. If not, choose the unexplored node identified with the least tentative distance, designate it as the new current node, and return to step 3 after that.

Pseudocode

```
function Dijkstra(Graph, source):

      for each vertex v in Graph.Vertices:
          dist[v] ← INFINITY
          prev[v] ← UNDEFINED
          add v to Q
      dist[source] ← 0

      while Q is not empty:
          u ← vertex in Q with min dist[u]
          remove u from Q

          for each neighbor v of u still in Q:
              alt ← dist[u] + Graph.Edges(u, v)
              if alt < dist[v] and dist[u] is not INFINITY:
                  dist[v] ← alt
                  prev[v] ← u

      return dist[], prev[]
```

Runtime Analysis

We ran the algorithm for a few source and destination cities:
The algorithm takes O(V^2) time.
V = number of vertices in the graph.

Results

Here we take Washington DC as our source city and Salt Lake City as our destination city.
The direct distance from Washington DC to Salt Lake City is 597.09 miles. Dijkstra's algorithm correctly finds the shortest path between them when a stop at Houston is taken because distance from Washington DC to Houston is 185.34 miles and Houston to Salt Lake City is 215.91 miles which brings the total to 401.25 miles.

```
Direct distance of SaltLakeCity from WashingtonDC is 597.09 miles
Shortest distance of SaltLakeCity from WashingtonDC is 401.25 miles
```

Conclusion

Upon the completion of this project, it has given us more insight as to how distance between locations affect route planning even though there are multiple variables that are at play in the industry. We've realized how broad route planning can be, we can use multiple algorithms with different computational complexities and we can modify existing algorithms to meet our goals. This project has also helped us reflect on what we have learned in the class and to translate it to practical knowledge. From the results obtained from the dataset that we created as an example, using hypothetical distances from the one city to other few cities, we can conclude that the Kruskal's algorithm which we use to find the minimal spanning tree (MST) in a graph where a shortest path is created from each and every point to every other point in a graph, can also be used to find the shortest distance between real time locations, when a proper data is given, similarly Dijkstra's algorithm gives the shortest distance between two points in a graph can give the shortest distance between locations on a real world map, a problem when formulated in a proper way using algorithms can be used for real world applications in an efficient way. Based on our observations we see that Dijkstra's is a great algorithm to form the basis for a route planning algorithm which can be optimized using different data structures or customized to meet a company's needs.

Future Scope

Due to the time complexity of the project, we are unable to explore more algorithms and huge datasets. Also, we intend to work on dynamic programmatic approaches for further exploration. In the future, we would also like to extend this project to explore and compare more approaches to come to a more well-informed opinion.

Individual Learning Experience

Vidhi Parekh:

With a background in Computer Science, I was able to dive deeper into the concepts of various shortest path algorithms, and come up with faster and efficient solutions to optimize route planning which can be further explored and used in numerous real-time applications. For instance, finding a shortest path or minimum spanning tree using Kruskal's algorithm provides an optimal cost-cutting solution to land cables or TV networks across a city or a large area. For me, I learn and retain much more by solving a real world problem due to the fact you can learn concepts in the course and apply them as real world problems in the project. This really makes me search for the answer and in return, I retain more information and try out different approaches to solve a problem. As stated in the problem context, this project is useful to me in terms of understanding the underlying concepts of Greedy and Dynamic programming algorithms and thoroughly implementing them to solve a real-world problem graphically. So, in this project, I learnt and presented a methodology to optimize route planning and provide an efficient solution for this problem.

Dhairya Amin:

I have definitely learned to appreciate the level of thinking that goes in creating efficient algorithms especially when it comes to route planning. With this project, I've learned how to create greedy algorithms from scratch and gained a better understanding of converting real-life problems into graph problems, especially shortest-path problems. We also had to create our own dataset which gave me an insight as to how data can be formatted in various ways and we have to come up with efficient methods to make use of it. As previously stated in my problem context, I will be able to apply these skills to my data science class projects and professional career.

Sai Rahul Ponnana:

I have learned about the value and operation of Kruskal's and Dijkstra's algorithms from the classroom sessions. I gained an understanding of the real-time use of these techniques through this project. In real-time applications, where the actual data is taken into account and the results are obtained, we have constructed the MST and discovered the shortest distance from one point in a graph to another point or vertex. I learned that a minimum spanning tree helps in the construction of a tree that connects all nodes, or all the locations/cities with the least amount of weight overall and in contrast, the traveling salesman problem (TSP) calls for visiting every location while returning to the starting node with the least amount of weight possible. In this way, a simple algorithm using the methods of Greedy and/or Dynamic Programming approach can be used to build and implement on a real world dataset or situation to get efficient and faster outcomes in this project of Route planning and Optimization and their comparable results.

References

1. https://www.udacity.com/course/data-structures-and-algorithms-nanodegree--nd256

2. https://www.sciencedirect.com/science/article/pii/S0005109821000303

Appendix

Code Repository:

The link to the video presentation including the dataset, source code and project report is in this Github repository:
https://github.com/Vidhi00/CS5800-Route-Planning-And-Optimization-Project

Python Notebook:

Access the Dataset

```
import numpy as np
import sys
```

```
# We have created a dataset to represent cities and flight distance (in miles) betwee
with open('/content/Dataset.txt') as f:
    lines = f.readlines()

#print(lines)
```

Custom Graph class and Kruskal's algorithm

```
# Testing Graph object
class Graph:
    # Graph object where
    # V is the number of vertices in the graph
    # graph is an array where each element
    #   is (start location, end location, distnace)
    # dict is dicttionary where keys are (city1 index, city2 index) and
    #   values are distances between them
    def __init__(self, vertices):
        self.V = vertices
        self.graph = []
        self.gdict = {}

    # Adds edge to the graph
    def add_edge(self, u, v, w):
        self.graph.append([u, v, w])

    # Union and Find functions to check if an edge creates a cycle or not

    def find(self, parent, i):
        if parent[i] == i:
            return i
        return self.find(parent, parent[i])

    def union(self, parent, rank, x, y):
        xroot = self.find(parent, x)
        yroot = self.find(parent, y)
        if rank[xroot] < rank[yroot]:
            parent[xroot] = yroot
        elif rank[xroot] > rank[yroot]:
            parent[yroot] = xroot
```

```
        else:
            parent[yroot] = xroot
            rank[xroot] += 1

    #  Applying Kruskal algorithm
    def kruskal(self):
        result = []
        i, e = 0, 0
        self.graph = sorted(self.graph, key=lambda item: item[2])
        parent = []
        rank = []
        for node in range(self.V):
            parent.append(node)
            rank.append(0)
        while e < self.V - 1:
            u, v, w = self.graph[i]
            i = i + 1
            x = self.find(parent, u)
            y = self.find(parent, v)
            if x != y:
                e = e + 1
                result.append([u, v, w])
                self.union(parent, rank, x, y)
        for u, v, weight in result:
          print("%s to %s with distance %.2f miles" % (cities[u], cities[v], weight))
```

Get Data and Create the Graph

```
# Get cities from the dataset
cities = []
for i in range(1, len(lines)):
  get_line = lines[i].split(' ')
  city1 = get_line[0]
  city2 = get_line[1]
  if city1 not in cities:
    cities.append(city1)
  if city2 not in cities:
    cities.append(city2)

# get source city user input
user_input_source = "WashingtonDC"

# make source city as index 0 to make it easier
cities[cities.index(user_input_source)] = cities[0]
cities[0] = user_input_source

# get dest city from user input
```

```
user_input_dest = "SaltLakeCity"
dest = cities.index(user_input_dest)

# Number of cities/nodes in the dataset
V = len(cities)
# Number of connections/edges in the dataset
E = len(lines)-1

# Create graph object
g = Graph(V)

# Add distances from data as edges to our graph
for i in range(1, len(lines)):
  get_line = lines[i].split(' ')
  city1 = get_line[0]
  city2 = get_line[1]
  dist = get_line[2]
  g.add_edge(cities.index(city1), cities.index(city2), float(dist))
  g.gdict[(cities.index(city1),cities.index(city2))] = float(dist)
```

Run Kruskal's algorithm

```
print("The minimum spanning tree with Kruskal's algorithm is:")
g.kruskal()
```

```
The minimum spanning tree with Kruskal's algorithm is:
Chicago to Boston with distance 117.45 miles
Minneapolis to Chicago with distance 148.27 miles
SaltLakeCity to Miami with distance 183.43 miles
WashingtonDC to Houston with distance 185.34 miles
Houston to SaltLakeCity with distance 215.91 miles
Denver to Albuquerque with distance 224.50 miles
KansasCity to Minneapolis with distance 245.71 miles
Nashville to Minneapolis with distance 269.04 miles
Atlanta to Nashville with distance 301.76 miles
Dallas to LasVegas with distance 316.74 miles
Omaha to KansasCity with distance 317.31 miles
Chicago to Dallas with distance 327.20 miles
Denver to StLouis with distance 331.12 miles
Atlanta to Denver with distance 350.38 miles
Seattle to Sacramento with distance 359.56 miles
Albuquerque to Sacramento with distance 359.68 miles
Houston to Boston with distance 384.70 miles
Phoenix to Omaha with distance 402.34 miles
Charolette to Omaha with distance 604.25 miles
```

Shortest Path using Dijkstra's algorithm

```
# Get next vertex to visit
def get_next():
  # Track if visited and distance
  global visited
  visit_this = -1
  for i in range(V):
    if visited[i][0] == 0 and (visit_this < 0 or visited[i][1] <= visited[visit_this]
      visit_this = i
  return visit_this

visited = [[0, 0]]
for i in range(V-1):
  # Initialize all nodes as not visited and distances as infinity.
  visited.append([0, sys.maxsize])

for v in range(V):
    # Get next vertex to visit
    visit_this = get_next()
    # Go through adjacent vertices
    for j in range(V):
      # If edge exists and node not visited
      if (visit_this, j) in g.gdict and visited[j][0] == 0:
        # Update new distances
        new_dist = visited[visit_this][1] + g.gdict[(visit_this,j)]
        if new_dist < visited[j][1]:
          visited[j][1] = new_dist

      visited[visit_this][0] = 1

print("Direct distance of", user_input_dest, "from", user_input_source, "is", g.gdict
print("Shortest distance of", user_input_dest, "from", user_input_source, "is", visite
```

```
Direct distance of SaltLakeCity from WashingtonDC is 597.09 miles
Shortest distance of SaltLakeCity from WashingtonDC is 401.25 miles
```