CS787: Advanced Algorithms	**Scribe:** Jeremy McMahan
Lecture 2: Fixed-Parameter Tractability	**Date:** January 24, 2019

2.1 Fixed-Parameter Tractability

Can we quantify the hardness of an instance of a problem? Yes, using parameters. A parameter assigns a numerical (integral) quantity to every instance of a problem.

A problem is called *fixed-parameter tractable*, or "FPT", if there is an exact algorithm for the problem that given an instance of size n with a parameter k runs in time $O(f(k)poly(n))$ for some computable function f.

VertexCover (VC) has a brute force algorithm that runs in time $O(n^k)$ by enumerating all subsets of V of size k and checking if any subset is a vertex cover. A running time of form $O(2^k + poly(n))$ is much beter than $O(n^k)$ since k can be $O(\log n)$ for the entire algorithm to run in polynomial time whereas k needs to be a constant for the brute force algorithm to run in polynomial time.

2.2 Bounded Search Trees

Let us consider the decision version of the VertexCover problem. Given a graph G and a positive integer k, we want to determine whether there exists a vertex set $S \subseteq V$ satisfying

1. $\forall e \in E$, S contains at least one endpoint of e, and
2. $|S| \leq k$.

Let's consider a recursive algorithm for VertexCover, Algorithm 1.

We know that for any edge one of its endpoints must be in the vertex cover by definition. Hence, the algorithm is correct.

Now, we analyze the running time of this algorithm. We can write a recurrence equation for the running time in terms of n and k as

$$T(n, k) = n + 2T(n - 1, k - 1).$$

We can reduce this to a single variable recurrence depending only on k:

$$\tilde{T}(k) \leq n + 2\tilde{T}(k - 1),$$

which solves to $\tilde{T}(k) = O(n2^k)$.

This is a *bounded search tree algorithm*, which gives a running time of $O(f(k)poly(n))$.[1] The general strategy for designing a bounded search tree algorithm is to ensure the degree of any node in the

[1] It is also called *branching algorithm* in some literature.

Algorithm 1 Bounded Search Tree Algorithm for VertexCover

```
VC(G, k) :
  if E = ∅ then
      return ∅
  end if
  pick any edge e = (u, v)
  T ← VC(G − u, k − 1)
  if Recursive solve was successful then
      return T ∪ {u}
  else
      T ← VC(G − v, k − 1)
      if Recursive solve was successful then
          return T ∪ {v}
      else
          return "unsuccessful"
      end if
  end if
```

recursion tree is bounded by a constant, the height of the tree is some function of k, and it takes polynomial time to process each node in the search tree. In this case, we potentially recurse on both u and v, so the degree of each node in the recursion tree is at most 2. Also, since we decrease k by one each time we recurse, we know that the height of the tree is at most k. Choosing an edge from a graph and producing the smaller instances for branching also takes polynomial time.

Note that reductions do not necessarily preserve fixed-parameter tractability. For example, we know VertexCover reduces to IndependentSet, but the small parameter value k becomes the large parameter value $n - k$ in the traditional reduction. Further,

Theorem 2.2.1 *There is no $n^{o(k)}$ time algorithm for* IndependentSet*, where k is the target size, unless ETH fails.* [2]

2.3 Kernelization

In fact, FTP can be defined as the problem having a $O(g(k) + poly(n))$ time algorithm instead, using the notion of *kernelization*. A kernelization algorithm takes as input an instance of size n and parameter k and in polynomial time:

- solves the instance, or
- reduces the instance to an instance of size $h(k)$, called a *kernel*.

A kernelization algorithm natually implies an $O(g(k) + poly(n))$ time algorithm for the problem: run the kernelization algorithm on the original instance in time $O(poly(n))$, and then run an

[2] The *exponential time hypothesis (ETH)* essentially conjectures that there is no subexponential time ($2^{o(n)}$) algorithm for SAT.

appropriate brute force algorithm on the kernel (of size $h(k)$) using $g(k)$ time, yielding an overall $O(g(k) + poly(n))$ running time. This shows that if a problem admits a kernelization algorithm, then the problem is FPT. Its converse is also true by the following theorem.

Theorem 2.3.1 *If a problem is FPT, then it admits a kernel of size $h(k)$.*

Proof: Given an algorithm that runs in time $O(f(k)poly(n))$, we note the following:

1. If $n > f(k)$, then this algorithm runs in time $O(poly(n))$
2. If $n \leq f(k)$, then the instance itself is a kernel

Further, we see there is an $O(2^{f(k)} + poly(n))$ algorithm for the problem. ■

2.4 Kernelization algorithms for VertexCover

In this section we discuss some kernelization algorithms for the VertexCover problem.

2.4.1 An $O(k^2)$ kernel

Let us derive a kernelization algorithm for the decision version of VertexCover.

Reduction rule 2.4.1 *Let (G, k) be an instance for the* VertexCover *problem. If there exists an vertex v with degree less than or equal to 1, remove v from G. The new instance is $(G - v, k)$.*

Let S be a vertex cover of a graph G. If v is isolated, then $S \setminus \{v\}$ is also a vertex cover of G. If v has degree 1, let its neighbor be u, and $S \setminus \{v\} \cup \{u\}$ would also be a vertex cover of G. Hence Reduction 2.4.1 is sound.

Reduction rule 2.4.2 *Let (G, k) be an instance for the* VertexCover *problem. If there exists an vertex v with degree larger than k, remove v from G and decrease k by 1. The new instance is $(G - v, k - 1)$.*

Since we are computing the vertex cover of size at most k, if v is not in the vertex cover, then all its neighbor shall be included, and the size of the vertex cover would be at least $k+1$, a contradiction. Hence v must be included in the vertex cover.

We apply Reduction 2.4.1 and 2.4.2 on the input instance (G, k) exhaustively. If in any instance k is less than zero, the algorithm would abort and return NO. Otherwise, we obtain a smaller instance (G', k'), where $k' \leq k$, and each vertex in G' has degree between 2 and k.

If a vertex cover of G' contains at most k vertices, since each vertex covers at most k edges, G' would contain at most k^2 edges and $k^2 + k$ vertices. Hence if G' has more than $k^2 + k$ vertices or more than k^2 edges, the algorithm could abort and return NO immediately. Otherwise, G' would be a kernel containinig at most $k^2 + k$ vertices.

Reduction 2.4.1 and 2.4.2 can both be implemented in $O(n + m)$ time, and they together can be applied at most n times. Hence the kernelization algorithm runs in $O(mn + n^2)$ time.

Algorithm 2 A kernelization Algorithm for VertexCover

$VC(G, k)$:

Apply Reduction 2.4.1 and 2.4.2 exhaustively, and abort whenver $k < 0$.
if G has more than k^2 edges **then**
  abort
else
  return G
end if

2.4.2 An $O(k)$ kernel

To derive a linear kernel for the VertexCover problem, we need the technique of *crown decomposition*.

Given a graph $G = (V, E)$, the tuple (C, H, R) is a crown decomposition of a graph G if

1. $C \cup H \cup R = V$;
2. C is an independent set;
3. no edge $(u, v) \in E$, satisfies $u \in C$ and $v \in R$;
4. there is a matching, M, between C and H with $|M| = |H|$.

We note that the smallest vertex cover for $C \cup H$ is H since $|H| = |M|$ by 4 and by 3 $|C| \geq |H|$, so M is a maximal matching.

Theorem 2.4.3 *Let G be a graph containing at least $3k + 1$ vertices. There exists a polynomial time algorithm that computes*

- *a crown decomposition of G, or*
- *a matching of size at least $k + 1$.*

We first note that this theorem yields a kernelization for VertexCover. If G has fewer than $3k$ vertices, then it is a kernel of linear size. Otherwise, if there exists a matching of size at least $k+1$, then there is no vertex cover of size at most k in G. Lastly, given a crown decomposition, we can add H to the vertex cover, and recursively solve VertexCover problem on R.

We now present a polynomial time algorithm for finding the crown decomposition.

1. Find a maximal matching M in G
2. If $|M| \geq k + 1$, then return M
3. $X \leftarrow \{u, v | \{u, v\} \in M\}$ and $I \leftarrow V \setminus X$

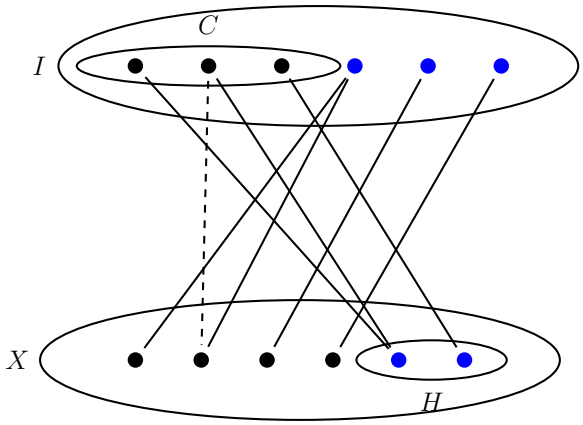

Figure 2.4.1: An illustration of the algorithm for crown decomposition. X comprises the endpoints of all edges in the maximal matching, and I is an independent set. The blue vertices denote S, a vertex cover of the bipartite graph G' with $V(G') = X \cup I$ and $E(G') = (I \times X) \cap E(G)$. The set C comprises the black vertices in I, and the set H comprises the blue vertices in X. The rest of the vertices form R. An edge connecting C and $X \setminus H$ (demonstrated by the dashed line) would imply that S is not a vertex cover, a contradiction.

4. Consider the bipartite graph between I and X. Find the maximum matching M' and minimum Vertex Cover S in the bipartite graph.

5. $C \leftarrow I \setminus S, H \leftarrow S \cap X$, and $R \leftarrow (I \setminus C) \cup (X \setminus S)$ is a crown decomposition for G.

If $|M| \geq k+1$, then we are done at step 1. Now we assume that $|M| \leq k$. Hence $|X| \leq 2k$ as each edge of M contributes two vertices to X. Note that I is an independent set, since otherwise there exists an edge in I that could be added to M, contradicting that M is a maximal matching.

If $|M'| \geq k+1$, then we again find a matching of size at least $k+1$. Hence we assume $|S| = |M'| \leq k$. Since $|V| = |I|+|X| \geq 3k+1$ and $|X| \leq 2k$, we have $|I| \geq k+1$, and hence $C = I \setminus S \neq \emptyset$. Therefore, there exists a matching between $C = I \setminus S$ and $H = S \cap X$. Since I is an independent set, there is no edge between C and $I \setminus C$. Suppose there is an edge $\{u, v\}$ with $u \in C$ and $v \in X \setminus S$, then by construction both $u, v \notin S$, and hence S is not a vertex cover of the bipartite graph formed by I and X, a contradiction. Hence there exists no edge connecting vertices in C and R, and (C, H, R) is indeed a crown decomposition. An illustration is available in Figure 2.4.1.