1 Introduction

This article will be looking at different methods at approximating graph metrics, and analysing the results the approximations give. There are two approximation methods that this article will go through. The first is using approximation algorithms; these are algorithms that do not calculate the exact value of a metric, but a similar value that can be used to estimate the exact value. The second method is to compute the graph metric on a *reduced* graph. In this article, a reduced graph of a graph G is a graph with the same number of vertices as G, but with less edges. The goal of these approximation methods is to decrease the run time of graph algorithms, while still getting good results in the graph metrics.

2 Approximation Algorithms

This section will cover the approximation algorithms used in the experiments to compute graph metrics.

2.1 Approximate Diameter

The diameter of a graph is the maximal distance between a pair of vertices. That is, the shortest path between any two vertices is at least as small as the graph's diameter. The algorithm that will be analysed in this article for approximating graph diameter will be based on iterations of "2-BFS".

Each iteration performs breadth-first search twice. The first BFS is performed starting at a random vertex v, to find a vertex u that is of maximal distance from v. The distance from u to v, d_v is recorded. Then, the second BFS is performed starting at u. The maximal distance from u to another vertex, d_u is recorded. (this is not necessarily the distance from u to v). Then an iteration of "2-BFS" will return $max(d_u, d_v)$. The approximate diameter algorithm runs "2-BFS" 1000 times, and returns the maximum value of the 1000 "2-BFS" iterations.

2.2 Graph Reductions

There are three main graph reductions used in the experiments below. All the reductions keep the same number of vertices, but try to lower the number of edges in the graph. Some of the graph metrics we want to measure characterize vertices in the graph, so there should be no loss of vertices.

The first reduction algorithm iterates through each vertex, and keeps the edges to the highest k degree neighbours.

The second reduction iterates through each vertex v, and prioritizes its high degree neighbours, but also attempts to avoid any triangles formed by the neighbours. On each vertex iteration, each neighbour is initially given a priority 1. The algorithm will choose to keep the highest degree neighbour with priority 1. Whenever an edge vu is kept, the algorithm goes through each pair of remaining neighbours and checks if the pair forms a triangle with u. If so, then the priority of the pair of neighbours increase by 1. The algorithm continues by adding the next highest degree neighbour that still has priority 1. If there are no more neighbours with priority 1, then neighbours of priority 2 will be added, and so on until k edges are kept. This reduction is detailed in Algorithm 1.

The third reduction combines the edges of several spanning trees to form a reduced graph. In the experiments below, the spanning trees are rooted at the highest k degree vertices of G.

3 Analysis

Algorithm 1 Graph Reduction Avoiding Triangles

Input: Original Graph G, Empty Graph H, integer k

for all vertices v of G **do**
  Add v to H
  for all neighbours u of v **do**
    Priority[u] $\leftarrow 1$
  end for
  EdgesAdded $\leftarrow 0$
  Priority $\leftarrow 1$
  while EdgesAdded != k **do**
    Find highest degree neighbour u where Priority[u] = Priority
    Add edge uv to H
    EdgesAdded = EdgesAdded +1
    for all neighbour pairs (w, x) of v **do**
      if (u, w, x) is a triangle in G **then**
        Priority[u] = Priority[u] +1
        Priority[w] = Priority[w] +1
        Priority[x] = Priority[x] +1
      end if
    end for
    Priority $\leftarrow$ Priority +1
  end while
end for

Output: Reduced Graph H