On The Town: Social Planning as a Search Problem

Hakeem Angulu, Louie Ayre, and Amadou Camara

December 18, 2018

Contents

1 Introduction

On The Town (OTT) is a web application designed to facilitate event planning between friends. Often, there are a variety of factors that complicate efforts to organize group outings. At what general location should the group meet to minimize travel without unfairly burdening some? What type of venue should be chosen to maximize utility? How much is everyone willing to spend? With *OTT*, friends that are in the same city, but in different locations, can determine a venue that is relatively equidistant from all people, and that also balances each user's preferences for type of activity, budget, and rating.

Given a set of users spread out in various locations and information on venues and activities in a desired destinations, *OTT* optimizes the process of meeting up with those that you care about. The problem of social planning here is well encoded as a search problem. *OTT* uses a variety of search algorithms (including depth-first search, breadth-first search, uniform cost search, greedy search, and A* search) to output a list of 7 places that not only satisfy users' preferences, but are also diverse in their offerings. Thus, *OTT* gives users a useful and varied set of options to execute.

The following is an example of the expected behavior: imagine 6 friends scattered around New York City: uptown, downtown, midtown, etc. Based on the factors they decide, both through polls and preferences within their user profiles, *OTT* minimizes the distance that each will have to travel in order to satisfy their desires, while also maximizing preference satisfaction and maintaining a diverse list of options. Given all of the users preferences, *OTT* outputs 7 locations. Then, the friends meet up and have a fun time!

2 Background and Related Work

Much of the background necessary for this project was gathered from the CS182 course material, including lectures and the textbook [4]. The developers were intrigued by the power of the search algorithms discussed and how different data structures could be utilized to adjust for the specifications of the problem at hand. The developers supplemented their understanding of these concepts with selections from *A Dictionary of Computer Science* [1]. When the complexity of designing an appropriate heuristic for this problem became apparent, *A Formal Basis for the Heuristic Determination of Minimum Cost Paths* was useful for guidance and inspiration [3].

3 Problem Specification

Problem: Given a set of preferences by a group of users, find a set of solutions that satisfies as many preferences as possible while maintaining diversity among the solutions in that set.

OTT encodes this problem as a search problem. As in other search problems, there are a few

parameters, defined for this problem below:

- a set of states: each state in this problem encodes a list of place for the users to visit
- a start state: an empty list
- a goal test: whether or not the length of the state is 7, the desired number of solutions
- a successor function: a function that takes in the current path and returns a subset of all possible next places that are dissimilar from the current state. "Similarity" is defined by the similarity function which checks the attributes (price, rating, etc.) of a group of places and returns a list of places that have sufficiently different attributes. This allows OTT to produce a diverse set of solutions.

4 Approach

The production version of OTT utilizes the A* search algorithm, which was chosen after the rigorous testing described below. The following are all the algorithms considered and studied for OTT:

- Depth-First Search
- Breadth-First Search
- Greedy Search
- Uniform Cost Search
- A* Search

The following are the data structures that support the implementation of these algorithms and *OTT*:

- the *User* class: Each user is initialized with their location and their preferences based on event, price, and ratings. This class has the following objects:
  - *name* (string): The user's name.
  - *location* (tuple): A tuple of latitude and longitude that encodes the user's location.
  - *organizer* (boolean): Whether or not the user is the organizer.
  - *pricePref* (int): The user's price preference as an integer between 0 and 4, inclusive.
  - *ratingPref* (float): The user's preference for a place's rating as a float between 1.0 and 5.0, inclusive.

  - *eventPref* (string): The user's preference for a type of event.

- the *Party* class: This holds the data about a given group of users including their center, a filtered list of places and more used to find the best events. This class has the following objects:
  - *users* (list): List of User objects.
  - *center* (tuple): Tuple of the center of all users' locations.
  - *places* (list): A list of potential places.
  - *findCenter* (method): The true center of geographical longitudes and latitudes.
  - *addToParty* (method): Add a user to the party.
  - *searchLocation* (method): Take the center location and search for all possible events within a given radius.
  - *addPlaces* (method): Iterate through the response data received from API calls and add the formatted places to the list of all the places.
  - *updatePlaces* (method): Iterate through all of the Google types and searches the location for each to give all possible locations in the area.
  - *filterList* (method): Check if events in the list of total places satisfy at least one user's preferences and add those events to a new list.
  - *updateAll* (method): Reset and repopulate the list of places.
  - *getDist* (method): Compare the distance that a user must travel to the location to the distance that a user must travel the geographical center.
  - *sadnessFunction* (method): Evaluate how each user in the party is affected by a specific event's type, price, rating, and distance, based on their preferences.
  - *assignSadness* (method): Assign a sadness list to each event.

- the *Algorithm* class: A container for different types of search algorithms.
  - *type* (string): A string denoting the type of algorithm.
  - *cost_fn* (function): The cost function for the algorithm.
  - *heuristic* (function): The heuristic used for A* search.
  - *dfsSearch* (method): The DFS algorithm.
  - *bfsSearch* (method): The BFS algorithm.
  - *greedySearch* (method): The Greedy Search algorithm.
  - *astarSearch* (method): The A* Search algorithm.
  - *ucsSearch* (method): The UCS algorithm.

- the *Stack* class: A container with a last-in-first-out (LIFO) queuing policy.
- the *Queue* class: A container with a first-in-first-out (FIFO) queuing policy.
- the *PriorityQueue* class: Implements a priority queue data structure.
- the *PriorityQueueWithFunction* class: Implements a priority queue with the same push/pop signature of the *Queue* and the *Stack* classes.

The algorithms are discussed in detail below.

Depth-First Search

Depth-First Search (DFS) performs a search of a directed graph (or tree) by starting at the root node (or selecting some arbitrary node as the root node in the case of a graph) and exploring as far as possible along each branch before backtracking.[2]

An initial starting vertex, $u = []$, is selected, its successors are generated and added to the frontier, and then it is "visited." The visit entails checking if the goal (a list of 7 solutions) has been met and logging that visit in a list. Then, each of the successors in the frontier is visited, and their successors are added to the frontier after each visit. In order to check all nodes at a certain branch, the frontier is traversed such that the last nodes added are the first ones removed (last-in first-out, FIFO), and thus encoded as a stack. The search terminates when the length of a node reaches 7.

The time complexity of DFS is $O(b^m) = O(b^7)$ and the space complexity is $O(b * m) = O(b)$, where b is the branching factor (the average number of successors per state) and $m = 7$ is the max depth.

The algorithm is described with the Pythonic pseudocode below in *Algorithm 1*, with the data structure, *dstruct*, instantiated as a *Stack* object.

Breadth-First Search

Breadth-First Search (BFS) performs a search of a directed graph (or tree) by starting at the root node and searching through the structure at level k before proceeding to nodes at level $k + 1$.[1]

An initial starting vertex, $u = []$, is selected, its successors are generated and added to the frontier, and then it is "visited." The visit entails checking if the goal (a list of 7 solutions) has been met and logging that visit in a list. Then, each of the successors in the frontier is visited, and their successors are added to the frontier after each visit. In order to check all nodes at a certain depth, the frontier is traversed such that the first nodes added are the first ones removed (first-in first-out, FIFO), and thus encoded as a queue. The search terminates when the length of a node reaches 7.

The time complexity of BFS is $O(b^m)$ and the space complexity is $O(b^m) = O(b^7)$, where b is the branching factor (the average number of successors per state), and $m = 7$ is the max depth.

The algorithm is described with the Pythonic pseudocode below in *Algorithm 1*, with the data structure, *dstruct*, instantiated as a *Queue* object.

Greedy Search

Greedy search performs a search of a directed graph (or tree) by starting at the root node and searching through the structure by visiting the nodes with the least value for the heuristic, $h(x)$, where x is the successor node. For *OTT*, the heuristic is described below in *Algorithm 2*.

An initial starting vertex, $u = []$, is selected, its successors are generated and added to the frontier, and then it is "visited." The visit entails checking if the goal (a list of 7 solutions) has been met and logging that visit in a list. Then, each of the successors in the frontier is visited, and their successors are added to the frontier after each visit. In order to check the nodes that have the least value for the heuristic, the frontier is a priority queue where the priority is the value of the heuristic, and thus the node with the lowest value is removed first. The search terminates when the length of a node reaches 7.

The time and space complexities of Greedy Search are determined by the heuristic, and thus cannot be reported with complete accuracy. Based on *OTT*'s heuristic, the estimate of the upper bound of complexity for both time and space is $O(b^d) = O(b)$, where b is the branching factor (the average number of successors per state), and d is the depth of the solution $= 7$.

The algorithm is described with the Pythonic pseudocode below in *Algorithm 1*, with the data structure, *dstruct*, instantiated as a *PriorityQueue* object. The push method of this object also takes an extra parameter, $h(x)$, the value of the heuristic at node x.

Uniform Cost Search

Uniform cost search performs a search of a directed graph (or tree) by starting at the root node and searching through the structure by visiting the nodes with the least value for the cost function, $c(x)$, where x is the successor node. For *OTT*, the cost function is the sadness function described below, and in *Algorithm 3*.

An initial starting vertex, $u = []$, is selected, its successors are generated and added to the frontier, and then it is "visited." The visit entails checking if the goal (a list of 7 solutions) has been met and logging that visit in a list. Then, each of the successors in the frontier is visited, and their successors are added to the frontier after each visit. In order to check the nodes that have the least value for the cost function, the frontier is a priority queue where the priority is the value of the cost function, and thus the node with the lowest cost is removed first. The search terminates when the length of a node reaches 7.

The time complexity of UCS is $O(b^{c*/\epsilon})$ and the space complexity is $O(b^{c*/\epsilon})$, where b is the branching factor (the average number of successors per state), $c * /\epsilon$ is the depth ($= 7$), and $c*$ is the cost.

The algorithm is described with the Pythonic pseudocode below in *Algorithm 1*, with the data structure, *dstruct*, instantiated as a *PriorityQueue* object. The push method of this object also takes an extra parameter, $c(x)$, the value of the cost function at node x.

A* Search

A* search performs a search of a directed graph (or tree) by starting at the root node and searching through the structure by visiting the nodes with the least value for the sum of the cost and heuristic functions, $h(x) + c(x)$, where x is the successor node. For *OTT*, the cost function is the sadness

function described below and in *Algorithm 3*, and the heuristic function is the function described below in *Algorithm 2*.

An initial starting vertex, $u = []$, is selected, its successors are generated and added to the frontier, and then it is "visited." The visit entails checking if the goal (a list of 7 solutions) has been met and logging that visit in a list. Then, each of the successors in the frontier is visited, and their successors are added to the frontier after each visit. In order to check the nodes that have the least value for $f(x) = h(x) + c(x)$, the frontier is a priority queue where the priority is the value, $f(x)$, and thus the node with the lowest sum of cost and heuristic is removed first. The search terminates when the length of a node reaches 7.

The time and space complexities of A* Search are determined by the heuristic (and the cost function), and thus cannot be reported with complete accuracy. Based on *OTT*'s heuristic, the estimate of the upper bound of complexity for both time and space is $O(b^d)$. In the worst case of an unbounded search space, the number of nodes expanded is exponential in the depth of the solution (the shortest path) $d = 7$: $O(b * d) = O(b)$, where b is the branching factor (the average number of successors per state).[4]

The algorithm is described with the Pythonic pseudocode below in *Algorithm 1*, with the data structure, *dstruct*, instantiated as a *PriorityQueue* object. The push method of this object also takes an extra parameter, $f(x)$, the value of the sum of the cost and heuristic functions at node x. The cost and heuristic functions are described below, and in *Algorithm 2* and *Algorithm 3*.

Evaluation and Comparison

The five algorithms above were run on random samples of parties to test their validity for this problem. At each step of these simulations, the average of the total sadness generated from the list of solutions served as a proxy for how well the algorithm did: the lower the average sadness, the closer the solutions were to satisfying all conditions, and the more satisfied the users. Another metric that was important was the time each algorithm takes to generate solutions, and that was also taken into account. Both are discussed below in the **Results** section.

Sadness Function

The cost function for the UCS and A* Search algorithms was the *OTT sadnessFunction* function. *sadnessFunction* takes an *event* and a *party* object, and assigns a cost, or "sadness," for each user by taking a weighted combination of multiple measurements of its satisfaction of the user's preferences. The weights are:

- the additional distance required for the user to travel with respect to the distance to center point of the party
- the difference in price between the user's budget and the venue's price rating
- the difference in rating between the user's preferred rating and the venue's rating

- whether or not the type of the venue is one of the user's preferred venue types

These weights were empirically determined by simulating 500 random parties and averaging the *sadness* attributed to each user for each event by the distance, price, rating, and type respectively. The resulting values were used to adjust the weights of each factor such that the inputs were comparable.

The final sadness for a state, party configuration is the sum of each user's sadness, plus the sadness of the least satisfied user (the user with the highest sadness) to account for *OTT*'s goal of minimizing total dissatisfaction while ensuring that everyone in the party is at a reasonable threshold of satisfaction.

Algorithm 1 Search Algorithm

```
procedure SEARCH(party)
    visited ← []                                   ▷ initialize variables to store the visited states and cost
    cost ← 0
    filteredList ← party.filteredPlaces            ▷ the list of filtered places from party class
    path ← []                                      ▷ the starting path is an empty list
    dstruct ← Stack()                              ▷ instantiate a frontier object
    dstruct.push((path, filteredList, cost))       ▷ push the state to the frontier
    while not dstruct.isEmpty do
        path, remaining, cost ← dstruct.pop()      ▷ get the next state from the frontier
        if len(path) == 7 then
            return path, cost                      ▷ check if at the goal
        else
            event ← path[−1]                       ▷ if the list is not empty get the last element
            if event not in visited then
                visited.append(event)              ▷ add the state to the visited list
                nextList ← similarity(event, remaining)   ▷ find the successors of the state
                for successor in nextList do       ▷ iterate through all successors
                    sadness ← successor["sadness"]   ▷ extract the sadness (cost) from the event
                    nextCost ← sum(sadness) + max(sadness)   ▷ compute the cost
                    dstruct.push((path + [successor], nextList, cost + nextCost))   ▷ push the
successor to the frontier
                                                   ▷ if there are no events in the path
                for successor in filteredList do
                    sadness ← successor["sadness"]   ▷ extract the sadness (cost proxy) from the
event
                    nextCost ← sum(sadness) + max(sadness)   ▷ compute the cost
                    newList ← copy.deepcopy(filteredList)   ▷ copy the list into a new structure
                    newList.remove(successor)      ▷ remove the successor from the list
                    dstructk.push((path + [successor], nextList, cost + nextCost))   ▷ push the
successor to the frontier
```

Algorithm 2 Search Heuristic

procedure HEURISTIC(*state*)
progress ← len(*state*[0]) ▷ encode the progress to the goal as the length of the list of previous states
if *progress* ≠ 0) **then** ▷ Goal distance is the number of places left to be added multiplied by the average of the sadness value to the current point
goal_distance = (7 − *progress*) ∗ *state*[2] / *progress*
else
goal_distance = 7
return: *goal_distance*

Algorithm 3 Cost Function: Sadness

procedure SADNESSFUNCTION(*place*)
weight ← {"*type*" : 0.319765608129, "*price*" : 0.170200961101, "*rating*" : 0.130747631926, "*dist*" : 0.0156106747366} ▷ weight the contributing factors to sadness
sadness = [0]∗ len(*self.users*) ▷ initialize the sadness list
for *i* in range(len(*self.users*)) **do**: ▷ iterate through user list to calculate their sadness
dist ←self.*getDist*(self.users[i],place)
if *dist* > 0 **then**: ▷ difference between distance to place and center
sadness[*i*]+ = *dist* ∗ *weights*["*dist*"] / *normalizers*["*dist*"]
rating ←self.users[i].ratingPref - place["rating"]
if *rating* > 0 **then**: ▷ difference between rating of place and rating preference
sadness[*i*]+ = *rating* ∗ *weights*["*rating*"] / *normalizers*["*rating*"]
price ← *float*(place["price"] - self.users[i].pricePref)
if *price* > 0 **then**: ▷ difference between price of place and price preference
sadness[*i*]+ = *price* ∗ *weights*["*price*"] / *normalizers*["*price*"]
type ← 0
for *event* in *self.users*[*i*].*eventPref* **do**:
if *event* not in *place*["*types*"] **then**: ▷ check if the type of the place is preferred
type+ = 1
if *type* == len(*self.users*[*i*].*eventPref*) **then**:
type = 1
else
type = 0
return: *sadness* ▷ return a list of the sadness incurred by a place for each user in the party
end procedure =0

5 Experiments

The algorithms take in a *Party* object, which holds the data about a given group of users including their center, a filtered list of possible places, and preferences. In order to test the implementation

and performance of the algorithms, data were generated to simulate many parties. 900 parties were randomly created using a random party generator, described in *Algorithm 4* below.

The following are the main aspects of a party that the algorithm creates:

- location data for each user, randomly generated from a list of possible coordinates in New York City. New York City was used because it was the impetus for this project, and it has a wide variety of available places
- activity type preference data for each user, randomly generated from a subset of the available activities Google encodes for the Google Places API
- rating preference data for each user, randomly generated from a list of 1 to 4
- price preference data for each user, randomly generated from a list of 1 to 5
- name, randomly created

These random processes allowed the developers to test a wide range of possibility automatically, streamlining assessment of both edge and general cases. Once these parties were created, each of the 5 algorithms was run on them, and the following data for each were collected:

- failure rate: the rate of failure of the algorithm, where failure is defined as taking more than 1 minute to reach a solution, or returning an empty list (which was uncommon, but possible in *BFS* and *DFS* due to the design of the successor function
- evaluation time: the time it takes an algorithm to reach a solution
- the average sadness: the average sadness that a solution gave to the party to which it applies

All these data were saved in `simulated_results.csv`, and the entire process can be replicated using the `evaluation.ipynb` Jupyter notebook. The analysis of the data lies in `analysis.ipynb`, and the major takeaways are reported below.

The evaluation and testing processes revealed a lot about the implementation, and a few changes had to be made along the way to handle edge cases the developers caught (for example, the case of returning no solutions, or the very long evaluation time of *BFS*).

5.1 Results

For algorithm-comparison projects: a section reporting empirical comparison results preferably presented graphically.

Algorithm 4 Random Party Generator

```
procedure RANDOMPARTY(num)
    coordList ← [NYcoords]                          ▷ create list of various coordinates in NY city
    typeList ← [types]                              ▷ create list of various type preferences
    party = Party()                                 ▷ Initialize party
    Coords ← []                                     ▷ Initialize empty list to hold party's coordinates
    for i in range(num) do
        coord = coordlist[np.random.randint(0,len(coordlist))]   ▷ Get a user's unique coordinates
        while coord in coords do
            coords = coordlist[np.random.randint(0,len(coordlist))]
        coords.append(coord)
        username = "User"+str(i)                    ▷ Assign a unique user name
        rating = 2 + 3∗np.random.random_sample()    ▷ Get rating preference, continuous [2, 5]
        price = np.random.randint(1, 5)             ▷ Get price preference, discrete [1, 4]
        types = np.random.choice(typelist, size = np.random.randint(1, 4),replace = False)
                                                    ▷ Get 1-3 type preferences
        user = c.User(username, coord[0], coord[1], price, rating, types)   ▷ Create user
        party.addToParty(user)                      ▷ Add user to party
    party.updateAll()                               ▷ Update Party
    return party
```

The following outline the results of the tests described above. All numbers are given to three decimal places of precision.

Failure Rate

Table 1 describes the percentage of times the algorithms failed, where failure is defined as returning an empty list of solutions or taking more than 1 minute to run. *Figure 1* shows the graphical representation of this data. It is clear that *BFS* is nearly unfeasible as an algorithm for this project, failing nearly 90% of the time. With more testing, the developers found that *BFS* could sometimes take over 5 hours to find a solution for a party of 6. On the other hand, *Greedy Search* performs the best, with a failure rate of only 14.889%.

Algorithm	Failure Rate (%)
DFS	32.000
BFS	88.889
Greedy Search	14.889
UCS	39.667
A* Search	15.889

Table 1: Failure Rate of Algorithms

Evaluation Time

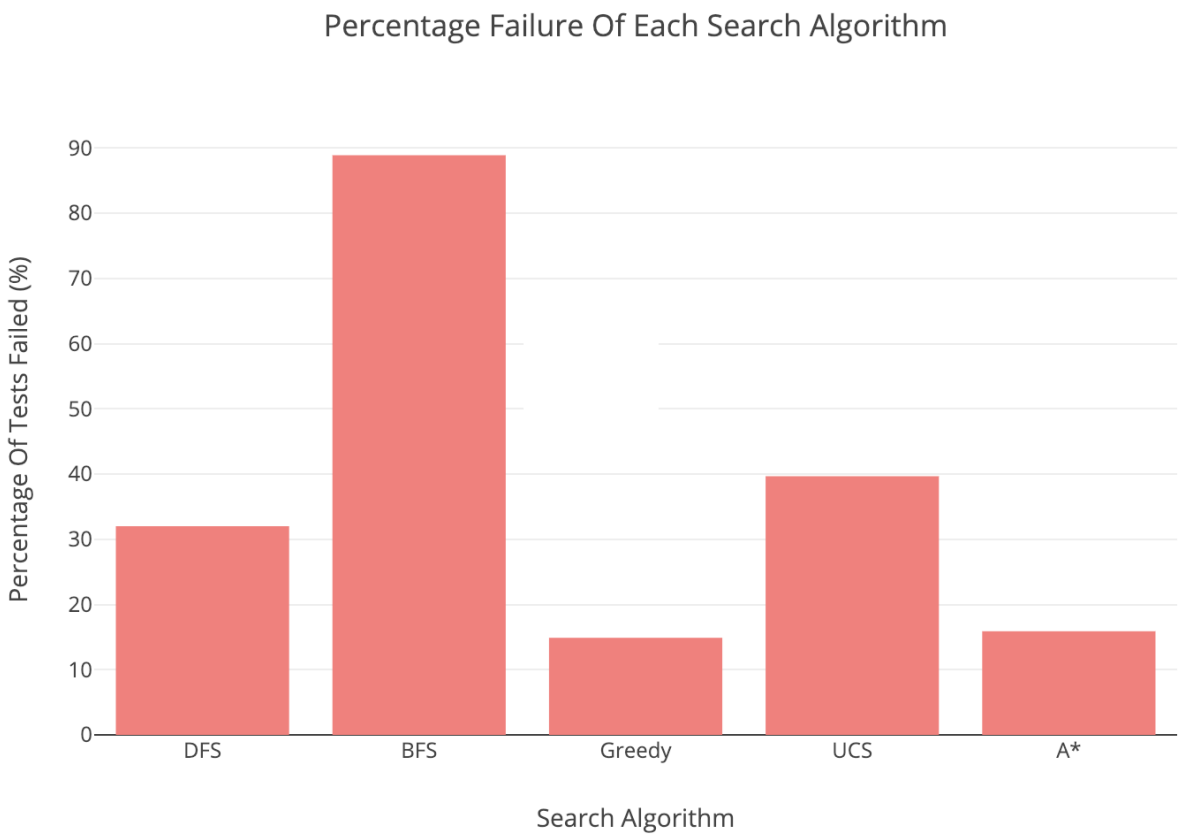

Figure 1: Percentage Failure of Each Search Algorithm

Table 2 describes the average time in seconds that each algorithm took to find a solution when a solution was found. It is important to note that the times do not reflect the failure rate described above, and thus, the two should be interpreted together. *Figure 2* shows the graphical representation of this data. To give further context to these point estimates, *Figure 3* shows the distribution of the evaluation time as histograms. From these graphs, one can see that again, *Greedy Search* performs remarkably well, and is the second fastest algorithm after *DFS*, but with much more consistency towards good performance, as evidenced by the extreme right skew of the data. On the other hand, *BFS* and *UCS* are some of the worst performers.

Algorithm	Average Evaluation Time (seconds)
DFS	0.289
BFS	10.948
Greedy Search	1.047
UCS	15.108
A* Search	3.0981

Table 2: Average Evaluation Time of Algorithms

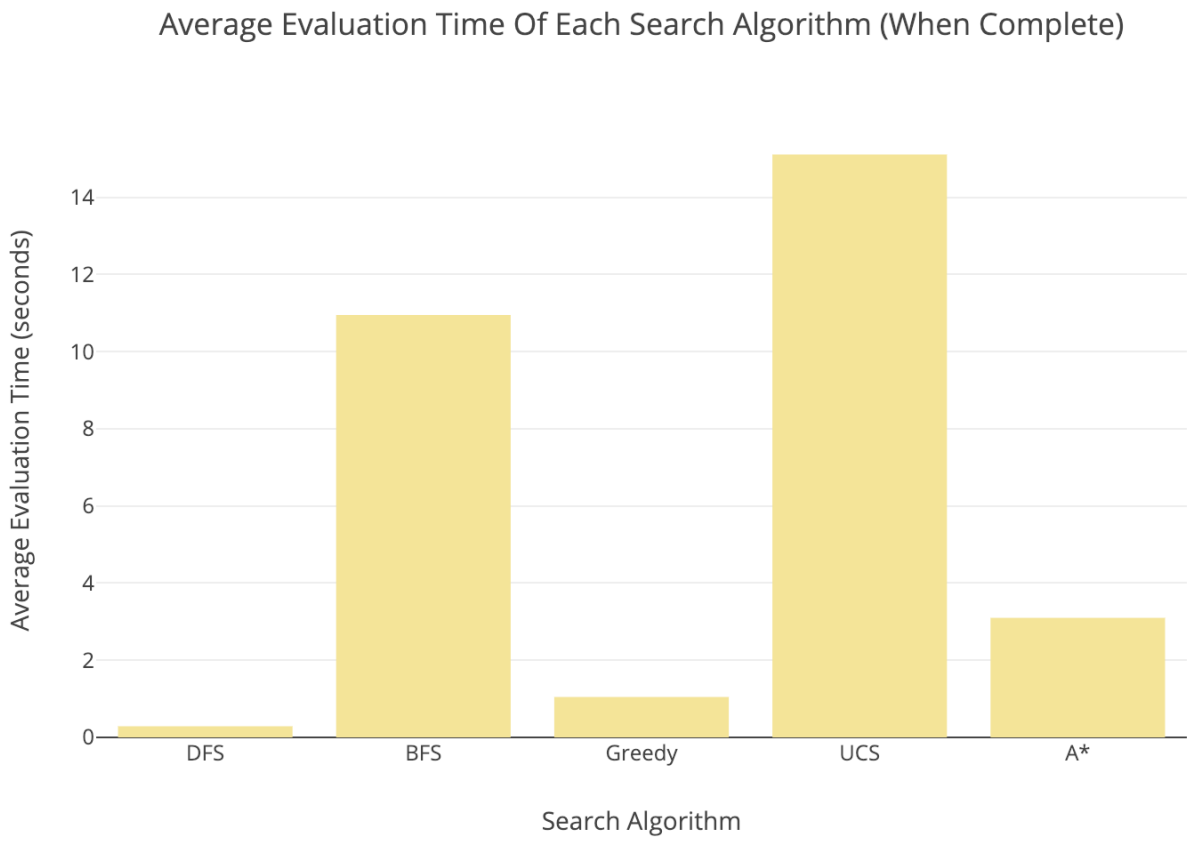

Figure 2: Average Evaluation Time of Each Search Algorithm

Sadness

Table 3 describes the average sadness that each algorithm incurred on its party when a solution was found. It is important to note that these sadness metrics do not reflect the failure rate described above, and thus, the two should be interpreted together. *Figure 4* shows the graphical representation of this data. To give further context to these point estimates, *Figure 5* shows the distribution of the evaluation time as histograms. Unlike previous tests, these graphs do not show much difference in sadness between algorithms. However, it is again remarkable that *Greedy Search* performed as well as it did: third best, after *UCS* and *A* Search*, both of which were deemed unusable by previous tests of failure rate and evaluation time. The distribution of all of the algorithms points to consistency in the quality of the sadness function.

All of the test point to the following conclusion: *Greedy Search* is the best algorithm for *OTT*. Not only is it reliable (proxied by a low failure rate), but it is also fast and effective.

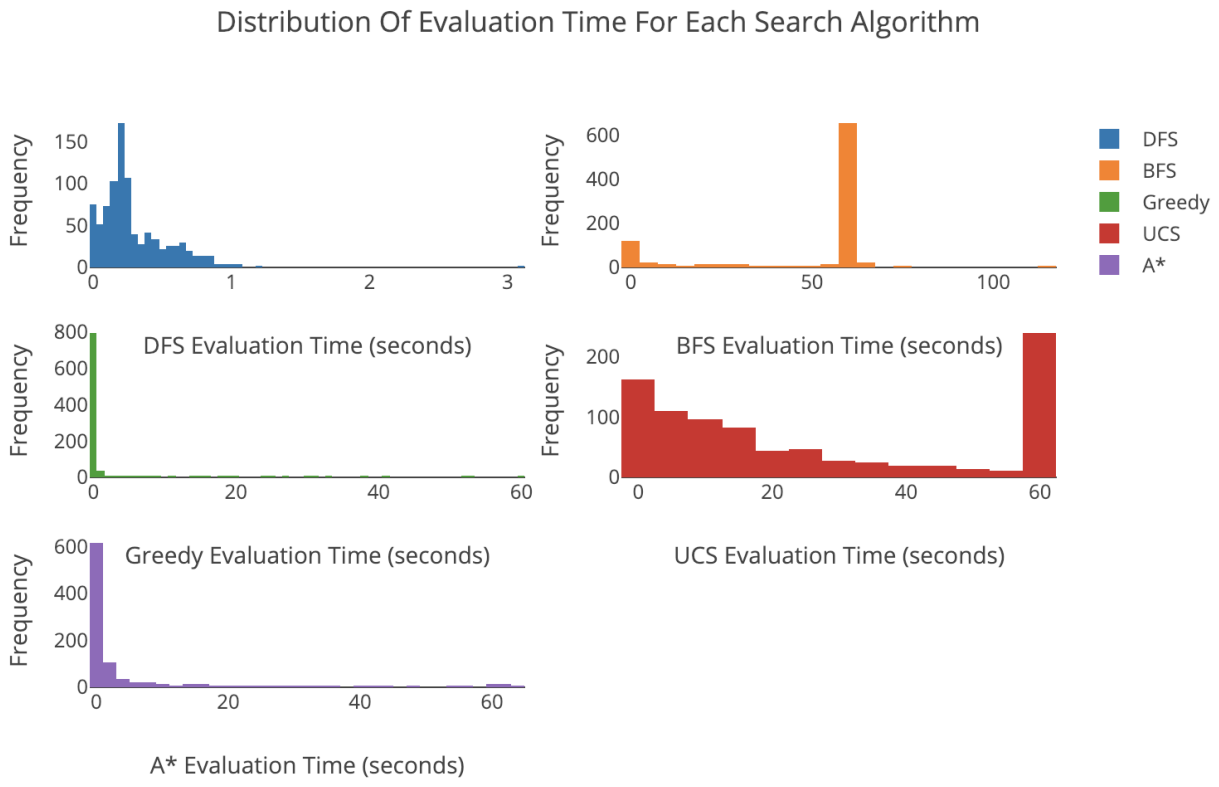

Figure 3: Distribution of Evaluation Time of Each Search Algorithm

Algorithm	Average Sadness)
DFS	28.189
BFS	26.998
Greedy Search	22.592
UCS	16.332
A* Search	19.047

Table 3: Average Sadness of Algorithms

6 Discussion

The original intention for this problem was encoding it as a Constraint Satisfaction Problem (CSP). After much investigation, testing, and discussion, the developers decided that the parameters of the problem fit better as a search rather than a constraint satisfaction problem. Instead of morphing constraints into the traditional structure of a CSP, one can use a much more intuitive approach to encode the problem as a search, described in detail in the **Problem Specification** and **Approach** sections. Specifically, the problem is structured as a tree search problem, with each node representing a particular venue being added to the list of solutions. Successors are created by the *similarity*

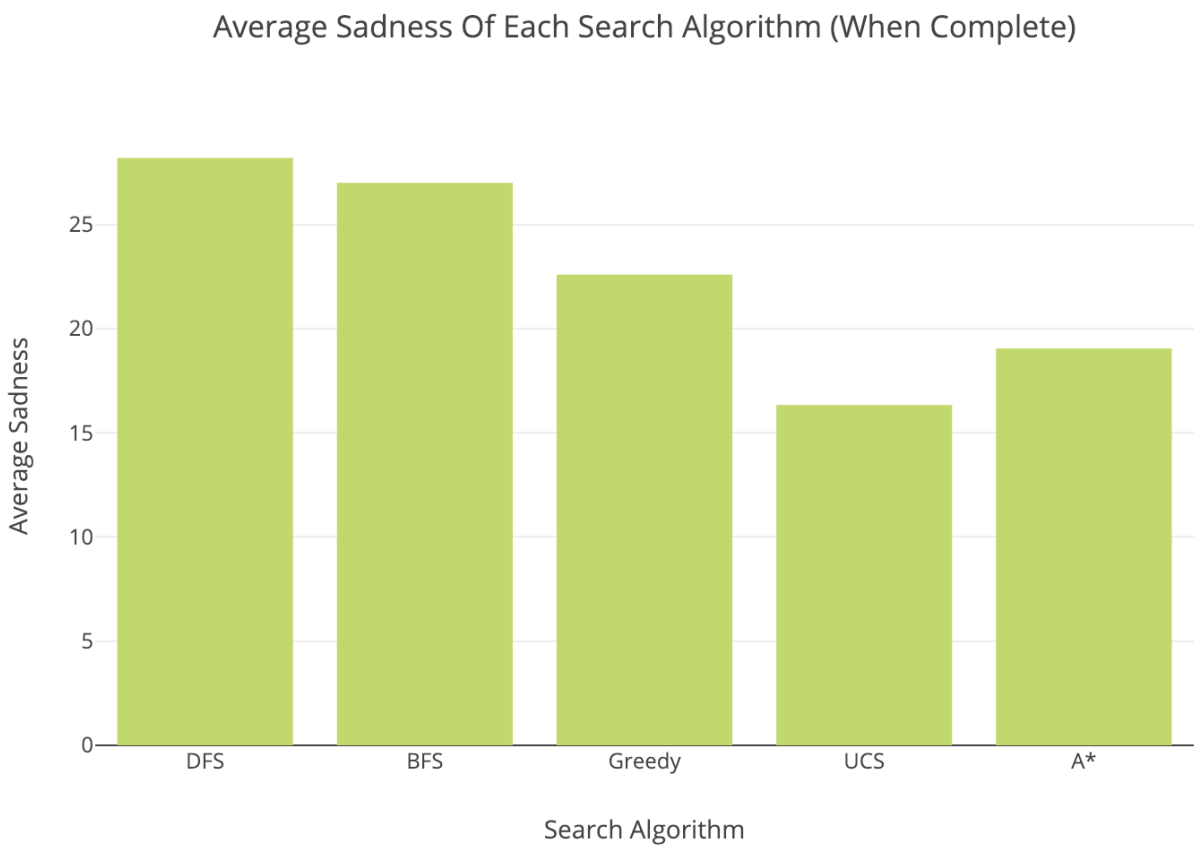

Figure 4: Average Sadness of Each Search Algorithm

function. To solve the newly specified search problem, the developers considered 5 algorithms to perform the search: *Depth-First Search (DFS), Breadth-First Search (BFS), Uniform Cost Search (UCS), Greedy Search, and A* Search*. Their performance on this problem is described below.

DFS runs quickly and fails infrequently, as it merely has to manoeuvre the tree to find a list of appropriate length, 7 (the goal test). However, the quality of the solution (proxied by the average *sadness*) is poor compared to the other algorithms because it does not take the cost of a path into account.

BFS takes a significant amount of time, and fails very often. Because it expands each level first and the branching factor (the average number of successors generated) is relatively large, *BFS* must traverse a very large number of nodes before reaching depth 7, the goal. For example, let the initial list of venues be 20, and let the *similarity* function filter out 2 elements from the remaining list at each level. The first level would add 20 elements to the frontier to be evaluated, each with 18 successors. Each of their successors would have 16 children, etc. Thus, there would be $(20) + (18 * 20) + (16 * 18 * 20) + (14 * 16 * 18 * 20) + (12 * 14 * 16 * 18 * 20) + (10 * 12 * 14 * 16 * 18 * 20) + 1 = 10,731,261$ nodes expanded before reaching the goal. The initial list is typically far more than 20 nodes, making this problem often unfeasible using *BFS*.

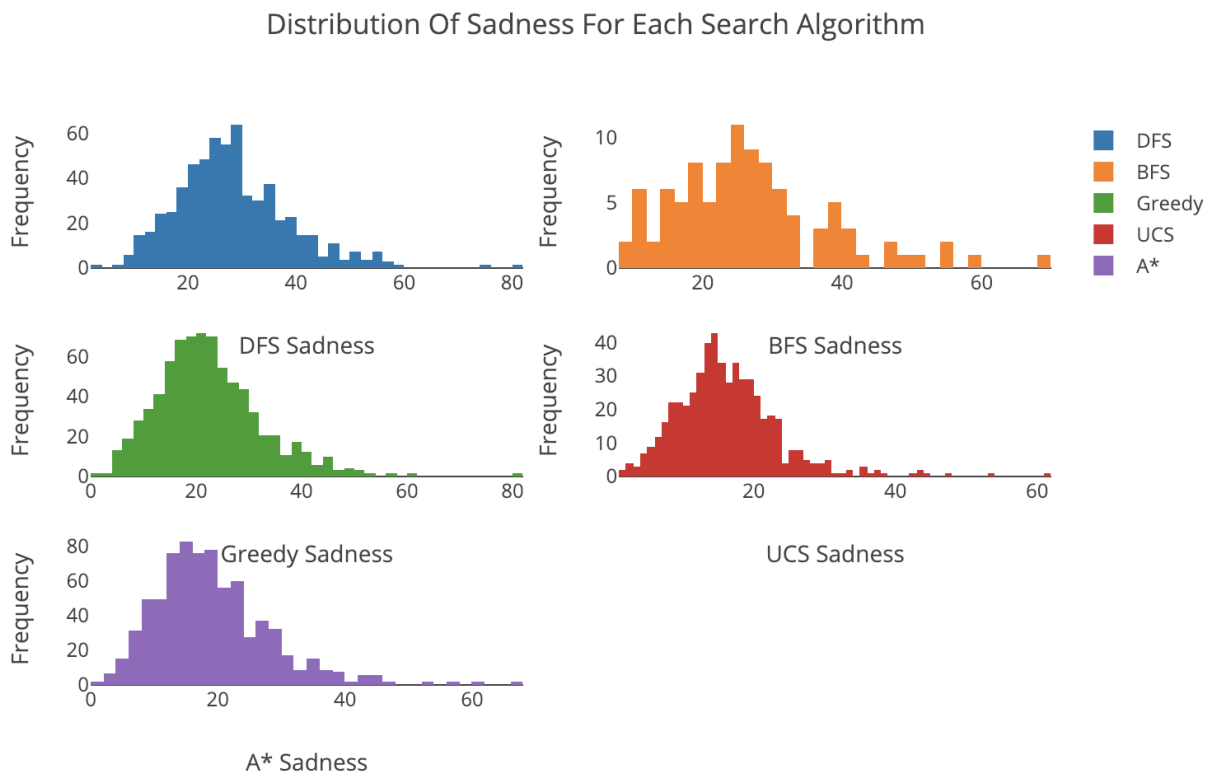

Figure 5: Distribution of Sadness of Each Search Algorithm

UCS runs quickly, but fails relatively frequently (with the second highest rate after *BFS*). It utilizes the *sadnessFunction* described in the **Approach** section above as a cost function to generate the priority for each element in its priority queue. Thus, it effectively produces diverse lists of 7 venues that are located around the users' center point and balance the users' preferences. The weights that were assigned by the developers in the *sadnessFunction* are fundamental to the algorithm's efficiency and effectiveness.

A Search* and *Greedy Search* both incorporate heuristics to order their frontiers (priority queues). As discussed above in the **Approach** section, A* Search uses the following linear combination of the heuristic, $h(x)$, and the cost, $c(x)$, to determine the priority of a successor, x: $f(x) = h(x) + c(x)$. *Greedy Search* uses the function $c(x)$ to determine the priority of a successor, x.

Finding a consistent (monotonic) heuristic for search problems is a fundamentally difficult problem and a topic of ongoing research.[3]. For this specific problem, that difficulty is augmented by the fact that the goal test is determined by the length of the list instead of the properties of the final node in the path. With that said, a heuristic that performs relatively well was developed and is described in **Algorithm 2**.

If this heuristic were either consistent or admissible, one would see reliably better performance by

the *A* Search* and *Greedy Search* algorithms over *DFS*, *BFS*, and *UCS* algorithms. This seems to be the case, given the results described in the **Results** section above. Both *A* Search* and *Greedy Search* perform better (they are faster, fail less frequently, and have lower sadness metrics) than the other three algorithms.

Greedy Search consistently out-performed all other algorithms in many metrics, pointing to the conclusion that the heuristic is good, and with *A* Search* near it in performance, the *sadnessFunction* is a reliable cost function.

Based on the evaluation of the 5 algorithms on test data relevant to this problem, the developers chose *Greedy Search* as the algorithm for the production version of OTT.

Development of this application will continue, and the following are future development goals:

- develop a consistent heuristic for *A* Search* and *Greedy Search*
- allow users to specify their own weights for their preferences
- tweak the *sadnessFunction* to include user-defined weights for preferences
- allow users to vote on outputted solutions, and incorporate those votes into a long-term model that tweaks *sadnessFunction* and the *similarity* function over time
- create a mobile application, and allow users to invite each other ahead of time via email and text message
- find a more sustainable way of making API calls to the Google Places API. The current implementation (under the free tier of Google Cloud Platform) limits these calls
- sell this product to Google

A System Description

On The Town (OTT) exists at a public repository on GitHub, here. The `README.md` file includes all the instructions necessary to run the application and perform the tests that generated the output. Those instructions are replicated below.

The core features of this application rely on the Google Places APIs, and search algorithms. The stack is as follows:

- Backend: Python, Flask, SQL
- Frontend: HTML, JavaScript, CSS, Jinja2

Installation

OTT runs on Python 2.7, and requires the following packages, included in the `requirements.txt` and `Pipfile` files:

- flask
- flask-session
- requests
- werkzeug
- requests-cache
- redis
- jupyter

After cloning this repository onto your computer, there are two ways to install these dependencies:

1. Pipenv [recommended]. Install `Pipenv` with `brew install Pipenv` if you're using Homebrew on MacOS or Linuxbrew on Linux, or `sudo dnf install pipenv` if you're using Fedora28. Otherwise, navigate here for more instructions. Next, navigate to the root directory of this repository and run `pipenv install` to install all dependencies at once and create a virtual environment for the project.
2. pip. Download `pip` from their website with the following command in a Terminal window: `curl https://bootstrap.pypa.io/get-pip.py -o get-pip.py`. Next, run `python get-pip.py`. Finally, navigate to the root directory of this repository and run `pip install -r requirements.txt` to install all dependencies at once.

If you plan to browse through the frontend, you must initialize the database and server as described below:

- Ensure that you have deleted `users.db` in the `server/` directory.
- Run `db_init.py`
- Run `redis-server`

To run the web application, run `python app.py` in a new Terminal window. The implementation of the frontend is currently limited by the fact that running it locally does not allow for the sharing of links, which is crucial to the party system. This is easily solved by migrating this product to its own domain, which is the next step of the development process.

Algorithm Testing

There is a file, `testing.py` that includes tests of all the *OTT* algorithms. This file allows one to check if the parties and users are being instantiated correctly, to investigate the process of that data structure creation, and test each of the algorithms individually. To run this file, type `python testing.py` into a Terminal window.

While `testing.py` tests the basic functionality of the application, a Jupyter notebook file was included to perform robust simulations. `evaluation.ipynb` performs many simulations of each algorithm on random parties. This file was used to generate the data used in the **Results** section. The data generated can also be found in the file `simulation_results.csv`. While there is an aspect of randomness to the data generation process (parties of random properties are instantiated), the data analysis process can be replicated by running the `analysis.ipynb` file, which utilizes the data the developers generated.

As mentioned in the **Results** and **Discussion** sections, *BFS* takes a very long time to run, so there were time limits imposed on its evaluation. These time limits are fungible, and one may also opt to not test *BFS* by removing or commenting out the lines of code that respond to its evaluation. For an idea of how long these may take, the developers performed overnight trials to gather the included data.

B Group Makeup

The Developers

The following are the three developers of **On The Town (OTT)** and their major contributions and responsibilities:

- Hakeem Angulu, Harvard College 2020, hangulu@college.harvard.edu
  - Documentation and commenting
  - UI and UX design and implementation
  - Algorithms design and implementation
  - Heuristic design and implementation
  - Database design and implementation
  - Backend (Flask and Jinja2) design and implementation
- Louie Ayre, Harvard College 2020, layre@college.harvard.edu
  - Algorithms design and implementation
  - Cost function design and implementation
  - Random party generation for testing and simulations
  - Diagnostics and empirical value determination
- Amadou Camara, Harvard College 2020, acamara@college.harvard.edu
  - Algorithms design and implementation
  - Testing of the Center.py and the Algorithms
  - Google Places API Implementation and Usage
  - Unifying the modules for seamless operation
  - Documentation and commenting

References

[1] Ngondi G. (Eds.) Butterfield, A. A dictionary of computer science: Breadth-first search. 2016.

[2] Ngondi G. (Eds.) Butterfield, A. A dictionary of computer science: Depth-first search. 2016.

[3] P. E. Hart, N. J. Nilsson, and B. Raphael. A formal basis for the heuristic determination of minimum cost paths. *IEEE Transactions on Systems Science and Cybernetics*, 4(2):100–107, July 1968.

[4] Peter Russell, Stuart; Norvig. *Artificial Intelligence: A Modern Approach (2nd ed.)*. Prentice Hall, 2003.