Parameterized Algorithms

Lecture 3:
Bounded Search Trees + Basic Kernelization
Hamming Distance. Given two strings of equal length, $S^1 = s_1^1 s_2^1 \ldots s_p^1$ and $S^2 = s_1^2 s_2^2 \ldots s_p^2$, the Hamming distance between $S^1$ and $S^2$, denoted by $\text{Ham}(S^1, S^2)$, is the number of positions where $S^1$ and $S^2$ have different letters.
Hamming Distance. Given two strings of equal length, \( S^1 = s_1^1 s_2^1 \ldots s_p^1 \) and \( S^2 = s_1^2 s_2^2 \ldots s_p^2 \), the Hamming distance between \( S^1 \) and \( S^2 \), denoted by \( \text{Ham}(S^1, S^2) \), is the number of positions where \( S^1 \) and \( S^2 \) have different letters.

\[
\text{Ham}(S^1, S^2) = |\{i \in \{1,\ldots,p\} : s_i^1 \neq s_i^2\}|.
\]
Closest String: $O^*(d^d)$-Time Algorithm

**Closest String.** Given $t$ strings of length $p$, $S^1, S^2, \ldots, S^t$, and a non-negative integer $d$, decide whether there exists a string $S^*$ of length $p$ such that the Hamming distance between $S^*$ and each $S^i$ is at most $d$. 
Closest String: $O^*(d^d)$-Time Algorithm

**Closest String.** Given $t$ strings of length $p$, $S^1$, $S^2$, ..., $S^t$, and a non-negative integer $d$, decide whether there exists a string $S^*$ of length $p$ such that the Hamming distance between $S^*$ and each $S^i$ is at most $d$.

**Parameter:** $p$. 
Closest String: $O^*(d^d)$-Time Algorithm

**Closest String.** Given $t$ strings of length $p$, $S^1$, $S^2$, ..., $S^t$, and a non-negative integer $d$, decide whether there exists a string $S^*$ of length $p$ such that the Hamming distance between $S^*$ and each $S^i$ is at most $d$.

**Parameter:** $p$.

Is the problem fixed-parameter tractable (FPT)?
Closest String: $O^*(d^d)$-Time Algorithm

**Closest String.** Given $t$ strings of length $p$, $S^1$, $S^2$, ..., $S^t$, and a non-negative integer $d$, decide whether there exists a string $S^*$ of length $p$ such that the Hamming distance between $S^*$ and each $S^i$ is at most $d$.

**Parameter:** $d$. (Not the solution size!)
Closest String: $O^*(d^d)$-Time Algorithm

**Closest String.** Given $t$ strings of length $p$, $S^1, S^2, \ldots, S^t$, and a non-negative integer $d$, decide whether there exists a string $S^*$ of length $p$ such that the Hamming distance between $S^*$ and each $S^i$ is at most $d$.

**Parameter:** $d$. (Not the solution size!)

Is the problem fixed-parameter tractable (FPT)?
Closest String: $O^*(d^d)$-Time Algorithm

**Closest String.** Given $t$ strings of length $p$, $S^1$, $S^2$, ..., $S^t$, and a non-negative integer $d$, decide whether there exists a string $S^*$ of length $p$ such that the Hamming distance between $S^*$ and each $S^i$ is at most $d$.

**Parameter:** $d$. (Not the solution size!)

Is the problem fixed-parameter tractable (FPT)?

**Note:** If $d>p$, then the given instance is a yes-instance.
Closest String: $O^*(d^d)$-Time Algorithm

**Closest String.** Given $t$ strings of length $p$, $S^1$, $S^2$, ..., $S^t$, and a non-negative integer $d$, decide whether there exists a string $S^*$ of length $p$ such that the Hamming distance between $S^*$ and each $S^i$ is at most $d$.

**Parameter:** $d$. (Not the solution size!)

Is the problem fixed-parameter tractable (FPT)?

**Note:** If $d>p$, then the given instance is a yes-instance. FPT w.r.t. $d \rightarrow$ FPT w.r.t. $p$. 
Closest String: $O^*(d^d)$-Time Algorithm

**Closest String.** Given $t$ strings of length $p$, $S^1, S^2, \ldots, S^t$, and a non-negative integer $d$, decide whether there exists a string $S^*$ of length $p$ such that the Hamming distance between $S^*$ and each $S^i$ is at most $d$.

\[
S^1 = s_1^1 \ s_2^1 \ s_3^1 \ \ldots \ s_p^1 \\
S^2 = s_1^2 \ s_2^2 \ s_3^2 \ \ldots \ s_p^2 \\
S^t = s_1^t \ s_2^t \ s_3^t \ \ldots \ s_p^t
\]
Closest String: $O^*(d^d)$-Time Algorithm

**Closest String.** Given $t$ strings of length $p$, $S^1$, $S^2$, ..., $S^t$, and a non-negative integer $d$, decide whether there exists a string $S^*$ of length $p$ such that the Hamming distance between $S^*$ and each $S^i$ is at most $d$.

\[
S^* = s_1^* \ s_2^* \ s_3^* \ ... \ s_p^*
\]

\[
S^1 = s_1^1 \ s_2^1 \ s_3^1 \ ... \ s_p^1
\]

\[
S^2 = s_1^2 \ s_2^2 \ s_3^2 \ ... \ s_p^2
\]

\[
S^t = s_1^t \ s_2^t \ s_3^t \ ... \ s_p^t
\]
**Closest String:** Given $t$ strings of length $p$, $S^1, S^2, ..., S^t$, and a non-negative integer $d$, decide whether there exists a string $S^*$ of length $p$ such that the Hamming distance between $S^*$ and each $S^i$ is at most $d$.

$S^* = s^*_1 s^*_2 s^*_3 ... s^*_p$

$S^1 = s^1_1 s^1_2 s^1_3 ... s^1_p$

$S^2 = s^2_1 s^2_2 s^2_3 ... s^2_p$

$S^t = s^t_1 s^t_2 s^t_3 ... s^t_p$

**Can $S^*$ be anything?**

We need to restrict our search space if we expect to find it in FPT time.
Closest String: $O^*(d^d)$-Time Algorithm

**Closest String.** Given $t$ strings of length $p$, $S^1, S^2, ..., S^t$, and a non-negative integer $d$, decide whether there exists a string $S^*$ of length $p$ such that the Hamming distance between $S^*$ and each $S^i$ is at most $d$.

For any solution $S^*$, $\text{Ham}(S^*, S^1) \leq d$. 

\[ S^* = S^*_1 \quad S^*_2 \quad S^*_3 \quad ... \quad S^*_p \]

\[ S^1 = S^1_1 \quad S^1_2 \quad S^1_3 \quad ... \quad S^1_p \]

\[ S^2 = S^2_1 \quad S^2_2 \quad S^2_3 \quad ... \quad S^2_p \]

\[ S^t = S^t_1 \quad S^t_2 \quad S^t_3 \quad ... \quad S^t_p \]
Closest String: $O^*(d^d)$-Time Algorithm

**Observation.** A given instance $(S^1, S^2, \ldots, S^t, d)$ of Closest String is a yes-instance if and only if it is possible to edit at most $\Delta=d$ entries in $S^1$ and obtain a string $S^*$ such that $\text{Ham}(S^*, S^i) \leq d$ for all $2 \leq i \leq t$.

\[
S^* = S^*_1 \ S^*_2 \ S^*_3 \ldots \ S^*_p
\]

\[
S^1 = S^1_1 \ S^1_2 \ S^1_3 \ldots \ S^1_p
\]

\[
S^2 = S^2_1 \ S^2_2 \ S^2_3 \ldots \ S^2_p
\]

\[
S^t = S^t_1 \ S^t_2 \ S^t_3 \ldots \ S^t_p
\]

For any solution $S^*$, $\text{Ham}(S^*, S^1) \leq d$. 
``New’’ Problem. Given $t$ strings of length $p$, $S^1$, $S^2$, ..., $S^t$, and a non-negative integer $\Delta$, decide whether we can edit at most $\Delta$ entries in $S^1$ and obtain a string $S^*$ such that $\text{Ham}(S^*,S^i) \leq d$ for all $2 \leq i \leq t$.

\[
S^1 = s_1^1 \ s_2^1 \ s_3^1 \ ... \ \ s_p^1 \\
S^2 = s_1^2 \ s_2^2 \ s_3^2 \ ... \ \ s_p^2 \\
S^3 = s_1^3 \ s_2^3 \ s_3^3 \ ... \ \ s_p^3 \\
S^t = s_1^t \ s_2^t \ s_3^t \ ... \ \ s_p^t 
\]
We have a list of rules of the form [Condition]: Action. We execute the **first** rule whose condition is satisfied.

A rule where the algorithm calls itself recursively at most once is a **reduction rule**, and a rule where it calls itself recursively at least twice is a **branching rule**.
Rule 1. If $\text{Ham}(S^1, S^i) \leq d$ for all $2 \leq i \leq t$, then answer Yes.

$S^1 = s^1_1 \ s^1_2 \ s^1_3 \ ... \ s^1_p$

$S^2 = s^2_1 \ s^2_2 \ s^2_3 \ ... \ s^2_p$

$S^3 = s^3_1 \ s^3_2 \ s^3_3 \ ... \ s^3_p$

$S^t = s^t_1 \ s^t_2 \ s^t_3 \ ... \ s^t_p$
Rule 1. If $\text{Ham}(S^1, S^i) \leq d$ for all $2 \leq i \leq t$, then answer Yes.

Rule 2. If $\Delta = 0$, then answer No.

\[
S^1 = s^1_1 \ s^1_2 \ s^1_3 \ldots \ s^1_p \\
S^2 = s^2_1 \ s^2_2 \ s^2_3 \ldots \ s^2_p \\
S^3 = s^3_1 \ s^3_2 \ s^3_3 \ldots \ s^3_p \\
S^t = s^t_1 \ s^t_2 \ s^t_3 \ldots \ s^t_p
\]
Closest String: $O^*(d^d)$-Time Algorithm

**Rule 1.** If $\text{Ham}(S^1, S^i) \leq d$ for all $2 \leq i \leq t$, then answer Yes.

**Rule 2.** If $\Delta = 0$, then answer No.

\[
S^1 = s_1^1 \ s_2^1 \ s_3^1 \ldots \ s_p^1 \\
S^2 = s_1^2 \ s_2^2 \ s_3^2 \ldots \ s_p^2 \\
S^3 = s_1^3 \ s_2^3 \ s_3^3 \ldots \ s_p^3 \\
S^t = s_1^t \ s_2^t \ s_3^t \ldots \ s_p^t
\]

There exists a string $S^i$ s.t. $\text{Ham}(S^1, S^i) > d$, and $d \geq 1$. 
Closest String: $O^*(d^d)$-Time Algorithm

Rule 1. If $\text{Ham}(S^1, S^i) \leq d$ for all $2 \leq i \leq t$, then answer Yes.

Rule 2. If $\Delta = 0$, then answer No.

Rule 3. Let $S^i$ such that $\text{Ham}(S^1, S^i) > d$. Let $P$ be a set of exactly $d+1$ positions where $S^1$ and $S^i$ differ.
Rule 1. If \( \text{Ham}(S^1, S^i) \leq d \) for all \( 2 \leq i \leq t \), then answer Yes.

Rule 2. If \( \Delta = 0 \), then answer No.

Rule 3. Let \( S^i \) such that \( \text{Ham}(S^1, S^i) > d \). Let \( P \) be a set of exactly \( d+1 \) positions where \( S^1 \) and \( S^i \) differ.
- For every position \( q \) in \( P \), call \( \text{ALG}(S^{1'}, S^2, \ldots, S^t, d-1) \), where \( S^{1'} \) equals \( S^1 \) with the \( q \)-th letter being the \( q \)-th letter of \( S^i \).
Rule 1. If $\text{Ham}(S^1, S^i) \leq d$ for all $2 \leq i \leq t$, then answer Yes.

Rule 2. If $\Delta = 0$, then answer No.

Rule 3. Let $S^i$ such that $\text{Ham}(S^1, S^i) > d$. Let $P$ be a set of exactly $d+1$ positions where $S^1$ and $S^i$ differ.
- For every position $q$ in $P$, call $\text{ALG}(S^{1'}, S^2, ..., S^t, d-1)$, where $S^{1'}$ equals $S^1$ with the $q$-th letter being the $q$-th letter of $S^i$.
Return Yes iff at least one of the calls returns Yes.
Closest String: $O^*(d^d)$-Time Algorithm

Rule 1. If $\text{Ham}(S^1, S^i) \leq d$ for all $2 \leq i \leq t$, then answer Yes.

Rule 2. If $\Delta = 0$, then answer No.

Rule 3. Let $S^i$ such that $\text{Ham}(S^1, S^i) > d$. Let $P$ be a set of exactly $d+1$ positions where $S^1$ and $S^i$ differ.

- For every position $q$ in $P$, call ALG($S^1'$, $S^2$, ..., $S^t$, $d-1$), where $S^1'$ equals $S^1$ with the $q$-th letter being the $q$-th letter of $S^i$.

Return Yes iff at least one of the calls returns Yes.

Why not to let $P$ be the set of all positions where $S^1$ and $S^i$ differ?
Rule 1. If $\text{Ham}(S^1, S^i) \leq d$ for all $2 \leq i \leq t$, then answer Yes.

Rule 2. If $\Delta = 0$, then answer No.

Rule 3. Let $S^i$ such that $\text{Ham}(S^1, S^i) > d$. Let $P$ be a set of exactly $d+1$ positions where $S^1$ and $S^i$ differ.
- For every position $q$ in $P$, call $\text{ALG}(S^1', S^2, \ldots, S^t, d-1)$, where $S^1'$ equals $S^1$ with the $q$-th letter being the $q$-th letter of $S^i$.

Return Yes iff at least one of the calls returns Yes.

Correctness: Induction on $d$. 
Rule 1. If $\text{Ham}(S^1, S^i) \leq d$ for all $2 \leq i \leq t$, then answer Yes.

Rule 2. If $\Delta = 0$, then answer No.

Rule 3. Let $S^i$ such that $\text{Ham}(S^1, S^i) > d$. Let $P$ be a set of exactly $d+1$ positions where $S^1$ and $S^i$ differ.
- For every position $q$ in $P$, call $\text{ALG}(S^1', S^2, \ldots, S^t, d-1)$, where $S^1'$ equals $S^1$ with the $q$-th letter being the $q$-th letter of $S^i$.
Return Yes iff at least one of the calls returns Yes.

Correctness: Induction on $d$. 
Rule 3. Let \( S^i \) such that \( \text{Ham}(S^1, S^i) > d \). Let \( P \) be a set of exactly \( d+1 \) positions where \( S^1 \) and \( S^i \) differ.
- For every position \( q \) in \( P \), call \( \text{ALG}(S^{1'}, S^2, \ldots, S^t, \Delta-1) \), where \( S^{1'} \) equals \( S^1 \) with the \( q \)-th letter being the \( q \)-th letter of \( S^i \).
Return Yes iff at least one of the calls returns Yes.

**Forward direction.** Suppose that there exists a solution. Then, why would the algorithm return Yes?
Rule 3. Let $S^i$ such that $\text{Ham}(S^1, S^i) > d$. Let $P$ be a set of exactly $d+1$ positions where $S^1$ and $S^i$ differ.
- For every position $q$ in $P$, call $\text{ALG}(S^1', S^2, \ldots, S^t, \Delta-1)$, where $S^1'$ equals $S^1$ with the $q$-th letter being the $q$-th letter of $S^i$.
Return Yes iff at least one of the calls returns Yes.

Forward direction. Suppose that there exists a solution. Then, why would the algorithm return Yes?

Reverse direction. Suppose that the algorithm returns Yes. Then, why does there exist a solution?
Rule 3. Let $S^i$ such that $\text{Ham}(S^1,S^i) > d$. Let $P$ be a set of exactly $d+1$ positions where $S^1$ and $S^i$ differ. 
- For every position $q$ in $P$, call $\text{ALG}(S^1', S^2, \ldots, S^t, \Delta-1)$, where $S^1'$ equals $S^1$ with the $q$-th letter being the $q$-th letter of $S^i$.
Return Yes iff at least one of the calls returns Yes.

Forward direction. Suppose that there exists a solution. Then, why would the algorithm return Yes?

Reverse direction. Suppose that the algorithm returns Yes. Then, why does there exist a solution?
Similar to the analysis for $d$-Hitting Set from Lecture 2.

Running time. Number of recursive calls at the leaves:

$N(\Delta) = (d+1)N(\Delta-1)$; $N(0) = 1$. \(\rightarrow\) $N(d) = (d+1)^d$. 

**Diagram:**

- $d$
- $d-1$
- $d-2$
- $0$

The diagram represents a tree structure with nodes labeled with $d$, $d-1$, $d-2$, and 0, showing the recursive calls and their relationships.
Kernelization algorithm. Given $(\Pi,k)$, output an equivalent instance $(\Pi',k')$ of the same problem in polynomial time, such that $|\Pi'| \leq f(k)$ and $k' \leq k$.

The problem admits a kernel of size $f(k)$. 
We have a list of rules of the form [Condition]: Action. We execute the first rule whose condition is satisfied.

A rule where the algorithm calls itself recursively at most once is a reduction rule, and a rule where it calls itself recursively at least twice is a branching rule.

Now, we execute only reduction rules. (The algorithm should run in polynomial time.)
Rule 1. If there exists an isolated vertex, then remove it.

Rule 2. If there exists a vertex with at least $k+1$ neighbors, then remove it and decrement $k$ by 1. If $k$ becomes negative, then answer No.

Rule 3. If $G$ contains more than $k^2$ edges, return NO.

Kernel with (at most) $k^2$ edges and $2k^2$ vertices.
Rule 1. If there exists an isolated vertex, then remove it.

Rule 2. If there exists a vertex with at least $k+1$ neighbors, then remove it and decrement $k$ by 1. If $k$ becomes negative, then answer No.

Rule 3. If $G$ contains more than $k^2$ edges, return NO.

Besides the size of the kernel, remember to prove correctness (induction) and polynomial runtime.
**Edge Clique Cover: $2^k$-Vertex Kernel**

**Edge Clique Cover.** Given a graph $G$, and a non-negative integer $k$, decide whether there exist at most $k$ cliques in $G$ such that every edge in $G$ belongs to at least one of these cliques.
**Edge Clique Cover: \(2^k\)-Vertex Kernel**

**Edge Clique Cover.** Given a graph \(G\), and a non-negative integer \(k\), decide whether there exist at most \(k\) cliques in \(G\) such that every edge in \(G\) belongs to at least one of these cliques.

\[k = 3\]
**Edge Clique Cover: $2^k$-Vertex Kernel**

**Edge Clique Cover.** Given a graph $G$, and a non-negative integer $k$, decide whether there exist at most $k$ cliques in $G$ such that every edge in $G$ belongs to at least one of these cliques.

$k = 3$

Yes-instance
Edge Clique Cover. Given a graph $G$, and a non-negative integer $k$, decide whether there exist at most $k$ cliques in $G$ such that every edge in $G$ belongs to at least one of these cliques.

$k = 3$

Yes-instance

this edge is in more than one clique
Edge Clique Cover. Given a graph $G$, and a non-negative integer $k$, decide whether there exist at most $k$ cliques in $G$ such that every edge in $G$ belongs to at least one of these cliques.
Rule 1. If there exists an isolated vertex, remove it.
Rule 1. If there exists an isolated vertex, remove it.
Rule 1. If there exists an isolated vertex, remove it.

Rule 2. If there exists an isolate edge, remove it and decrement $k$ by 1.
Rule 1. If there exists an isolated vertex, remove it.

Rule 2. If there exists an isolate edge, remove it and decrement $k$ by 1.
Rule 1. If there exists an isolated vertex, remove it.

Rule 2. If there exists an isolate edge, remove it and decrement $k$ by 1.

Rule 3. If there exist two vertices $u, v$ such that $N[u] = N[v]$, then delete $v$. ($k$ is not changed.)
Rule 1. If there exists an isolated vertex, remove it.

Rule 2. If there exists an isolate edge, remove it and decrement $k$ by 1.

Rule 3. If there exist two vertices $u, v$ such that $N[u] = N[v]$, then delete $u$. ($k$ is not changed.)

\[
N[u] = N(u) \cup \{u\},
\]

\[
N[u] = N[v] \rightarrow \{u, v\} \text{ belongs to } E(G).
\]
**Rule 1.** If there exists an isolated vertex, remove it.

**Rule 2.** If there exists an isolate edge, remove it and decrement \( k \) by 1.

**Rule 3.** If there exist two vertices \( u,v \) such that \( N[u]=N[v] \), then delete \( v \). (\( k \) is not changed.)

**Intuitively**, since \( u,v \) are neighbors, they can belong to the same clique, and since they have the same set of neighbors, let us just put them together.
Rule 3. If there exist two vertices \( u,v \) such that \( N[u] = N[v] \), then delete \( v \). (\( k \) is not changed.)

**Forward direction.** Suppose that \((G,k)\) is a yes-instance. We need to show that \((G-v,k)\) is a yes-instance.
Rule 3. If there exist two vertices $u, v$ such that $N[u] = N[v]$, then delete $v$. ($k$ is not changed.)

Forward direction. Suppose that $(G, k)$ is a yes-instance. We need to show that $(G-v, k)$ is a yes-instance. Let $C_1, ..., C_k$ be a solution to $(G, k)$. Remove $v$ from all the cliques that contain it. This results in a solution to $(G-v, k)$. 
**Rule 3.** If there exist two vertices $u,v$ such that $N[u]=N[v]$, then delete $v$. ($k$ is not changed.)

**Reverse direction.** Suppose that $(G-v,k)$ is a yes-instance. We need to show that $(G,k)$ is a yes-instance.
Rule 3. If there exist two vertices $u,v$ such that $N[u]=N[v]$, then delete $v$. ($k$ is not changed.)

Reverse direction. Suppose that $(G-v,k)$ is a yes-instance. We need to show that $(G,k)$ is a yes-instance. Let $C_1,...,C_k$ be a solution to $(G-v,k)$. Add $v$ to every clique that contains $u$. 
Rule 3. If there exist two vertices $u, v$ such that $N[u] = N[v]$, then delete $v$. ($k$ is not changed.)

Reverse direction. Suppose that $(G-v,k)$ is a yes-instance. We need to show that $(G,k)$ is a yes-instance.

Let $C_1,...,C_k$ be a solution to $(G-v,k)$. Add $v$ to every clique that contains $u$.

- Do we still have a collection of cliques?
**Rule 3.** If there exist two vertices $u,v$ such that $N[u]=N[v]$, then delete $v$. ($k$ is not changed.)

**Reverse direction.** Suppose that $(G-v,k)$ is a yes-instance. We need to show that $(G,k)$ is a yes-instance.

Let $C_1,...,C_k$ be a solution to $(G-v,k)$. Add $v$ to every clique that contains $u$.

- Do we still have a collection of cliques?
- Why are all edges incident to $v$ covered?
Rule 3. If there exist two vertices $u, v$ such that $N[u] = N[v]$, then delete $v$. ($k$ is not changed.)

Reverse direction. Suppose that $(G-v,k)$ is a yes-instance. We need to show that $(G,k)$ is a yes-instance.

Let $C_1, ..., C_k$ be a solution to $(G-v,k)$. Add $v$ to every clique that contains $u$.
- Do we still have a collection of cliques?
- Why are all edges incident to $v$ covered?
  1. $\{w,v\}$ for $w \neq u$?
Rule 3. If there exist two vertices $u,v$ such that $N[u]=N[v]$, then delete $v$. ($k$ is not changed.)

Reverse direction. Suppose that $(G-v,k)$ is a yes-instance. We need to show that $(G,k)$ is a yes-instance.

Let $C_1,...,C_k$ be a solution to $(G-v,k)$. Add $v$ to every clique that contains $u$.

- Do we still have a collection of cliques?
- Why are all edges incident to $v$ covered?
  1. $\{w,v\}$ for $w\neq u$?
  2. $\{u,v\}$?
**Rule 3.** If there exist two vertices $u, v$ such that $N[u] = N[v]$, then delete $v$. ($k$ is not changed.)

**Reverse direction.** Suppose that $(G-v, k)$ is a yes-instance. We need to show that $(G, k)$ is a yes-instance.

Let $C_1, \ldots, C_k$ be a solution to $(G-v, k)$. Add $v$ to every clique that contains $u$.

- Do we still have a collection of cliques?
- Why are all edges incident to $v$ covered?
  1. $\{w, v\}$ for $w \neq u$?
  2. $\{u, v\}$? **Here, correctness relies on Rule 2.**
Rule 3. If there exist two vertices \( u, v \) such that \( N[u] = N[v] \), then delete \( v \). (\( k \) is not changed.)

Reverse direction. Suppose that \((G-v,k)\) is a yes-instance. We need to show that \((G,k)\) is a yes-instance.

Let \( C_1, \ldots, C_k \) be a solution to \((G-v,k)\). Add \( v \) to every clique that contains \( u \).

- Do we still have a collection of cliques?
- Why are all edges incident to \( v \) covered?
  1. \( \{w,v\} \) for \( w \neq u \)?
  2. \( \{u,v\} \)?

This results in a solution to \((G,k)\).
Rule 1. If there exists an isolated vertex, remove it.

Rule 2. If there exists an isolated edge, remove it and decrement $k$ by 1.

Rule 3. If there exist two vertices $u,v$ such that $N[u]=N[v]$, then delete $v$. ($k$ is not changed.)
Rule 1. If there exists an isolated vertex, remove it.

Rule 2. If there exists an isolate edge, remove it and decrement $k$ by 1.

Rule 3. If there exist two vertices $u,v$ such that $N[u]=N[v]$, then delete $v$. ($k$ is not changed.)

Rule 4. If there exist more than $2^k$ vertices, then return No.

Kernel with (at most) $2^k$ vertices.
Rule 4. If there exist more than $2^k$ vertices, then return No.

Correctness. Suppose that $(G,k)$ is a yes-instance (on which Rules 1-3 do not apply). Let $C_1,...,C_k$ be a solution to $(G,k)$. We need to show that $|V(G)| \leq 2^k$. 
Correctness. Suppose that \((G,k)\) is a yes-instance (on which Rules 1-3 do not apply). Let \(C_1,\ldots,C_k\) be a solution to \((G,k)\). We need to show that \(|V(G)| \leq 2^k\).
Correctness. Suppose that \((G,k)\) is a yes-instance (on which Rules 1-3 do not apply). Let \(C_1,...,C_k\) be a solution to \((G,k)\). We need to show that \(|V(G)| \leq 2^k\).

With every vertex \(v\), associate a binary vector \((b_1,...,b_k)\) where \(b_i=1\) iff \(v\) belongs to \(C_i\).
**Edge Clique Cover: \(2^k\)-Vertex Kernel**

**Correctness.** Suppose that \((G,k)\) is a yes-instance (on which Rules 1-3 do not apply). Let \(C_1,\ldots,C_k\) be a solution to \((G,k)\). We need to show that \(|V(G)| \leq 2^k\).

With every vertex \(v\), associate a binary vector \((b_1,\ldots,b_k)\) where \(b_i=1\) iff \(v\) belongs to \(C_i\).
- No vertex is associated with 0=(0,...,0) (by Rule 1).
**Correctness.** Suppose that \((G,k)\) is a yes-instance (on which Rules 1-3 do not apply). Let \(C_1,...,C_k\) be a solution to \((G,k)\). We need to show that \(|V(G)| \leq 2^k\).

With every vertex \(v\), associate a binary vector \((b_1,...,b_k)\) where \(b_i=1\) iff \(v\) belongs to \(C_i\).

- No vertex is associated with \(0=(0,...,0)\) (by Rule 1).
- No two vertices can have the same non-0 vector (by Rule 3).
Edge Clique Cover: $2^k$-Vertex Kernel

**Correctness.** Suppose that $(G,k)$ is a yes-instance (on which Rules 1-3 do not apply). Let $C_1,\ldots,C_k$ be a solution to $(G,k)$. We need to show that $|V(G)| \leq 2^k$.

With every vertex $v$, associate a binary vector $(b_1,\ldots,b_k)$ where $b_i=1$ iff $v$ belongs to $C_i$.
- No vertex is associated with $0=(0,\ldots,0)$ (by Rule 1).
- No two vertices can have the same non-0 vector (by Rule 3).
→ No two vertices can have the same vector.
Correctness. Suppose that \((G,k)\) is a yes-instance (on which Rules 1-3 do not apply). Let \(C_1,\ldots,C_k\) be a solution to \((G,k)\). We need to show that \(|V(G)| \leq 2^k\).

With every vertex \(v\), associate a binary vector \((b_1,\ldots,b_k)\) where \(b_i = 1\) iff \(v\) belongs to \(C_i\).

- No vertex is associated with \(0 = (0,\ldots,0)\) (by Rule 1).
- No two vertices can have the same non-0 vector (by Rule 3).

\[\rightarrow\] No two vertices can have the same vector.

\[\rightarrow\] Because there are \(2^k\) binary vectors with \(k\) entries, \(G\) has at most \(2^k\) vertices.
Theorem. A decidable parameterized problem is FPT if and only if it admits a kernel.
Theorem. A decidable parameterized problem is FPT if and only if it admits a kernel.

Easy direction. Let Q be a decidable parameterized problem that admits a kernel. Then, Q has a decision algorithm A, and a kernelization algorithm B.
Theorem. A decidable parameterized problem is FPT if and only if it admits a kernel.

Easy direction. Let Q be a decidable parameterized problem that admits a kernel. Then, Q has a decision algorithm A, and a kernelization algorithm B. Given an instance $(\Pi,k)$ of Q,
Theorem. A decidable parameterized problem is FPT if and only if it admits a kernel.

Easy direction. Let Q be a decidable parameterized problem that admits a kernel. Then, Q has a decision algorithm A, and a kernelization algorithm B. Given an instance $(\Pi,k)$ of Q,
- Run the kernelization algorithm B (polynomial time).
**Theorem.** A decidable parameterized problem is FPT if and only if it admits a kernel.

**Easy direction.** Let Q be a decidable parameterized problem that admits a kernel. Then, Q has a decision algorithm \( A \), and a kernelization algorithm \( B \). Given an instance \((\Pi,k)\) of Q,
- Run the kernelization algorithm \( B \) (polynomial time).
- Run the decision algorithm \( A \) on the output of \( B \) (whose size is a function of \( k \)).
Theorem. A decidable parameterized problem is FPT if and only if it admits a kernel.

Easy direction. Let \( Q \) be a decidable parameterized problem that admits a kernel. Then, \( Q \) has a decision algorithm \( A \), and a kernelization algorithm \( B \). Given an instance \((\Pi,k)\) of \( Q \),
- Run the kernelization algorithm \( B \) (polynomial time).
- Run the decision algorithm \( A \) on the output of \( B \) (whose size is a function of \( k \)).
- Return the same answer as \( A \).
**FPT ↔ Kernel**

**Theorem.** A decidable parameterized problem is FPT if and only if it admits a kernel.

**Easy direction.** Let $Q$ be a decidable parameterized problem that admits a kernel. Then, $Q$ has a decision algorithm $A$, and a kernelization algorithm $B$. Given an instance $(\Pi,k)$ of $Q$,
- Run the kernelization algorithm $B$ (polynomial time).
- Run the decision algorithm $A$ on the output of $B$ (whose size is a function of $k$).
- Return the same answer as $A$.

1. **Running time.**
2. **Correctness.**
**Theorem.** A decidable parameterized problem is FPT if and only if it admits a kernel.

**Easy direction.** Let Q be a decidable parameterized problem that admits a kernel. Then, Q has a decision algorithm $A$, and a kernelization algorithm $B$. Given an instance $(\Pi, k)$ of Q,
- Run the kernelization algorithm $B$ (polynomial time).
- Run the decision algorithm $A$ on the output of $B$ (whose size is a function of $k$).
- Return the same answer as $A$.

1. Running time. FPT.
2. Correctness. ✔
Theorem. A decidable parameterized problem is FPT if and only if it admits a kernel.

Surprising direction. Let Q be a parameterized problem that admits a parameterized algorithm A. Given an instance $(\Pi,k)$ of Q, A decides it in time $f(k)|\Pi|^c$. 
Theorem. A decidable parameterized problem is FPT if and only if it admits a kernel.

Surprising direction. Let Q be a parameterized problem that admits a parameterized algorithm A. Given an instance \((\Pi, k)\) of Q, A decides it in time \(f(k)|\Pi|^c\).

- Run A on \((\Pi, k)\) for at most \(|\Pi|^{c+1}\) steps.
- If A terminated, return its decision.
- Else, return \((\Pi, k)\).
Theorem. A decidable parameterized problem is FPT if and only if it admits a kernel.

Surprising direction. Let \( Q \) be a parameterized problem that admits a parameterized algorithm \( A \). Given an instance \((\Pi, k)\) of \( Q \), \( A \) decides it in time \( f(k)\pi^c \).
- Run \( A \) on \((\Pi, k)\) for at most \( |\Pi|^{c+1} \) steps.
- If \( A \) terminated, return its decision.
- Else, return \((\Pi, k)\).

1. Running time.
2. Correctness.
**Theorem.** A decidable parameterized problem is FPT if and only if it admits a kernel.

**Surprising direction.** Let Q be a parameterized problem that admits a parameterized algorithm A. Given an instance $(Π,k)$ of Q, A decides it in time $f(k)|Π|^c$.
- Run A on $(Π,k)$ for at most $|Π|^{c+1}$ steps.
- If A terminated, return its decision.
- Else, return $(Π,k)$.

1. **Running time.** Polynomial time.
2. **Correctness.** Equivalent.
**Theorem.** A decidable parameterized problem is FPT if and only if it admits a kernel.

**Surprising direction.** Let $Q$ be a parameterized problem that admits a parameterized algorithm $A$. Given an instance $(\Pi, k)$ of $Q$, $A$ decides it in time $f(k) |\Pi|^c$.

- Run $A$ on $(\Pi, k)$ for at most $|\Pi|^{c+1}$ steps.
- If $A$ terminated, return its decision.
- Else, return $(\Pi, k)$.

1. **Running time.** Polynomial time.
2. **Correctness.** Equivalent.
3. **Size.**
FPT ↔ Kernel

Theorem. A decidable parameterized problem is FPT if and only if it admits a kernel.

Surprising direction. Let Q be a parameterized problem that admits a parameterized algorithm A. Given an instance $(\Pi,k)$ of Q, A decides it in time $f(k)|\Pi|^c$.
- Run A on $(\Pi,k)$ for at most $|\Pi|^{c+1}$ steps.
- If A terminated, return its decision.
- Else, return $(\Pi,k)$.

2. Correctness. Equivalent.
3. Size.
**FPT ↔ Kernel**

**Theorem.** A decidable parameterized problem is FPT if and only if it admits a kernel.

**Surprising direction.** Let $Q$ be a parameterized problem that admits a parameterized algorithm $A$. Given an instance $(\Pi, k)$ of $Q$, $A$ decides it in time $f(k) |\Pi|^c$.
- Run $A$ on $(\Pi, k)$ for at most $|\Pi|^{c+1}$ steps.
- If $A$ terminated, return its decision.
- Else, return $(\Pi, k)$.

1. **Running time.** Polynomial time.
2. **Correctness.** Equivalent.
3. **Size.**
   - If $A$ terminated, $f(k) |\Pi|^c > |\Pi|^{c+1}$.
**Theorem.** A decidable parameterized problem is FPT if and only if it admits a kernel.

**Surprising direction.** Let $Q$ be a parameterized problem that admits a parameterized algorithm $A$. Given an instance $(\Pi,k)$ of $Q$, $A$ decides it in time $f(k)|\Pi|^c$.

- Run $A$ on $(\Pi,k)$ for at most $|\Pi|^{c+1}$ steps.
- If $A$ terminated, return its decision.
- Else, return $(\Pi,k)$.

**1. Running time.** Polynomial time.
**2. Correctness.** Equivalent.
**3. Size.**

If $A$ terminated, ✓
Else, $f(k)|\Pi|^c > |\Pi|^{c+1}$.
→ $|\Pi| < f(k)$. 
Theorem. A decidable parameterized problem is FPT if and only if it admits a kernel.

Surprising direction. Let Q be a parameterized problem that admits a parameterized algorithm A. Given an instance $(\Pi,k)$ of Q, A decides it in time $f(k)|\Pi|^c$.
- Run A on $(\Pi,k)$ for at most $|\Pi|^{c+1}$ steps.
- If A terminated, return its decision.
- Else, return $(\Pi,k)$.

2. Correctness. Equivalent.
3. Size. $< f(k) + k$.

If A terminated, $f(k)|\Pi|^c > |\Pi|^{c+1}$.
Else, $f(k)|\Pi|^c > |\Pi|^{c+1}$.
$\rightarrow |\Pi| < f(k)$. 
**FPT ↔ Kernel**

**Theorem.** A decidable parameterized problem is FPT if and only if it admits a kernel.

**Surprising direction.** Let Q be a parameterized problem that admits a parameterized algorithm A. Given an instance \((\Pi, k)\) of Q, A decides it in time \(f(k)|\Pi|^c\).
- Run A on \((\Pi, k)\) for at most \(|\Pi|^{c+1}\) steps.
- If A terminated, return its decision.
- Else, return \((\Pi, k)\).

1. **Running time.** Polynomial time.
2. **Correctness.** Equivalent.
3. **Size.** \(< f(k) + k\).

For an NP-hard problem, we cannot extract a poly. kernel from this proof.