**CS4234: Optimization Algorithms** Lecture 1

# The Limits of Tractability: Vertex Cover

*Lecturer: Seth Gilbert* *August 11, 2015*

**Abstract**

Today we are talking about the problem of *vertex cover*. Vertex cover is a classic NP-hard 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 tree. Then, we look at an exponential algorithm parameterized by the size of the vertex cover. Finally, we looked at approximation algorithms (both a simple randomized approach and a deterministic approach).

## 1 Overview

Today we will study the problem of **vertex cover**, a classical NP-hard problem. 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 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 nodes in the set $S$.

**Definition 2** *The* ***Minimum 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 no one in Singapore is too far away from a Starbucks! See Figure 1 for an example.

As we will see shortly, 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 problem like vertex cover, there are three things that you want a solution to be:

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

When you have an NP-hard problem, you can only have two of these three! You can sacrifice speed, and have an algorithm that runs in exponential time. You can sacrifice optimality, and explore heuristics and approximation algorithms—try to find a solution that is *good enough*[1]. Or you can sacrifice universality, and focus on solving special cases. Today, we will talk today about these three different approaches for coping with an otherwise intractible problem.

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

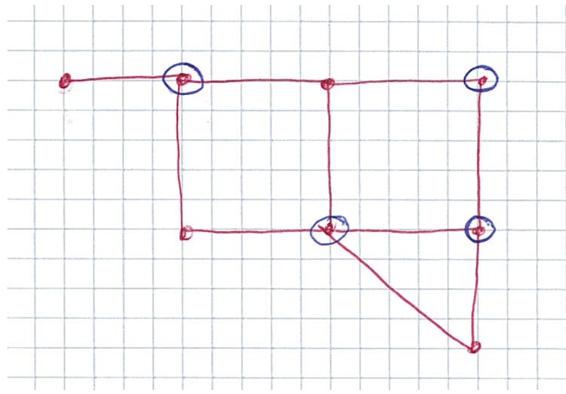

**Figure 1:** Example of a graph with a vertex cover of size 4. The circled nodes are in the vertex cover.

- 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 vertex cover on a tree. (Here we sacrifice universality.)
- Second, we will see a *parameterized* solution that can guarantee good performance when the 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.)
- Finally, we will consider approximation algorithms that do not find an *optimal* solution, but a find a solution that is not too far from optimal. We will discuss a randomized 2-approximation algorithm, and also a deterministic 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 the proof that Vertex Cover is NP-complete.[2]

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 at most $k$, 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 $S$.

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 3SAT, and we could reduce 3SAT to Vertex Cover. Instead we focus on a different problem, that of finding the largest clique[3] in a graph.

**Definition 3** *The **Max-Clique** problem is defined as follows: given a graph $G = (V, E)$ and an integer $k$, is there a subset $K \subseteq V$ of size at least $k$ such that $K$ is a clique in $G$?*

The goal of the Max-Clique problem is to determine whether a given graph $G$ contains a clique of size at least $k$. Max-Clique is known to be NP-complete. We can show that Vertex Cover is NP-hard by giving an algorithm for

[2] To be a bit pedantic, it is actually a mistake to refer to Vertex Cover as NP-complete. The complexity class NP refers only to decision problems, not to optimization problems. The version of Vertex Cover that *is* NP-complete is as follows: given a graph $G = (V, E)$ and a parameter $k$, does there exist a vertex cover of $G$ of size at most $k$? Clearly, if we could efficiently find a minimum sized vertex cover, then we could also efficiently answer the decision version of the question. Hence finding a minimum sized vertex cover is at least as hard as the decision version, and hence we term it NP-hard.

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

solving Max-Clique by using 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*** *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 Max-Clique problem. Assume we have an algorithm $VC$ which solves the problem of vertex cover. On an input of graph $G = (V, E)$ with $n$ nodes, and parameter $k$, execute the vertex cover algorithm on the complement of $G$; return **true** if and only if $G'$ has a vertex cover of size $\leq n - k$. The correctness

**Algorithm:** MaxClique($G = (V, E)$, $k$)
**Procedure:**
    Let $G'$ be the complement of $G$.
    Execute `VertexCover` on $G'$ with parameter $n - k$.
    **if** $\exists$ *vertex cover of $G'$ of size $\leq n - k$* **then**
        **return true**.
    **else return false**.

of this algorithm for Max-Clique immediately shows that Vertex Cover is NP-complete and depends on the following claim:

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

The proof is left as an exercise for the problem set.

# 3 Vertex Cover on a Tree

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

For each node $v$ 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 node $v$ both $in(v)$ and $out(v)$. For a leaf, we can trivially define $in(v) = 1$ and $out(v) = 0$. Now consider a node $v$ 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 minimum sized 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 non-binary trees, and also for weighted variants of vertex cover (where each node has a cost for including it in the vertex cover).

# 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 of size $k = 2$, then there is an easy polynomial time algorithm: simply test every pair of nodes (and every individual node) and check whether that node is a valid vertex cover. Assuming that you can check the vertex cover of a graph $G = (V, E)$ with $n$ nodes and $m$ edges in time $O(m)$, then this algorithm runs in time $O(n^2 m)$.

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 rapdily becomes unusable, even for small values of $k$. For example, if $k = 10$ and $n = 1000$ and you use a CPU that can perform 12 billion instructions per second (e.g., an Intel Core i7), then it would take more than one trillion 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 node $z \in V$, 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:

**Algorithm:** ParameterizedVertexCover($G = (V, E)$, $k$)
**Procedure:**
    `/* Base case of recursion: */`
    **if** $k = 0$ **and** $E \neq \emptyset$ **then return** $\perp$
    **if** $E = \emptyset$ **then return** $\emptyset$
    `/* Recurse on arbitrary edge e: */`
    Let $e = (u, v)$ be any edge in $G$.
    $S_u =$ `ParameterizedVertexCover`$(G_{-u}, k - 1)$
    $S_v =$ `ParameterizedVertexCover`$(G_{-v}, k - 1)$
    **if** $S_u \neq \perp$ **and** $|S_u| < |S_v|$ **then**
        **return** $S_u \cup \{u\}$
    **else if** $S_v \neq \perp$ **and** $|S_v| < |S_u|$ **then**
        **return** $S_v \cup \{v\}$
    **else return** $\perp$

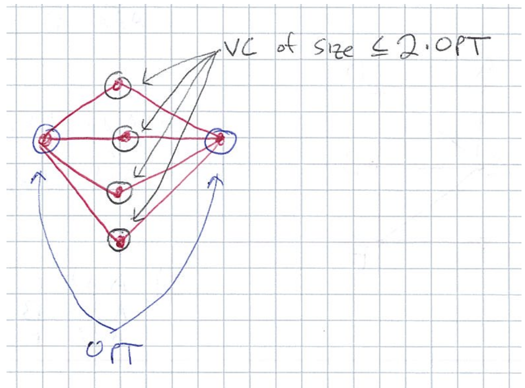


**Figure 2:** Example of a graph with an optimal vertex cover of size 2. The black circled nodes in the middle also form a vertex cover of size 4. This black vertex cover is not optimal, but is a 2-approximation.

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 node from the graph and add it to the vertex cover, and we recurse twice. As a base case, if we have removed $k$ nodes 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.

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:

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

(where I am assuming it takes approximately $O(m)$ time to delete a node and its adjacent edges from a graph, and I am ignoring 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 ($k = 10$, $n = 1000$), this improved algorithm would run in about 1ms!

# 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 vertex cover. For a given graph $G$, define $OPT(G)$ to be a minimal-sized vertex cover[4].

**Definition 7** *We say that algorithm $A$ is an **$\alpha$-approximation algorithm** for vertex cover if, for every graph $G$:*

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

That is, an algorithm $A$ is a good aproximation algorithm for vertex cover if it guarantees that it will find a vertex cover that is not too much larger than the best possible vertex cover.

[4]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 fix one.

In the context of approximation algorithms, vertex cover is a particularly interesting problem to study. It is very easy to design a 2-approximation algorithm (and we will see two such algorithms today). However, the best known approximation for vertex cover is only marginally better, achieving an approximation factor of $2 - O(1/\sqrt{\log n} = 2 - o(1)$. In fact, we know that if $P \neq NP$, then there in no polynomial time approximation algorithm that achieves an approximation factor of better than 1.3606. (This follows from the famous PCP theorem.) And if you assume that the Unique Games Conjecture[5] is true, then you cannot achieve an approximation factor of $2 - \epsilon$ for any $\epsilon > 0$. In the latter case, the simple 2-approximation algorithms that we will see today are, effectively, the best you can do!

## 5.1 Randomized Approximation Algorithm

We first 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 ← ∅
    /* Repeat until every edge is covered:                                  */
    while E ≠ ∅ do
        Let e = (u, v) be any edge in G.
        b ← 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 ← C ∪ {z}
        G ← 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 node 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 node to $C$, we have a $1/2$ 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:

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

[5] This is a conjecture that has widespread implications in complexity theory.

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

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

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

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

Finally, we conclude that calculation:

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

This yields the following theorem:

**Theorem 8** *The algorithm* `RandomizedVertexCover` *yields a 2-approximation for vertex cover (in expectation).*

## 5.2 Deterministic Approximation Algorithm

Next, we consider a deterministic algorithm for approximating vertex cover. There are three natural candidate algorithms (see below). Each of these algorithms greedily adds nodes to the cover, with progressing attempts at cleverness. The first algorithm simply adds arbitrary nodes 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 node that covers the most edges that remain uncovered. Stop for a minute and think about which algorithm you think is best.

---

`/* This algorithm adds nodes 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:**

  $C \leftarrow \emptyset$

  `/* Repeat until every edge is covered: */`

  **while** $E \neq \emptyset$ **do**

    Let $e = (u, v)$ be any edge in $G$.

    $C \leftarrow C \cup \{u\}$

    $G \leftarrow G_{-u}$ `//` Remove $u$ and all adjacent edges from $G$.

  **return** $C$

---

`/* This algorithm adds nodes 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:**

  $C \leftarrow \emptyset$

  `/* Repeat until every edge is covered: */`

  **while** $E \neq \emptyset$ **do**

    Let $e = (u, v)$ be any edge in $G$.

    $C \leftarrow C \cup \{u, v\}$

    $G \leftarrow G_{-\{u,v\}}$ `//` Remove $u$ and $v$ and all adjacent edges from $G$.

  **return** $C$

---

`/* This algorithm adds nodes greedily, one at a time, until everything is covered. At each step, the algorithm chooses the next node that will cover the most uncovered edges. */`

**Algorithm:** ApproxVertexCover-3($G = (V, E)$)

**Procedure:**

  $C \leftarrow \emptyset$

  `/* Repeat until every edge is covered: */`

  **while** $E \neq \emptyset$ **do**

    Let $d(x) =$ number of uncovered edges adjacent to $x$.

    Let $u = \textbf{argmax}_{x \in V} d(x)$

    $C \leftarrow C \cup \{u\}$

    $G \leftarrow G_{-\{u\}}$ `//` Remove $u$ and all adjacent edges from $G$.

  **return** $C$

---

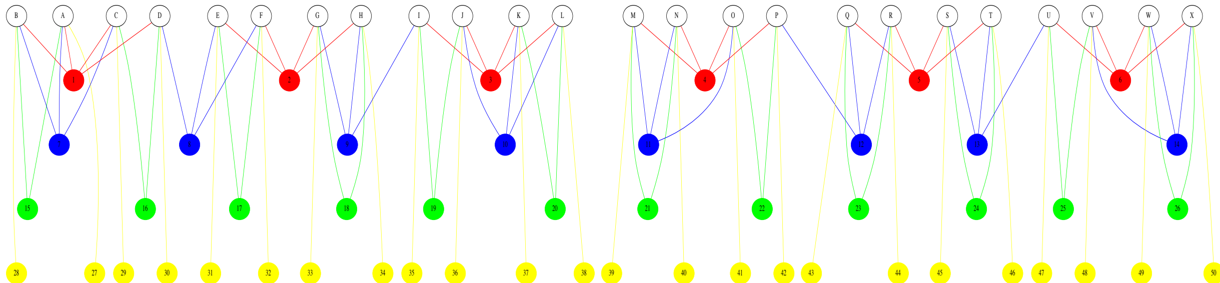

**Figure 3:** Counterexample for the greedy vertex cover algorithm. In this example, $n = 4$. The top row consists of $n! = 24$ white nodes, each of degree $n = 4$. The next row consists of $n!/n = 6$ red nodes, each of degree $n = 4$. The next row consists of $n!/(n-1) = 8$ blue nodes of degree 3. The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

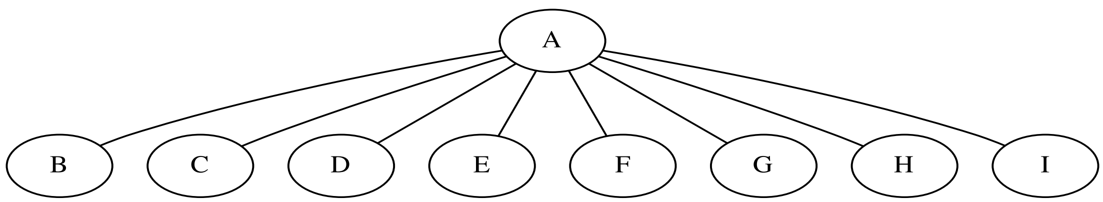


**Figure 4:** Counterexample for the simple vertex cover algorithm where an arbitrary node is added in each step. When each edge is considered, the leaf node is added, yielding a vertex cover of size $n-1$. The optimal vertex cover contains only node $A$.

It turns out that neither Algorithm 1 nor Algorithm 3 achieve a good approximation ratio. (As an exercise, draw a graph for which each of these algorithms performs 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 nodes being added is a "correct" node, 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 trick to 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:

**Definition 9** *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 nodes: each node 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 node in the graph is adjacent to an edge in the matching, then it is a ***perfect matching***.

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 10** *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 node 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 11** *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 nodes $u$ 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 12** `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 $|C| = 2|E'|$, since for every edge in $E'$ we added two nodes to $C$. Since $E'$ is a matching, we know that $|OPT(G)| \geq |E'|$, and hence we conclude:

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

□

There are a couple interesting facts to take away from this proof. First, I want to emphasize the idea of trying to bound the optimal solution, and using this bound to show a good approximation. Second, notice the close relationship between finding a vertex cover and finding a matching. It turns out that vertex covers and matchings are related in a special way: they are, in a sense, *dual* problems (*if considered in a fractional sense*). We will come back to this idea of duality later in the semester.