Deterministic branching algorithms for parameterized Co-Path/Cycle Packing and three variants

R. Kuipers

A thesis presented for the degree of
Master of Science in Computing Sciences

Department of Information and Computing Sciences
Utrecht University
The Netherlands
October 2017

Abstract

The Co-Cycle/Path Packing problem that tries to find a set of vertices that, when removed, leaves a graph of maximum degree 2 is a prominent problem in the graph theory field. The related Vertex Cover problem, which finds a deletion set where the remaining graph has maximum degree 0, is one of the most famous graph theory problems. In this thesis we describe a deterministic parameterized algorithm for Co-Cycle/Path Packing which uses branch-and-bound techniques. This algorithm is shown to have a time complexity of $\mathcal{O}^*(3.0607^k)$, which improves upon the previous best known deterministic bound. A new problem which looks for a deletion set such that the remaining graph is 2-regular is also discussed, and a branching algorithm with time-bound $\mathcal{O}^*(3^k)$ is shown for it. Additionally, two variants of these two problems that add the requirement that the remaining graph is a single connected component are introduced and shown to both have an algorithm that runs in $\mathcal{O}(2^k n^3)$ time. For the three new problems, $\mathcal{NP}$-completeness is also proven.

Contents

1 Introduction

In this thesis we will consider the Co-Path/Cycle Packing problem, and several problems related to it. Co-Path/Cycle Packing is a graph theory problem which has applications in computational biology [CFF+10], where we try to find a set of vertices that when removed leave a graph in which all vertices have degree 2 or lower. A different name for this problem is Bounded-Degree-2 Vertex Deletion, and it is a specific version of the Bounded-Degree-d Vertex Deletion problem where d is the maximum degree of the remaining graph, 2 in our case. Another specific version of this problem is Vertex Cover, which is equal to Bounded-Degree-d Vertex Deletion when $d = 0$.

We will describe and analyze a deterministic branching algorithm for the parameterized Co-Path/Cycle Packing problem. Additionally, we construct and analyze such algorithms for three related problems which, contrary to the original problem, have so far not been discussed in the literature as far as we know. The Co-Cycle Packing problem changes the requirement that the remaining graph has maximum degree 2, to a requirement that every vertex in it has exactly degree 2, in other words that it is 2-regular. This ensures the remaining graph is a set of cycles, instead of a set of paths and cycles. The Induced Cycle Deletion Set and Induced Path/Cycle Deletion Set are variations on the Co-Cycle Packing and Co-Path/Cycle Packing problems respectively, that add the requirement that the remaining graph has only one connected component. This leaves the remaining graph as a single cycle or path.

All these problems are $\mathcal{NP}$-complete, which we will prove for the three new problems. For Co-Path/Cycle Packing we refer to a proof in [LY80] which proves the entire class of hereditary vertex deletion problems is $\mathcal{NP}$-complete. For the two connected variants we provide two similar deterministic parameterized algorithms that both run in $\mathcal{O}(2^k n^3)$ time, and an algorithm for Co-Cycle Packing that runs in $\mathcal{O}^*(3^k)$ time. The algorithm we provide for Co-Path/Cycle Packing has time complexity $\mathcal{O}^*(3.0607^k)$, which is slightly faster than the algorithm previously found in [Xia16] which has time complexity $\mathcal{O}^*(3.0645^k)$.

1.1 Previous Work

As stated above, the only one of these problem for which we have found previous research was Co-Path/Cycle Packing. The earliest algorithm we found for it was in [CFF+10], which describes an $\mathcal{O}^*(3.24^k)$ parameterized algorithm, and provides a $37k$ vertex kernel for this problem. It also shows

the problem is APX-hard, which means it does not have a kernel smaller than $2k$ unless the Exponential time hypothesis is false [ACG+12].

In [Xia16] an deterministic algorithm for Bounded-Degree-d Vertex Deletion is described, which has time complexity $\mathcal{O}^*((d+1)^k)$ for any $d \geq 3$, and $\mathcal{O}^*(3.0645^k)$ for $d = 2$. It also proves Bounded-Degree-d Vertex Deletion is W[2]-hard with respect to k for unbounded d, but FPT for any fixed $d \geq 0$. A proof for the W[2]-hardness for unboudned d was also given in [FGMN11], which also proves that for any $\epsilon > 0$ a $\mathcal{O}(k^{1+\epsilon})$ vertex kernel exists for $d \geq 2$.

It is shown in [BBNU12] that Bounded-Degree-d Vertex Deletion with a fixed d is W[1]-hard with respect to the parameter treewidth, but FPT when parameterized by the combined parameter treewidth and k, or when parameterized by the feedback edge set number.

A randomized parameterized algorithm for Co-Path/Cycle Packing was shown in [FWLC15], which has a running time of $\mathcal{O}^*(3^k)$. This is faster than the algorithm provided in this thesis, but the algorithm presented here is the fastest deterministic algorithm thus far.

If we look at algorithms for other fixed cases of Bounded-Degree-d Vertex Deletion , we find there is a $\mathcal{O}(1.2738^k + kn)$ time algorithm for Vertex Cover ($d = 0$) in [CKX10]. For $d = 1$, [Wu15] uses a technique called Measure and Conquer to describe a $\mathcal{O}(1.882^k + |V||E|)$ time algorithm. For specific values of $d \geq 3$ we have found no research, so the $\mathcal{O}^*((d+1)^k)$ time algorithm [Xia16] appears to be the best algorithm for these problems.

1.2 Preliminaries

All graphs discussed in this thesis is an undirected simple graph. Graphs are denoted as $G = (V, E)$, where V is the set of vertices, and E the set of edges. The singleton set $\{v\}$ will usually be written as v, and an edge $\{u, v\}$ is sometimes shortened to uv. The symbol n will denote the number of vertices $|V|$, and $m = |E|$ denotes the number of edges. The *open neighborhood* of a vertex $v \in V$ is denoted as $N(v) = \{u \in V \mid uv \in E, u \neq v\}$, and the *closed neighborhood* as $N[v] = N(v) \cup v$. If $X \subseteq V$, then $N[X] = \bigcup_{v \in X} N[v]$ and $N(X) = N[X] \setminus X$. The *degree* of vertex v is denoted as $d(v) = |N(v)|$. A graph where each vertex has degree d or lower is called a maximum degree d graph, and a graph where all vertices have degree exactly d is called *d-regular*. If a vertex v is said to be *deleted* from the graph, this means both the vertex and any incident edges are removed from the graph. The notations $G - v$ and $G - X$ are shorthand for the resulting graph after the deletion of a vertex and a set of vertices respectively. Conversely, $G[X]$ denotes the graph when all vertices in $V \setminus X$ are deleted. The degree within specific graph H is denoted $d_H(v)$, but if we are only taking about one graph $d(v)$ is usually used. Two

operations we occasionally use are *contracting* a vertex and *subdividing* an edge. For a vertex to be contracted, it needs to be degree 2 and the two neighbors cannot be adjacent to each other. When contracted, a vertex is removed from the graph and an edge is added between the two neighbors of the removed vertex. The opposite of this action is subdivision of an edge, in which case the edge is removed from the graph, and a new vertex which is adjacent to the two endpoints of the edge is added.

The algorithms discussed in this thesis are branch-and-bound algorithms, a widely used technique which traces back to [DP60]. When analyzing such algorithms, we focus on the number of base cases, each of which can often be solved in polynomial time. In the parameterized algorithms discussed in this thesis have a maximum number of vertices k that are allowed to be put in the deletion set. Therefore, if we have two branches that each put one vertex in the deletion set, we know the number of base cases can at most be 2^k, since each branch doubles it and we can have at most k layers of branching. We would say this branching rule has branching factor 2 and branching vector $(1, 1)$. If we have a more complex branching rule that has three branching that each put one vertex in the deletion set, and one branch that puts two vertices in it, it has branching vector $(1, 1, 1, 2)$. If we now solve the equation

$$x^k = 3x^{k-1} + x^{k-2}$$

for x we get the value 3.3027, which is the branching factor of this rule. That means an algorithm with this as the slowest branching rule has at most 3.3027^k base cases, which would lead to a time complexity of $\mathcal{O}^*(3.3027^k)$. In this thesis, we use a new notation for branching vectors. The previous branching vector $(1, 1, 1, 2)$ can also be written as $(\overset{3}{\overrightarrow{1}}, 2)$. This notation is often used to show the difference between certain types of branches. For example, suppose we have a vertex x with $d(x) = 3$, and we have two types of branches:

- One branch that puts x in the deletion set.
- Three branches that each put one of the neighbors of x in the deletion set.

In this case, we would write this as $(1, \overset{3}{\overrightarrow{1}})$, to differentiate between the two types of branches that put one vertex in the deletion set.

2 Connected variants

We start this thesis by looking at two problems with a requirement that the graph, after removing the deletion set, consists of a single connected component. This is a strong requirement, which means these algorithms are faster on average, since more options do not meet the requirements and less options need to be considered.

2.1 Induced Cycle Deletion Set

We start off with the problem with the strongest requirements, the Induced Cycle Deletion Set problem.

Definition 1. Induced Cycle Deletion Set
Given: Graph $G = (V, E)$, integer k
Question: Does a set $S \subseteq V$ exist with $|S| \leq k$ such that the graph $G - S$ is 2-regular and consists of exactly one connected component? If so, give that set.

2.1.1 NP-Completeness

To prove the minimum Induced Cycle Deletion Set problem is $\mathcal{NP}$-complete we use a reduction from Hamiltonian Cycle.

Definition 2. *Hamiltonian cycle*
A cycle that contains all vertices of a given graph exactly once.

Definition 3. Hamiltonian Cycle
Given: Graph $G = (V, E)$
Question: Does a Hamiltonian cycle exist for G? If so, give that cycle.

The Hamiltonian Cycle problem was one of the first problems that was proven to be $\mathcal{NP}$-complete [Kar72].

Lemma 2.1. *Say we have a graph $G = (V, E)$ and a graph $G' = (V \cup V_E, E')$ which is the result of subdividing each edge of G, where V_E is the set of new vertices. G has a Hamiltonian cycle if and only if G' has an induced cycle deletion set of size at most $|E| - |V|$.*

Proof. First, suppose G has a Hamiltonian cycle $C \subseteq E$. By definition, $|C| = |V|$, hence there are $|E| - |V|$ edges not in the cycle. The set of vertices created by subdividing those $|E| - |V|$ edges is a solution to the

Induced Cycle Deletion Set problem on G', since removing those vertices from G' leaves a cycle.

Now suppose we have a valid induced cycle deletion set S for G' with $|S| \leq |E|-|V|$. The length of the remaining cycle in $G'-S$ is $|V|+|E|-|S| \geq |V|+|E|-(|E|-|V|) = 2 \cdot |V|$. We know in this cycle every edge is between a vertex in V and one in V_E, so exactly half of the vertices is in V. Therefore, if $|S| < |E| - |V|$, there are more than $2 \cdot |V|$ vertices in the cycle, which is a contradiction if half of them are in V. Thus, we know $|S| = |E| - |V|$ and there are exactly $2 \cdot |V|$ vertices in the cycle, including all vertices in V. If we now take the edges in E corresponding to the vertices in $V_E \cap S$, we have a Hamiltonian cycle in G. □

Lemma 2.2. *The Induced Cycle Deletion Set problem is in $\mathcal{NP}$.*

Proof. The deletion set S is a polynomial size certificate. Removing the vertices in S from a graph and checking whether the resulting graph is 2-regular and consists of a single connected component can be done in linear time. □

Since the transformation in Lemma 2.1 can trivially be done in time linear in $|E|$, the Lemmas 2.1 and 2.2 together prove that Induced Cycle Deletion Set is $\mathcal{NP}$-complete.

Theorem 2.3. *The Induced Cycle Deletion Set problem $\mathcal{NP}$-complete.*

2.1.2 Algorithm

A simple branching algorithm can solve Induced Cycle Deletion Set in $\mathcal{O}(2^k n^3)$ time. Suppose we have an instance $(G = (V, E), k)$. We work with an instance we reduce, which we call $(G' = (V', E'), k')$ and which initially is equal to the given instance. In the algorithm a connected component C is constructed, until it forms a cycle. All vertices adjacent to the component but not in it are removed from the graph. Whenever the neighborhood or degree of a vertex is discussed in this description, we mean the neighborhood or degree in G'.

1. While G' has a vertex v of degree 0 or 1, remove v and decrease k' by one.

2. Create a branch for every $v \in V'$. In the branch where v is chosen, we say that v is the starting vertex; v is put in C.

3. Create a branch for every neighbor x of v. In the branch where x is chosen, x is added to C. The vertices v and x form the beginning of the component C.

4. We now perform a number of checks to see whether we can directly decide this branch. If $|N(x) \cap N(v)| \geq 1$ and $k' \geq |V' \setminus C| - 1$, we add an arbitrary vertex from $N(x) \cap N(v)$ to C, and return yes with $S = V \setminus C$ as a valid solution set. Otherwise, every vertex w with $d(w) = 1$ and $w \in N(x)$ and all vertices in $N(x) \cap N(v)$ are removed from the graph, lowering k' by one for each removed vertex. If $k' < 0$, or if $d(x) = 1$, the branch fails and returns no.

5. If all the conditions from the previous step are false, we branch again. In each branch we select a vertex y from $N(x) \setminus C$, which we know contains at least one vertex since $d(x) > 1$ and x has exactly one vertex in C. We add y to C and remove all other vertices in $N(x) \setminus C$ from the graph, lowering k' accordingly. Finally, we rename y to x and go back to step 4, where y now fulfills the role of x.

Lemma 2.4. *If the above algorithm returns yes, the solution set it gives is valid. If it returns no, there is no valid solution set.*

Proof. If the algorithm returns yes, there was a branch where the first check in Step 4 succeeded and we get a solution set S that leaves the final C as the remaining graph when deleted. We know $k' \geq |V'| - |C|$, and since k' is lowered by one every time a vertex is removed from V' we know $k - k' = |V| - |V'|$, thus $k \geq |V| - |C| = |S|$. We also know C contains two adjacent vertices after Step 3, and every time Step 5 is applied it adds a vertex adjacent to the previous vertex added to C. Since the vertices adjacent to x that are not in C get removed in Step 5, C remains a path while the algorithm is running. When the algorithm ends the last vertex added to this path is one adjacent to both ends of the path, turning the path into a cycle. This means the solution set is valid.

Suppose the algorithm returns no, but there is a valid solution set S^* that leaves the cycle C as a remaining graph when deleted. Since every vertex in C has degree 2 there was a branch that started on one of those vertices v. There was also a branch that picked one of the vertices in $N(v) \cap C$ in Step 3, since all neighbors of v were chosen in one branch. From here there must be a vertex in C that was deleted in all branches of the algorithm in Step 4 or Step 5, since S^* was not found. When Step 4 removes a vertex it is either degree 1 and can therefore not be part of a cycle, or it closes the cycle when it is too small, which is not the case for C. Each vertex that is removed in

Step 5 is chosen in another branch of that same step, so no vertex is removed in all branches here. Hence none of the vertices of C can be removed in all branches, and S^* cannot exist without being found. □

2.1.3 Analysis

We will now prove a running time bound for the described algorithm.

Lemma 2.5. *The algorithm in Section 2.1.2 runs in $\mathcal{O}(2^k n^3)$ time.*

Proof. As usual with branching algorithms, we first look at the number of base cases. Step 1 and 4 do not branch. Step 2 and Step 3 both create n branches, resulting in $\mathcal{O}(n^2)$ branches after Step 3. Step 5 branches recursively. Note that x starts this step with exactly 1 neighbor in C, so Step 5 creates $|N(x)| - 1$ branches. Each branch lowers k' by $|N(x)| - 2$, which gives us the following recurrence:

$$T(k') = (|N(x)| - 1) \cdot \ T(k' - |N(x)| + 2)$$

The worst branching factor occurs when $|N(x)| = 3$:

$$T(k') = 2 \cdot \ T(k' - 1)$$

This gives a branching factor of 2. Since $k' \leq k$, this leaves us with $\mathcal{O}(2^k n^2)$ branches. If we look at the other actions, Step 1 takes $\mathcal{O}(n)$ time, and Step 2 and Step 3 both take $\mathcal{O}(1)$ time per branch. The checks in Step 4 can be done in $\mathcal{O}(1)$ time, but the removal of vertices takes $\mathcal{O}(n)$ time per branch. In Step 5 the removal again takes $\mathcal{O}(n)$ time. Since we have $\mathcal{O}(2^k n^2)$ branches and $\mathcal{O}(n)$ time is used per branch, the final running time of the algorithm is $\mathcal{O}(2^k n^3)$. □

2.2 Induced Path/Cycle Deletion Set

The other connected variant we will look at is the Induced Path/Cycle Deletion Set problem.

Definition 4. Induced Path/Cycle Deletion Set

Given: Graph $G = (V, E)$, integer k

Question: Does a set $S \subseteq V$ exist with $|S| \leq k$ such that the graph $G - S$ has maximum degree 2 and consists of exactly one connected component? If so, give that set.

2.2.1 NP-Completeness

The proof that Induced Path/Cycle Deletion Set is $\mathcal{NP}$-complete is similar to the corresponding proof for Induced Cycle Deletion Set , except Hamiltonian Path is used instead of Hamiltonian Cycle.

Definition 5. *Hamiltonian path*
A path that contains all vertices of a given graph exactly once.

Definition 6. HAMILTONIAN PATH
Given: Graph $G = (V, E)$
Question: Does a Hamiltonian path exist for G? If so, give that path.

Like finding a Hamiltonian cycle, Hamiltonian Path was one of the first problems that was proven to be $\mathcal{NP}$-complete [Kar72].

Lemma 2.6. *Say we have a graph $G = (V, E)$ and a graph $G' = (V \cup V_E, E')$ which is the result of subdividing each edge of G, where V_E is the set of new vertices. G has a Hamiltonian path if and only if G' has an induced path/cycle deletion set of size at most $|E| - |V| + 1$.*

Proof. First, suppose G has a Hamiltonian path P. If P is a cycle, we can use the exact same proof as we did in Lemma 2.1 since$|E|-|V|+1 > |E|-|V|$. If P is not a cycle, we know by definition that $|P| = |V| - 1$, and $|E| - |V| + 1$ edges are not part of the path. The vertices from V_E that are created by subdividing those edges that are not part of the path form a deletion set that is a valid solution to the Induced Path/Cycle Deletion Set problem on G', since removing those vertices leaves a single path.

For the other direction, suppose we have a valid induced path/cycle deletion set S for G' with $|S| \leq |E| - |V| + 1$. The number of vertices in the path $G' - S$ is $|V| + |E| - |S| \geq |V| + |E| - (|E| - |V| + 1) = 2 \cdot |V| - 1$.

If the path is a cycle, we know the number of vertices from V and V_E in it are equal, otherwise we would have at most $2 \cdot |V| - 2$ vertices in the cycle. In that case removing the edges corresponding to the vertices in $S \subseteq V_E$ from G leaves a Hamiltonian cycle, which is a Hamiltonian path.

If the path is not a cycle, we look at whether the endpoints are in V or V_E. If both endpoints are in V_E, there either are two vertices from V in S, or there is one vertex from V is S which is adjacent to both endpoints. In the former case there are at most $2 \cdot (|V| - 2) + 1 = 2 \cdot |V| - 3$ vertices in the path, which is not enough. In the latter case removing that vertex from S is also a valid solution (since any vertices it was adjacent to were either the two endpoints or were also in S) which leaves a cycle, which means there is a Hamiltonian cycle in G as before. If one endpoint is in V and one endpoints

is in V_E we can also remove the vertex in V from S, as all of its neighbors are either that one endpoint from V_E or are also in S. There were no other vertices from V in S, since if there were the path would be too small again. Therefore this path corresponds to a Hamiltonian path as well. Finally, if both endpoints are in V there are no vertices from V in S, since there would be at most $V-1$ vertices from V and $V-2$ vertices from V_E in the path, which is too small. Hence removing the edges corresponding to the vertices in $S \subseteq V_E$ from G leaves a Hamiltonian path. □

Lemma 2.7. *The Induced Path/Cycle Deletion Set problem is in $\mathcal{NP}$.*

Proof. The deletion set S is a polynomial size certificate. Removing the vertices in S from a graph and checking whether the resulting graph is has maximum degree 2 and consists of a single connected component can be done in linear time. □

The transformation in Lemma 2.6 can be completed in linear time in $|E|$. Therefore the Lemmas 2.6 and 2.7 prove that Induced Path/Cycle Deletion Set is $\mathcal{NP}$-complete.

Theorem 2.8. *The Induced Path/Cycle Deletion Set problem $\mathcal{NP}$-complete.*

2.2.2 Algorithm

This algorithm is based on the algorithm from Section 2.1.2. Only Step 1 and Step 4 are changed, but for reading clarity we repeat the other steps here as well.

1. If $k \geq n-1$, an arbitrary vertex is chosen and put in C. Then $S = V \setminus C$ is a valid solution and we return yes. Otherwise, remove all vertices of degree 0 from G' and decrease k' by one for each removed vertex.

2. Create a branch for every $v \in V'$. In the branch where v is chosen, we say that v is the starting vertex; v is put in C.

3. Create a branch for every neighbor x of v. In the branch where x is chosen, x is added to C. The vertices v and x form the beginning of the component C.

4. We perform a number of checks to see whether we can directly decide this branch. If $d(x) \geq 2$ and $k' \geq |V' \setminus C| - 1$, we add an arbitrary vertex from $N(x)$ to C, and return yes with $S = V \setminus C$ as a valid solution set. Otherwise, every vertex w with $d(w) = 1$ and $w \in N(x)$ and all vertices in $N(x) \cap N(v)$ are removed from the graph, lowering

k' by one for each removed vertex. If $k' < 0$, or if $d(x) = 1$, the branch fails and returns no.

5. If all the conditions from the previous step are false, we branch again. In each branch we select a vertex y from $N(x) \setminus C$, which we know contains at least one vertex since $d(x) > 1$ and x has exactly one vertex in C. We add y to C and remove all other vertices in $N(x) \setminus C$ from the graph, lowering k' accordingly. Finally, we rename y to x and go back to step 4, where y now fulfills the role of x.

Lemma 2.9. *If the above algorithm returns yes, the solution set it gives is valid. If it returns no, there is no valid solution set.*

Proof. If the algorithm returns yes, there was a branch where either the check in Step 1 or the first check in Step 4 succeeded. In the former case, any isolated vertex is a valid remaining graph. In the latter case, suppose solution set S is returned that leaves the final C as the remaining graph when deleted. We know $k' \geq |V'| - |C|$, and since k' is lowered by one every time a vertex is removed from V' we know $k - k' = |V| - |V'|$, thus $k \geq |V| - |C| = |S|$. We also know C contains two adjacent vertices after Step 3, and every time Step 5 is applied it adds a vertex adjacent to the previous vertex added to C. Since the vertices adjacent to x that are not in C get removed in Step 5, C remains a path while the algorithm is running. The final vertex added in Step 4 is also adjacent to the previous vertex added to C, so C is still a path.

Suppose the algorithm returns no, but there is a valid solution set S^* that leaves the path or cycle C as a remaining graph when deleted. If C is a path, there was a branch that started on one of the endpoints v of C. If C is a cycle, there was a branch that started on one oft he vertices v of C. There was also a branch that picked one of the vertices in $N(v) \cap C$ in Step 3, since all neighbors of v were chosen in one branch. From here there must be a vertex in C that was deleted in all branches of the algorithm in Step 4 or Step 5, since S^* was not found. When Step 4 removes a vertex, adding that vertex to C would end this branch by either adding a degree 1 vertex or a vertex adjacent to v and thereby closing the cycle. Since the first check in Step 4 failed, we know the component is still too small to end this branch successfully, hence removing these vertices will not lead to C. Each vertex that is removed in Step 5 is chosen in another branch of that same step, so no vertex is removed in all branches here. Hence none of the vertices of C can be removed in all branches, and S^* cannot exist without being found. □

2.2.3 Analysis

In this section we will prove the running time of the algorithm from the previous section.

Lemma 2.10. *The algorithm in Section 2.2.2 runs in $\mathcal{O}(2^k n^3)$ time.*

Proof. First we look at the number of branches created. Since Step 1 and 4 do not branch, we know the number of branches is the same as in the proof of Lemma 2.5; a total of $\mathcal{O}(2^k n^2)$ branches. The changes to Step 1 and Step 4 do not alter their running times, so they still take $\mathcal{O}(1)$ and $\mathcal{O}(n)$ time respectively. This means the algorithm still runs in $\mathcal{O}(2^k n^3)$ time. □

3 Co-Cycle Packing

In the following problem we will look at, the requirement that the graph should be a single connected component after deleting the vertices is removed.

Definition 7. Co-Cycle Packing

Given: Graph $G = (V, E)$, integer k

Question: Does a set $S \subseteq V$ exist with $|S| \leq k$ such that the graph $G - S$ is 2 regular? If so, give that set.

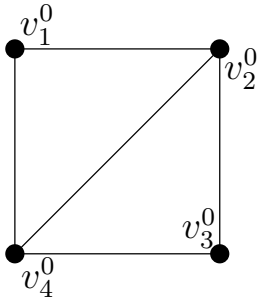
3.1 NP-Completeness

We can prove that the Co-Cycle Packing problem is $\mathcal{NP}$-hard by using a reduction from Vertex Cover, a famous $\mathcal{NP}$-hard problem [Kar72].

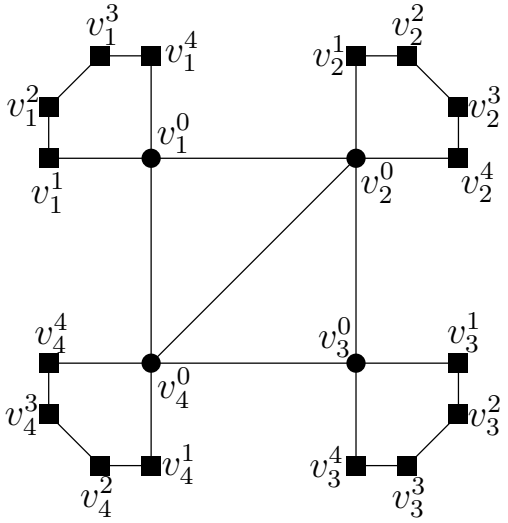
Definition 8. Minimum Vertex Cover

Given: Graph $G = (V, E)$, integer k

Question: Does a set $S \subseteq V$ exist with $|S| \leq k$ such that each edge in E is incident to at least one vertex in S. If so, give that set.



(a) *G, the input graph*



(b) *G', the result of the transformation*

Figure 1: *An example of the transformation in Lemma 3.1. The vertices represented as a square are added by the transformation.*

Lemma 3.1. *Say we have an graph $G = (V, E)$ and $|V| = n$. Let $V = \{v_1^0, v_2^0, \ldots, v_n^0\}$. We create n new vertex sets $V_i = \{v_i^0, v_i^1, v_i^2, \ldots, v_i^n\}$ with $1 \leq i \leq n$, consisting of one vertex from V and n new vertices each. We create n edge sets $E_i = \{\{v_i^0, v_i^1\}, \{v_i^1, v_i^2\}, \ldots, \{v_i^{n-1}, v_i^n\}, \{v_i^n, v_i^0\}\}$ with $1 \leq i \leq n$, such that each set V_i forms a cycle. Let $V' = \bigcup_{i=1}^{n} V_i$ and $E' = E \cup (\bigcup_{i=1}^{n} E_i)$. An example of this transformation can be seen in Figure 1. G has a vertex cover of size at most k if and only if G' has a co-cycle packing of size at most $k \cdot (n+1)$.*

Proof. First suppose there is a vertex cover S of G with $|S| \leq k$. We claim that the set $S' = \{v_i^j | 1 \leq i \leq n, 0 \leq j \leq n, v_i^0 \in S\}$ is a co-cycle packing. If v_i^0 is not in S', it is not in S which means $N_G(v) \subseteq S \subseteq S'$ because S is a vertex cover. Since v_i^0 has exactly two neighbors in V_i, it has exactly degree 2 in $G' - S'$. If v_i^j with $j > 0$ is not in S', both neighbors of v_i^j are also not in S' because V_i is either entirely in S' or entirely disjunct from it. Therefore the subgraph $G' - S'$ is 2-regular and S' is a valid co-cycle packing. For every vertex in S, there are exactly $n + 1$ vertices in S', so $|S'| = |S| \cdot (n+1) \leq k \cdot (n+1)$.

Now suppose we have the minimum size co-cycle packing S' of G' with $|S'| \leq k \cdot (n+1)$. We claim $S = S' \cap V$ is a vertex cover of G. Each cycle in $G' - S'$ is either an entire set V_i, or consists entirely of vertices in V. Suppose we have a vertex v_i^0 that is not in S', but the rest of V_i is in S'. In $G' - S'$ v_i^0 is part of cycle C. If we take $S^* = (S' \cup C) \setminus V_i$ we get a co-cycle packing of the same size or smaller than S'. Therefore we can assume for each V_i it is either entirely in, or entirely disjunct from S', which we know to be minimum size. Because a vertex v_i^0 is never adjacent to another vertex from V in $G' - S'$, $S = S' \cap V$ is a vertex cover of G, of size $|S| = \dfrac{|S'|}{n+1} \leq \dfrac{k \cdot (n+1)}{n+1} = k$. □

Theorem 3.2. *Co-Cycle Packing is $\mathcal{NP}$-complete.*

Proof. The construction of G' as in Lemma 3.1 can be done in quadratic time. This shows Co-Cycle Packing to be $\mathcal{NP}$-hard, which means all that is left to do is prove Co-Cycle Packing is in $\mathcal{NP}$. The deletion set is a polynomial size certificate. Removing these vertices from the graph and checking all remaining vertices to have degree 2 can trivially be done in linear time. □

3.2 Algorithm

In this section a branching algorithm is detailed for the Co-Cycle Packing problem. The reduction and branching rules are exhaustively applied in the

described order, such that when a rule is applied, no previous rule can be applied. In this section, a deletion set D is build up. When a vertex is put in D it is effectively removed from the graph. When the degree or neighborhood of a vertex is discussed, neighbors in D are not considered. Additionally, a weight function $w(v)$ is introduced, with $w(v) = 1$ for all $v \in V$ at the start of the algorithm. When a vertex is put in D, the weight of that vertex is subtracted from k. This is used for one of the Reduction Rules, which combines multiple vertices into a single vertex. We also talk about a residue set R, which contains the vertices that are definitively not in D.

3.2.1 Reduction Rules

We start by applying a number of simple Reduction Rules. We will prove that the instance before applying the rule (G, k) is a yes-instance if and only if the instance after applying the rule (G', k') is a yes-instance. If this is the case, we call the rule *safe*.

Reduction Rule 3.1. *If a vertex v has degree 0, add v to D.*

Reduction Rule 3.2. *If a vertex v has degree 1, add v to D.*

Reduction Rule 3.3. *If two vertices u and v have $N(u) = \{v, p\}$, $N(v) = \{u, q\}$ and $p \neq q$, contract u and v into a single vertex x. Set $N(x) = \{p, q\}$ and $w(x) = w(u) + w(v)$.*

Reduction Rule 3.4. *If a vertex $v \in R$ has $N(v) = \{u, w\}$ put u and w in R.*

Reduction Rule 3.5. *If a vertex $v \in R$ has degree 3 or more, and $|N(v) \cap R| = 2$, add all vertices in $N(v) \setminus R$ to D.*

Lemma 3.3. *Reduction Rules 3.1, 3.2, 3.4, and 3.5 are safe.*

Proof. Safeness of Reduction Rules 3.1 and 3.1 is trivial, since all vertices in R will have to be part of a cycle, and vertices of degree 0 or 1 can never be part of a cycle.

For Reduction Rule 3.3 consider the case when one of the vertices, say u, is added to D. Now v is degree 1 and also needs to be put in D. Conversely, if u is added to R, v also needs to be added to R. Therefore u and v will always end up in the same set and we can contract them into a single vertex that combines their weights.

Rule 3.4 is safe because u and w cannot be put in D, since then v would have degree lower than 2 while v is already in R.

The safeness of Rule 3.5 follows from the fact that adding any of the vertices in $N(v) \setminus R$ to R would give v, which is already in R, degree 3 in $G[R]$. □

Finally, we have a trivial rule that ends the branch.

Reduction Rule 3.6. *If $k \leq 0$ and the graph is not 2-regular, or if a vertex exists that is both in D and in R, end this branch and return no.*

3.2.2 Degree 4 or more vertices

Case 3.1 In this branching step we deal with vertices of degree 4 or more. We pick vertex v with the highest degree in the graph, of degree at least 4. We have two types of branches:

- One branch where we put v in D.
- $\binom{d(v)}{2}$ branches where we put v and two vertices from $N(v)$ in R, and all other vertices from $N(v)$ in D.

3.2.3 Degree 2 vertices

Case 3.2 If there is a vertex with exactly degree 2 in the graph, say vertex v, we decide what to do with v. Let the neighbors of v be u and w. There are a number of subcases to consider, depending on the number of vertices adjacent to both u and w. We know u and w have degree 3, if their degree was higher Case 3.1 would have handled it, if it was degree 1 or 2 Reduction Rules 3.2 or 3.3 would have applied, respectively.

Regardless of the subcase, there is a branch where v is put in D. Since this is identical for all subcases, we do not repeat this for each individual case. Instead, in each subcase we look at what happens when u, v, and w are put in R.

Case 3.2.1 $N(u) = \{v, p, q\}, N(w) = \{v, r, s\}$
In this case, we put one of $\{p, q\}$ and one of $\{r, s\}$ in D, and put the two unchosen vertices in R.

Case 3.2.2 $N(u) = \{v, p, w\}, N(w) = \{v, r, u\}$
If v is in R, we need to put p and r in D.

Case 3.2.3 $N(u) = \{v, p, q\}, N(w) = \{v, r, q\}$
We have two new types of branches:

- One branch where we put q in D, and p and r in R.
- One branch where we put p, r, and possibly (if q is degree 3) the third neighbor of q in D, adding only q to R.

Case 3.2.4 $N(u) = \{v, w, q\}, N(w) = \{v, u, q\}$
If v is in R, we always put q in D.

Case 3.2.5 $N(u) = \{v, p, q\}, N(w) = \{v, p, q\}$
If p and q are adjacent, or if q has degree 2, we do not need to branch because $p \in D$ and $\{q, u, v, w\} \subseteq R$ is always an optimal solution. Otherwise, suppose $N(p) = \{u, w, r\}$ and $N(q) = \{u, w, s\}$. We have two new types of branches:

- One branch where we put q and r in D and p in R.
- One branch where we put p and s in D and q in R.

3.2.4 Cycles of length 3

After the above branching rules, we are left with a 3-regular graph. The next step is dealing with all length 3 cycles. Figures depicting each of the cases in this section can be found in Figures 2 and 3. In this section the vertices in the cycle we found are a, b, and c.

Case 3.3.1 $N(a) = \{b, c, u\}, N(b) = \{a, c, v\}, N(c) = \{a, b, w\}$
In this most basic case, shown in Figure 2(a), a, b, and c are all adjacent to a different vertex. We have three types of branches:

- One branch where we put a, b, c in D.
- One branch where we put u, v, w in D and a, b, c in R.
- Three branches where we put one of $\{a, b, c\}$ in D, and put the two vertices that were not chosen in R.

(a) *Case 3.3.1* ***(b)*** *Case 3.3.2*

Figure 2: *The vertex shapes have different meanings. Circle: The entire neighborhood of the vertex is shown. Square: The vertex has exactly two neighbors that are not shown.*

Case 3.3.2 $N(a) = \{b, c, v\}, N(b) = \{a, c, v\}, N(c) = \{a, b, w\}$

This case is shown in Figure 2(b). Say $N(v) = \{a, b, r\}$. The different branches are:

- Put $a, b, c, v \in D$.
- Put $v, w \in D$, $a, b, c \in R$.
- Put $c, r \in D$, $a, b, v \in R$.
- Put $b \in D$, $a, c, r, v, w \in R$.

For the last option, we can also swap a and b, but since the vertices that are adjacent to the rest of the graph, r and w are still in R, this would be equivalent.

Case 3.3.3 $N(a) = \{b, c, v\}, N(b) = \{a, c, v\}, N(c) = \{a, b, v\}$

We know v is degree 3, therefore we have a fully connected component of 4 vertices, as shown in Figure 3(a). We do not need to branch, as putting v in D and the other three vertices in R gives an optimal solution.

Case 3.3.4 $N(a) = \{b, c, v\}, N(b) = \{a, c, v\}, N(c) = \{a, b, w\}, N(v) = \{a, b, w\}$

This case is shown in Figure 3(b). Since w is the only vertex that is adjacent to a vertex we do not know about, we can choose to put it in D or in R:

- One branch where we put v and w in D, and a, b, and c in R.
- One branch where we put b and the third neighbor of w in D, and a, c, v, and w in R.

(a) *Case 3.3.3* ***(b)*** *Case 3.3.4*

Figure 3: *The vertex shapes have different meanings. Circle: The entire neighborhood of the vertex is shown. Square: The vertex has exactly two neighbors that are not shown.*

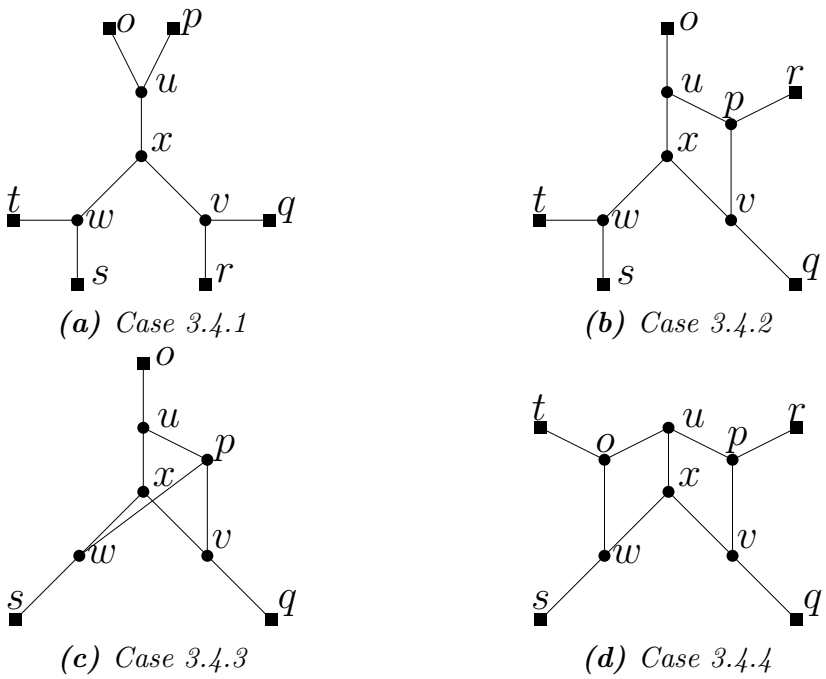
3.2.5 Degree 3 vertices

The final branching rule has eight cases in which it handles all remaining degree 3 vertices. Say the degree 3 vertex we have is x with $N(x) = \{u, v, w\}$. The cases depend on the number of neighbors and edges u, v, and w have in common.

Case 3.4.1 $N(u) = \{x, o, p\}, N(v) = \{x, q, r\}, N(w) = \{x, s, t\}$

In the most straightforward case, depicted in Figure 4(a), u, v, and w have no neighbors or edges in common, except for the vertex x. We have a four types of branches:

- One branch where we put x in D.
- Four branches where we put u, one of $\{q, r\}$, and one of $\{s, t\}$ in D. We also put v, w, and the two unchosen vertices of $\{q, r, s, t\}$ in R.
- Four branches where we put v, one of $\{o, p\}$, and one of $\{s, t\}$ in D. We also put u, w, and the two unchosen vertices of $\{o, p, s, t\}$ in R.
- Four branches where we put w, one of $\{o, p\}$, and one of $\{q, r\}$ in D. We also put u, v, and the two unchosen vertices of $\{o, p, q, r\}$ in R.



(a) *Case 3.4.1* ***(b)*** *Case 3.4.2*

(c) *Case 3.4.3* ***(d)*** *Case 3.4.4*

Figure 4: *The vertex shapes have different meanings. Circle: The entire neighborhood of the vertex is shown. Square: The vertex has exactly two neighbors that are not shown.*

Case 3.4.2 $N(u) = \{x, o, p\}, N(v) = \{x, p, q\}, N(w) = \{x, s, t\}$

We handle this case, shown in Figure 4(b), in much the same way as the previous one, except for the branches where u and v were both put in R. We now have five types of branches:

- One branch where we put x in D.
- Four branches where we put u, one of $\{p, q\}$, and one of $\{s, t\}$ in D. We also put v, w, and the two unchosen vertices of $\{p, q, s, t\}$ in R.
- Four branches where we put v, one of $\{o, p\}$, and one of $\{s, t\}$ in D. We also put u, w, and the two unchosen vertices of $\{o, p, s, t\}$ in R.
- One branch where p and w are put in D, and o and q are put in R.
- One branch where we put $\{o, q, r, w\}$ in D and p in R.

Case 3.4.3 $N(u) = \{x, o, p\}, N(v) = \{x, p, q\}, N(w) = \{x, p, s\}$
In this case, shown in Figure 4(c), the vertices p and x are symmetrical, since their neighborhoods are identical. There are four branches..

- We put p in D.
- We put u, q, and s in D.
- We put v, o, and s in D.
- We put w, o, and q in D.

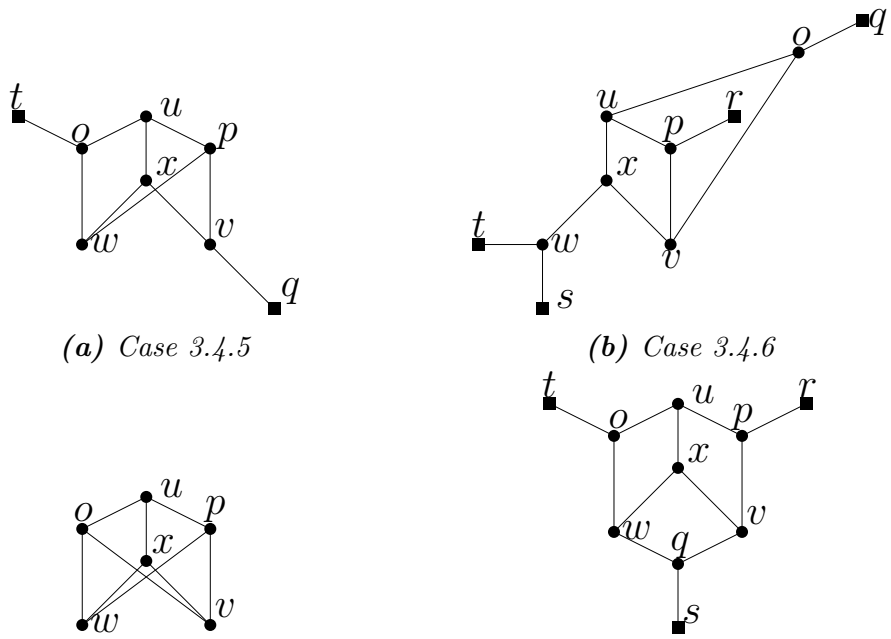
Case 3.4.4 $N(u) = \{x, o, p\}, N(v) = \{x, p, q\}, N(w) = \{x, o, s\}$
This case, depicted in Figure 4(d), has a total of twelve branches, which we describe in Table 1.

D	R
p, v, s, t	o, u, w, x
o, q, r, w	p, u, v, w
p, u, v, x	o, s, t, w
o, u, w, x	p, q, r, v
u, x	o, p, q, r, s, t, v, w
o, p, u, v, w, x	–
v, w, x	o, p, r, t, u
q, s, u	o, p, r, t, v, w, x
o, p, u	q, s, v, w, x
r, t, x	o, p, q, s, u, v, w
p, w	o, q, t, u, v, x
o, v	p, q, t, u, w, x

Table 1: *The twelve branches of Case 3.4.4.*

Case 3.4.5 $N(u) = \{x, o, p\}, N(v) = \{x, p, q\}, N(w) = \{x, o, p\}$
This case can be seen in Figure 5(a). Here we have four branches.

- One where $\{p, t, v\}$ are put in D.
- One where $\{o, q, w\}$ are put in D.
- One where p and w are put in D, and $\{o, q, t, u, v, x\}$ are put in R.
- One where o and v are put in D.



(a) *Case 3.4.5* ***(b)*** *Case 3.4.6*

(c) *Case 3.4.7* ***(d)*** *Case 3.4.8*

Figure 5: *The vertex shapes have different meanings. Circle: The entire neighborhood of the vertex is shown. Square: The vertex has exactly two neighbors that are not shown.*

The vertices that get put in R are only listed for the third branch, since for the other branches the vertices that could get put in R form a complete cycle, in which case it does not matter.

Case 3.4.6 $N(u) = \{x, o, p\}, N(v) = \{x, o, p\}, N(w) = \{x, s, t\}$

This case is shown in Figure 5(b). We have five types of branches that create a total of twelve branches.

- Four branches where u, one of $\{o, p\}$, and one of $\{s, t\}$ are added to D.
- Two branches where u, x, and possibly w are added to D.
- Two branches where w, and either $\{p, q\}$ or $\{o, r\}$ are added to D.
- Two branches where q, r, x, and possibly w are added to D.
- Two branches where o, p, u, v, x, and possibly w are added to D.

Case 3.4.7 $N(u) = \{x, o, p\}, N(v) = \{x, o, p\}, N(w) = \{x, o, p\}$
In this case, shown in Figure 5(c), o and p are degree 3, we know their entire neighborhoods and the entire connected component. We do not need to branch, and put p and v in D.

Case 3.4.8 $N(u) = \{x, o, p\}, N(v) = \{x, p, q\}, N(w) = \{x, o, q\}$
This case is shown in Figure 5(d), and has four types of branches.

- One branch where we put $\{r, s, t, x\}$ in D.
- Three branches where we put either $\{p, q, t, v\}$, or $\{o, q, r, w\}$, or $\{o, p, s, u\}$ in D.
- Three branches where we put either $\{q, u\}$, or $\{o, v\}$, or $\{p, w\}$ in D.
- One branch where we put $\{o, p, q, u, v, w, x\}$ in D.

3.3 Analysis

In this section we will analyze the time complexity of the algorithm from the previous section, case by case.

Case 3.1 deals with vertices of degree 4 or more, and has two types of branches. The first one is only one branch, that puts one vertex in D. The other type of branch creates $\binom{d(v)}{2}$ branches that each put $d(v) - 2$ vertices in D. We can say this has the branching vector $(1, \overrightarrow{d(v) - 2}^{\binom{d(v)}{2}})$. The worst branching vector occurs when $d(v) = 4$: $(1, \overrightarrow{2}^{6})$.

Case 3.2 decides what to do with vertices of exactly degree 2. It has five subcases, for which the branching vectors follow from the descriptions in a straightforward manner. Recall that the descriptions of the subcases leave out the branch where v is put in D, which is identical to all subcases. The branching vectors of these Cases is shown in Table 2. Note that for Case 3.2.3 it is assumed q is degree 2; if it is degree 3 the vector is strictly faster. The slowest of these cases is Case 3.2.1.

Case #	Branching Vector
3.2.1	$(1, \overrightarrow{2}^{4})$
3.2.2	$(1, 2)$
3.2.3	$(1, 1, 2)$
3.2.4	$(1, 1)$
3.2.5	$(1, 2, 2)$

Table 2

Case 3.3 deals with cycles of length 3 within the 3-regular graph. Such cycles can be detected in cubic time. There are four subcases, of which only the first has a complex analysis. With a simple analysis, Case 3.3.1 would have branching vector $(3, 3, \overrightarrow{1}^{3})$. We can improve this by looking at what happens after this branch, specifically in the three branches where a, b, or c were put in D. In the branch where a is put in D, b and c are now adjacent degree 2 vertices. They are contracted into a single vertex d by Reduction Rule 3.3. We know v and w, the two neighbors of d, are both degree 3, and since there are no degree 4 or more vertices in the graph, Case 2 will be the next branching rule to apply. Therefore we can include the branching vector of that case in the branching vector of this case. The slowest subcase of 3.2 was Case 3.2.1, with vector $(1, \overrightarrow{2}^{4})$. Combining this into our previous branching vector gives $(3, 3, \overrightarrow{2}^{3}, \overrightarrow{3}^{12})$.

The other cases, Cases 3.3.2 through 3.3.4, can be analyzed in a more simple manner. Case 3.3.2 has branching vector $(4, 2, 2, 1)$, and Case 3.3.4 has branching vector $(2, 2)$. Case 3.3.3 does not branch, and can be solved in polynomial time.

The subcases of Case 3.4 have a lot of branches, but are straightforward to analyze. The branching vectors of the Cases 3.4.1 through 3.4.8 are shown in Table 3, except for Case 3.4.7. That case does not branch, and can be solved in polynomial time. Of these cases, 3.4.1 is the slowest.

Case #	Branching Vector
3.4.1	$(1, \overrightarrow{3}^{4}, \overrightarrow{3}^{4}, \overrightarrow{3}^{4})$
3.4.2	$(1, \overrightarrow{3}^{4}, \overrightarrow{3}^{4}, 2, 4)$
3.4.3	$(1, 3, 3, 3)$
3.4.4	$(\overrightarrow{2}^{3}, \overrightarrow{3}^{4}, \overrightarrow{4}^{4}, 6)$
3.4.5	$(3, 3, 2, 2)$
3.4.6	$(\overrightarrow{3}^{4}, 2, 3, \overrightarrow{3}^{2}, 3, 4, 5, 6)$
3.4.8	$(4, \overrightarrow{4}^{3}, \overrightarrow{2}^{3}, 7)$

Table 3

If we calculate the branching factors for the vectors in this analysis, we see Case 3.1 is the slowest part of this algorithm, when dealing with degree 4 vertices. The branching vector $(1, \overrightarrow{2}^{4})$ gives a branching factor of 3, which means the time complexity of the described algorithm is $\mathcal{O}^*(3^k)$.

4 Co-Path/Cycle Packing

The final problem we look at in this thesis is the Co-Path/Cycle Packing problem.

Definition 9. Co-Path/Cycle Packing
Given: Graph $G = (V, E)$, integer k
Question: Does a set $S \subseteq V$ exist with $|S| \leq k$ such that the graph $G - S$ has maximum degree 2? If so, give that set.

4.1 NP-Completeness

For Co-Path/Cycle Packing we will not write a full $\mathcal{NP}$-completeness proof here, since it falls in the class of *non-trivial* and *hereditary* node-deletion problems. Non-trivial means there is an infinite number of yes-instances and an infinite number of no-instances, and hereditary means that if the required property (maximum degree 2) holds for some graph, it also holds for all vertex-induced subgraphs of that graph. The previously discussed problems have non-hereditary properties, since the induced subgraphs might have multiple connected components, or leave vertices with degree 0 or 1. However, the Co-Path/Cycle Packing problem does belong to this class of problems, which all have been proven to be $\mathcal{NP}$-complete in a previous paper [LY80].

Theorem 4.1. *The Co-Path/Cycle Packing problem is $\mathcal{NP}$-complete.*

4.2 Algorithm

In this section the algorithm used for Co-Path/Cycle Packing will be detailed, which will be analyzed in the next section. The various steps of the algorithm are applied in the order they are described in here; for instance, after the branching rule that deals with vertices of degree 5 or more, we can assume the maximum degree of the graph is degree 4. In the algorithm, two sets are build, D and R. They are the deletion and residue sets respectively, and at the end of the algorithm every vertex is in exactly one of these sets in such a way that $G[R] = G - D$ has maximum degree 2. After a vertex is put in D, k is decreased by one and the vertex is ignored and does not influence the degree of it's neighbors. For example, if vertex v has degree 1 and it's neighbor gets put in D, v will have degree 0 afterwards. When a vertex is put in R, it always has maximum degree 2.

4.2.1 Reduction Rules

The algorithm starts with a number of reduction rules. We will prove these rules are safe. We call vertices with degree 3 or more *illegal*.

Reduction Rule 4.1. *If a vertex v has degree 0, add v to R.*

Reduction Rule 4.2. *If a vertex v has degree 1, add v to R.*

Reduction Rule 4.3. *If two vertices u and v are adjacent and both are degree 2, add both u and v to R.*

Reduction Rule 4.4. *If a vertex v has degree 2 and one of its neighbors is in R, add v to R.*

Reduction Rule 4.5. *If a vertex u has $d(u) \geq 2$ and a vertex v has $d(v) \leq 2$ such that $(N(v) \setminus \{u\}) \subseteq N(u)$ and $\forall w \in N(v).\, d(w) \leq 3$, then v is put in R.*

Lemma 4.2. *Reduction Rules 4.1, 4.2, 4.3, 4.4, and 4.5 are safe.*

Proof. Reduction Rule 4.1 is safe because v is not connected to any illegal vertices, therefore putting it in D never reduces the degree of such a vertex. Reduction Rule 4.2 is safe since putting v in D reduced the degree of at most one illegal vertex, the neighbor of v, say u. Therefore if we have a valid deletion set S with $v \in S$, we can create a new deletion set $S' = (S \setminus \{v\}) \cup \{u\}$ with $|S'| \leq |S|$.

Reduction Rule 4.4 is safe for a similar reason. Say $N(v) = \{u, w\}$ with $u \in R$. We know u is at most degree 2 since it is in R, therefore replacing v with w in any valid deletion set does not affect the validity of the set and does not increase its size.

For Reduction Rule 4.3, say $N(v) = \{u, w\}$ and $N(w) = \{v, x\}$. Since deleting v reduces the degree of at most one illegal vertex u, deleting u instead is safe, and the same holds for replacing w with x. Therefore a valid deletion set S can be used to create $S' = (S \setminus \{v, w\}) \cup \{u, x\}$ a new deletion set with $|S'| \leq |S|$.

Reduction Rule 4.5 has two options. If a solution set S contains both u and v, the set $S' = (S \setminus \{v\})$ would be a smaller set which would also be a valid solution, since the vertices in $N[v]$ that are not in S are at most degree 2 (because u is in S). If a solution S contains v but not u, the set $S' = (S \setminus \{v\}) \cup \{u\}$ would be a valid solution because the degree of the vertices in $N(v)$ is not increased and v itself has at most degree 2. □

Reduction Rule 4.6. *If a vertex v has three or more neighbors in R, put v in D.*

Reduction Rule 4.7. *If a vertex v has $d(v) \geq k+2$, put v in D.*

Lemma 4.3. *Reduction Rules 4.6 and 4.7 are safe.*

Proof. Reduction Rule 4.6 is clearly safe, since putting v in R would give $G[R]$ a degree 3 (or more) vertex, meaning v will always have to end up in D. Reduction Rule 4.7 is safe because we can put at most k neighbors of v in D, after which it would still be at least degree 3. Therefore any valid deletion set will contain v. □

Finally, there is a trivial rule to end the current branch.

Reduction Rule 4.8. *If $k \leq 0$ and there is a vertex v with $d(v) > 2$, end this branch and return no.*

4.2.2 Degree 5 or more vertices

Case 4.1 In the first branching step we deal with all vertices with degree 5 or more. Say we have vertex v of degree $d(v) \geq 5$. We have four types of branches.

- One branch where v is put in D.
- $\binom{d(v)}{2}$ branches where all but two neighbors of v are put in D.
- $d(v)$ branches where all but one neighbor of v are put in D.
- One branch where all neighbors of v are put in D.

4.2.3 Degree 4 vertices

After the above step the maximum degree of the graph is 4, and in this step we will deal with all degree 4 vertices. In this section, we have a degree 4 vertex v and one of its neighbors u. We will look a number of different cases for u, and provide a branching rule for each. We look at these cases in the order described here.

A neighbor in R

Case 4.2 If u is in R, we either put v in D, or put two of the three other neighbors in D. The latter type of branch can be done in three ways. Note that, if v has more than one neighbor in R some of these branches can be ignored.

A degree 4 neighbor
In the case where u is degree 4, we have four different subcases depending on the number of common neighbors between u and v.

Case 4.3.1 If u and v have no common neighbors, the branches are very straightforward. For each branch, we look at whether both, one or none of u and v will be put in D, and then which vertices should additionally be added.

- One branch where both u and v are put in D.
- Six branches where one of u and v, and one neighbor of the other are put in D.
- Nine branches where neither u and v are in D. In this case two neighbors of both vertices should be put in D.

Since we know u and v are degree 4, they each have 3 neighbors beside each other, which gives us six and nine branches for the latter two types of branch.

Case 4.3.2 If u and v share one common neighbor, say o, the branches where at least one of u and v is put in D are the same, so we will not repeat them. There are two other types of branches, depending on whether o is put in D.

- Four branches where o is put in D, together with one other neighbor of both u and v.
- One branch where o is not put in D, but both other neighbors of u and both other neighbors of v are put in D.

Case 4.3.3 When u and v share two common neighbors, say o and p, the branches where at least one of u and v is put in D are the same once again. The two new types of branches originate from putting one or both of o and p in D; putting neither D means u and v will not become degree 2 or lower.

- One branch where o and p are put in D.
- Two branches where either o or p is put in D, together with the other neighbor of u and that of v.

Case 4.3.4 Finally, when u and v share three common neighbors (o, p, and q), we say they *dominate* each other, which means $N[u] \subseteq N[v]$ and vice versa. This means putting u in D but not v, is equivalent to putting v in D but not u. Therefore the branches where u is put in D but v is not can be ignored, reducing the total number of branches.

- One branch where both u and v are put in D.
- Three branches where v, and one of $\{o, p, q\}$ are put in D.
- Three branches where two of $\{o, p, q\}$ are put in D.
- One branch where o, p, and q are put in D.

A degree 3 neighbor

If u is degree 3, we have three subcases depending on how many neighbors u and v have in common, similar to when u is degree 4.

Case 4.4.1 When u and v have no neighbors in common, we have three types of branches.

- One branch where v is put in D.
- Three branches where u and another neighbor of v are put in D.
- Six branches where one neighbor of u and two neighbors of v are put in D.

Case 4.4.2 If u and v share one neighbor o, the first two types of branches, where u and v are put in D, also appear so we will not repeat them. In addition, we have two new types of brances depending on whether o is put in D.

- Two branches where o and another neighbor of v are put in D.
- One branch where the two other neighbors of v and the other neighbor of u are put in D.

Case 4.4.3 When u and v have two neighbors in common (o and p), the branches change a bit, since v now dominates u. Therefore the branches where u is put in D (and v is not) can be ignored since they will never lead to the optimal solution. This leaves us with only two branches.

- One branch where v is put in D.
- One branch where both o and p are put in D.

A degree 2 neighbor
Case 4.5 In the case where u is degree 2, we use a very simple branching rule.

- One branch where v is put in D.
- Six branches where two neighbors of v are put in D.

The fact that u is degree 2 is only relevant in the analysis, because if u and v are not put in D, Reduction Rule 4.3 applies.

4.2.4 Degree 3 vertices

Now that all vertices with degree higher than 3 have been handled, we will take a look at the degree 3 vertices in the graph. This will be done by taking a degree 3 vertex and looking at the neighborhood of that vertex.

Two neighbors in R
Case 4.6 First we handle degree 3 vertices with 2 neighbors in R. Say we have x with $N(x) = \{u, v, w\}$ and $u, v \in R$. In this case, either w or x has to be in D, and we branch on either of these options.

One neighbor in R
Say there is a degree 3 vertex x with 1 neighbor in R, so $N(x) = \{u, v, w\}$ and $u \in R$. Of v (and w) we know it is either degree 2 or degree 3, and if it is degree 2 its other neighbor is degree 3. If any of the degrees are lower, v would be in R because of Reduction Rules 4.2 and 4.3.

Case 4.7.1 If at least one of v and w is degree 2, we can branch on putting either of them in D. We do not need a branch where we put x in D, since at most one of its neighbors is degree 3, and therefore any deletion set containing x can replace it with its degree 3 neighbor (or a degree 2 neighbor if there is no degree 3 neighbor) and it would still be a valid deletion set of the same size.

Case 4.7.2 If both v and w are degree 3 and they are neighbors, the same can be done since removing one makes the other degree 2 (and x never needs to be removed for the same reasons as above). Finally, if they are not neighbors we need a more complex branching rule. In this case there are a total of six branches:

- One branch where x is put in D.
- One branch where v and w are put in D.
- Two branches where v and either of the two vertices in $N(w) \setminus \{x\}$ are put in D.
- Two branches where w and either of the two vertices in $N(v) \setminus \{x\}$ are put in D.

Length 3 cycles

Case 4.8.1 After the previous cases we know there are no degree 3 vertices with a neighbor in R remaining. We look for a vertex v with at least two degree 3 neighbors u and w that share an edge between them. If u, v, and w do not have other neighbors in common, i.e. when $|N(\{u, v, w\})| = 3$, we branch on putting either of the three vertices u, v, and w in D, or putting all three outer vertices in D, resulting in 4 branches.

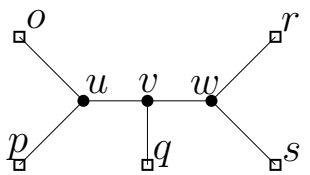
Case 4.8.2 If two of the vertices do share a neighbor, say $N[u] = N[v] = \{o, u, v, w\}$ we know o is degree 3. If it was degree 2, Reduction Rule 4.5 would have applied. Say $N(o) = \{p, u, v\}$ and $N(w) = \{q, u, v\}$. We create three branches:

- One branch where u is put in D.
- One branch where o and q are put in D.
- One branch where w and p are put in D.

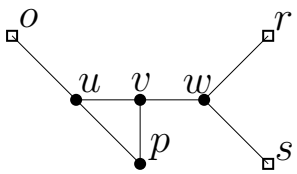
In the case where o is also adjacent to v, we do not need to branch as none of the vertices have a connection to the rest of the graph, so we can simply put one of the vertices in D.

Length 3 chains

Now we look for a degree 3 vertex v with at least two degree 3 neighbors, say u and w. This can occur in a number of cases, which will be discussed below. In all these cases, we will look at the neighborhoods of u, v and w.



(a) *Case 4.9.1*



(b) *Case 4.9.2*

Figure 6: *The vertex shapes have different meanings. Circle: The entire neighborhood of the vertex is shown. Open square: The vertex has one or two neighbors that are not shown.*

Case 4.9.1: $N(u) = \{o, p, v\}, N(v) = \{q, u, w\}, N(w) = \{r, s, v\}$

This is the most basic case, shown in Figure 6(a). The vertices u, v and w do not have other common neighbors. We create ten total branches for this, of four types:

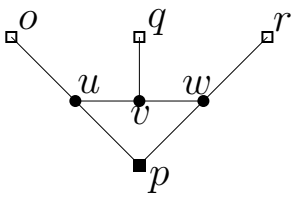
- One branch where only v is put in D.
- One branch where u and w are put in D.
- Four branches where either u and one of $\{r, s\}$, or w and one of $\{o, p\}$ are put in D.
- Four branches where one of $\{o, p\}$ and one of $\{r, s\}$ and q are put in D.

Note that none of the sets of vertices that are put in D in any of the branches is a subset of the set of vertices put in D in another branch. Because v can be put in D on it's own, it is not considered in any of the other branches.

Case 4.9.2: $N(u) = \{o, p, v\}, N(v) = \{p, u, w\}, N(w) = \{r, s, v\}$

In this case, shown in Figure 6(b), u and v have a common neighbor p. We know that p is degree 2, if it was degree 3 we would have a length three 3 cycle of degree 3 vertices, and it would previously have been found. Here we have eight branches that each ensure the degree 3 vertices u, v, and w are either in D or have at least one neighbor in D:

- One branch where only v is put in D.
- Three branches where u and one of $\{r, s, w\}$ is put in D.
- Two branches where w and one of $\{o, p\}$ is put in D.
- Two branches where p and one of $\{r, s\}$ is put in D.



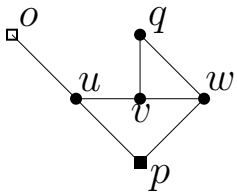
Case 4.9.3

Figure 7: *The vertex shapes have different meanings. Circle: The entire neighborhood of the vertex is shown. Open square: The vertex has one or two neighbors that are not shown. Closed square: The vertex has exactly one neighbor that is not shown.*

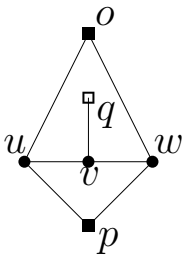
Case 4.9.3: $N(u) = \{o, p, v\}, N(v) = \{q, u, w\}, N(w) = \{p, r, v\}$

In this case, which is shown in Figure 7, u and w have a common neighbor p. We know p is degree 3; if it was degree 2 Reduction Rule 4.5 would have applied. There are five types of branches.

- One branch where only v is put in D.
- One branch where u and w are put in D.
- Two branches where either u and r, or o and w are put in D.
- One branch where o, q and r are put in D.
- One branch where p and q are put in D.



(a) *Case 4.9.4*



(b) *Case 4.9.5*

Figure 8: *The vertex shapes have different meanings. Circle: The entire neighborhood of the vertex is shown. Open square: The vertex has one or two neighbors that are not shown. Closed square: The vertex has exactly one neighbor that is not shown.*

Case 4.9.4: $N(u) = \{o, p, v\}, N(v) = \{q, u, w\}, N(w) = \{p, q, v\}$
This case is shown in Figure 8(a). Here, u and w have a common neighbor p again, and so do v and w with q. We know p is degree 3, otherwise Reduction Rule 4.5 would have applied. We also know q is degree 2; if it were degree 3 there would be a cycle that would have been found previously. Say $N(p) = \{r, u, w\}$. We have a total of nine branches:

- Three branches where o, v, and one or none of $\{p, r\}$ are put in D.
- Three branches where u, and one of $\{p, r, v\}$ are put in D. If r is put in D, we also add w to D.
- Three branches where v and one of $\{p, r, w\}$ are put in D.

In the branch where we put u, r and w in D, w is added since after removing u and r it is still degree 3.

Case 4.9.5: $N(u) = \{o, p, v\}, N(v) = \{q, u, w\}, N(w) = \{o, p, v\}$
In this case, shown in Figure 8(b), u and w have two common neighbors o and p. We know they both are degree 3, otherwise Reduction Rule 4.5 would have applied. We have $N(o) = \{s, u, w\}$ and $N(p) = \{r, u, w\}$.

- Seven branches where one of $\{o, s\}$, one of $\{p, r\}$, and one of $\{q, v\}$ are put in D, with at least one of $\{o, p, v\}$ being put in D.
- Three branches where u and one of $\{o, p, v\}$ are put in D.
- One branch where u and w are put in D.

In the first type of branch, there are three binary options, resulting in $2^3 = 8$ variants of that branch, and the restriction removes the combination $\{q, r, s\}$. In the second type of branch we could exchange u and w since they are symmetrical.

Two adjacent degree 3 vertices
Case 4.10 If we now have two adjacent degree 3 vertices u and v, we know all vertices in $N(\{u, v\})$ have degree 2; if any of them had degree 3 one of the previous cases would have applied. This allows for a simple branching rule.

- Two branches where one of $\{u, v\}$ is put in D.
- Four branches where a vertex from $N(u) \setminus \{v\}$ and one vertex from $N(v) \setminus \{u\}$ are put in D.

4.2.5 Equality to Edge Cover

What remains now is a graph where all vertices are either degree 2 or 3, with the additional knowledge that the two neighbors of a degree 2 vertex are always degree 3 vertices, and the three neighbors of a degree 3 vertex are always degree 2 vertices. Say $G = (V, E)$ and $V = V_2 \cup V_3$, where V_2 contains all degree 2 vertices in G, and V_3 contains all degree 3 vertices. We can now say G is a biregular bipartite graph, with V_2 and V_3 as the two color classes.

Theorem 4.4. *If a bipartite graph has one partition with only vertices of degree 2 and one partition with only vertices of degree 3, we can construct an Co-Path/Cycle-Packing on it in polynomial time.*

We will prove this theorem in the rest of this section. For this, we transform the graph and solve the Minimum Edge Cover problem on the new graph.

Definition 10. Minimum Edge Cover
Given: Graph $G = (V, E)$
Question: Give the set $S \subseteq E$ of minimum possible size such that each vertex in V is incident to at least one edge in S.

Lemma 4.5. *There is always a minimum cardinality co-path/cycle packing which contains no vertices from V_3.*

Proof. Suppose we have a minimum co-path/cycle packing S with $v \in S \cap V_3$. If any of the neighbors of v are in S, $S' = S - v$ would be a smaller co-path/cycle packing than S since v and all it's neighbors have maximum degree 2 in $G - S$. This cannot be the case, since S has minimum cardinality. Therefore $N(v) \cap S = \emptyset$. Say u is a neighbor of v, $S'' = (S - v) \cup \{u\}$ is a co-path/cycle packing with $|S| = |S''|$. We can repeat this for all vertices in $S'' \cap V_3$. □

In the following Lemmas we will use $G = ((V_2 \cup V_3), E)$ and $G' = (V_3, E')$, where G' is the graph G' with all vertices in V_2 contracted. It will be used to perform the Edge Cover algorithm on.

Lemma 4.6. *If S is a co-path/cycle packing in graph G with $V_3 \cap S = \emptyset$, $S_{EC} = \{uw \mid v \in S \vee uv \in E \vee vw \in E\}$ is an edge cover in graph G'.*

Proof. If S_{EC} is not an edge cover, there must be a vertex $v \in V_3$ for which no adjacent edges are in S_{EC}. This would mean no neighbors of v in G are in S, which would mean v is still degree 3 in $G[(V_2 \cup V_3) \setminus S]$, but since S is a co-path/cycle packing this cannot be the case. □

Lemma 4.7. *If S_{EC} is an edge cover in graph G', $S = \{v \mid uw \in S_{EC} \vee uv \in E\}$ is a co-path/cycle packing in G.*

Proof. If S is not a co-path/cycle packing in G, there must a vertex $v \in V_3$ with all three neighbors not in S, which would mean no edges adjacent to v in G' are in S_{EC}. Since S_{EC} is an edge cover, this cannot be the case. □

Lemma 4.8. *We can construct a minimum co-path/cycle packing in G in polynomial time.*

Proof. Constructing graph G' can trivially be done in polynomial time, and we can find a minimum edge cover S_{EC} on it in polynomial time by finding a maximum matching and extending it greedily [Law76]. Using S_{EC} we can construct a co-path/cycle packing S according to Lemma 4.7, which also takes polynomial time.

All that is left to show is that S has minimum size. Suppose there is some co-path/cycle packing S^* with $|S^*| < |S|$. We can use Lemma 4.6 to construct an edge cover on G', say S^*_{EC}. By construction we have $|S^*_{EC}| = |S^*|$ and $|S_{EC}| = |S|$, which means $|S^*_{EC}| < |S_{EC}|$, but since we know S_{EC} is a minimum edge cover, we have a contradiction and thereby show that S has minimum size. □

In summary, we know we can construct a minimum size solution with only degree 2 vertices through Lemma 4.5. After we contract them, these vertices become edges in G', and Lemmas 4.6 and 4.7 show that an edge cover on G' is also a Co-Path/Cycle-Packing on G, and vice versa. Finally Lemma 4.8 shows that if the edge cover on G' is of minimum size, so is the Co-Path/Cycle-Packing on G, and that we can construct these sets in polynomial time.

4.2.6 A note on correctness

Usually a proof of correctness would be shown, proving that the algorithm will find a solution if it is available, and that any solution it finds is valid. However, due to the extensive case analysis making up a large part of the algorithm, these proofs are straightforward and repetitive, and will therefore not be shown.

One can see that a solution will be found if it is available, by looking at each individual case and seeing that each minimal sub-solution for the described subgraph is included in one of the branches. A sub-solution is a set of vertices that leaves only vertices of degree 2 or lower when removed from the subgraph. Such a set is minimal if no vertex can be removed from the set and still have it be a valid sub-solution.

One can also easily see that any solution the algorithm finds will be valid. Suppose it is not, and we have a solution set S provided by the algorithm, and a vertex v that is still degree 3 or more in $G - S$. We know v cannot be degree 5 or more, since Case 4.1 would have applied. If it was degree 4 one of the cases in Section 4.2.3 would have applied, and if it was degree 3 one of the cases in Section 4.2.4 would have applied. After any case is applied, one or more vertices are added to D and that same case can no longer apply in that same place. Therefore we know a v cannot have a degree higher than 2 in $G - S$.

4.3 Analysis

In this section we will analyze the algorithm described in the previous section, part by part.

Case 4.1 deals with vertices of degree 5 or more. It is slowest vertices with degree 5, in which case it has branching vector $(1, \overrightarrow{3}^{10}, \overrightarrow{4}^{5}, 5)$. While the number of branches grows for higher degrees, the number of vertices put in D also grows, which outweighs the increased number of branches and ensures higher degrees get faster.

In the next step, vertices of degree 4 or more are handled. We start with Case 4.2, a vertex that has a neighbor in R, which has the branching vector $(1, \overrightarrow{2}^{3})$.

The next phase, which deals with two adjacent degree 4 vertices, has four subcases 4.3.1 through 4.3.4, depending on the number of common neighbors between the degree 4 vertices. In order of increasing case number, the branching vectors are $(2, \overrightarrow{2}^{6}, \overrightarrow{4}^{9})$, $(2, \overrightarrow{2}^{6}, \overrightarrow{3}^{4}, \overrightarrow{4}^{1})$, $(2, \overrightarrow{2}^{6}, 2, \overrightarrow{3}^{2})$, and $(2, \overrightarrow{2}^{3}, \overrightarrow{2}^{3}, 3)$.

The step after that deals with a degree 4 vertex adjacent to a degree 3 vertex. It has three subcases 4.4.1 through 4.4.3, with branching vectors (in order of increasing case number again) $(1, \overrightarrow{2}^{3}, \overrightarrow{3}^{6})$, $(1, \overrightarrow{2}^{3}, \overrightarrow{2}^{2}, 3)$, and $(1, 2)$.

In Case 4.5 we deal with a degree 4 vertex with a degree 2 neighbor. It has no subcases, and has a branching vector $(1, \overrightarrow{2}^{6})$.

The part that deals with degree 3 vertices consists of a number of different cases.

Case 4.6 handles degree 3 vertices with two neighbors in R. Two branches are created that both put one vertex in D, so this has branching vector $(1, 1)$.

In Case 4.7.1 vertices with one neighbor in R are dealt with. Two branches are created with one vertex added to D in each, giving the branching vector $(1, 1)$ again. Case 4.7.2 (where v and w are degree 3 and not adjacent) creates six branches and results in the branching vector $(1, \overrightarrow{2}^{5})$.

The phase that deals with cycles of length 3 consisting of degree 3 vertices has two distinct cases. The first one, Case 4.8.1, where the vertices have no common neighbors apart from each other, results in the branching vector $(\overrightarrow{1}^{3}, 3)$. This is very slow, but we can improve upon this when we look at the branches where one of the vertices in the cycle is put in D.

We will try to find a vertex of degree 3 with a neighbor in R. Say $N(u) = \{r, v, w\}$, $N(v) = \{s, u, w\}$, and $N(w) = \{t, u, v\}$. We know r, s, and t are either degree 3, or degree 2 with two degree 3 neighbors. If we now put v in D (the case is symmetrical when u or w is put in D), Reduction Rule 4.3 will put u and w in R. If either r or t is degree 3, we have found the vertex we are looking for. If not, both are degree 2, and we apply rule 4.4 to put r in R. The other neighbor of r was degree 3 before this step, and if it still is we have found the vertex we are looking for. The only vertices that now have a lower degree than before this step are s, u, and w, and we know r is not adjacent to any of them. We know the neighborhoods of u and w, and if it was adjacent to s rule 4.5 would have applied. Therefore this neighbor of r is still degree 3 and we have found a vertex that meets both of the requirements.

If this degree 3 vertex we found has three or more neighbor in R, it is put in D by rule 4.6, so the branch puts two vertices in D. Otherwise, the next time a branching step is performed, it is the one that applies to a degree 3 vertex with a neighbor in R, which has a branching vector of $(1, 1)$ or $(1, \overrightarrow{2}^{5})$, where the latter is the slower one. Since this is always slower than putting two vertices in D, this branching rule has branching vector $(\overrightarrow{2}^{3}, \overrightarrow{3}^{15}, 3)$.

The other case, Case 4.8.2, where two of the vertices from the cycle have a common neighbor, is much simpler than the previous one, and has branching vector $(1, 2, 2)$.

The step that handles three adjacent degree 3 vertices is split into a number of cases. Since most of the branching vectors are straightforward from the branching rule, they are summarized in Table 4. Case 4.9.2 is the only case with a special analysis and we will deliberate on it.

In the first and last types of branches of Case 4.9.2, we can apply the same technique as we did in Case 4.8.1. We will first look at the case in the first branch, where v is put in D. After v gets put in D, Reduction Rule 4.2

Case #	Branching Vector
4.9.1	$(1, 2, \overrightarrow{2}^{4}, \overrightarrow{3}^{4})$
4.9.2	$(2, \overrightarrow{3}^{5}, \overrightarrow{2}^{3}, \overrightarrow{2}^{2}, \overrightarrow{3}^{2}, \overrightarrow{4}^{10})$
4.9.3	$(1, 2, \overrightarrow{2}^{2}, 3, 2)$
4.9.4	$(2, \overrightarrow{3}^{2}, \overrightarrow{2}^{2}, 3, \overrightarrow{2}^{3})$
4.9.5	$(\overrightarrow{3}^{7}, \overrightarrow{2}^{3}, 2)$

Table 4

will put p in R, after which u will be put in R by rule 4.4. As before, we look for a degree 3 vertex with a neighbor in R. If o is degree 3, we have found it. If o is degree 2, it had two degree 3 neighbors before v was put in D. Since the other neighbor cannot be one of the vertices that was adjacent to o (since we know the complete neighborhood of those three vertices) it is still degree 3. Because o is put in R by rule 4.3 the other neighbor meets the requirements we are looking for.

Similar to Case 4.8.1, the degree 3 vertex with a neighbor in R we have found will either get put in D by Reduction Rule 4.6, or it will still have a neighbor in R when the next branching rule is performed. In the former case, which can happen when r and s are degree 2 and also adjacent to this degree 3 vertex, this branch puts two vertices in D, and in the latter case it guarantees a branching rule with branching vector $(1, \overrightarrow{2}^{5})$ as the next branching rule performed.

In the last type of branches, where p and one of $\{r, s\}$ are put in D, this happens as well. The vertices u, v, and w are now degree 2 and get put in R by rule 4.3 and rule 4.4. Say in this branch r is put in D, then o and s are the vertices that are either the degree 3 vertex we are looking for, or they are degree 2 and get put in R by rule 4.4. We will focus on s. In the latter case where it is degree 2 the other neighbor will be the degree 3 vertex we are looking for. It cannot have been adjacent to r, since then rule 4.5 would have been applied. Therefore we have found a vertex that meets the requirements.

As before, this will either lead to a removal of this vertex by rule 4.6, or the branching rule with branching vector $(1, \overrightarrow{2}^{5})$ can be applied next. Since

guaranteeing the application of the branching rule is slower than putting an extra vertex in D, this slower branching vector is listed in the table.

The final branching rule, Case 4.10, uses the same technique as Cases 4.8.1 and 4.9.2 do. A simple analysis would yield branching vector $(\overrightarrow{1}^{2}, \overrightarrow{2}^{4})$, but in the two branches were u or v are put in D, the two degree 2 neighbors of that vertex will become degree 1 get put in R by Reduction Rule 4.2. Additionally, if u is put in D, v and both of its neighbors are also put in R by rules 4.3 and 4.4, and the opposite happens when v is put in D. This means there are four vertices (the ones that started in $N(\{u, v\})$ that are now in R that have a degree 3 neighbor that is not in R. That means there is either a vertex that gets put in D by 4.6, or a degree 3 vertex that has two or one neighbor in R. The slowest of these cases is the latter; it guarantees a branching step with vector $(1, \overrightarrow{2}^{5})$ which gives the branching final vector $(\overrightarrow{2}^{2}, \overrightarrow{3}^{10}, \overrightarrow{2}^{4})$ for this case.

Theorem 4.4 proves that the final step can be done in polynomial time. Considering all this, the slowest is the branching factor for Case 4.9.1, which was shown in Table 4 to be $(1, 2, \overrightarrow{2}^{4}, \overrightarrow{3}^{4})$. This has branching factor 3.0607, which means this algorithm has a running time of $\mathcal{O}^*(3.0607^k)$. This is slightly faster than the algorithm of [Xia16] which has a running time of $\mathcal{O}^*(3.0645^k)$.

5 Conclusion

In this thesis we have shown four deterministic parameterized algorithms for four related vertex deletion problems. Three of these problems are newly introduced in this thesis, and we have proven they are $\mathcal{NP}$-complete. Table 5 summarized the time complexities we have shown for each problem.

Problem name	Time complexity
Induced Cycle Deletion Set	$\mathcal{O}(2^k n^3)$
Induced Path/Cycle Deletion Set	$\mathcal{O}(2^k n^3)$
Co-Cycle Packing	$\mathcal{O}^*(3^k)$
Co-Path/Cycle Packing	$\mathcal{O}^*(3.0607^k)$

Table 5

We have improved the deterministic time-bound on Co-Path/Cycle Packing , the only problem which previously had one, from $\mathcal{O}^*(3.0645^k)$ to $\mathcal{O}^*(3.0607^k)$.

5.1 Outlook

Possible further research could look into an analysis of the algorithms for Co-Cycle Packing and Co-Path/Cycle Packing using a Measure and Conquer approach [VFGK09]. This technique has previously been used to improve the parameterized time-bound on the Bounded-Degree-1 Vertex Deletion problem, which is very related to the problems discussed in this paper [Wu15]. We expect the time-bound on the described algorithms can be improved with this technique because deciding a vertex definitively does not go to the deletion set (and gets put in the residue set R) currently does not improve the time-bound on that branch. Using Measure and Conquer might change this.

For the Co-Cycle Packing algorithm the branching rule which is the bottleneck is the one that deals with vertices of degree 4. For this case, a very simple branching rule is used. This might be improved upon this by doing a case analysis, similar to the case analysis for vertices of degree 3. The second slowest branching rule, for Case 3.4.1, has branching factor 2.6759, which means the time-bound can be improved significantly if the current bottleneck is improved.

References

[ACG+12] Giorgio Ausiello, Pierluigi Crescenzi, Giorgio Gambosi, Viggo Kann, Alberto Marchetti-Spaccamela, and Marco Protasi. *Complexity and approximation: Combinatorial optimization problems and their approximability properties*. Springer Science & Business Media, 2012.

[BBNU12] Nadja Betzler, Robert Bredereck, Rolf Niedermeier, and Johannes Uhlmann. On Bounded-Degree Vertex Deletion parameterized by treewidth. *Discrete Applied Mathematics*, 160(1):53 – 60, 2012.

[CFF+10] Zhi-Zhong Chen, Michael Fellows, Bin Fu, Haitao Jiang, Yang Liu, Lusheng Wang, and Binhai Zhu. A Linear Kernel for Co-Path/Cycle Packing. In Bo Chen, editor, *Algorithmic Aspects in Information and Management*, pages 90–102, Berlin, Heidelberg, July 2010. Springer.

[CKX10] Jianer Chen, Iyad A. Kanj, and Ge Xia. Improved upper bounds for vertex cover. *Theoretical Computer Science*, 411(40):3736 – 3756, 2010.

[DP60] Martin Davis and Hilary Putnam. A computing procedure for quantification theory. *Journal of the ACM (JACM)*, 7(3):201–215, 1960.

[FGMN11] Michael R. Fellows, Jiong Guo, Hannes Moser, and Rolf Niedermeier. A generalization of Nemhauser and Trotters local optimization theorem. *Journal of Computer and System Sciences*, 77(6):1141 – 1158, 2011.

[FWLC15] Qilong Feng, Jianxin Wang, Shaohua Li, and Jianer Chen. Randomized parameterized algorithms for P_2-Packing and Co-Path Packing problems. *Journal of Combinatorial Optimization*, 29(1):125–140, Jan 2015.

[Kar72] Richard M Karp. Reducibility among combinatorial problems. In *Complexity of computer computations*, pages 85–103. Springer, 1972.

[Law76] Eugene L Lawler. *Combinatorial optimization: networks and matroids*. Courier Corporation, 1976.

[LY80] John M. Lewis and Mihalis Yannakakis. The node-deletion problem for hereditary properties is NP-complete. *Journal of Computer and System Sciences*, 20(2):219 – 230, 1980.

[VFGK09] Fedor V. Fomin, Fabrizio Grandoni, and Dieter Kratsch. A Measure & Conquer Approach for the Analysis of Exact Algorithms. *Journal of the ACM (JACM)*, 56(5):25, August 2009.

[Wu15] Bang Ye Wu. A Measure and Conquer Approach for the Parameterized Bounded Degree-One Vertex Deletion. In Dachuan Xu, Donglei Du, and Dingzhu Du, editors, *International Computing and Combinatorics Conference*, pages 469–480, Beijing, China, August 2015. Springer.

[Xia16] Mingyu Xiao. A Parameterized Algorithm for Bounded-Degree Vertex Deletion. In Thang N. Dinh and My T. Thai, editors, *International Computing and Combinatorics Conference*, pages 79–91, Ho Chi Minh City, Vietnam, August 2016. Springer.