Harnessing the Extraordinary Power of Randomness, Parallel Processing, and Local Search to Perform "Smart" Brute Force

Rohan Chilukuri, Jordan Chernof, and Margaret Misyutina

(CS170 Spring 2020 Final Project)

Problem Statement:

The goal of the project is to minimize the average path length of a tree that is a dominating set in a graph with n vertices.

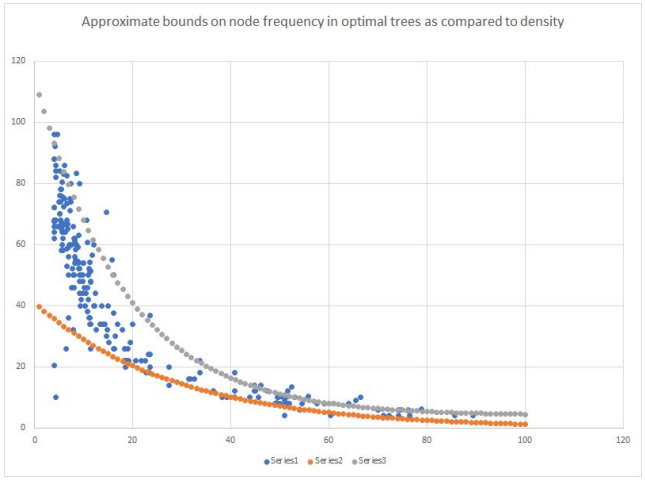
Considerations:

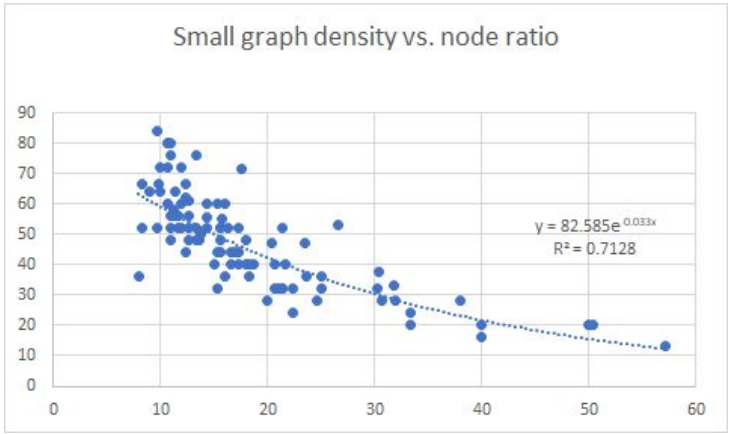
- There exist $2^{\binom{n}{2}}$ possible graphs in a graph with n vertices. It is clearly not feasible to consider every graph, decide if it is a tree connected dominating set, and choose the one of minimum average path length. We can get a rough idea of a viable tree connected dominating set through random permutations of the graph, but to find the optimal solution efficiently we turn to more intelligent ways to pick graphs, and polynomial time algorithms.

- Networkx uses many randomized and approximation algorithms to calculate NP-Hard problems in polynomial time, thus running these multiple times will yield a wider breadth of results and possible solutions to use in our algorithms.

- Every maximal independent set is a dominating set. We will use this fact extensively in limiting the vertex set for our potential tree connected dominating sets.

- A plot of "graph density vs. number of vertices in the optimal tree" shows an inverse relationship. Linear regression with error bounds gives an idea of which potential solutions to consider first.





- Upon finding a potential solution, we feed it back into the program to improve what we already have. However, using local search algorithms, we risk getting stuck on local minima, so we always keep a broad approach in exploring new solutions.

- Running each input file separately in parallel will not lead to overwriting.

Approaches/Strategies/Algorithms:

We used the following strategies (named after the boolean flags in our solver.py file):

Brute Force - Generate a random maximal independent set, or a random permutation of vertices to be considered as a tree connected dominating set

Brute Edges - Generate a random permutation of edges (vertex set are the endpoints of the edges) to be considered as a tree connected dominating set

Maximum Spanning Tree - Find the maximum spanning tree of the graph.

Dominating Set - Find the minimum dominating set of the graph and connect the nodes with a Steiner Tree approximation algorithm.

Edge Tinkering - Add and remove a random subset of edges connected to degree one vertices from the solution tree.

Add Small Edges - Try adding the smallest edge that keeps the solution a tree. If improvement, try again.

Add Nodes - Try adding all nodes and incident edges one at a time to the solution tree. If adding the node and edges creates a cycle, remove a random edge from a cycle in the graph until it results in a tree.

Remove Degree 1/2/3/4 Nodes - Remove all nodes of degree one from the solution tree as long as the result is still a dominating set. Remove degree two nodes, and connect the resulting two connected components. Remove degree three nodes, and connect the resulting three connected components. Same with four.

Tree Cut - Create a cut removing the largest edges of the graph, and find better edges to connect the two connected components across the cut.

Replace Large Edges - Remove an edge of heavy weight, and try to find a shorter path between the two endpoints attached to the edge using Dijkstra's Shortest Path algorithm.

Performance/Results:

First, the methods that failed to produce effective results. Maximum Spanning Tree, as expected, proved ineffectual and generally did not yield solutions close to optimal. Dominating Set, while returning decent results for some of the graphs, failed to continue to improve due to it being deterministic.

Now for the methods that worked. Brute Force and Brute Edges were very effective, yet inefficient, for finding solutions at the start of the project. Since these methods essentially try every possible solution (up to some iteration), they will eventually find the optimal solutions, however on even medium sized graphs, these strategies proved infeasible. However, they did give a baseline to improve on.

The rest of the strategies seek to improve upon a solution by modifying the tree in some way. Edge Tinkering, Add Small Edges, Add Nodes, Remove Degree 1/2/3 Nodes, Tree Cut, and Replace Large Edges can be seen as local search algorithms. As such, they improved the previous solutions from Brute Force and Brute Edges significantly when being run on the outputs initially, but over time the solutions converged to some local minima.

Overall, we balanced the usage of the completely random Brute Force and Brute Edges strategies with the local search strategies to attempt to find other local minima, and possibly a global minima.

Note: parallel processing decreased the computation time by a factor of about 10 depending on the machine used.

Services:

We used cloud services freely available for anyone to use in order to perform our computations. We used the Google Cloud Platform (GCP) free trial credit of $300 to spin up three virtual machines, which ran most of the computations. We ran some of the small graph computations locally on our laptops. We then would download the outputs from the VMs, combine them locally, and push to github/upload to gradescope.