**5/6/2005**

**Revised: 5/20/2005**

# Vertex Cover Problem with Hard Capacities

**Completed by Colin Dixon as part of the requirement for departmental honors in computer science at the University of Maryland in College Park under the advisement of Samir Khuller.**

**Abstract**

Over the past 2 years we have worked on the capacitated vertex cover problem with hard capacities which can be expressed as follows. Given a graph G = (V,E) pick the minimum number of vertices such that all edges are assigned to a vertex, but each vertex, v in V, can only have a certain number, $k_v$, edges assigned to it. Specifically, we have been working on implementing and trying to improve the 2-approximation algorithm for capacitated vertex cover presented by Samir Khuller. We implemented the algorithm as well as a variety of specialized classes for dealing with graphs using Java and the CPLEX linear program solver. We examined several thousand test graphs looking for interesting trends in the solutions to the linear program which could be exploited to refine the algorithm and/or it's analysis. This included writing tools which looked over output from various stages of the algorithm for a variety of patterns. In the end little fruitful information was gained, but at least one promising pattern was disproved and a set of tools which could ease further research were developed.

**Introduction**

The vertex cover problem is one of the most studied problems in approximation algorithms. It is an even more interesting problem because there are no current approximation algorithms which achieve an approximation factor which is asymptotically less than 2, even though there is no proof that this must be the case. Many different generalizations of the problem have also been studied including looking at situations where each vertex can only cover a certain number of edges (capacitated vertex cover) and situations where only a certain number of the edges need to be covered (partial vertex cover).

Some of these generalizations have led to interesting results in that some generalizations result in different approximation factors, while others manage to maintain the conjectured 2-approximation bound for the regular vertex cover problem. For instance the weighted, capacitated vertex cover problem with hard capacities is known to be at least as hard as

the set cover problem meaning that there is no constant factor approximation algorithm.

The particular generalization to the vertex cover problem which we considered at was the unweighted, capacitated vertex cover problem with hard capacities, which is a generalization of the unweighted, capacitated vertex cover problem. The problem can be stated as follows: Given a graph G = (V,E) pick the minimum number of vertices such that all edges are assigned to a vertex, but each vertex, v in V, can only have a certain number, $k_v$, edges assigned to it. This is separated from the weighted version because all vertices are assumed to have equal weight and thus cost the same amount to include in the cover, and from the version with soft capacities in that a vertex can only be picked once. In the version with soft capacities vertices can be picked multiple times and each time allows for another $k_v$ edges to be assigned to that vertex.

**The Algorithm**

We examined the solution to the unweighted capacitated vertex cover problem with hard capacities proposed by Gandhi et al. in "An Improved Approximation Algorithm For Vertex Cover with Hard Capacities." The paper gives an algorithm and a proof that it provides a 2-approximation, but both the algorithm and the proof are somewhat complicated. The algorithm involves six steps, and the proof is long and difficult to follow. Additionally, the algorithm is randomized and relies on solving a linear program which many other solutions to generalizations of the vertex cover problem do not. For those reason Samir Khuller thought that there should be a way to improve either the algorithm, the proof or both. we worked with him on and off for about a two years implementing the algorithm (and a variety of tools to support it) and then looking for patterns in the solution data.

The basic idea of the algorithm is to do a little bit of preprocessing such that the capacity 1 vertices are picked to be in the solution or out of the cover before the rest of the algorithm. The fact that there are only capacity 2 and higher vertices left after this plays a key role in the analysis and proof of a 2 approximation bound. Then the linear program

relaxation of the integer program representing the problem is solved, and through a reasonably complex process vertices are put in the cover until a valid solution with integrally picked vertices, but maybe fractionally assigned edges is formed. At this point the integrality of flows guarantees that we can find an integral assignment for the edges giving a valid solution which has expected cost of at most twice the optimal solution.

The portion which we studied in the most detail was the solutions to the formulated linear program which is shown below. In the linear program, $x_v$ represents the amount which a vertex v has been picked and $y_{ev}$ represents how much an edge e has been assigned to one of it's vertices v. It is easy to see that if $x_v$ and $y_{ev}$ were binary variables taking on values of only 0 or 1, that the resulting integer program would be exactly equivalent to optimally solving the problem.

$$\text{Minimize } \sum_{v \in V} x_v \text{ such that,}$$
$$y_{eu} + y_{ev} = 1 \quad \forall e = (u, v) \in E$$
$$k_v x_v - \sum_{e \in E(V)} y_{ev} \geq 0 \; \forall v \in V$$
$$x_v \geq y_{ev} \quad \forall v \in e \in E$$
$$y_{ev} \geq 0 \quad \forall v \in e \in E$$
$$0 \leq x_v \leq 1 \quad \forall v \in V$$

The first step of the algorithm is to look at all capacity one vertices and see if they are required for the solution, this is tested by creating a special form of network flow problem in which a bipartite graph is formed in which one set of vertices represents the edges of the original graph and the other set represents the vertices of the original graph. Edges are placed connecting the vertices representing edges to the vertices representing their endpoints with capacity for 1 unit of flow. The source is connected to all of the vertices representing edges with edges that can support 1 unit of flow, and finally the sink is connected to each vertex representing a vertex from the original graph with the same capacity as the vertex had in the original node. If at this point the maximum flow through the newly constructed flow network is greater than or equal to |E| there is a feasible

solution.

In this way each capacity one vertex can be “removed” by setting it's capacity to 0 and then the graph can be tested for feasibility. If the graph has a feasible solution that vertex's capacity is left at 0. The capacity one vertices which are required for feasibility are automatically picked by setting the corresponding $x_v$ variable to 1.

Once this preprocessing is done the LP is constructed and solved. At this point the vertices whose corresponding $x_v$ variable is greater than or equal to ½ are included in the cover automatically. This does not guarantee a solution, so the remaining vertices are included with probability equal to $2x_v$, for the corresponding $x_v$ variable. This still does not guarantee a solution, so in a relatively complicated manner a few more vertices are selected for the cover. These vertices are paid for out of the excess that was paid in order to include all of the vertices with $x_v$ larger than ½, or $\sum_{v \in V} (2x_v - 1)$ . At this point we have a feasible solution in which the vertices are integrally picked and assigned to the cover or not, but the edges may be fractionally assigned to vertices.

Finally the same flow network used in the feasibility checking is constructed and the edges are integrally assigned to vertices using the integrality of flows property which guarantees a solution in which the constraints are still satisfied, but that each edge carries an integral amount of flow.

The analysis is significantly more complicated than the algorithm and can be found in the referenced paper, but the most complicated portion of the analysis deals with how to charge the vertices added to the cover in the last step.

**Experiments and Results**

The main idea in this project was that by looking at the way the algorithm functioned and the way the linear program solutions looked, we might be able to find some kind of

pattern which could be exploited to either simplify the algorithm itself or find some property that would enable to proof to go through in simpler manner.

To this end, we constructed several thousand sets of random graphs of varying size and ran them through the algorithm to gather a significant amount of information in which to search for patterns. From the smaller graphs which we had generated at first, we had conjectured that in the linear program solution $x_v = y_{ev}$ for all edges assigned to a vertex, but in the larger test graphs this was not always the case. In order to make sure that this was not due to any coding errors in the algorithm we looked for a smaller example which would still violate this condition.

After some more experimentation we was able to find a relatively simple graph which was hand checkable so that we could actually determine whether we had disproved this property. The graph has 10 vertices and 20 edges and results in one vertex being seven tenths picked, but all of the edges are assigned in quarters. See the end of this paper for the full graph, formulated linear program and solution.

From there we set out looking for some other patterns in the data, we thought that maybe the fractional picking of vertices and fractional assignment of edges would all be multiples of some unit fraction, and this is usually the case (actually always the case since you can just find the common denominator), but we could find no relation between the obvious properties of the graph and that unit fraction.

**Implementation**

We decided to implement the algorithm in Java because it's heavily object oriented nature lent itself to construction the objects which would be required to build the algorithm and it enables relatively quick development. After looking for a good set of classes for representing and working with graphs, we decided that it would probably be easiest to simply implement my own because we could add the functionality which was required for the specific task. The only problems implementing my own set of graph classes in Java

created for me were that we could not capitalize on the many different random graph generation tools which are available (and largely written in C) and secondly we could not make use of a more optimized network flow algorithm and ended up implementing a rather simple push-relabel algorithm, which probably ends up being the Achilles heel of the implementation.

Another small implementation glitch is that the algorithm makes use of solving a linear program, and so this required the use of an external tool. For my implementation we used the CPLEX linear program solver because it was what we had access to. The Java algorithm writes a linear program in the format CPLEX understands and then reads the output to see the solution which is obviously not nearly as easily changed to use another linear program solver as would be desirable.

The implementation essentially all stems from the graph class which implements all of the core functionality the algorithm requires. It can store arbitrary graphs, and also allows for one piece of auxiliary information to be stored. This gave a relatively flexible way to use the graph class in a more general sense. we used the auxiliary information to store state while running the push-relabel maximum network flow algorithm and also used it to store capacities while running the general algorithm. The graph class also allows for quick random generation of graphs as well as random assignment of capacities.

Random graphs are created by specifying a number of edges and a number of vertices. The edges are then placed by repeatedly picking two endpoints uniformly at random from the vertex set. Capacities can be assigned to the graph in a variety of ways. First a global distribution for capacity can be specified (either uniform or normal), as well as a simple approach which assigns each vertex a capacity equal to a certain fraction of its degree.

The final tool which bears speaking of is the analyzer which processes the output from the CPLEX linear program solver. It provides a relatively simple way to look up information about linear program solution, including looking up any vertex or edge and seeing all of the related information through a command line interface. When a vertex is

looked up it shows all adjacent edges and and how much they have been assigned to that edge, and when an edge is looked up it shows the two related vertices and how much of each vertex was assigned. It can also be easily configured to display the ratios between any of the related values as well as to check if a given ratio or other property holds for the entire solution.

## Conclusions

After several experiments and tests, we were unable to find any patterns in the results which were useful to simplifying either the algorithm or the analysis. Despite this, we were able to disprove at least one of the more promising patterns which appeared in the smaller test graphs, and were able to create a successful implementation of the algorithm and create a reasonably general and extensible library of graph classes for Java which could be used to continue the research into this and other graph theoretic problems. Just because we were unable to find exploitable patterns in the data does not mean that they are not there to be found, in fact, it is still likely that there is some substantially simpler algorithm and/or analysis for this problem.

## Appendix

Sample graph for which the property that $x_v = y_{ev}$ does not hold:

```
Verticies:
  <vID: 0, vCAP: 6>
  <vID: 1, vCAP: 3>
  <vID: 2, vCAP: 4>
  <vID: 3, vCAP: 3>
  <vID: 4, vCAP: 3>
  <vID: 5, vCAP: 4>
  <vID: 6, vCAP: 6>
  <vID: 7, vCAP: 4>
  <vID: 8, vCAP: 4>
  <vID: 9, vCAP: 5>
Edges:
  edge 0 from 1 to 7
  edge 1 from 4 to 0
  edge 2 from 9 to 9
  edge 3 from 8 to 4
  edge 4 from 8 to 8
```

```
  edge 5 from 7 to 2
  edge 6 from 2 to 9
  edge 7 from 7 to 6
  edge 8 from 1 to 8
  edge 9 from 6 to 2
  edge 10 from 2 to 0
  edge 11 from 0 to 6
  edge 12 from 4 to 9
  edge 13 from 3 to 1
  edge 14 from 9 to 7
  edge 15 from 8 to 5
  edge 16 from 1 to 5
  edge 17 from 6 to 9
  edge 18 from 0 to 9
  edge 19 from 1 to 6
```

CPLEX Linear Program:

```
minimize
x0 + x1 + x2 + x3 + x4 + x5 + x6 + x7 + x8 + x9

such that
y0_1 + y0_7 = 1
y1_4 + y1_0 = 1
y2_9 + y2_9 = 1
y3_4 + y3_8 = 1
y4_8 + y4_8 = 1
y5_2 + y5_7 = 1
y6_2 + y6_9 = 1
y7_6 + y7_7 = 1
y8_1 + y8_8 = 1
y9_2 + y9_6 = 1
y10_0 + y10_2 = 1
y11_0 + y11_6 = 1
y12_4 + y12_9 = 1
y13_3 + y13_1 = 1
y14_7 + y14_9 = 1
y15_5 + y15_8 = 1
y16_1 + y16_5 = 1
y17_9 + y17_6 = 1
y18_9 + y18_0 = 1
y19_6 + y19_1 = 1

6x0 - y1_0 - y10_0 - y11_0 - y18_0 >= 0
3x1 - y0_1 - y8_1 - y13_1 - y16_1 - y19_1 >= 0
4x2 - y5_2 - y6_2 - y9_2 - y10_2 >= 0
3x3 - y13_3 >= 0
3x4 - y1_4 - y3_4 - y12_4 >= 0
4x5 - y15_5 - y16_5 >= 0
6x6 - y7_6 - y9_6 - y11_6 - y17_6 - y19_6 >= 0
4x7 - y0_7 - y5_7 - y7_7 - y14_7 >= 0
4x8 - y3_8 - y4_8 - y4_8 - y8_8 - y15_8 >= 0
5x9 - y2_9 - y2_9 - y6_9 - y12_9 - y14_9 - y17_9 - y18_9 >= 0

x1 - y0_1 >= 0
x7 - y0_7 >= 0

x4 - y1_4 >= 0
x0 - y1_0 >= 0
```

```
x9 - y2_9 >= 0
x9 - y2_9 >= 0

x4 - y3_4 >= 0
x8 - y3_8 >= 0

x8 - y4_8 >= 0
x8 - y4_8 >= 0

x2 - y5_2 >= 0
x7 - y5_7 >= 0

x2 - y6_2 >= 0
x9 - y6_9 >= 0

x6 - y7_6 >= 0
x7 - y7_7 >= 0

x1 - y8_1 >= 0
x8 - y8_8 >= 0

x2 - y9_2 >= 0
x6 - y9_6 >= 0

x0 - y10_0 >= 0
x2 - y10_2 >= 0

x0 - y11_0 >= 0
x6 - y11_6 >= 0

x4 - y12_4 >= 0
x9 - y12_9 >= 0

x3 - y13_3 >= 0
x1 - y13_1 >= 0

x7 - y14_7 >= 0
x9 - y14_9 >= 0

x5 - y15_5 >= 0
x8 - y15_8 >= 0

x1 - y16_1 >= 0
x5 - y16_5 >= 0

x9 - y17_9 >= 0
x6 - y17_6 >= 0

x9 - y18_9 >= 0
x0 - y18_0 >= 0

x6 - y19_6 >= 0
x1 - y19_1 >= 0


y0_1 >= 0
y0_7 >= 0
y1_4 >= 0
y1_0 >= 0
y2_9 >= 0
```

```
y2_9 >= 0
y3_4 >= 0
y3_8 >= 0
y4_8 >= 0
y4_8 >= 0
y5_2 >= 0
y5_7 >= 0
y6_2 >= 0
y6_9 >= 0
y7_6 >= 0
y7_7 >= 0
y8_1 >= 0
y8_8 >= 0
y9_2 >= 0
y9_6 >= 0
y10_0 >= 0
y10_2 >= 0
y11_0 >= 0
y11_6 >= 0
y12_4 >= 0
y12_9 >= 0
y13_3 >= 0
y13_1 >= 0
y14_7 >= 0
y14_9 >= 0
y15_5 >= 0
y15_8 >= 0
y16_1 >= 0
y16_5 >= 0
y17_9 >= 0
y17_6 >= 0
y18_9 >= 0
y18_0 >= 0
y19_6 >= 0
y19_1 >= 0

x0 >= 0
x0 <= 1

x1 >= 0
x1 <= 1

x2 >= 0
x2 <= 1

x3 >= 0
x3 <= 1

x4 >= 0
x4 <= 1

x5 >= 0
x5 <= 1

x6 >= 0
x6 <= 1

x7 >= 0
x7 <= 1
```

```
x8 >= 0
x8 <= 1

x9 >= 0
x9 <= 1

End
```

CPLEX Linear Program Solution:

```
Welcome to CPLEX Linear Optimizer 6.5.2
  with Mixed Integer & Barrier Solvers
Copyright (c) ILOG 1997-1999
CPLEX is a registered trademark of ILOG

Type 'help' for a list of available commands.
Type 'help' followed by a command name for more
information on commands.

CPLEX> Problem 'out3.lp' read.
Read time =    0.01 sec.
CPLEX> Tried aggregator 1 time.
LP Presolve eliminated 66 rows and 2 columns.
Aggregator did 18 substitutions.
Reduced LP has 46 rows, 28 columns, and 118 nonzeros.
Presolve time =    0.00 sec.

Iteration log . . .
Iteration:     1    Infeasibility =            29.499999
Iteration:    10    Objective     =             9.500000

Primal - Optimal:  Objective =    5.2000000000e+00
Solution time =    0.00 sec.  Iterations = 51 (9)

CPLEX> Variable Name           Solution Value
x0                            0.500000
x1                            1.000000
x2                            0.500000
x4                            0.500000
x5                            0.250000
x6                            0.500000
x7                            0.500000
x8                            0.750000
x9                            0.700000
y0_1                          0.500000
y0_7                          0.500000
y1_4                          0.500000
y1_0                          0.500000
y2_9                          0.500000
y3_4                          0.500000
y3_8                          0.500000
y4_8                          0.500000
y5_2                          0.500000
y5_7                          0.500000
y6_2                          0.500000
y6_9                          0.500000
y7_6                          0.500000
y7_7                          0.500000
y8_1                          0.250000
```

```
y8_8                          0.750000
y9_2                          0.500000
y9_6                          0.500000
y10_0                         0.500000
y10_2                         0.500000
y11_0                         0.500000
y11_6                         0.500000
y12_4                         0.500000
y12_9                         0.500000
y13_1                         1.000000
y14_7                         0.500000
y14_9                         0.500000
y15_5                         0.250000
y15_8                         0.750000
y16_1                         0.750000
y16_5                         0.250000
y17_9                         0.500000
y17_6                         0.500000
y18_9                         0.500000
y18_0                         0.500000
y19_6                         0.500000
y19_1                         0.500000
All other variables in the range 1-48 are zero.
CPLEX> CPLEX> CPLEX>
```