CS4234: Optimisation Algorithms Lecture 1

The Limits of Tractability: MIN-VERTEX-COVER

V1.0: Seth Gilbert, V1.1: Steven Halim *August 15, 2017*

Abstract

Today, we are talking about the MIN-VERTEX-COVER problem. MIN-VERTEX-COVER is a classic NP-hard optimization problem, and to solve it, we need to compromise. We look at three approaches. First, we consider restricting our attention to a special case: a (binary) tree. Then, we look at an exponential algorithm parameterized by the size of the vertex cover. Finally, we look at approximation algorithms (both a deterministic approach and a simple randomized approach) and analyze their approximation ratios (both are 2-approximation).

1 Overview

Today, we will study the problem of MIN-VERTEX-COVER, a classical NP-hard optimization problem. MIN-VERTEX-COVER is a great model problem to think about at the beginning of the semester because, on the one way, it has a relatively simple structure and the algorithms we will see today are quite simple; on the other hand, it illustrates several of the basic techniques we will use this semester, and many problems that show up in the real world are quite similar to MIN-VERTEX-COVER.

We begin by defining the problem.

Definition 1 *A **vertex cover** for a graph $G = (V, E)$ is a set $S \subseteq V$ such that for every edge $e = (u, v) \in E$, either $u \in S$ or $v \in S$.*

That is, every edge is covered by one of the vertices in the set S.

Definition 2 *The* **MIN-VERTEX-COVER** *problem is defined as follows: Given a graph $G = (V, E)$, find a minimum-sized set S that is a vertex cover for G.*

Imagine, for example, that you are given a map of Singapore and you want to choose where to open Starbucks coffee shops to ensure that, no matter where you are in Singapore, there is a Starbucks on a nearby corner. (Assume that you have decided to open your coffee shops only at an intersection.) If you find a vertex cover, you can be sure that nobody in Singapore is too far away from a Starbucks! But is your vertex cover of minimum possible size? See Figure 1 for an example.

As we will see shortly in Section 2, MIN-VERTEX-COVER is a hard problem—specifically it is NP-hard. We are not likely to come up with an efficient algorithm (unless $P = NP$).

When you have a hard computational problem, there are three things that you want a solution to be:

1. Fast (i.e., polynomial time)
2. Optimal (i.e., yielding the best solution possible)
3. Universal (i.e., good for all instances/inputs)

But when you have an NP-hard problem like MIN-VERTEX-COVER, you can only have two of these three[1]! You can sacrifice speed, and have an algorithm that runs in exponential time. You can sacrifice optimality, and explore heuristics

[1] These options appear in challenging programming contest problems, see [3].

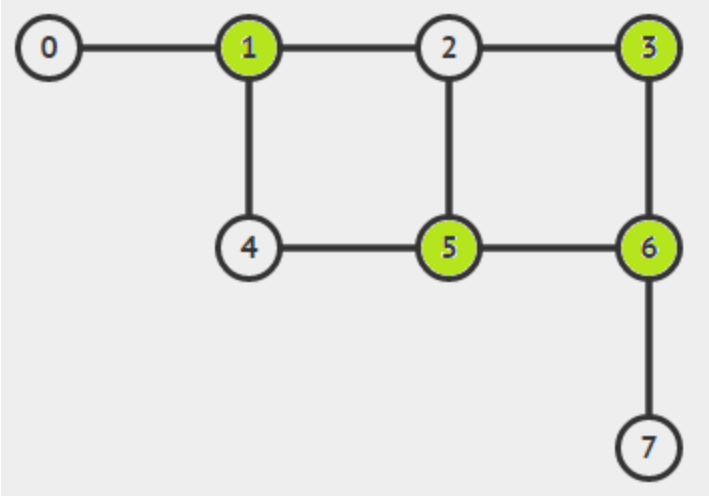

Figure 1: Example of a graph with a vertex cover of size 4. The green highlighted vertices are in the vertex cover. Is 4 the size of the MIN-VERTEX-COVER? If yes, is the solution unique? If no, can you find MIN-VERTEX-COVER of size 3 or less?

and approximation algorithms—try to find a solution that is *good enough*[2]. Or you can sacrifice universality, and focus on solving special cases. Today, we will talk about these three different approaches for coping with an otherwise intractable problem.

1. First, we consider the case where the input to your problem is somehow special (and easier than the general case); specifically, we will look at finding a MIN-VERTEX-COVER on a (binary) tree. (Here we sacrifice universality.)
2. Second, we will see a *parameterized* solution that can guarantee good performance when the MIN-VERTEX-COVER being sought is small (e.g., we cannot afford to open more than 10 Starbucks, regardless). In this case, we given an algorithm that achieves an exponential-time solution that is exponential only in the parameter k, not the size of the problem n. (Here we sacrifice speed: It is exponential-time, but hopefully reasonable.)
3. Third, we will consider approximation algorithms that do not find an *optimal* solution, but find a solution that is not too far from optimal. We will discuss a deterministic 2-approximation algorithm and also a randomized 2-approximation algorithm. (Here we sacrifice optimality, finding a solution that is "good enough.")

2 NP-completeness

Before we get to algorithms, let us quickly review[3] the proof that VERTEX-COVER is NP-complete (for details, see Chapter 34 of [1])[4].

It is relatively easy to show that VERTEX-COVER (as a decision problem) is NP-complete. First, it is easy to observe that it is in NP: Given a set S of size k, we can easily verify whether or not it is a vertex cover simply by checking whether, for every edge, one of the two endpoints can be found in S.

[2] "Le mieux est l'ennemi du bien," according to Voltaire. ("The best is the enemy of the good.")

[3] Recall CS3230!

[4] To be a bit pedantic, it is actually a mistake to refer to MIN-VERTEX-COVER as NP-complete. The complexity class NP refers only to *decision* problems, not to *optimization* problems. The version of MIN-VERTEX-COVER that *is* NP-complete is the VERTEX-COVER problem described as follows: Given a graph $G = (V, E)$ and an integer k, does graph G has a vertex cover of size k (vertices)? Clearly, if we could efficiently find a MIN-VERTEX-COVER, then we could also efficiently answer the decision version of the question. Hence solving the optimization version of MIN-VERTEX-COVER is at least as hard as the decision version, and hence we term it NP-hard.

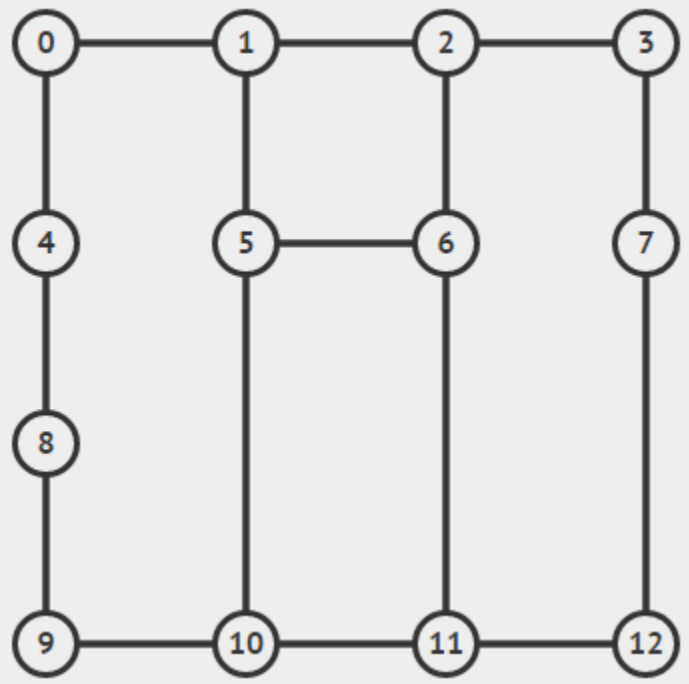

Figure 2: Mini challenge: What is the minimum-sized vertex cover of this graph?

Next, we have to show that it is NP-hard. We can show this by reducing some other NP-hard problem to VERTEX-COVER, thus showing that solving VERTEX-COVER is at least as hard as solving this other problem. The original NP-complete problem is 3-SAT (see Section 34.4 of [1] or Chapter 3 of [2]), and we could[5] reduce 3-SAT to VERTEX-COVER. Instead, we focus on a different problem: CLIQUE, that of finding the a clique[6] of size k in a graph.

Definition 3 *The (decision version of)* **CLIQUE** *problem is defined as follows: Given a graph $G = (V, E)$ and an integer k, is there a subset $C \subseteq V$ of size k (vertices) such that C is a clique in G?*

The goal of the CLIQUE problem is to determine whether a given graph G contains a clique of size k. CLIQUE is known to be NP-complete[7]. We can show that VERTEX-COVER is NP-hard by giving an algorithm for solving CLIQUE by using VERTEX-COVER, that is, we reduce CLIQUE $\leq_P$ VERTEX-COVER. First, we need to define the ***complement*** of a graph G, i.e., the graph that contains only the edges that are *not* in G:

Definition 4 *Given a graph $G = (V, E)$ we define the* ***complement***[8] *of G to be the graph $G' = (V, E')$ where edge $e \in E'$ if and only if $e \notin E$.*

We can now give an algorithm for solving the CLIQUE problem. Assume we have an algorithm VC-ALGO which solves the problem of VERTEX-COVER. On an input of graph $G = (V, E)$ with n vertices, and an integer k, execute the VERTEX-COVER algorithm on G', the complement of G; return **true** if and only if G' has a VERTEX-COVER of size $\leq n - k$.

[5] But we wouldn't, as it is too complicated.

[6] A clique is a set of vertices $K \subseteq V$ such that every pair of vertices in K is connected by an edge.

[7] We can also say upfront that VERTEX-COVER is known to be NP-complete (e.g. from CS3230, a.k.a. 'proof from previous module') without going through all these proofs in this Section 2. However, for the sake of CS3230 review, we assume that we 'don't know' that VERTEX-COVER is NP-complete but we somehow recall that CLIQUE is NP-complete.

[8] We can compute the complement of a graph G in $O(V^2)$ time.

Algorithm: Clique($G = (V, E)$, k)
Procedure:
    /* computable in polynomial time $O(V^2)$ */
    Let G' be the complement of G.
    Execute VC-ALGO on G' with parameter $n - k$.
    if $\exists$ *vertex cover of* G' *of size* $\leq n - k$ **then**
        return true.
    else return false.

The correctness of this algorithm for CLIQUE immediately shows that VERTEX-COVER is NP-complete and depends on the following claim:

Lemma 5 *Graph $G = (V, E)$ with n vertices has a clique of size at least k if and only if its complement G' has a vertex cover of size at most $n - k$.*

The complete proof can be reviewed in Section 34.5.2 of [1].

3 MIN-VERTEX-COVER on a (Binary) Tree

While finding a minimum sized vertex cover is NP-hard in general, there are many special cases that are tractable. Consider an example where the graph $G = (V, E)$ is a rooted binary tree. In this case, we can find a MIN-VERTEX-COVER in linear time using Dynamic Programming (DP). Here we outline the solution for determining the *size* of the MIN-VERTEX-COVER. It is relatively easy to turn this idea into an algorithm for finding the MIN-VERTEX-COVER.

For each vertex v in the tree, we define two variables:

- $in(v)$ is the size of the minimum vertex cover for the sub-tree of G rooted at v, assuming that v is contained in the vertex cover.
- $out(v)$ is the size of the minimum vertex cover for the sub-tree of G rooted at v, assuming that v is not contained in the vertex cover.

Notice that if r is the root of the tree, the size of the minimum vertex cover is simply $\min[in(r), out(r)]$.

We can now traverse the tree starting from the leaves up to the root calculating for every vertex v both $in(v)$ and $out(v)$. For a leaf, we can trivially define $in(v) = 1$ and $out(v) = 0$. Now consider a vertex v in the tree with children w and z.

- To calculate $out(v)$: In this case, both children w and z *must* be included in the vertex cover. Thus $out(v) = in(w) + in(z)$.
- To calculate $in(v)$: In this case, we can either include or not include w or z in the tree, depending on which solution is better. In either case, we add 1 to the count, since we are including v. Thus, $in(v) = 1 + \min[in(w), out(w)] + \min[in(z), out(z)]$.

It is easy to see that this calculation can be performed in linear time, and that it results in the MIN-VERTEX-COVER. (The proof would proceed by induction over the height of the tree.)

Note (in addition) that the same idea can be used for more general trees, and also for weighted variants of MIN-VERTEX-COVER (where each vertex has a cost for including it in the vertex cover) — discussed in the next Lecture

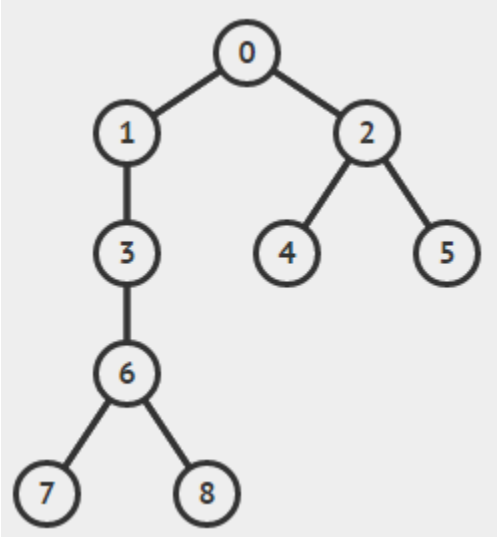

Figure 3: Example of a Binary Tree. Apply the Dynamic Programming algorithm discussed in this section to obtain the MIN-VERTEX-COVER of this Binary Tree.

02. A sample implementation of MIN-VERTEX-COVER (unweighted version) on not-necessarily-binary trees in C++ is given in Section 4.7.1 of [4].

4 Parameterized Complexity

Another special case is when you know, in advance, that the vertex cover is small. For example, if you are told that the vertex cover is up to $k = 2$ vertices, then there is an easy polynomial time algorithm: Simply test every pair of vertices (and every individual vertex[9]) and check whether that vertex is a valid vertex cover. Assuming that you can check the vertex cover of a graph $G = (V, E)$ with n vertices and m edges in time $O(m)$, then this algorithm runs in time $O(n^2m)$.

More generally, if you know that the vertex cover is of size at most k, you can find it in time $O(n^k m)$. Unfortunately, as n gets large, the term n^k becomes prohibitively large and the algorithm rapidly becomes unusable, even for small values of k. For example, if $k = 10$, $n = 100$, $m = 1\,000$ and you use a CPU that can perform 10 billion instructions per second (e.g., an Intel Core i7), then it would take more than 300 thousand years to find the vertex cover.

Instead, we would like an algorithm that runs in time $O(f(k)g(n, m)$, for some functions f and g where g is a polynomial in n and m. That is, the algorithm may be exponential in k, but the exponential function should not have any dependence on n and m. If k is small, this might yield a usable algorithm.

Here's the basic idea. For every edge $e = (u, v)$, we know that either u or v is in the vertex cover. So let's consider both cases. For every vertex $z \in V$, define G_{-z} to be the graph G where z and all edges adjacent to z have been deleted. If we can find a vertex cover of size $k - 1$ in either G_{-u} or G_{-v}, when we can construct a vertex cover for G. The key lemma here is as follows:

Lemma 6 *Let $G = (V, E)$ be a graph and let $e = (u, v)$ be an edge in E. The graph G has a vertex cover of size k if and only if either G_{-u} or G_{-v} has a vertex cover of size $k - 1$.*

(We leave the proof as an exercise.) This yields the following algorithm:

[9]There is a possibility that the answer is just 1 vertex, so we also need to check for this possibility.

```
1  Algorithm: ParameterizedVertexCover(G = (V, E), k)
2  Procedure:
       /* Base case of recursion:  ⊥ = failure, ∅ = empty set                */
3      if k = 0 and E ≠ ∅ then return ⊥
4      if E = ∅ then return ∅
       /* Recurse on arbitrary edge e:                                       */
5      Let e = (u, v) be any edge in G.
6      S_u = ParameterizedVertexCover(G_{-u}, k − 1)
7      S_v = ParameterizedVertexCover(G_{-v}, k − 1)
8      if S_u ≠ ⊥ and |S_u| < |S_v| then
9          return S_u ∪ {u}
10     else if S_v ≠ ⊥ and |S_v| ≤ |S_u| then
11         return S_v ∪ {v}
12     else return ⊥
```

The algorithm picks an arbitrary edge $e = (u, v)$, and recursively searches for a vertex cover of size $k - 1$ in G_{-u} and G_{-v}. When it has explored both options, it then return the smaller of the two vertex covers (adding either u or v as is appropriate).

Notice that this recursive algorithm has the following structure: At every step we remove one vertex from the graph and add it to the vertex cover, and we recurse twice. As a base case, if we have removed k vertices from the graph, and yet we have not found a vertex cover (i.e., some edges still remain in the graph), then we abort and return failure. Another base case is when we have no more edge to be covered (and $k \geq 0$), in this case we return an empty set.

Assuming that the initial graph G has a vertex cover of size at most k, the running time of this algorithm is captured by the following recurrence:

$$T(k, m) \leq 2 \times T(k - 1, m) + O(m)$$

(here we assume that it takes approximately $O(m)$ time to delete a vertex and its adjacent edges from a graph, and we ignore the fact that the number of edges is reduced in the recursive calls).

Solving this recurrence, we find that $T(n) = O(2^k m)$. For the setting above ($k = 10$, $n = 1\,000$), this improved algorithm would run in just about 0.1ms :O...

5 Approximation Algorithms

Alas, sometimes a problem is still too hard to solve. The graph does not satisfy any of the (known) special cases. The parameterized complexity is *still* too large (i.e., the vertex cover is not sufficiently small). In this case, we cannot guarantee that you will find an optimal solution.

Instead, we relax our goals: We will not find an *optimal* solution, but we will find an *almost* optimal solution. This is the goal of an approximation algorithm. Consider, as an example, the problem of MIN-VERTEX-COVER. For a given graph G, define $OPT(G)$ to be a minimal-sized vertex cover[10].

Definition 7 *An algorithm A is an* ***α-approximation algorithm*** *for* MIN-VERTEX-COVER *if, for every graph G:*

$$|A(G)| \leq \alpha \times |OPT(G)| \,.$$

[10]Notice that we often seem to implicitly assume there is only *one* possible optimal solution. This is not true! There may be many possible optimal vertex covers. Here we just choose one.

That is, an algorithm A is a good approximation[11] algorithm for MIN-VERTEX-COVER if it guarantees that it will find a vertex cover that is not too much larger than the best possible vertex cover, e.g. see Figure 4.

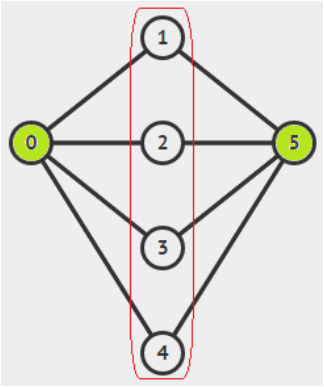

Figure 4: Example of a graph with an optimal vertex cover of size 2 (the two green-highlighted vertices: Vertex $\{0, 5\}$). The red-highlighted vertices (Vertex $\{1, 2, 3, 4\}$) in the middle also form a vertex cover of size 4. This vertex cover of size 4 is not optimal. To be precise, it is two times larger than the optimal vertex cover.

In the context of approximation algorithms, MIN-VERTEX-COVER is a particularly interesting problem to study. It is very easy[12] to design a 2-approximation algorithm (and we will see two such algorithms today).

5.1 Deterministic Approximation Algorithm

First, we consider a deterministic algorithm for approximating MIN-VERTEX-COVER. There are three natural candidate algorithms (see below). Each of these algorithms greedily adds vertices to the cover, with progressing attempts at cleverness. The first algorithm simply adds arbitrary vertices until every edge is covered. The second algorithm considers the edges in arbitrary order, but adds *both* endpoints. The third algorithm is the 'cleverest', greedily adding the vertex that covers the most edges that remain uncovered. Stop for a minute and think about which algorithm you think is best.

```
/* This algorithm adds vertices greedily, one at a time, until everything
   is covered.  The edges are considered in an arbitrary order, and for
   each edge, an arbitrary endpoint is added.                              */
Algorithm: ApproxVertexCover-1(G = (V, E))
Procedure:
    C ← ∅
    /* Repeat until every edge is covered:                                 */
    while E ≠ ∅ do
        Let e = (u, v) be any edge in G.
        C ← C ∪ {u}
        G ← G_{−u} // Remove u and all adjacent edges from G.
    return C
```

[11] $\alpha = 1$ means an optimal algorithm. α is > 1 for all approximation algorithms for a minimisation problem and Computer scientists aim to make α as close as possible to 1.

[12] Whether $\alpha = 2$ is a good bound or not depends on the requirements. There are other possible algorithms that we will learn later in this course that can reach better performance than the 2-approximation algorithms that we study today even though they have no such bound.

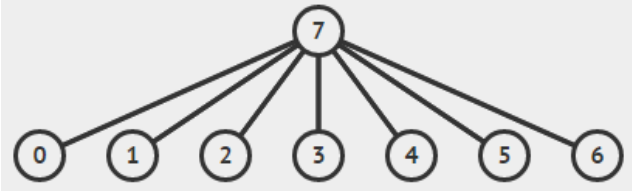

Figure 5: Counterexample for the simple vertex cover algorithm 1 where an *arbitrary* vertex is added in each step. When each edge is considered, we can be very unlucky whereby the leaf vertex is always chosen, yielding a vertex cover of size $n - 1$. The minimum vertex cover contains only vertex 7.

```
/* This algorithm adds vertices greedily, two at a time, until everything
   is covered.  The edges are considered in an arbitrary order, and for
   each edge, both endpoints are added.                                    */
Algorithm: ApproxVertexCover-2(G = (V, E))
Procedure:
    C ← ∅
    /* Repeat until every edge is covered:                                */
    while E ≠ ∅ do
        Let e = (u, v) be any edge in G.
        C ← C ∪ {u, v}
        G ← G_{−{u,v}} // Remove u and v and all adjacent edges from G.
    return C
```

```
/* This algorithm adds vertices greedily, one at a time, until everything
   is covered.  At each step, the algorithm chooses the next vertex that
   will cover the most uncovered edges.                                    */
Algorithm: ApproxVertexCover-3(G = (V, E))
Procedure:
    C ← ∅
    /* Repeat until every edge is covered:                                */
    while E ≠ ∅ do
        Let d(x) = number of uncovered edges adjacent to x.
        Let u = argmax_{x∈V} d(x)
        C ← C ∪ {u}
        G ← G_{−{u}} // Remove u and all adjacent edges from G.
    return C
```

It turns out that neither Algorithm 1 nor Algorithm 3 achieve a good approximation ratio. See Figure 5 for a graph where Algorithm 1 can perform poorly and Figure 6 for a graph where Algorithm 3 can perform poorly. We now proceed to analyze `ApproxVertexCover-2` and show that it is a 2-approximation algorithm. This may seem somewhat surprising: We are wastefully adding *both* endpoints of an edge e, when only one would suffice. On the other hand, thinking back to the analysis of the randomized algorithm, we know that each time at least *one* of the two vertices being added is a "correct" vertex, i.e., one that is added by $OPT(G)$. This guarantees that we get a vertex cover that is at most twice as big as optimal.

We now try to formalize this intuition. For pedagogic purposes, we are going to do this a little more carefully and less directly. In general, the technique for analyzing an approximation algorithm is to find some way to bound the size of OPT. If we can show that OPT cannot be too small, then it is easier to show that our algorithm is close to OPT. To this end, we are going to define another combinatorial concept:

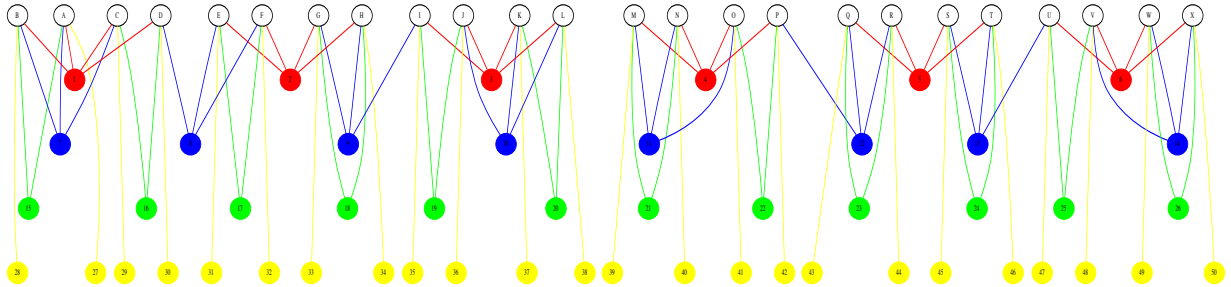

Figure 6: Counterexample for the greedy vertex cover algorithm 3. In this example, $n = 4$. The top row consists of $n! = 24$ white vertices, each of degree $n = 4$. The next row consists of $n!/n = 6$ red vertices, each of degree $n = 4$. The next row consists of $n!/(n-1) = 8$ blue vertices of degree 3. The next row consists of $n!/(n-2) = 12$ green vertices of degree 2. The last row consists of $n!/(n-3) = 24$ yellow vertices of degree 1. Notice that there is a vertex cover of size $n!$, i.e., all the white vertices. However, the greedy algorithm might first add the red vertices (reducing the degree of every white vertex by one), then add the blue vertices (reducing the degree of every white vertex by one), then add the green vertices, and then the yellow vertices. The total number of vertices added to the vertex cover is $n! \sum_{i=1}^{n}(1/i) = n! \log n$, which is a factor of $\log n$ from optimal.

Definition 8 *Given a graph $G = (V, E)$, we say that a set of edge $M \subseteq E$ is a **matching** if no two edges in M share an endpoint, i.e., $\forall e_1, e_2 \in M : \ e_1 \cap e_2 = \emptyset$.*

A matching is a set of edges that pairs up vertices: Each vertex in the matching gets exactly one partner. Note that a matching can be of any size, consisting of one edge or up to $\lfloor n/2 \rfloor$ edges. If every vertex in the graph is adjacent to an edge in the matching, then it is a ***perfect matching***. We will later revisit this problem in the middle of the semester.

The key observation here is that for every edge in a matching, one of the two endpoints *must* be in the vertex cover. This yields the following lemma:

Lemma 9 *Let M be any matching of graph $G = (V, E)$. Then $|OPT(G)| \geq |M|$.*

Proof Observe that for every edge $e = (u, v)$ in the matching M, either u or v has to be in the vertex cover $OPT(G)$. Moreover, since M is a matching, neither u nor v cover any other edges in M. Thus every edge in M must have a unique vertex in $OPT(G)$, and hence $|OPT(G)| \geq |M|$. □

This lemma gives us a nice way of showing that our vertex cover approximation algorithm is good: All we need to do is show that there exists some matching that is not too much smaller than the vertex cover it produces. Luckily, the algorithm constructs a matching as it executes:

Lemma 10 *Let E' be the set of edges considered by* `ApproxVertexCover-2` *during its execution. Then E' is a matching.*

Proof After each edge $e = (u, v)$ in E' is considered, both the vertices u and v are removed from the graph G. Thus no edge considered later can contain either u or v. (Formally, the proof proceeds by induction, maintaining the invariant that for every edge e in E', neither endpoint of e remains in G and hence the new edge being considered can be safely added to E'.) □

Combining these two lemmas yields our final claim:

Theorem 11 `ApproxVertexCover-2` *is a 2-approximation algorithm for vertex cover.*

Proof For a given graph $G = (V, E)$, let C be the set output by the algorithm and let E' be the edges considered. Notice that $|C| = 2 \times |E'|$, since for every edge in E' we added two vertices to C. Since E' is a matching, we know that $|OPT(G)| \geq |E'|$, and hence we conclude:

$$\begin{aligned} |C| &= 2 \times |E'| \\ &\leq 2 \times |OPT(G)| \end{aligned}$$

□

5.2 Randomized Approximation Algorithm

Next, we consider a very simple randomized approach: For every edge in the graph, simply flip a coin and randomly choose one of the two endpoints of the edge to add to the vertex cover. Repeat this process until every edge is covered.

Algorithm: RandomizedVertexCover($G = (V, E)$)
Procedure:
  $C \leftarrow \emptyset$
  `/* Repeat until every edge is covered: */`
  while $E \neq \emptyset$ **do**
    Let $e = (u, v)$ be any edge in G.
    $b \leftarrow$ **Random**$(0, 1)$ `// Returns` $\{0, 1\}$ `each with probability` $1/2$.
    if $b = 0$ **then** $z = u$
    else if $b = 1$ **then** $z = v$
    $C \leftarrow C \cup \{z\}$
    $G \leftarrow G_{-z}$ `// Remove` z `and all adjacent edges from` G.
  return C

It is easy to see that, when the algorithm completes, the resulting set C is a vertex cover: Each edge is removed from G only when it is covered by a vertex that has been added to C.

We now argue that the resulting vertex cover is, in expectation, at most twice the size of $OPT(G)$. The intuition here is that each time we add a vertex to C, we have (at least) $1/2$ probability of being right: For every edge $e = (u, v)$, either $u \in OPT(G)$ or $v \in OPT(G)$ (or both of them are), and so with probability (at least) $1/2$ we choose the correct one. Thus we would expect C to be only about twice as large as OPT.

To make this intuition a little more formal, we define two sets: $A = C \cap OPT(G)$ and $B = C \setminus OPT(G)$. The set A contains all the "correct" elements that we have added to C, i.e., all the elements that are in both C and $OPT(G)$. The set B contains all the "incorrect" elements that we have added to C, i.e,. all the elements that are in C but not in $OPT(G)$. For every edge e considered by the algorithm, define the indicator random variable:

$$X_e = \begin{cases} 0 & \textbf{if} \quad \text{the vertex added to } C \text{ after considering } e \text{ is } \textbf{not} \text{ in } OPT \\ 1 & \textbf{if} \quad \text{the vertex added to } C \text{ after considering } e \text{ is in } OPT \end{cases}$$

Notice that $\mathrm{E}[X_e] \geq 1/2$, since the probability of choosing the correct vertex is at least $1/2$. Let E' be the set of edges considered by the algorithm. Since $|A| = \sum_{E'} X_e$, we conclude that:

$$\begin{aligned}
\mathrm{E}\left[|A|\right] &= \mathrm{E}\left[\sum_{E'} X_e\right] \\
&= \sum_{E'} \mathrm{E}\left[X_e\right] \qquad (\textit{linearity of expectation}) \\
&\geq |E'|/2 \; .
\end{aligned}$$

Similarly, we see that $|B| = \sum_{E'}(1 - X_e)$, and hence $\mathrm{E}\left[|B|\right] \leq |E'|/2$. Putting these two together, we observe that:

$$\mathrm{E}\left[|B|\right] \leq \mathrm{E}\left[|A|\right] \; .$$

The final observation we need is that $|A| \leq |OPT(G)|$ (since A only contains vertices that are also in $OPT(G)$), and hence $\mathrm{E}\left[|A|\right] \leq |OPT(G)|$. Putting together the preceding equations, we see:

$$\mathrm{E}\left[|B|\right] \leq \mathrm{E}\left[|A|\right] \leq |OPT(G)| \; .$$

Finally, we conclude that calculation:

$$\begin{aligned}
\mathrm{E}\left[|C|\right] &= \mathrm{E}\left[|A|\right] + \mathrm{E}\left[|B|\right] \qquad (\textit{linearity of expectation}) \\
&\leq |OPT(G)| + |OPT(G)| \\
&\leq 2 \times |OPT(G)|
\end{aligned}$$

This yields the following theorem:

Theorem 12 `RandomizedVertexCover` *yields a 2-approximation for* MIN-VERTEX-COVER *(in expectation).*

Note: Since April 2017, Steven and his FYP student: Muhammad Rais has created `https://visualgo.net/en/mvc` visualization tool that covers many parts of this lecture [5].

References

[1] Thomas H. Cormen, Charles E. Leiserson, Ronald L. Rivest, and Cliff Stein. *Introduction to Algorithm*. MIT Press, 3rd edition, 2009.

[2] Michael R. Garey and David S. Johnson. *Computers and Intractability: A Guide to the Theory of NP-Completeness*. W.H. Freeman and Company, 1979.

[3] Jonathan Irvin Gunawan. Understanding Unsolvable Problem. *Olympiads In Informatics*, 10:87–98, 2016.

[4] Steven Halim and Felix Halim. *Competitive Programming: The New Lower Bound of Programming Contests*. Lulu, 3rd edition, 2013.

[5] Steven Halim and Muhammad Rais. VisuAlgo - Min Vertex Cover Visualization, howpublished= https://visualgo.net/en/mvc.

Index