# 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.

Table of Contents

[Comparative survey of existing fixed-parameter tractable Vertex Cover algorithms and their implementation i](#_Toc7548007)

[Chapter 1: Introduction 1](#_Toc7548008)

[Preliminaries 1](#_Toc7548009)

[Chapter 2: Literature review 3](#_Toc7548010)

[Kernel reductions for Vertex Cover 3](#_Toc7548011)

[Linear Programming 5](#_Toc7548012)

[Crown decomposition 6](#_Toc7548013)

[Branching and bounding 8](#_Toc7548014)

[Vertex Cover Above LP 9](#_Toc7548015)

[Interleaving 9](#_Toc7548016)

[Non-standard vertex-degree based parametrizations of Vertex Cover 10](#_Toc7548017)

[Chapter 4: Design and testing 10](#_Toc7548018)

[Programming language considerations 10](#_Toc7548019)

[Implementation 10](#_Toc7548020)

[Pseudocode 12](#_Toc7548021)

[Profiling 13](#_Toc7548022)

[Improvements 14](#_Toc7548023)

[Chapter 5: Benchmarks and relative performance comparison 17](#_Toc7548024)

[Famous benchmark instances 18](#_Toc7548025)

[PACE instances 20](#_Toc7548026)

[Chapter 6: Conclusions 20](#_Toc7548027)

[Theory and practical tests 20](#_Toc7548028)

[Program functionality 20](#_Toc7548029)

[Further work 20](#_Toc7548030)

[References 21](#_Toc7548031)

[Appendices 22](#_Toc7548032)

# 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 problem are presented (Preliminaries), problem is formulated as an instance of problem, algorithm for and different techniques that are employed to improve time complexity said algorithm are surveyed (Literature Review). Justification of the choice of tooling for this project is shown (Requirements and analysis & Design). In Implementation and Testing, program is explained with accompanying pseudocode, benchmarks are analysed and further improvements are proposed.

### 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. Certain python and iGraph specific implementation details are presented.

## 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:

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

# Chapter 2: Literature review

Many concepts in 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 vertex cover 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, 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 vertex cover 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 crown , head and the rest , 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

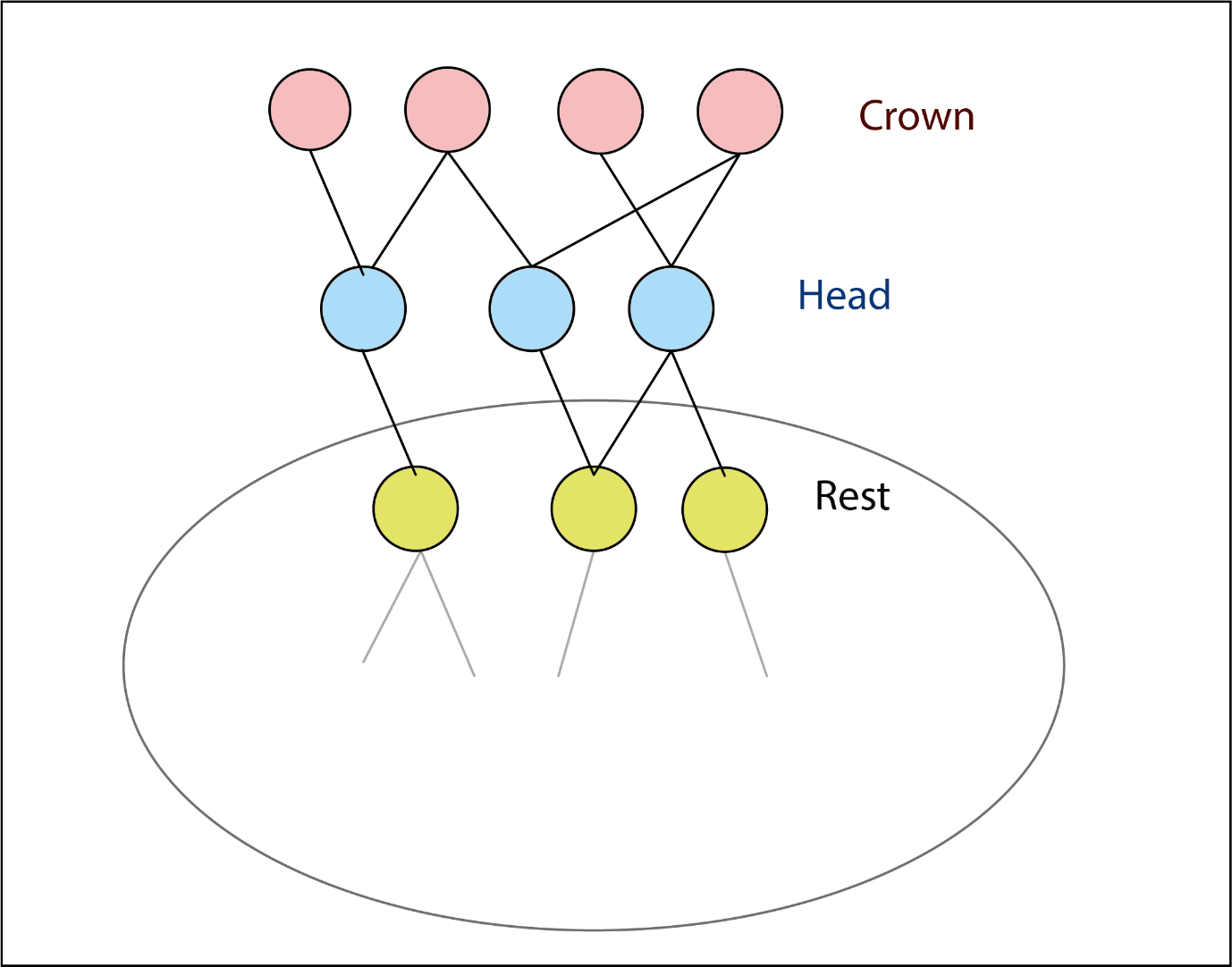


Figure 1: illustration of crown structure

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.

Next, two theorems are explained which are needed later to obtain crown decomposition in a graph.

### Kőnig’s theorem

Kőnig’s theorem states that, in bipartite graph, size of minimum vertex cover equals size of maximum matching [10]

Construction of such a cover described is polynomial [11, pp. 74–75].

### Hopcroft-Karp algorithm

Given a bipartite graph, it produces a maximum cardinality matching in .[12]

Definitions:

Edge is **matched** if it belongs to a matching . Vertex is called **free** if it is not incident to a matched edge. **Augmenting path** is a path through the graph which starting and ending vertices are free and the edges alternate between matched and not matched.

Main observation for the algorithm: having some matching and an augmenting path , 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 matching by the number of augmenting paths found.

### Crown lemma

There exists a polynomial algorithm that for instance of with that returns either:

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

High-level description of the algorithm:

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.
6. If then algorithm is finished. Therefore we can assume (Konig’s theorem)
7. We can show that [3, p. 28]. Let denote the subset of such that every edge from has exactly one endpoint in and let denote the set of endpoints of edges in .

We define head: , crown , and the remaining part

.

## Branching and bounding

Bounded search trees 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 limit the size of the search tree created by branching only in such a way so that it decreases the parameter - 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]

### Bounding

Branch method can be bound by calculating a minimum bound for a given branch and comparing it to the parameter. If minimum bound is bigger than the parameter, branch is rejected, and pursued further otherwise.

## 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].

### Above Guarantee Vertex Cover

Lokshtanov et al.[14] provide in time , where k is size of solution minus size of maximum matching in . It is a feasible method for instance, where solution size is large, but maximal matching size is large as well.

## 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]. In the implementation, two versions of interleaving are implemented: ‘local’ where, only vertices neighbouring the one branched on are considered when applying preprocessing rules, and ‘naïve’, where all vertices are iterated upon until the instance cannot be reduced anymore.

## Non-standard vertex-degree based parametrizations of Vertex Cover

There is an alternative approach to in case of large instances. If a graph has edges and all degrees are bound from above by , we know that vertex cover will be at least of cardinality . Therefore, can be parametrized as follows:

For where , and , is there a vertex cover with at most vertices?[17]. Gutin, Kim, Lampis and Mitsou in this paper provide number of similar formulations of , which will not be focused on further in this work.

# Chapter 4: Design and testing

In Design chapter, justification of the choice of tooling for the project is presented along the alternatives, implemented modules and their functionality are described. Pseudocode for crucial parts of the final version of the algorithm is offered. Tools for profiling the performance of the program are given (cProfile, timeit and performance\_counter()[[1]](#footnote-1)) and those are used to identify inefficient parts of the implementation. Corrections are proposed, and advanced version of the program is benchmarked against initial one over PACE problem instances, which proves that changes were justified and shortened the runtime of the program considerably.

## Programming language considerations

Several libraries to implement the algorithm were considered[[2]](#footnote-2). There are advantages to using a pre-written library – focus is placed solely on developing algorithms. However writing data structures from the ground up allows more fine-grained control over how the program performs.

I have chosen to implement the solver using Python and iGraph library[[3]](#footnote-3).

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[[4]](#footnote-4), iGraph implementation in python does not differ in speed of particular functions over iGraph library in C and R. On the other side, due to being higher-level language, using Python allowed to write functions in a concise manner.

## Implementation

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

* vc\_solver.py –main program to solve instances. It also has performance measures to compare different versions of kernels. Those measures use Python’s performance counter [[5]](#footnote-5), which is a default tool for benchmarking in Python modules[18, p. 633], and number of vertices in the graph visited, saving them to performance log.
* 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\_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. Additionaly, checker includes simple plotting subroutine to visualize solutions using graphic library Pycairo[[6]](#footnote-6)
* 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.[20]
* vc\_profiler.py – to cut down and process data about function calls into human-readable format. Also provides auxiliary functions to retrieve size of solutions from solution files.
* Graph\_generator.py – function generate\_random\_graphs generates graphs and saves them to a folder. Takes 4 arguments of: folder to save graphs to, number of instances to create, vertices in each instance, density of a graph ( from 0 to 1).
  + Usage: python3 graph\_generator.py f n v d

When vc\_solver is invoked, it requires one positional 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.

Additional arguments taken by vc\_solver are:

* – choice of algorithm, where
  + 0 is naïve branching on highest degree vertex,
  + 1 is branch and bound – for each branch, upper bound is and lower bound is size of naively found maximal matching,
  + 2 –simple quadratic kernel – naïve preprocessing; each of the reduction rules is applied to each of the vertices, in total time ,
  + 3 – interleaving (preprocessing after each branch),
  + 4 – addition of crown decomposition (linear) kernel,
  + 5 – local preprocessing (described in improvements);
* –draw each solution with iGraph’s graphics library (for graphs , visualisation takes considerable time to draw. This function was used only to check correctness and present the example solutions from famous benchmark.),
* – set timeout on a single instance to ,
* – verbosity of the program, mainly used for debugging purposes. When set to 0, program does not print to stdout. Each consecutive number produces information about kernelizations and branching with increasing granularity.

Example usage:



To get decomposition of running time for particular functions and primitives, use:



To run generated tests:

python3 graph\_generator [folder] [# of instances] [# of vertices in an instace] [density (0-1)]

python3 vc\_solver [folder]

Finally, python3 vc\_solver –h for possible configurations.

## Pseudocode

Main functions of solvers are and . Lower bound of the search tree was adapted from work of Akiba, Takuya, Iwata, Yoichi [21].

Solver first calculates for solution size as the size of maximal matching and sets to . Using binary search over for parameter , it invokes .

In , parameter is partial solution at a given branch in the algorithm.

Vertices are added to in and in , as well as deleted from instance .

is passing the best solution found so far in the search tree.

**Algorithm 1**: branch and bound algorithm for

1. **Function** :
   1. If –o flag set to {3, 4} then
   2. If –o flag set above 0 and then return
   3. if theb return
   4. let be maximum degree vertex
   5. **Return**
2. **Function**
   1. Let be neighbourhood of , and union of neighbourhoods of vertices in
   3. If –o flag set to 5 then
   4. **return**

function guarantees linear kernel in respect to vertex cover size. Using it inside executes interleaving. bounds tree depth according to branching on maximum degree vertex.

## Profiling

Profiling using cProfile[[7]](#footnote-7) has shown important property of the program. It is a tool for *deterministic profiling*- that is, shows how long various parts of the program were executed. For public PACE instances and timeout of 120 seconds per instance, the program took 6720 seconds to execute.

This version of the program was using deepcopy module extensively, which resulted in majority of the execution time spent on copy subroutines.



Figure 2: profiling of solver with cProfile over PACE benchmark tests

## Improvements

### Local preprocessing

One way to make preprocessing rules faster is implementation of local preprocessing. When branching, it queues neighbours of vertex branched on, and apply preprocessing reductions only to those vertices, adding their neighbours to queue in case one of the rules was applied. One can see it as performing breadth-first search from neighbours of the vertex branched on.

**Algorithm 2**: local preprocessing

1. **function**
   1. **While** is not empty:
      1. **if** one of the preprocessing rules allows to reduce :
         1. , add to partial solution if needed (depending on particular reduction rule)
2. **return**

This local preprocessing algorithm does not look at all vertices of in every invocation; instead, after first iteration, it looks at at most vertices without any deletion (if a vertex of degree was branched on). Additional vertices will be retrieved only if some vertices are added to the solution. Efficiency of this addition is measured by solving the same instances with one version of algorithm using local preprocessing rules, and one of them standard preprocessing (iterating over all vertices) and comparing number of nodes visited.

Additionally, local preprocessing uses all of the reduction rules in the same iteration, in contrast to initial version, which iterated over all vertices for each reduction rule separately.

### Python and iGraph-specific improvements

Implementation of local preprocessing was attempted. However, because deletion of vertices in iGraph causes ids of other vertices to change[[8]](#footnote-8), retrieving neighbours of a given vertex requires assigning additional attribute to them and finding those vertices based on their attributes in updated graph. It requires time, which defeats the purpose of making the preprocessing rules local – if the reduction still has to look through all of the vertices of it’s time complexity remains unchanged.

#### Difference in performance for different vertex selection

Due to one of the undocumented features of iGraph, finding vertices in updated graphs can be achieved in using “name” attribute, because this attribute is the only attribute of vertices in iGraph that is indexed[[9]](#footnote-9). This allowed to complete local preprocessing to improve worst-case time complexity. In tests performed, it did not influence time performance of the program greatly, due to the fact that copying elements is leading in total time and resource consumption of the program.

Comparison of retrieving vertices was made using IPython’s magic built-in functions[[10]](#footnote-10). For randomly generated graph with 10 000 vertices, retrieval of vertex using ‘name’ attribute turned out to take on average , while the same operation but using different attribute averaged at . It reaffirmed results from iGraph’s mailing list[[11]](#footnote-11) – find executes around 636 times faster using ‘name’ attribute.



Figure 3 time measure comparison for "name" and arbitrary attribute

#### Difference in performance for different graph copying methods

Various parts of the program make extensive use of deepcopy method from as well as . Comparison of three different copying methods was done –from deepcopy, pickle and graph module respectively.

Comparison was done using the same method as for vertex retrieval in figure 3. For graph of 10000 vertices, deepcopy module took 7.04 seconds on average, pickle – 1.48 seconds and iGraph 24.9 milliseconds (figure 4). Therefore, using iGraph’s copy is on average **282** times faster.



Figure 4 timeit benchmarks for different copying methods

Figure 4 shows difference in time performance between the program (using linear kernel –o 4) . For 99 PACE instances provided, initial version of the algorithm executed for 33 of them with timeout set to 120 seconds. Number of nodes and time of execution visited has grown exponentially with the size of the solution (and, in case of those instances, with as well – see table 1 and 2 in appendix for performance measures for particular instances.

Figure 5 comparison of performance for different copy functions used

#### Recursive depth problem

Python sets what is considered by some to be conservative[[12]](#footnote-12) limit on a maximum number of recursive program calls in order to avoid segmentation fault errors [18, p. 561]. On largest instances from PACE, program was throwing RecursionError. This was rectified by increasing recursion limit to 10 000. However, this does not guarantee that the error is not going to be thrown for even larger instances – something that could be ultimately fixed by rewriting program to work iteratively.

# Chapter 5: Benchmarks and relative performance comparison

First, initial implementation and results are discussed. Performance of the algorithm is analysed, improvements in data structures and functions used are proposed. Finally, best produced version is presented and possible future improvements are considered.

Results are presented using survival plots as recommended defined in SAT community practices paper [22]. With each algorithm version separately, it sorts the time of execution on each instance into increasing order. Thus, it has clearly shown which algorithms perform faster on average.

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

* public instances provided by PACE organisers,
* famous benchmark dataset from website wolfram alpha, including grid graphs, cage graphs and other edge cases to test soundness of the algorithm

Time of execution was measured using Python’s performance counter[[13]](#footnote-13)

## Famous benchmark instances

Famous test includes many small, regular graphs with varying structures. It is widely used benchmark set for graph based problems, e.g. for SAT Encodings of Treedepth or Branchwidth [23][24].

Those tests served for debugging purposes for preprocessing rules and manual examination of correctness of the algorithm. Those instances were also manually checked against optimal solutions obtained from Wolfram Alpha website[[14]](#footnote-14).

Since famous graphs are relatively small, difference in performance by omitting preprocessing rules and/or bounding the kernel by maximal matching is shown.

is the solver run with flag –o 0,

is the solver run with flag –o 1,

is the solver run with flag –o 3.

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 it timed out (10 seconds) on instances with for quadratic kernel starts notably outperforming .

On average, performed over 5 times faster than , and configuration using linear kernel performed faster than .

Figure 6 famous benchmark instances measurement for three algorithm versions

|  |  |  |  |  |
| --- | --- | --- | --- | --- |
| Time performance for three different versions of vertex cover solver on famous benchmark set | | | | |
| Graph instance | |V| | time of execution (timeout 10) | | |
|  |  | branch | branch&bound | bnb+linear kernel |
| Ellingham.edge | 78 | TIMEOUT | 0.175 | 0.034 |
| B10Cage.edge | 70 | TIMEOUT | 0.051 | 0.025 |
| Watsin.edge | 50 | TIMEOUT | 0.066 | 0.016 |
| Holt.edge | 27 | 0.369 | 0.019 | 0.012 |
| 5x5-grid.edge | 25 | 0.122 | 0.014 | 0.004 |
| McGee.edge | 24 | 0.164 | 0.014 | 0.005 |
| Nauru.edge | 24 | 0.175 | 0.016 | 0.007 |
| Kittell.edge | 23 | 0.096 | 0.013 | 0.011 |
| Brinkmann.edge | 21 | 0.060 | 0.008 | 0.007 |
| Desargues.edge | 20 | 0.054 | 0.007 | 0.005 |
| Dodecahedron.edge | 20 | 0.080 | 0.008 | 0.007 |
| FlowerSnark.edge | 20 | 0.053 | 0.010 | 0.005 |
| Folkman.edge | 20 | 0.032 | 0.008 | 0.006 |
| Robertson.edge | 19 | 0.040 | 0.017 | 0.007 |
| Pappus.edge | 18 | 0.037 | 0.017 | 0.004 |
| Errera.edge | 17 | 0.022 | 0.009 | 0.007 |
| Paley17.edge | 17 | 0.017 | 0.017 | 0.010 |
| 4x4-grid.edge | 16 | 0.014 | 0.004 | 0.003 |
| Clebsch.edge | 16 | 0.019 | 0.007 | 0.006 |
| Hoffman.edge | 16 | 0.014 | 0.005 | 0.004 |
| Shrikhande.edge | 16 | 0.017 | 0.013 | 0.008 |
| Sousselier.edge | 16 | 0.018 | 0.010 | 0.003 |
| Poussin.edge | 15 | 0.011 | 0.012 | 0.006 |
| Paley13.edge | 13 | 0.007 | 0.009 | 0.006 |
| Chvatal.edge | 12 | 0.005 | 0.005 | 0.004 |
| Durer.edge | 12 | 0.007 | 0.004 | 0.003 |
| Franklin.edge | 12 | 0.005 | 0.003 | 0.002 |
| Frucht.edge | 12 | 0.007 | 0.005 | 0.003 |
| Tietze.edge | 12 | 0.012 | 0.007 | 0.002 |
| Goldner.edge | 11 | 0.002 | 0.002 | 0.002 |
| Grotzsch.edge | 11 | 0.003 | 0.003 | 0.002 |
| Herschel.edge | 11 | 0.003 | 0.002 | 0.002 |
| Petersen.edge | 10 | 0.004 | 0.005 | 0.002 |
| 3x3-grid.edge | 9 | 0.002 | 0.003 | 0.002 |
| Pmin.edge | 9 | 0.004 | 0.004 | 0.001 |
| Wagner.edge | 8 | 0.003 | 0.003 | 0.001 |
| Moser.edge | 7 | 0.002 | 0.003 | 0.001 |
| Prism.edge | 6 | 0.001 | 0.003 | 0.001 |
| Bull.edge | 5 | 0.001 | 0.001 | 0.000 |
| Butterfly.edge | 5 | 0.001 | 0.002 | 0.000 |
| Diamond.edge | 4 | 0.001 | 0.001 | 0.000 |
| path4.edge | 3 | 0.000 | 0.000 | 0.000 |
| tri.edge | 3 | 0.001 | 0.001 | 0.000 |
| *Table 1: time of execution for three (inefficient) configurations of solver for all “famous” instances* | | | | |

## PACE instances

PACE provided large instances, ranging from 200 to over 60 000 vertices. All configurations were run with timeout of 120 seconds. and have timed out for all instances.

Contrary to expectation, quadratic kernel version and linear kernel version of the algorithm were performing quite similarly on those tests. Possible explanation of that is that graphs provided were rather small – most of the instances which executed have . For graphs with few vertices, quadratic bound may not differ much from linear one. However, this is intuitive observation and should be treated as such – there might be specific types of graphs for which crown decomposition might outperform quadratic kernel.

## Random instances

Graphs of different density were generated to test the hypothesis above crown decomposition being

Random instances yielded mixed results; while the performance proved to be consistent over a large set of instances, linear kernel and quadratic kernel were performing similarly on most tests; moreover, due to the specific way iGraph’s GRG function[[15]](#footnote-15) of random graphs generates instances, it is unclear whether graphs generated this way cover all types of graphs. Due to time constraints tests on randomly generated instances were not pursued further.

# Chapter 6: Conclusions

## Theory and practical tests

Tests have confirmed FPT models for algorithms. “Famous “ benchmark has shown that bounding the search tree and reductions rules both provide great improvements in the performance of the program, and the gains they yield are additive.

Benchmarks on PACE instances demonstrated that choice of modules used to implement the algorithm is crucial to optimize real-life performance; Version of the algorithm using iGraph’s copying function outperformed version using deepcopy’s copying module greatly.

Randomly generated instances have revealed (marginal) benefit of using local preprocessing rules and higher efficiency of linear kernels on denser graphs.

## Program functionality

Majority of features planned for was implemented, including parsers for PACE-style graphs, branch and bound algorithm, reduction rules (both naïve and more refined, local version) and crown decomposition kernel.

However, crown decomposition was the only linear kernel implemented; linear programming is more thoroughly explored kernelization of , therefore objective of implementing linear kernels is only partly met.

As a side effect of implementing the solver itself, tools to benchmark the performance of a given configuration of the algorithm (vc\_profiler.py) and to generate different sets of graph instances (graph\_generator.py) were made. Thus, with little alteration the module can function as fundamentals for measuring performance of different graph-based algorithms.

## Further work

The project could be enhanced by implementing LP based linear kernel – that would allow direct comparison of two most popular FPT linear kernels. Above that, as very large instances from PACE have shown, in some cases it may be more efficient to parametrize by something else than solution size. More refined parameters include solution size minus maximal matching [14] and vertex-degree based parameters[17]. Therefore, natural progression of the project would be to explore literature and existing algorithms using different parameters and compare their performance with the ones already prepared.

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

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

[12] 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.

[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] G. Gutin, E. J. Kim, M. Lampis, and V. Mitsou, “Vertex Cover Problem Parameterized Above and Below Tight Bounds.”

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

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

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

[21] T. Akiba and Y. Iwata, “Branch-and-Reduce Exponential/FPT Algorithms in Practice: A Case Study of Vertex Cover,” 2014.

[22] M. Nyx Brain, J. H. Davenport, and A. Griggio, “Benchmarking Solvers, SAT-style.”

[23] R. Ganian, N. Lodha, S. Ordyniak, and S. Szeider, “SAT-Encodings for Treecut Width and Treedepth,” in *2019 Proceedings of the Twenty-First Workshop on Algorithm Engineering and Experiments (ALENEX)*, Philadelphia, PA: Society for Industrial and Applied Mathematics, 2019, pp. 117–129.

[24] N. Lodha, S. Ordyniak, and S. Szeider, “A SAT Approach to Branchwidth \*,” 2017.

# Appendices

Full table for PACE instances for two versions of the algorithm:

|  |  |  |  |
| --- | --- | --- | --- |
| Pace instances, timeout = 120 seconds, copy.deepcopy() used, naïve preprocessing rules | | | |
| instance | |V| | time elapsed [timeout = 120s] | nodes visited |
| vc-exact\_001.gr | 6160 | TIMEOUT |  |
| vc-exact\_003.gr | 60541 | TIMEOUT |  |
| vc-exact\_007.gr | 8794 | TIMEOUT |  |
| vc-exact\_009.gr | 38452 | TIMEOUT |  |
| vc-exact\_011.gr | 9877 | TIMEOUT |  |
| vc-exact\_013.gr | 45307 | TIMEOUT |  |
| vc-exact\_015.gr | 53610 | TIMEOUT |  |
| vc-exact\_017.gr | 23541 | TIMEOUT |  |
| vc-exact\_021.gr | 24765 | TIMEOUT |  |
| vc-exact\_023.gr | 27717 | TIMEOUT |  |
| vc-exact\_025.gr | 23194 | TIMEOUT |  |
| vc-exact\_027.gr | 65866 | TIMEOUT |  |
| vc-exact\_029.gr | 13431 | TIMEOUT |  |
| vc-exact\_039.gr | 6795 | TIMEOUT |  |
| vc-exact\_047.gr | 200 | TIMEOUT |  |
| vc-exact\_055.gr | 200 | TIMEOUT |  |
| vc-exact\_079.gr | 26300 | TIMEOUT |  |
| vc-exact\_087.gr | 13590 | TIMEOUT |  |
| vc-exact\_089.gr | 57316 | TIMEOUT |  |
| vc-exact\_095.gr | 15783 | TIMEOUT |  |
| vc-exact\_097.gr | 18096 | TIMEOUT |  |
| vc-exact\_099.gr | 26300 | TIMEOUT |  |
| vc-exact\_101.gr | 26300 | TIMEOUT |  |
| vc-exact\_103.gr | 15783 | TIMEOUT |  |
| vc-exact\_105.gr | 26300 | TIMEOUT |  |
| vc-exact\_107.gr | 13590 | TIMEOUT |  |
| vc-exact\_109.gr | 66992 | TIMEOUT |  |
| vc-exact\_111.gr | 450 | TIMEOUT |  |
| vc-exact\_113.gr | 26300 | TIMEOUT |  |
| vc-exact\_115.gr | 18096 | TIMEOUT |  |
| vc-exact\_117.gr | 18096 | TIMEOUT |  |
| vc-exact\_119.gr | 18096 | TIMEOUT |  |
| vc-exact\_121.gr | 18096 | TIMEOUT |  |
| vc-exact\_123.gr | 26300 | TIMEOUT |  |
| vc-exact\_125.gr | 26300 | TIMEOUT |  |
| vc-exact\_127.gr | 18096 | TIMEOUT |  |
| vc-exact\_129.gr | 15783 | TIMEOUT |  |
| vc-exact\_133.gr | 15783 | TIMEOUT |  |
| vc-exact\_135.gr | 26300 | TIMEOUT |  |
| vc-exact\_137.gr | 26300 | TIMEOUT |  |
| vc-exact\_139.gr | 18096 | TIMEOUT |  |
| vc-exact\_141.gr | 18096 | TIMEOUT |  |
| vc-exact\_143.gr | 18096 | TIMEOUT |  |
| vc-exact\_145.gr | 18096 | TIMEOUT |  |
| vc-exact\_147.gr | 18096 | TIMEOUT |  |
| vc-exact\_149.gr | 26300 | TIMEOUT |  |
| vc-exact\_151.gr | 15783 | TIMEOUT |  |
| vc-exact\_153.gr | 29076 | TIMEOUT |  |
| vc-exact\_155.gr | 26300 | TIMEOUT |  |
| vc-exact\_157.gr | 2980 | TIMEOUT |  |
| vc-exact\_159.gr | 18096 | TIMEOUT |  |
| vc-exact\_161.gr | 138141 | TIMEOUT |  |
| vc-exact\_163.gr | 18096 | TIMEOUT |  |
| vc-exact\_165.gr | 18096 | TIMEOUT |  |
| vc-exact\_167.gr | 15783 | TIMEOUT |  |
| vc-exact\_169.gr | 4768 | TIMEOUT |  |
| vc-exact\_171.gr | 18096 | TIMEOUT |  |
| vc-exact\_173.gr | 56860 | TIMEOUT |  |
| vc-exact\_179.gr | 15783 | TIMEOUT |  |
| vc-exact\_181.gr | 18096 | TIMEOUT |  |
| vc-exact\_183.gr | 72420 | TIMEOUT |  |
| vc-exact\_189.gr | 7400 | TIMEOUT |  |
| vc-exact\_193.gr | 7030 | TIMEOUT |  |
| vc-exact\_195.gr | 1150 | TIMEOUT |  |
| vc-exact\_197.gr | 1534 | TIMEOUT |  |
| vc-exact\_199.gr | 1534 | TIMEOUT |  |
| vc-exact\_177.gr | 5066 | 74.7946373 | 17612308 |
| vc-exact\_191.gr | 4579 | 62.5852474 | 13586683 |
| vc-exact\_187.gr | 4227 | 53.0780551 | 10966147 |
| vc-exact\_085.gr | 11470 | 49.5567036 | 27252144 |
| vc-exact\_175.gr | 3523 | 36.9099557 | 7935483 |
| vc-exact\_185.gr | 3523 | 36.7892153 | 8004151 |
| vc-exact\_033.gr | 4410 | 33.6643743 | 12395728 |
| vc-exact\_131.gr | 2980 | 27.4342354 | 6126310 |
| vc-exact\_091.gr | 200 | 5.62535871 | 421432 |
| vc-exact\_057.gr | 200 | 5.32362473 | 392446 |
| vc-exact\_083.gr | 200 | 5.17146441 | 368178 |
| vc-exact\_093.gr | 200 | 5.13506833 | 395642 |
| vc-exact\_067.gr | 200 | 5.07773096 | 360184 |
| vc-exact\_081.gr | 199 | 5.04589218 | 373481 |
| vc-exact\_051.gr | 200 | 4.7935966 | 375226 |
| vc-exact\_073.gr | 200 | 4.73089617 | 399112 |
| vc-exact\_053.gr | 200 | 4.50447667 | 362952 |
| vc-exact\_041.gr | 200 | 4.43585832 | 369410 |
| vc-exact\_069.gr | 200 | 4.43145153 | 328746 |
| vc-exact\_063.gr | 200 | 4.34266859 | 357700 |
| vc-exact\_065.gr | 200 | 4.3260781 | 356224 |
| vc-exact\_071.gr | 200 | 4.14351419 | 359404 |
| vc-exact\_045.gr | 200 | 4.13912673 | 350600 |
| vc-exact\_061.gr | 200 | 4.02033358 | 367582 |
| vc-exact\_043.gr | 200 | 3.98862391 | 383602 |
| vc-exact\_077.gr | 200 | 3.97993408 | 357562 |
| vc-exact\_049.gr | 200 | 3.80338567 | 344396 |
| vc-exact\_059.gr | 200 | 3.74029756 | 357562 |
| vc-exact\_031.gr | 200 | 3.44846995 | 340922 |
| vc-exact\_035.gr | 200 | 3.3451836 | 336080 |
| vc-exact\_037.gr | 198 | 3.2125033 | 343132 |
| vc-exact\_019.gr | 200 | 3.12815104 | 321772 |
| vc-exact\_005.gr | 200 | 2.71463387 | 315270 |

Results for final version of the algorithm:

|  |  |  |  |  |
| --- | --- | --- | --- | --- |
| PACE instances, timeout: 120 seconds, Graph.copy() used, local preprocessing used | | | | |
| instance | |V| | time elapsed | v visited | VC size |
| vc-exact\_197.gr | 1534 | TIMEOUT |  |  |
| vc-exact\_183.gr | 72420 | TIMEOUT |  |  |
| vc-exact\_173.gr | 56860 | TIMEOUT |  |  |
| vc-exact\_167.gr | 15783 | TIMEOUT |  |  |
| vc-exact\_161.gr | 138141 | TIMEOUT |  |  |
| vc-exact\_155.gr | 26300 | TIMEOUT |  |  |
| vc-exact\_149.gr | 26300 | TIMEOUT |  |  |
| vc-exact\_143.gr | 18096 | TIMEOUT |  |  |
| vc-exact\_137.gr | 26300 | TIMEOUT |  |  |
| vc-exact\_135.gr | 26300 | TIMEOUT |  |  |
| vc-exact\_133.gr | 15783 | TIMEOUT |  |  |
| vc-exact\_125.gr | 26300 | TIMEOUT |  |  |
| vc-exact\_123.gr | 26300 | TIMEOUT |  |  |
| vc-exact\_119.gr | 18096 | TIMEOUT |  |  |
| vc-exact\_113.gr | 26300 | TIMEOUT |  |  |
| vc-exact\_109.gr | 66992 | TIMEOUT |  |  |
| vc-exact\_105.gr | 26300 | TIMEOUT |  |  |
| vc-exact\_101.gr | 26300 | TIMEOUT |  |  |
| vc-exact\_099.gr | 26300 | TIMEOUT |  |  |
| vc-exact\_089.gr | 57316 | TIMEOUT |  |  |
| vc-exact\_079.gr | 26300 | TIMEOUT |  |  |
| vc-exact\_075.gr | 26300 | TIMEOUT |  |  |
| vc-exact\_029.gr | 13431 | TIMEOUT |  |  |
| vc-exact\_025.gr | 23194 | TIMEOUT |  |  |
| vc-exact\_021.gr | 24765 | TIMEOUT |  |  |
| vc-exact\_015.gr | 53610 | TIMEOUT |  |  |
| vc-exact\_009.gr | 38452 | TIMEOUT |  |  |
| vc-exact\_153.gr | 29076 | 72.37512947 | 3477157 | 18000 |
| vc-exact\_159.gr | 18096 | 57.28820424 | 6001545 | 11211 |
| vc-exact\_163.gr | 18096 | 57.11087035 | 5974107 | 11211 |
| vc-exact\_171.gr | 18096 | 56.94076883 | 5965978 | 11214 |
| vc-exact\_097.gr | 18096 | 56.69097022 | 6003332 | 11209 |
| vc-exact\_141.gr | 18096 | 56.45237157 | 5979457 | 11209 |
| vc-exact\_181.gr | 18096 | 56.39230369 | 5957171 | 11212 |
| vc-exact\_147.gr | 18096 | 56.11389199 | 5914953 | 11212 |
| vc-exact\_165.gr | 18096 | 56.08975463 | 5932235 | 11209 |
| vc-exact\_145.gr | 18096 | 56.06724477 | 5909598 | 11209 |
| vc-exact\_117.gr | 18096 | 55.97180596 | 6009299 | 11215 |
| vc-exact\_139.gr | 18096 | 55.83879908 | 5927538 | 11211 |
| vc-exact\_127.gr | 18096 | 55.66654798 | 5912651 | 11205 |
| vc-exact\_121.gr | 18096 | 55.6446845 | 5918237 | 11211 |
| vc-exact\_115.gr | 18096 | 55.39813337 | 5937776 | 11210 |
| vc-exact\_129.gr | 15783 | 42.3163103 | 4599245 | 9781 |
| vc-exact\_095.gr | 15783 | 42.12759876 | 4547295 | 9771 |
| vc-exact\_151.gr | 15783 | 41.87351869 | 4564820 | 9777 |
| vc-exact\_179.gr | 15783 | 41.75978476 | 4641970 | 9778 |
| vc-exact\_103.gr | 15783 | 41.69855494 | 4581475 | 9774 |
| vc-exact\_087.gr | 13590 | 31.05602064 | 3423740 | 8421 |
| vc-exact\_107.gr | 13590 | 30.94421945 | 3371890 | 8417 |
| vc-exact\_189.gr | 7400 | 9.926784618 | 1531676 | 4815 |
| vc-exact\_085.gr | 11470 | 9.833961252 | 657526 | 7042 |
| vc-exact\_193.gr | 7030 | 8.807831451 | 1377110 | 4574 |
| vc-exact\_111.gr | 450 | 7.474712614 | 251818 | 425 |
| vc-exact\_039.gr | 6795 | 7.451148908 | 913479 | 4208 |
| vc-exact\_177.gr | 5066 | 5.050961082 | 833918 | 3279 |
| vc-exact\_169.gr | 4768 | 4.142735044 | 747473 | 3079 |
| vc-exact\_191.gr | 4579 | 4.068244902 | 678625 | 2982 |
| vc-exact\_187.gr | 4227 | 3.474310844 | 575457 | 2755 |
| vc-exact\_033.gr | 4410 | 3.419165856 | 422261 | 2733 |
| vc-exact\_007.gr | 8794 | 2.666387988 | 2479619 | 4397 |
| vc-exact\_185.gr | 3523 | 2.366861915 | 412329 | 2291 |
| vc-exact\_175.gr | 3523 | 2.293946653 | 416802 | 2293 |
| vc-exact\_157.gr | 2980 | 1.856767798 | 360371 | 1933 |
| vc-exact\_131.gr | 2980 | 1.709514737 | 314257 | 1929 |
| vc-exact\_011.gr | 9877 | 0.923762236 | 329444 | 4981 |
| vc-exact\_001.gr | 6160 | 0.354561174 | 182969 | 2586 |
| vc-exact\_093.gr | 200 | 0.249105044 | 30742 | 147 |
| vc-exact\_091.gr | 200 | 0.245997406 | 32464 | 150 |
| vc-exact\_083.gr | 200 | 0.239254007 | 29779 | 147 |
| vc-exact\_051.gr | 200 | 0.235643393 | 29416 | 144 |
| vc-exact\_047.gr | 200 | 0.231810394 | 29848 | 144 |
| vc-exact\_067.gr | 200 | 0.226670403 | 29710 | 145 |
| vc-exact\_057.gr | 200 | 0.22643953 | 29654 | 145 |
| vc-exact\_041.gr | 200 | 0.222131201 | 29065 | 142 |
| vc-exact\_081.gr | 199 | 0.222089318 | 29727 | 145 |
| vc-exact\_053.gr | 200 | 0.221672967 | 27940 | 142 |
| vc-exact\_073.gr | 200 | 0.217764297 | 30163 | 145 |
| vc-exact\_045.gr | 200 | 0.205406799 | 26329 | 140 |
| vc-exact\_043.gr | 200 | 0.205267797 | 28932 | 144 |
| vc-exact\_069.gr | 200 | 0.200870814 | 26501 | 142 |
| vc-exact\_063.gr | 200 | 0.197663842 | 28448 | 143 |
| vc-exact\_065.gr | 200 | 0.193519548 | 27291 | 141 |
| vc-exact\_071.gr | 200 | 0.190962708 | 26508 | 138 |
| vc-exact\_077.gr | 200 | 0.187099496 | 26033 | 139 |
| vc-exact\_055.gr | 200 | 0.186776004 | 27307 | 136 |
| vc-exact\_049.gr | 200 | 0.186588082 | 26108 | 139 |
| vc-exact\_059.gr | 200 | 0.184983741 | 26033 | 139 |
| vc-exact\_061.gr | 200 | 0.179798959 | 28381 | 139 |
| vc-exact\_035.gr | 200 | 0.171751311 | 24634 | 138 |
| vc-exact\_031.gr | 200 | 0.167459476 | 26429 | 141 |
| vc-exact\_037.gr | 198 | 0.167441933 | 24406 | 135 |
| vc-exact\_019.gr | 200 | 0.160938689 | 22525 | 132 |
| vc-exact\_005.gr | 200 | 0.138240306 | 21595 | 131 |

1. <https://docs.python.org/3/library/time.html#time.perf_counter> [↑](#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://igraph.org/redirect.html> [↑](#footnote-ref-3)
4. <https://graph-tool.skewed.de/performance> [↑](#footnote-ref-4)
5. <https://docs.python.org/3/library/time.html> [↑](#footnote-ref-5)
6. <https://cairographics.org/pycairo/> [↑](#footnote-ref-6)
7. <https://docs.python.org/3.6/library/profile.html> [↑](#footnote-ref-7)
8. <https://lists.nongnu.org/archive/html/igraph-help/2010-03/msg00078.html>, <https://stackoverflow.com/questions/20245234/delete-vertices-while-preserving-nodes-ids> [↑](#footnote-ref-8)
9. <https://lists.nongnu.org/archive/html/igraph-help/2014-05/msg00069.html> [↑](#footnote-ref-9)
10. <https://ipython.readthedocs.io/en/stable/interactive/magics.html> [↑](#footnote-ref-10)
11. <https://lists.nongnu.org/archive/html/igraph-help/2014-05/msg00069.html> [↑](#footnote-ref-11)
12. <https://stackoverflow.com/questions/3323001/what-is-the-maximum-recursion-depth-in-python-and-how-to-increase-it>, <https://bugs.launchpad.net/beautifulsoup/+bug/1471755> [↑](#footnote-ref-12)
13. <https://docs.python.org/3/library/time.html#time.perf_counter> [↑](#footnote-ref-13)
14. <https://www.wolframalpha.com/> [↑](#footnote-ref-14)
15. <https://igraph.org/python/doc/igraph-pysrc.html#Graph.GRG> [↑](#footnote-ref-15)