Final Project Report
Group Logins : cs170-asi, abo, aou, aru

Our solution to the Horse problem was to use 3 different randomized and optimized algorithms, and continuously run the algorithms on several different machines, and pick up the best results as time progressed. We ran final scripts that compared the outputs generated by all 3 algorithms, and took the best 600 values to be our final solution.

The first algorithm (horse.py), started by finding a starting vertex, source nodes (0 incoming edges) were chosen first, and then after we used all source nodes, we randomly chose starting vertices. After finding a starting vertex, we decided to pick the “best” neighbor by using a calculation that took the value (neighbor i has a value of adj[i, i]) of the neighbor, the number of neighbors that neighbor had (choosing this neighbor will give us a better chance of finding the best path because it has more options), and the number of incoming edges that the neighbor had. It was a weighted calculation that also included some randomness, but mostly was based on these values. From there, we kept traversing, until we reached a source, or the algorithm decided to stop the path (randomly picking a neighbor, or not picking a neighbor). Once a path was found, it was deleted from the graph, and the process continued until there were 0 vertices left.

The second algorithm (bogo.py) was a completely random algorithm that was similar to the first, except that it chose neighbors completely at random. The rest of the procedure was the same. We included this because we thought it would help by finding smaller edge cases that the other algorithms didn’t find. Bogo also had the advantage of being reliant purely on time rather than any other metric, and given enough time, would eventually find the best path. This algorithm was mostly used by running in parallel with the other two and constantly updating and recording any new or better paths it would discover. The sheer amount of runs that this found would guarantee that we found some paths that were better from the ones generated by the other two algorithms and could be used as a supplement as well as a way to check if a path was truly “optimal” by comparing if the result from bogo was the same as the other algorithms.

The last algorithm (horse2.py) used a different metric to choose the next vertex to traverse. The approach was to find a starting vertex like above, find the “best” neighbor of that starting vertex, and then traverse the neighbor. Now, the “best” neighbor was chosen based on the score of the Strongly Connected Component that it was in. So if the neighbor, w, was part of a Strongly Connected Component that had a larger score (sum of the elements in the SCC times the number of elements), it had a higher probability of being selected. This approach was taken because we needed a way to determine how good a certain neighbor could potentially be, and we decided that if a neighbor was part of a strongly connected component that yielded the highest score out of the neighbors, there will be a greater chance of finding the optimal solution. The algorithm follows the path through this scc, and if along the way, a better one was found, the score could be improved even more. At the very beginning of the algorithm, after constructing the graph, we find all the strongly connected components and calculate the score for each scc, and assign this value to each of the member of each scc (i.e., if node 5 was in an scc with score 20, node 5 would contain this value and it would be used to calculate the best neighbor). Then as we traverse the neighbors, we check the scc scores of the neighbors and choose with a higher probability if the scc score is higher than the others. Now, if a node is in its own scc, alone, then we use the heuristic mentioned in algorithm one.