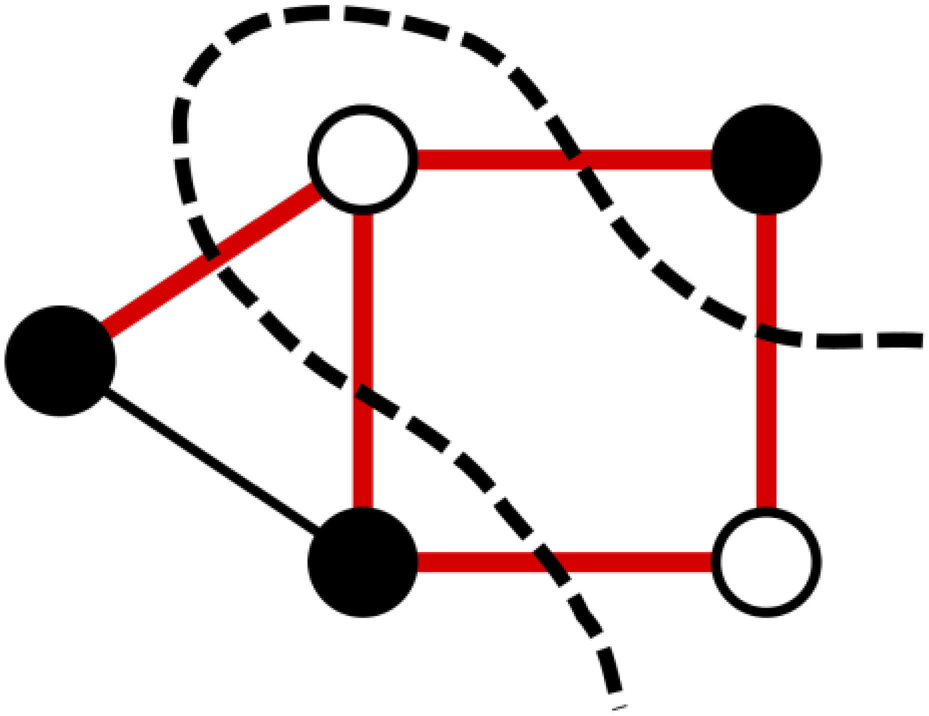
Computing the Maximum Cut
A Branch and Cut Algorithm



Sakibul Alam
Emory University 2015
Rice University 2017

Alex Gardner
Rice University 2018

Vincent Gonzales
Rice University 2019

Table of Contents

Introduction

One of Richard Karp's 21 NP-Complete problems, the weighted max cut problem seeks to find a cut that has a capacity greater than or equal to any other possible cut in a given weighted graph, *G = (V, E)*, where *V* and *E* are the nodes and edges of the graph, respectively. The problem boils down to an issue of set partitioning where we seek to separate a graph into two disjoint sets by removing the edges in the cut from *E*. In finding a maximum cut, we seek to remove the edges that have the costliest sum possible in the given graph.

There are several linear programming (LP) formulations to solve the max cut problem, but we will utilize a branch-and-cut technique to solve the formulation given below:

$$\begin{aligned}
&\textbf{Maximize} && \sum_{(u,v)\in E} e_{uv} \\
&\textbf{subject to} && e_{uv} \le \begin{cases} x_u + x_v \\ 2-(x_u+x_v) \end{cases}, \qquad \forall (u,v)\in E \\
& && x_u \in [0,1], \qquad \forall u \in V \\
& && e_{uv} \in [0,1], \qquad \forall (u,v)\in E
\end{aligned}$$

To solve our Max-Cut Problem, we will utilize a branch and cut technique where we relax the integer program above. In this relaxation, we will change our variables in such a way that our linear program will no longer select a solution that gives any information; instead it will set all of the edges to one and all of the nodes to one-half. As we would like a true integer solution, we eliminate non-integer solutions through a combination of branch-and-bound, and adding odd cycle constraints. However, adding a constraint for every odd cycle is extremely inefficient. Instead, we aim for the same optimal solution without necessarily incorporating every odd cycle constraint. Choosing these constraints in an effective manner is now the primary concern so as to obtain the optimal solution to the Max-Cut Problem. Using a branch-and-cut method, we shall find these and add these odd cycles to our relaxed LP formulation.

While a polynomial time algorithmic solution for general graphs is not known, we seek to implement a model that solves smaller graph instances quickly, perhaps enabling us to solve many real-world problems. There several applications for such and implementation such as very large-scale integrated (VLSI) circuits. These circuits have thousands of transistors placed on a single chip and one of the most pressing problems facing this technology is the necessity to minimize the number of vias. Vias are the pathways where wires move from one layer of the circuit to another. They pose a hardware concern since the addition of bored holes in the chip compromises its physical stability. In a model where each edge represents a segment where wires are allowed to pass through vias and each node is a specific area where a via is not present, a maximum cut solution will yield the desired objective result of minimizing the number of vias on the circuit. Another potential application occurs in planar graphs where it can be shown that the max cut is the dual to the Chinese postman problem, the problem of starting at a node, visiting every edge in a graph at least once with minimal distance, and returning back to the starting node. A solution to the max-cut problem could hold many applications.

Branch and Cut Algorithm

Branch and Cut Technique

By solving a linear program by branch and cut, we initially solve the LP relaxation of a program typically formulated as an integer program without any cut constraints. Then after solving the initial relaxation, we add odd cycle cutting constraints to our model necessary to remove limit infeasible, fractional max cuts. Then we re-solve the LP relaxation with these added constraints, find a fractional decision variable, and branch on that variable. In branching we create two new linear programs, each with a new constraint: one that forces the found fractional variable to zero and the other forcing the fractional variable to take on a value of one. We repeat this until there our queue of models is empty and we have solved the maximum cut problem. The general procedure for solving the max cut via branch and cut is summarized as follows:

1. Solve the LP relaxation of the Max Cut problem without any odd cycle constraints. In general, the optimal solution, x*, will not be integer-valued for all decision variables at this point.
2. Find odd cycle cutting constraints for every node in the graph, using x* as the edge-weights on our odd cycles. Cuts with odd cycle weight greater than one less than the size of the cycle will become the odd cycle cutting constraints.
3. Add these cuts with weight greater than the size of the cycle minus 1 to the initial LP relaxation and re-solve it.
4. Chose a non-integer variable, x^*_k and branch from it. When we branch on a variable, we create two copies of our original LP, one where $x^*_k \leq 0$ and another where $x^*_k \geq 1$.
5. In each branch, re-solve the new LP and return to step 2. Finish when x* is integer, there are no violated odd cycle constraints and there are no models remaining to compute solutions to.

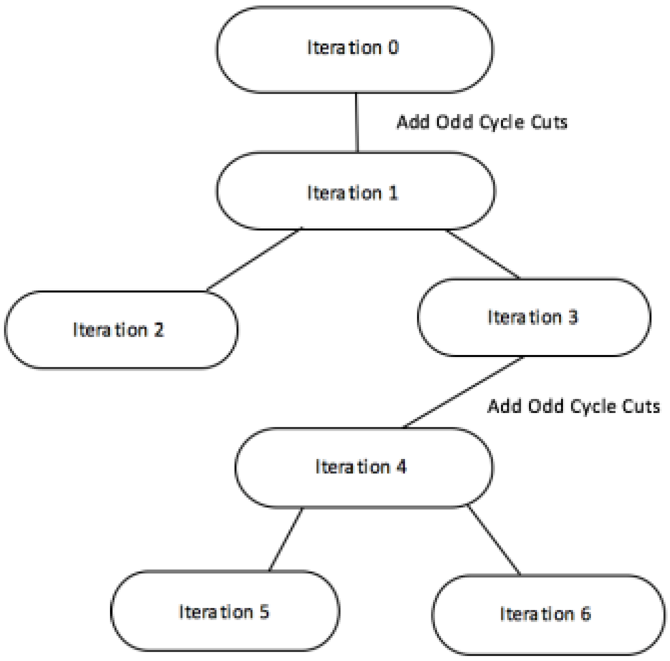


Figure 1: Our branch and cut implementation represented as a graph

Our Implementation – Generating Valid Cuts

The most interesting, and pertinent portion of our implementation centers around our constraint generation. It is as a result of our cutting planes that we are able to implement a branch and cut solution that "cuts" the non-integral solutions from our solution space. We first consider our odd cycle cutting constraints:

$$\sum_{(u,v)\epsilon S} e_{\mathrm{uv}} < 1 - |S|;\ |S|\ odd$$

This is equivalent to:

$$\sum_{(u,v)\epsilon S} (1 - e_{\mathrm{uv}}) < 1;\ |S|\ odd$$

By relaxing our integer program to be continuous, we require that there be no odd cycles whose total edge weight sum is greater than one less than the number of edges in the cycle. If such a cycle exists, we are not optimal as this is not a feasible solution in the original formulation. Thus, we seek to find odd cycles whose weight is greater than one less than the number of elements in the cycle.

To find these cycles, we will implement the algorithm outlined in the pseudocode below:

```
model = build_odd_cuts(model):
"""
Takes in a model, builds odd cycle constraints, and returns the model with the
violated odd cycle constraints added in.
"""
flag ← True
while flag:
    added_list ← [ ]
    pseudograph ← build_pseudograph(model)
    for node in model.vars:
        constraint, violated_flag ← build_constraint(pseudograph, node)
        if violated_flag and constraint not in model.constrs:
            added_list.add(violated_flag)
            model.addConstr(constraint)
    model.optimize()
    If model integer or sum(added_list) == 0:
        flag ← False
return model
```

We run the above algorithm on every model throughout our branch and cut algorithm. As one can see, after we add a bulk of cuts, one for every node in the graph at maximum, we optimize our model to see if we have found an integral solution. If we have, there is no reason to add any more cuts, so we terminate and move to the next model in our queue, if there is one.

The most critical part of the algorithm above is held in build_constraint but before we can explore how we do that, we must build a pseudograph. This is what we will use in order to find the shortest odd cycle from each node. As such we mush construct it before we can utilize it. We will now detail the pseudograph construction. For some graph $G = (V, E)$, we form $G' = (V', E')$ as a pseudograph of G. Both our edge set and vertex set are different from G as we now have twice as many vertices and edges in G' as we did in G. G' is constructed as follows:

- Make a duplicate of all of the vertices in G. We will denote the original vertices as $v \epsilon G'$ and the copies as $v' \epsilon G'$
- There exists an edge $e_{uv'} \epsilon G'$ with cost c_{uv} if $e_{uv} \epsilon G$ with cost c_{uv}
- There exists an edge $e_{u'v} \epsilon G'$ with cost c_{uv} if $e_{uv} \epsilon G$ with cost c_{uv}

An example of this construction is detailed below.

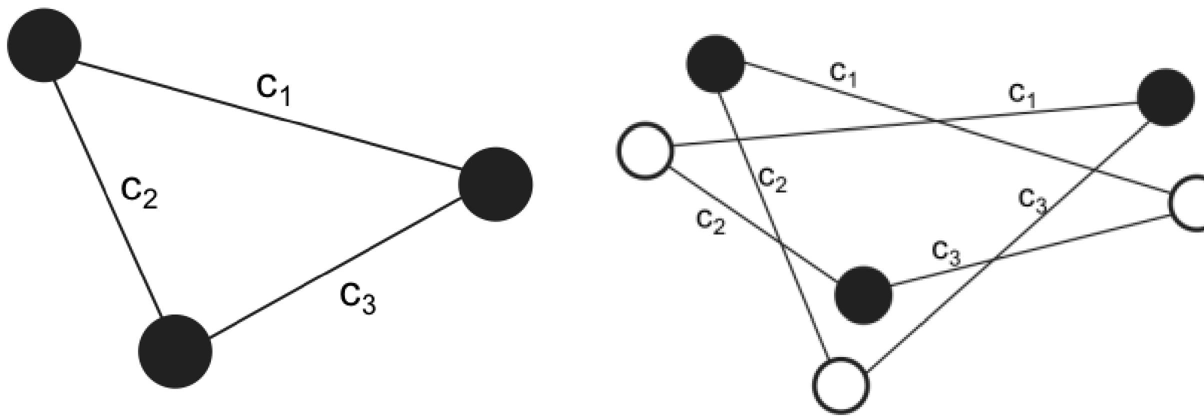


Figure 2: An example of the pseudograph construction with G on the left and G' on the right

One can then prove that from a given node $u \epsilon G'$, all odd paths end in nodes $v' \epsilon G'$, and all even paths end in nodes $v \epsilon G'$. We will not detail this proof here, but we utilize the result to construct our odd cycles.

In order to find the shortest odd cycles, we first find the shortest odd path from some node $v \epsilon G'$ to $v' \epsilon G'$. We eventually do this by running Dijkstra's algorithm on G' with the starting node $v, \forall v \epsilon G'$. Once we have our odd path, we need to make sure that v, v' is the only pair of nodes corresponding to the same node in G that is in our path. This guarantees that we have formed a cycle in G and that we have no repeated edges. We then build our constraint of all of the edges in our cycle and check to see if the sum of the optimal edge weights greater than $1 - |S|$, as in the odd cycle cutting constraint above. If it is, we return a flag along with the constraint that adds our new constraint to our model. The addition of this constraint is successful so long as the same constraint is not already added in the model, in which case we do not add it. Once we do this for every node in G, we optimize. If our optimal solution is integer, we stop adding cuts. Otherwise we try to add more cuts as our edge weights have been updated as a result of our latest optimization. If before we achieve an integer optimal solution we reach a point where there are no longer any constraints added, we stop adding constraints and proceed back to the main branch and bound algorithm. This method, though sometimes time consuming, enables us to greedily add cutting constraints to arrive and an integer optimal solution.

Lower Bound Heuristic

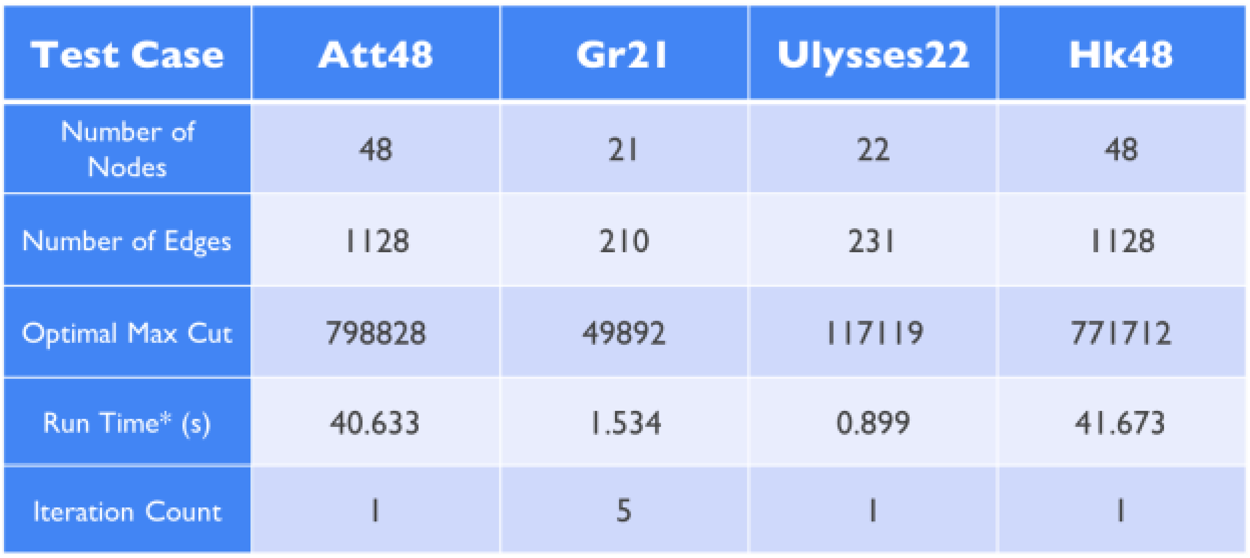
Our lower bound heuristic does very little in actually limiting the number of computations that we run in the branch and cut stage of the implementation. We note that as we are computing the maximal cut of a graph, and other valid cut forms a valid lower bound on a given graph's maximum cut. Thus, we can naively compute a feasible solution at the beginning of the algorithm that will form a lower bound on the maximum cut value. In theory, with this bound we no longer need to visit as many branches as we would otherwise. This is because any branch that produces an optimal cost lower than the best cost for a previously found integer solution terminates the current branch. The following implementation is particularly naïve and in many cases, it does not make the algorithm terminate any faster than it otherwise would; it is far more likely that we will find a higher lower bound within the first few stages of the branch and cut algorithm, thereby improving our lower bound almost immediately.

To compute an initial feasible solution, we create an arbitrary partition in the graph using a ½-approximation greedy algorithm. In this we assign each of the nodes to be in one of two sets with equal probability. It is the separation between these two sets that forms our partition that we utilize for our initial feasible solution. This solution is clearly integer as we set all of the edges between the two sets to take on a value of one and all others to take a value of zero. We then add all of the edges that go between the two sets to set our initial lower bound.

Computational Results

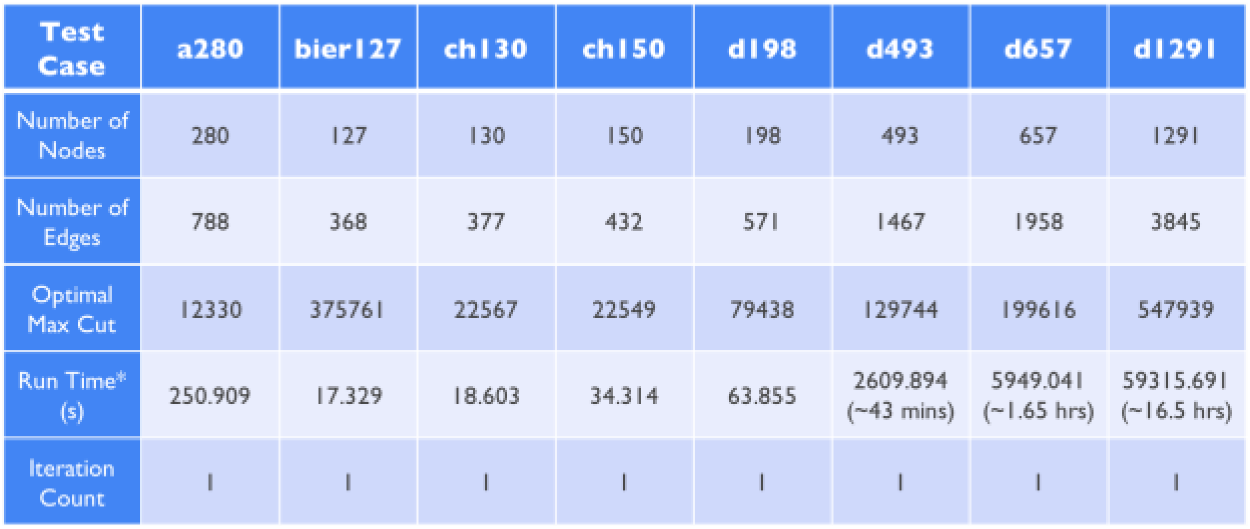
Implementation and Run Time Results

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.



Test Case	Att48	Gr21	Ulysses22	Hk48
Number of Nodes	48	21	22	48
Number of Edges	1128	210	231	1128
Optimal Max Cut	798828	49892	117119	771712
Run Time* (s)	40.633	1.534	0.899	41.673
Iteration Count	1	5	1	1

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



Test Case	a280	bier127	ch130	ch150	d198	d493	d657	d1291
Number of Nodes	280	127	130	150	198	493	657	1291
Number of Edges	788	368	377	432	571	1467	1958	3845
Optimal Max Cut	12330	375761	22567	22549	79438	129744	199616	547939
Run Time* (s)	250.909	17.329	18.603	34.314	63.855	2609.894 (~43 mins)	5949.041 (~1.65 hrs)	59315.691 (~16.5 hrs)
Iteration Count	1	1	1	1	1	1	1	1

**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 1.6 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 and hk48 both solved to optimality in 1 iteration in just under a minute. 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 Dijkstra's algorithm potentially hundreds of thousands of times on large instances is not feasible in practice, but it solves smaller instances to optimality quickly.

Concluding Remarks

Overall, the Maximum Cut Problem has unique applications that will increase in variety as technological innovations arise. In our branch-and-cut method of solving the max-cut problem, we were able to handle the issue of odd cycles in our constraints by relaxing the initial integer program. The number of constraints in the problem formulation is highly variable as a result of the algorithm used to generate them. By adding additional nodes to our graph, the number of additional constraints is exponential, especially in complete graphs. By computing the odd cycles from each node through an implementation of Dijkstra's algorithm, we added a relatively low number of odd cycle constraints, thereby allowing us to reach our optimal solution faster. However, our implementation does create opportunities for future improvement, namely in building our odd cycles. In general, our branch-and-cut implementation produces generally efficient results, but there are several aspects of the algorithm's implementation that can be considered to enhance performance.

Future Work

In implementing our branch and cut algorithm, while effective on small instances, we discovered several potential areas of improvement that could ultimately lead to an increase in the algorithm's performance on larger graphs. The first of these that we encountered is our implementation of Dijkstra's algorithm. Each time we add find an odd cycle, we run Dijkstra's algorithm on the pseudograph described in the section regarding our algorithm. For a batch of n potential constraints, one for each node in the original graph, we run Dijkstra's algorithm n times. This becomes unwieldy when the number of nodes and edges increase in the graph. Dijkstra's then takes considerably longer to run, and we need to run it more times. Furthermore, there is no guarantee that running Dijkstra's algorithm was even necessary for a given node as the generated odd cycle may not even correspond to a violated constraint, and thus running the algorithm does not even result in adding a new constraint to our linear program. For this there are several possible improvements, namely an implementation of Dijkstra's that runs more quickly on sparse graphs. Since such modifications to Dijkstra's exist, implementing it would likely lower our running time on the planar graphs significantly. Furthermore, in adding our constraints, we run a check to make sure that the computed constraint is not already in our linear program. With this, we do not then find a new, slightly costlier odd cycle. Instead we move on to the next node and find an odd cycle from there. This allows the possibility that we are repeatedly trying to add odd cycle cutting constraints that are either already in the model or do not cut off a part of the solution. Implementing either of these solutions would likely reduce the algorithm's run time significantly.

We now consider two minor improvements that have the potential to decrease the running time on extremely large and dense instances. We consider first the implementation of a depth-first-like solution as opposed to the current breadth-first solution. For large instances that likely require a lot of branching, our solution maintains a queue of models that we then add cuts to and solve. Maintaining a large queue is unwieldy, and can be avoided through a DFS implementation. In this, we will only maintain a single model, but we will add and remove constraints from it as necessary according to where we are in our branch and bound tree. This will likely get us to an integer solution in fewer iterations, eliminating more potential solutions in the process. Further, having a large queue is extremely taxing from a memory perspective, so maintaining just one model should reduce run time for large instances that develop longer queues as the algorithm progresses. Finally, the implementation of a better lower bound algorithm also has the potential to decrease the run time significantly. The max cut problem is APX-hard, meaning that there is no polynomial time approximation scheme that is arbitrarily close to the optimal solution unless P=NP. However, there are lower bound algorithms better than the one that we have implemented. Our half approximation algorithm partitions the nodes into two sets by putting them in one set or another with 50% probability. The edges between our two arbitrary sets form our initial cut. However, Goemans and Williamson implemented an approximation using semi-definite programming and randomized rounding that is far more accurate than our implementation. While it may not help on smaller instances, implementing Goemans and Williamson's approximation algorithm may also decrease the number of computations run throughout the branch and cut phase of our implementation.

References

Cook, William J., William H. Cunningham, William R. Pulleyblank, Alexander Schrijver. *Combinatorial Optimization*. New York: Wiley, 1997. Print.

Goemans, Michel X.; Williamson, David P. (1995), *"Improved approximation algorithms for maximum cut and satisfiability problems using semidefinite programming"*

Verweij, Bran; Aardal, Karen (1999), *"An Optimisation Algorithm for Maximum Independent Set with Applications in Map Labelling"*