15-859FF: Coping with Intractability **CMU, Fall 2019**
Lecture #5: Iterative Compression September 18, 2019
Lecturer: Anupam Gupta Scribe: Amulya Musipatla

1 Iterative Compression

In previous lectures, we explored the uses of bounded depth search trees, kernelization, and randomization in approaches to problems like vertex cover, feedback vertex set, and longest path. Today, we look at how we can use iterative compression to obtain FPT algorithms.

Iterative compression was introduced by Reed, Smith, and Vetta [**?**] when approaching odd cycle transversals, and the idea has been adapted to other problems since. The main idea is to use a compression routine, that takes a solution of some size and uses it to find a solution of a smaller size. We'll see how these compression algorithms can be used to approach generally hard problems; in particular, we will apply this to vertex cover and feedback vertex set.

2 Vertex Cover

Recall the vertex cover problem: given a graph $G = (V, E)$ with n vertices and a parameter k, find a vertex cover of G of size less than or equal to k. We previously discussed how by using a greedy maximal matching algorithm on a graph, we can obtain a vertex cover with at most $2k$ vertices. We want to try to use this given solution to construct a vertex cover of size at most k.

2.1 Method 1

Suppose, for some graph $G = (V, E)$, we are given a vertex cover Z of size $2k$. We first try use Z to directly find a vertex cover of size k.

Definition 5.1. The *neighborhood* of a set of vertices X in a graph $G = (V, E)$, denoted $N(X)$, is defined as the set of all vertices adjacent to any vertex in X. That is,

$$N(X) = \{v \in V : \exists u \in X, (u, v) \in E\}$$

.

Suppose X^* is the optimal vertex cover for G. We note that X^* may or may not contain vertices from Z. For example, Figure 5.1 shows a case in which we can have a vertex cover Z that doesn't intersect with the optimal vertex cover X^* at all.

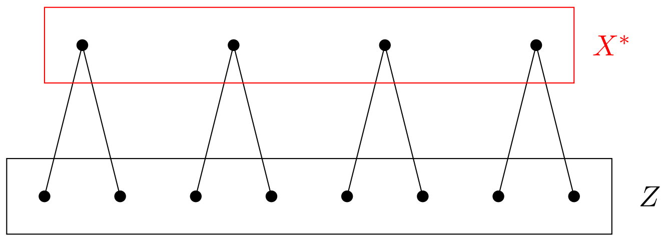

Figure 5.1: Example graph with $Z \cap X^* = \emptyset$

Given this fact, we can think of the graph G as having the structure shown in Figure 5.2. Looking at this figure, we see that all edges in G should exist within Z or between Z and $V \setminus Z$ in order for Z to be a valid VC. Assuming $Z_1 = Z \cap X^*$, only edges hitting Z_2 are not covered by Z_1. Since Z_2 and X^* are disjoint, these are covered in $N(Z_2)$. Thus, $X^* = Z_1 \cup N(Z_2)$.

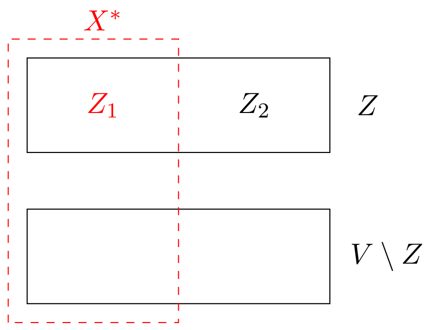

Figure 5.2: Generalization of how a graph G looks, given vertex covers Z and X^*

The algorithm is then as follows:

Algorithm 1 (CompressVC): On input (G, k, Z) :

```
1. Guess $Z \cap X^*$
2. Check if $|(Z \cap X^*) \cup N(Z \setminus X^*)| \leq k$
3. If so, output this set. Otherwise, go back to step 1
```

Claim 5.2. *Algorithm 1 takes time $O(4^k m)$ to find a vertex cover of size at most k.*

Proof. When guessing $Z \cap X^*$, we only need to guess subsets of Z of up to size k. There are $\sum_{i \leq k} \binom{|Z|}{i}$ different subsets, and it takes $O(m)$ time to find the neighborhood of $Z \setminus X^*$ and check the size of the VC. This results in an algorithm that takes $2^{2k} m$ time, or $4^k m$ time. ☐

We note that this algorithm takes a much longer time than previous algorithms we considered. However, we see that if we were simply to go from a nonoptimal vertex cover of size $k+1$ to one of size k, this compression algorithm would only take $O(2^k m)$ time. We use this idea to motivate our next approach.

2.2 Method 2

With this method, we want to use the fact that going from a solution of size $k+1$ to one of k doesn't take too much time.

The basic idea is as follows: if we want to solve VC on an instance $(G = (V, E), k)$, we can pick any vertex $v \in V$ and remove it. We know that, if $VC(G)$ denotes the size of a vertex cover of G, then $VC(G - \{v\}) \leq VC(G) \leq VC(G - \{v\}) + 1$. This is true since we can add v to any vertex cover of $VC(G - \{v\})$ and obtain a vertex cover. We call this property *monotonicity*, and say that the vertex cover is *monotone*.

Our algorithm is given below:

Algorithm 2 (IterateVC): On input (G, k)

```
1. Pick any vertex $v$ from $G$, and remove it from $G$ to obtain a graph $H$.
2. Let $Z$ be the vertex cover returned by IterateVC($H, k$)
3. Return CompressVC($G, k, Z \cup \{v\}$)
```

Theorem 5.3. *Let $T(n, k)$ be the runtime of IterateVC on an input (G, k) where G has n vertices and m edges. Then, $T(n, k) \leq O(2^k nm)$*

Proof. (by induction on n)

Base Case: When G has k vertices, we can simply return these vertices.

Inductive Hypothesis: Assume that $T(n-1, k) \leq 2^k(n-1)m$.

Inductive Step: Suppose G has n vertices. IterateVC calls itself after removing a vertex v, and then runs the CompressVC algorithm after adding v to the vertex cover. Thus,

$$\begin{aligned} T(n,k) &\leq T(n-1,k) + 2^k m \\ &\leq 2^k(n-1)m + 2^k m && \text{[by the induction hypothesis]} \\ &\leq 2^k nm \end{aligned}$$

□

Observation 5.4. We note that the above algorithm and runtime analysis only relies on the fact that vertex cover is monotone under vertex deletion. This gives rise to our next theorem.

Theorem 5.5. *If there exists an algorithm for compression from $k+1$ to k for a monotone problem taking time $f(k)n^c$, then there exists an algorithm to solve any instance taking $f(k)n^{c+1}$ time.*

The above methods illustrate two important ideas for problems that are monotone under vertex deletion:

1. Find a recursive solution, add a vertex back, and then compress this solution.
2. Guess the given solution's intersection with the optimal solution. This gives a structure that is easier to work with.

These points provide motivation for us to work with disjoint versions of problems and focus on compression by 1.

2.3 Disjoint Vertex Cover

The disjoint vertex cover problem is as follows: given an input (G, k, Y), find a vertex cover of G with at most k vertices that is disjoint from Y.

Claim 5.6. *If there is an algorithm for DisjointVC that takes time $a^k n^c$, then there is a compression algorithm for VC that takes time $(a+1)^k n^c$.*

The algorithm would work as follows:

Algorithm 3: (CompressVC2) On input (G, k, Z):

1. Guess $Z \cap OPT$
2. use DisjointVC$(G \setminus (Z \cap OPT), k - |Z \cap OPT|, Z \setminus (Z \cap OPT))$ to find the rest of OPT.

Exercise: Show that the Claim 5.6 holds true.

3 Disjoint Feedback Vertex Set (FVS)

We now focus on the disjoint feedback vertex set problem: given an input (G, k, Z), where Z is a feedback vertex set for G of size $k+1$, we want to find a set X of size $\leq k$ that hits all cycles in G (i.e, $G \setminus X$ is a forest), such that X and Z are disjoint.

Theorem 5.7. *If there is an algorithm with runtime $4^k n^c$ for DisjointFVS, there is an algorithm with runtime $5^k n^{c+1}$ for FVS.*

Note: In any YES instance of DisjointFVS, we know that $G[Z]$ and $G[V \setminus Z]$ must be forests. Since Z is a FVS, we know that by definition, $G[V \setminus Z]$ must be acyclic. In order to be able to find another FVS completely disjoint from Z, $G[Z]$ must be acyclic. Otherwise, we would need to include a vertex from Z in any FVS.

We can reduce any instance (G, k, Z) by applying the following rules:

1. If $G[Z]$ induces a cycle, say NO
2. If there is a vertex v of degree ≤ 1, delete it: $(G, k) \rightarrow (G - \{v\}, k)$

   We can do this because we know that it can't be in any cycles, so there is no need to pick it for an FVS.

3. If there is a vertex $v \in V \setminus Z$ such that $G[Z \cup \{v\}]$ contains a cycle, add v to the FVS: $(G, k) \rightarrow (G - \{v\}, k-1)$

   We know that in this case, v must necessarily be in any FVS disjoint from Z because otherwise we would have a cycle.

4. If there exists a vertex v of degree 2 that has at least one neighbor in $V \setminus Z$, delete it and add a new edge between the neighbors a and b: $(G, k) \rightarrow (G - \{v\} + \{a, b\}, k)$

   Figure 5.3 illustrates the two cases in which this rule applies. We see that in either case, we don't need to keep v as long as we connect its neighbors, because any of its neighbors in $V \setminus Z$ could take the place of v in an FVS.

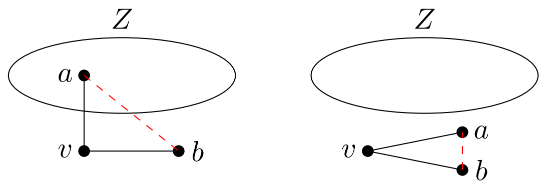

Figure 5.3: The two cases in which rule 4 would apply.

These rules give rise to the following algorithm:

Algorithm 4: On input (G, k, Z):

1. If none of the rules 1-4 apply, pick a vertex v in $V \setminus Z$ of degree ≤ 1 in $G[V \setminus Z]$
2. Branch on v: recurse on $(G, k, Z + \{v\})$ and $(G - \{v\}, k - 1, Z)$

Claim 5.8. *We know that there exists a vertex v whose degree in $G[V \setminus Z]$ is ≤ 1 since $G[V \setminus Z]$ is a forest, and none of our rules change this property when applied. We also know that the following claims hold:*

1. *v has $\geq$ 1 neighbor in Z* *[by rule 2]*
2. *v has $\geq$ 2 neighbors in Z* *[by rule 4]*
3. *All of v's neighbors in Z are in different components* *[by rule 3]*

One potential worry is that the branch in which we don't reduce k will end up taking a lot of time. However, by the third point in Claim 5.8, we see that whenever we add v to Z, we reduce the number of connected components in Z. Therefore, if we track the expression $k + \#\text{components}$ over the recursion, this expression decreases by 1 on each recursion branch. Moreover, its value is at most $2k$ in the root branch. So there are at most 2^{2k} leaves in the recursion tree, and we obtain the following:

Lemma 5.9. *The runtime of Algorithm 4 is $O(2^{k+\#\text{components}(Z)}m) \leq O(2^{2k}m) = O(4^k m)$.*

Recall that the base exponent increases by 1 going from the disjoint problem to the original problem (Theorem 5.7), and there is an additional factor of n in the running time. So the final running time is $O(5^k mn)$.