# Exploring Efficient Social Graph Search

Jason Stein and Maxwell Levenson

December 22, 2016

## 1 Introduction

For our final project, we decided to expand upon our lessons in informed and uninformed search in class and investigate efficient searching of social graphs. Finding links between users in large scale social graphs takes considerable computational time because of the high density of interconnections between nodes. In 2011, Facebook had tens of billions of friendship links (which can only be significantly larger now that their user base has tripled). These edges present a computational problem that makes using BFS to find the degrees of separation between two users inefficient because it considers every edge from every node it takes off its frontier–the branching factor can be in the hundreds or even thousands.

Social graphs are a particularly prevalent and interesting type of graph structure due to their usage in social media and more importantly internet advertisement targeting technology. They are characterized by specific structural properties like low average degrees of separation between nodes, and high density of edges. In class, we were introduced to several uninformed and informed search algorithms, however only some are useful for finding shortest paths between nodes. We decided to compare the efficiency of both classical algorithms defined in scientific literature like Breadth First Search and 2-sided BFS, with our own modified algorithms like a modified Depth-Limited Search and A* Search with a customized heuristic. Our goal was to investigate the efficiency of various search algorithms on these types of search problems as well as to possibly invent a conditional improvement upon basic BFS for some social graph search problems.

## 2 Background and Related Work

The immediate result when searching for content on this problem is [2]. We also considered such papers as [3] [4] [5].

# 3 Problem Specification

The problem being investigated is how to find the shortest path from one given node to another given node in a social graph as fast as possible. In general terms, this means to find the degrees of separation between two people using their social connections.

# 4 Approach

There were two significant pieces of original exploration done in this project, the generation of realistic social graphs (for testing purposes, and out of curiosity for how to accurately replicate human social structure), and our original heuristic aimed to make A* search perform significantly faster than BFS on social graphs. Besides this, we also used preexisting literature to recreate basic BFS, 2-sided BFS, and a modified Depth-Limited Search. We tested these four algorithms for run speeds over several iterations for predetermined pairs of nodes (start and end) for search parameters.

## 4.1 Social Graph Creation

One necessary consideration when starting our project was whether to use publicly available online social graph data-sets to test our algorithms, or create our own data-sets. We chose initially to pursue the creation of our own social graphs because we could be flexible with our branching factors and graph sizes. This proved to be a difficult endeavour and for purposes of complete consideration, we eventually downloaded a Facebook social graph data-set to test our implementations on a real world social graph [1].

Generation a realistic social graph is an interesting problem. How can we model the diversity of human interests and friendships that make some people have few friends and some have thousands? Initially, we thought to simplify the problem to the idea of the Harvard campus. In this world, each node would be a student, and each student would have several attributes. After generating all of the nodes in the graph, we would build our adjacency matrix of edges by comparing the attributes of two nodes and then giving the two an additive probability of being friends based on their shared attributes. We would then use a pseudo-random number generator to choose based on that probability if they would actually be friends and update the adjacency matrix. Our three initial attributes were ”House” with a domain of 12, ”Class year” with a domain of 4, and ”Interest group” (sporty, academics, arts) with a domain of 3.

This model however we realized was far too simplistic and fails to account for the vast differences in interests as well as for variability between peoples' extroversion within an interest group. We adjusted our model to instead assign people a randomized set of interests. Interests have varying sizes, and each shared interest between two people additively increments the probability by a value inversely proportional to the size of the group. This mimicked ”exclusivity,” essentially the assumption that two people that are members of a small group are relatively more likely to know each other than two people

that share membership in a large group. Individuals are also assigned one more overarching, unique characteristic (which we label as "major," which has a stronger effect on friendship probability. This setup was found to yield more clustered circles of friends, something we thought would give our A* search agent an advantage.

Another problem with our initial approach was that we estimated the probabilities of connection between people to be far too high. While social graphs have generally high branching factors, the probabilities were so high that using an algorithm like DFS would almost always find an optimal path immediately between nodes (even if the algorithm itself is not guaranteed to be optimal for our search problem). We adjusted and experimented with smaller probabilities accordingly.

## 4.2 Novel Heuristic for A* Search

Uninformed search seemed to be the standard for social graph search, but we wanted to see if we could create a heuristic that we could train on the graph to help an informed search algorithm, A* search, perform better than BFS on a social graph while retaining optimality (a condition we would struggle with). This would entail developing a consistent heuristic that could assist in determining how close two people would be.

We were inspired by the class lecture on clustering and thought of a clustering heuristic. People would be randomly assigned positions in a linear space (which for simplicity we chose to be two dimensional). This would be the initial configuration. Similarly to our data structure for storing edges, the adjacency list (which we chose because of the high density of edges in social graphs), we created a heuristic matrix that for every entry, had the Cartesian distance between two the two nodes used to index to that matrix position. This heuristic would then be used along with A* search to find paths.

We then trained the heuristic by iteratively selecting two random nodes, Node 1 and Node 2, and changing the position of Node 2 along the vector from Node 1 to Node 2. Node 2 would be "pushed away" if it was not connected to Node 1, and would be "pulled closer" if it was connected. We hoped that this iterative training would develop clusters where people with similar connections would be brought closer together and it would improve social graph queries for people in the same clusters.

## 4.3 2-Sided BFS

The next improvement we attempted was running a two-sided breadth first search. Since our search problem consists of one start node and exactly one end node (henceforth referred to as A and B), we could expand outward from each simultaneously, rather than just from the start node. We track two frontiers, one for each exploration, and expand one node from each on each iteration. The goal condition for each node expanded then becomes membership in the opposite frontier: from this point that we have reached from person A, have we already found a path to this node from person B? The reverse is also checked. The algorithm then returns the path from A to the joining point concatenated

with the reverse of the path from B to the joining point, yielding one path from A to B. This algorithm theoretically expands $O(b^{d/2})$ nodes, compared to standard BFS which expands $O(b^d)$. BFS2 runs significantly faster than both BFS and both versions of A* and uses much less memory.

### 4.4 Modification on Depth Limited Search

One other algorithm we wanted to try modifying for our purposes was iterative deepening. For social graphs, we were able to experimentally determine that there is an average path length of 3 to get from one person to another. Therefore we could start iterative deepening at depth 3, and then if we didn't find a result, continue for larger depths, but if we did find a result then try depth-limited search for depths 2 and 1 to see if there is a more optimal path. While this sounds promising in theory (depth first searching first and the most common depth), it was in fact quite slow compared to our other algorithms.

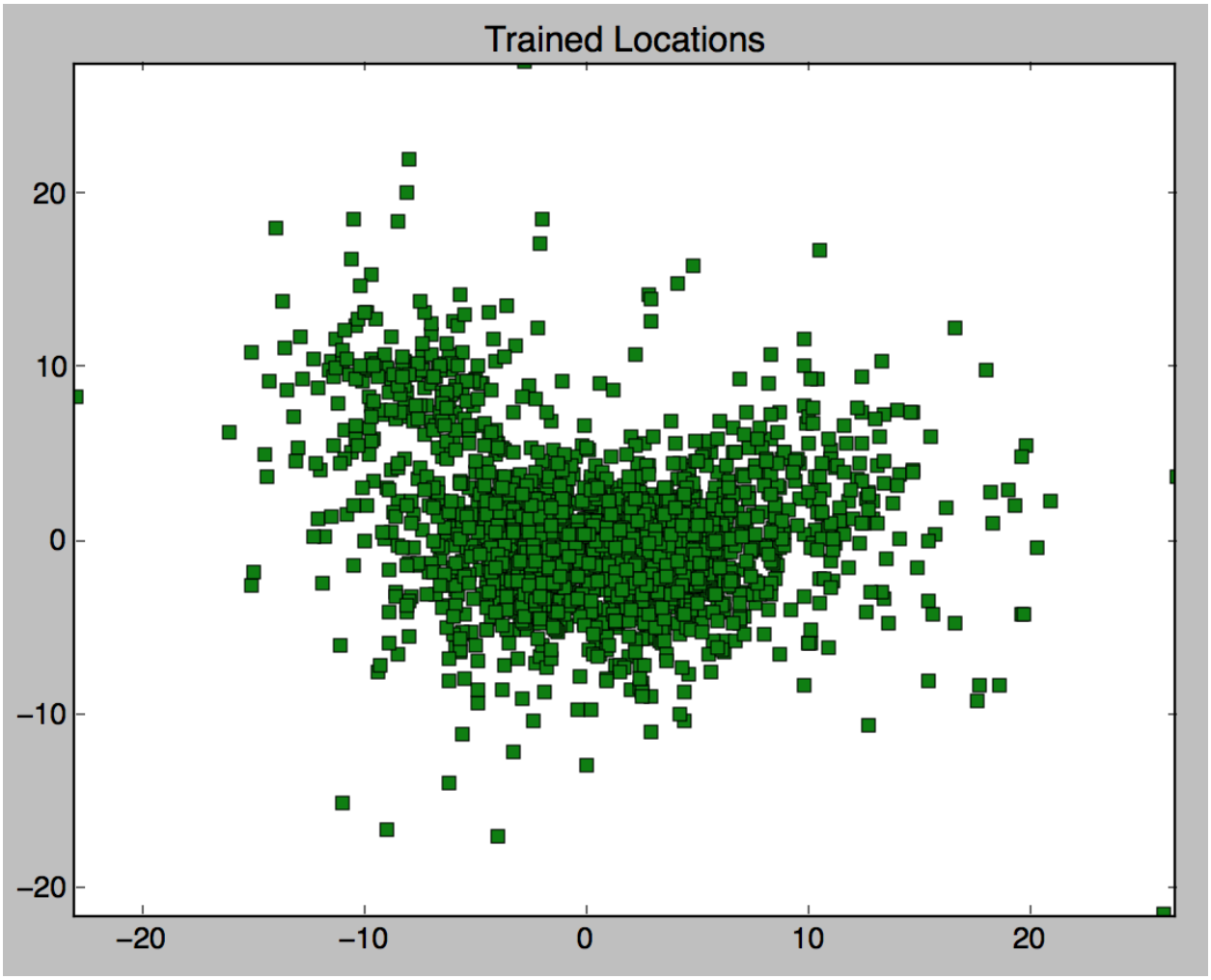
## 5 Experiments

The majority of our testing consisted of tweaking parameters in our graph generation, essentially altering the features of connectivity in our graph. We experimented with giving people more friends as a fraction of population by increasing the base probability of friendship, as well as clustering friend groups by increasing friendship probability over shared interests. We also manipulated the parameters of how training was carried out; different constants for pushing points closer / further based on friendship yielded different mappings. By altering these constants and monitoring how they affected features of the graph such as average degree of separation, average friend count per person, and visible clustering in our trained locations, and how those features affect the performance of each of our search algorithms.

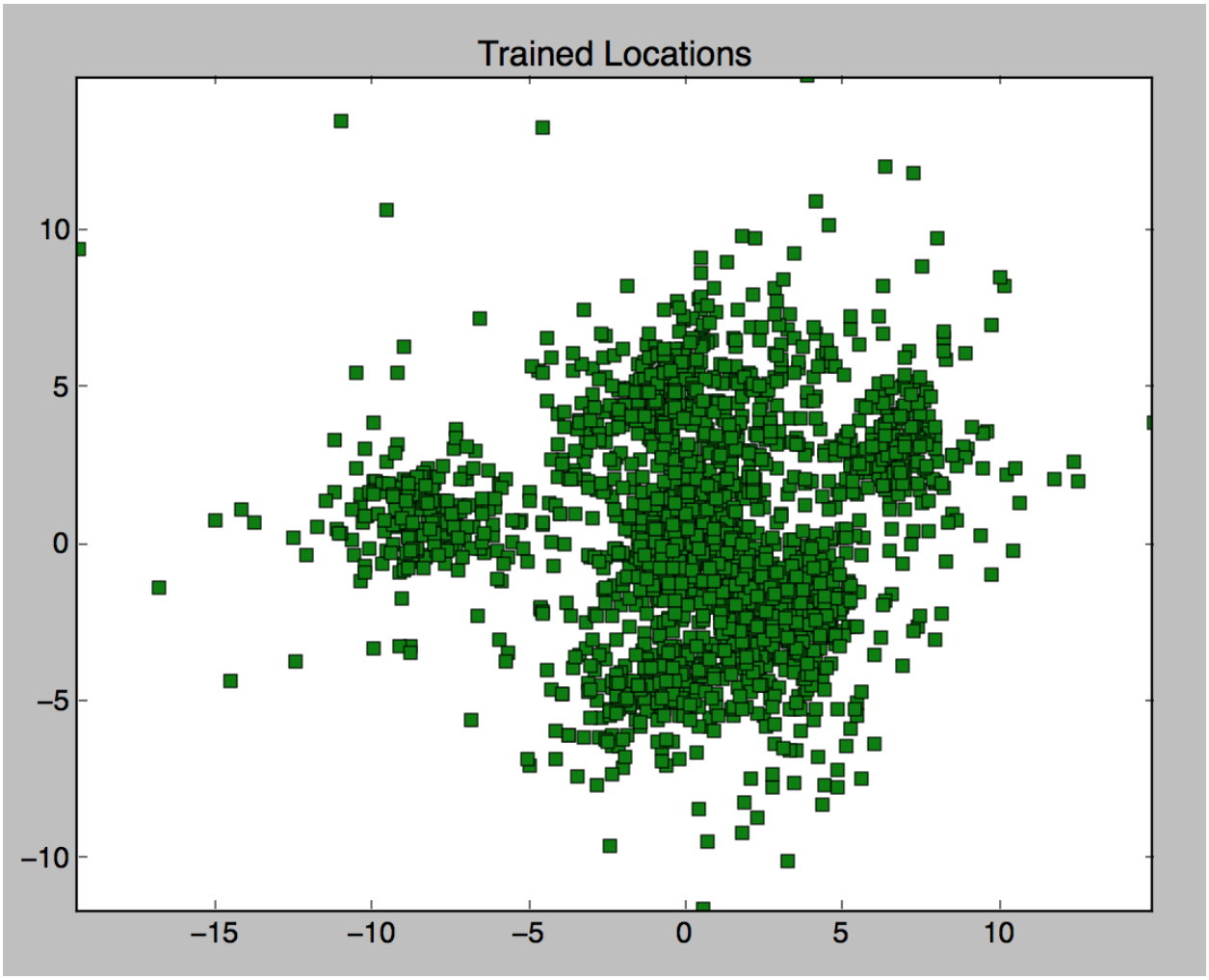
### 5.1 Results

Overall, A* search consistently outperformed BFS, especially in graphs that were more connected (individuals had more friends, so the branching factor was higher). However, this was true for A* with both the untrained as well as trained heuristics. Trained A* only consistently outperformed untrained A* with very specific graph parameters, and even then only by a very slim margin. The following are examples of output from various arrangements of constant parameters:

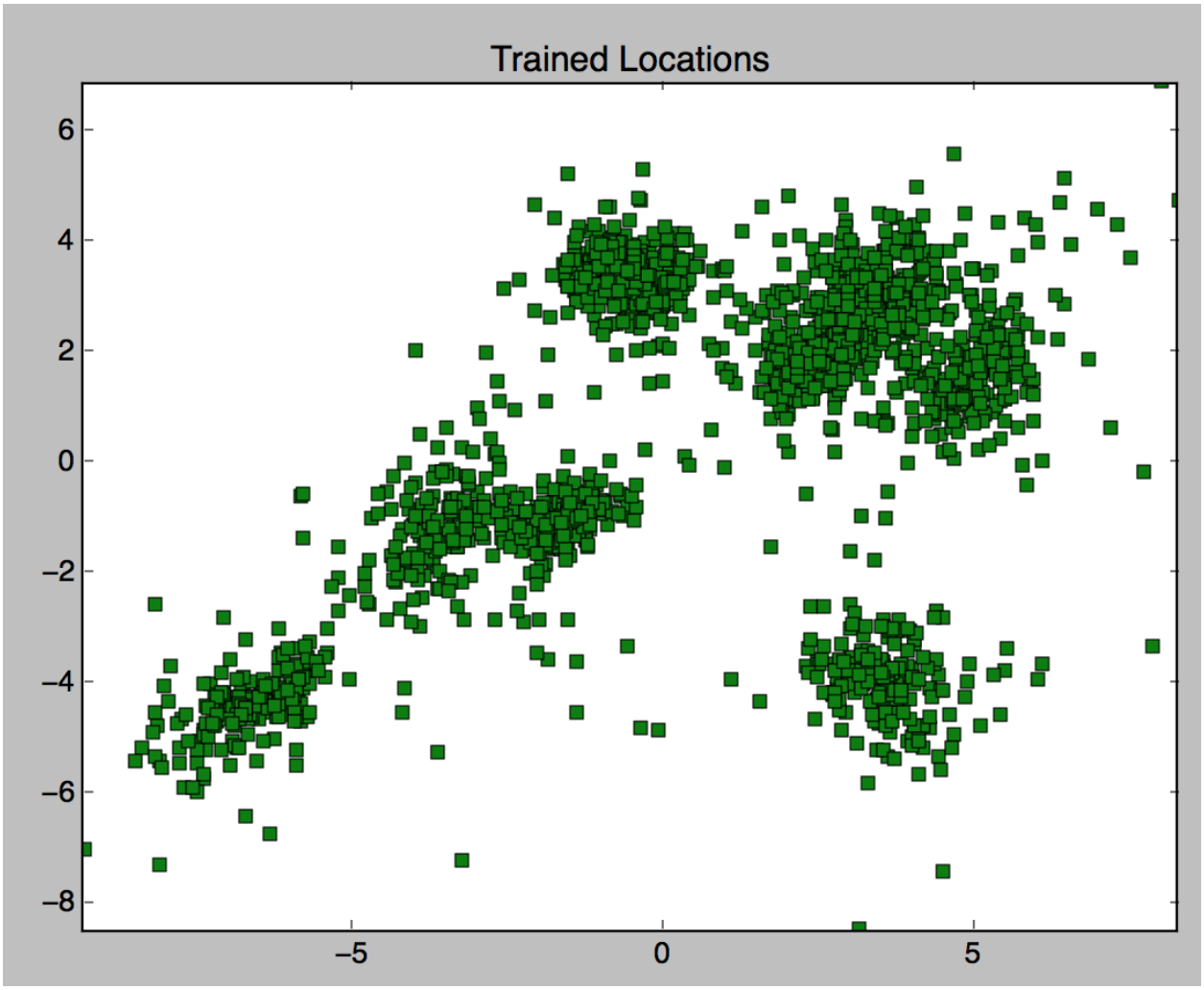
```
NSTUDENTS=2000 NINTERESTS=200 NMAJORS=10 BASE_PROB=0
INTEREST_PROB_INC=0.05 MAJOR_PROB_INC=.2

Generating data set ( n = 2000 )
The average person has 41.318 friends.
The average person has 3.0155 interests.

BFS -- Time: 20.637 Avg Nodes: 847
      Avg Separation: 2.86 Iterations: 50
A* untrained -- Time: 10.612 Avg Nodes: 395
      Avg Separation: 2.86 Iterations: 50
A* trained -- Time: 10.907 Avg Nodes: 395
      Avg Separation: 2.86 Iterations: 50
2-sided BFS -- Time: 1.178 Avg Nodes: 51
      Avg Separation: 2.86 Iterations: 50
```



```
NSTUDENTS=2000 NINTERESTS=200 NMAJORS=10 BASE_PROB=0
INTEREST_PROB_INC=0.01 MAJOR_PROB_INC=.25

Generating data set ( n = 2000 )
The average person has 50.203 friends.
The average person has 3.036 interests.

BFS -- Time: 27.281 Avg Nodes: 984
      Avg Separation: 3.56 Iterations: 50
A* untrained -- Time: 17.771  e Avg Nodes: 592
      Avg Separation: 3.56 Iterations: 50
A* trained -- Time: 18.057 Avg Nodes: 592
      Avg Separation: 3.56 Iterations: 50
2-sided BFS -- Time: 11.227 Avg Nodes: 272
      Avg Separation: 3.56 Iterations: 50
```



```
NSTUDENTS=2000 NINTERESTS=200 NMAJORS=10 BASE_PROB=0
INTEREST_PROB_INC=0.02 MAJOR_PROB_INC=.35

Generating data set ( n = 2000 )
The average person has 71.126 friends.
The average person has 3.1615 interests.

BFS -- Time: 37.985 Avg Nodes: 1046
      Avg Separation: 3.02 Iterations: 50
A* untrained -- Time: 19.178 Avg Nodes: 489
      Avg Separation: 3.02 Iterations: 50
A* trained -- Time: 19.618 Avg Nodes: 489
      Avg Separation: 3.02 Iterations: 50
2-sided BFS -- Time: 5.401 Avg Nodes: 111
      Avg Separation: 3.02 Iterations: 50
```



# 6 Discussion

We attempted three different approaches to improve on naive breadth-first search to find degrees of separation between two nodes in a social graph. A modified version of iterative deepening, 2-side breadth-first search, and a trained heuristic with A* search.

The modified iterative deepening depth-first search while seemingly useful, turned out to be atrociously slow. This makes sense because it would only get faster queries than BFS occasionally for paths of depth 3. If the optimal path was greater than 3, it would have tried every path of 3 first before trying greater length paths creating redundancy. There's also a bit of redundancy when going backwards and checking depth 1 and depth 2 paths after finding a depth 3 path.

The most interesting experimentation was with our customized heuristic and A* search. Initially, our results were very promising. It looked like our heuristic was speeding up the search queries by a factor of 4! However we quickly realized that not only did the trained heuristic speed it up significantly, but so did the random heuristic that had no clusters. This meant that there was a fundamental problem with our heuristic implementation. We explored the problem by modifying the parameters defining our social graph generation. We discovered that making the graph sparser (lowering the branching factor) made the random heuristic closer in speed to naive BFS, however when they became the same, our trained heuristic had gotten worse than naive BFS. Eventually we concluded that the issue was the consistency of our heuristic

While we were able to make the heuristic admissible by scaling the distance values to be from 0 to 1, we were unable to reach consistency. In order to enforce consistency, we would have to know what at least one of the search parameters would be while training the heuristic. That way we could make sure the agent only explored paths that had a decreasing heuristic value for each step. However because we want our heuristic to be applicable to any search query and not just very specific ones, there is a conflict of constraints.

Our most effective method, however, was 2-sided BFS. The algorithm expands $O(b^{d/2})$ nodes, as compared to BFS's $O(b^d)$ (where $b$ is the branching factor and $d$ is the solution depth. Since we are working with very large values of $b$, this leads to huge savings even though $d$ is relatively shallow. Even though the goal test is more complicated (linear rather than constant, because we need to test for membership in the frontier and not simply equality with the goal node, the algorithm runs, in some cases, several time faster.

Our major takeaway from this investigation is that 2-sided BFS is by far the fastest algorithm we explored and in real-world situations, this is most likely the algorithm to use for social graph search. While A* search may be a useful tool for searching AI state spaces, it has heuristic issues when it comes to searching social graphs. In future work, it would be interesting to see if we could switch to using A* search for nodes in the same cluster once the heuristic has been trained vs 2-sided BFS for the general case and see if this per problem switching added improvement over a large amount of queries. On a general science note, this project taught us about the very real existence of negative findings. While in theory our clustering technique had promise for searching social graphs,

in practice, the experimental data showed that there was underlying scientific flaws. The scientific process is a wonderful thing.

# A System Description

To run all search agents and compare, as well as view the trained locations of the individuals and some summary statistics, run

```
python app.py
```

You may also want to change constants as we did, namely `NSTUDENTS`, `BASE_PROB`, `INTEREST_PROB_INC`, `MAJOR_PROB_INC`, `NINTERESTS`, and `NMAJORS`, all of which are in `generate.py`. Altering these will change how the graph is generated, and yield different features within the graph as well as different algorithm performance.

To see that the algorithms do indeed all find paths of equal length, you can run `python testscript.py` to see the paths returned by various searches on the same source and target people.

## A.1 Included Files

All files for this project can be found at `https://github.com/MaxwellLevenson/cs182-final-project`

`facebook/` - Real Facebook social graph data-set
`app.py` - Algorithm comparison apparatus
`generate.py` - Graph generation
`load.py` - Loads real data-sets
`searchAgents.py` - Graph search algorithms
`snap.py` - Downloaded library for data-set importing
`testscript.py` - Simple script for testing accuracy of algorithms
`util.py` - Assorted utility classes and functions

# B Group Makeup

Jason Stein: Search agents, data generation, testing application.

Files / functions worked on: `BFS`, `BFS2`, `aStar` search agents; `generate.py` except for `train`, `changeDistance`, `prettyPrintHeuristic`, and `app.py`

Maxwell Levenson: Developed the customized heuristic, adapted real data-sets for our use, wrote the iterative depth-limited search algorithm, and collaborated on the testing apparatus, the social graph generation design, and the graph structure code.

Files/functions worked on: `load.py`; `app.py`; `generate.py`: `train`, `changeDistance`, `prettyPrintHeuristic`, `SocialGraph.init`; search agents: `DLS`, `IDDLS`;

`util.py` was adapted from the UC Berkeley AI assignment framework's Pacman application.

# C Special Thanks

Special thanks to our teaching fellow Shai Szulanski who provided invaluable assistance in helping us frame and design our project.

# References

[1] Stanford large network dataset collection: Facebook circles. `https://snap.stanford.edu/data/egonets-Facebook.html`. Accessed: 2016-11-30.

[2] Optimizing breadth-first search for social networks. `https://www.sumologic.com/author/karthik/`, 2014. Accessed: 2016-10-28.

[3] Paul T. Stanton and Randal Burns. I/o efficient search of large social networks. *5th Annual Network Science Workshop*, 2010.

[4] Bin Yu and Munindar P. Singh. Searching social networks. *ACM Press*, 2003.

[5] Xiaohan Zhao, Alessandra Sala, Haitao Zheng, and Ben Y. Zhao. Fast and scalable analysis of massive social graphs, Jun 2011.