Exploration of a Simplified Reinforcement Learning Model for Travel Itinerary Application

IBM Data Science Professional Certificate Capstone Project

Johanna Posey

February 2019

Abstract

Many web based services on the market today have revolutionized travel; however, there is a lack of services that help travelers plan the fine-grain details of their itinerary. The scope of this project is to propose a simplified reinforcement learning model for a trip itinerary application and explore a rudimentary version of such an application. The Walking Tour application, which we will use to explore reinforcement learning, allows the user to input their starting location, number of places they would like to visit, distance they are willing to travel between stops and venue categories they are interested in. The algorithm uses that information and Foursquare data to produce an ordered list of venues for the user. We analyze the results of the application to gain insight into how it behaves, discuss recommendations for further analysis of the algorithm, improvements to the application and ideas for expanding the application. More analysis is needed to determine whether the proposed simplified reinforcement learning is viable for travel itinerary applications. These travel itinerary applications would be best implemented as a B2B tool for travel agencies first with the possibility of a B2C tool for simpler applications such as the Walking Tour or after further development based on B2B feedback.

TABLE OF CONTENTS

1 INTRODUCTION

1.1 PROBLEM

Many web based services on the market today have revolutionized travel. These services handle everything from booking flights and accommodations ahead of time to hailing a ride in the destination city. However, there is a lack of services that help travelers plan the fine-grain details of their itinerary.

1.2 PROPOSAL

Companies, such as Foursquare, provide data on possible venues to visit including location, business hours, trending times and more. Even more information, such as travel time between locations, can be found in additional databases. This project proposes the use of a simplified reinforcement learning model to compile information from these databases and generate a travel itinerary based on user preferences. Such an application would be of interest for travel agents and DIY travelers alike.

Reinforcement learning algorithms can run through a problem numerous times and eventually converge on an optimal solution. A RL application that explores the possible places to visit in a city taking into consideration business hours, travel times and user preferences will take a significant amount of time to run and will need to request data from multiple databases. The cost of running such an algorithm becomes very expensive as we explore larger regions and add more considerations. For this reason, we are not interested in convergence; rather, we want the algorithm to intelligently explore the environment to provide valuable solutions with as few iterations as possible.

1.3 SCOPE OF REPORT

The scope of this project is to propose a simplified reinforcement learning model for a trip itinerary application and explore a rudimentary version of such an application. We call the rudimentary application the Walking Tour application. This application helps us explore the use of simplified RL for more complex applications and provides a framework that can be built upon to create these more complex applications. The Walking Tour application explores a problem that can be solved directly allowing us to analyze the results of the application for insight into how it behaves and evaluate its performance. To conclude, we will discuss recommendations for further analysis of and improvements to the Walking Tour application and outline ideas for expanding the application.

1.4 REINFORCEMENT LEARNING

Reinforcement learning (RL) is an area of machine learning separate from both supervised and unsupervised learning. In RL, the algorithm interacts with the data and receive feedback through which it learns the optimal way to achieve its goal. RL is commonly used to train computers to play a game, such as chess. As the computer plays, it learns more about the game and what moves result in high rewards in each situation. If we tried to solve this problem without RL, we would have to define every possible series of moves that could result for each configuration and determine if each would lead to winning the game. To see why this is an issue, let's consider just the first turn. There are 20 choices for white's first move and 20 choices for black's first move. Now, each player has only taken one move, but we already have 400 (20*20) possible configurations. The number of possible configurations will grow very quickly making the problem extraordinarily complex.

To explain reinforcement learning, let's start by looking at some of its components.

- Environment: the world or domain of the problem
- Agent: the "actor" or "learner"
- State: where the agent is in the environment
- Action: what the agent does in its state- taking an action results in a new state
- Reward: the feedback the agent receives after taking an action
- Q-value: reward and expected future rewards of taking a specific action in a specific state

Take a simple example of a mouse navigating a maze, see Figure 1. The agent is the mouse, the environment is the maze, and the action is moving through the maze. The mouse will eventually find cheese, the exit or a cat. If the mouse finds the exit, it will be rewarded for reaching its goal and it will learn that that path is a good path. If the mouse finds cheese, it will get a lesser reward and continue navigating the maze. If the mouse comes across a cat, the mouse will get a negative reward and learn to avoid that path. Every time the mouse runs through the maze it learns more about the maze until it is trained to take the optimal path.

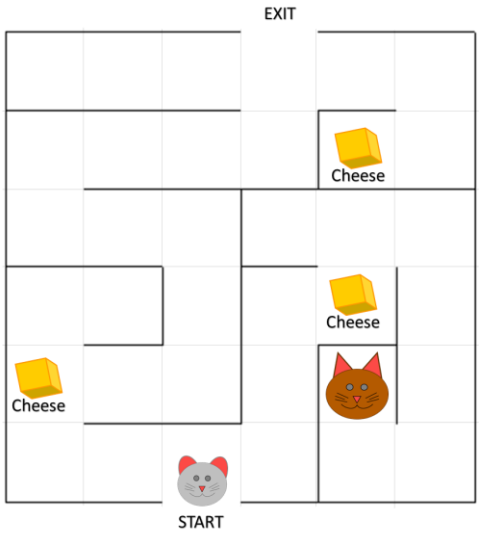

Figure 1: Mouse maze where each square in the grid is a state with a set of actions and rewards associated with that state

Continuing our mouse example, the Q-value is the reward for taking an action in its current state and discounted future rewards the mouse might receive in the new state. The rewards are discounted the further into the future they will occur since we value immediate rewards over anticipated rewards; a bird in the hand is worth two in the bush. Take our mouse at the start of the maze as in Figure 1. If the mouse moves to the square to the left, it can receive cheese in just two actions. If it goes to the right, it must take three more actions before arriving at cheese. Both lead to cheese and the reward for each first step is equal, but the value of moving left is higher because it leads to cheese in fewer steps.

1.4.1 Q-Learning

There are many types of reinforcement learning with various learning mechanisms. The RL technique we will use for our algorithm is Q-learning. Equation 1 outlines how learning takes place in Q-learning. $Q(s_n,\ a_n)$ is the Q-value for the action a_n taken in the state s_n. The state of the agent after taking the action is s_{n+1}. In the state s_{n+1}, the agent may take one of several actions, a, from the set of all possible actions in that state, $a \in A_{s_{n+1}}$. For each action there exists $Q(s_{n+1},\ a_{n+1})$. Q-learning takes r_n, the reward for taking action a_n in state s_n, and the maximum $Q(s_{n+1}, a)$ across all $a \in A_{s_{n+1}}$ and updates $Q(s_n,\ a_n)$.

$$Q(s_n,\ a_n) \leftarrow (1-\alpha) * Q(s_n,\ a_n) + \alpha[r_n + \gamma * \max\big(Q(s_{n+1}, a)\big)] \quad \textbf{(1)}$$

The learning rate, α, determines how quickly the agent learns from new information. As $\alpha \to 1$ the new information overrides the old information, whereas, $\alpha \to 0$ the algorithm doesn't learn any new

information from the action. Similarly, γ, determines how much to prioritize future rewards. As $\gamma \to 0$, the agent is only interested in instant gratification and doesn't consider future rewards.

Q-learning is an on-policy type of reinforcement learning. Policy is the strategy the agent takes to determine its next action and helps balance between exploration and exploitation. For our mouse, exploration would mean taking a path it hasn't tried yet and exploitation would be sticking to a path it knows produces a positive reward. If the mouse just explores, it will never converge on an optimal path. If it just exploits, it will always take the first path it tried. An ε-greedy policy can be implemented to ensure the agent exploits its current knowledge while occasionally exploring new actions. This policy works by generating a random number between 0 and 1. Let's take $\varepsilon = 0.1$ for example. If the random number is less than 0.1, then the agent randomly selects an action from the available actions. Otherwise, the agent exploits its knowledge to select the action with the highest estimated value. With $\varepsilon = 0.1$, the agent will exploit 90 percent of the time.

1.4.2 Simplified Reinforcement Learning

The learning rate, α, protects the algorithm from misinformation usually from human error. It is also useful when the reward for a state-action pair is not constant. For our proposed application, we assume the reward for each state-action pair will never change; therefore, we want the algorithm to learn from new information as quickly as possible. Setting $\alpha = 1$, the algorithm overrides old information with the new information. The equation to update q becomes:

$$Q(s_n,\ a_n) \leftarrow r_n + \gamma * \max\big(Q(s_{n+1}, a)\big) \quad \textbf{(2)}$$

1.5 WALKING TOUR ALGORITHM

The application we will use to explore reinforcement learning generates a "walking tour" of an area based on user preferences. The user inputs their starting location, number of places they would like to visit, distance they are willing to travel between stops and venue categories they are interested in. The algorithm takes that information and uses Foursquare data to produce an ordered list of venues for the user. Each stop is within the distance the user is willing to travel from the previous stop and the last stop on the "tour" is within that distance of the starting location, creating a loop.

2 DATA

2.1 FOURSQUARE

The Walking Tour algorithm will select venues from a list provided by Foursquare. Foursquare stores data for venues which can be accessed using various API endpoints. The SEARCH endpoint returns a list of venues within a given radius of a location that match the given parameters. Table 1, below, outlines the SEARCH parameters used in this project. A category parameter limits the results to the selected categories. The response of the SEARCH endpoint provides the id, name, location, and categories for each venue matching the search parameters.

Table 1: SEARCH Endpoint Parameters

SEARCH Endpoint Parameters		
Name	**Description**	**Value**
ll	comma separated latitude and longitude values for the location of the search	The latitude and longitude of the start location
intent	intent of the search	browse: Finds all venues within the radius that match the categoryId
radius	radius, in meters, from the desired search area	The radius will be calculated using the distance the user is willing to travel and the number of stops they want to make
limit	number of results to return	The maximum number of results Foursquare returns is 50
categoryId	comma separated list of category IDs to search for	The categories the user is interested in

To better understand the Foursquare data and the SEARCH endpoint, we will look at an example. Let's search for venues within 300 meters of the Sheraton Dallas Hotel in Dallas, TX in the category 'Dessert Shop'. Table 2 outlines the parameters used in our first search. Refer to Table 1 for a description of these parameters.

Table 2: Example SEARCH Endpoint Parameters

Example SEARCH Endpoint Parameters	
Name	**Value**
ll	32.785150, -96.794982
intent	checkin
radius	300
limit	50
categoryId	4bf58dd8d48988d1d0941735

Once we have defined the parameters we can conduct our search. From the response field we will extract relevant information and create a data frame. Table 3 shows us what this data frame will look like for our example. Each column represents a property and each row a venue.

Table 3: Example SEARCH Endpoint Results

Example SEARCH Endpoint Results						
	Name	**Category**	**Category ID**	**Latitude**	**Longitude**	**Distance from Start**
0	Chill Frozen Yogurt	Frozen Yogurt Shop	4bf58dd8d48988d1d0941735	32.7854	-96.7953	41
1	Yumi Yogurt	Ice Cream Shop	4bf58dd8d48988d1d0941735	32.7869	-96.7956	208

Notice we searched for ‘Dessert Shops’, but the category column doesn’t contain that category name. Foursquare classifies venues according to a hierarchical scheme. When you search using a category id, Foursquare returns all venues within that category and all sub-categories. The first category in the response is the lowest level category ascribed to that venue. This information will give us, and the user, more information about the venues used in the algorithm. The category ID field in Table 3 is the category ID that was used in the search rather than the ID of the first category returned. This field is to help train the algorithm to create a loop with variety in the categories of the venues visited.

The ‘Distance from Start’ is the distance in meters from the location of the search, in our example, the distance from the hotel. This will be used to train the algorithm to select venues that can loop back to the start.

2.2 WALKING TOUR

The Foursquare data will be used in the Walking Tour algorithm and in solving the problem directly. This provides us with additional data, which we will evaluate and compare. The first group of data comes from iterating through the state space to find and evaluate all possible loops. The data frame OLoops contains the loop value and list of stops for each possible loop sorted in descending order by loop value. The index is the loop number. The first five rows for an example problem are presented in Table 4.

Table 4: Sample of results for all possible loops for example problem

Example First 5 Rows of all Possible Loops Results					
	Value	**Stop 1**	**Stop 2**	**Stop 3**	**Stop 4**
4037	80.251993	Corridor Of Art	Candy Island	Chi'lantro BBQ	Eugene Bremond House
11331	79.815465	Austin Woman's Club	5th And Bowie	MBipin Pawar	Eugene Bremond House
12435	79.700453	Austin Woman's Club	MBipin Pawar	5th And Bowie	Eugene Bremond House
10232	79.565465	Eugene Bremond House	MBipin Pawar	5th And Bowie	Austin Woman's Club
4048	79.561230	Corridor Of Art	Candy Island	MBipin Pawar	Eugene Bremond House

We want to compare the results for all possible loops to that of running the algorithm. The algorithm behaves randomly, so we will run the algorithm multiple times for each case. The results will be named and stored according to the iteration for that case and the parameters for that case. For example, the results for Example 1, with $N\_steps = 500$, $\varepsilon = 0.01$, $\gamma = 0.9$ and the third iteration is stored as 'E1Loop500e10g90_3.csv'. Note epsilon and gamma are multiplied by 100 to avoid decimals in the naming scheme.

Table 5: Example of algorithm results. First 5 rows of Loop500E1e10g90_3

Example First 5 Rows of Loop Results						
	Loop	**Value**	**Stop 1**	**Stop 2**	**Stop 3**	**Stop 4**
0	1	56.530763	Austin recovery center	My Thai Mom	Twyla	5th And Bowie
1	2	56.530763	Austin recovery center	My Thai Mom	Twyla	5th And Bowie
2	3	60.512629	Austin recovery center	My Thai Mom	Eugene Bremond House	Texana Lodge No. 123
3	4	61.630814	Austin recovery center	My Thai Mom	Twyla	Corridor Of Art
4	5	52.641708	Austin recovery center	Frank Frazetta Musuem	Austin Art Alliance	Texana Lodge No. 123

We will also track and store the number of complete loops and incomplete loops, i.e. bad loops, for each iteration of a case. This data for the example above is stored as 'vars500E1e10g90.csv'. The columns are the number of completed loops, L and bad loops, BL, and each row is an iteration.

Table 6: Example of number of loops and bad loops for a case. First 5 rows of vars500E1e10g90

Example First 5 Rows of Variable Results		
	L	**BL**
i=1	20	0
i=2	20	0
i=3	20	0
i=4	16	4
i=5	17	3

3 METHODOLOGY

3.1 BUILDING ALGORITHM

3.1.1 Functions

A library of functions will carry out the reinforcement learning process. There are four parent functions which are described in Table 7. The trainalgorithm function takes the user preferences and return a data frame with the loop number, venues, and value (sum of all rewards) for each loop the algorithm constructs. The function tunealgorithm takes the venues data frame, coordinated of the starting location, list of categories, D and N, but it also takes a value for ε, γ and N_steps to adjust these parameters without editing the trainalgorithm function. The testalgorithm function is similar to the tunealgorithm function, but rather than specifying the number of steps to take you specify the percentile you want the algorithm to reach before terminating the while loop. The function optimalloop iterates through the environment and finds all possible solutions.

Table 7: Parent functions for Walking Tour application

Parent Functions			
Function	**Description**	**Input**	**Output**
trainalgorithm	Mocks how an actual client would use the application.	address, N, D, categories	Loop_det, Loop, L, BL
tunealgorithm	Similar to train algorithm, but allows modification of parameters and reduces upfront computation time.	start, venues, categories, D, N, epsilon_, gamma_, N_steps	Loop_det, Loop, L, BL
testalgorithm	Similar to tunealgorithm, but sets a parameter to terminate the while loop once that value has been exceeded	start, venues, categories, D, N, epsilon_, gamma_, MAX	Loop_det, Loop, L, BL
optimalloop	Directly solves problem by iterating through all possible solutions.	start, venues, categories, D, N	OLoop, BL

The parent functions call the other functions to acquire the venue data and construct the loops. We will look at the trainalgorithm function to explore the child functions. The trainalgorithm function imports packages, defines the RL parameters, converts the address to latitude and longitude, gets venues from Foursquare, gets the distance between each pair of venues, initializes variables, then runs a while loop

to find solutions. Table 8 outlines the child functions used to get the venues and distances, and Table 9 outlines the child functions used in the while loop to find solutions.

Table 8: Child functions for setting up problem environment Walking Tour application

Child Functions to Setup Environment			
Function	**Description**	**Input**	**Output**
get_all_venues	creates data frame of all venues in the problem *calls get_venues_in_category	D, N, categories, start	venues
get_venues_in_category	sends Foursquare API request to get all venues of a category within a radius of a location	location, category, radius, limit	search_df
getdistances	calculates the distance between every pair of venues	venues	distance

Table 9: Child functions for finding solutions in Walking Tour application

Child Functions to Find Solutions			
Function	**Description**	**Input**	**Output**
chooseaction	selects an action based on the current state *calls getactionchoices	s, N, D, venues, epsilon, q, distance	a2
getactionchoices	returns all available actions for a given state	s, N, D, venues, distance	available
changestate	changes the state based on the selected action	s, a	s2
learnq	updates the Q-table *calls changestate, getreward, getactionchoices	s, a, N, D, venues, q, distance, gamma	-
getreward	calculated the total reward and all reward components *calls assign dr	s, a, D, N, distance	dist, dr, cr, reward
assigndr	calculates the distance component of the reward	d, D	dr

The functions in Table 9 are called inside a while loop in the trainalgorithm function. Figure 2 shows a simplified flow of how the while loop works. The squares represent functions while the circles represent values. We can see in Figure 2 that there are two parts to the while loop. The loop labeled agent shows the flow of how the agent "moves" about the environment. The feedback portion shows how the environment provides feedback to the agent. The loop starts with the initial state, and based on that state the chooseaction function returns an action. This action and the current state are fed into the getreward function. The resulting reward is given to the learnq function which updates the Q-table. The action is also given to the changestate function to produce a new state. This new state is again fed into the choosaction function. The state is also an input to changestate, and state and action are inputs for learnq. After our first action (when the Q-table is no longer empty), the chooseaction function also uses the Q-table to generate an action. After each time we change the state we add one to the variable step.

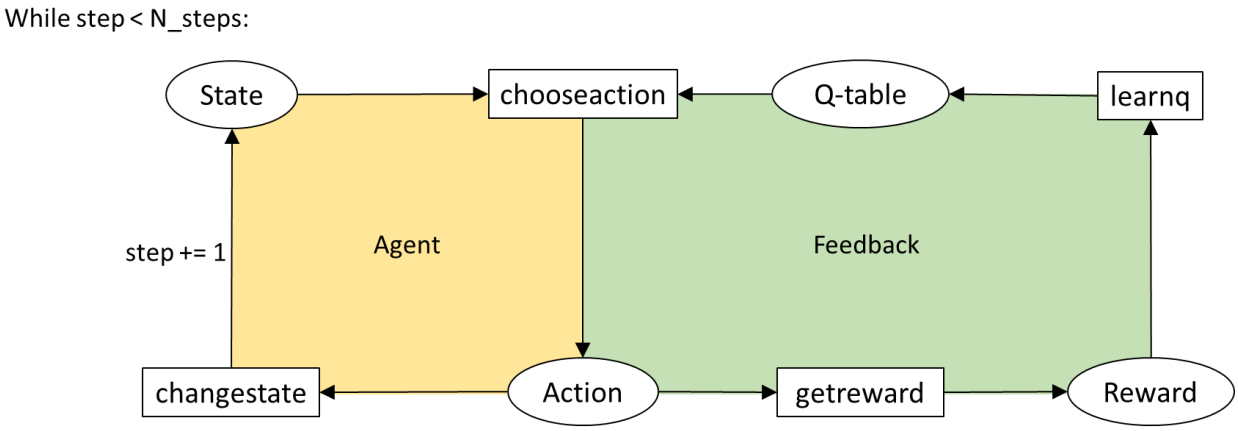

Figure 2: flowchart of while loop in trainalgorithm and tunealgorithm parent functions

This flowchart is a good explanation of how the while loop operates when the number of stops visited, n, is greater than 0 and less than N, the total number of stops on the loop. When n=0, the state is the initial state. When n=N, we calculate the value of the loop, print the value and stops to our data frame Loop, and set the state to the initial state rather than using the changestate function. If there is no available action, the chooseaction function returns None. In this case, we override the Q-value for that state action pair to a large negative number, add one to our variable BL, and set the state to the initial state.

3.1.2 Feedback

In order for our algorithm to learn, we need to provide feedback. The qualities we desire in a loop are minimum distance traveled and uniqueness of categories visited. The reward for each action can be

broken down into a distance reward (dr), category reward (cr), and finishing reward (fr). The reward used to update the Q-value is the sum of these rewards.

The distance reward ranges from 5 to 10 and is inversely proportional to the distance traveled between stops. That is, if the distance between the start and the first stop is D, the maximum distance, then dr = 5. As the distance approaches 0, dr approaches 10. The equation for the distance reward is $dr = \frac{5(D-d)}{D} + 5$ where d is the distance between stops and D is the maximum distance between stops. For the first stop, d is the distance from the start to the first stop.

The category reward ranges from -5 to 10, and is influenced by the number of venues on the loop belonging to the same category as the new venue. The category reward is always 0 for the first action. For the other actions, if the venue matches 100% of the venues already visited, the category reward will be zero. If it is an entirely new category, the category reward will be 10. Another condition exists to encourage the algorithm to avoid repeating a category at consecutive stops. If the new venue category matches the last venue category, we will subtract 5 from cr.

The finishing reward is based on the distance back to the hotel for the last venue added to the loop, and is 0 otherwise. The finishing reward only influences the reward if it is the last stop on the loop. The finishing reward is calculated similarly to the distance reward, but uses the distance back to the start rather than the distance between the stops. Assuming the user will value a shorter walk back to the start more than a shorter walk between stops, we will multiply the equation for dr by 2. The equation for fr becomes $fr = \frac{10(D-ds)}{D} + 10$ where ds is the distance from the start.

3.2 TESTING ALGORITHM

3.2.1 Expected Behavior

Under our initial parameters we can make predictions about how the algorithm will behave. These expected behaviors can help identify issues in the code and provide further insight for improving the application.

1.) Values will fall in the range of 30 to 90.

The lower bound assumes all venues are of the same category and are the maximum distance apart from one another, while the upper bound assumes all unique categories and no distance between stops.

The actual distribution will have values closer to 30 than 90 because it is more likely we will find venues of the same category spread out.

2.) We will have less than 34.39% unique loops when $\varepsilon = 0.01$.

For the algorithm to choose the best loop it has found, it must generate a random number greater than or equal to epsilon at each step. The chances of this happening is $(1 - \varepsilon)^N$. Even if the number is less than epsilon it may still choose the best action. This gives us greater than 65.61% chance it will repeat the best loop and less than 34.39% chance it will not. Additionally, there are more loops than the best loop that the algorithm may repeat making it even more unlikely it will explore a completely new loop.

3.) The "best loop" will be the highest valued loop the algorithm has previously found.

The "best loop" refers to the most commonly repeated loop during a given section of the algorithm's history. Meaning, we won't see the algorithm find a high-valued loop then return to finding the same lower value loop multiple times.

4.) The algorithm will quickly learn not to take actions that result in an incomplete loop.

Meaning, the number of bad loops should not grow linearly. Rather it should show logarithmic growth.

3.2.2 Case Study

We evaluate the behavior of the algorithm by defining a case for the algorithm to solve. Our case is a user visiting Austin, Texas, staying at the Wyndham hotel and interested in places in the categories Art Gallery, Historic Site, Asian Restaurant, Dessert Shop and Trail. Our user is willing to travel 300 meters between stops and wants to visit 4 places before returning to the hotel. The input for this case can be found in Table 10 below.

Table 10: Input variables for Example 1

E1 Input Variables	
Input Variable	**Value**
address	'516 W 8th Street Austin TX'
categories	['4bf58dd8d48988d1e2931735', '4deefb944765f83613cdba6e', '4bf58dd8d48988d142941735', '4bf58dd8d48988d1d0941735', '4bf58dd8d48988d159941735']
N	4
D	300

3.2.3 Direct Solution

Once we have defined the input for our case, we write a function to find all the possible loops for that case. This will provide us with a distribution of loop values which can be used to evaluate the performance of our algorithm. The function optimalloop finds all states that can be reached from the initial state. Then, for each of those states it finds all possible second states and so on until we have found all of the possible final states. Using those final states, the algorithm calculates the final value for each loop and sorts the loops in descending order by value. The function also stores the number of states that don't have any available actions in a variable BL. BL represents the number of possible bad loops the algorithm can find.

3.2.4 Walking Tour Analysis

For our analysis we select 0.01 for ε and 1 for γ. We select $\gamma = 1$ because we are trying to maximize the final value rather than the reward for any particular step. An $\varepsilon = 0.01$ means the algorithm will choose a random action approximately 10% of the time. We could set $\varepsilon = 1$ to maximize the number of unique loops, but we assume that loops that start off with high rewards result in better loops than ones that start off randomly.

To check for errors in our algorithm and to learn more about its behavior we will run the algorithm for 1500 steps 5 times. 1500 steps will result in approximately 300 loops in our example. We select 1500 to balance the number of solutions and the computation time. Additionally, we will conduct 30 trials where we run the algorithm till it finds a solution in the 95th percentile, and 30 trials to find a solution in the 99th percentile.

4 RESULTS

4.1 PROBLEM ENVIRONMENT

We will first look at the environment for the problem, i.e. the venues and their locations. Table 11 shows the number of venues in each category and their average distance from the start. We can see from this table that the majority of the venues are in the category Art Gallery and there are no venues in the category Trail. We can also so that while there are only 3 historic sites, they are on average the closest to the start. We expect to see art galleries, historic sites and Asian restaurants in many of our solutions.

Table 11: Venue count and mean distance from start by primary category

Venues by Primary Category			
Primary Category ID	**Primary Category Name**	**Count**	**Mean Distance From Start**
4bf58dd8d48988d142941735	Asian Restaurant	11	479.9
4bf58dd8d48988d1d0941735	Dessert Shop	3	526.7
4bf58dd8d48988d1e2931735	Art Gallery	20	457.5
4deefb944765f83613cdba6e	Historic Site	3	182.3
4bf58dd8d48988d159941735	Trail	0	-

Looking at the breakdown of venues by sub-category in Table 12 we can get an even better picture of the state space and the Foursquare data. One particular piece of data to note is the Nightclub categorized as an art gallery.

Table 12: Venue count and mean distance from start by primary category

Venue Count by Secondary Category			
Primary Category ID	**Sub-Category Name**	**Count**	**Mean Distance from Start**
4bf58dd8d48988d142941735	Asian Restaurant	2	504.5
	Chinese Restaurant	2	431.5
	Food Truck	2	483.5
	Japanese Curry Restaurant	1	535.0
	Japanese Restaurant	1	500.0
	Sushi Restaurant	2	445.0
	Thai Restaurant	1	515.0
4bf58dd8d48988d1d0941735	Dessert Shop	1	560.0
	Food Truck	1	540.0
	Frozen Yogurt Shop	1	480.0
4bf58dd8d48988d1e2931735	Art Gallery	19	458.3
	Nightclub	1	441.0
4deefb944765f83613cdba6e	Historic Site	3	182.3

Figure 3 shows a map of the venues by their primary category. The blue marker represents the start and the blue circle represents an area with a 300 m radius. The first and last stop for any loop must be within this circle. From this map we can see there are at least two venues that are in our environment, but cannot be used in any solution. The two most northern, red dots are not reachable by any other venues. We can also see a cluster of art galleries east of our starting location. While these venues are close together giving a high distance reward, loops in this area will likely repeat categories causing their category reward to suffer.

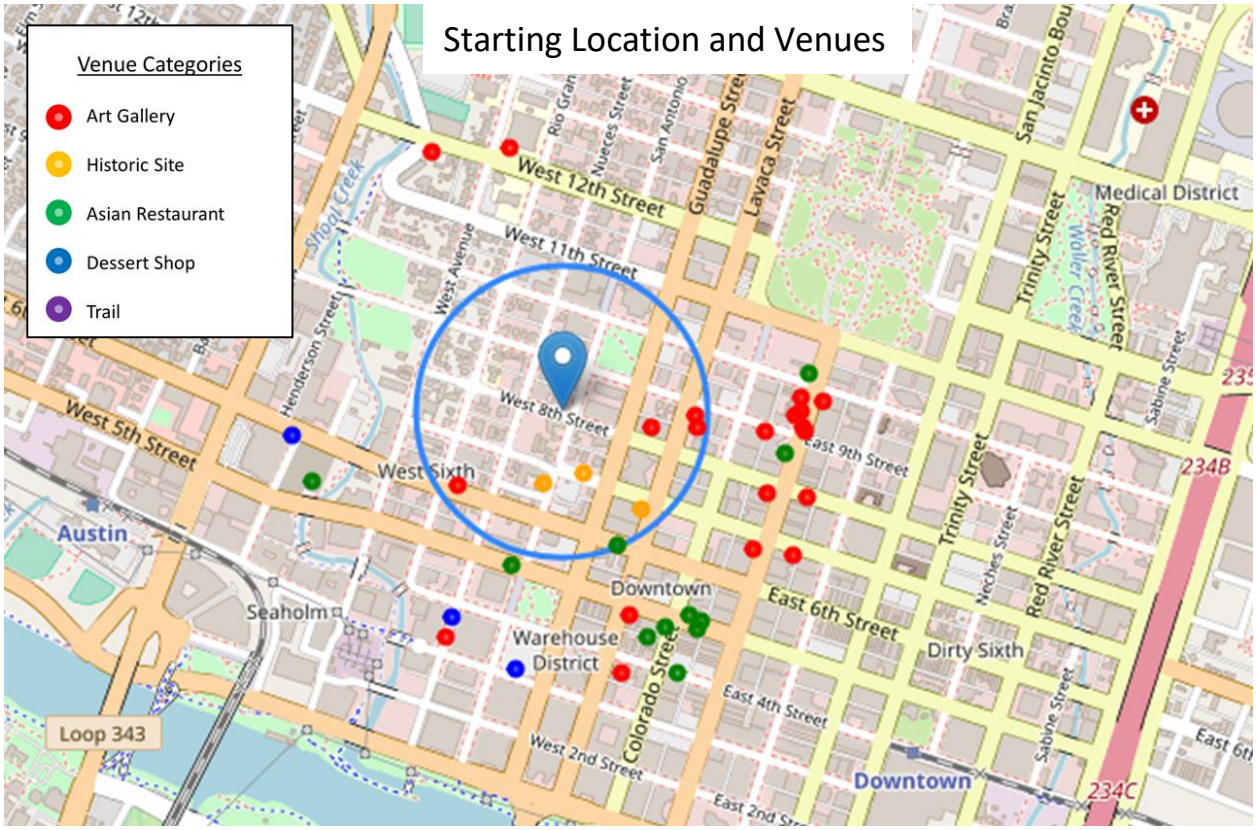

Figure 3: Map of problem environment; i.e. starting location and venues

4.2 ALL SOLUTIONS

Running the function optimalloop for our example case we obtain a distribution of possible values and a count of possible solutions and possible incomplete loops. Table 13 shows the summary statistics of the solutions while Table 14 shown the number of solutions and bad loops. We can see in Table 13 that our values fall within the range of 33.16 - 72.79 with an average of 54.16.

Table 13: Summary statistics for values of all possible solutions

Values Summary for all Possible Solutions	
mean	54.16
std	9.15
min	33.16
25%	48.57
50%	56.00
75%	61.06
max	72.79

In Table 14 we see that there are 1990 possible solutions and 103 possible bad loops giving us 4.92% bad loops. This percentage can be compared to the percent of bad loops for each of the models to help us assess their performance.

Table 14: Count of possible Solutions and possible bad solutions from optimalloop results

Possible Solutions and Bad Solutions	
Possible Solutions	1990
Possible Bad Loops	103
Percent Bad Loops	4.92 %

Figure 4 shows a boxplot and histogram of the possible values. We can see there are two humps in the distribution of values. This may be due to the cluster of art galleries we saw in Figure 3.

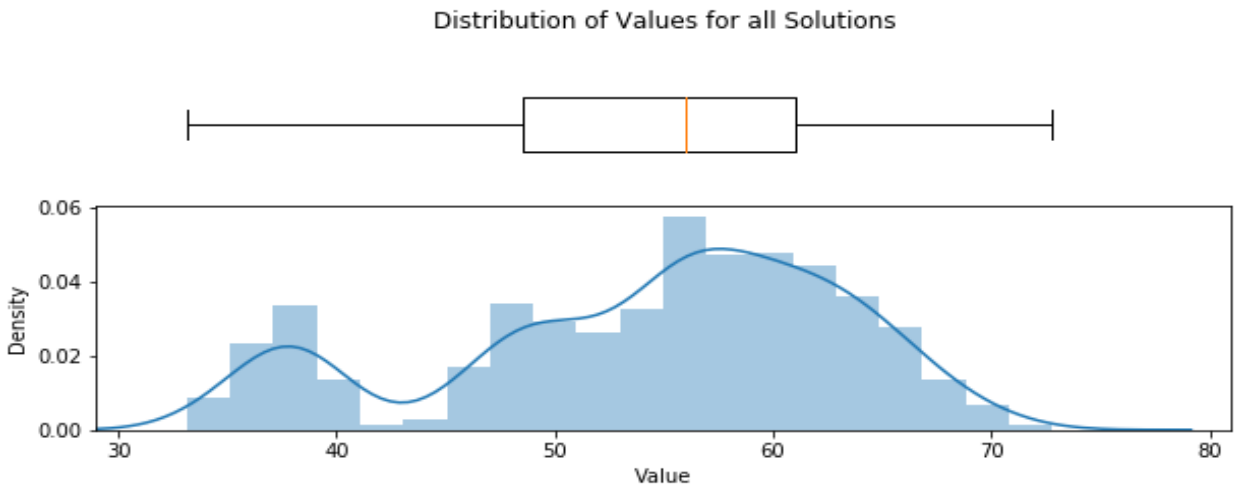

Figure 4: Distribution of values for all possible solutions

Looking at a map of our top 2 solutions as seen in Figure 5 we can see that the best solution and the second best solution are reverse of one another. This is also true for the 3rd and 4th best solutions. We can also see that there are repeated categories on these loops, but the venues are all in out 300 m circle of the start.

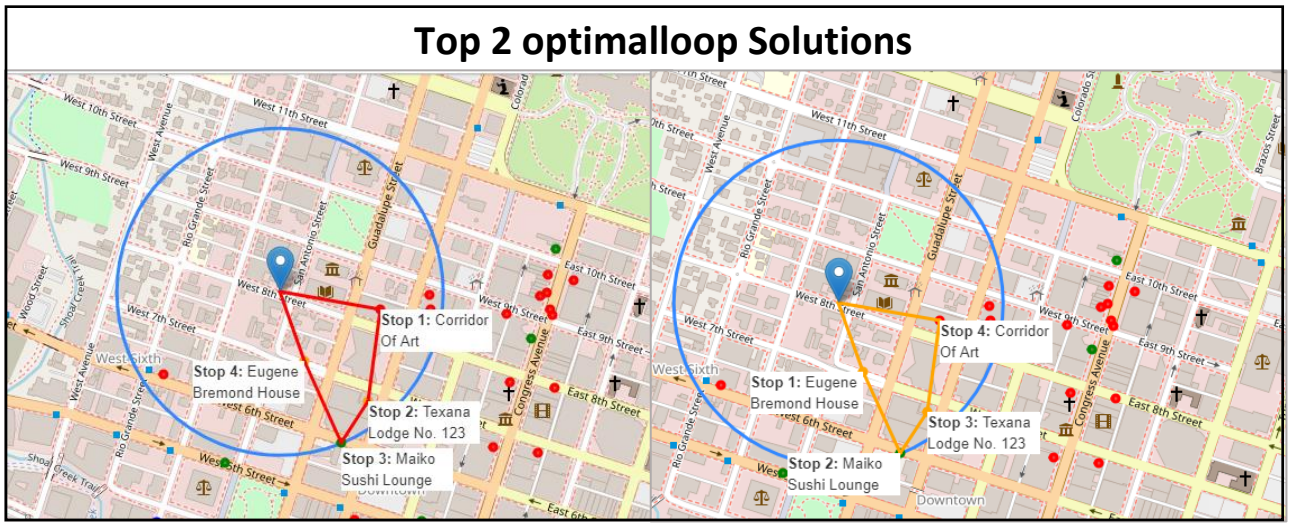

Figure 5: Map of top two solutions from optimalloop function

4.3 RESULTS FOR N_STEPS = 1500

For the second part of our analysis we run the algorithm for 1500 steps 5 times. Each of these 5 sets of data will have approximately 300 solutions.

4.3.1 Values

When we look at the results of our algorithm we are particularly interested in the unique values as the repeated loops do not give us any additional information. In Table 15 we can see the average, standard deviation, highest 3 values and lowest values for each set of unique solutions. For each set the range of values falls within the range of values provided by optimalloop. Sets 4 and 5 have the largest average values, but not necessarily the highest top values. Set 1 has the lowest minimum, but the highest maximum of the 5 sets. For Set 3 our 2nd and 3rd highest results are more than a full standard deviation from the highest value.

Table 15: Values of unique loops for all sets of N_steps=1500 analysis

Values of Unique Loops						
	Mean Value	**Value St.Dev.**	**Top Value**	**2nd Value**	**3rd Value**	**Minimum Value**
Set 1	52.39	11.02	70.49	69.73	67.60	33.28
Set 2	53.29	7.81	67.60	65.36	65.00	37.74
Set 3	53.23	9.28	70.31	66.50	64.96	35.03
Set 4	58.41	8.11	67.67	66.71	66.35	37.12
Set 5	58.45	6.20	70.31	69.85	66.97	38.52

In Figure 6 we have the distribution of values of all solutions. Unlike Table 15, this figure shows the distribution of all values, not just unique values. We can see that sets 4 and 5 have a small inner quartile range and many outliers. Set 3 has the largest IQR but with a much larger range for the 3rd quartile.

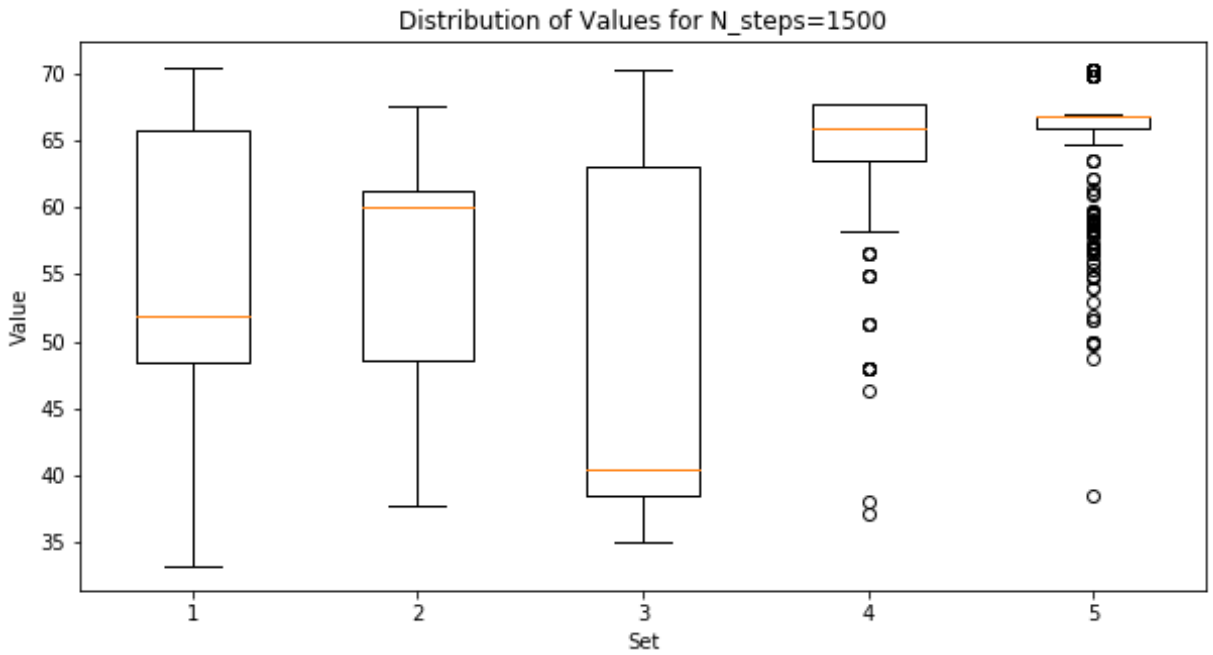

Figure 6: Distribution of values for all sets of N_steps=1500 analysis

Looking at a history of the loops we can get a better idea of the behavior of the algorithm and how it caused the distributions above. Figure 7 marks the value of each loop in the order the algorithm found them. Each color represents a set. Where the markers seem to make a line can be thought to represent the loop that the algorithm "thinks" is the best loop. That loop is repeatedly found because the algorithm is exploiting its knowledge 90% of the time. We can see that sets 4 and 5 started out with high values and only explored lower values occasionally leading to the distributions we saw in Figure 6. It is

worth noting here that while Set 4 started out high and Set 3 started out low, Set 4's maximum value is lower that Set 3's maximum value. We will further explore this behavior later in the report.

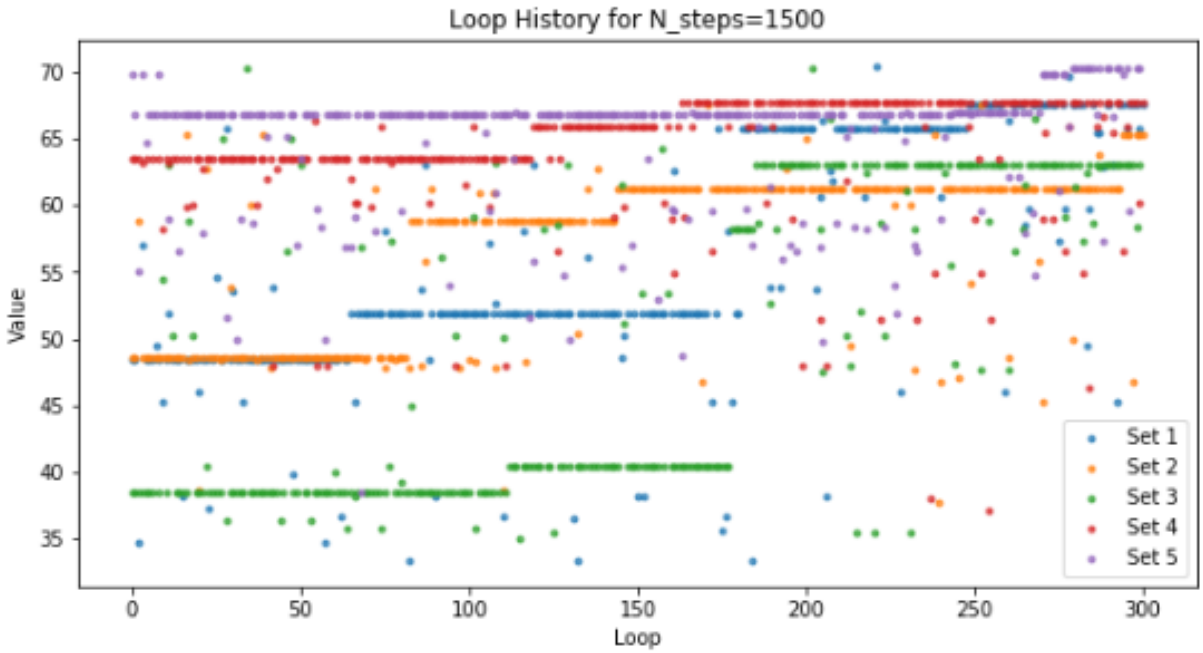

Figure 7: Loop history for all sets of N_steps=1500 analysis

We can also see from Figure 7 that our algorithm finds values higher than that of the loop it "thinks" is best, but it goes back to the loop it was on before rather than switching to the higher valued loop. To better understand this behavior we can look at a portion of the loop history for Set 1. Figure 8 shows the loop history from loop 0 to loop 200 for set 1. The blue dots represent loops and are connected by a blue line to clearly distinguish the order the loops were found. The green markers represent the first time the algorithm has explored a particular loop. We can see that a loop with a value of approximately 48 is the primary loop our algorithm found for the first 60 loops. We can think of this as the loop our algorithm thought was the best. However, at loop 11 the algorithm found a solution with a value of about 53, and at loop 28 it found a loop valued at approximately 66. These loops are labeled with red and yellow squares, respectively. Although these loops are of higher value, the algorithm didn't start primarily choosing those loops. The next red square at loop 65 indicates another instance of finding the loop with value approximately 53. After the third time loop 68, the algorithm changed its behavior to find that loop repeatedly. The same thing can be seen for the solutions marked with the yellow squares at loops 28, 174 and 181. This indicated the algorithm is not learning right away that the higher loop it discovered is the best loop.

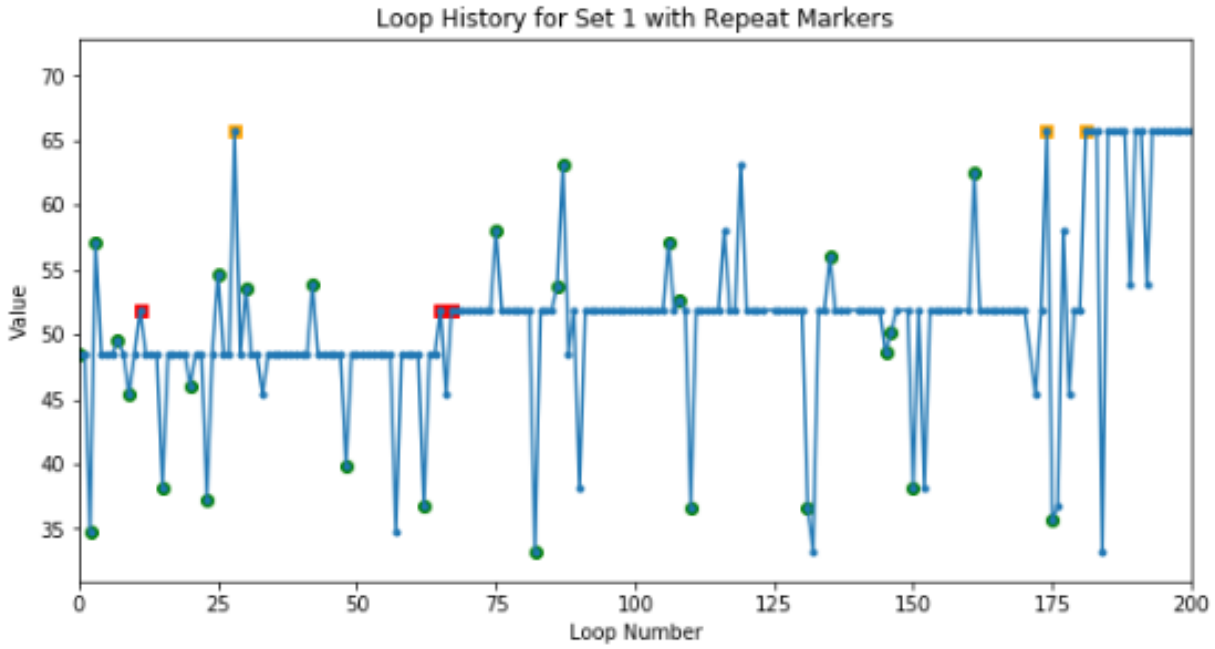

Figure 8: Loop history with repeat markers for Set 1 of N_steps=1500 analysis

4.3.2 Unique Loops

We are interested in the percent of unique loops our algorithm finds because unique loops give us more information about the solution space where as repeated loops just incur cost. Table 16 shows the number of completed loops and the percent repeated and unique loops for each set.

Table 16: Loops, percent repeated loops and percent unique loops for N_steps=1500 analysis

Loops and Unique Loops			
	Total Completed Loops	**Percent Repeated Loops**	**Percent Unique Loops**
Set 1	295	86.44 %	13.56%
Set 2	293	88.74 %	11.26 %
Set 3	296	83.45 %	16.55 %
Set 4	293	91.13 %	8.87 %
Set 5	294	82.99 %	17.01 %

4.3.3 Incomplete Loops

We are also interested in how many times the algorithm gets stuck and cannot finish a loop. Table 17 shows the number of attempted loops (completed loops (L) + bad loops (BL)), the percent bad loops, and the number of repeated bad loops. The percent bad loops for each set is less than the percent bad loops for all possible solutions. We want our algorithm to avoid taking actions that will lead to a bad

loop, so we assign it a large negative reward when it finds a bad loop. However, if the action that led to the bad loop is the only action that has been explored at that state, the reward for that action- although it is negative- is still the highest because we initialized our Q-table with empty Q-values rather than zero Q-values. A bad loop may also be repeated if our random number is less than ε and the action leading to the bad loop is randomly selected.

Table 17: Attempted loops, bad loops and number of repeated bad loops for N_steps=1500 analysis

Attempted Loops and Bad Loops			
	Total Attempted Loops	**Percent Bad Loops**	**Number of Repeated Bad Loops**
Set 1	301	1.99 %	3
Set 2	301	2.66 %	4
Set 3	300	1.33 %	1
Set 4	301	2.66 %	3
Set 5	301	2.33 %	3

In addition to not repeating bad loops, we want our algorithm to learn quickly not to find bad loops. That is, we want the number of bad loops to grow logarithmically rather than linearly. Figure 9 shows the number of bad loops throughout the loop history for each set. We see at the end of Set 2 and the beginning of sets 4 and 5 somewhat logarithmic growth, but we don't see that pattern everywhere.

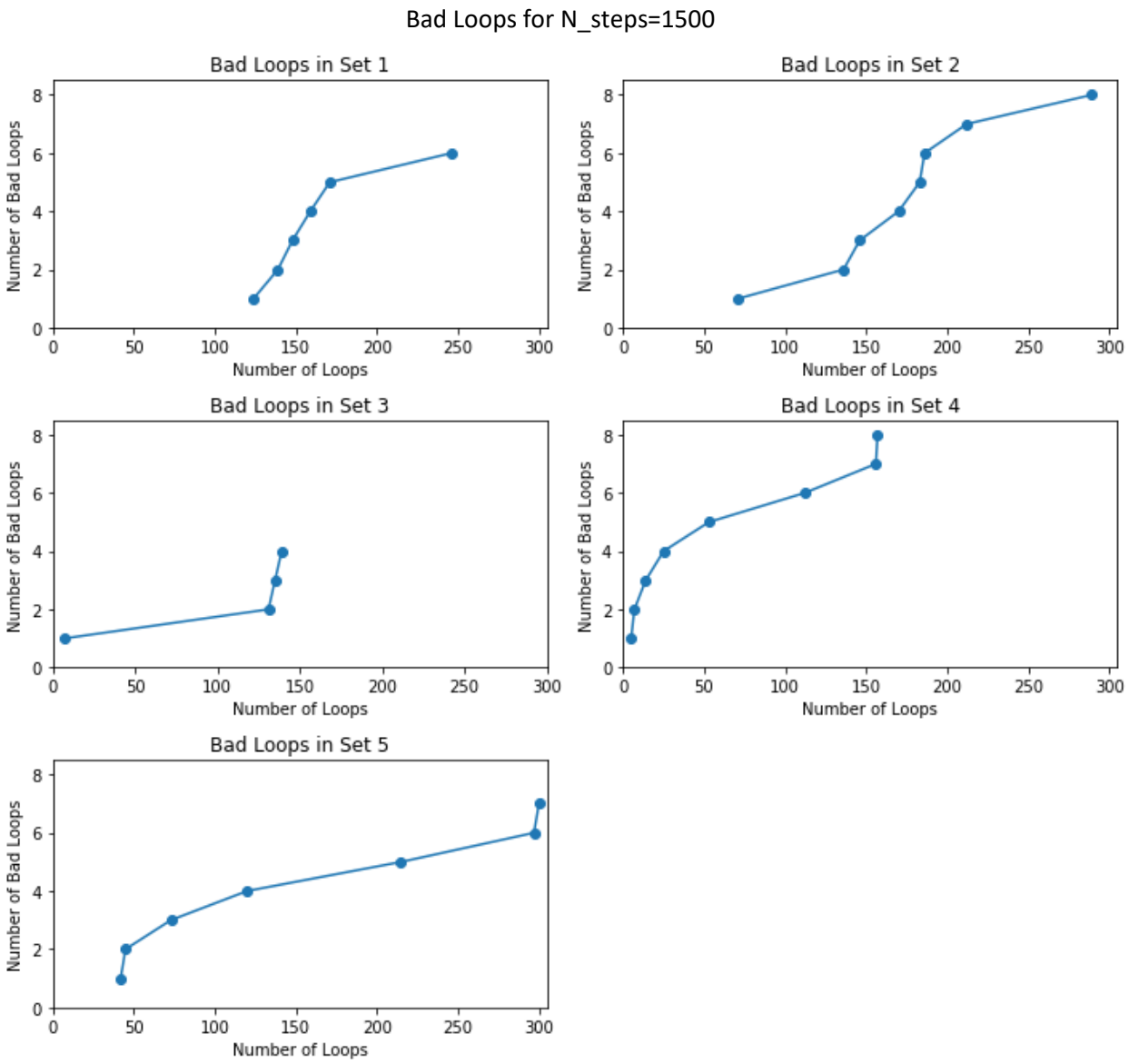

Figure 9: History of bad loops for all sets of N_steps=1500 analysis

To better understand how the number of bad loops grows we will overlay these plots with the loop history of each set, see Figure 10. We can see in sets 2, 4 and 5 where we saw the logarithmic growth, the algorithm is primarily exploring the same loop. When the algorithm moves to a new loop, it is taking a new path through the environment, meaning the actions it needs to avoid to prevent bad loops change.

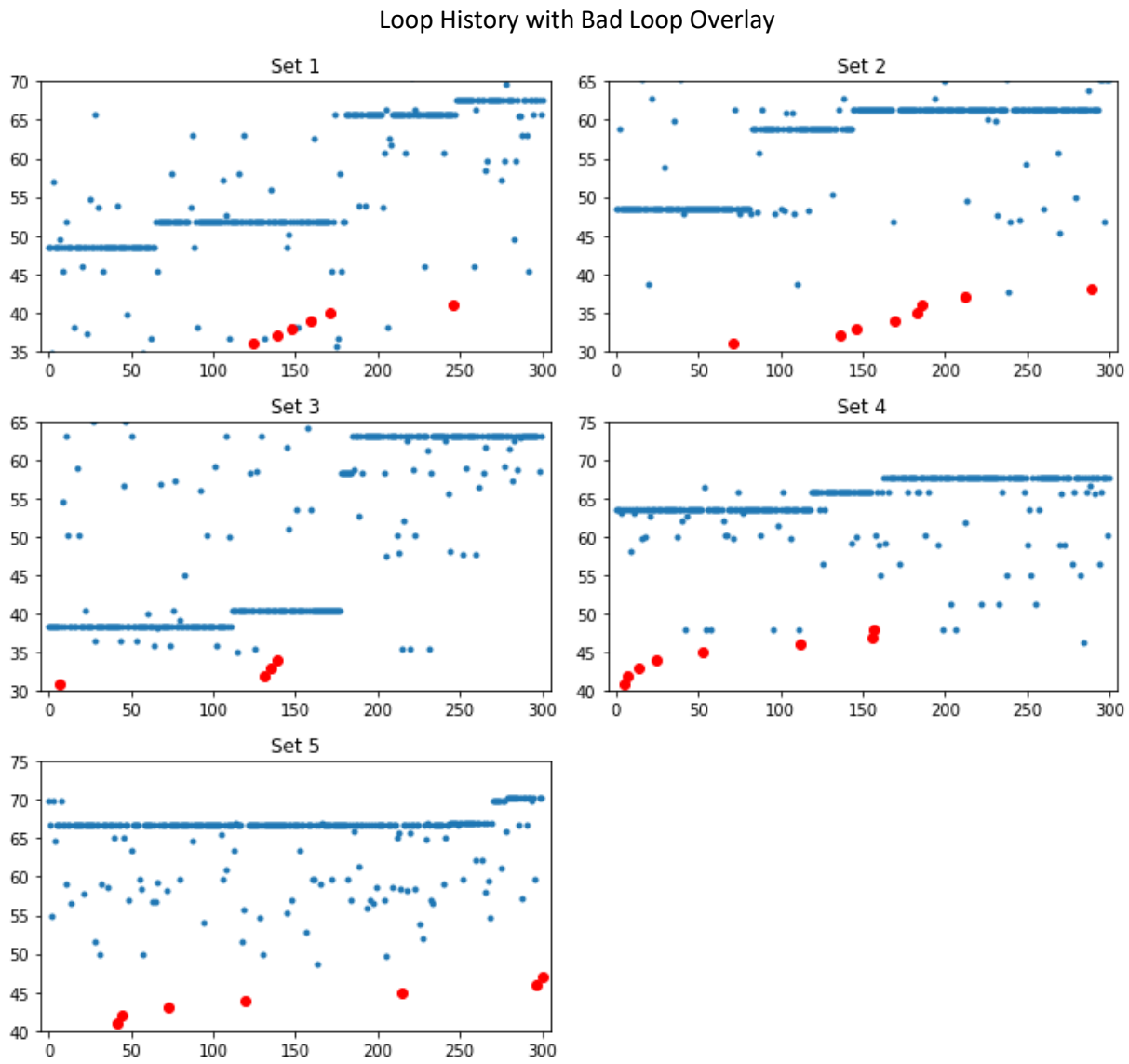

Figure 10: Loop history overlain with bad loops history for all sets of N_steps=1500 analysis

4.4 Results for Running till 95th and 99th Percentile

For our final analysis we will set the algorithm to run until it has found a solution in a given percentile. We will repeat this analysis 30 times for a solution in the 95th percentile and again for the 99th percentile. We will then look at how many solutions the algorithm had to find before finding a "good enough" solution.

4.4.1 95th Percentile

We begin with the 95th percentile. Figure 11 shows the distribution of the number of loops the algorithm found before terminating for each of the 30 trials. Based on this distribution we can assume the algorithm will find a solution in the 95th percentile after about 200 loops. However, there are outliers, one of which is more than three times the number of loops we can confidently say we need.

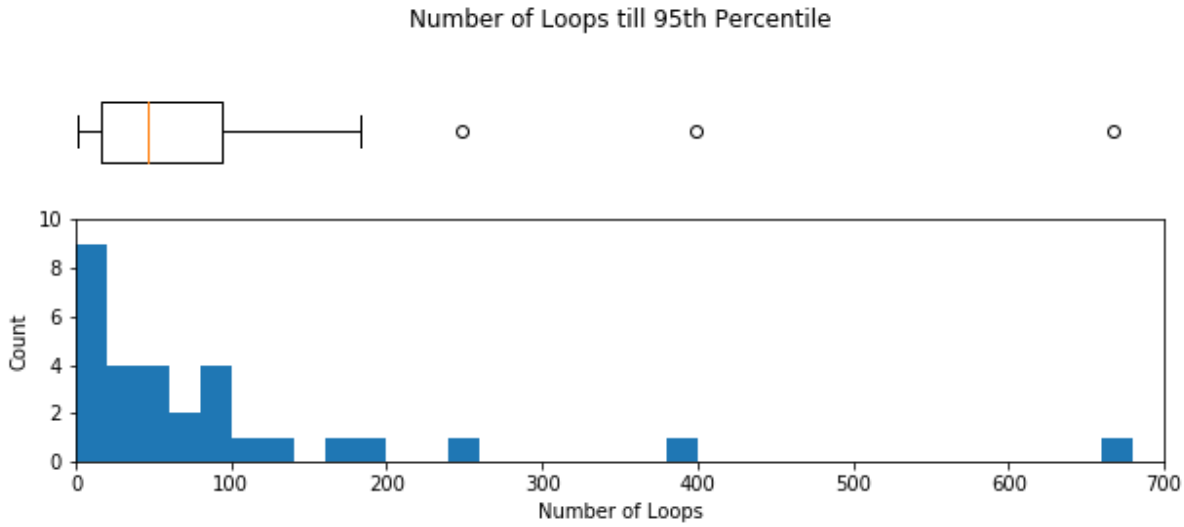

Figure 11: Distribution of number of loops till 95th percentile analysis

4.4.2 99th Percentile

Looking at the 30 trials of running the algorithm till we find a solution in the 99th percentile, we can see a pattern similar to what we saw in the 95th percentile analysis. Generally we can assume the max solution is in the 99th percentile after about 2000 loops; however, there are significant outliers.

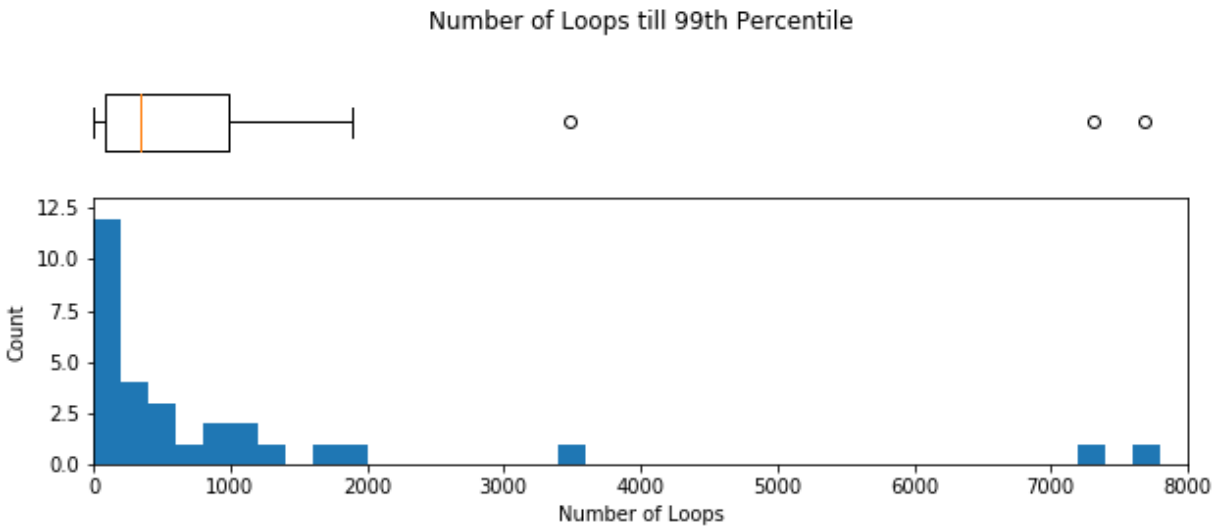

Figure 12: Distribution of number of loops till 99th percentile analysis

To understand what is causing the outliers to have to work so hard to find an acceptable solution, we can look at the loop history of one of the outliers. Figure 13 shows the loops history for Set 30 in the 99th percentile analysis. We see from Figure 13 that the algorithm started with a relatively high valued solution and spent the majority of its time on this solution. This conflicts with our assumption that loops that start off with high rewards result in better loops than ones that start off randomly. We will discuss ways to modify the code to prevent our algorithm from getting "stuck" on a solution.

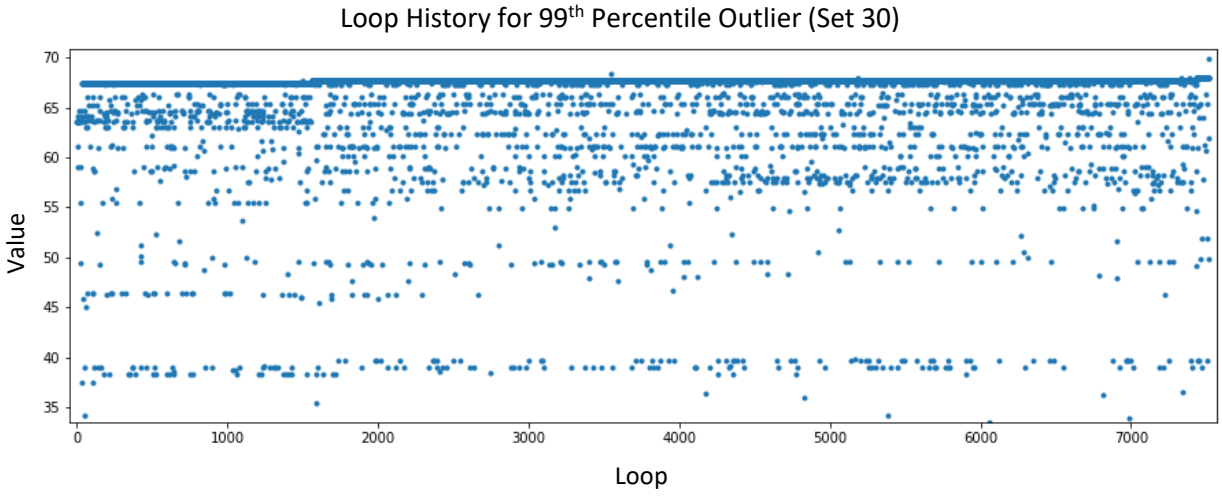

Figure 13: Loop history for outlier (Set 30) in loops till 99th percentile analysis

5 DISCUSSION

5.1 OBSERVATIONS

5.1.1 Environment

There will not always be venues in every category the user selected. Alerting the user when there are no venues of a category they selected can allow them to not only adjust their preferences, but also their expectations. If our example user wanted to visit a trail, but none of the options the algorithm provided included a trail, they might develop an unwarranted negative perception of the application.

Additionally, the Foursquare categories are not always assigned as expected. This is evident from the INFEST night club listed as an art gallery. If the Walking Tour application were to be expanded to look at the venue details, the algorithm may learn to avoid selecting that venue. Alternatively, the user category preferences could be restricted to subcategories.

There are entries in our venues data frame that are not used in the analysis. For more complex problems, the number of unused venues might be large enough to cause an issue. One solution is to remove these venues by evaluating the distances data frame. Another solution is to send the GET request to the Foursquare server each time you are at a new location and storing the available solutions only after visiting that location. While this option would lead to more calls than necessary, it would allow us to store the available venue data for each location in a separate data frame allowing us to remove the getactionchoices function.

5.1.2 Expected Behavior

1.) Values will fall in the range of 30 to 90.

The values for all possible solutions as well as the values for each analysis fell within the range of 30 – 90. There were many solutions closer to 30 whereas the maximum solution was just over 70. Matching our intuition, this indicates it is more likely that we will encounter spaced out venues of the same category than a variety of categories clustered near our starting location.

2.) We will have less than 34.39% unique loops.

The percent unique loops for each case fell around 10% - 20%. This is well below our maximum expected unique loops. To increase the percent unique loops, we can increase epsilon.

3.) The “best loop” will be the highest valued loop the algorithm has previously found.

We saw from the loop history that this is not the case (see Figure 8). There are times when a loop must be found two or three times before it is recognized to be better than the loop the algorithm was previously on. This is because when we update the Q-value, we only update the last Q-value. Let’s say the algorithm takes a familiar path denoted by s0 -> a0i -> s1i -> a1i -> s2i -> a2i -> s3i -> a3i and the Q-values for each state-action pair are called q0i, q1i, q2i and q3i, respectively. Now, let’s assume the algorithm takes a random and new action a4i leading to s4i and q4i and assume that the value of that loop is the highest we have found before. Next time we are at s3i and are trying to make a decision we will look at all the q4 values and see that q4i is the highest. But what if it was at s3i when we took the random action? In that case, we would have updated q4i, but not q3i. Now when we are at s3i, we don’t know that taking a3i will lead to a higher value then taking, say a3j. It isn’t until we find the loop a second time that q3i is updated. For a short application like the Walking Tour, it is likely worthwhile to update all subsequent Q-values after each step. This may not hold with longer problems, but more analysis is needed to better understand how to address this issue.

4.) The algorithm will quickly learn not to take actions that result in an incomplete loop.

While we did see logarithmic growth in the number of bad loops, we saw a repeat of bad loops due to our initialization of q as empty. We could begin by initializing q as 0 so that our negative rewards for bad loops are not seen as the highest, but this would not prevent the algorithm from randomly choosing bad actions. An alternative solution is to store the venue details is separate data frames, where each data frame represents the available actions from that location. In this case, we could delete the action from the data frame preventing the algorithm from ever exploring that loop again.

5.1.3 Additional Observations

The first and second solutions in OLoop (representing the highest and second highest valued loops) were the opposite of one another. This was also true for the third and fourth solutions. If the algorithm finds a solution, the reverse is also a solution because we are creating a loop. Therefore, we can have our algorithm look automatically for the reverse any time it finds a new loop.

5.2 RECOMMENDATIONS

5.2.1 Additional Analysis

The first analysis we should focus on is timing the various processes and functions. This will inform us on where to focus our optimization efforts. If we can optimize the algorithm, we can run more models to better understand the potential of such an application.

The extent of the analysis of the Walking Tour function was very limited. In order to have a better understanding of how the algorithm behaves we need to conduct further analysis. We need to run the algorithm for more cases. That is, different starting locations, categories, numbers of stops and distances between stops. The more cases we run, the more likely we are to run into situations where the algorithm fails. Each time we encounter a situation where the algorithm fails, we can improve it.

An analysis of similar cases should also be conducted. This would be, for example, our same user but visiting 5, 6, 7, 8, etc. stops or willing to travel 400, 500, 600, etc. meters between stops. This analysis will give us an idea for how well the algorithm performs when we change just one input variable.

Resources permitting, when any new case is analyzed, the influence of the parameters on the results should also be analyzed. Meaning, the case should be run for various combinations of ε and γ to see how these parameters affect the results for that case. The more parameter analysis we conduct the better recommendations we can make for parameter selection for more complex applications.

5.2.2 Improvements to Walking Tour

Some recommendation for improving the Walking Tour application that have already been discussed are storing the available actions for each step independently and deleting any action that leads to a bad loop, updating all subsequent Q-values upon learning new information and evaluating the reverse of every newly discovered loop.

The Walking Tour application can be further improved by policy optimization. I.e. implementing a process by which the algorithm searches unexplored territory rather than taking a random action. This change would be implemented in the chooseaction function and would reference the Q-table to identify which actions have not been explored in that state. Additionally, we can set conditions to prevent the algorithm from getting "stuck" on a solution as we saw in the 99th percentile outlier analysis. Possible conditions are temporarily increasing ε or forcing a random/new action at a certain step in the process if the algorithm has found the same solution x number of times.

Additional feedback to the algorithm in the form of user input can further customize the users experience with the application. Implementing user selected parameters that affect the weight of distance and category rewards relative to one another is one example. Another is to allow the user to add a weight to each separate category if they, for example, prefer art galleries to Asian restaurants. Finally, the application can provide the user with a list of venues in the state space and the user can eliminate and/or prioritize venue options either before or while the algorithm runs.

6 CONCLUSION

6.1 REINFORCEMENT LEARNING FOR TRAVEL ITINERARY APPLICATIONS

More analysis is needed to determine whether the proposed simplified reinforcement learning is viable for travel itinerary applications. The computation time using the current code and resources is too long to even allow for sufficient analysis of the rudimentary Walking Tour application. Upon optimizing the code and accessing higher quality computation resources further analysis can be built on the analysis in this report to determine if the optimization reduces the cost enough to make it viable for more complex applications.

An exploration of the parameters ε and γ for various cases should also be conducted to determine if simplified RL is viable for more complex travel itinerary applications. In order for the approach to be considered RL, the algorithm must "learn" how to explore the state space based on feedback from the environment. If further analysis were to find that $\varepsilon = 1$ is the optimal policy, the algorithm would just be randomly searching the space and it would not be RL. It is possible that this is the case since we saw from the outlier analysis that loops that start good do not always lead to better loops. This might, however, not be true for problems with a longer path to a solution.

Finally, the use of alternative RL methods such as the SARSA (state-action-reward-state-action) method should be explored to see if any of the various learning mechanisms improve the algorithm's performance.

6.2 IMPROVEMENTS TO WALKING TOUR

Some recommendation for improving the Walking Tour application are:

- Storing the available actions for each step independently
- Deleting any action that leads to a bad loop from the set of available actions
- Updating all subsequent Q-values upon learning new information
- Evaluating the reverse of every newly discovered loop
- Optimizing the policy to search unexplored territory rather than taking a random action
- Setting conditions to prevent the algorithm from getting "stuck" on a solution
- Implementing additional user preferences and feedback

6.3 Building on Walking Tour

6.3.1 Better Choices

When expanding the Walking Tour algorithm for more complex applications, we should consider more factors to determine the quality of an action. First, we should consider the timing of the stops. This would involve setting a start time, considering actual travel times, business hours, average time spent at a venue and meal times. For actual use it may be necessary to build in the ability to update the itinerary in the application if the user gets off schedule.

Next, we can implement Foursquare's recommendations like those used in the checkin intent such as trending times, popular destinations, and similar user's preferences. We can also allow optional input from the user such as places they know they do or don't want to visit or transportation preferences.

Finally, we should modify the algorithm to create a path rather than just a loop for users who are traveling through a region. Options for the path include designating a start and end point or designating a start point and finding accommodations near the end point

6.3.2 Keeping Expense Down

For the more complex applications the expense of creating an itinerary increases significantly. To keep the cost down the itinerary can be discretized by day or even further. The results of the previous time periods can be inputs for the next implementation of the algorithm. For example, we could have a traveler moving through France who is visiting historic sites and vineyards. Assume the first day there were no vineyards in the area the user was traveling through. The second day, the area had both vineyards and historic sites. Although the algorithm is only looking at the second day, it will still know to prioritize vineyards because on the previous day the user only visited historic sites.

Expanding on the discretization, the user can select the starting and ending locations as inputs to the algorithm to reduce the computational load. These starting and ending points can be hotels, restaurants, or other destinations. If a user knows they want to be at a specific location for lunch, the algorithm can create the itinerary for before lunch and after lunch separately.

More complex travel itinerary applications would be best implemented as a B2B tool for travel agencies first. These professionals have the knowledge to provide more information to the algorithm and interpret the results critically. These businesses can provide valuable feedback to the algorithm and to the application managers to improve the algorithm. A B2C application may be feasible for simpler application like our Walking Tour or after improvement based on business applications.