# Comparative survey of existing fixed-parameter tractable Vertex Cover algorithms and their implementation

Tymon Solecki

COM3610 – Dissertation Project

This report is submitted in partial fulfilment of the requirement for the degree of BSc Computer Science by Tymon Solecki.

20/11/2018 University of Sheffield

All sentences or passages quoted in this report from other people's work have been specifically acknowledged by clear cross-referencing to author, work and page(s). Any illustrations that are not the work of the author of this report have been used with the explicit permission of the originator and are specifically acknowledged. I understand that failure to do so amounts to plagiarism and will be considered grounds for failure in this project and the degree examination as a whole.

-Tymon Solecki

### Abstract

Graphs are a great tool for modelling real world systems and interactions. From flow of city traffic to phenotype identification, graphs and graph manipulation prove to have plentiful of real-world applications. In this paper, we introduce fundamental -complete problem of Computer Science from domain of graph theory, Vertex Cover. We go over a range of possible methods to solve it, and delve deeper into fixed parameter tractable algorithms and compare their theoretical and practical performance.

We present a plan to implement different FPT algorithms in C and compare their efficiency. Plans for implementation of a library allowing others to freely create their own versions using different kernel techniques is introduced.

s

Table of Contents

[Chapter 1: Introduction 1](#_Toc531605771)

[Preliminaries 1](#_Toc531605772)

[Chapter 2: Literature review 2](#_Toc531605773)

[Kernel reductions for Vertex Cover 3](#_Toc531605774)

[Linear Programming 5](#_Toc531605775)

[Crown decomposition 5](#_Toc531605776)

[Branching 6](#_Toc531605777)

[Vertex Cover Above LP 7](#_Toc531605778)

[Interleaving 7](#_Toc531605779)

[Chapter 3: Requirements and analysis 7](#_Toc531605780)

[Chapter 4: Conclusions and project plan 8](#_Toc531605781)

[References 9](#_Toc531605782)

# Chapter 1: Introduction

Graphs are a great tool for modelling real world systems and interactions. From flow of city traffic to phenotype identification, graphs and graph manipulation prove to have plentiful of real-world applications. One of the prominent NP-complete problems in graph theory is Vertex-Cover problem. It is used in complexity theory to prove NP-hardness of other problems. Due to the seemingly intractable nature of V-C, focus have been historically concentrated on the design of heuristics and other approximate, polynomial-time algorithms [1], [2]. One relatively recent alternative has emerged - it is fixed parameter tractability (FPT).

In this paper, concepts needed to analyse Vertex Cover problem are presented (Preliminaries), V-C problem is formulated as an instance of FPT problem, FPT algorithm for V-C and different techniques that are employed to improve time complexity said algorithm are surveyed (Literature Review). To finish, implementation and justification of the choice of tooling for this project is shown (Requirements and analysis & Design & Implementation).

### Aims, objects and deliverables of the project

Main objective for the project is to exhaustively explore known FPT techniques for V-C. Theoretical upper-bound complexity and practical performance is compared. Furthermore, implementation of the algorithm that computes exact solution to V-C problem is provided. Once implemented, different combinations kernelization and preprocessing methods to minimise the time measure for large graph instances are explored, with the purpose of submitting it to PACE challenge[[1]](#footnote-1).

## Preliminaries

It is assumed that the reader has familiarity with algorithm topics on second year computer science undergraduate level.

In this chapter, we go over topics and definitions necessary for understanding of Vertex Cover and FPT algorithms. All of those are introduced briefly, for more comprehensive reading overview see excellent Introduction to Algorithms (Third ed.), by Cormen, Thomas; Leiserson, Charles; Rivest, Ronald; Stein, Clifford (2009).

### Graphs

Throughout the paper, is string representation of finite and non-empty graph. It consists of vertex set and edge set . Graph is undirected unless stated otherwise - that is, if for and in is in , then is in .

If is in , we call and adjacent, and non-adjacent otherwise. Let us call in . In this case, we call and (as well as and ) incident.

Set of vertices adjacent to we denote as ; from that, degree of a vertex is going to be .

*Matching* of into exists if every edge of connects a vertex of and a vertex of and every vertex of is an endpoint of some edge of . In this situation, we also say that saturates [3]

### Input size

Input size will be interpreted differently depending on what problem is being studied. For many, most natural measure is number of items in the input. For others (like multiplying integers) it would be total number of bits needed to represent the input in binary.

In terms of Vertex Cover, input for the program is a graph and a number of size of vertex cover (in its decision version). Therefore, input size could be described by the number of vertices in the graph, or by the number of edges in the graph, or by some combination of both. In this work, unless stated otherwise, input size is described as a pair , which denotes number of vertices and edges respectively.

### Running time

Running time of an algorithm on some input is the number of primitive "steps" it takes to execute. It would be ideal to have some model of "step" that is as machine-independent as possible. For practical purposes, one "simple arithmetic operation" - that is, addition, subtraction, multiplication, division - is treated as a singular step, as well as assignment of value.

It is convenient to express running time as a function of input size. We will be using big-O notation[4, p. 47]. We will also use notation – means that there exists an algorithm which runs in time . It is used to omit polynomial part of the full algorithm when we are interested in parametrized part only.

In context of this project, worst-case scenario is assumed. Knowing worst-case running time, there is a guarantee that an algorithm is going to complete its run in that time regardless of input.

### Growth of functions

Order of growth is measured by considering only the leading term in the formula - for example, for polynomial running time it is going to be its highest polynomial. This simplifying abstraction is useful for comparison of the efficiency of different algorithms - an algorithm is considered more efficient from other algorithm if its worst-case order of growth is smaller than that of the other algorithm.

### P vs NP

To analyse fixed-parameter tractable algorithms, the concept of tractability is introduced. Class of problems is called if and only if that for size n of the input can be solved in for some constant k. Similarly, class of problems is called if and only if it is not solvable in and solution instance is verifiable in for some constant . Additionally, problem belongs to -hard class if every problem from can be reduced to [5]. Problem is -complete if it belongs both to and -hard classes [6].

### Vertex Cover

Vertex-cover problem examines a question: what is the minimum number of vertices that form a valid vertex cover? In other words, to solve V-C problem we need to find V-C of minimal size. To reformulate it as a decision problem (which will let us parametrize it later), following question is asked: “Does a graph has a V-C of size k?” We can define a language of V-C in this way:

V-C problem is -complete [4, p. 1090].

# Chapter 2: Literature review

Many concepts in V-C kernelization can be found in “Parameterized Algorithms” (Daniel Lokshtanov, Dániel Marx, Fedor V. Fomin, Marek Cygan, and Saket Saurabh), which is extensively referred in this paper. Apart from that, research papers which examine those and other kernels theoretically and empirically are examined. Other approaches, including interleaving and parallelization, are investigated as well.

### Fixed-parameter tractable algorithms

A parameterized problem is a language, where is a fixed, finite alphabet. For an instance , is called the parameter [3]. With that definition, problems can be parametrized in different ways. In case of Vertex Cover, it is parametrized based on the solution size. Let be an instance of vertex cover; is going to represent undirected graph encoded as a string over the alphabet and is a positive integer. That means that pair belongs to parametrized language of vertex cover if and only if correctly encodes an undirected graph and contains a vertex cover of size .

### Kernelization

Kernelization (preprocessing) is used in computer implementations that aim to tackle NP-hard problem. Its goal is to solve the “easy parts” of the problem instance, thus reducing it to difficult, “core” structure. Kernelization methods (kernels) that reduce problem instance to an equivalent “smaller sized” instance in time polynomial in the input size are considered. *Reduction* rule for parameterized problem is defined as a function that maps an instance of to an equivalent instance of such that is computable in time polynomial in and . Two instances of are equivalent if belongs to if and only if belongs to . This will be also referred to as *safeness* or soundness of the reduction rule [3].

We will measure the output size of preprocessing algorithm A as function size:

We look at all possible instances of with fixed parameter ; then we take the “biggest” size of the output of on these instances. Kernelization algorithms are those preprocessing algorithms whose output size is finite and bounded by a computable function of parameter.

**A kernelization algorithm**, or simply a kernel, for a parameterized problem is an algorithm that, given an instance of , works in polynomial time and returns an equivalent instance of . Moreover, it is required that for some computable function . [3]

## Kernel reductions for Vertex Cover

There are various ways to reduce VC instance - kernels range from simple ones, which allow us to remove independent and low degree vertices from the instance, to more complex rules, where specific substructures of graphs are identified. Simple rules (also called preprocessing rules) are examined, followed with Linear Programming, Crown Reduction.

First reduction follows from the fact that including isolated vertex in solution would not cover any edge - thus rendering the solution not optimal. From that, it is trivial that an isolated vertex is not going to be in optimal VC, and removing it from G is not going to change the solution - thus the following rule is safe.

### Reduction VC.1

If is an isolated vertex in , delete from . New instance is

The second rule is based on the observation that for each vertex , either is in VC or is included in the solution – one can think of it as a division based on which vertices will cover edges incident to – it is either going to be , or all of the vertices adjacent to . Moreover, if the degree of is more than , then it should be included in every VC of size at most . Otherwise, from above, are included in the solution, but . More than vertices are added to VC, but VC cam be of size at most - we arrived at contradiction. From that:

### Reduction VC.2

If there is a vertex of degree at least , then delete (and its incident edges) from and decrement the parameter by one. The new instance is . [3]

After application of VC.1 and VC.2, an instance where the degree of each vertex v is

is obtained.

For VC.3, one more observation is needed:

If a graph has maximum degree , then a set of vertices can cover at most edges. [3]

After application of VC.1 and VC.2, for each Vertex Cover , each vertex outside of VC should be adjacent to some vertex from . Each vertex has degree at most . From above observation and hence [3]. If there exists a yes-instance then there is a VC of size , so .

### Reduction VC.3

Let be an input instance such that Reductions VC.1 and VC.2 are not applicable to . If and has more than vertices, or more than edges, then it is a no-instance. [3]

From VC.3, VC admits a kernel with vertices and edges.

From here preprocessing rules proposed by Abu-Khzam and others in Kernelization Algorithms for the Vertex Cover Problem: Theory and Experiments” are investigated. Duplicates of VC.1 - VC.3 are omitted.

Observation: in case of pendant vertex (that is, of degree one) there is an optimal vertex cover that does not contain the pendant vertex but does contain its unique neighbour. Thus, in G , both the pendant vertex and its neighbour can be eliminated [7].

### Reduction VC.4

For, if is in and , then add to VC and delete and . New instance is . Addition of might create new isolated and/or pendant vertices, so we apply VC.4 repeatedly until there are no more pendant vertices.

For next rule, a degree-two vertex with adjacent neighbours is considered. Call the vertex and its neighbours and . At least two of the vertices from must be in VC. Choosing would only cover edges and while eliminating and including and could possibly cover not only these but additional edges [7].

Therefore, there exists an optimal VC that includes and but does not include .

### Reduction VC.5

For , if is in and and is in , delete and add and to VC. New instance is . This rule is applied repeatedly until all degree-two vertices with adjacent vertices are eliminated [7].

Let us consider situation where a degree-two vertex has non-adjacent neighbours. Either it is included in the VC, which leaves other edges of and not covered; or and are included, thus covering all edges from and . This equivalent to the following:

### Reduction VC.6

For , if is in and and is not in , replace and with , whose neighborhood is the union of neighborhoods of and in . New instance is

.

In other words, if vertex is chosen in further steps, it is equivalent to choosing and in the original . If it is not chosen, it is equivalent to choosing in the original graph.

## Linear Programming

If we can specify the objective as a linear function of certain variables, and if we can specify the constraints on resources as equalities or inequalities on those variables, then we have a *linear programming* problem.

[4, p. 864]

In this section -vertex kernel for V-C is presented. Linear programming formulation of VC () is encoded as:

[3, p. 34]

Where is cost of vertex . Notice that if additional requirement were to be added, only then we would have an accurate representation of VC problem. Since requirement for cost function to only take integer values is dropped, we call it a *relaxation* of integer linear programming problem. It can be assumed that without loss of generality [8].

### LP reduction

Let be an optimum solution to in a Vertex Cover instance . If , then is a no-instance, because is relaxed V-C. Otherwise, greedily take into the vertex cover the vertices that were given weight 1 in solution. adapted from [3].

LP reduction rule is safe and admits a kernel with at most 2k vertices in time , where is number of edges and is number of vertices in .[3, pp. 36–37]

## Crown decomposition

Crown decomposition is a general kernelization technique that can be used to obtain kernels for many problems. The technique is based on the classical matching theorems of Kőnig and Hall.

[3]

In recent years, Crown Decomposition has become a popular alternative to Linear Programming approach for Vertex Cover. Despite having worse upper bound for kernel size, in practice it allows for faster kernelization, often yielding similar gains as LP [9]. Also, surprisingly, despite looking for different substructure in a graph it does not appear to be orthogonal to LP [7]. In practice, that means that using those kernels together won’t yield better results.

Crown Decomposition is defined as:

A partitioning of into three parts and , such that

1. is nonempty.
2. is an independent set.
3. There are no edges between vertices of and . That is, separates and .
4. Let be the set of edges between vertices of and . Then  contains a matching of size . In other words, contains a matching of into [3]

One can think of Crown decomposition as a generalization of VC.4 - in case of pendant vertex , is the crown, while is the head. Analogous operation as in VC.4 exists in case of Crown Decomposition - choose all vertices from head to VC over crown vertices.

Note that since contains a matching of size , it means it also contains a matching of into . Since there are no edges between and , all of the edges incident to have to be covered by vertices from and .

Since there is a matching of the edges between and , any vertex cover must contain at least one vertex from each matched edge. Thus, the matching requires at least vertices. is independent, so can be chosen as the set of vertices for VC. Furthermore, choosing a vertex from allows covering more edges than choosing a vertex from . As a result, minimum-size VC contains all and none [7]

Once found, reduction of the instance occurs as follows: is added to VC and delete and . New instance is

For our purposes, if exists, crown decomposition in graph with at least vertices can be found in polynomial time using crown lemma[3, p. 27]. Thus, crown decomposition allows kernel of size.

Crown decompositions can be classified further into flared and straight crowns[7], which may allow further improvement in time complexity of crown decomposition.

### Hopcroft-Karp algorithm

Given a bipartite graph, it produces a maximum cardinality matching in .[10]

Main observation for the algorithm: having some matching and an augmenting path , where augmenting path is defined as one beginning and ending on a free vertex (not covered by matching), symmetric difference of and will yield a matching of size .

Consider bipartite graph. The algorithm starts breadth-first search from , where, in first layer, all vertices are free. From each free vertex, it computes depth first search alternating between matched and unmatched edges. For each vertex, continues until exhaustion or alternating path is found. That way, for each layer of , *maximal* set of vertex-disjoint augmenting paths is found. Taking symmetric difference of current matching and union of those paths will increase the size of the machting by the number of augmenting paths found.

### Kőnig’s theorem

Kőnig’s theorem states that, in bipartite graph, size of minimum vertex cover equals size of maximum matching [11]

Construction of such a cover described is polynomial [12]/

### Crown lemma

There exists a polynomial algorithm that for instance of returns either

* Matching of size k+1
* Non-empty crown decomposition

The algorithm in pseudocode:

1. Find greedy maximal matching of
2. If size of , then algorithm is finished
3. Let be endpoints of . ()
4. is an independent set (because is maximal)
5. Consider the bipartite graph formed by edges of between and . We compute a minimum-sized vertex cover and a maximum sizedmatching of by Hopcroft-Karp algorithm and Konig’s theorem.

## Branching

Bounded search trees, or simply branching, is one of the simplest and most commonly used techniques in parameterized solutions for vertex cover problem. In context of VC, this algorithm tries to build a feasible solution by deciding, one by one, whether a given vertex should be included in VC or not. It *branches* on the vertex, and creates two subproblems - one where given vertex is included in VC and on where it is not. Then the same operation is made recursively on just created subproblems. One may interpret the algorithm as a search tree, with the root being the original problem, and if we find a solution, we terminate the algorithm. In decision version of VC, we can bind the size of the search tree created by branching by the parameter alone - thus bounded search trees belong to FPT algorithms [3, p. 51].

### Application of branching for Vertex-Cover

Choice of the vertex that we branch on can be done in clever way - worst-case time complexity may depend on this. Therefore, instead of choosing the vertex at random, or sequentially, vertex that has most neighbours is chosen.

If highest degree in the graph is then the solution is trivial.

Running time of this algorithm is going to be

Time taken by each node is , and recursive formula can be applied to bind number of leaves in a bounded search tree with formula:

Solving that recurrence yields upper bound of .[3, p. 54]

Using preprocessing methods described earlier, all vertices of degree 2 or less can be eliminated, thus further improving upper bound of branching to .[3, p. 55]

## Vertex Cover Above LP

“If the minimum value of linear programming formulation of VC problem is , then the size of a minimum vertex cover is at least . This leads to the following parameterization of Vertex Cover, which is called : Given a graph and an integer , is there a vertex cover of of size at most , but instead of seeking an FPT algorithm parameterized by as for Vertex Cover, the parameter now is .” [3].

This approach to defining parameter is called *above guarantee parametrization* – in this case, an algorithm that has running time of . In other words, lower bound is found for the solution size ( ) and instead of parametrizing purely by , try to parametrize by .This approach makes sense for large , since then algorithms parametrized purely by would still have long running time.

In description for this algorithm, by *optimum solution* to we mean a feasible solution that minimizes the objective function (analogously to minimizing in VC)[13].

There has been some research into above guaranteevariant of the problem[14], which might prove useful when trying to minimize time measure of V-C algorithm.

## Interleaving

Up until now, kernelizations were analysed independently. However, as decomposition proceeds, additional reductions can be often realized by re-application of previous preprocessing methods[15]. In context of FPT it is called *interleaving.* Because it has non-negligible computational overhead and gives variable effects for different types of graphs, it is not obvious how often it should be used [16].

# Chapter 3: Requirements and analysis

Here are things that I would like to pursue further in this project:

### Further theoretical research

Techniques described in Chapter 2 are in no way exhaustive list. In particular, section about *above guarantee* variant of LP kernel needs to be explored more thoroughly. Also, there are kernels which use combination of those and their more generalised versions[17], [18]. I would like to explore whether implementation of more intricate methods is feasible in the timeframe of my project.

Another area of optimization is use of parallel computing for V-C. Parallelization can be easily applied to branching algorithm, although to fully utilize parallel resources it needs some form of dynamic task re-distribution[16].

Implementing kernels beyond LP and Crown reduction is dependent on this task.

### Implementation of efficient version of FPT algorithm

I looked into several libraries to implement the algorithm[[2]](#footnote-2). There are advantages to using a pre-written library – I can implementation of data structures and focus solely on developing algorithms. Still, because objects and methods in those libraries are inter-dependent, I foresee it to be time consuming and impractical to change the source code of those libraries. It narrows my ability to optimize code – and PACE competition, for which I want to submit my solution, depends on the algorithm being efficient. In short, writing my own data structures gives me more fine-grained control over how my program performs.

I have not chosen whether I will be using a library or implementing my own; however, it is most probable that I am going to implement my on structures in C programming language.

It will also be worth looking at open-sourced implementations of V-C or similar problems (Max-Clique, Hamiltonian Cycles) to familiarise oneself with standard practices used with those kinds of problems.

Once I have particular kernels and preprocessing rules implemented, I can compare their effectiveness based on which combination works best, which ordering of rules works better, how do they compare on different problem instances.

# Chapter 4: Design

For the implementation of the solver, to use igraph library and Python programming language. Main trade-off of using Python instead of C or igraph’s implementation in C is lack of control over memory allocation. According to benchmarks[[3]](#footnote-3), igraph implementation in python does not differ in speed of particular functions over C and R igraph library. On the other side, due to being higher-level language, using Python allowed to implement more functionality and tests.

# Chapter 5: **Implementation and testing**

## Implementation

Implementation of solver is divided into separate modules serving different purposes:

* vc\_preprocessing.py is a module where reduction rules are implemented. Each of the rules is passed graph and parameter (size of VC), and applies the rules, returning equivalent problem instance consisting of new graph, new parameter and vertices to take to vertex cover of the original instance. There are also two subroutines to apply preprocessing rules together: apply\_preprocessing, which applies all the rules until none of them diminishes the size of the instance, and no\_param\_preprocessing, which applies those rules that do not require the instance to be parametrized.
* Crown\_decomposition.py – this module provides a crown decomposition kernel for . Additionally, it implements: finding maximal matching (greedy), Hopcroft-Karp algorithm to find maximum matching in bipartite graph, Konig’s theorem to get minimum Vertex Cover from maximum matching in bipartite graph[19]
* vc\_io.py – module that creates igraph graph instances from text files provided on PACE website. It is also used to export solutions to format required by PACE, as well as convert graph files to other formats for the sake of benchmarking the program with different sets of graphs.
* vc\_solver.py – main function is solving for instances provided by using binary search over parameter k; it also has performance measures to compare different versions of kernels. Those measures use Python’s performance counter [[4]](#footnote-4), which is a default tool for benchmarking in Python modules[20, p. 633], and number of vertices in the graph visited.
* vc\_checker.py – module to check correctness of other modules. check\_correctness, given a graph and a vertex cover, verifies whether the solution is correct, though it doesnot check whether it is optimal. There are also subroutines to check and visualize correctness of crown decomposition and maximal matching.
* branchbound.py – main function branch\_and\_reduce is used to branch on for instances to which preprocessing cannot be applied. branch function, given instance and a vertex to branch on, produces two smaller instances after branching. It uses maximal matching as a lower bound and greedy 2-approximation solution as an upper bound.[21]
* vc\_profiler.py – to cut down and process data about function calls into human-readable format.

Main function of vc\_solver works as follows:

1. read the graph
2. apply simple (quadratic) preprocessing rules until exhaustion
3. apply crown decomposition and go back to 2.
4. Calculate lower bound (greedy maximal matching)
5. Set upper bound to number of vertices in the graph (it is at most 2-approximation, since isolated vertices were eliminated in 2.)
6. Perform binary search for solution for over with bounds set above. Note: if we find a solution and it is smaller than , it is still valid (only some of the popular vertex reductions that might have been employed had been chosen closer to its optimal value were not executed).
   1. Apply steps 2 and 3 to the instance
   2. Once exhausted, find vertex of highest degree and branch on and
7. Once solution is found, verify solution via vc\_checker
8. Save solution to file and record time spent and nodes visited

Steps 2 and 3 guarantee linear kernel in respect to vertex cover size. Step 6.1 implements interleaving. Step 6.2 minimizes tree depth according to branching on maximum degree vertex.

When vc\_solver is invoked, it requires one argument. It is a name of a folder where instances of are contained. Script will save solution in a format described on PACE website in the same folder.

For graphs , visualisation takes considerable time to draw. This function was used only to check correctness and present the example solutions from famous benchmark.

Additional arguments taken by vc\_solver are:

* – choice of algorithm, where 0 is naïve branching, 1 – branch and bound, 2 – quadratic kernel (simple preprocessing), 3 – interleaving (preprocessing after each branch), 4 – addition of crown decomposition kernel.
* –draw each solution with igraph’s graphics library.
* – set timeout on a single instance to .

Example usage:



## Testing

Program was tested on Intel Core i7 2620M processor on following datasets:

* public instances provided by PACE organisers,
* randomly generated instances,
* famous benchmark dataset from website wolfram alpha, including grid and cage graphs and some other edge cases.

### PACE instances

**TODO**: Results here

PACE provided large instances, ranging from 200 to over 10000 vertices. Those instances, which after kernelization had lower bound above 1000 did not execute on my laptop with timeout set to 120 seconds – search tree becomes too big. It is consistent with theoretical assumptions that FPT algorithms work for small values of the parameter. Interestingly, preprocessing rules have made the kernel linear; In 0 out of 100 instances in this set crown decompositions prerequisites were met – that is, instance after reduction rules satisified. This may be because those graphs are relatively sparse.

Thu Apr 25 18:26:19 2019 solver\_profile

1539475553 function calls (1301532693 primitive calls) in 754.776 seconds

Ordered by: internal time

List reduced from 1123 to 20 due to restriction <20>

ncalls tottime percall cumtime percall filename:lineno(function)

237964940/33907 258.293 0.000 599.190 0.018 copy.py:132(deepcopy)

66955738 119.809 0.000 362.218 0.000 copy.py:219(\_deepcopy\_tuple)

475964590 60.614 0.000 60.614 0.000 {method 'get' of 'dict' objects}

66955737 46.360 0.000 236.934 0.000 copy.py:220(<listcomp>)

56064 42.263 0.001 578.948 0.010 copy.py:210(\_deepcopy\_list)

10548 37.032 0.004 51.888 0.005 branchbound.py:13(get\_maximal\_matching)

19581 34.094 0.002 39.547 0.002 vc\_preprocessing.py:76(pendant\_v\_reduction)

305311477 25.817 0.000 25.817 0.000 {built-in method builtins.id}

90041 20.575 0.000 20.575 0.000 {method 'delete\_vertices' of 'igraph.Graph' objects}

57257206 17.284 0.000 17.284 0.000 \_\_init\_\_.py:2495(vs)

170830583 16.164 0.000 16.164 0.000 copy.py:190(\_deepcopy\_atomic)

19581 13.511 0.001 29.623 0.002 vc\_preprocessing.py:105(degree\_two\_reduction)

107813453 10.764 0.000 10.764 0.000 {method 'append' of 'list' objects}

22209 9.233 0.000 9.233 0.000 {method 'get\_edgelist' of 'igraph.Graph' objects}

22249 8.626 0.000 8.724 0.000 \_\_init\_\_.py:144(\_\_init\_\_)

99 6.354 0.064 8.557 0.086 vc\_io.py:5(readgraph)

9506133 5.364 0.000 5.364 0.000 {method 'neighbors' of 'igraph.Graph' objects}

9243438 3.075 0.000 3.075 0.000 {method 'get\_eid' of 'igraph.Graph' objects}

19581 2.558 0.000 7.495 0.000 vc\_preprocessing.py:54(popular\_v\_reduction)

19581 2.480 0.000 7.576 0.000 vc\_preprocessing.py:35(isolated\_v\_reduction)

### Famous benchmark instances

Famous benchmark test includes a lot of regular graphs with varying structures. Those served for debugging purposes for preprocessing rules and manual examination of correctness of the algorithm. Module to visualise solutions for those instances was also implemented. While is small for all instances, difference in time needed to find a solution is consistent with theory. For some we can see naïve algorithm perform substantially worse than the other two, and for quadratic kernel outperforms .

|  |  |  |  |  |
| --- | --- | --- | --- | --- |
| Graph instance | |V| | time of execution (timeout 10) | | |
|  |  | branch | branch&bound | bnb+linear kernel |
| famous/Ellingham.edge | 78 | TIMEOUT | 0.175 | 0.034 |
| famous/B10Cage.edge | 70 | TIMEOUT | 0.051 | 0.025 |
| famous/Watsin.edge | 50 | TIMEOUT | 0.066 | 0.016 |
| famous/Holt.edge | 27 | 0.369 | 0.019 | 0.012 |
| famous/5x5-grid.edge | 25 | 0.122 | 0.014 | 0.004 |
| famous/McGee.edge | 24 | 0.164 | 0.014 | 0.005 |
| famous/Nauru.edge | 24 | 0.175 | 0.016 | 0.007 |
| famous/Kittell.edge | 23 | 0.096 | 0.013 | 0.011 |
| famous/Brinkmann.edge | 21 | 0.060 | 0.008 | 0.007 |
| famous/Desargues.edge | 20 | 0.054 | 0.007 | 0.005 |
| famous/Dodecahedron.edge | 20 | 0.080 | 0.008 | 0.007 |
| famous/FlowerSnark.edge | 20 | 0.053 | 0.010 | 0.005 |
| famous/Folkman.edge | 20 | 0.032 | 0.008 | 0.006 |
| famous/Robertson.edge | 19 | 0.040 | 0.017 | 0.007 |
| famous/Pappus.edge | 18 | 0.037 | 0.017 | 0.004 |
| famous/Errera.edge | 17 | 0.022 | 0.009 | 0.007 |
| famous/Paley17.edge | 17 | 0.017 | 0.017 | 0.010 |
| famous/4x4-grid.edge | 16 | 0.014 | 0.004 | 0.003 |
| famous/Clebsch.edge | 16 | 0.019 | 0.007 | 0.006 |
| famous/Hoffman.edge | 16 | 0.014 | 0.005 | 0.004 |
| famous/Shrikhande.edge | 16 | 0.017 | 0.013 | 0.008 |
| famous/Sousselier.edge | 16 | 0.018 | 0.010 | 0.003 |
| famous/Poussin.edge | 15 | 0.011 | 0.012 | 0.006 |
| famous/Paley13.edge | 13 | 0.007 | 0.009 | 0.006 |
| famous/Chvatal.edge | 12 | 0.005 | 0.005 | 0.004 |
| famous/Durer.edge | 12 | 0.007 | 0.004 | 0.003 |
| famous/Franklin.edge | 12 | 0.005 | 0.003 | 0.002 |
| famous/Frucht.edge | 12 | 0.007 | 0.005 | 0.003 |
| famous/Tietze.edge | 12 | 0.012 | 0.007 | 0.002 |
| famous/Goldner.edge | 11 | 0.002 | 0.002 | 0.002 |
| famous/Grotzsch.edge | 11 | 0.003 | 0.003 | 0.002 |
| famous/Herschel.edge | 11 | 0.003 | 0.002 | 0.002 |
| famous/Petersen.edge | 10 | 0.004 | 0.005 | 0.002 |
| famous/3x3-grid.edge | 9 | 0.002 | 0.003 | 0.002 |
| famous/Pmin.edge | 9 | 0.004 | 0.004 | 0.001 |
| famous/Wagner.edge | 8 | 0.003 | 0.003 | 0.001 |
| famous/Moser.edge | 7 | 0.002 | 0.003 | 0.001 |
| famous/Prism.edge | 6 | 0.001 | 0.003 | 0.001 |
| famous/Bull.edge | 5 | 0.001 | 0.001 | 0.000 |
| famous/Butterfly.edge | 5 | 0.001 | 0.002 | 0.000 |
| famous/Diamond.edge | 4 | 0.001 | 0.001 | 0.000 |
| famous/path4.edge | 3 | 0.000 | 0.000 | 0.000 |
| famous/tri.edge | 3 | 0.001 | 0.001 | 0.000 |
| **Average** |  | **0.037** | **0.007** | **0.004** |
| note: average time exclude graphs for which there was at least one timeout | | | | |

## Possible improvements

Various parts of the program make extensive use of deepcopy method from as well as . Latter has proven to be time-consuming in real use cases[[5]](#footnote-5). Apart from rewriting the algorithm in different language, module and two methods used currently could be compared as to their time efficiency.

# Chapter 6: Conclusions and project plan

So far I am content with the pace of my work – given theoretical nature of the subject, I expected to spend a lot of time exploring techniques developed so far.

I will implement one algorithm, iterate on it, possibly creating alternative versions with different kernelization and preprocessing sequences, and compare those versions on different problem test cases. Therefore, order of event is analogous to deliverables of the project (with cautious internal deadlines in the brackets; as for approximation of the time it is going to take to implement something, in the past I rarely found it meaningful):

1. Research more FPT techniques for Vertex Cover problem (ongoing)
2. Explore existing libraries and decide on possible usage of any (15/01)
3. Implement bounded search tree algorithm (01/02)
4. Implement ‘simple’ preprocessing rules for V-C instances (14/02)
5. Implement LP kernel, Crown reduction kernel (14/03)
   1. Debugging and testing (21/03)
6. Iterate on the algorithm, benchmarking it with instances provided by PACE and other real-life graph instances (01/04)
7. Compete the final report (14/04)

If I proceed with listed items faster than predicted, I will continue my research into more complex FPT techniques (like parallelization) and implement those. Alternatively, I may work on a more complete version of C library for efficient graph algorithms, expanding beyond structures and functions I need for V-C problem set.

## References

[1] F. C. Gomes, C. N. Meneses, P. M. Pardalos, G. Valdisio, and R. Viana, “Experimental Analysis of Approximation Algorithms for the Vertex Cover and Set Covering Problems,” *Comput. Oper. Res.*, vol. 33, pp. 3520–3534, 2006.

[2] S. Cai, K. Su, C. Com, and A. Sattar, “NuMVC: An Efficient Local Search Algorithm for Minimum Vertex Cover Chuan Luo,” 2013.

[3] M. Cygan, F. V. Fomin, Ł. Kowalik, D. Lokshtanov, D. Marx, M. Pilipczuk, M. Pilipczuk, and S. Saurabh, *Parameterized Algorithms*. 2015.

[4] T. H. Cormen, *Introduction to algorithms*. MIT Press, 2009.

[5] D. E. Knuth, “Postscript about NP-hard problems,” *ACM SIGACT News*, vol. 6, no. 2, pp. 15–16, Apr. 1974.

[6] J. van (Jan) Leeuwen, *Handbook of theoretical computer science*. Elsevier, 1994.

[7] F. N. Abu-Khzam, M. R. Fellows, M. A. Langston, and W. H. Suters, “Crown Structures for Vertex Cover Kernelization,” *Theory Comput. Syst.*, vol. 41, no. 3, pp. 411–430, Oct. 2007.

[8] G. L. Nemhauser and L. E. Trotter, “Vertex packings: Structural properties and algorithms,” *Math. Program.*, vol. 8, no. 1, pp. 232–248, Dec. 1975.

[9] F. N. Abu-Khzam, R. L. Collins, M. R. Fellows, M. A. Langston, W. H. Suters, and C. T. Symons, “Kernelization Algorithms for the Vertex Cover Problem: Theory and Experiments \*.”

[10] J. E. Hopcroft and R. M. Karp, “A n5/2 algorithm for maximum matchings in bipartite,” in *12th Annual Symposium on Switching and Automata Theory (swat 1971)*, 1971, pp. 122–125.

[11] D. Kőnig, “Gráfok és alkalmazásuk a determinánsok és a halmazok elméletére,” *Mat. és Természettudományi Értesítő*, vol. 34, pp. 104–119, 1916.

[12] J. A. (John A. Bondy and U. S. R. Murty, *Graph theory with applications*. North Holland, 1976.

[13] K. Thulasiraman, S. Arumugam, A. Brandstädt, and T. (Takao) Nishizeki, *Handbook of graph theory, combinatorial optimization, and algorithms*. .

[14] D. Lokshtanov, N. S. Narayanaswamy, V. Raman, M. S. Ramanujan, and S. Saurabh, “Faster Parameterized Algorithms using Linear Programming ∗,” *Arxiv Prepr. arXiv1203.0833*, pp. 1–38, 2012.

[15] R. Niedermeier and P. Rossmanith, “A general method to speed up fixed-parameter-tractable algorithms,” *Inf. Process. Lett.*, vol. 73, no. 3–4, pp. 125–129, Feb. 2000.

[16] F. N. Abu-Khzam, M. A. Langston, P. Shanbhag, and C. T. Symons, “Scalable parallel algorithms for FPT problems,” *Algorithmica (New York)*, vol. 45, no. 3, pp. 269–284, Jul. 2006.

[17] J. Chen, I. A. Kanj, and G. Xia, “Improved upper bounds for vertex cover,” *Theor. Comput. Sci.*, vol. 411, pp. 3736–3756, 2010.

[18] J. Chen, I. A. Kanj, and W. Jia, “Vertex Cover: Further Observations and Further Improvements,” *J. Algorithms*, vol. 41, no. 2, pp. 280–301, Nov. 2001.

[19] W. J. Cook, W. H. Cunningham, W. R. Pulleyblank, and A. Schrijver, *Combinatorial Optimization*. Wiley, 1997.

[20] M. Lutz, *Learning Python.* O’Reilly Media, 2013.

[21] C. H. Papadimitriou and K. Steiglitz, *Combinatorial optimization : algorithms and complexity*. Prentice Hall, 1982.

1. [https://pacechallenge.org/2019/#](https://pacechallenge.org/2019/) [↑](#footnote-ref-1)
2. <https://graph-tool.skewed.de/>, <https://www.boost.org/doc/libs/1_58_0/libs/graph/doc/>, <https://igraph.org/redirect.html> [↑](#footnote-ref-2)
3. <https://graph-tool.skewed.de/performance> [↑](#footnote-ref-3)
4. <https://docs.python.org/3/library/time.html> [↑](#footnote-ref-4)
5. <https://stackoverflow.com/questions/24756712/deepcopy-is-extremely-slow>, <https://stackoverflow.com/questions/8957400/what-is-the-runtime-complexity-of-pythons-deepcopy>, <https://writeonly.wordpress.com/2009/05/07/deepcopy-is-a-pig-for-simple-data/> [↑](#footnote-ref-5)