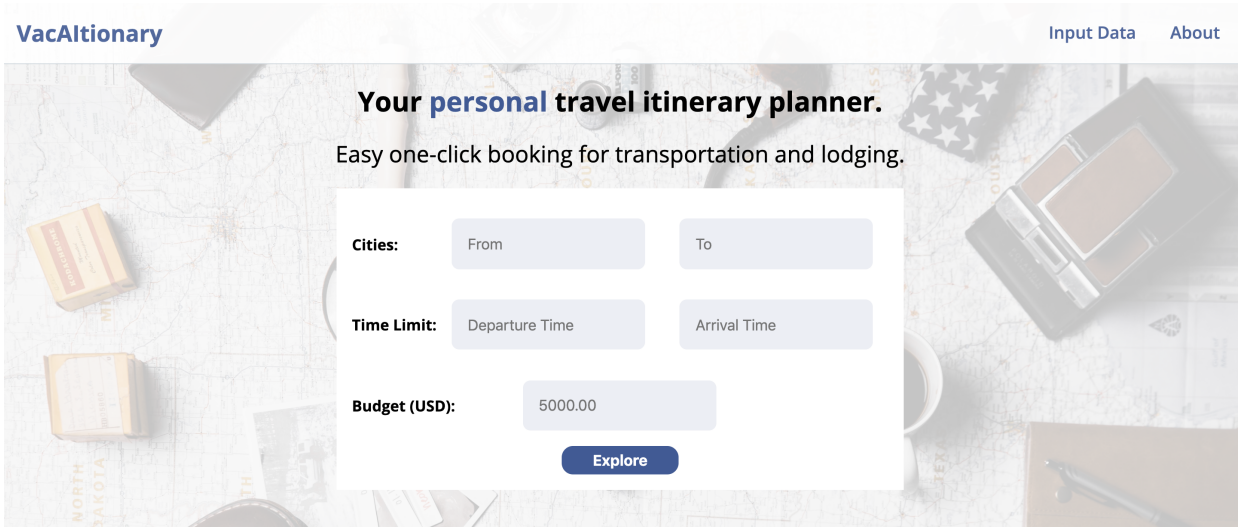
VacAItionary: AI Travel Itinerary Planning Based on Ant Colony Optimization*

Peggy (Yuchun) Wang
Department of Computer Science
Stanford University
peggy.yuchun.wang@cs.stanford.edu

Lauren Zhu
Department of Computer Science
Stanford University
laurenz@cs.stanford.edu



***Abstract*—Itinerary planning is a huge headache for travelers, especially with budget and time constraints. Existing commercial travel planners offer flight planning separately from lodging planning, so users have to manually search and book lodgings after deciding on a flight itinerary. Instead of considering planning for flights and lodgings separately, we generate a travel itinerary based on the best utilities of both. We take into account a combination of flights, cities, and lodgings while adhering to traveler constraints. We show that our approach, based on the Ant Colony Optimization (ACO) algorithm, generates high quality travel itineraries both quantitatively and qualitatively. Specifically, we maximize utility as well as the variety of cities traveled, comparing our results with a Greedy Search baseline. In the future, we plan to expand this project into a product with day-to-day city-level itinerary planning and true booking ability for flights and lodgings.**

I. Introduction

Often times, a group of travelers has a brainstorming list of desired destinations, where some subset of those destinations will become the final itinerary. The problem is that especially with a large initial set of potential destinations, the process of creating the final itinerary can be very time consuming and tedious. To alleviate this burden, VacAItionary is a travel planning optimizer that takes in a set of user inputs for travel plans and outputs several optimal traveling schedule itineraries.

*Github Link to Code: https://github.com/PeggyYuchunWang/vacAItionary

As an example of a use case for our product, consider the following story as told by Lauren: My senior project team was tasked with presenting a software demo in Munich. With a two week available time window of travel, we wanted to travel to as many cities and places as possible that was feasible and reasonable according to the following constraints: time, convenience, price, and personal preference. After many weeks of discussion and quarrel as busy students, we finally scheduled locations, flights, and lodging. However, with access to VacAItionary we could input a list of all our potential destinations and specify that we needed to start from Munich. We would receive a list of itineraries that would meet our time and destination constraints as well as optimize for the parameters we specified. This would have saved us an incredible hassle of narrowing down from the infinite choices that we had, and provide us with an optimal set of itineraries to choose from.

We formulate the travel itinerary problem as an optimization problem that maximizes total utility of an itinerary given traveler constraints of budget, time, and provided start and end cities. This problem can be formulated as an Orienteering Problem, a variant of the Traveling Salesman Problem. We modeled the problem as a graph search problem, where cities are the nodes of the graph, and flights and lodgings are directed edges of the graph. Each edge has a utility based on a weighted combination of price and other attractiveness factors. The output is at most ten top itineraries returned by the algorithm,

consisting of the path, including cities, flights, and lodgings.

To experiment with methods of solving this problem, we implemented both a Greedy Search Algorithm and an Ant Colony Optimization (ACO) algorithm. We conclude that ACO produces better travel itineraries than Greedy, both qualitatively and quantitatively—in terms of average utility and average path length. Considering a case study of ACO and Greedy, we also see more diversity in cities and flight paths with ACO, whereas Greedy tends to have similar flight paths and optimizes for different lodgings. The downside to ACO is that it runs exponentially longer than Greedy, which is a concern if it will be deployed in real systems. However, we could run ACO in parallel, mitigating those concerns.

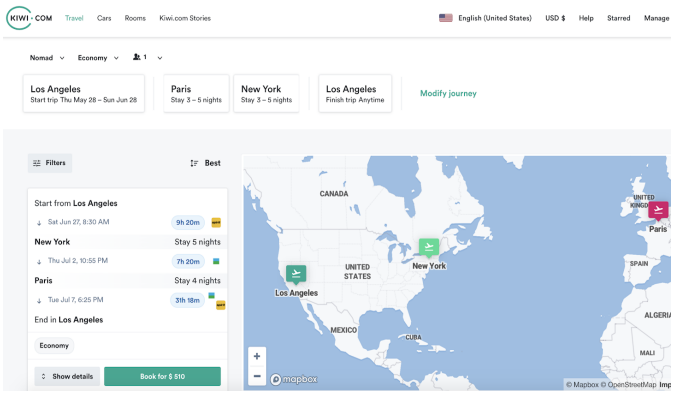
II. Related Work

Several existing commercial travel planners attempt to solve the travel itinerary planning problem, specifically in regards to multi-city flights. Google Trips[1] offers the option to book flights with multiple cities as well as lodging and vacation package recommendations. However, users have to manually specify starting and ending cities as well as dates for each leg of the multi-city flight, and then manually choose a flight out of all possible flights. Furthermore, users have to book lodgings separately from flights, specify additional constraints, and manually pick one lodging out of all possible listings. This process, although perhaps fair for a simple trip, is tedious, manual, and suboptimal when expanded to the task of multi-city travel itinerary planning.

Eighty Days[2] and Kiwi's Nomad Search[3] search for multi-city itineraries in Europe, and output several top itineraries with flight and train transportation between cities as well as the total price. This is almost exactly what we want in a multi-city itinerary planner. However, there are two main limitations with these service-travelers cannot input budget constraints, and neither service takes into consideration lodgings. Instead, these sites link to Booking.com and Airbnb and require travelers to manually book their accommodations after booking their flights. Although these services are an upgrade compared with Google Trips, not being able to search for lodgings while searching for flights and transportation results in suboptimal total costs and still requires manual booking from the traveler.

VacAItionary outputs several optimal multi-city itineraries while taking into account budget constraints as well as lodging options, addressing both the limitations described previously. We note that the formulation of the multi-city travel itinerary problem can be considered to be an Orienteering Problem (OP) [1], a special case of the Traveling Salesman Problem (TSP). OP is a routing problem where the goal is to find the best path (defined as having the maximum score) in a graph, where a subset of nodes are visited and the time limit is not exceeded. Recent variations and applications of the OP are explained in [2], including the Tourist Trip Design Problem. The TSP, OP, and its variants are NP-hard [3]. However, several algorithms give very good approximations to the optimal solution.

[1]https://www.google.com/travel/
[2]https://app.eightydays.me/
[3]https://www.kiwi.com/en/nomad/



An example interaction with Kiwi Nomad demonstrates limitations with travel itinerary planning. Travelers cannot input budget constraints, and must consider lodgings separately and after flights.

Previous work has been done on developing orienteering algorithms that optimize itineraries for multi-day trips based on Google historical visit data and Foursquare data [4]. The authors formulated the problem as a graph search with start and end nodes, duration costs, and time costs. However, this algorithm was applied in the case of multiple day itineraries with points of interest (POI) and attractions within the same city. Although our problem can be formulated similarly, our application is inter-city travel rather than intra-city travel.

An example paper that generates itineraries with multiple cities has been done in [5], where constraints such as time spent on each city and and travel cost are considered. The authors use a genetic algorithm called NSGA-II to find several Pareto optimal travel itineraries consisting of transportation and lodging stays in between cities.

Although we initially considered using the NSGA-II genetic algorithm to generate optimal travel itineraries, we found that ACO is more reliable and more effective than genetic algorithms [6]. Additionally, several examples of ant colony optimization algorithms were used for variants of the Traveling Salesman Problem and Orienteering Problem [7]–[10].

In particular, Yang et al. successfully implemented an ACO algorithm for both inter-city travel and intra-city travel itineraries, where they considered transportation, lodging, and attractions inside cities [10]. Considering the successes of ACO in the previous experiments, and the similarity of problem formulation, we decided to implement ACO for this paper.

III. Approach

This is a unique task with a set of complex data types, so we carefully designed our models, data, and algorithms. This section is split up into four parts. The first is the itinerary planning model, which outlines the different class types and utilities we create for flights, lodgings, and itineraries. The second section discusses why and how we generate data. We then discuss our two algorithmic approaches to optimizing this problem -

a Greedy Search algorithm for a baseline comparison and an improved Ant Colony Optimization Algorithm [11].

We first generated a random set of data points including flight information and lodgings for several destinations in Europe. Then, we ran our algorithms on this data to generate several optimal itineraries, which would ideally resemble a Pareto frontier. Since our utility function is modeled as a weighted average of multiple discrete objectives, an utopia point is often not attainable and optimizing one component typically requires a trade-off in another component. Therefore, our output itineraries will approximate different points along the Pareto frontier. From there the user can choose their favorite itinerary.

A. Itinerary Planning Model Design

We modeled the itinerary object as having its own class. Each itinerary in our implementation is built with several different components: cities, transportation between cities, and lodgings at each city (besides the start and end cities).

Listing 1: Itinerary Class

```
class Itinerary:
    def __init__(self, transportation=[],
        cities=[], lodgings=[], price=0,
        utility=0):
        self.dates = # dates of travel
        self.transportation = #list of flights
        self.lodgings = #list of lodgings
        self.cities = #list of cities
        self.price = #total price
        self.utility = #total utility
```

We built classes for each of these three components as well. Note that in our preliminary implementation, flights are the only form of transportation, although this could be easily expanded to other forms of transportation in the future (trains, buses, etc). A complete itinerary as a search result will have the populated dates of travel, the list of flights taken, the list of lodgings at each destination city, the list of cities visited, the total price of flights and lodging, and the total utility of the trip.

Pseudocode for our itinerary class can be found in Listing 1. Pseudocode for our city, lodging, and flight components can be found in Listing 2.

Listing 2: City, Lodging, and Flight (Transportation) Classes

```
class City:
    def __init__(self, name, score,
        medianStay):

class Lodging:
    def __init__(self, name, address, city,
        prices, datesAvailable, roomType,
        numOccupancy, type, tier):

class Transportation:
    def __init__(self, departDatetime,
        departLoc, arriveDatetime, arriveLoc,
        price):

class Flight(Transportation):
    def __init__(self, departDatetime,
        departLoc, arriveDatetime, arriveLoc,
        price, airline, flightNumber):
```

Because this is a path planning problem, we design what a path looks like within an itinerary. We define a complete path as a repeated sequence of alternating flights and lodgings. To visualize this as a graph traversal, we define the following:

- Cities are nodes
- Flights are directed edges from city X to city Y
- Lodgings are self-loops from city X to city X
- After taking a flight, we must take exactly one self-loop Lodging edge
- We do not visit the same city twice, unless it is the same start and end

Each edge must have some utility, and for consistency we keep all of these values positive. Each edge type (flight or lodging) has a utility function, described below.

1) Flight Utility: In order to design the flight utility function, we take into consideration the features of a flight that would make it a better choice and thus give it a higher utility. Given three factors of a flight—price, destination, and duration—we can intuitively come to the following conclusions:

- Lower price has higher utility
- More desirable arrival destination has higher utility
- Shorter duration has higher utility

We weigh each of these features and sum them to calculate utility $U(f)$ of a flight f. Thus, given price p, destination desirability score s, duration d, and respective flight weights, our utility function is as follows:

$$U(f) = \frac{\omega_p}{p(f)} + \omega_s s(f) + \frac{\omega_d}{d(f)}.$$

We used $\omega_p = 2000$, $\omega_s = 4$, and $\omega_d = 2000$. Notice that we divide by $p(f)$ and $d(f)$ because those features are inversely correlated with utility. The values of ω are chosen in a way that modifies each component to be of a fair fraction of the actual utility. In our case, we assume in this utility function that destination score s is the main driver of a high flight utility, so $\omega_s s(f)$ makes up a larger part of $U(f)$ than the other two (people tend to book trips by destination, not wherever flights are cheapest).

2) Lodging Utility: Similarly to the flight utility function, we must understand which features are directly or inversely correlated with utility. We simplify the problem to ignore the room type, so given the remaining three factors of a lodging—price, tier, and occupancy—we conclude:

- Lower price has higher utility
- Higher tier has higher utility
- Higher occupancy has higher utility

We weigh each of these features and sum them to calculate utility $U(l)$ of a lodging l. Thus, given price p, lodging tier

Table I: Flight Data Example

	Flight 1	Flight 2	Flight 3
Airline	UD	FA	EM
Flight Number	UD4869	FA8591	EM7954
Departure Time	01-17 20:26	01-05 23:04	01-30 15:33
Arrival Time	01-17 21:03	01-06 03:51	01-30 19:20
Departure Loc	Florence	Budapest	Amsterdam
Arrival Loc	Edinburgh	Zurich	Budapest
Price	47.56	314.42	82.59
Travel Time	0:37	4:47	3:46

Each flight has randomly generated strings for the airline and flight number. Random cities are chosen from a set for departure and arrival locations, and a date and random but reasonable duration are chosen for the flight given a time frame.

Table II: Lodging Data Example

	Lodging 1	Lodging 2	Lodging 3
Name	The Respectful	The Stable	The Festive
Address	9235 Zebra Plaza	8613 Rain Bend	207 Table Landing
City	Paris	Barcelona	London
Lodging Type	Hotel	Hostel	Airbnb
Room Type	Suite	Normal Room	Apartment
Occupancy	6	4	6
Tier	4	2	3

Each lodging has randomly generated strings for the name and address. The location is a random city. The room type, availability, occupancy, type, and tiers are also randomly chosen. Each lodging has a set of available dates, each with a different price (not shown).

t, occupancy o, and respective lodging weights, our utility function is as follows:

$$U(l) = \frac{\mu_p}{p(l)} + \mu_t t(l) + \mu_o o(l).$$

We used $\mu_p = 20000$, $\mu_t = 15$, and $\mu_o = 2.5$. Notice again that we divide by $p(l)$ because price is inversely correlated with utility, whereas tier and occupancy are not. The weights μ are chosen to give equal weight to the price and tier, whereas the occupancy or size of the room matters less.

B. Data Generation

Because our focus of the project is to implement Ant Colony Optimization and compare it to our Greedy Baseline algorithm, it made sense to focus our implementation on the algorithms instead of on the data. Using real data entails web scraping for flights and lodgings across many different platforms, so we decided to generate our own data. We do, however, want our simulated data to be as realistic as possible. We take that into consideration during the generation process.

1) Flight Data: We generated 1000 flights, which have a departure and arrival location randomly drawn without replacement from a set of 16 European cities. In our graph, a flight is formulated as a directed edge from one city node to another. The flight object contains an airline, flight number, departure and arrival time, departure and arrival city, price, and travel time. Three sample flights are shown in Table I.

2) Lodging Data: We also generated 200 lodging options in total across the 16 cities. A lodging is formulated in the graph as a self-loop edge at a city node. The lodging object contains a name, address, city, lodging type, room type, occupancy, and tier. It is available on a certain set of dates, each date with a different price associated with it. Three sample lodging options are shown in Table II.

C. Greedy Algorithm

Our Greedy Search algorithm is an original implementation that resembles a greedy version of Dijkstra's algorithm. Specifically, it is an adaptation of breadth-first search that uses a priority queue and greedily builds and sorts itineraries based on utility.

As shown in Algorithm 1, we begin with an empty priority queue and populate it with itineraries that solely consist of all flights leaving from the start city. Those are then appended to the priority queue, giving priority to the flights with higher utility. Until we have the maximum number of itineraries we wish to keep, we continue to greedily prioritize flights and lodgings with higher utility until we have created a full list of complete paths.

D. Ant Colony Optimization Algorithm

We implemented the Ant Colony Optimization algorithm, using the formulation in *Algorithms for Optimization* as a guide [11]. Adaptations were made to generic ACO for the specific problem, including:

Algorithm 1 Greedy Algorithm Pseudocode

```
def greedyBaseline(num_its_keep=10):

   queue = PriorityQueue()
   complete_its = []

   for f in start_flights:
      it = Itinerary()
      queue.put((-f.utility, it)) #higher utility = higher priority

   while not queue.empty() and len(complete_its) < num_its_keep:
      if it.lastcity == endcity:
      complete.append(it)

      next_flights = filterFlights(nextstart, enddate, currcity, it)
      sort(next_flights) #decreasing order sort by utility

      for f in next_flights:
         flight_it = copy(it)
         flight_it.append(f) #add flight to it

         next_lodgings = filterLodgings(currcity, starttime, f.departtime)
         sort(next_lodgings) #decerasing order sort by utility

         for l in nextLodgings:
            lodging_it = copy(flight_it)
            lodging_it.lodgings.append(l) #add lodging to it

            if lodging_it.price <= budget:
               queue.put((-lodging_it.utility, lodging_it))

   return sort(complete) #sort final list by utility and return
```

- Adapting pheromones and priors to fit both flights and lodgings
- Using utility instead of inverse path length
- Ranking top paths
- Traversing a lodging edge after a flight edge, and vice versa (except at the end city)
- Tuning hyperparameters

The full ACO pseudocode is listed in Algorithm 2. The helper functions — *edge_attractiveness()* and *run_ant()* — are listed in the Appendix in Algorithms 3 and 4, respectively.

IV. Experiments

We ran experiments to compare Greedy and ACO both quantitatively and qualitatively. Our ACO Hyperparameters are shown in Table III. Our algorithm inputs include the start city, end city, start time, end time, and budget. We ran both algorithms on all possible pairwise combination of the 16 European cities for the start and end cities, leading to a total of 256 different input combinations. We set our start time to January 03, 2020 and our end time to January 17, 2020, with a budget of 3000.00. We allowed both algorithms to return a maximum of 10 optimal itineraries.

For each of the 16 starting cities, we recorded the average price, utility, path length, and runtime across the 16 ending city combinations in Tables VII and VIII in the Appendix. For clarity, we also averaged these results across all 256 runs for each algorithm.

Table III: ACO Hyperparameters

Ants	Iterations	α	β	ρ
1000	250	1.0	1.1	0.01

V. Results and Discussion

With a total of 16 different starting cities, ACO is able to find better average itineraries (defined as having a higher utility) for 15 of them compared with Greedy. The itinerary prices of ACO are also higher for 15 of the starting cities, and the path lengths of ACO itineraries are consistently longer, meaning that the itineraries visited more cities within the same budget. One drawback of ACO is the runtime, which is significantly slower than our greedy implementation, although runtime may be significantly sped up with running ants in ACO in parallel. These numbers can be found in Tables VII and VIII in the Appendix, and they are further summarized below in Table IV.

Table IV shows our global experimental run across both algorithms. ACO produces itineraries that are, on average, of higher utility and higher quality. That is, it is better optimized for cheaper and shorter flights to desirable cities, as well as

Algorithm 2 Ant Colony Optimization Pseudocode

```
def ant_colony_optimization(num_its_keep=10, num_ants=1000, iters=500, alpha=1.0, beta=1.1,
    rho=0.01):

   start_flights = filterFlights(startdate, enddate, startcity)

   prior = {} #eta
   pheromones = {} #tau
   visited = []

   #initialize prior and pheromones
   for flight in flights:
      pheromones[flight] = 1, prior[flight] = flight.utility
   for lodging in lodgings:
      pheromones[lodging] = 1, prior[lodging] = lodging.aveUtility #ave util over avail days

   top_paths = []

   for i in range(iters):
      A = edge_attractiveness(pheromones, prior, alpha, beta)

      for key, p_val in pheromones: # key is flight/lodging
         pheromones[key] = (1-rho)*p_val

      for ant in range(num_ants):
         it, pheromones = run_ant(pheromones, A)

         if it: # if ant found full path
            if len(top_paths) == num_its_keep and it.utility > top_paths[-1].utility:
               top_paths.pop()
            if len(top_paths) < num_its_keep and it not in visited:
               sort(top_paths.append(it)) #sort by utility
            visited.append(it)

   return top_paths
```

Table IV: Comparison of Greedy Baseline and ACO

	Price	**Utility**	**Path Length** (cities)	**Runtime** (s)
Greedy	2190.64	1244	4.73	0.43
ACO	2444.43	1457	5.30	84.06

*Values above are **averages** across 16 start cities, equivalent to averaging across 256 combinations of start and end city. Note that path length includes the start and end cities.*

cheaper, larger, and more luxurious lodgings all while staying within the budget and time constraints. Regardless of whether the data is real or simulated, it is very unlikely that there is an achievable utopia point. However, ACO is much more likely to find itineraries that lie closer to the Pareto frontier. This is implied from the nature of the utility function designs.

A. Case Study

Because our algorithms optimize numerical metrics, we must manually inspect the itineraries to ensure they are providing better flight and lodging features that are associated with higher quality trips. Our case study will take a closer look at sample algorithmic runs for Greedy and ACO for itineraries from Edinburgh to Venice, with a budget of 4000.00.

We requested the five best itineraries from both Greedy and ACO. The two tables in V clearly illustrate how ACO outperforms Greedy from a utility perspective, especially in this case where a higher budget of 4000 gives ACO more freedom to explore more potential paths (ACO uses the same hyperparameters from the experiments, with a change to 500 iterations). We can see how the Greedy algorithm lacks diversity in its paths because it terminates once it finds a sufficient number of itineraries. ACO, however, runs a specified number of ants for a specified number of iterations.

We now analyze the flights of the best itinerary of Greedy and of ACO in Table VI. Greedy fails to explore the many potential paths that could lead to a diverse itinerary within the given budget. It chooses the highest utility flight as the first leg, and eventually finds a complete path. This eliminates many other paths that would potentially increase utility via later legs or undiscovered lodgings (not shown). This juxtaposition displays how ACO is superior as a direct method that searches across a very large space of paths and still finds high utility itineraries under the budget constraints.

These tables show the flight utility function at play, and

Table V: Greedy vs. ACO Top 5 Itineraries from Edinburgh to Venice

	Path	Price	Utility
Rank 1	E→Ba→A→V	1059.65	**1182**
Rank 2	E→Ba→A→V	1180.65	**1148**
Rank 3	E→Ba→A→V	1346.65	**1144**
Rank 4	E→Ba→A→V	1464.65	**1133**
Rank 5	E→Ba→A→V	1123.65	**1125**

(a) *Top 5 itineraries of **Greedy Baseline**. Notice all the paths are similar, and the utilities similarly low.*

	Path	Price	Utility
Rank 1	E→F→Ba→Bu→A→R→V	2917.62	**2510**
Rank 2	E→F→Ba→A→R→V	3381.27	**2050**
Rank 3	E→Ba→Bu→A→R→V	2665.42	**2049**
Rank 4	E→Ba→Bu→A→R→V	2740.42	**2003**
Rank 5	E→Ba→Bu→A→R→V	3145.42	**1981**

(b) *Top 5 itineraries of **ACO**. The paths are longer and much more diverse, yet have significantly higher utility.*

Table VI: Greedy vs. ACO Best Itinerary from Edinburgh to Venice

Leg	F Price	F City Score	F Duration	F Utility
E→Ba	18.47	55	1:16:00	354
Ba→A	288.94	42	1:41:00	195
A→V	30.24	73	2:06:00	374

(a) *Flights from the best itinerary from **Greedy Baseline**. F refers to flight.*

Leg	F Price	F City Score	F Duration	F Utility
E→F	44.48	51	0:40:00	299
F→Ba	73.05	55	0:30:00	314
Ba→Bu	392.59	65	3:57:00	273
Bu→A	153.82	42	2:41:00	193
A→R	54.48	75	3:47:00	346
R→V	104.20	73	0:50:00	351

(b) *Flights from the best itinerary from **ACO**. F refers to flight.*

demonstrate how flight prices, destination score, and flight duration affect the overall desirability of the flight.

VI. Conclusion

As seen from the results, Ant Colony Optimization produces better itineraries than Greedy Baseline when optimizing over flights and lodging in multiple cities. The itineraries generated by ACO are realistic and diverse, and are a starting point for a one-step itinerary planner for flights and lodging. Additionally, our aim is to eventually deploy this project on the web and include intra-city planning with attractions and restaurants.

A. Future Work

Since we eventually want to turn this project into a product, Peggy made a website mockup using HTML, CSS, and JavaScript (shown as an image in the first page of this paper and also in the Appendix). Users are able to click the "Explore" button and receive a list of example output itineraries.

For the future, our goal is to also take into account traveler preferences (such as their preferred city or type of lodging) as inputs and produce a custom utility function based on those inputs and real-world data. Weights on utility could be formulated via a slider input UI on the importance of certain factors to the traveler.

In a real-world deployment setting, it is likely that running time would matter as much as itinerary quality. One way to speed up ACO would be to implement parallel processing for each ant, which would speed up the running time linearly with the number of cores available. We can also run both ACO and Greedy and pick the best itineraries between both algorithms, or experiment with additional algorithms used for the Traveling Salesman Problem or Orienteering Problem.

To make the system run in real-time, we would also need to incorporate real-time web searching and web APIs of flights and lodgings, which updates by the second. Additionally, we will want to incorporate additional transportation options such as trains and buses and also expand to intra-city planning with day-to-day itineraries incorporating attractions and other experiences. We see a lot of potential in this project going forward, and plan on continuing to work on this project after the end of the class.

Contributions

Peggy and Lauren both implemented class types and built the Greedy Baseline and ACO algorithms for flights and lodgings. This was the bulk of the project.

On an individual level, Peggy ran full experiments for Greedy and ACO results, created the website mockup and design, and did an extensive literature review (the latter two as the additional contribution for 4 units).

Lauren performed the case study, comparing qualitative Greedy and ACO results.

Acknowledgments

Thanks to Professor Mykel Kochenderfer for all his suggestions, help, and inspiration throughout the process, and for his excellent class and teaching! Thanks also to the Stanford CS361/AA222 Course Staff and TAs for all their feedback and encouragement throughout the process.

References

[1] T. Tsiligirides, "Heuristic methods applied to orienteering," *Journal of the Operational Research Society*, vol. 35, no. 9, pp. 797–809, 1984.

[2] A. Gunawan, H. C. Lau, and P. Vansteenwegen, "Orienteering problem: A survey of recent variants, solution approaches and applications," *European Journal of Operational Research*, vol. 255, no. 2, pp. 315–332, 2016.

[3] A. Blum, S. Chawla, D. R. Karger, T. Lane, A. Meyerson, and M. Minkoff, "Approximation algorithms for orienteering and discounted-reward tsp," *SIAM Journal on Computing*, vol. 37, no. 2, pp. 653–670, 2007.

[4] Z. Friggstad, S. Gollapudi, K. Kollias, T. Sarlos, C. Swamy, and A. Tomkins, "Orienteering algorithms for generating travel itineraries," in *Proceedings of the Eleventh ACM International Conference on Web Search and Data Mining*, ser. WSDM '18. New York, NY, USA: Association for Computing Machinery, 2018, p. 180–188. [Online]. Available: https://doi.org/10.1145/3159652.3159697

[5] X. Li, J. Zhou, and X. Zhao, "Travel itinerary problem," *Transportation Research Part B: Methodological*, vol. 91, pp. 332 – 343, 2016. [Online]. Available: http://www.sciencedirect.com/science/article/pii/S0191261515302125

[6] R. Putha, L. Quadrifoglio, and E. Zechman, "Comparing ant colony optimization and genetic algorithm approaches for solving traffic signal coordination under oversaturation conditions," *Computer-Aided Civil and Infrastructure Engineering*, vol. 27, no. 1, pp. 14–28, 2012.

[7] Y.-C. Liang and A. E. Smith, "An ant colony approach to the orienteering problem," *Journal of the Chinese Institute of Industrial Engineers*, vol. 23, no. 5, pp. 403–414, 2006. [Online]. Available: https://doi.org/10.1080/10170660609509336

[8] Z. C. S. S. Hlaing and M. A. Khine, "An ant colony optimization algorithm for solving traveling salesman problem." Fifth Local Conference on Parallel and Soft Computing, 2010.

[9] Y. Chen, W. Sun, and T. Chiang, "Multiobjective orienteering problem with time windows: An ant colony optimization algorithm," in *2015 Conference on Technologies and Applications of Artificial Intelligence (TAAI)*, 2015, pp. 128–135.

[10] L. Yang, R. Zhang, H. Sun, X. Guo, and J. Huai, "A tourist itinerary planning approach based on ant colony algorithm," in *International Conference on Web-Age Information Management*. Springer, 2012, pp. 399–404.

[11] M. J. Kochenderfer and T. A. Wheeler, *Algorithms for optimization*. Mit Press, 2019.

APPENDIX

Table VII: Greedy Results

Start City	Avg Price	Avg Utility	Avg Path Length	Avg Time (s)
Amsterdam	2434.30	1415.82	4.97	0.29
Florence	2398.97	1168.72	4.58	1.07
London	2483.26	1043.55	4.5	1.47
Edinburgh	2042.10	1125.88	4.45	0.16
Rome	2386.23	1158.62	4.7	0.78
Budapest	1792.08	1075.21	4.29	0.61
Prague	2274.38	1445.14	5.02	0.38
Berlin	1939.53	1122.49	4.36	0.14
Munich	2268.60	1542.34	5.24	0.18
Zurich	2166.33	1231.58	4.62	0.42
Barcelona	2052.26	1116.06	4.29	0.26
Venice	2341.30	1283.71	4.95	0.31
Athens	1872.08	1157.99	4.48	0.25
Paris	2160.48	1488.02	5.29	0.31
Nice	2130.97	1175.05	4.84	0.11
Dublin	2307.30	1361.33	5.05	0.21

Table VIII: ACO Results

Start City	Avg Price	Avg Utility	Avg Path Length	Avg Time (s)
Amsterdam	2537.72	1530.75	5.37	99.02
Florence	2400.27	1459.45	5.2	81.97
London	2516.93	1351.66	5.21	71.78
Edinburgh	2502.43	1501.3	5.41	78.05
Rome	2404.84	1499.03	5.53	79.76
Budapest	2366.80	1446.97	5.24	73.2
Prague	2514.56	1574.12	5.42	90.68
Berlin	2530.92	1584.33	5.57	106.41
Munich	2432.66	1535.41	5.31	96.56
Zurich	2444.17	1528.75	5.42	83.88
Barcelona	2282.62	1211.79	4.73	75.73
Venice	2341.20	1283.77	5.05	68.46
Athens	2433.73	1467.31	5.4	68.05
Paris	2464.40	1495.32	5.34	82.13
Nice	2481.98	1374.73	5.24	88.77
Dublin	2455.67	1467.13	5.25	100.49

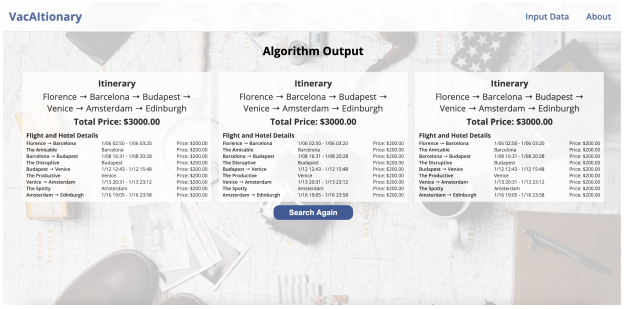


Figure 1: Web Mockup

Algorithm 3 Edge Attractiveness (ACO) Pseudocode

```
def edge_attractiveness(pheromones, prior,
    alpha=1.0, beta=1.1):
   A = {}
   for f in flights:
      v = (pheromones[f]**alpha) *
          (prior[f]**beta)
      A[f] = v
   for l in lodgings:
      v = (pheromones[l]**alpha) *
          (prior[l]**beta)
      A[l] = v
   return A
```

Algorithm 4 Run Ant (ACO) Pseudocode

```
def run_ant(pheromones, A):
   it = Itinerary()
      while len(it.cities) < len(allcities):
         if it.lastcity == endcity: break
         next_flights = filterFlights(nextstart, enddate, currcity, it)

         if len(next_flights) == 0: #ant got stuck
            return None, pheromones

         attracts_f = [A[f] for f in next_flights]
         it.append(np.random.choice(next_flights, 1, attracts_f))

         #find lodging in between two already selected flights
         if len(it.transportation) >= 2:
            next_lodgings = filterLodgings(currcity, last_flight_end, next_flight_start)

            if len(next_lodgings) == 0: #ant got stuck
               return None, pheromones
            attracts_l = [A[l] for l in next_lodgings]
            it.append(np.random.choice(next_lodgings, 1, attracts_l))

      #update pheromones
      for flight in it.transportation:
         pheromones[flight] += 0.001*flight.utility
      for lodging in it.lodgings:
         pheromones[lodging] += 0.001*lodging.utility

      if it.price <= budget:
         return it, pheromones
      else: #over budget
         return None, pheromones
```