UC Berkeley - CS170 – Spring 2020 – Project Reflection – Philipp Kurz - 3035440074

My progress in the project can mostly be split into two different aspects: Coming up with good heuristics and approximations, and setting up a framework that allows for optimizing outputs using randomization

Approaches for solver algorithms

1. "Destructive" algorithms
   - Create a tree T on the input graph G first
   - Then take away edges/vertices so that minimum average pairwise distance gets reduced without conflicting with the requirement that T is a dominating set

   To find a starting tree T, different algorithms were employed:

   - Naive DFS with random tiebreaking and random starting vertex
   - Naïve BFS with random tiebreaking and random starting vertex
   - Uniform Cost Search (Dijkstra) using a min-heap
   - A minimum spanning tree (MST) from NetworkX's library (Kruskal's algorithm)
   - BFS with random tiebreaking with vertex with highest degree as starting vertex
   - DFS where tiebreaking is according to inverted weights (DFS with weight heuristic)

   To prune edges/vertices from the original tree, mostly two approaches were employed:

   - List all leaves in T; shuffle list; pop from list until empty and for every leaf, remove leaf if it results in a better score (randomized greedy approach with one iteration)
   - Like first approach, but keep looking for new leaves resulting from previous iteration

2. "Constructive" algorithms
   Instead of creating a tree and then pruning, create a small dominating set first and then keep adding edges if they improve score until convergence

   For this, mostly one algorithm was employed:
   - Non-deterministically pick starting vertex (the higher the degree, the higher the probability)
   - Keep picking non-deterministically with same heuristic until dominating set is achieved
   - Keep looking at edges that could potentially add new edges to T and check if adding those edges and vertices would result in better score (picking edges at random)
   - Repeat until convergence

Approaches for automizing solver

1. Storing past results

Randomization only pays off for optimization for our project, if we only store new outputs if they are better than old ones. To do so, I had a Python dictionary that would hold the best score for a certain input so far (mapping best score so far to each input) which was serialized (scores.obj). This allowed, that a lot of different solvers could be tested on both my local machine and all old outputs were retained without ever overwriting them with worse solutions. Doing so, randomized solvers can be run until convergence before employing a new solver algorithm.

2. Utilizing instructional machines for better hardware

Because my laptop does not have the fastest processor, I started using the Derby instructional machines. Since I always had my code in a GitHub repo, I simply had to clone my repository and pull changes whenever I edited my code locally. Then, I could run my code and push the improved updates back on my repo. This enabled me to run my code basically 24/7 which brought randomized solvers to convergence much more effectively.

3. Skipping already ideal outputs

I wrote a script that could extract my position for every input from the online leaderboard. It then stored all inputs where I already had the first position in a Python dictionary, which was serialized (updates.obj). This allowed me to skip already solved inputs and thus spend more time on the harder ones.

4. Visualizing inputs

On some inputs, I had very bad performance. To get a better understanding of the input graph and to see if my algorithm had any shortcomings, I wrote a script that visualized the input graph using Python's MatPlotLib, which was very straightforward using NetworkX.

Overall, the project was a lot of fun. I enjoyed implementing many theoretical topics from class (Graph traversal, randomization and approximation) and see how small changes could greatly improve the quality of my solver.