Proof Review Exercise #1

September 28, 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.*

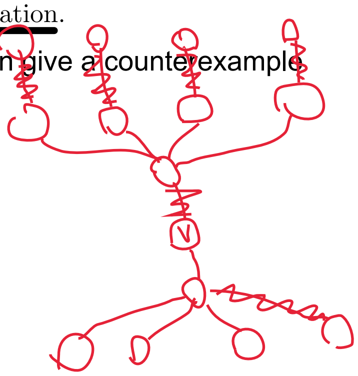
Proof. Denote our graph by G. I would like denoting G=(V,E) major revision

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 not, there exists some uncovered edge not in the matching, which contradicts the maximality of our matching. Therefore, if we hope to get a maximum matching of size at most k, we also hope that an optimal vertex cover for this graph has size at most $2k$. 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. Examine the graph $G \setminus X$, and let v be an isolate in this graph (if one exists). We know that the neighbors of v must be high-degree vertices (which are endpoints of matched edges), therefore we know that we would never add an edge incident to v to our maximal matching. In general, then, we may remove all isolated vertices in $G \setminus X$ from G. See Figure for a visualization.

This method is problematic, we can give a counterexample



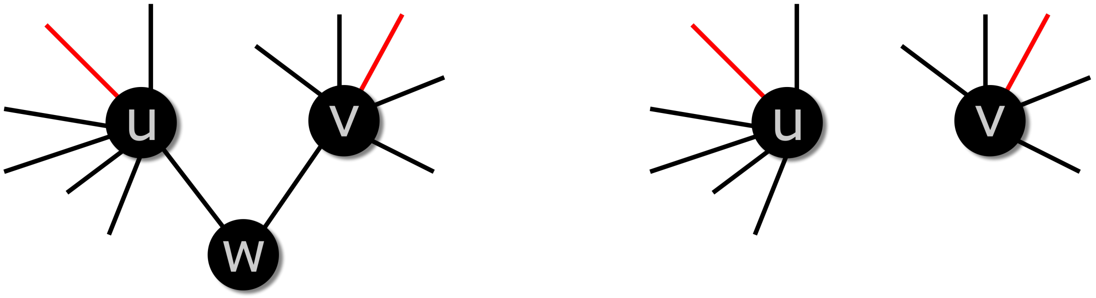


Figure 1: (Left) Let u and v be two high-degree vertices. Because of how we defined "high-degree," we know that u and v must be endpoints of matched edges, shown in red. (Right) If w is only connected to u and v then we can remove w from the graph. This is safe because if the edge (u, w) (or (v, w)) was in the matching, then we simply add an edge incident to u (or w) to the matching – this guarantees that u (or w) is still covered.

This figure is correct, but the proof strategy is incorrect as we described above

Formalizing this discussion, we define the following two reduction methods:

RR1. Identify the set of vertices with degree at least $2k + 1$. Denote this set X. Do not remove X from G, merely identify them.

RR2. ~~Identify the set of isolated vertices in G/X and remove them from G.~~

Run time: Identifying the set of high-degree vertices X will take $O(n^2)$ if the graph is stored as an adjacency matrix. ~~Identifying the isolates of $G \setminus X$ will also take $O(n^2)$.~~ Therefore both reduction routines execute in $O(n^2)$ 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 $4k^2$. ~~By the second rule and the fact that we have a *maximal* matching~~, we know that every vertex must be an endpoint of a matched edge. There are at most k such edges, each of whose endpoints can have degree at most $2k$, therefore there are at most $4k^2$ vertices in $G \setminus X$. In total, then, the produced instance has at most $O(k^2)$ vertices. □

As we show in the picture we draw above, this is problematic.

Week-3

minor revision

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.

This proof is basically correct

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).

should specify what is u,v,w

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 an 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.

G should has "at least" one edge

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 `NO` otherwise.

This description is a bit ambiguous, it should be "If a triangle is found and k=0, return NO"

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

major revision

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.*

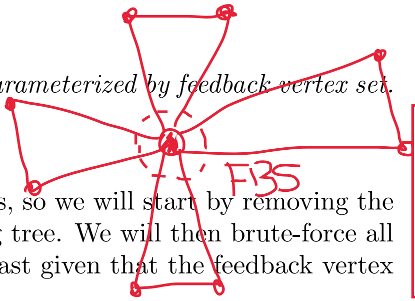
k-DominatingSet	
Input:	A graph $G = (V, E)$ and a non-negative integer k
Parameter:	k
Question:	Does G have a dominating set of size at most k?

Main Result

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

Proof. Denote our graph by G.

A counterexample of this method



If we find dominating set on trees firstly, then we cannot find the dominating set in the picture on the left.

High-level strategy: NP-hard problems typically are easy on trees, so we will start by removing the feedback vertex set and solving dominating set on the remaining tree. We will then brute-force all ways of merging the feedback vertex set into the tree, which is fast given that the feedback vertex set is small.

High-level strategy is incorrect. we cannot directly brute-force all ways of merging the feedback vertex set into the tree. We should consider all possibilities that whether each vertex in the feedback vertex set is in the dominated set or not. Solution for one tree cannot represent for all trees for a vetex in the feedback vertex set.

Algorithm description: Let F be our feedback vertex set and D be our (initially-empty) dominating set. By definition, $G \setminus F$ is a tree. We can use a “dynamic programming”-style approach to compute an optimal dominating set: Starting from the leaves, check whether it is better to have a vertex or one of its (possibly-non-existent) children in a dominating set. At this point we have an optimal dominating set, but only for the tree $G \setminus F$. We now test every way of merging F into this solution using brute force over F. If it was possible to finish the dominating set for G with $|D| \leq k$ then output `YES`, otherwise output `NO`.

Since the high-level strategy is incorrect, it's a fatal mistake for this problem.

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 and we can compute the exact answer here, so a solution of size at most k will always be found if it exists, and otherwise we answer `NO`~~.

Run time: Computing the exact dominating set on the tree takes at most $O(n)$ time if we start from the leaves and ascend up the tree. Computing all dominating sets of the feedback vertex set takes time $\binom{n}{k} = O(n^k)$, which is a polynomial since k is a constant. Therefore the full algorithm takes $O(n^k n) = O(n^{k+1})$ time. □

Should be O(2^|S| |E|), the algorithm should be parameterized by the size of the feedback vertex set, and the running time analysis is wrong since the incorrectness of the strategy.