Grade: Major revisions
- Proof 1 uses unsafe RR
- Proofs 2-3 have minor problem

Proof Review Exercise #1

October 11, 2017

Week-2

Background

k-MaximalMatching	
Input:	A graph $G = (V, E)$ and a non-negative integer k
Parameter:	k
Question:	Does G have a maximal matching of size at most k?

Main Result

Theorem 1. *There is a $O(k^2)$ kernel for k-MaximalMatching.*

Proof. Denote our graph by $G = (V, E)$.

High-level strategy: We will first relate maximal matchings to vertex cover, which gives us a notion of "high-degree vertices." We will then use these vertices to identify "useless" vertices that can be safely removed from the graph. After applying this reduction method, we will reason that we are left with a $O(k^2)$ kernel.

Reduction methods and safeness: Note that every maximal matching can be turned into a vertex cover by taking both endpoints of every matched edge. If there exists some uncovered edge, this means that all the edges connecting to its neighbors are not in the matching and we can add this to the matching set we originally have, which contradicts the maximality of our matching. Therefore, if we hope to get a maximal matching of size at most k, this is equivalent to get the minimum vertex cover which has size less than $2k + 1$. Using this fact, let X be the set of vertices whose degree is at least $2k + 1$. We know that each vertex in X must be an endpoint of a matched edge, otherwise all of its neighbors must be endpoints of matched edges, which cannot be done with a matching of size at most k.

While we have identified this set of "high-degree vertices," we cannot directly remove them from the graph. Specifically, we know each high-degree vertex is an endpoint of a matched edge, but we don't actually know *which* edge. Instead, we will use X to derive another reduction.

We briefly describe our idea here. We first search the graph G to find such vertex that has degree at least $2k + 1$. If such high-degree vertices have more than $2k$ numbers, then this graph cannot be k-maximal matching. If the number of such vertices is less than $2k$ numbers, then the size of X is at most $2k$ and each vertex in X has degree $2k+1$. Then we focus on the set $G \setminus X$, since each vertex in the set $G \setminus X$ has degree at most $2k$, then we choose arbitrary edge between a vertex $v \in G \setminus X$ and its neighbor $\texttt{neighbor}(v)$. Also the degree of $\texttt{neighbor}(v)$ also has at most $2k$ degrees, then $G \setminus X$ has at most $(n-2k) \cdot 2k$ edges which implies that $G \setminus X$ has $O(k^2)$ vertices if the number of edges in the set $G \setminus X$ is large enough. This is equivalent to the condition that if $|G \setminus X| > 4k^2$ then we get a `NO` answer. Therefore in total we have a kernel of size $O(k^2)$ (having consider those $2k$ vertices that has $2k+1$ degrees) for the k-MaximalMatching.

Formalizing this discussion, we define the following reduction method:

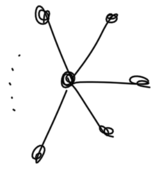
1. Identify the set of vertices with degree at least $2k+1$. Denote this set X.
2. If $|X| \geq 2k+1$ or $|G \setminus X| > 4k^2$, we have a `NO` answer; otherwise we get a `YES` answer, which means that we have a $O(k^2)$ kernel.

Run time: Identifying the set of high-degree vertices X will take $O(n^2)$ if the graph is stored as an adjacency matrix. Therefore the reduction procedure would take $O(n^2)$ time.

Output bound: First, we know that $|X| \leq 2k$, otherwise we'd have enough vertices to pair off to get a maximal matching larger than k. Second, in a `YES`-instance, the graph $G \setminus X$ is bounded by $O(k^2)$ as we analyzed above. Since each vertex in the set $G \setminus X$ has degree at most $2k$. Thus we select an arbitrary edge and add it to the matching and then delete all the edges incident on two endpoints of this arbitrary edge, we remove at most $4k-1$ edges each time. Therefore, there are at most $4k^2$ edges in the set $G \setminus X$ if a graph has more than $4k^2$ edges. Therefore, in total we have $O(k^2)$ vertices in G. Namely we have a $O(k^2)$ kernel for the k-MaximalMatching problem. □

Not safe

$4k^2$ leaves



Yet has min maximal matching 1

Week-3

Background

Definition 1. *A graph is* perfect *if, for every induced subgraph, the clique number is equal to the chromatic number.*

Definition 2. *We denote an induced cycle on three vertices as a* triangle*. A graph is* triangle-free *if it does not contain a triangle.*

Definition 3. *Let $G = (V, E)$ be a graph. An* odd cycle transversal *$S \subseteq V$ is a set of vertices such that $G \setminus S$ is bipartite.*

k-OCT	
Input:	A graph $G = (V, E)$ and a non-negative integer k
Parameter:	k
Question:	Does G have an odd cycle transversal of size at most k?

Main Result

Lemma 1. *A perfect graph is bipartite if and only if it is triangle-free.*

Proof.
Forward direction: Assuming that G is perfect and bipartite, we will show that it is triangle-free. Suppose not, and there exists an induced triangle on vertices u, v, w. Since G is perfect and the triangle is a 3-clique, G must have chromatic number at least 3. But this contradicts the assumption that G is bipartite, since every bipartite graph is 2-colorable (e.g. use a unique color per partite set).

Backwards direction: Assuming that G is triangle-free and perfect, we will show that it is bipartite. Since G is triangle-free, it cannot contain a clique larger than K_2. We split into two cases:

Case 1. G has no edges. Then G is trivially bipartite – put every vertex in one partite set.

Case 2. G has at least one edge. Then by the assumption that G is perfect and the fact that its largest clique is K_2, it has a 2-coloring. This coloring implies a bipartite construction – have a partite set per color.

In all cases, G is bipartite. □

Theorem 2. *k-OCT has a $3^k n^{O(1)}$ branching algorithm in perfect graphs.*

Proof.

High-level strategy: Using the previous lemma, we know that we need to break every triangle to make the graph bipartite. We know that every triangle contains a vertex in an optimal OCT set, but we don't know which one to add – so we will branch!

Algorithm description: Define a recursive method that takes as input a graph G, a parameter k, and a (partial) OCT set S. Go through all triples of vertices u, v, w until a triangle is found. If a triangle is found, then branch into three subproblems – one where u is added to S, one where v is added to S, and one where w is added to S; in all cases remove the vertex from G and decrease k by one when recursing. If a triangle is not found, then return `YES` if $k \geq 0$, and if a triangle is found and $k = 0$, return `NO`.

Should check $k \leq 0$ first

Correctness: We terminate when no triangle could be found, so the graph is bipartite (by Lemma 1) and S is a valid OCT set. If $k \geq 0$ then we know that $|S| \leq k$ and the solution should be `YES`. Likewise, if $k < 0$ then we know that $|S| > k$ and the solution should be `NO`. Therefore our algorithm always constructs a valid OCT set, and correctly decides if it is too large or not.

Run time: Enumerating all triples of vertices takes $\binom{n}{3} = O(n^3)$ time, and checking if a triple is a triangle takes $O(1)$ time if we store the graph with an adjacency matrix. Examining k to decide `YES` or `NO` takes $O(1)$ time, so each node of the branching tree takes $O(n^3)$ time to execute. Every time we branch we decrease k by exactly 1, therefore the branching tree has depth k. Additionally, we always branch into three subproblems, therefore there are at most $O(3^k)$ nodes in the branching tree. In total, then, the run time is $O(3^k n^3)$. □

Observation. *Note the similarities to the $O(k^2)$ kernel for k-ClusterEdit! Would you expect to also get a $O(k^2)$ kernel for k-OCT? Would you expect to get a $3^k n^{O(1)}$ branching algorithm for k-ClusterEdit?*

Week-4

Background

Definition 4. *Let $G = (V, E)$ be a graph. A* feedback vertex set *$S \subseteq V$ is a set of vertices such that $G \setminus S$ is a forest (e.g. every cycle ("feedback loop") is broken).*

Definition 5. *Let $G = (V, E)$ be a graph. A* dominating set *$S \subseteq V$ is a set of vertices such that, for all vertices $v \in V$, either $v \in S$ or v has a neighbor in S.*

k-DOMINATINGSET	
Input:	A graph $G = (V, E)$ and a non-negative integer k
Parameter:	k
Question:	Does G have a dominating set of size at most k?

Main Result

Theorem 3. *There is an FPT algorithm for k-dominating set parameterized by feedback vertex set.*

Proof. Denote our graph by $G = (V, E)$.

High-level strategy: We will start by considering all possibilities for searching trees in order to find optimal solution for each tree from the given feedback vertex set X. There are $2^{|X|}$ combinations we can obtain from X. Finally we combine the minimum one of each tree of all possible combinations of X. We provide more details below.

Algorithm description: Given a feedback vertex set X, we consider all the combinations for which a vertex in X is put in the dominating set or not.

- If a vertex v in X is in the dominating set. We have known that $G \setminus X$ is a forest by definition, so removing v's neighbors that are in $G \setminus X$ doesn't have impact on our result. So this step is safe.
- We recursively do the following process for each tree until all the vertices in the tree is dominated by the vertices put into the dominating set: [Find the parent vertex of those leaves that won't separate the trees further even if we remove them,] and then we put the parent vertex into the dominating set and delete this parent and its child. ?
- Finally we have an optimal dominating set for each combination in X. We consider all combinations from X. If it is possible to find the dominating set `DS`, where `DS` $\leq k$, for the graph G, then this is a `YES` instant, otherwise we output `NO` instance.

Correctness: We check every possible dominating set on the feedback vertex set, there are no other options here. Everything else must live on a tree. Observe that all the trees are separated by the feedback vertex set by its definition, then we note that the dominating set for each tree cannot have impact on any other tree. Namely, each tree is independent from each other. Therefore, we can compute a optimal solution for each tree using dynamic programming. Thus the solution must be found using our method if it exists and outputs `YES`, otherwise our algorithm would outputs `NO`.

Run time: Enumerating on all combinations from the feedback vertex set X takes time $O(2^{|X|})$. Then for finding optimal solution in each tree we use dynamic programming from the bottom of the tree to the root and it takes time $O(|E|)$. Therefore in total we take time $O(2^{|X|}|E|) = O(2^{|X|}n)$, where n is the size of graph, which is parameterized by the size of the feedback vertex set. □