**Computing the Maximum Cut**

***A Branch and Cut Algorithm***

Sakibul Alam

*Emory University 2015*

*Rice University 2017*

Alex Gardner

*Rice University 2018*

Vincent Gonzales

*Rice University 2019*

***Table of Contents***

1. *Introduction………………………………………………………………………………………………. 3*
2. *The Algorithm ………………………………………………………………………………………….. 4*
   1. *Branch-and-Cut Technique*
   2. *Our Implementation - Generating Valid Cuts*
   3. *Lower Bound Heuristic*
3. *Computational Results ……………………………………………………………………………… 7*
   1. *Implementation and Run Time Results*
   2. *Computational Summary*
4. *Future Work …………………………………………………………………………………..………... 8*
5. *Concluding Remarks ……………………………………………………………………………….… 8*
6. *References ……………………………………………………………………………………………… 10*

***Introduction***

***Branch and Cut Algorithm***

*Branch and Cut Technique*

*Our Implementation – Generating Valid Cuts*

*Lower Bound Heuristic*

***Computational Results***

*Implementation and Run Time Results*

We implemented our solution in Python using Gurobi and ran all tests on a Late-2013 Macbook Pro with the following specifications: 2.6GHz Intel i5 Processor, 8GB RAM @ 1600 MHz DDR3. The results and run times for the TSP instances are shown in the table below.



The next table below summarizes the results for the planar graph instances we tested.



*\*All run times were gathered from our tests on the Macbook Pro mentioned above*

*Computational Summary*

The majority of our test cases completed after just one iteration, meaning that our algorithm never entered the branching phase. This was expected with the planar graphs as by definition they are weekly bipartite. Completing the algorithm in one iteration indicates that we add all of the necessary odd cycle cut constraints in the first iteration. However, on some of the larger graphs with hundreds of nodes and over one thousand edges, adding cut constraints became extremely time consuming. As a result, we concluded that while our method of adding cuts is effective in reaching the solution, it is likely making our solution computationally slower than it could be. The number of potentially violated cut constraints increases exponentially as the number of nodes and edges increase. As a result, we notice that the runtime increases exponentially as well, though more than we were anticipating for even the large planar instances.

Even though the planar graphs all finished after one iteration, our implementation performed best on the smallest TSP instances, gr21 and ulysses22 with both completing in under 2.5 seconds. Ulysses22 also completed in just one iteration as no branching was necessary to deliver the optimal solution. Gr21 took only five iterations, indicating that adding our initial set of cuts is our most time-intensive process. This furthers our suspicion that the solve time increases exponentially as the number of nodes and edges increases in a graph. As such, gr21 and ulysses22 have only 21 and 22 nodes respectively, and solutions were found quickly as a result. In considering the remaining travelling salesman instances, att48 solved to optimality in 14 iterations in just over three and a half minutes while hk48 was solved in slightly over a minute after just one iteration. Again, from just the travelling salesman instances alone we could deduce that process of adding all of the cut constraints initially is our most time intensive process.

Now we consider our planar test instances. As all of these completed in just one iteration, it is clear that we add hundreds of cuts in the first iteration which causes a bottleneck of sorts as a result. While we expected some of these instances to take a long time to complete due to their relative size, we were not expected our largest instances to take several hours to solve to optimality. This is likely due to Dijkstra’s algorithm. As we previously mentioned, in order to find the shortest odd cycle in a graph we build a “pseudo-graph” and use Dijkstra’s algorithm to find a shortest *(v, v’)*-path. In the largest instances, we do this thousands of times before we even branch. We run Dijkstra’s algorithm n times (where n is the number of nodes in the graph instance), before every optimization step. Then after we optimize, we run it n more times and repeat. We stop this process and branch when we go through the entire set of odd cycle constraints only to find that none of our found generated constraints are feasible. Running Dijsktra’s algorithm potentially hundreds of thousands of times on large instances is not feasible in practice, but it solves smaller instances to optimality quickly.

***Future Work***

***Concluding Remarks***

***References***