Lecturer: Jason Li Scribe: David Zeng, George Lu, Mounira Tlili

1 Background

Many problems of both theoretical and practical interest are **NP**-hard. As a result, we do not know of any efficient (in **P**) algorithms to solve this problem. It is possible efficient algorithms do not even exist. However, there are several approaches to restrict or relax the domain / requirements of the problem so that they can be solved 'efficiently'. One possible approach is to study approximation algorithms, where the goal is to find solutions which are not too far from the optimal. This lecture covers a different approach, which is fixed parameter tractable (FPT) algorithms.

1.1 Definition

A problem is fixed parameter tractable if it can be solved by algorithm with runtime $f(k) \cdot poly(n)$, where n is the input size and k is some parameter of the input or solution, defined specific to that problem is. Note that there is no restriction on the asymptotic complexity of $f(k)$. The existence of an efficient algorithm for the parameterized problem demonstrates that if we limit the value of k for the inputs, we can efficiently solve the problem. This also gives us a better sense of what kinds of instances of an **NP**-hard problem are hard.

2 Problems

We provide several examples of fixed parameter tractable problems and corresponding algorithms to solve them.

- Vertex Cover parameterized by OPT (k in the decision version)
  - $2^k \cdot \text{poly}(n)$
  - $poly(n) + 2^k \cdot \text{poly}(k)$
- k-paths parameterized by k
  - $k! \cdot \text{poly}(n)$
  - $c^k \cdot \text{poly}(n)$
- Feedback vertex set parameterized by k
  - $6^k \cdot \text{poly}(n)$

2.1 Vertex Cover

Recall the definition of the decision version of the Vertex Cover Problem. We are given a graph $G = (V, E)$ and a value k and our task is to determine if $\exists V' \subseteq V$ such that $|V'| \leq k$ and $\forall (u, v) \in E$, we have $u \in V' \vee v \in V'$. This problem is **NP**-complete. We parameterize vertex cover with the built-in parameter k.

2.1.1 Randomized Selection

We make a simple observation - that since every edge has to be incident with a vertex in the vertex cover if it exists, then for every edge, picking a random endpoint has at least a probability $\frac{1}{2}$ chance of picking a vertex cover. This leads to algorithm 1.

Algorithm 1 Randomized k-Vertex Cover

```
1:  procedure VertexCover1Run(V,E,k)
2:      j ← 0
3:      for j < k do
4:          e' ← rand(e ∈ E)
5:          v' ← rand(v ∈ e)
6:          E ← E − {e ∈ E | v ∈ e}
7:          j ← j + 1
8:      end for
9:      return E ≡ ∅
10: end procedure
11:
12: procedure VertexCover1(V,E,k)
13:     j ← 2^k log(n)
14:     for j > 0 do
15:         if VertexCoverRun1(V, E, k) then
16:             return true
17:         end if
18:         j ← j − 1
19:     end for
20:     return false
21: end procedure
```

We observe that if a vertex cover of size k exists, each call to VertexCoverRun1() succeeds with probability at least $\frac{1}{2^k}$, as each time, it picks an uncovered edge, meaning that it has a $\geq \frac{1}{2}$ chance of picking a vertex in the cover. If $E \equiv \emptyset$ at the end of the loop, then we can infer that every edge must have been incident to such a vertex, and so we have found a size k vertex cover. Thus, after $O(2^k \log(n))$ iterations, we know we will succeed with high probability (i.e $1 - \frac{1}{n^2}$). Since in each run, we potentially have to iterate over every edge, this means that the runtime will be $2^k \operatorname{poly}(n)$

2.1.2 Derandomization

We can simultaneously reduce the runtime and derandomize the above algorithm using branching. If there does exist a k vertex cover, when we pick an edge, at least one of the vertices will be in the vertex cover. So we can simply recurse on both options, guaranteeing that if there is a k vertex cover, it will be picked in one of our branches. We will present the algorithm 2.

Algorithm 2 Deterministic k-Vertex Cover

```
1: procedure VertexCover2(V,E,k)
2:     if k = 0 then
3:         if E ≡ ∅ then return true
4:         else return false
5:         end if
6:     end if
7:     (u, v) ← rand(e ∈ E)                ▷ We use rand here, but any arbitrary e ∈ E works
8:     E_u ← E \ {e ∈ E | u ∈ e}
9:     E_v ← E \ {e ∈ E | v ∈ e}
10:    return VertexCover2(V, E_u, k − 1) ∨ VertexCover2(V, E_v, k − 1)
11: end procedure
```

The runtime of this modified version can be understood as a recurrence relation, expressed as

$$T(0) = n$$

$$T(k) = 1 + 2T(k-1)$$

We push the counting of edge deletions to the leaf level, but this is an upper bound, as any edge deleted in a non-leaf node will not be deleted in any of its children. This solves to a runtime of $O(2^k\, n)$

2.1.3 Kernelization

We can further improve the runtime with kernelization - i.e. replacing the problem instance with a smaller one. In this case, we make 2 simple observations

- If a vertex v has degree $> k$, it must be in every vertex cover of size $\leq k$
- If a vertex v has degree 0, if there is a vertex cover of size $\leq k$ which contains v, then there must be a vertex cover of size $\leq k$ which does not contain v.

To prove first property, consider a vertex cover which does not contain v. Observe that it still must cover all edges incident to v, which means it must contain $N(v)$. But $|N(v)| > k$, so such a vertex cover will have cardinality $> k$. For the latter property, observe that for any vertex cover which uses a degree 0 vertex, the vertex can be safely removed from the cover. This leads to the following reduction rules.

- If $v \in G : deg(v) > k$, then VertexCover$(G, k) \equiv$ VertexCover$(G \setminus v, k-1)$.
- If $v \in G : deg(v) = 0$, then VertexCover$(G, k) \equiv$ VertexCover$(G \setminus v, k)$

In this context, $G \setminus v$, refers to the graph created by deleting v from the set of vertices and removing any edges incident to v.

This gives the following improvement on Algorithm 2.

Claim 12.1. *If neither of the above reduction rules apply, G has a k-vertex cover only if $n \leq k^2 + k$.*

Algorithm 3 Kernelized k-Vertex Cover

```
1: procedure KernelizeVertexCover(V,E,k)
2:     while ∃v ∈ V : |{e ∈ E | v ∈ e}| = 0 ∨ |{e ∈ E | v ∈ e}| > k do
3:         if |{e ∈ E : v ∈ e}| > k then
4:             k ← k − 1
5:         end if
6:         V ← V \ {v}
7:         E ← {e ∈ E | v ∉ e}
8:     end while
9:     if |V| ≥ k^2 + k then
10:        return false
11:    end if
12:    return VertexCover2(V, E, k)
13: end procedure
```

Proof. Suppose neither reduction rule applies. Then we know that the degree of all vertices in vertex cover V_C must be in the range $[1, k]$. Thus, we can bound the degree of every vertex in V_C with k. This means that $N_G(V_C) \leq k^2$. But we know that $N_G(V_C) \cup V_C$ must be the set of vertices in G, as all vertices have at least one incident edge, and by definition, every edge is adjacent to a vertex in V_C. So we can take the sum to bound $|V|$ by $k^2 + k$. □

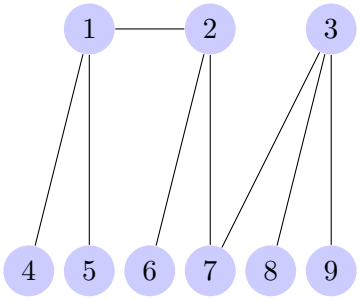

Figure 12.1: The first level contains vertices in the vertex cover. All other vertices in G must be neighbors to some v in the optimal vertex cover (second level)

The application of each reduction rule takes $O(n)$ time and is done $O(n)$ times, so the kernelization step takes poly(n). In addition, since the kernelized instance has size at most $O(k^2)$, we can use our previous algorithm to solve it in $O(2^k\, k^2)$, bounding the total runtime by $O(n^2 + 2^k\, k^2)$

2.2 K-Paths

We can define the k path problem as follows: Given an input consisting of a graph G and a length k, $\langle G, k\rangle$ is as YES instance if and only if there exists a simple path with k vertices in G. This problem is also **NP**-complete, but can be parameterized with k to yield efficient fixed parameter algorithms.

2.2.1 Naive Solution

A naive solution for k-paths is to try all possible subsets of k vertices. However, this is not an FPT algorithm, since the number of possible combinations of edges is $\Omega(n^k)$, which is not in

$O(f(k)\text{poly}(n))$ for any $f(k)$.

2.2.2 Dynamic Programming and Randomization

Once again, we will try to use the power of randomization to 'pick' a correct solution with some probability $f(k)$ and then verify or disprove each such random assignment in $poly(n)$, just like in our randomized FPT algorithm for vertex cover. In this case, we will assign the vertices of G a permutation of the numbers $[|V|]$.

Claim 12.2. *If there is a length k path, there is at least a $\frac{1}{k!}$ probability that the vertices along the path are assigned values in increasing order.*

Proof. Consider any random assignment of numbers to V. If we consider the vertices on the path, there are $k!$ possible permutations for their relative orderings. Since all are equally likely, this gives a minimum $\frac{1}{k!}$ probability of success. □

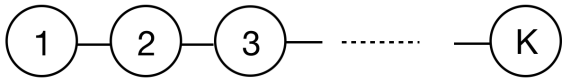

Figure 12.2: One possible labeling for a success case

So we only need $\approx k!$ attempts to find such an assignment if it exists.

Note that if we have such an assignment, we can find the path of length k efficiently using dynamic programming. Let v_i be the vertex with label i. We can write the following recurrence

$$L(v_i) = 1 + \max\{\{L(v_j) \mid (v_j, v_i) \in E \wedge j < i\} \cup \{0\}\}$$

L computes the longest increasing path ending on v_i ending on v_i. If the random assignment successfully labels a path of length k, we can simply check that $\max\{L(v) \mid v \in G\} \geq k$ to verify a length k path. This DP takes linear time with respect to n, as there are a linear number of subproblems, and we only need to consider every edge in E in one maximization. Taken together, this gives complexity $\tilde{O}(k! \cdot n)$.

We can see this is correct by a simple induction argument that for any vertex v_i in the path, $L(v_i)$ must be at least the length of the prefix of the path ending at $L(v_i)$, as we know the assignment on the path is increasing.

2.2.3 Boosting Probability of Success

We notice in the above algorithm, we have an exceptionally efficient (linear time) method of verifying any random assignment produces a k path, but as a sort of tradeoff, the probability of success in actually randomly stumbling upon such an assignment is also quite low. The high level idea here is to improve overall runtime by increasing the probability of success of the random assignments by enforcing less stringent constraints on them, possibly at the cost of greater complexity in checking if any assignment is successful.

To do this, this time we assign every vertex with a value uniformly and independently randomly from $[k]$, and we define a 'successful' assignment on an instance with a k path to be one where every vertex on the k path is assigned a different number.

Claim 12.3. *The expected number of trials to find a successful assignment if one exists (i.e. there is a k-path) is $O(e^k)$.*

Proof. If we consider the assignment on the path, there are k^k possible assignments, of which $k!$ are permutations of $[k]$, meaning the values are unique. Thus, the probability of success on a single run is

$$\frac{k!}{k^k}$$

. Using Stirling's Approximation, we can lower bound this by

$$\frac{k!}{k^k} \approx \frac{k^k \cdot e^{-k}\sqrt{2\pi k}}{k^k} = \frac{\sqrt{2\pi k}}{e^k} \geq \frac{1}{e^k}$$

Since this is a geometric random variable, we know the expectation is e^k □

So now we know we only need e^k trials to find a successful assignment. Now if we can verify each assignment in $c^k\, poly(n)$, this will multiply to $(c \cdot e)^k\, poly(n)$, which accomplishes our goal.

To check for a length k path, this time, we employ subset DP as described by the following recurrence, with subproblems parameterized over vertex $v_i \in G$ with label i and subset $I \subseteq [k]$.

$$KP([i], v_i) = 1$$
$$KP([i], v_j) = 0 (i \neq j)$$
$$KP(I, v_i) = \bigvee \{KP(I \cup \{j\}, v_j) \mid (v_i, v_j) \in E \wedge j \notin I\}$$

So the idea is that $KP(I, v)$ is true iff there is a path of length $|I|$ with exactly one vertex with each label in I ending on vertex v, and so we claim $\bigvee_{v \in G} KP(I, v)$ is true if there is a successfully labeled k-path.

To show correctness, we consider the k-path $\{x_{p_1}, x_{p_2}, ..., \}$. We can use an induction argument on j to see that $\forall j \in [k]$ $KP(\{p_i\}_{i \leq j}, x_{p_j})$ all return 1. For the other direction, we note that if the recurrence returns 1 for some instance $KP(I, v)$, then we can recover a length k path by taking the path in the subproblem which evaluated to 1 and extending it with v and inductively building our path until the base case, where our path is just v_i.

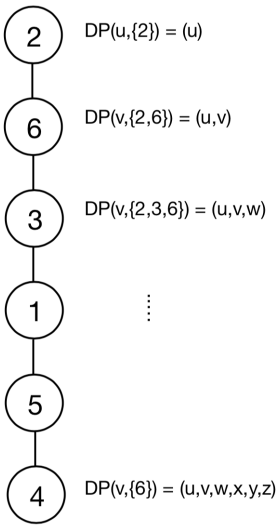

Figure 12.3: Example of DP on path of length k

We can see that the state space of this dynamic programming $O(2^k n)$ from $|\mathcal{P}([k])| = 2^k$ and $|V| \in O(n)$. We can bound the complexity of each recursive call with $O(n)$, which gives us a running time bound by $O(2^k n^2)$. Combined with the expected $O(e^k)$ trials needed for success, this results in an $O((2e)^k n^2)$ algorithm for k-path.

This technique described above is known as color coding, and was introduced by Alon et. al. [AYZ95].

2.3 Feedback Vertex Set

We can define the feedback vertex set as follows: Given an input consisting of a graph G and a length k, $\langle G, k\rangle$ is a YES-instance if and only if we can choose a set of k vertices in G to delete to make it acyclic. In this setting, we allow for multi-graphs (loops and multi-edges allowed).

As with the previous problems, we parameterize the problem with the built-in parameter k.

2.3.1 Randomized Branching With Reductions

Similar to Vertex Cover, we make some observations that justify a set of four reduction rules which allow us to kernelize our input.

- Reduction 1: If v has a self-loop, then greedily choose v. $\text{FVS}(G, k) \equiv \text{FVS}(G \setminus v, k-1)$.

  To justify the reduction, observe that the only way to remove this loop is to delete v.

- Reduction 2: If (u, v) has edge multiplicity ≥ 3, reduce it to multiplicity 2.

- Reduction 3: If v has degree ≤ 1, delete v. $\text{FVS}(G, k) \equiv \text{FVS}(G \setminus v, k)$.

  Observe that v cannot be part of a loop, so can be safely deleted.

- Reduction 4: If v has degree 2, delete v and connect its neighbors by an edge. Given $(u, v), (v, w) \in G$, then $\text{FVS}(G, k) \equiv \text{FVS}((G \setminus v) \cup (u, w), k)$.

  Observe that any loop containing v must also contain u, w, so it is sufficient to directly connect u and w.

Claim 12.4. *Let $G' = G[V \setminus \text{OPT}]$, the graph created after removing the k vertices in OPT. If none of the above reduction rules apply to G, if we pick an arbitrary $e \in G$, $e \notin G'$ with probability at least $\frac{1}{3}$.*

Proof. Observe that if none of the reduction rules can be applied, then all vertices in G must have degree at least 3. This implies that G has $\geq \frac{3}{2}n$ edges. However,if we consider the graph induced by $V \setminus OPT$, we know the graph is acyclic, so there are at most $|V \setminus OPT| - 1 \leq n - k$ edges. Therefore, $\frac{3}{2}n - (n - k) \geq \frac{n}{2}$ edges are not in G'. So $e \notin G'$ with probability at least $\frac{1}{3}$. □

This leads to the following algorithm: Apply reduction rules 1-4 until they can no longer be applied. Then, we pick a random edge and choose a random endpoint to delete. We repeat this until we have deleted k vertices, and then check if the resulting graph is acyclic. We repeat this procedure $O(6^k \log n)$ times.

From Claim 12.4, we know that with probability $\frac{1}{3}$, we pick an edge adjacent to a vertex in OPT. Then with probability $\frac{1}{2}$, we pick a vertex in OPT from the edge. In total, each run of the algorithm has a $\frac{1}{6^k}$ probability of succeeding. So running the procedure $O(6^k \log n)$ times results in a high probability of success. The overall runtime is then $O(6^k \text{poly}(n))$.

References

[AYZ95] Noga Alon, Raphael Yuster, and Uri Zwick. Color-coding. *J. ACM*, 42(4):844–856, July 1995. 2.2.3