CS5800 Final Project

Fall 2023

The Most Efficient Beer Crawl Route in Portland, ME

James Bebarski, Leise Crandall, Sarah Gallant, Dan Graziano

Video link - https://www.youtube.com/watch?v=8VxrGCSkV_k

Introduction

There's no denying that drinking is an integral part of American social culture – in 2023, 62% of responders to a Gallup poll reported drinking at least occasionally[1]. Of this demographic, 69% have had a drink in the past 7 days. For most of these respondents, the drink of choice was beer. While big-name brewers hold the greatest share of beer sales annually, the craft beer industry has shown tremendous growth in both the number of breweries and the value of sales. In 2022, there were 9,552 craft breweries in the United States[2], and together, these breweries generated a retail dollar value of 28.4 billion dollars in sales[3]. Given that just 10 years earlier, craft beer sales amounted to just 11.8 billion dollars, it is evident that craft beer has rapidly cemented its place as a popular consumer choice.

In developing our research question, we considered the importance of the brewing industry in Portland, Maine, where the members of our research team are based. There is a surplus of brewery options in the state of Maine, which is second only to Vermont in number of breweries per capita. With a grand total of 14 breweries per 100,000 residents over the age of 21, navigating the Maine beer scene is a hobby in itself[4]. Thus, we aim to determine the following:

> *How do we optimize the route for a Portland, Maine brewery crawl starting at the Roux Institute by the shortest walking path in minutes?*

In our research, we apply Dijkstra's algorithm, arguably the premier algorithm used for routing, to traverse the paths between breweries in the city of Portland. Dijkstra's algorithm is a graph traversal algorithm that finds the shortest path between two nodes in a graph. It does this by repeatedly adding the node with the shortest distance between the source and destination to a set of visited nodes. The algorithm terminates when the destination node is reached, and the shortest path is returned. In our case, this information can be used to plan brewery tours or to find the most efficient way to visit multiple breweries.

Outside of the implications this research could have on encouraging brewery tourism in the Portland metropolitan area, we each hold a personal connection to the question, listed by researcher name as follows:

- **Sarah Gallant:** As a member of a family that owns a brewery here in Maine, I am always interested in what is happening with the Maine beer scene. I have observed the expansion from what was once a handful of locally successful breweries into what is now a sea of exceptional breweries throughout the state with a concentration in Portland, many of them nationally recognized. This Portland-based beer hub specifically is a large

[1] Gallup. (2023). *Alcohol and Drinking.* https://news.gallup.com/poll/1582/alcohol-drinking.aspx
[2] Statista. (2023, July). *Number of operating craft breweries in the U.S. 2006-2022.* *https://www-statista-com.ezproxy.neu.edu/study/20232/craft-beer-in-the-us-statista-dossier/*
[3] Statista. (2023, July). *Craft beer's retail dollar value in the U.S. 2011-2022.* *https://www-statista-com.ezproxy.neu.edu/study/20232/craft-beer-in-the-us-statista-dossier/*
[4] Statista. (2023, April). *U.S. craft beer breweries per capita 2022, by state.* https://www.statista.com/statistics/319978/craft-beer-breweries-per-capita-in-the-us-by-state/

tourism draw, marrying well with the ever-growing restaurant scene, the slower pace, the beautiful summers, and the ability to "get outdoors" easily. From this, it is not uncommon to see groups of people participating in beer crawls throughout the city, whether conducted commercially or privately among friends and family. This is where Dijkstra's algorithm can be helpful to us -- with inputted data describing the various breweries on the peninsula in Portland and the walking time between them, we can easily find the most efficient tour path from a specific starting point to a destination brewery. We'll minimize the overall walking time between breweries to allow more time for enjoyment on the crawl.

- **Leise Crandall:** I am a recent transplant to Portland, having moved here in August of 2022 from Virginia. While moving here opened many opportunities for me, it also meant that I was 10 hours away from any of my friends or family. Despite suddenly living in a city, my world was smaller than ever, and even surrounded by restaurants, bars, and activities, making new friends was a challenge. It was hard to know where to begin! For me, utilizing Dijkstra's algorithm to determine an efficient route between breweries, which are integral to the social scene of the Portland area, is a means to broaden my own horizons outside of the Roux Institute and my work. Especially as winter approaches, figuring out ways to visit breweries to spend time with new friends without spending too much time walking out in the cold will enliven my social prospects.

- **James Bebarski:** As a Maine resident and a student at the Roux Institute, I'm quite familiar with Portland's brewery scene. When friends and family visit, and time permits, planning a bar crawl or brewery tour is usually on our agenda. However, efficiently organizing such an outing can be challenging, given the abundance of options in downtown Portland. This is where employing an algorithm like Dijkstra's becomes invaluable in a real-world context. Dijkstra's algorithm can allow me to customize the tour by planning the most efficient route. It helps in determining the best path from our chosen starting point to our final destination, while minimizing the travel time between each brewery stop, considering the realistic time constraints of a single outing.

- **Dan Graziano:** Portland, Maine is celebrated for its eclectic food and arts scenes, but it truly comes alive with its remarkable craft beer culture. In fact, Portland has more craft breweries per capita than any other city.[5] Each brewery here offers a unique experience, often featuring inviting outdoor spaces, games, and ample room for groups to socialize, making them ideal for visitors of all ages. As someone relatively new to Maine, I'm still a novice when it comes to exploring this bustling beer scene. This causes a problem when friends and family come to visit and look for recommendations on the best spots. The challenge lies not just in choosing from the plethora of options but in planning a coherent and enjoyable walking route. This is where our project holds immense potential. By harnessing Dijkstra's algorithm, we aim to craft an optimized walking tour of breweries on the Portland peninsula. This tool won't just ease my personal predicament, but it has

[5] *C + R Research (2019, April 19). Which Cities Have the Most Craft Breweries* https://www.crresearch.com/blog/which-cities-have-most-craft-breweries/

the potential to transform the way visitors and locals alike explore Portland's flourishing craft beer scene.

Analysis

The analysis element of our project included a process of data parameterization and collection, followed by iterative development of the algorithms used to answer the question. As a team, we had to make choices regarding the specific scope of the project, the best use of resources at each stage of our project, and how best to proceed at each stage of development toward our goal of finding the optimal beer crawl route in Portland, ME from the Roux Institute based on minutes spent walking.

Data

The data collection process for our analysis began during the early discussion phase of our project. As noted in our introduction, we considered the importance of the brewing industry in Portland, ME as a broader topic driver for our algorithmic project. We then had to pinpoint the specific question that we wanted to answer, which inherently included a component of scope with respect to which Portland breweries should be considered as part of our project.

Since our premise was the very real and common situation of a brewery crawl through Portland, we were able to apply an initial parameter of including only those breweries that are located on the geographical peninsula of downtown Portland, ME. This parameter generally allowed for a reasonable walking distance in terms of minutes, our weight metric for use with Dijkstra's algorithm.

We also had to define what we considered a brewery, which was any alcohol-producing establishment that created its own beer, kombucha, or cider. With these parameters, the team was able to use a combination of personal experience, the Maine Brewers' Guild list of breweries from their website, data from Google Maps, and the websites of individual breweries to compile a list of potential breweries for consideration. After some discussion as to the reasonability and applicability of including a large number of potentially dispersed breweries for our walkers – and thus users of our algorithm – we added the parameter that breweries included in our final list must be within 15 minutes of another location. This last parameter we knew would be applied once we had collected our minutes data.

In our initial proposal, we included a short list of additional breweries located within the vicinity of the famed Allagash Brewing in Portland, ME, due to their importance to the Maine brewing industry. However, we determined during the next phase of our data collection that the added breweries wouldn’t be ideal given their walking distances greatly surpassed our 15-minute radius constraint. Therefore, these breweries were excluded from our final list.

For the collection of the minutes data, we utilized Microsoft Excel to create a matrixed list of all of the breweries (and the Roux Institute) under consideration. Every brewery was listed in both the first row and column such that you would find it takes 0 minutes at the cross-section of a brewery and itself. The edge weight data expressing the distance in minutes walked from any location on the list to any other separate location on the list was captured using Google Maps. This data was collected manually versus using the more efficient Google Maps API because it allowed us to both diversify the workload among team members and confirm that the breweries we were considering were still in business. From this list, we were able to determine the aforementioned off-peninsula breweries, as well as a few on-peninsula breweries, as they no longer meet our criteria. Thus, we developed our final list of locations – vertices – and our data in minutes for distances between the locations – edges and their weights – to use in our algorithm.

Distances Between Locations (minutes)							
	Apres	Austin Street	Batson River	Belleflower	Brickyard Hallow	Goodfire	...
Apres	**0**	4	16	2	17	5	
Austin Street	4	**0**	12	6	17	9	
Batson River	16	12	**0**	18	24	21	
Belleflower	2	6	18	**0**	19	4	
Brickyard Hallow	17	17	24	19	**0**	22	
Goodfire	5	9	21	4	22	**0**	
Gritty McDuff's	24	20	17	26	8	29	
Liquid Riot	28	22	17	29	9	32	
Lone Pine	4	9	21	4	22	1	
Newscapes	7	9	21	6	19	7	
Orange Bike	3	3	15	5	17	8	
Oxbow	9	10	22	11	13	12	
Rising Tide	4	1	13	6	16	8	
Roux Institute	18	18	27	20	6	22	
Shipyard	16	15	24	18	6	21	
Stars & Stripes	23	18	12	25	13	28	

fig. 1

Figure 1 captures a sampling of our data with the full list of included locations shown vertically in alphabetical order.

Algorithm

Early in our process, once we had a specific idea of what we wanted to analyze but had not narrowed our scope down to the final polished question, we were able to determine that we would use a greedy algorithm in our analysis. As noted previously, some of our research into routing supported the idea of using Dijkstra's algorithm to find the shortest paths and thus give us an avenue to address our question. Dijkstra's algorithm is an algorithm that we had all been exposed to in our previous education, but that we gained more familiarity with during the lessons in Module 6 (6.3 Shortest Paths) of our CS5800 course.

Since we had decided to use Dijkstra's algorithm to help answer our question concerning the optimal Portland, ME beer crawl route, we knew we had to begin our program by creating a graph using the minute distance data that we had gathered. We chose to create the program using Python due to the team's common experience with the language. Then, to utilize the helpful matrix format of the collected data, we opted to use Pandas, a Python data analysis library, to load data directly from the spreadsheet.

Our initial strategy was to apply a traditional implementation of Dijkstra's algorithm; however, this proved problematic due to the structure of our graph. Specifically, the graph we constructed was fully connected, or complete, as each brewery vertex was connected by an edge to every other vertex. In this case, the shortest path from the Roux Institute to any brewery was simply the direct edge between the two. This implementation failed to achieve our goal of creating an efficient brewery walking path that visits all – or at the very least multiple – breweries. Ultimately, our team recognized that we had constructed a variation of the Traveling Salesman Problem (TSP), which required us to rethink our earlier approach.

We were still interested in utilizing Dijkstra's algorithm, however, so we adapted its application to better suit our needs. Starting from the Roux Institute, we used Dijkstra's algorithm to determine the nearest unvisited brewery. Upon identifying this brewery, we added it to our walking path, which we stored in a dictionary. We then reapplied Dijkstra's algorithm from this new location to find the next closest unvisited brewery. This process was repeated until all breweries were visited. Throughout this process, we tracked the walking time – denoted by the weights of the edges – to gauge the total walking duration our route entailed. We then iterated through the dictionary to print our path back to the user.

This iterative path construction, while straightforward, brought up a practical consideration. Realistically, visiting 15 breweries with around 4 hours of walking might not be an enjoyable experience for most people. To tailor our algorithm for more feasible tours, we introduced adjustable constraints. These allowed us to customize the number of breweries to visit and cap the walking duration to a set number of minutes. Additionally, by factoring in the time spent at each brewery, we could estimate the total time commitment required for the tour.

This approach is fairly efficient given the small size of our graph. The most demanding part is the implementation of Dijkstra's algorithm using a priority queue. The time complexity for this is $O(V + E * \log V)$, where V is the number of vertices (breweries), E is the number of edges (paths between breweries), and logV accounts for the heap operations (push and pop) performed for every vertex. When we loop through every vertex applying Dijkstra's algorithm, the complexity increases to $O(V^2 + VE * \log V)$ since each call to Dijkstra's is $O(V + E * \log V)$ and we do this V times. Therefore, our time complexity in the worst-case scenario is a polynomial time of $O(V^2 + VE * \log V)$.

Although we were quite satisfied with our Traveling Salesman approach to creating our crawl route, we also explored the possibility of using a pure Dijkstra approach as a method for

obtaining a route. This involved generating a graph with our initial walk time data excluding edges that exceed a certain threshold. Specifically, if we set the threshold to 10, all edges with a weight greater than 10 were removed from the graph. By limiting the number of edges, we deconstructed the TSP and were able to utilize Dijkstra's successfully in its typical form. This strategy was quite simple yet effective for determining the shortest path, considering an ideal start point and ending point for the crawl. The rationale behind this approach was practical, taking into account that an individual might not want to spend excessive time walking between each brewery.

Despite the major benefits of this approach, a significant limitation of this method is the potential inability to reach the ideal destination when edges are excluded based on this constraint. For instance, if the edge threshold is set too low, there might be no viable paths to the destination in the generated graph. This was particularly evident when examining our walking time data; creating a graph with edge weights limited to just 1 resulted in almost no edges. We considered adding extra constraints to bring this approach on par with the TSP solution in terms of features. However, this would require modifying Dijskstra's directly, which would deviate from the entire objective of employing a purely Dijkstra-based method.

From a runtime perspective, Dijsktra's operates with a time complexity of O(V + E * logV), where V represents the number of vertices (breweries) and E represents the number of edges (paths between breweries). So using the purely Dijkstra approach toward creating a crawl route, would just be O(V + E * logV) as it is the main operation. This log-linear approach would be more efficient when dealing with large numbers of vertices/breweries, but the difference in performance is marginal for solving a problem like ours. Given that the TSP approach provides more constraints for route creation, we can conclude that the TSP approach is probably more useful for our purposes.

Functional Overview

The programmatic implementation of these algorithms relied on the following libraries for Python:

- Pandas
    - Used to read and process the Excel spreadsheet containing the brewery locations and walk times. Crucial for building the graph.
- Heapq
    - Used in the implementation of Dijsktra's algorithm within the Graph class. It helps us to continuously determine the next closest vertex (brewery) to visit, which is integral to our Dijsktra's code.

Further, our implementations included a series of functions to perform the algorithms successfully, as listed here:

Pure Dijkstra Approach:

- generate_graph:
    - Purpose: Constructs the graph based on the walking time data in our Excel spreadsheet. Each brewery is a vertex in the graph, and the edges represent the path between each brewery. The walking times make up the edge weights.
    - Role: The optional max_weight_threshold parameter is critical here, as it allows the function to filter out edges with weights (walking times) exceeding a specified limit. This ensures that the graph only contains practical walking paths. This preprocessing step is essential for applying Dijsktra's algorithm in a useful manner, as it decreases the connectedness of the graph.

- Dijskra's Algorithm (within the 'Graph' class):
    - Purpose: Finds the shortest path between vertices in a graph.
    - Role: It calculates the most efficient route between two breweries given the graph constructed by generate_graph.

TSP Approach:

- Crawl:
    - Purpose: Implements a heuristic method to construct a path through the graph starting from a given brewery.
    - Role: Applying a nearest-neighbor approach to sequentially visit breweries, it incorporates various optional constraints like the maximum number of breweries, total walking time, the overall time limit for the crawl, and time spent at each brewery to form a more practical route based on these preferences. This function is the core of the TSP approach.

- find_nearest_unvisited:
    - Purpose: Identifies the nearest unvisited brewery from the current location.
    - Role: This is an important helper function for the crawl function, as it helps in finding the next brewery to visit.

- Dijkstra's Algorithm:
    - Purpose: As before, it finds the shortest path between nodes in a graph.
    - Role: It's used within find_nearest_unvisited to calculate the shortest distance to each unvisited brewery from the current location.

Overall, the pure Dijkstra approach focuses on the optimal path between two points using a pre-filtered graph, while the TSP approach uses a combination of the Dijkstra's shortest path calculations, fiind_nearest_unvisted, and crawl functions to construct a feasible and enjoyable multi-stop brewery tour. For a more in-depth explanation of the individual pieces of code, I would recommend reading the in-code comments in the provided appendices.

Conclusion

When we began this project, we sought to address the question "How do we optimize the route for a Portland, Maine brewery crawl starting at the Roux Institute by the shortest walking path in minutes?" utilizing Dijkstra's algorithm. As described in our Analysis section, our final program explored our question from several different angles, using both pure- and impure Dijkstra's approaches. Our formulation of the graph utilized the full data set regarding qualifying breweries and the distance in minutes from each one to every other one, giving us a complete, fully connected graph. This graph structure had implications for how we would proceed with our implementation.

Dijkstra's at Each Step (with adjustable constraints)

As part of the modifications to our program from a pure-Dijkstra's approach to solving the traveling salesman problem presented by our graph, we both adjusted the algorithm to apply Dijkstra's anew at each leg to find the nearest unvisited brewery and added adjustable constraints. These constraints provided the ability to modify components of the crawl for a more realistic experience. With our parameters set very loosely – max breweries = none, max walking time = none, time limit = none, time at each brewery = 10 minutes – we find that we spend 228 total minutes on the crawl, 78 of which are spent walking between locations. The specific path of the brewery crawl beginning at the Roux Institute is captured in Figure 2 below.

Roux Institute -> Shipyard -> Brickyard Hallow -> Gritty McDuff's -> Liquid Riot -> Stars & Stripes -> Batson River -> Austin Street -> Rising Tide -> Orange Bike -> Apres -> Belleflower -> Goodfire -> Lone Pine -> Newscapes -> **Oxbow**

Figure 2

Pure Dijkstra's

After our successful traversal allowing for the above path to capture all 15 breweries, we reconsidered our implementation to direct our efforts towards a more pure Dijkstra approach. We recognized that the problem was created by the graph, and thus we created a modification of our graph that excluded all edges that exceeded a certain minute threshold. When we set our max edge-weight threshold to 10 minutes and input a starting vertex (location) of the Roux Institute and a destination vertex (brewery) of Belleflower Brewing, we find the path in Figure 3, which takes 21 minutes of total travel time.

Roux Institute -> Oxbow -> Apres -> **Belleflower**

Figure 3

Limitations, Alternatives, & Further Exploration

The solutions we explored to address the question posed allowed us to have a thorough and successful view of a beer crawl in Portland, Maine. Due to our early recognition of the limitations our complete graph presented for us for honoring a pure-Disjkstra's algorithm approach, we were able to implement adaptations that allowed us to sidestep many of the weaknesses of our original idea. Initially, we found the graph structure to be a weakness, and then the reality of our first approach without constraints, as it produced a nearly 4-hour brewery tour of 15 breweries. The addition of constraints provided us with the reality we sought but still presented some limitations as far as breweries included in the tour.

When we turned to the regeneration of the graph to exclude edges with a certain maximum edge weight, we made strides to create both a realistic tour and allow ourselves to set both a starting location and a destination brewery. The key limitation of this approach is that a particular destination brewery may not be reachable given a certain setting for maximum distance in minutes.

While we find our concluding program to be a successful answer to our question, we did consider some alternate algorithms that could be used in future research. Kruskal's Algorithm to create a minimum spanning tree may have helped us to create a map of sub-crawls and certainly a minimum-path crawl (see CS5800 Module 7.3). We could also have considered trying a dynamic approach for a different lens on the problem, as with the Knapsack problem in module 8.4 of our CS5800 course, which would have allowed us to use the sub-problems of the previously calculated distance to a brewery. Additionally, we could have approached the initial TSP using Held-Karp, a dynamic programming algorithm that finds the optimal solution to the problem in $\Theta(2^n n^2)$ time by storing the solutions to subproblems found recursively.

Further, were we to expand on this research for real-world applicability, we would likely consider expanding our scope. The Maine Brewers' Guild keeps a list of breweries across the state that serve as a Maine Beer Trail to encourage tourism for the industry. We could expand our graph to include more breweries within the state and broaden our time constraint for driving time (versus walking time), to help with a broader application for the guild. This program could also be applied to other tourism venues for the city or state, such as the restaurant scene and formal or informal food tours. We each also have specific thoughts regarding the value of this research and what we have learned:

- **Sarah Gallant:** My biggest takeaway from this project is a greater ability to apply the concept of a known algorithm to a real-world situation. In this course, we have been growing this muscle throughout the semester, enhancing not only our knowledge of

numerous well-known and useful algorithms but also our ability to apply them in various scenarios. Further, this project has been useful in its expansion on Dijkstra's algorithm specifically, giving us the opportunity to take what we knew the algorithm to be and apply constraints and modifications that stretched it for our purposes, giving us experience in problem-solving with each other and the algorithm. Overall, I found the project to be useful and interesting, and I could certainly see the work applied to a mobile application used by beer devotees in Maine. My hope is to take the skills I've learned from not only this project, but from the course, to future courses and work experiences.

- **Leise Crandall:** This project was a helpful learning experience as it gave me the opportunity to consider the real-world possibilities of Dijkstra's algorithm (and related algorithms as I researched the topic). I came into this program very nervous about how I would perform in the mathematics aspects of computer science. I historically struggle to work through problems when I can't visualize them in reality, and some algorithms can feel very hard to conceptualize. I enjoyed the chance to apply the knowledge I've gained in this course to something that is both applicable to the real world and also a little silly- definitely didn't consider that planning bar crawls would be a skill gained from studying computer science. Additionally, I found it very meaningful to work with a team on this project, as discussing the algorithm and its possible implementations with others deepened my own understanding. I am hopeful that the skills I gained from this will be useful in later courses- particularly for mobile and web development as I work to develop efficient and effective solutions to problems.

- **James Bebarski:** I really enjoyed applying what I learned towards a project that has real-world applications (even if it's just creating an efficient pub crawl). Often in these types of courses, I find myself more immersed in the theory, which drains my creativity and makes it challenging to envision practical problem-solving applications out of the standard examples presented in coursework. The most intriguing part of this project was discovering how to adjust and maneuver around the limitations of algorithms that we initially believed would be the best solution, and then adapting when things didn't turn out exactly as expected. When we realized that using Dijstra's algorithm purely with a graph, like what our raw data described, we didn't panic. Instead, we brainstormed ways that we could make our data work with the solution we had in mind and even provided an alternative solution. This project has enhanced my ability to think about manipulating various aspects of a problem to make it work both theoretically and practically. Whether it's by reimagining how to use data, adapting a well-established algorithm, or using only small parts of it to fit our needs, the experience was enlightening. It was refreshing to work on something that bridged theory and engineering. I'm hopeful that this class and the knowledge I've gained will serve me well in the future as I delve into classes related to machine learning and artificial intelligence.

- **Dan Graziano:** Working on this project was a refreshing experience. One challenge I've encountered in this program, particularly in theory-heavy courses, was the difficulty in visualizing how certain concepts could be applied to real-world problems. My interest in

Dijkstra's algorithm, which was sparked in CS5002, was largely theoretical. I understood its potential in applications like Google Maps but had yet to apply it personally. This project changed that. The brainstorming sessions with my teammates were pivotal, allowing us to pinpoint a common issue that Dijkstra's could solve. Although my days of lengthy bar crawls are behind me, the algorithm we developed is something I see myself using for organizing gatherings with friends and family in the future. It offers a practical way to plan efficient routes, tailoring the distance to suit the weather or our preferences. This project not only made abstract concepts more tangible but also provided additional experience in collaborating and problem-solving with others, as well as helping to improve my programming skills. Overall, it was an enjoyable project to work on and helped me better understand how one of the algorithms we learned can be applied to solve real-world problems.

Having given our thoughts on the importance of algorithmic work – and Dijkstra's specifically – to the world of beer, we leave you with a final thought on the importance of beer to algorithms from Edsger W. Dijkstra himself:

> *“I was coding in a conceptually nice and clean interface, but in spite of its conceptual simplicity apparently hopelessly inadequate. It was one of those rare beautiful days in which one can work in the garden, but in spite of the shining sun I was close to desperate. There was only one thing I could do: put all papers away, pour myself a glass of beer, look into the blue sky and figure out where I had got stuck.*
>
> *One glass of beer — even parts of it! — sufficed.”*
>
> – Dijkstra, 5th July 1975, Variations on a theme: an open letter to C.A.R.Hoare.

Appendix A: Code for our graph (used for both approaches)

```
import heapq
from vertex import Vertex

class Graph:
    """
    A class to represent a graph, or in the context of the project, our cluster of
breweries.
    Each brewery is a vertex and the edges are the walking distances between them.
    """

    def __init__(self):
        self.vertices = set()
        self.edges =  {}
        self.vertices_dict = {}

    # Method to add a vertex to the graph.
    def add_vertex(self, vertex):
        self.vertices.add(vertex)
        self.vertices_dict[vertex.name] = vertex
        if vertex not in self.edges:
            self.edges[vertex] = {}

    # Method to add an edge to the graph.
    def add_edge(self, vertex_a, vertex_b, weight):
        if vertex_a != vertex_b: # we don't want to consider self loops
            self.edges[vertex_a][vertex_b] = weight

    # String representation of a graph as an adjacency list.
    def __str__(self):
        graph = "Graph:\n"
        for vertex in self.vertices:
            connections = [f"{str(neighbor)}({self.edges[vertex].get(neighbor, 'NA')})"
for neighbor in self.vertices if neighbor != vertex]
            graph += f"{str(vertex)} -> {', '.join(connections)}\n"
        return graph

    def dijkstra(self, start_vertex):
        '''
```

```
        Implementation of Dijkstra's algorithm

        Parameters: Starting vertex

        Return: A tuple of two dictionaries - distances and predecessors

        distances: A dictionary of distances from the starting vertex to each vertex in
the graph
        predecessors: A dictionary of predecessors for each vertex in the the shortest
path from the starting vertex
        '''
        # Initialize distances dictionary, and set all distances to positive infinity
        distances = {}
        for vertex in self.vertices:
            distances[vertex] = float('inf')
        distances[start_vertex] = 0

        # Initialize predecessors dictionary
        predecessors = {}
        for vertex in self.vertices:
            predecessors[vertex] = None

        # Create priority queue and add starting vertex with distance of 0
        priority_queue = []
        heapq.heappush(priority_queue, (0, start_vertex.name))

        # Keep track of visited vertices
        visited = set()

        # Loop until queue is empty
        while len(priority_queue) > 0:
            # Pop the vertex with the smallest distance
            current = heapq.heappop(priority_queue)
            current_distance = current[0]
            current_vertex_name = current[1]
            current_vertex = None

            # Find the vertex object based on the name
            for vertex in self.vertices:
                if vertex.name == current_vertex_name:
                    current_vertex = vertex
                    break
```

```
            # Check if vertex has been visited
            if current_vertex in visited:
                continue

            # Mark current vertex as visited
            visited.add(current_vertex)

            # Check all neighbors of current vertex
            for neighbor in self.edges[current_vertex]:
                weight = self.edges[current_vertex][neighbor]
                # Calculate the new potential distance to this neighbor
                new_distance = current_distance + weight

                # Update the distance and predecessor if a shorter path is found
                if new_distance < distances[neighbor]:
                    distances[neighbor] = new_distance
                    predecessors[neighbor] = current_vertex
                    heapq.heappush(priority_queue, (new_distance, neighbor.name))

        return distances, predecessors
```

```
class Vertex:
    """
    A class that represents a vertex in a graph.
    Each vertex really just represents the name of the breweries we are interested in.
    """

    def __init__(self, name):
        self.name = name

    def __str__(self):
        return self.name

    def __repr__(self):
        return self.name
```

Appendix B: Code for TSP-Approach

```
import pandas as pd
```

```
from graph import Graph
from vertex import Vertex

def generate_graph(path, max_weight_threshold=None):
    """
    Generates the graph data from the excel file with breweries and walking distances.

    args:
        path (str): Path to the excel file with the data.
        max_weight_threshold (int, optional): Threshold for maximum edge weight.

    returns:
    graph: A graph where vertices are locations and edges represent walking distances.
    vertices: A dictionary of vertices.
    """
    graph = Graph()
    graph_data = pd.read_excel(path, index_col=0)

    # Add the vertices
    vertices = {name: Vertex(name) for name in graph_data.columns}
    for vertex in vertices.values():
        graph.add_vertex(vertex)

    # Add the edges
    for i in graph_data.index:
        for j in graph_data.columns:
            i = i.strip() # Removes any extra spaces in the spreadsheet.
            j = j.strip()
            if i != j and pd.notna(graph_data.at[i, j]):
                weight = graph_data.at[i, j]
                # Add edge only if weight is below the threshold (if specified)
                if max_weight_threshold is None or weight <= max_weight_threshold:
                    graph.add_edge(vertices[i], vertices[j], weight)


    return graph, vertices

def find_nearest_unvisited(graph, start_vertex, visited):
    '''
    Uses Dijkstra's algorithm to find the nearest unvisited vertex.

    Parameters: a graph, a starting vertex, and a list of visited vertices
```

```
    Return: The nearest unvisited vertex and its distance from the start_vertex
    '''

    distances, predecessors = graph.dijkstra(start_vertex)  # Extract only the
distances

    nearest = None
    min_distance = float('inf')

    # Loop through distance dictionary
    for vertex, distance in distances.items():
        # If the vertex has not been visited and its distance is less than the current
minimum
        if vertex not in visited and distance < min_distance:
            # Update nearest vertex and minimum distance
            nearest = vertex
            min_distance = distance

    return nearest, min_distance


def crawl(graph, start_vertex_name, max_breweries=None, max_walking_time=None,
time_limit=None, time_at_each=10):
    """
    Constructs a path through the graph using the nearest neighbor approach, starting
from the given vertex.
    The path may be constrained by the minimum and maximum number of breweries to
visit, the maximum walking time, or the overall time limit.

    args:
    graph (Graph): The graph representing breweries and walking times.
    start_vertex_name (str): The name of the starting brewery.
    max_breweries (int, optional): Max number of breweries to visit.
    max_walking_time (int, optional): Max total walking time (in minutes).
    time_limit (int, optional): Max total time for the crawl (in minutes).
    time_at_each (int): Time spent at each brewery (default 10 minutes).

    returns:
    tuple: Path of vertices visited, total time, and total walk time.
    """
    start_vertex = graph.vertices_dict[start_vertex_name]
```

```
    visited = set() # need to track visited vertices to avoid revisits
    path = [start_vertex] # initialize path with the start vertex
    total_time_spent = 0 # time spent in total (including time at each brewery and
walking time)
    total_walk_time = 0 # time spent walking (not including time at each brewery)
    brewery_count = 0 # number of breweries visited

    while True:
        current_vertex = path[-1] # current position in the crawl
        visited.add(current_vertex) # count it as visited

        # find the nearest unvisited vertex
        nearest, distance = find_nearest_unvisited(graph, current_vertex, visited)

        # if there are no more unvisited vertices, or if the nearest vertex is already
visited, stop
        if nearest is None or nearest in visited:
            break

        # if the nearest vertex is not the Roux Institute, add the time spent at each
brewery to the total time spent
        # we don't add time spent at the Roux Institute because in this context it
isn't a brewery.
        next_vertex_time = total_time_spent + distance + (time_at_each if nearest.name
!= "Roux Institute" else 0)

        # if the next vertex would violate any of the constraints, stop
        if (max_walking_time is not None and total_walk_time + distance >
max_walking_time) or \
            (max_breweries is not None and brewery_count >= max_breweries) or \
            (time_limit is not None and next_vertex_time > time_limit):
            break

        # add the nearest vertex to the path and update the total time spent and total
walk time
        path.append(nearest)
        total_walk_time += distance
        total_time_spent += distance

        # increment brewery count and add time at the brewery if it's not the Roux
Institute
        if nearest.name != "Roux Institute":
```

```
            brewery_count += 1
            total_time_spent += time_at_each

    return path, total_time_spent, total_walk_time

def print_crawl_info(graph, start_vertex_name, max_breweries=None,
max_walking_time=None, time_limit=None, time_at_each=10):
    """
    Prints information about the path generated by the crawl function, and calls the
crawl function.

    args:
    graph (Graph): The graph representing breweries and walking times.
    start_vertex_name (str): The name of the starting brewery.
    max_breweries, max_walking_time, time_limit, time_at_each: Constraints for the
crawl function.
    """
    path, total_time_spent, total_walk_time = crawl(graph, start_vertex_name,
max_breweries, max_walking_time, time_limit, time_at_each)

    print("\nWalking tour path via Traveling Salesman/neighbor: ", " -->
".join(vertex.name for vertex in path))
    print("Total time spent: ", total_time_spent)
    print("Total travel time:", total_walk_time, "minutes")
    print("\n")

def main():

    # This is the path to the vanilla spreadsheet of our walking times.
    # The only thing different about this one is there are no edges directed toward the
Roux Institute.
    path = '../cs5800-project/data/Data.xlsx'

    # Demonstrates the heuristic/traveling salesman approach to our problem.
    # Currently, this offers more constraints than the purely Dijkstra approach,
    # including a max number of breweries, max walking time, and overall time limit.
    graph2, vertices2 = generate_graph(path)
    print_crawl_info(graph2, "Roux Institute", max_breweries=5, max_walking_time=60,
time_limit=120, time_at_each=10)

if __name__ == '__main__':
    main()
```

Appendix C: Code for Pure Dijkstra's Approach

```
import pandas as pd
from graph import Graph
from vertex import Vertex

def generate_graph(path, max_weight_threshold=None):
    """
    Generates the graph data from the excel file with breweries and walking distances.

    args:
        path (str): Path to the excel file with the data.
        max_weight_threshold (int, optional): Threshold for maximum edge weight.

    returns:
    graph: A graph where vertices are locations and edges represent walking distances.
    vertices: A dictionary of vertices.
    """
    graph = Graph()
    graph_data = pd.read_excel(path, index_col=0)

    # Add the vertices
    vertices = {name: Vertex(name) for name in graph_data.columns}
    for vertex in vertices.values():
        graph.add_vertex(vertex)

    # Add the edges
    for i in graph_data.index:
        for j in graph_data.columns:
            i = i.strip() # Removes any extra spaces in the spreadsheet.
            j = j.strip()
            if i != j and pd.notna(graph_data.at[i, j]):
                weight = graph_data.at[i, j]
                # Add edge only if weight is below the threshold (if specified)
                if max_weight_threshold is None or weight <= max_weight_threshold:
                    graph.add_edge(vertices[i], vertices[j], weight)


    return graph, vertices
```

```
def print_shortest_path_info(graph, start_vertex, destination_vertex):
    """
    Prints the shortest path and total travel time between two vertices using
Dijkstra's algorithm.

    args:
    graph (Graph): The graph representing breweries and walking times.
    start_vertex (Vertex): The starting vertex.
    destination_vertex (Vertex): The destination vertex.
    """
    distances, predecessors = graph.dijkstra(start_vertex)

    total_distance = distances[destination_vertex]

    if total_distance == float('inf'):
        print(f"No path from {start_vertex.name} to {destination_vertex.name}")
        return

    # Construct the shortest path
    path = []
    current_vertex = destination_vertex
    while current_vertex is not None:
        path.insert(0, current_vertex.name)
        current_vertex = predecessors[current_vertex]

    # Print the shortest path and travel time.
    print(f"Shortest path from {start_vertex.name} to {destination_vertex.name}: {' -->
'.join(path)}")
    print(f"Total travel time: {total_distance} minutes")


def main():

    # This is the path to the vanilla spreadsheet of our walking times.
    # The only thing different about this one is there are no edges directed toward the
Roux Institute.
    path = '../cs5800-project/data/Data.xlsx'

    # Demonstrates a more simple approach to our problem, using Dijkstra's algorithm
purely.
```

```
    # Currently the only constraint added here is setting a weight threshold in graph
generation.
    # This is akin to saying we don't want to walk more than the specified minutes
between breweries.
    graph1, vertices1 = generate_graph(path, max_weight_threshold=10)
    print_shortest_path_info(graph1, vertices1["Roux Institute"],
vertices1["Belleflower"])

if __name__ == '__main__':
    main()
```