COMPUTING CARVING AND BRANCH DECOMPOSITIONS FOR APPROXIMATION ALGORITHMS

AN INDEPENDENT STUDY WITH PROFESSOR KLEIN.

Ken Noh
Brown CS '20
khnoh@brown.edu

David Oyeka
Brown CS '20
david_oyeka@brown.edu

1 Introduction

The `Planarity` codebase [1] implements an approximation algorithm for the Steiner Traveling Salesman problem. Given some input graph, a branch decomposition is constructed - this part of the program is crucial as it ensures the polynomial running time of the approximation algorithm. Previously, this branch decomposition was computed in a dynamic fashion. First, all the trivial boundaries are formed, usually consisting of two original vertices. To merge clusters, we first check vertices found in both boundaries to determine if they are corners, using degree maps, which store the degree of a vertex within a specific cluster. In this independent study, we explored some of the problems that arise with this implementation and the changes we made to address them.

2 Key Issues

We were initially motivated to replace the previous branch decomposition implementation because of a bug arising from a method: `compute_alignments`. As we explored this bug further, we discovered two key issues arising that the old implementation was not capable of/prepared to handle.

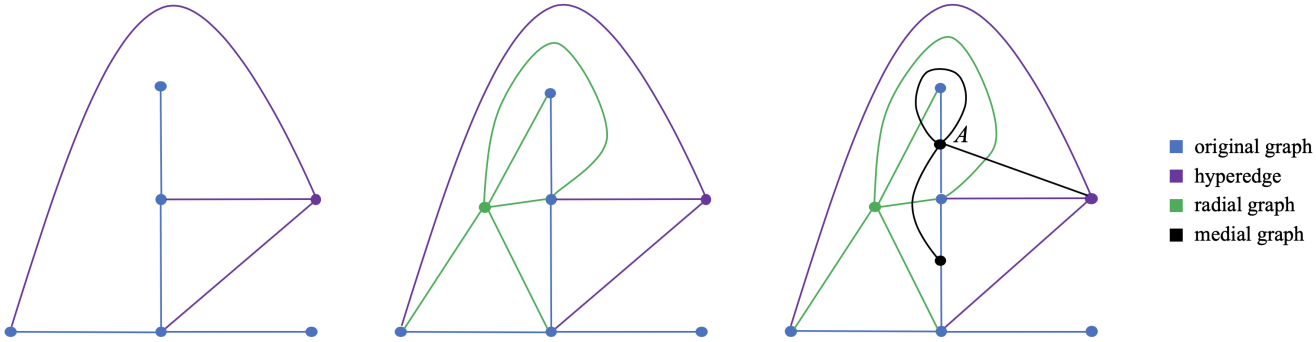

Figure 1: Example of a self-loop in the medial graph.

2.1 Self-Loops in the Medial Graph

The first issue that breaks the old branch decomposition implementation is possibility of self loops in the medial graph. Looking at Figure 1, observe that the medial node labeled A has a self loop. Naively adding both endpoints of the original edge represented by the medial node A is incorrect. The trivial cluster containing A should consist of a boundary of only the bottom vertex. In our new implementation, trivial clusters in the carving decomposition are computed by taking the orbit of a medial node. In the particular example from Figure 1, this would result in two darts of the same edge being added to a carving boundary twice. We introduce a simple check to disregard self-loops when forming the trivial carving clusters. Thus, our code will correctly construct the trivial cluster containing A with a boundary

[1] See `http://planarity.org/` and `https://github.com/pnklein/planarity`

consisting of two darts (the branch decomposition will only consist of one vertex because of the radial graph is bipartite in faces and vertices).

2.2 False Assumption of Simple Boundaries

Another issue that was discovered from the failure in `compute_alignment` was that boundaries are not necessarily simple. Instead, we can only guarantee that any given boundary is half-simple. This suggests that an original vertex can appear on a boundary of a branch decomposition cluster more than once. In Figure 2 below, we see that the node A exists on the red boundary twice.

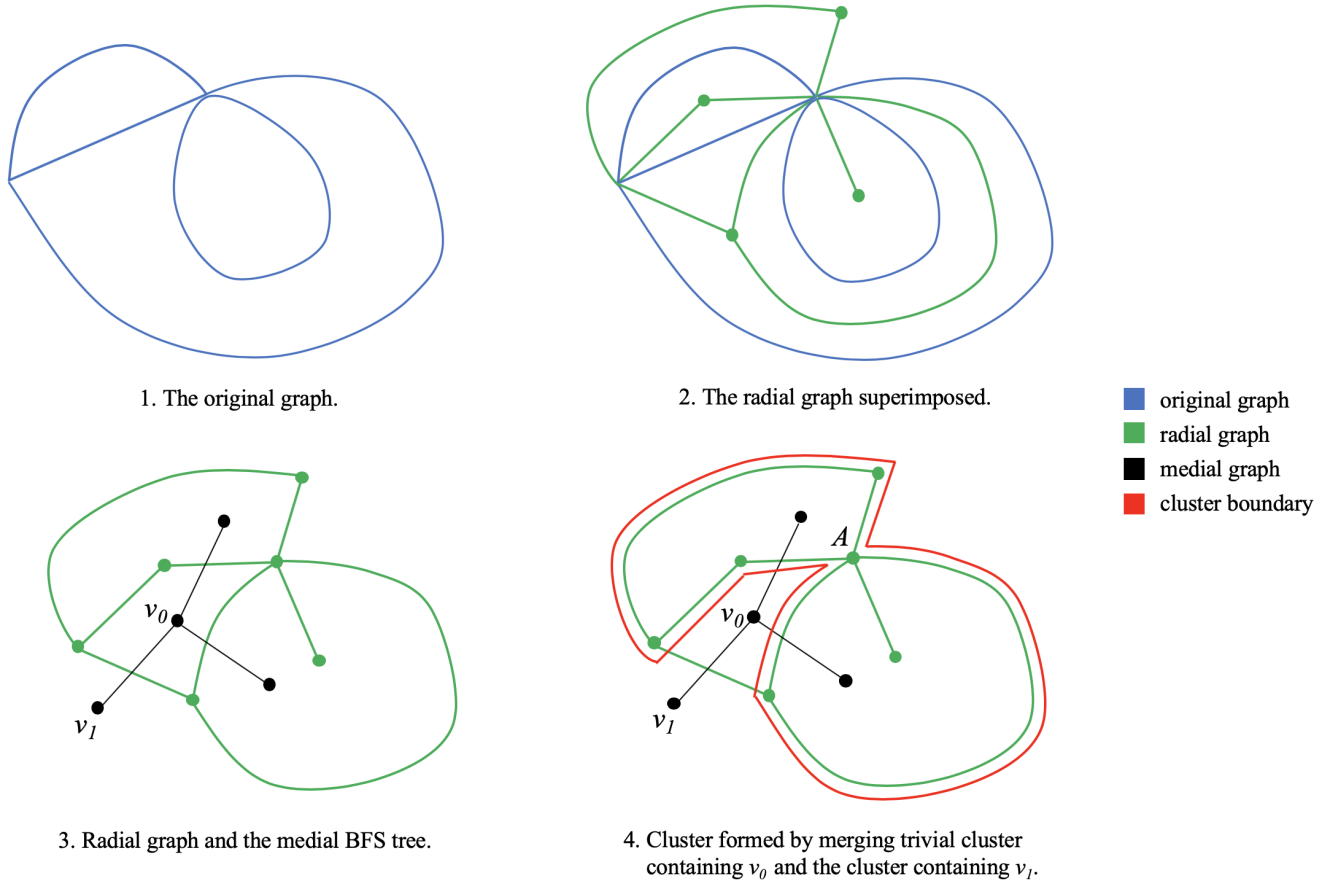

Figure 2: Example of a vertex appearing twice in a boundary.

The prior implementation assumes that all boundaries are simple through its usage of an `unordered_set<Node>` to store the boundary. This assumption also comes up in `compute_alignment` when we are assigning boundary ID's of the two child clusters from a parent cluster. In our implementation, we first form a carving decomposition defined by the medial BFS tree. We observe that even though an original vertex can appear twice on a branch decomposition boundary, a radial dart cannot show up more than once in a carving decomposition boundary. Thus, when forming carving decomposition clusters, this issue of duplicate vertices is abstracted away. When going from carving decomposition to branch decomposition, we allow for duplicate vertices by storing every boundary as a `vector<Node>`. In our implementation, computing alignments does not rely on the assumption of simple boundaries as we compute the alignment of the two child clusters using the in-order shared boundary of the parent cluster.

3 Our Implementation

This last section will get into the details of our implementation and discuss additional benefits besides fixing the two issues discussed above. Our implementation consists of two major components: the carving decomposition, then the branch decomposition.

3.1 Carving Decomposition

The carving decomposition code is straightforward. Clusters are formed recursively following the structure of the medial BFS tree. The majority of the carving decomposition code is dedicated to merging the boundaries of two clusters. Starting from the shared dart between the two clusters, we iterate in a consistent manner (clockwise in one, counter-clockwise in the other, and vice-versa) to compute the entirety of the shared boundary. After computing the shared boundary, we calculate the new boundary starting in the "parent" cluster (we call one of the clusters we are merging the parent because it is closer to the root in the medial BFS tree).

We argue that there are several advantages in computing the carving decomposition as an intermediary step to the branch decomposition (compared to going straight to the branch decomposition as the prior implementation does). In the prior implementation, the boundary and shared boundary for every cluster was maintained as an `unordered_set<Node>`. However, now that we are working directly with the medial and radial graphs, the vectors representing the boundary and shared boundary are in a valid ordering. Now, instead of having to check every vertex that is shared between two child boundaries is a corner, we only need to check if the two endpoints of the shared boundary are corners (conditional on the endpoints being vertex nodes and not face nodes).

3.2 Branch Decomposition

Our branch decomposition code takes in a carving decomposition as input. Turning a cycle of radial darts into a cycle of original vertices is trivial. The majority of the complexity comes from computing corners. If an end of a shared boundary is a vertex node (as opposed to a face node), we check the degree of that vertex within the cluster by mapping the incoming and outgoing radial darts in the boundary to original edges, then counting the original edges in between the two. We repeat this and add to the counter for every instance of the vertex on the boundary. Then, if our counter is not equal to the total degree of the vertex in the original graph, this vertex must be a corner. Otherwise, if they are equal, this vertex is not a corner. This method of computing corners, albeit complicated, removes the need to store a degree map within every cluster.

3.3 Other

Aside from the concrete solutions this new implementation has to offer, we argue that the new implementation is more robust. By running into many edge cases while testing, we feel confident in the correctness of this code. We've added many new checks (for example, to ensure the correctness of the shared boundary in the carving decomposition) that did not previously exist. We also argue that this new implementation is much more logical. Whereas the old implementation could be ambiguous and hard to read at times, our new implementation closely follows the structure of the graph and is more intuitive to follow. This all adds to the robustness of this part for the program, and will help us find and fix future bugs much more efficiently.