On the performance of edge coloring algorithms for cubic graphs

Edvin Berglin
edvinberglin@gmail.com

March 16, 2014

Master's thesis work carried out at
the Department of Computer Science, Lund University.

Supervisor: Thore Husfeldt, thore.husfeldt@cs.lth.se
Examiner: Jonas Skeppstedt, jonas.skeppstedt@cs.lth.se

Abstract

This thesis visits the forefront of algorithmic research on edge coloring of cubic graphs. We select a set of algorithms that are among the asymptotically fastest known today. Each algorithm has exponential time complexity, owing to the NP-completeness of edge coloring, but their space complexities differ greatly. They are implemented in a popular high-level programming language to compare their performance on a set of real instances. We also explore ways to parallelize each of the algorithms and discuss what benefits and detriments those implementations hold.

Keywords: edge coloring, graph algorithm, cubic graph, performance test, performance comparison

Acknowledgements

We would like to thank the makers of Gephi Graph Visualization and Manipulation software, which has been a pain and a blessing when debugging our own software.

Much gratitude is owed to Carl Fagerlin and Paul Rizescu for their support and encouragement, when morale was faltering and in need of a good kick in the pants.

We also extend our deep appreciation to Marcus Klang, for graciously lending all of his gigabytes.

Finally, we thank the international community of video game composers for all their wonderful work. Can't program without some good music!

Contents

Chapter 1
Introduction

In the area of graph theory, the edge coloring problem asks whether it is possible, using only a given number of colors, to assign colors to all the edges of a graph so that for every vertex, none of its adjacent edges share a color. The "colors" are purely abstract entities and do not necessarily translate to any chromatic phenomena.

The problem is related to and inspired by the four color map theorem, which is an instance of *vertex* coloring and one of the first studied problems of graph theory, dating back to at least 1852. It is also one of its most famous results after having been finally proven in 1976. In 1880 — long before any valid proof was discovered for the theorem itself — Tait showed that it could be equivalently phrased as a statement about the coloring of the *edges* of different but related graphs [Tait]. Thus began research into the topic of edge coloring, which continues to see a high amount of scholarly attention today.

The problem has applications to optimal scheduling in several real-life areas, such as round-robin tournaments and fiber optic communication. The case of fiber optics is noteworthy as an instance where our abstract "colors" represent the actual colors of light passing through a physical cable. Perhaps more importantly, edge coloring has implications towards both the ever-famous P vs NP problem, as well as the broader area of graph theory.

In this thesis we implement and examine the running times of three existing algorithms for optimal edge coloring of cubic graphs, measured in seconds rather than asymptotic notation. Before delving into the workings of these algorithms, we give a formal problem statement and definitions of graph theoretical terms that will be used throughout the paper.

1.1 Necessary graph terms and expressions

A graph is *k-regular* if every vertex has degree k. As a special case, 3-regular graphs are called *cubic* graphs.

A *2-partition* of a graph is two disjoint sets V_1 and V_2 where $V_1 \cup V_2 = V(G)$. The set

of edges crossing from V_1 to V_2 are known as the *cut*. A graph is *bipartite* if it has a 2-partition where every edge is in the cut.

A *bisection* is a 2-partition where $||V_1| - |V_2|| \leq 1$.

An edge is called a *bridge*, if its removal will cause a connected component to split into two connected components. A graph is *bridgeless* if it has no bridges, or equivalently, if it has no 2-partition where the cut is of size 1.

A graph is *planar* if it can be drawn in a plane without any of the edges crossing each other except in common vertex endpoints.

Given a graph G and a set of edges F, $G \setminus F$ is the result of removing every edge in F from $E(G)$.

A *matching* for a graph G is a set of edges $M \subseteq E(G)$, such that every vertex in $V(G)$ is adjacent to at most one edge in M. A vertex is *matched* if M contains one of its adjacent edges. M is *perfect* if every vertex is matched.

Given a graph G_1 and a set of vertices $S \subseteq V(G_1)$, graph G_2 is *induced by S on G_1* if $V(G_2) = S$ and $E(G_2) = \{\{u, v\} \mid \{u, v\} \in E(G_1), u \in S, v \in S\}$.

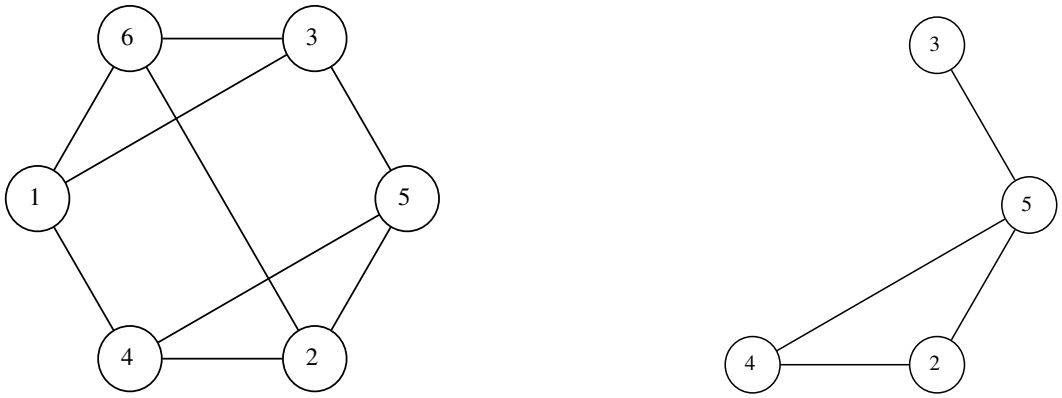

Figure 1.1: Graph G_1 and graph G_2 induced by $\{2, 3, 4, 5\}$ on G_1

1.2 Edge coloring

Given a graph G without self-loops, and a positive integer k, the *edge coloring problem* asks whether there exists a function $F : E(G) \to \{1 \dots k\}$ such that

$$\exists \{v, u\}, \{v, w\} \in E(G) \implies (u = w \text{ or } F(\{v, u\}) \neq F(\{v, w\})) \tag{1.1}$$

Assuming its existence, the function F is then called a *k-edge-coloring of the graph G*, or simply an *edge coloring* when G and k are implicit. A graph may have several *k-edge-colorings* for any given k. Related problems that we will deal with are:

- to count how many edge colorings exist (*count*).
- to find an edge coloring (*find*).
- to find all edge colorings (*find-all*).

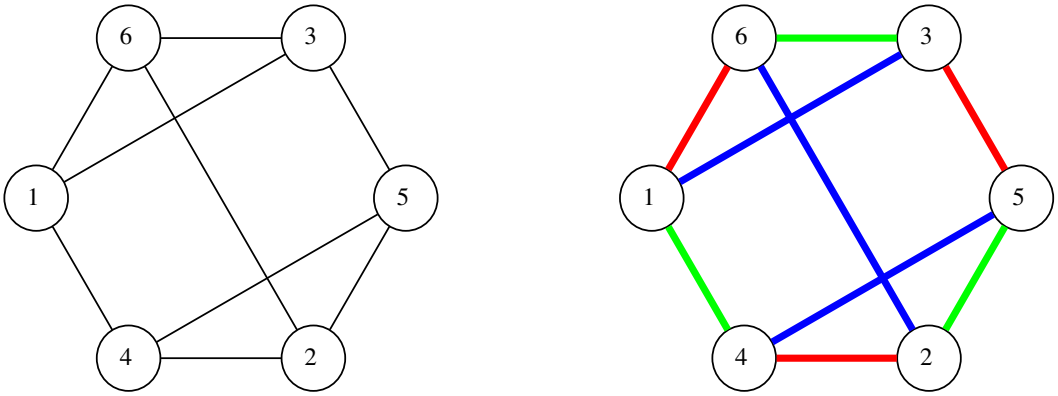

Figure 1.2: Graph G and a 3-edge-coloring for G

Note that the problems are hierarchically ordered; solving *find-all* implies a solution to both *find* and *count*, either of which imply a solution to *edge coloring*.

In this paper we consider colorings F_1 and F_2 to be equal if there exists a bijection $B : \{1 \dots k\} \to \{1 \dots k\}$ such that $\forall e : B(F_1(e)) = F_2(e)$. That is to say, we do not consider the actual value of a color to carry any meaning.

A function $P : S \to \{1 \dots k\}$ is a *partial edge coloring* of G if $S \subseteq E(G)$ and P satisfies the condition in (1.1). P is a partial edge coloring even if there exists no (proper) edge coloring F such that $\forall e \in S : P(e) = F(e)$. However, if no such F exists, P may become *invalidated* as more edges are added to S and the necessary condition (1.1) is forced to be broken.

The smallest k for which an edge coloring exists is called the *chromatic index of* G, $\chi'(G)$. Let $\triangle(G) = \max_{v \in V} \deg(v)$. We may then trivially observe that $\triangle(G) \leq \chi'(G)$; there exists a vertex with $\triangle(G)$ adjacent edges and these edges must be assigned $\triangle(G)$ different colors. In his hallmark 1964 paper, Vizing [Viz64] proved that $\chi'(G) \leq \triangle(G) + 1$. Graphs with $\chi'(G) = \triangle(G)$ are said to be of *class 1* while graphs with $\chi'(G) = \triangle(G) + 1$ are of *class 2*. Therefore, determining the class of a graph is the same thing as solving the edge coloring problem for $k = \triangle(G)$.

The function F is called an edge coloring simply because in a drawing of a graph, it is natural to illustrate the range $\{1 \dots k\}$ as k different physical colors and to draw the edges using the color associated with their respective function value, as seen in figure 1.2.

The edge coloring problem is NP-complete for arbitrary graphs [Holy] but trivial for graphs with $\triangle(G) = 2$. Such graphs are merely a collection of paths and cycles. Paths and even cycles may always be colored using 2 alternating colors, while odd cycles require 3. In this thesis we will mainly concern ourselves with cubic graphs, and it remains NP-complete to determine the class of an arbitrary cubic graph [Holy]. We also consider only simple and undirected graphs except when otherwise stated.

1.3 Path decomposition

Given a graph G, a *path decomposition* is a list (or *path*) of sets $X_1 \dots X_l$, called *bags*, where every $X \subseteq V(G)$, the length l is a positive integer, and the following conditions are met:

1. Every node $v \in V(G)$ belongs to some bag.
2. If two bags X_i and X_j both contain a specific vertex for some $i < j$, then so does X_k for all $i < k < j$.
3. For every edge $\{v, u\} \in E(G)$, there exists a bag that contains both v and u.

The *width* of a path decomposition is defined as $\max_i |X_i| - 1$. The smallest width for all legal path decompositions of a graph is known as the *pathwidth* of that graph, and path decompositions of this width are *optimal*. It is NP-complete to determine the pathwidth of a graph [Leng][Kinn].

A path decomposition is *nice* if $X_1 = \{\}$ and for $i > 1$:

$$\exists v \in V : (X_i = X_{i-1} \cup \{v\} \text{ or } X_i = X_{i-1} \setminus \{v\}) \tag{1.2}$$

That is, every bag from X_2 starts with the previous bag and either adds one new vertex ("introduce bag"), or removes one vertex ("forget bag"). It is straightforward to transform any path decomposition into a nice path decomposition of length $2n$ without increasing its width, in linear time [BR]. In the remainder of this paper we will consider only path decompositions that are nice and have length $2n$. Figures 1.3, 1.4 and 1.5 show an example graph and two different nice path decompositions of it, the first of which is optimal.

Path decompositions are a special case of *tree decompositions*, which are defined by similar rules but connecting the bags as a tree rather than a list.

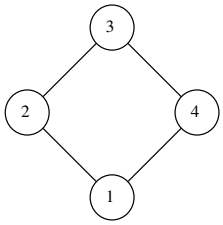

Figure 1.3: Graph G

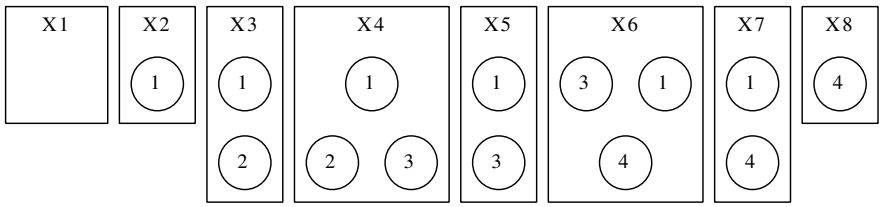

Figure 1.4: Nice path decomposition of G, with width 2

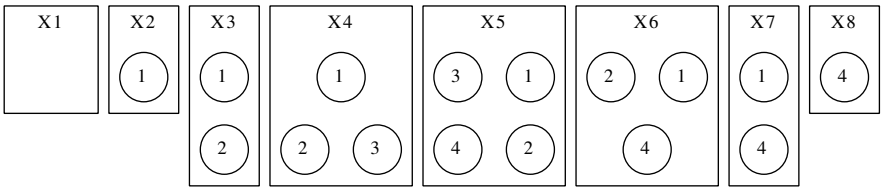

Figure 1.5: Nice path decomposition of G, with width 3

Chapter 2
Approach

The goal of this thesis is to implement and test some existing algorithms for edge coloring and the related problems outlined above. We hope this may provide the insight to make an informed decision for a reader who wishes to practically solve this problem in a real setting.

We will examine three different algorithms for cubic graphs and $k = 3$. One is due to Kowalik [Kow], while the other two stem from the same paper by Golovach, Kratsch and Couturier [GKC]. Two of them exploit certain properties of cubic graphs and therefore do not work on graphs of higher degree. The last one is applicable to graphs of any degree, but not considered competitive against other generalized algorithms except when k is small.

Both papers employ a special asymptotic notation: for a super-polynomial $f(n)$, $O^*(f(n))$ is the same as $O(f(n) \cdot p(n))$ where $p(n)$ is a polynomial. These polynomials are hidden because they may be implementation-specific, and because the growth of p may be dwarfed by that of f for sufficiently large n. We adopt that notation here.

Following are brief descriptions of the workings of each algorithm; their full details may be found in their respective original papers. Details specific to our implementations are found in chapter 3.

2.1 EnumColors

The first algorithm of Golovach *et al.* [GKC] is a branching algorithm that enumerates all edge colorings. It is very uncomplicated and the only one to attack the problem by actually trying to assign colors to the edges, backtracking when it runs into a dead end as well as after finding and printing a valid edge coloring.

Consider the graph G and a partial edge coloring C where S is the set of colored edges. As long as there are vertices with only one uncolored edge, those edges are colored with the only available color. When no such vertex exists, construct the graph $H = G \setminus S$; if H contains only cycles it is straightforward to color the edges not in S, otherwise try to reduce

its connected components down to cycles. Choose an edge $e_1 \notin S$ adjacent to $e_2 \in S$, and create two new partial edge colorings C_1 and C_2 by coloring e_1 with the two colors not assigned to e_2. Recurse on the mutually exclusive C_1 and C_2.

The algorithm runs in $O^*(2^{5n/8}) = O^*(1.542^n)$ time and $O(n^2)$ space, discounting any space used for remembering any found colorings since there may be exponentially many. The quadratic space is because of the new graph H that is constructed at every recursive call; there may be up to $O(n)$ different H graphs alive at any time. The time complexity is not competitive with those of the other two algorithms for solving the edge coloring problem. We include EnumColors in spite of this fact, because of its simplicity to implement and due to it being presented in the same paper as CountColors.

2.2 CountColors

The second algorithm by Golovach *et al.* is a dynamic programming algorithm to count the number of edge colorings in $O^*(1.201^n)$ time, which is the asymptotically fastest out of all three. At the time of this writing, it is also the fastest publicly known algorithm for the cubic edge coloring problem. The space complexity of CountColors is $O^*(1.201^n)$ as well, which makes it the only algorithm to require more than polynomial space. To run, it requires a nice path decomposition of the graph, and builds upon the work of Fomin and Høie [FH] which in time linear to n produces a path decomposition of upper bounded width p:

$$p + 1 \leq (1/6 + \epsilon)n, \epsilon > 0 \tag{2.1}$$

After a path decomposition is constructed, the graph is torn down and rebuilt piece-by-piece according to the order that vertices appear in the decomposition. That is, let G_i be the graph induced by $\bigcup_{1 \leq n \leq i} X_n$ on G. Define a *characteristic* as a pair (S, σ) where S is a function $S : X_i \to \mathcal{P}\{1 \ldots k\}$, and σ is an integer corresponding to the number of edge colorings F of G_i (which are partial edge colorings of G) that exist under the two constraints:

$$\forall v \in V(G_i) : \forall \{v, u\} \in E(G_i) : F(\{v, u\}) \in S(v) \tag{2.2}$$

$$\forall v \in V(G_i) : |S(v)| = \deg_{G_i} v \tag{2.3}$$

S therefore contains information about which colors are "taken" for the vertices in X_i. As vertices are forgotten, characteristics may be merged into each other and have their σ values summed.

Then, update a table containing every characteristic where $\sigma > 0$. On introducing a vertex v, a new characteristic is quickly formed from one in the previous table by creating a partial edge coloring for the set of the edges $\{v, u\}$, and then making sure that the assigned color is not already in $S(u)$.

These tables are the cause of the space complexity; each of the $2n$ tables may reach a size exponential to the graph's current size. A table may contain up to $k! = 6$ times as many entries as the previous table, if none of the partial edge colorings are invalidated by the insertion of the new vertex and none of them merge into the same characteristic with increased σ. The total number of edge colorings is then the sum of the σ over all characteristics in the table for $G_{2n} = G$. Once the counting algorithm finishes, if the tables

for each intermediary step are kept, it is straightforward to produce an edge coloring by backtracking through the tables.

In itself the CountColors algorithm is correct for any valid nice path decomposition, but the bound on the width p due to Fomin and Høie (2.1) is necessary to derive the bound on its running time. We store functions S mapping from X_i to $\mathcal{P}\{1 \dots k\}$ of which there are at most $(2^k)^{|X_i|}$, and $p+1$ is defined to be the maximum cardinality of any bag. Furthermore, the rules we have placed on these functions will lower the base from 2^k; e.g. placing a color in the set $S(v)$ must also place that color in $S(u)$ for some neighbor u of v. The actual base is $\binom{k}{k/2}$, which for $k = 3$ is 3. Due to this binomial coefficient base, the algorithm is only considered competitive when k is very small. With the width bound (2.1), we achieve the bound on the size of a single table, $O(3^{(1/6+\epsilon)n})$. There are $2n$ tables and the time to test if partial edge colorings are invalidated depends only on k. Thus we arrive at the time complexity:

$$O(3^{(1/6+\epsilon)n} \cdot 2n) \to O^*(1.201^n) \text{ as } \epsilon \to 0 \tag{2.4}$$

Every choice of ϵ is tied to an integer n_ϵ, and graphs of size $n < n_\epsilon$ are not guaranteed to be path decomposable using this method. The value of n_ϵ is not known exactly, but is bounded by:

$$n_\epsilon \leq \frac{4}{\epsilon} \cdot \ln(\frac{1}{\epsilon}) \cdot (1 + \frac{1}{\epsilon^2}) \tag{2.5}$$

Future advances to finding path decompositions of smaller width (in sub-exponential time) would automatically improve the time complexity of CountColors. The width $(1/6 + \epsilon)n \to 0.167n$ is an upper bound on the pathwidth for all cubic graphs, and there exists types of cubic graphs with pathwidth at least $0.082n$ as noted in [FH]. Hence, the upper-bound time complexity for this approach cannot be improved past $O^*(3^{0.082n}) = O^*(1.094^n)$.

2.3 Kowalik

The method described by Kowalik in [Kow], referred to herein simply as *Kowalik*, in its original form only answers the yes/no edge coloring problem, but its author remarks that it is straightforward to extend it to yield an edge coloring. It works on a simple principle: if we can find a perfect matching M for G, $G \setminus M$ is then a graph of maximum degree 2, for which it is easy to determine the class as noted above. A perfect matching M that leaves no odd cycles in $G \setminus M$ is called a fitting matching. The algorithm begins by reducing the graph to so-called *semicubic* graphs, which are graphs where most vertices have degree 3 but degree 2 vertices are allowed to exist according to some rules. For an overview of the reduction rules, we refer to Kowalik's original paper.

The graph reduction includes three cases where the algorithm is forced to branch; G is 3-edge-colorable if at least one of G_1 and G_2 is 3-edge-colorable, where G_1 and G_2 are the result of removing some vertices and edges from G plus possibly adding some new edges. This has the implication that both G_1 and G_2 may at some point further reduce into the same graph G_3, which is in contrast to the branching behavior of EnumColors. Not every reduction rule removes a vertex, but every reduced graph is smaller than G in terms of $n + m$.

To find a fitting matching, keep two graphs G_0, G and a matching M on G_0. G_0 is a semicubic graph, G is a graph where $V(G)$ is the set of vertices not matched in M, and $E(G) \in E(G_0) \setminus M$. The graph G_0 is not modified in this stage, while G progressively shrinks, reaching the empty graph when M is perfect. At this point we require two more definitions:

- A *switch* is a 4-path $xvuy$, such that $xvuy$ is a connected component in G and x, y have degree 2 in G_0 while v, u have degree 3.
- Matching M is *semi-perfect* if all the connected components in G are switches.

The search is done in two stages; generate a semi-perfect matching through a method that has exponential-time complexity in itself, in the second stage that matching is manipulated into a fitting one. The structure of the switches allows the second part to be performed through iterative improvements. If the second stage is successful, it means we have determined that the graph G_0 is class 1. If we cannot find a fitting matching for G_0, report the failure to the graph reduction algorithm which then continues the search for a different semicubic graph.

To produce an edge coloring once a fitting matching M is found, first color all edges in M with color 1 and use colors 2 and 3 as necessary for the paths and cycles in $G \setminus M$. Then apply the reverse graph reductions and update the edge coloring along the way. Some reverse reductions require a specific transformation of the edge coloring, while others allow a the re-added edges to be colored greedily. Regardless, every update is performed in constant time and the entire process to produce an edge coloring takes only $O(m)$ extra time.

Kowalik may be applied to graphs that have edges of multiplicity higher than 1, and some of its reductions may cause edges to increase in multiplicity. Graphs also do not need to be cubic; there may be any number of vertices v in the starting graph with $\deg v \in \{0, 1, 2\}$. This makes Kowalik more general than the other two algorithms. It is however no easy task to modify it to count or list all edge colorings, because some reductions hide necessary information and because the branching reduction rules to not create mutually exclusive sub-problems.

It runs in $O^*(1.344^n)$ time, which was the previous record until CountColors was discovered. The analysis for the time complexity is very complicated and will not be reproduced here. The space complexity is linear if graph reductions are reversible; store only a single graph plus an $O(n)$ size stack containing information on how to reverse the reductions, which are all constant-time operations.

Algorithm synopsis

All three algorithms may be used to solve more than just the edge coloring problem itself. EnumColors is able to enumerate all edge colorings, and is therefore trivially able to count them or produce just one of the edge colorings. Kowalik, after minor modification, is able to return an edge coloring, but not able to count them all. CountColors may, through extensive backtracking, enumerate all edge colorings, although that in itself has an exponential time complexity. To produce a single edge coloring would require only $\Theta(n)$ extra time, provided that one can find specific characteristics in $O(1)$ time. This is achievable if the

tables have constant-time access operations, or if the characteristics store a pointer to any of their "predecessors". It is however important to note that CountColors has to finish counting all colorings before finding one of them. See table 2.1 for a summary of what algorithm solves which problems.

	edge coloring	*find*	*count*	*find-all*
EnumColors	Yes	Yes	Yes	Yes
Kowalik	Yes	Yes	No	No
CountColors	Yes	Yes (backtracking)	Yes	Yes (backtracking)

Figure 2.1: Problems solvable by algorithm

CountColors currently has the best time complexity of any cubic edge coloring algorithm. However, as stated it is attached to a smallest graph size n_ϵ, determined by the choice of ϵ. Hence, to achieve a time complexity equal to or better than Kowalik's $O^*(1.344^n)$, we must use $\epsilon \leq 0.102446$, and can thereby only guarantee that the algorithm works for graphs of size

$$n \geq 8565 \tag{2.6}$$

The algorithm descriptions of both EnumColors and Kowalik use language of the form "find an X in the graph", where X may be e.g. a cycle, a vertex of a certain degree, or some other local substructure. As the graph may contain several X at any time, this makes them both Las Vegas algorithms; they make randomized choices over the input, but always terminate and always give the correct result. The graph bisection algorithm uses similar language, and often there are several different bisections of equal cut size which can lead to different path decompositions of equal width. The running time of CountColors depends on how early we are able to invalidate dead-end partial colorings, which is determined by the order that vertices are introduced in the path decomposition. Hence, CountColors is a Las Vegas algorithm as well when using the graph bisection algorithm. That is not necessarily true for different methods of finding path decompositions.

2.4 Related research

Despite the NP-completeness for graphs with $\triangle(G) \geq 3$, some types of graphs are known to be of class 1. Bridgeless planar cubic graphs are always $\triangle(G)$-edge-colorable [Tait], as are bipartite graphs of any degree [Konig] as well as graphs where at most two vertices have degree $\triangle(G)$ [Viz64]. A cubic bridgeless graph of class 2 is known as a *snark*. Bridgelessness, planarity, bipartiteness and number of max-degree vertices are all properties that may be tested for in $O(n)$ time [Tarj][HT][KT], so in a realistic setting these tests should be performed before any exponential-time algorithm is attempted. But as we are primarily interested in the efficiency of the tested algorithms and less so in any actual output, we forego performing these tests.

Furthermore, planar graphs of maximum degree ≥ 7 are of class 1, as shown by Sanders and Zhao [SZ]. Planar graphs of class 2 are known for maximum degrees 2 through 5; for degree 2 any odd cycle is class 2, and for degrees $3, 4, 5$ class 2 graphs may be constructed from the platonic solids, as demonstrated by Vizing [Viz65]. It remains an open problem whether there are any class 2 planar graphs of maximum degree

6. It is also known that k-regular graphs of odd size n must be of class 2 for any k [Hara]. However, since by the handshaking lemma $\sum_{v \in V} \deg v = nk$ must be even, it cannot be the case that both n and k are odd. Hence this result has no bearing on the case that $k = 3$.

A polynomial-time algorithms is known for finding edge colorings with $\triangle(G)+1$ colors, by Misra and Gries [MG]. It may be employed if the graph is known to be of class 2, or if using more than the minimum amount of colors is not a concern. But if the colors are considered to be a scarce resource, the case that $\triangle(G) = 3$ is of special interest; coloring class 1 graphs with 4 colors is a 33% waste. For class 1 graphs of higher degree, the ratio of waste $\frac{\triangle(G)+1}{\triangle(G)}$ diminishes.

For optimal coloring of graphs already known to be of class 1, Cole, Ost and Schirra showed in 2001 [COS] how bipartite graphs may be optimally colored in near-linear time. In 2008 Cole and Kowalik [CK] discovered a linear-time algorithm for planar graphs of $\triangle(G) \geq 9$.

The fastest known algorithm for optimally coloring arbitrary graphs is due to Björklund *et al.* [BHK] and runs in $O(2^m m^{O(1)}) = O^*(2^m)$ time and exponential space, where m is the number of edges in the graph. For the cubic case, $m = 3n/2$, giving the complexity of $O^*(2^{3n/2}) = O^*(2.828^n)$ which is not expected to be competitive with the three chosen algorithms for anything but trivially small graphs.

Chapter 3
Software specifics

For this project Java was chosen as programming language, mostly for its ease of development and familiarity to the author. The conscious decision was made to rely as little as possible on third party software, in order to be able to release the code in full without any software license hassle. Not counting basic elements from the Java SE Class Library, all code was therefore written from the ground up.

The result is five different software packages: one for representing and manipulating graphs, one to find path decompositions, and finally one each for every main algorithm. Parallelized versions were created for each algorithm, mostly as proofs of concept to demonstrate that speed-up via parallelization is viable. It is an interesting challenge in itself to load-balance these multi-threaded implementations.

The code will be available as-is at the Lund University Publications website, released under the BSD-3 license.

3.1 Graph

As we are working with graph algorithms, the `graph` package is the natural central tool for the rest of our software. Apart from being a holder class for sets of `Vertex` and `Edge`, `Graph` provides methods for various graph manipulations and searches such as finding connected components that are cycles, and creating $G \setminus E^*$ from set of edges E^*. Algorithms Kowalik and EnumColors both require the ability to find cycles.

A `Vertex` is defined only by its unique integer id number and contains its own `Edge` set of all adjacent edges. An `Edge` is defined only by its two vertex endpoints; multiple edges between the same pair of vertices is represented as a single `Edge` with an increased multiplicity.

An important feature of this package is that it allows removed vertices to be easily reinserted with all their old edges intact, provided that they are reinserted in the reverse order of their removal. This is important for the iterative reconstruction of G in CountCol-

ors. It is also a crucial feature in order to achieve the $O(n)$ space complexity of Kowalik; if the reductions were not possible to undo, the graph would have to be copied for every branching, potentially resulting in exponential memory use.

This package features a class `Generator` for generating `Graph` instances. It can recreate a graph from the output of its `toString()` method, and it may be used to create randomized graphs. The randomization is more powerful than is needed for this project; multigraphs, non-regular graphs and graphs of higher maximum degree than 3 are able to be created. When generating k-regular, simple graphs of size n, the returned graph is randomly chosen from all such graphs with equal probability. The graph generation is a Las Vegas algorithm running in $O(e^{k^2/2}nk)$ expected time, which is linear in n for any constant k [MKW]. This algorithm is not guaranteed to terminate, but that is of no practical concern since it is not the algorithm being investigated.

Since one of our most heavily employed data structures is the hash set, the `Generator` class has the ability to set a universal "hash seed" affecting the hash value of every `Vertex` and `Edge`. When iterating through the elements of a Java `HashSet`, the order in which they appear are determined by the element hashes. Consequently, this helps demonstrate that the success of our implementations do not hinge on vertices or edges being accessed in a specific order.

Hash seeds also help us de-randomize our algorithms, because in our implementation the random choices depend on the order that elements are returned from hash set. If we control the hash values, we control the order and achieve a deterministic running time.

Finally we include a non-essential class `Snark`, which holds some pre-generated snarks of up to 50 vertices. As snarks are of class 2 by definition, they may be of value for testing or demonstrative purposes.

The essential parts of `graph` make up a total of 739 lines of code.

3.2 Pathdecomp

The package `pathdecomp` takes a `Graph` and a `double epsilon` to create a path decomposition, required by CountColors, according to the method described by Fomin and Høie [FH]. Its main class `Pathdecomp` is a very simple data structure, essentially being a wrapper class for `ArrayList<Set<graph.Vertex>>` from the standard library and the `graph` package. Since path decompositions have width $< (1/6+\epsilon)n$ and length $2n$, this yields an $O(n^2)$ space complexity which could have been improved to $O(n)$ by exploiting the fact that every bag differs from the previous one by only one vertex. However, as we will see in a later chapter, neither the time consumption to generate a path decomposition, nor its internal memory structure, act as any form of bottleneck for the CountColors algorithm. Consequently no effort was dedicated to optimize this package.

Fomin's and Høie's method starts out by referencing the algorithm by Monien and Preis [MP] to bisect a graph such that the size of the cut is no larger than $(1/6+\epsilon)n$. This is contained in the class `Bisection` and two auxiliary classes `RBGraph` and `RBVertex` (for *red-black graphs*, a special structure employed in their algorithm). Preis provides a software library called PARTY [PARTY] which can be used to find this bisection. Due to our stated goal not to depend on any third party software, the algorithm was re-implemented from scratch. Our implementation has not been as thoroughly tested as one would like,

but is believed to be correct as it has not failed to bisect any of our randomly generated graphs that fulfil the size condition $n > n_\epsilon$. It should be fairly straightforward to substitute our `Bisection` implementation for PARTY, if one desires more well-tested and robust code.

The main path decomposition algorithm further references papers by Kinnersley [Kinn] and Ellis *et al.* [EST], demonstrating how to create a path decomposition of a tree graph, of width $\log_3 n$. This is implemented as class `Layout`.

3.3 EnumColors

As we consider permutated colorings equal, we start off by selecting a random vertex v, and coloring any two of its adjacent edges with colors 1 and 2. This cuts down every $k! = 6$ permutations of valid edge colorings into 1. When used to count all colorings, this speeds up the program by factor six. When used to enumerate all colorings, if the goal is to enumerate every permutation, the equivalent permutations of a coloring are trivially implied by that coloring itself and do not need to be computed separately.

EnumColors was the easiest algorithm to implement. In itself its code is straightforward, as most of its programmatical difficulty lies in the graph search methods, part of the `graph` package. It also requires no special data structures other than a `Map<Edge,Color>` to represent the partial edge coloring; `Edge` is provided by the `graph` package and Color may be represented as an integer. The `EnumColors` class encompasses 210 lines of code.

3.3.1 Parallelization

The parallel version gains a central object through which all threads communicate. A working thread follows same code as in the serial version, but on a branching it sends one of the branches C_2 to the communication object where it is picked up by any idle thread. After branch C_1 is computed, we await the result of C_2 from another thread, or start computing C_2 ourselves if no other thread has done so.

As we create a copy of the partial coloring on every branch, this version uses more memory than the single-threaded one. Every map is of size $O(m)$. But every thread may branch at most m times before going back to calculate or await the result of the other path in the branch. There are not more than t active threads where t is set at the start of the program and does not depend on the graph. Thus our memory use is $O(tm^3)$, $m = 3n/2$, which is still polynomial in n for any constant t.

This version uses 126 additional lines of code.

3.4 Kowalik

The main `kowalik` package contains three classes; the main program `Kowalik` plus two support classes `Matching` and `Switch`. `Matching` is essentially a `HashSet<Edge>` extended with some basic convenience methods, while a `Switch` holds only four vertices and a static method to find them.

Since branching graph reductions create subproblems that are not mutually exclusive, it is tempting to use memoization to avoid repeat work. We experimented with some approaches to this idea, but ultimately did not find a solution that was helpful. The memory usage increased severely as old graphs needed to be stored, and running time did not seem to improve but actually increased for many large graphs. We hypothesize the time increase is in no small part because of the constant re-hashing of graphs; the standard `hashCode()` method for Java `Set` is an $O(n)$ operation. While it could certainly have been overridden with a constant-time hash method, we still did not experience any benefits from memoization and the idea was therefore abandoned.

The necessary classes of `kowalik` use 492 lines of code.

3.4.1 Parallelization

Kowalik has branches in two different parts of the algorithm, both in the graph reductions and in the search for a semi-perfect matching. We create two parallel versions of Kowalik, one for each of these two branch types. Both of them use the same simple architecture as the parallel EnumColors, which keeps a central communication object and asks any sleeping thread for help when the code branches. Refer to the version that off-loads work on graph reductions as `KowalikParallel`, and the other as `KowalikParallel2`.

It is an important point that both versions parallelize only a single aspect of the original algorithm. In `KowalikParallel` several threads search for semicubic graphs, but when one is found that thread will individually handle the search for a fitting matching for that graph. Conversely, `KowalikParallel2` has a single thread performing all graph reductions, and employs its multi-thread capacities only when searching for a fitting matching. All of its threads therefore operate on the same G_0, which simplifies the implementation as there is no risk for concurrency errors when the main thread performs its graph reductions.

This design dichotomy provides an obvious observation: `KowalikParallel2` should be best suited when the semicubic graphs are few but large, while the reverse should hold for `KowalikParallel`. It should be possible, but architecturally more complex, to write a third parallel version that inherits the strength of both. We do not create such a program, as our stated intention is to write architecturally simple parallelizations to demonstrate that speed-ups are possible. Still, we hope our parallel test results can help the optimization efforts of such an implementation.

Like parallel EnumColors, the copies of the working instance must be created at every branch. For `KowalikParallel` this is the graph G. There are up to $O(n)$ graph reduction branches before a graph is reduced to semi-cubic form, and each copy is of size $O(n)$. This gives the same $O(tn^2)$ space complexity.

For `KowalikParallel2`, the data that must be copied is the current matching M of G_0 plus the unmatched graph $G = G_0 \setminus M$. That yields the same space complexity, as the matching and the unmatched graph together are not larger than G_0.

`KowalikParallel` and `KowalikParallel2` use an extra 161 and 126 lines of code, respectively.

3.4.2 Finding an edge coloring

The single-threaded Kowalik was extended to produce an edge coloring. This class is called `KowalikPrint`. As touched briefly on earlier, our graph package represents multiple edges between the same pair of vertices as a single `Edge` object with a multiplicity attribute. This is a sensible solution when existence is their only relevant property and they do not need to be told apart. As only the reductions in the Kowalik algorithm can lead to these edges, it makes no difference for EnumColors or CountColors.

In the coloring-finding version of Kowalik, this design becomes an issue; double and triple edges will now have multiple colors, and as the graph reductions are reversed those color sets need to be carefully split apart to adhere to the rules of edge coloring. This caused a high amount of code clutter, so it was decided to leave it incompatible with the multi-threaded versions. The clutter should be avoidable with a differently designed graph package that allows multiple `Edge` objects between the same vertices.

Unlike what we do for our parallel implementations, the original `Kowalik` class was not modified to easily let `KowalikPrint` inherit it as a super-class. Instead the entire `Kowalik` class was copied and modified. `KowalikPrint` uses 526 lines of code, which should be able to be drastically reduced with better forethought in the design of the `graph` package.

3.5 CountColors

While we work with a cubic starting graph G, the degree of a freshly introduced vertex in G_i may range from 0 to 3 depending on how many of its neighbours in G have previously been introduced. When introducing a vertex of degree 1, there are three ways to color its single edge. Degree 2 vertices have $3 \cdot 2 = 6$ such possibilities and degree 3 vertices have $3! = 6$. Each possibility needs to be tested against every characteristic in the previous table. Since permutations of a coloring are of no interest, they are eliminated early by a method that we call *forcing*: the first time a vertex of degree 2 or 3 is introduced, we consider only one of its six possibilities. This is the same idea as the preparatory work for EnumColors which speeds up the program by factor six. In this case it also cuts the table size by factor six, which is a simple but important strategy to combat the exponential memory use.

Because the tables will still grow extremely large, we keep only the most recent table. Characteristics are removed from the previous table as soon as possible, further limiting the amount of memory used at any given time. This decision prevents our implementation from being used to find an edge coloring.

We represent the table as a `TreeMap<Characteristic, Characteristic>`, with every key mapping to itself. Fundamentally, the table implements a set of `Characteristic` objects, but in case of duplicate insertions we require access to the already included element in order to modify its σ value. Java `TreeSet` does not provide that functionality. Furthermore, as a `TreeSet` is itself backed by a `TreeMap` where every value is `null`, this choice does not increase our memory usage.

A static `TreeMap` uses slightly more overhead data and has slower element access than does a `HashMap`, but the former was favored because `HashMap` is backed by an

array that may neither grow nor shrink. This structure causes problems when the tables contain very many elements, in three ways:

- When a `HashMap` grows past its current capacity, it needs to allocate a new, larger array and copy its contents from the old one. Hence a `HashMap` may peak at higher memory usage, when two very large arrays are held in memory simultaneously.

- Upon a vertex introduction to create G_i, the table t_i may grow up to six times the size of the previous t_{i-1} or shrink to a fraction of it. If t_i is allocated too large it may prevent the allocation of t_{i+1} and cause the program to crash. On the other hand, if the new table t_i is too small it may cause a high amount of re-hashing, leading to a worse time performance than a `TreeMap`.

- Because a `TreeMap` is able to dynamically shrink, if table t_i is close to the maximum size our memory can hold, we may still be able to create table t_{i+1} of equal size. This only works if t_i is depopulated at the same pace that t_{i+1} is filled, however.

While the choice of `TreeMap` may negatively impact the average running times, it should enable us to run the algorithm for graphs with a somewhat thicker path decomposition. However, as `TreeMap` is implemented as a red-black tree, it will automatically perform tree rotations to stay balanced between every element removal. Our program will always iteratively remove every element, wherefore the balanced structure is not interesting to us once the removing process has started. These balancing operations are a waste of time for our purposes.

An attempt was made to reduce the memory footprint of `CountColors` by having the `Characteristic` class implement the `Serializable` interface. A serialized object is transformed into a byte array, which under the right circumstances can be significantly smaller than the original object. Our `Characteristic` objects may reach a few hundred kilobytes in size and are thus a prime candidate for serialization. The table of characteristics would then store these byte arrays instead, with automatic serialization and deserialization to maintain the outward appearance of storing `Characteristic` objects as normal. Unfortunately our serialization efforts ultimately failed, for reasons that not fully understood but appear to be rooted in the design of the `graph` package.

The `CountColors` class and its necessary help classes (including the whole `pathdecomp` package) use an approximate 2619 lines of code in total, and was the most complex to write out of all the three. However, the heaviest class `Bisection` in itself makes up 976 lines and its two supplementary classes 404 lines – work which could have been avoided by using the existing software PARTY.

3.5.1 Setting epsilon

As the time complexity of CountColors is exponential to the width of the path decomposition, it is important not to let the width grow too large. We start out by finding a "good" ϵ through the bisection method, starting from end points 0 and 1. Define $\hat{n}_\epsilon$,

$$\hat{n}_\epsilon = \frac{4}{\epsilon} \cdot \ln(\frac{1}{\epsilon}) \cdot (1 + \frac{1}{\epsilon^2}) \tag{3.1}$$

as the highest value n_ϵ can take (2.5). ϵ is *good* if $\hat{n}_\epsilon \leq n \leq 1.05\hat{n}_\epsilon$. We thereby guarantee an ϵ for which a path decomposition can be found and, because we desire a small ϵ and $\hat{n}_\epsilon$ grows as ϵ shrinks, we also ensure that n is not greatly larger than $\hat{n}_\epsilon$.

The bound n_ϵ provides a smallest graph size for which a path decomposition is guaranteed to be found. This does not imply failure for graphs smaller than n_ϵ. Therefore the program includes a flag `aggressive` which, if set, will cause the program to attempt to find a "better" ϵ_2. After a good ϵ_1 has been found, simply bisect for ϵ_2 between 0 and ϵ_1 and try to produce a new path decomposition at every step of the bisection. We stop after five consecutive values of ϵ_2 that did not amount to a valid path decomposition, and return the composition of smallest width.

3.5.2 Parallelization

Because CountColors lacks code branching, it is parallelized by a different principle than EnumColors and Kowalik. When a forced table of characteristics reaches a certain size, it is broken up into several smaller tables of equal size which are assigned to one thread each. These threads behave as if they are all individually solving the whole problem, unaware of each other. Apart from synchronizing to always work on the same induced graph G_i, they are all running the serial version of the code (where the force has already occured) on their respective tables. The final result for the whole graph is simply the sum of their individual results.

Because tables are broken up and no longer representing the whole problem, it may happen that two edge colorings F_1 and F_2, which are both compatible with some characteristic, end up in tables belonging to different threads. As any pair of threads are unaware of each other, both will need to store equivalent characteristics in their respective table, whereas the serial version would store a single characteristic with a higher σ value. Hence, the parallel version may be more memory-expensive.

At the time of the initial splitting into several tables, it is not possible to know which partial colorings for G_i will end up being invalid for graph G_j, $j > i$; if it were, just set $j = 2n$ to purge all invalid colorings of G immediately. Hence threads may end up with greatly differing table sizes. Because the threads run without any regard to each other, the tables are never rebalanced and some threads may perform disproportionately large amount of work. This is an obvious area where the code may be improved. Rebalancing the tables could have the beneficial side-effect of merging equivalent characteristics from different tables.

The parallelized version uses around 170 extra lines of code.

Implementation synopsis

The implementation details above give us an updated table 3.1 of which problem is solved by which algorithms.

	edge coloring	*find*	*count*	*find-all*
EnumColors	Yes	Yes	Yes	Yes
Kowalik	Yes	Yes	No	No
CountColors	Yes	No	Yes	No

Figure 3.1: Problems solvable by algorithm implementation

Chapter 4
Testing

4.1 Computer hardware

We employ four physical machines for our tests.

Machine 1 has an Intel Core i7 2930MHz Quad-core CPU, with 16 GiB RAM. Due to hyper-threading, the four physical cores make for eight logical ones.

Machine 2 has equal hardware specs to Machine 1.

Machine 3 uses two Intel Xeon CPUs E5-2620 0 at 2.00Ghz with 256 GiB RAM. Each CPU has six physical cores, which are hyper-threaded to a total of 24 logical cores.

Machine 4 has an Intel Core2Quad Q9400 2666MHz CPU with eight logical cores and 64 GiB RAM.

Ideally all tests would have been performed on the same machine, to make results directly comparable. Machine 3 is the most powerful set-up, but access to its CPU time has been limited. For this reason, our tests had to be split up across several machines.

Machine 1 runs OpenJDK 1.6.0.24 on a 64-bit Mandriva 2011. The other three run OpenJDK version 1.7.0.45 on 64-bit Mageia 3. Despite the different software, we will consider machines 1 and 2 to be equivalent. Each machine has a relatively small swap partition, and we do not write any data to file system. We are thereby limited to using physical RAM.

4.2 Preliminary testing

This section details some preliminary results on small graphs, which will guide the choices about the testing of larger graphs. As these tests are small in scope and not very taxing on our hardware, every preliminary test was performed on machine 1.

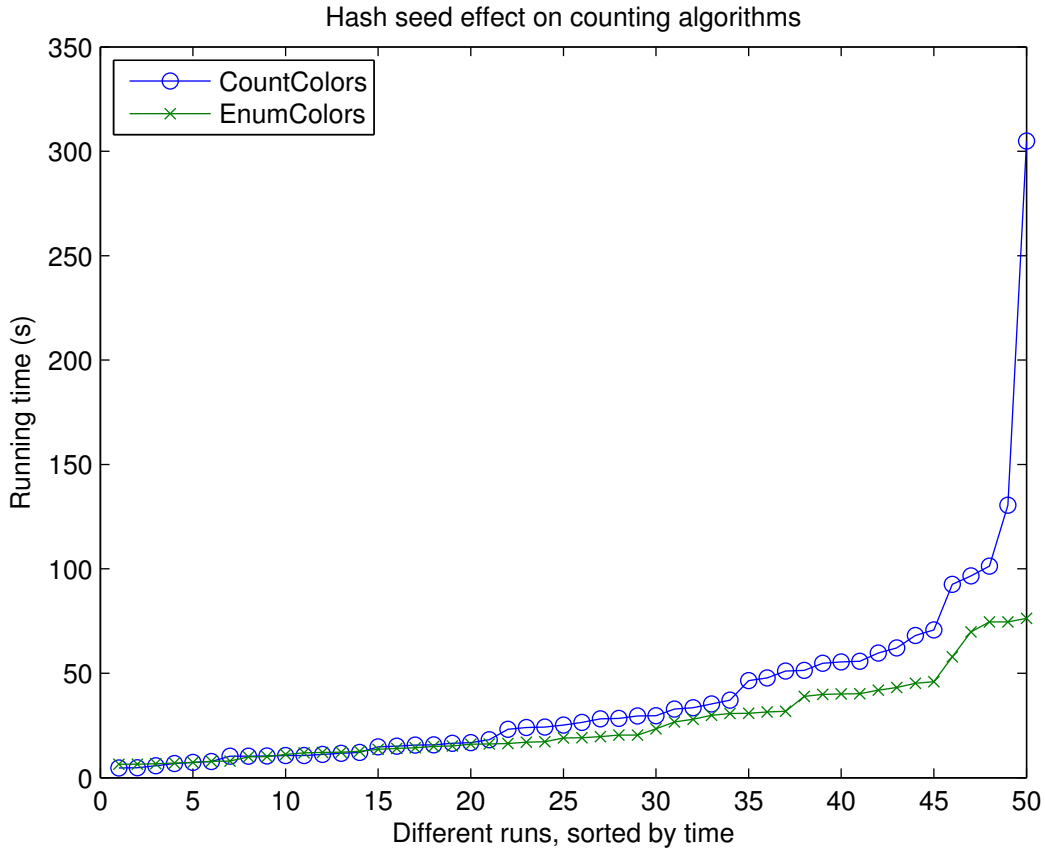

Figure 4.1: Counting EnumColors and CountColors hash seed dependency

4.2.1 Hash seeds

We begin by testing what effect the choice of hash seed has on the performance our algorithms. We use a graph of 50 vertices and apply the single-threaded versions of EnumColors and CountColors to count the number of edge colorings. `CountColors` does not use the `aggressive` flag and `EnumColors` is not set to print. Their running times are shown in figure 4.1.

From these figures it is apparent that a bad hash seed can have a very negative consequences for our running times. We opt to use a number of different hash seeds for every graph in our primary tests.

For the determining of the graph's class, we perform similar tests using single-threaded `EnumColors` and `Kowalik`. `EnumColors` is again set not to print. We use graphs of size 70 because the non-counting running times on size 50 graphs are very short. Results are seen in figure 4.2.

As seen here the hash seed does not appear to be noticeably impactful for our `Kowalik` program, while it still makes a difference for `EnumColors`. Note the two different y-scales for the two plots; EnumColors takes up to 35 seconds to run, while every run of Kowalik finished in 10 milliseconds or less.

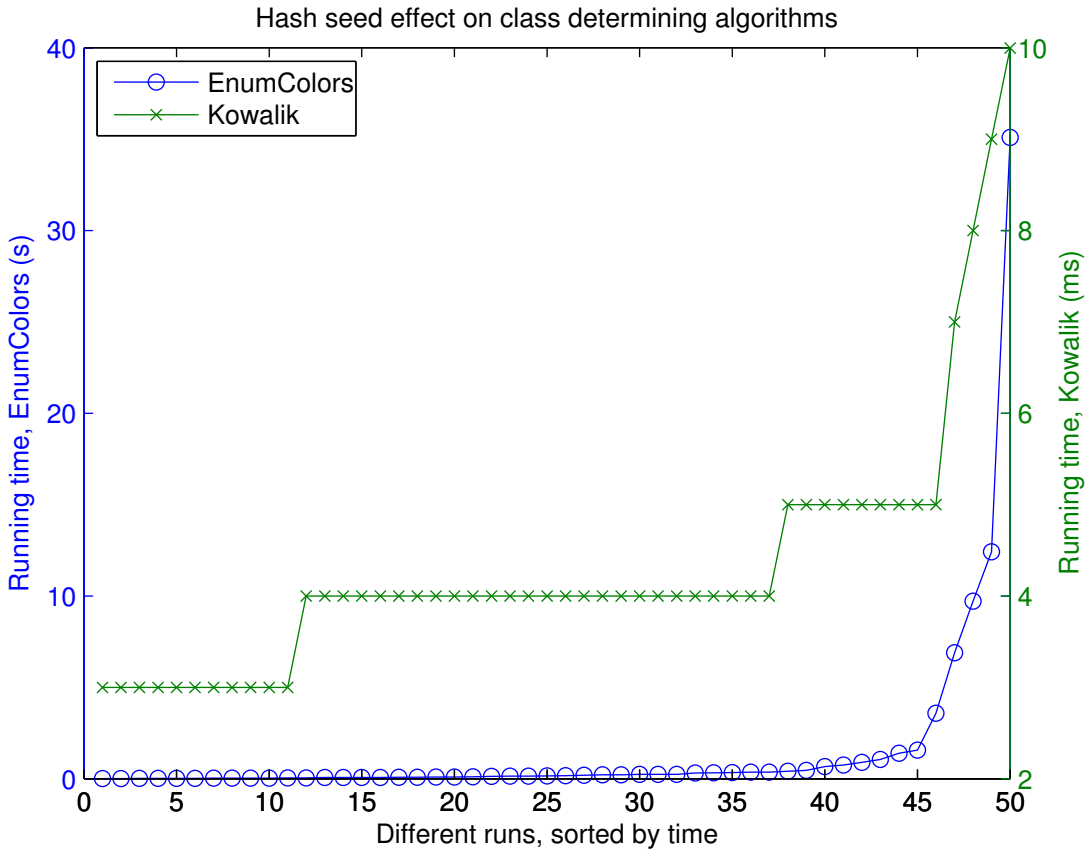

Figure 4.2: EnumColors and Kowalik hash seed dependency

4.2.2 Aggressive epsilon

To test the significance of choice of ϵ, we run the path decomposition generator twice on several graphs of 50 vertices and with 10 hash seeds; first with `aggressive` off, and then with it on. Widths and times are averages over the 10 seeded runs in figure 4.3. We do the same for size 100 graphs, shown in 4.4. The high average `aggressive` time for size 100 graph 8 is due to one run that needed 29 seconds to finish; the other runs ranged between 200 and 440 milliseconds.

	Non-aggressive		Aggressive	
	Width	Time (ms)	Width	Time (ms)
Graph 1	21	4.1	12	208
Graph 2	21	1.5	16	111
Graph 3	19	2.8	14	347
Graph 4	21	1.4	14	108
Graph 5	21	1.5	15	100
Graph 6	22	2.1	14	196
Graph 7	21	1.3	15	110
Graph 8	21	1.7	15	130
Graph 9	22	1.4	14	101
Graph 10	20	1.6	15	104
Average	21	1.94	14	153

Figure 4.3: Efficiency of aggressive flag on size 50 graphs

	Non-aggressive		Aggressive	
	Width	Time (ms)	Width	Time (ms)
Graph 1	40	4.3	23	376
Graph 2	40	4.1	23	350
Graph 3	39	3.9	23	351
Graph 4	39	4.1	30	358
Graph 5	39	4.2	24	374
Graph 6	41	4.3	21	384
Graph 7	40	3.9	24	395
Graph 8	40	4.4	22	3220
Graph 9	40	3.8	24	319
Graph 10	40	3.8	22	395
Average	40	4.1	24	652

Figure 4.4: Efficiency of aggressive flag on size 100 graphs

Clearly, using the aggressive flag is well worth the investment. For the smaller graphs we spend only some 150 milliseconds to decrease the width by 33%. For the larger graphs, we spend 650 milliseconds more to decrease the width by 40%. Thus we opt to use the aggressive flag in all tests of CountColors; the cost-to-gain ratio is simply too good to ignore.

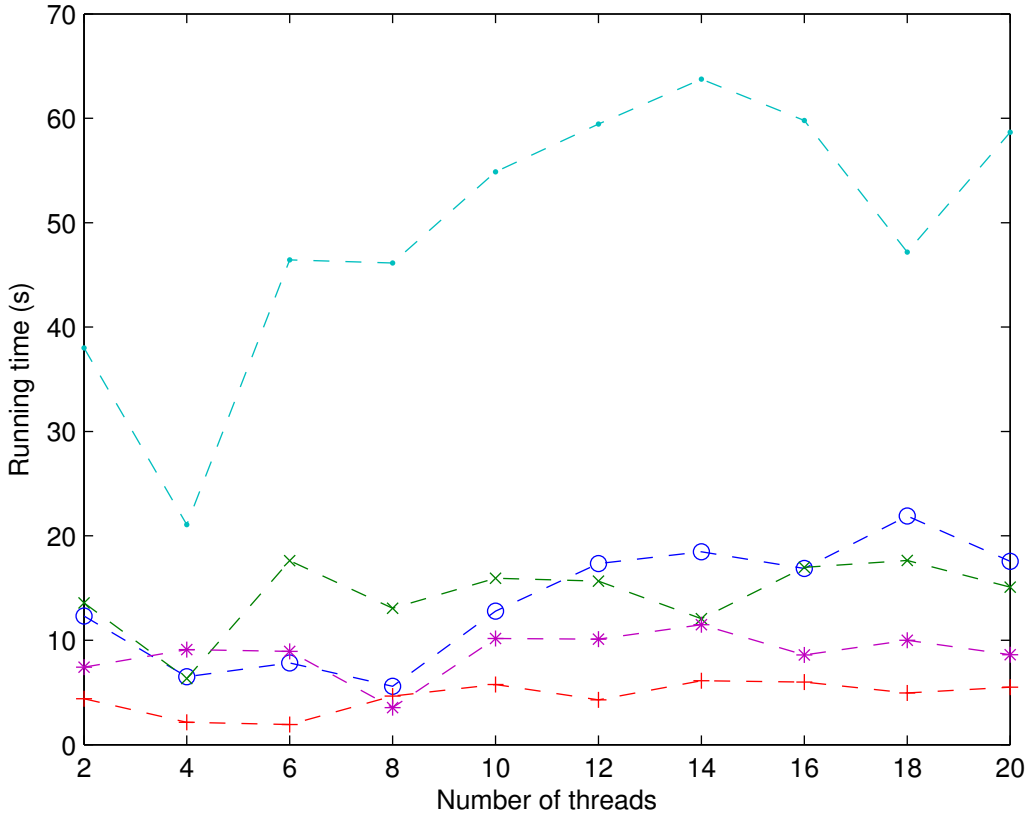

Figure 4.5: EnumColorsParallel efficiency by thread count

4.2.3 Number of threads

We know how many concurrently active threads each CPU can handle – 24 for machine 3, 8 for the others. Due to the parallel architecture of CountColors, it is immediately clear that we gain the greatest speed-up factor from using one thread per core. For EnumColors and Kowalik, things are not as clear; threads ask for help when the code branches, and one branch may finish sooner than the other. If the assisting thread finishes first, it goes back to the pool of available threads, but if the original thread finishes first it will simply await the other, which could imply a waste of CPU time. Furthermore, as the threads need to communicate, there is necessarily some amount of synchronization between them. Excessive synchronization can cause the program to slow down.

We run `EnumColorsParallel` in counting mode on five different graphs of size 50, using between 2 and 20 threads and no hash seeding. The results, shown in figure 4.5, seem to dip at 4 or 8 threads for different graphs. As we possess two equal machines, we opt to run parallelized EnumColors with 4 threads on machine 1 and 8 threads on machine 2.

Because single-threaded Kowalik is already so fast on these graph sizes, we would not be able to draw any conclusions from running either of the parallel versions on them. Instead we suppose that 8 is a good thread number as with EnumColors, as their parallelization architectures are similar.

4.3 Results

Taking into account our preliminary results, we conduct our main tests the following way. We randomly generate cubic graphs of even sizes in [50, 150); ten graphs of every size. To account for the impact of hash seeds, we choose 10 random hash seeds for every graph. The hash seeds are generated using Java's standard random generator, `Random.nextInt()`. This creates a total of 5000 seeded graphs to use in every test. The seeds are stored with the graph file, so we can easily repeat a test and get the same running time. We expect that not every test will be able finish on all the 5000 graphs, but each will be allowed to run for at least 96 hours in total.

Additionally, we generate a smaller set of graphs of the same size range, which are all known to be of class 2. There are five graphs per size and 10 hash seeds per graph, for a total of 2500 runs. This lets us compare the "time to fail" between the different algorithms, which may be an important property in itself as it reveals the waiting time before applying one of the polynomial-time algorithms for finding an $(\triangle(G) + 1)$-edge-coloring. The set was generated by the regular randomization method, and using Kowalik to reject those of class 1.

Despite being randomly generated, it turns out that every graph in the first set is class 1. We may therefore refer to the two sets as the *class 1 set* and the *class 2 set*.

For determining graph classes and finding a single edge coloring, we employ Kowalik, parallel Kowalik, EnumColors and parallel EnumColors. CountColors is not expected to be competitive for any of the graphs, since it must first solve the counting problem. However, as the counting algorithm may discard all characteristics as invalid at an early step when the graph is not 3-edge-colorable, we include CountColors for the set of class 2 graphs.

For counting edge colorings, we use single-threaded and multi-threaded versions of both CountColors and EnumColors.

We do not perform tests to enumerate all edge colorings. `EnumColors` is the only package even able to solve the problem, as the `CountColors` package does not save intermediary tables. Additionally, as the number of colorings may be very large, we may experience performance bottlenecks from I/O or memory consumption, depending on how we choose to print the discovered edge colorings, and those may render our measurements meaningless.

4.3.1 EnumColors

4.3.1.1 Finding an edge coloring

We begin by running `EnumColors` on the class 1 set graphs. As noted earlier, by finding an edge coloring we automatically also determine that the graph is class 1. The results may be found in figure 4.6. We get up to size 100 before having to break to perform other tests. `EnumColorsParallel` with 8 threads runs on the same set and also manages to get to size 100, as shown in 4.7.

The figures look very similar to each other, and it is not obvious which program was faster. Therefore we plot the running times for every individual seeded graph, as a division `EnumColors/EnumColorsParallel`, in figure 4.8. Values above 1 mean that

`EnumColorsParallel` was faster, and vice versa. This figure is clear only in the message that the results vary greatly – no implementation can be said to be faster than the other. However, it appears that the time quotient is below 1 more often than not. We therefore choose not to run the 4-threaded `EnumColorsParallel` for this problem; it appears to be a waste of CPU time that can be spent on other tests.

4.3.1.2 Counting all edge colorings

For the counting problem, we run `EnumColors` (4.9) and `EnumColorsParallel` with 4 (4.10) and 8 (4.11) threads. The two parallel programs were both started and halted at the same time as each other, and therefore had the same total running time. One can note that the 8-threaded version got up to size 76, while the 4-threaded version did not make it past 74. We can therefore immediately state that the 8-threaded version turned out to be somewhat faster. We also plot the averages of each test together in 4.12. Here we see again that the 8-threaded version is a bit faster than the 4-threaded one. We also see that `EnumColors` starts to lag behind significantly for about 62 vertices.

4.3.2 Kowalik

4.3.2.1 Determining class of a graph

The figure 4.13 shows the results of the standard Kowalik running on the class 1 set of graphs. As shown in these tables, Kowalik performs exceptionally well, finishing on the largest graphs in a matter of milliseconds. Even the slowest runs all finish in less than a second. The performance is so strong that we cannot even perform any meaningful curve fitting to these data points. We therefore generate a set of new graphs; starting from size 150 and incrementing in steps of 10 up to 1500, ten graphs per size, and only three different hash seeds. Call this the *large set*. Kowalik's running times on the large set are shown in figure 4.14.

The parallel versions are applied to the same high degree graphs. `KowalikParallel1` is shown in figure 4.15 and `KowalikParallel2` in 4.16.

The figures for both parallel versions are very similar to that of the single-threaded Kowalik. We compare their average times in 4.17. Regular `Kowalik` was given more time to run, which is why it has values for larger graphs. A zoomed plot of all times exceeding 10 seconds may be found in figure 4.18, showing that their averages are very similar and that no implementation can be categorically stated to be faster than the others, although `Kowalik` may have the edge in the majority of cases.

4.3.2.2 Finding an edge coloring

We run `KowalikPrint` on the class 1 set and compare them with regular `Kowalik`, shown in 4.19. These running times are indistinguishable from one another, so we see how `KowalikPrint` fares with the large set. This is shown in figure 4.20.

As seen here, Kowalik requires virtually no extra time to find an edge coloring for any of the graphs. This is expected, since the halting condition for the original version is that a

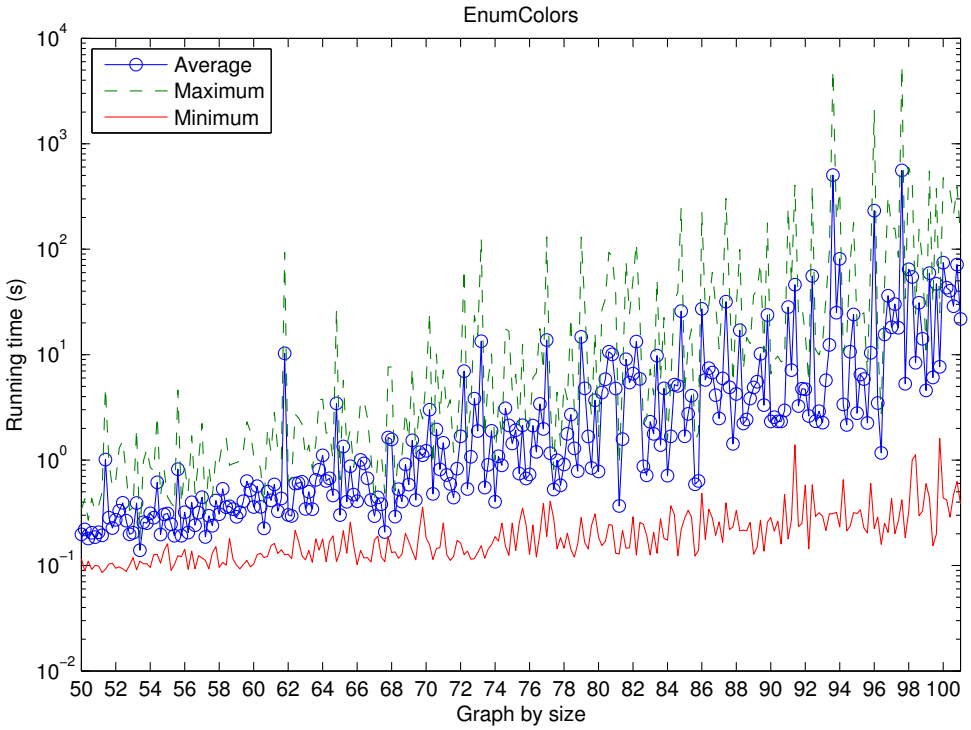

Figure 4.6: EnumColors on the class 1 set

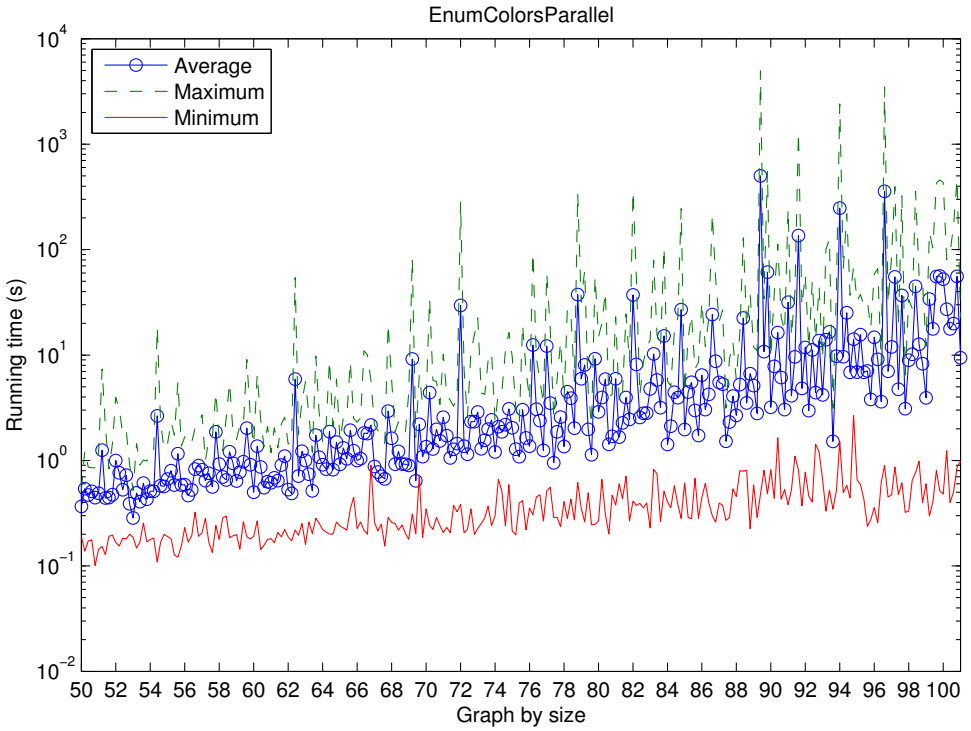

Figure 4.7: EnumColorsParallel on the class 1 set

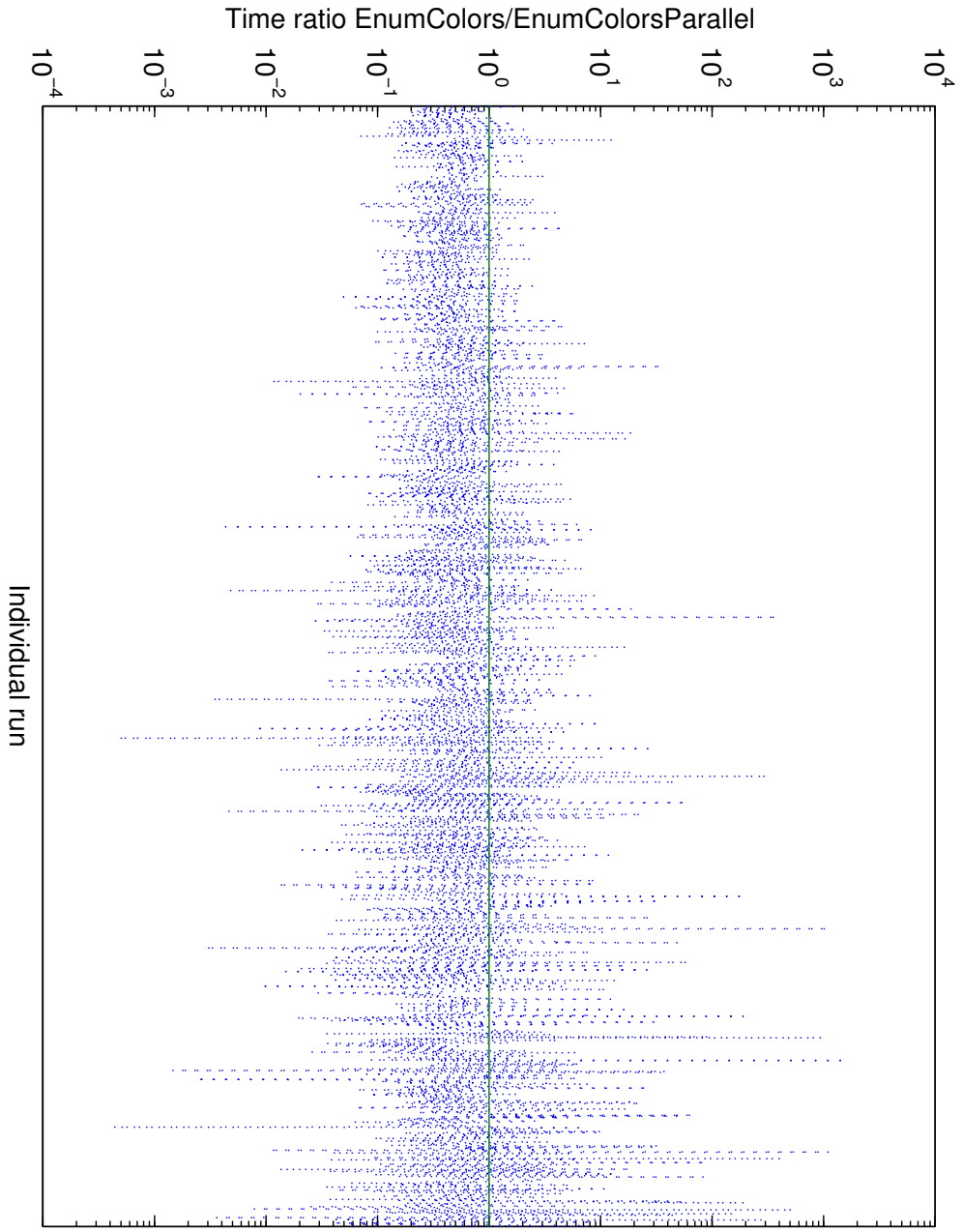

Figure 4.8: EnumColors and EnumColorsParallel

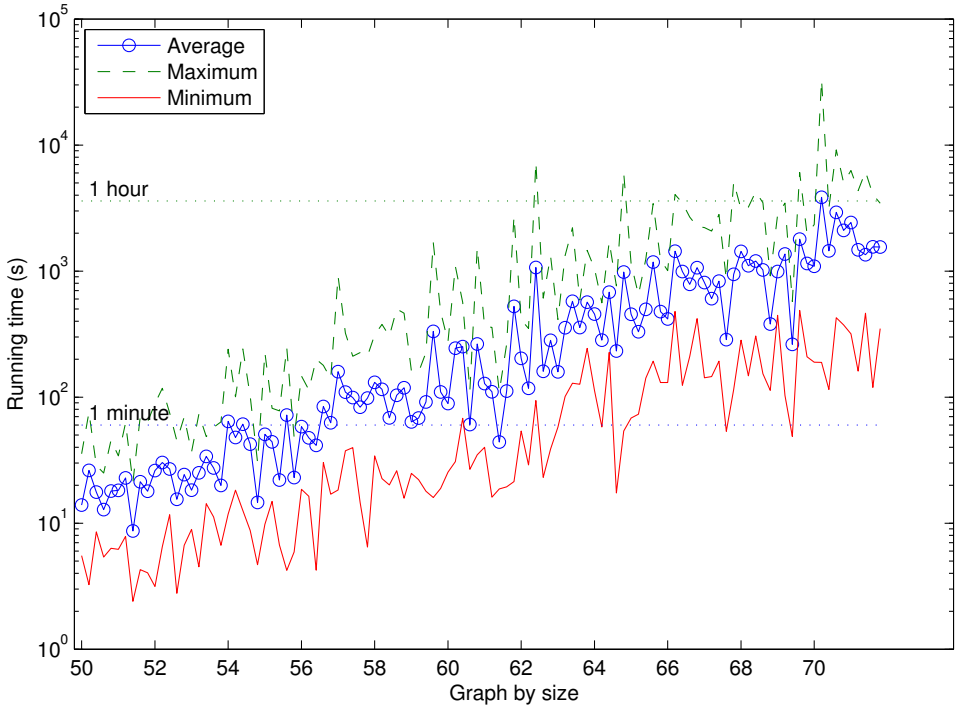

Figure 4.9: EnumColors counting

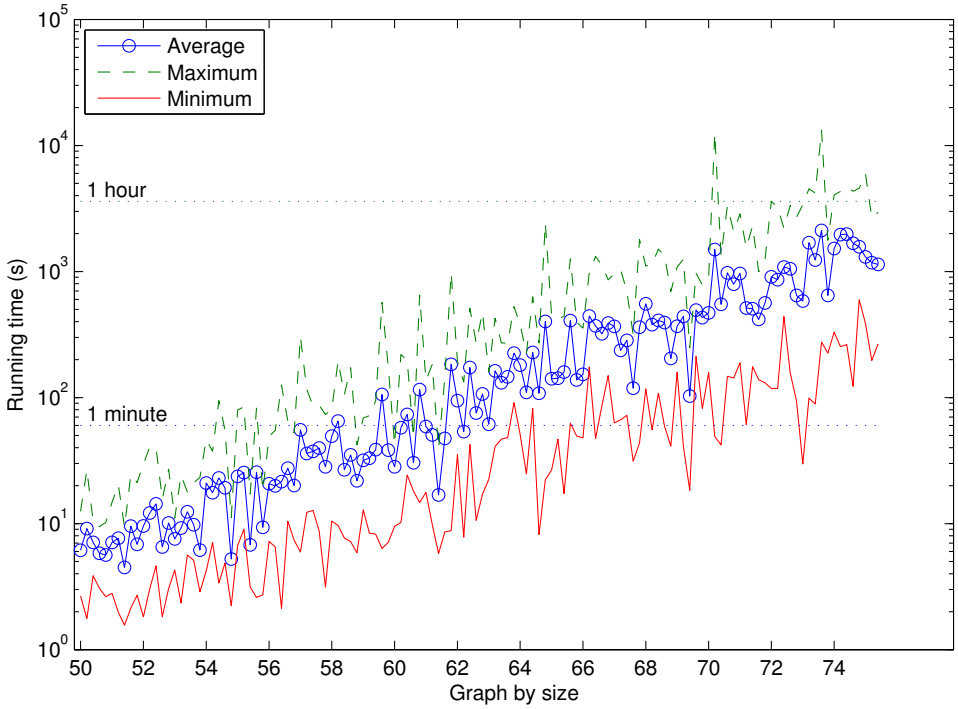

Figure 4.10: EnumColorsParallel counting with 4 threads

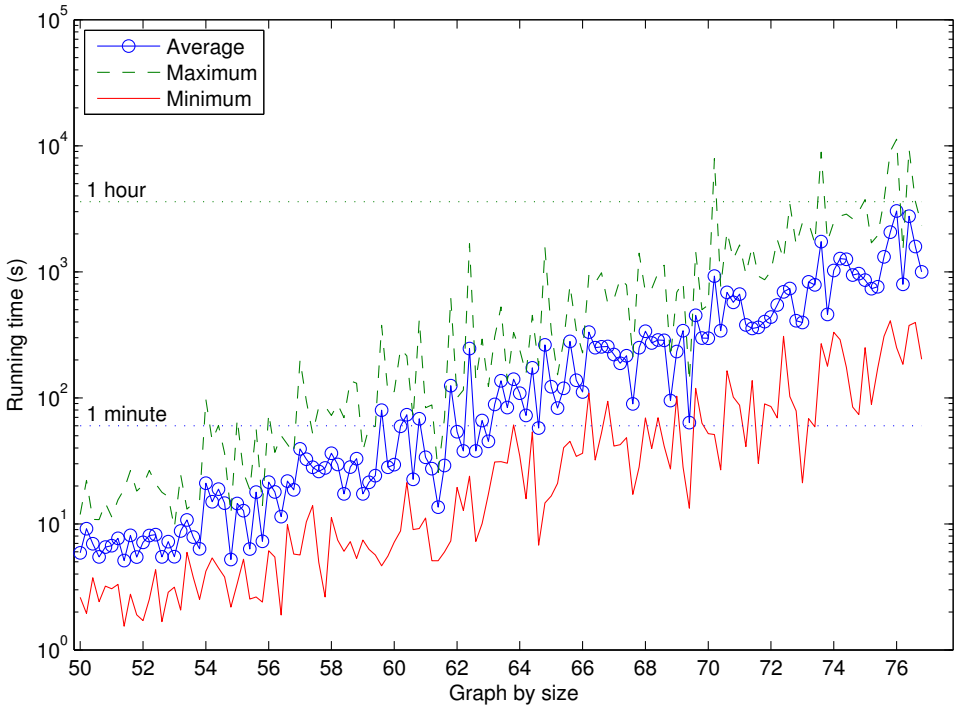

Figure 4.11: EnumColorsParallel counting with 8 threads

fitting matching has been found, and the creation of an edge coloring is an $O(m)$ operation if a fitting matching is known.

4.3.3 CountColors

4.3.3.1 Single-thread version

We test the first set of graphs on machine 3, with 256 GiB of RAM. 245 GiB was dedicated to the Java heap, to ensure that the operating system and the garbage collector could still run smoothly. The peak memory usages reported in this section are approximations reported by Java's `Runtime.totalMemory() − Runtime.freeMemory()`. They were also sampled between each of the $2n$ steps, when there is only one table of characteristics alive on the heap but old tables may linger as garbage. Maximal simultaneous memory usage may therefore be up to twice the numbers presented here, in case of two consecutive tables of equal size. A more in-depth view of memory use is given in section 4.3.3.3.

We begin by examining the running time and peak memory use of the graphs in the class 1 set. Because we had limited access to machine 3, we cut out some graphs starting from size 82. We kept 7 different graphs, and 7 hash seeds for each for a total of 49 runs per size between 82 to 90 nodes. Even then, only a few of the size 90 graphs had enough time to be tested, as we moved on to some size 100 graphs in order to see some higher failure rates. We detail running time average and ranges for every individual graph in 4.21. They are color coded by graph size for easier viewing.

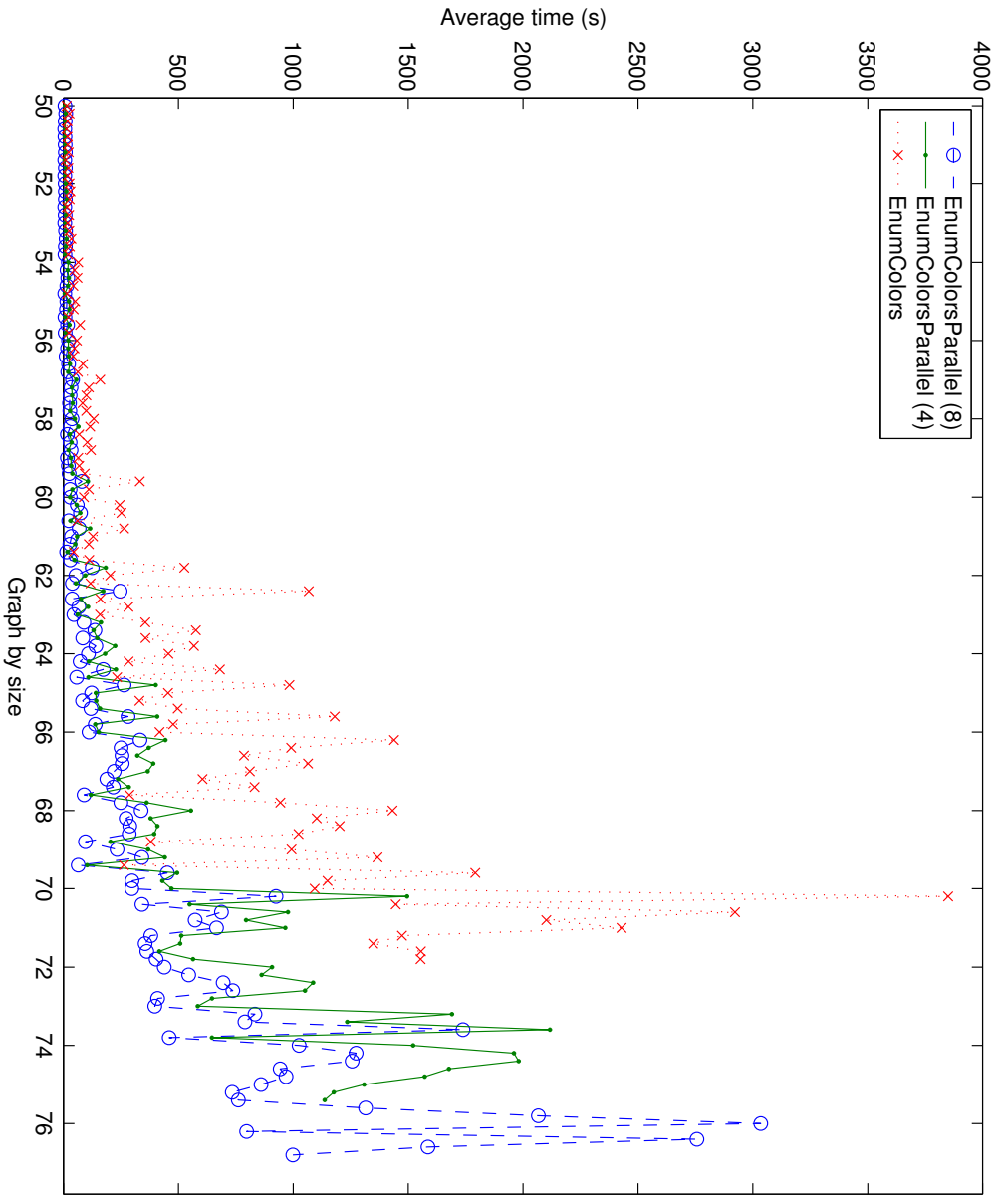

Figure 4.12: EnumColorsParallel counting, 4 and 8 threads

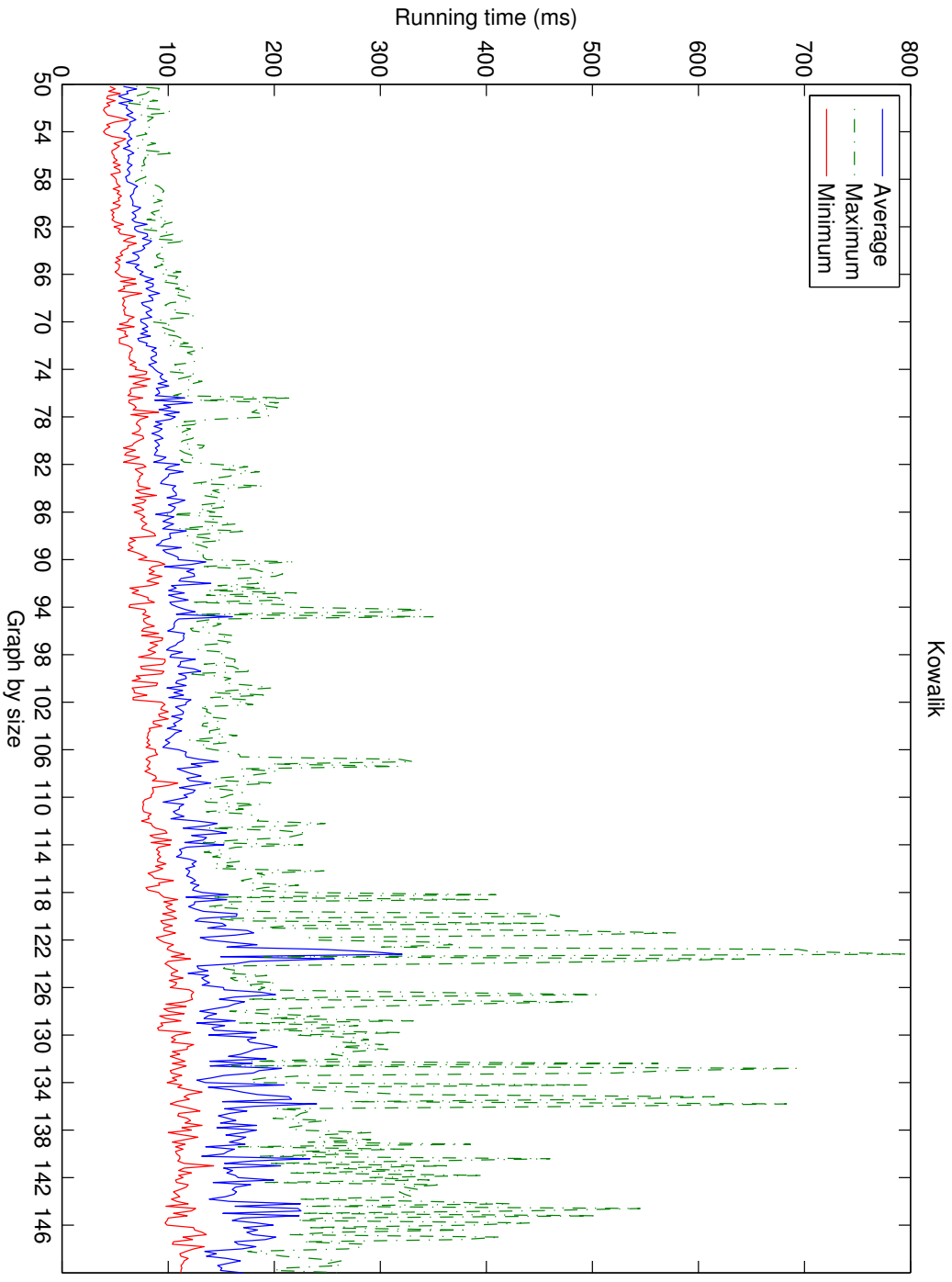

Figure 4.13: Kowalik on class 1 set

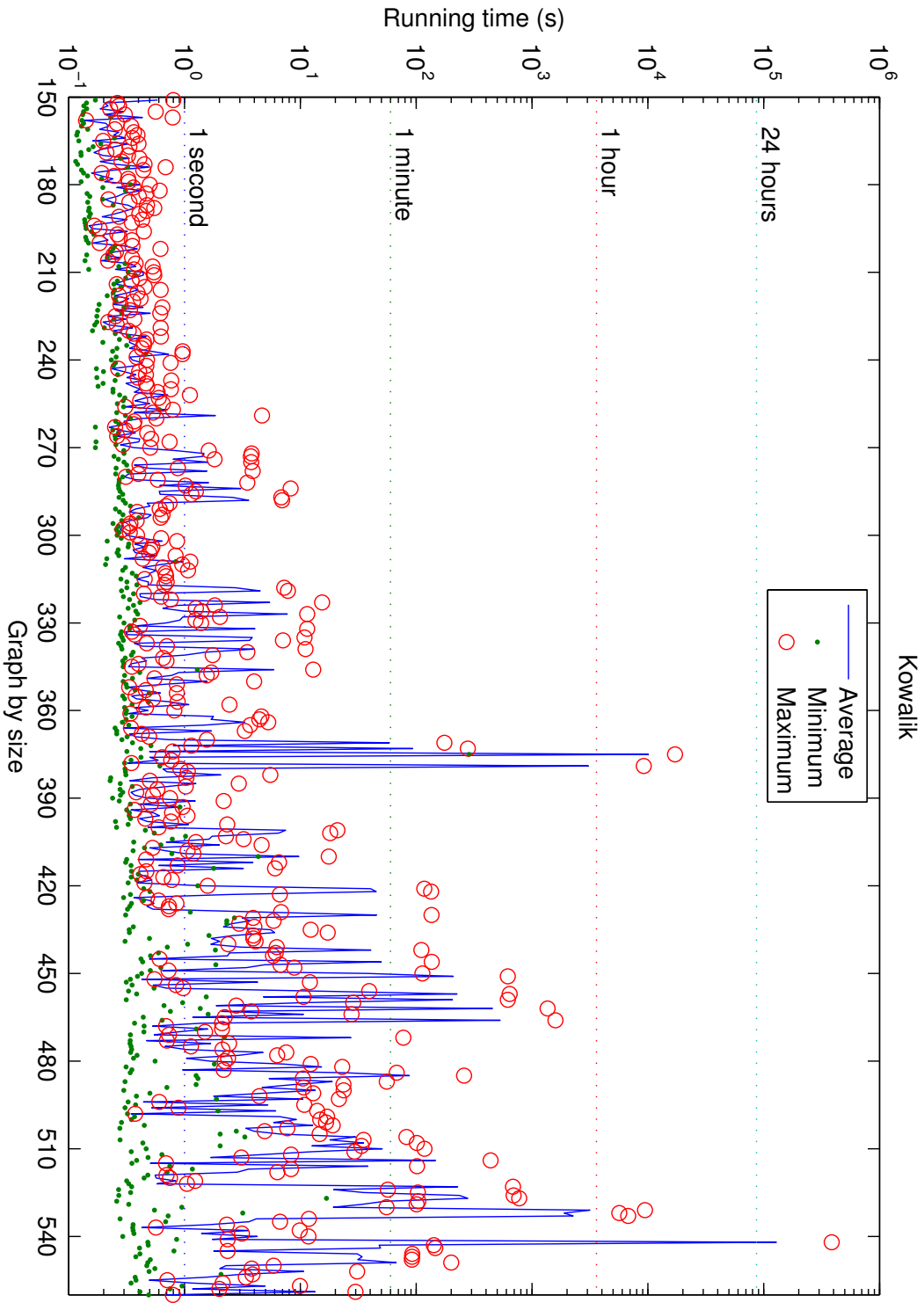

Figure 4.14: Kowalik on large set

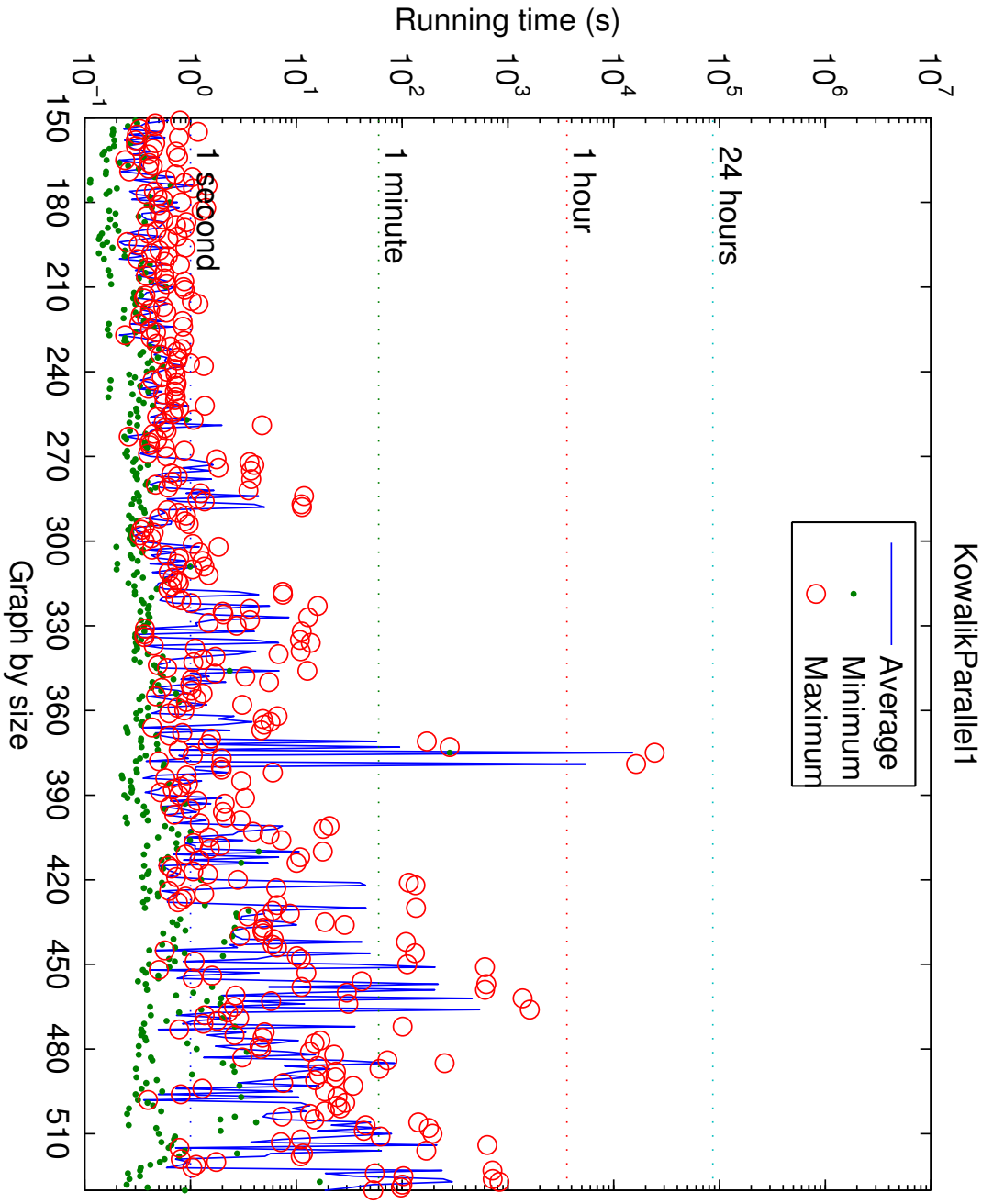

Figure 4.15: KowalikParallel1 on class 1 set

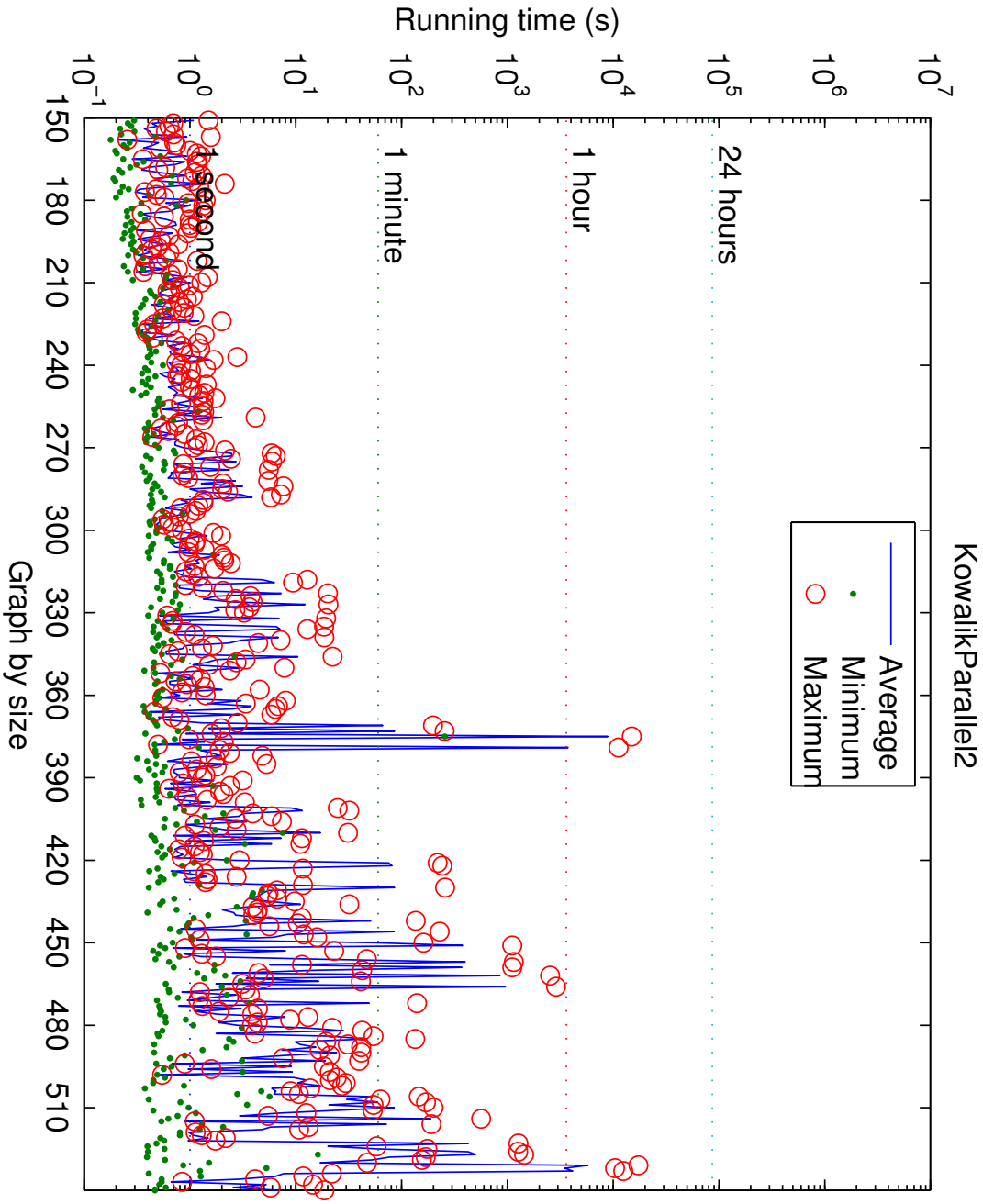

Figure 4.16: KowalikParallel2 on large set

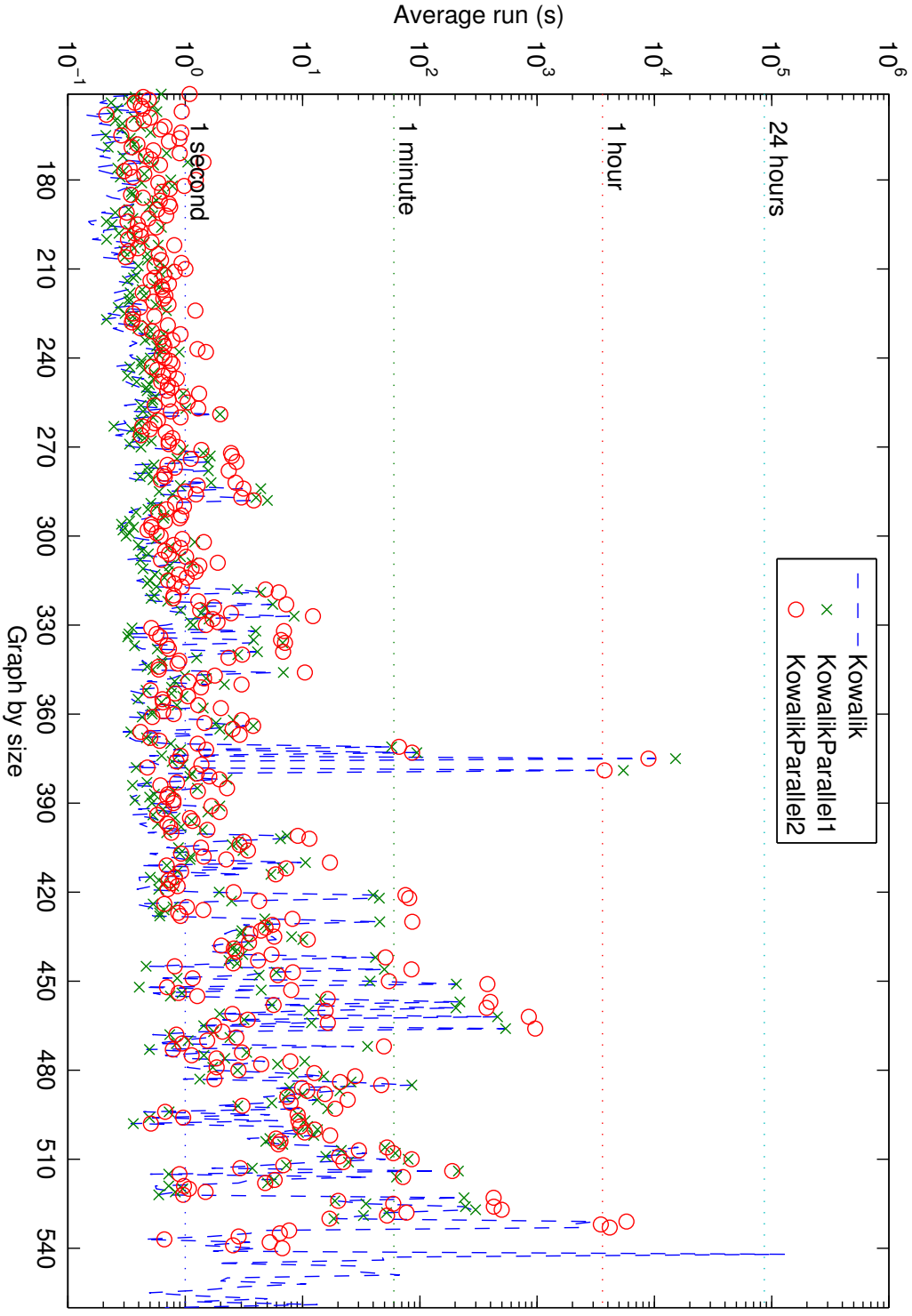

Figure 4.17: Single-threaded and parallel Kowalik versions

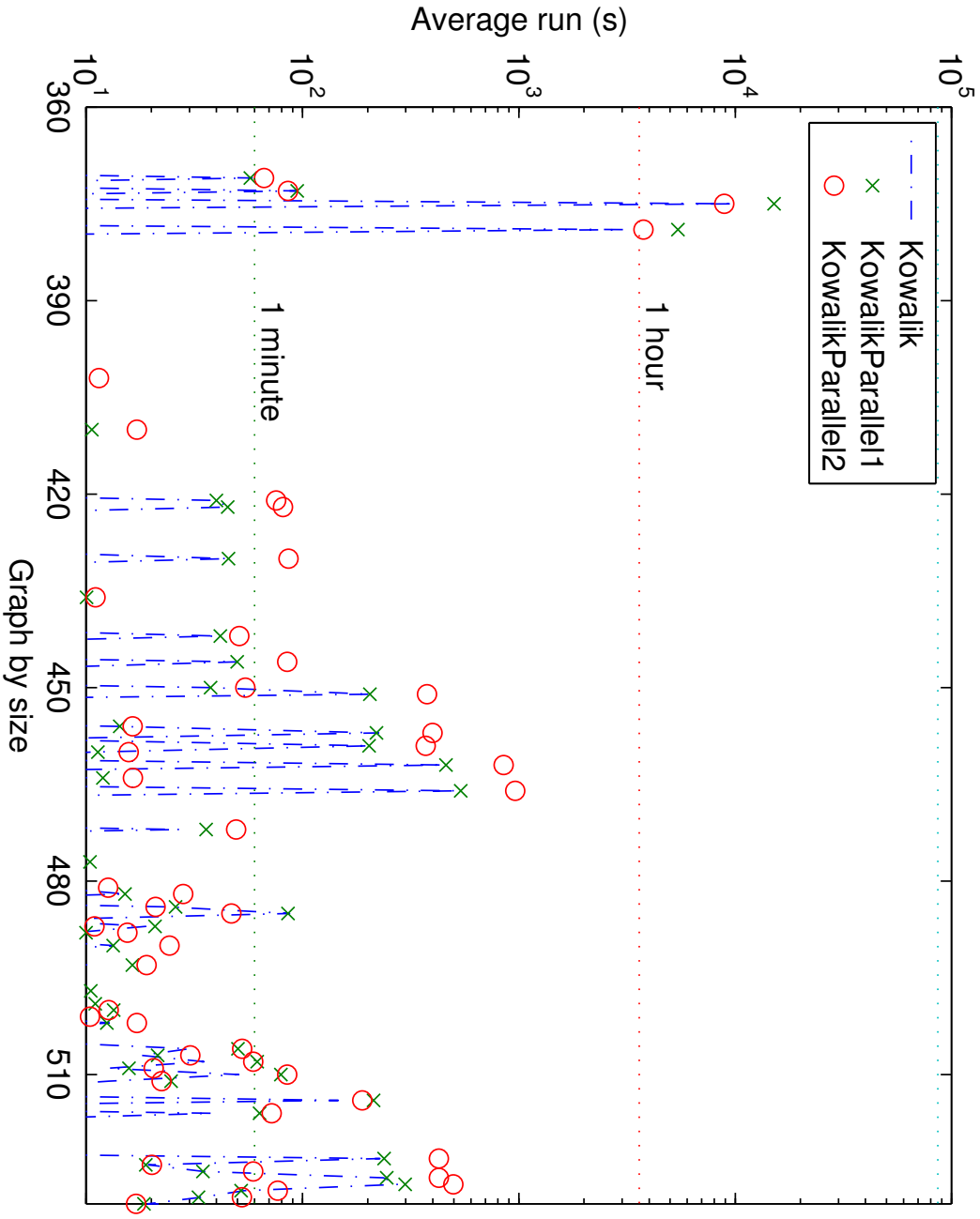

Figure 4.18: Single-threaded and parallel Kowalik versions, zoomed

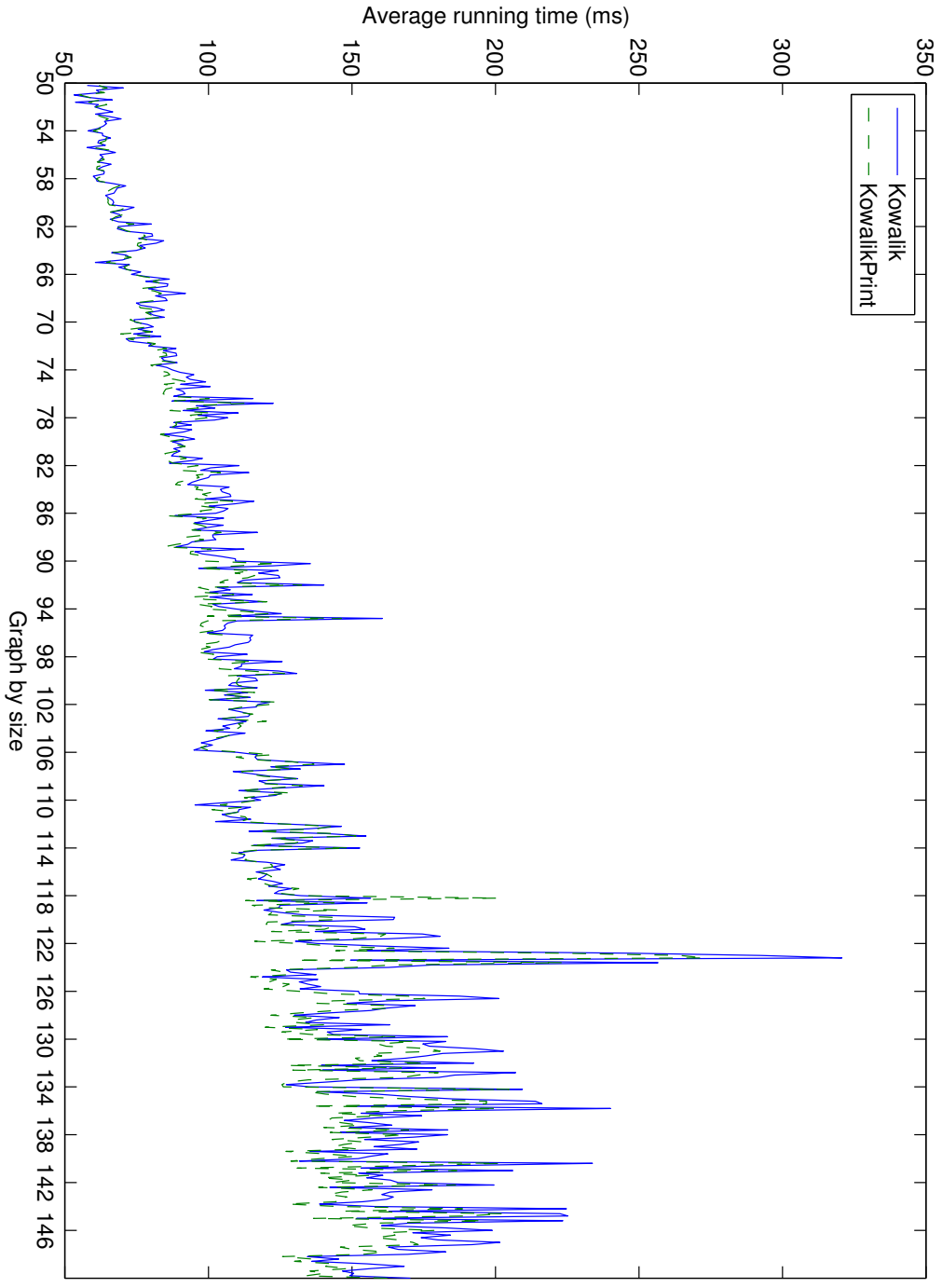

Figure 4.19: Kowalik and KowalikPrint on class 1 set

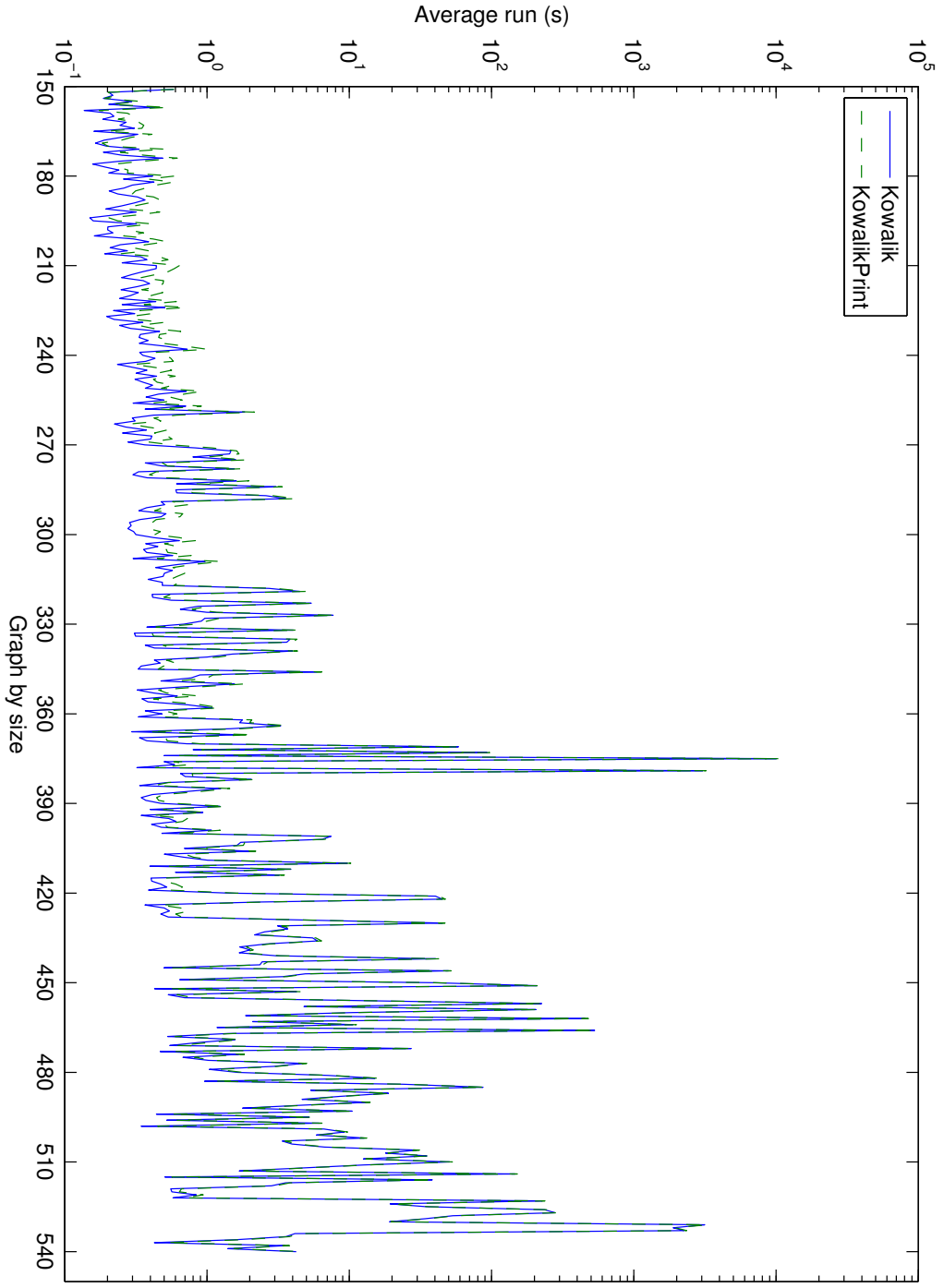

Figure 4.20: Kowalik and KowalikPrint on large set

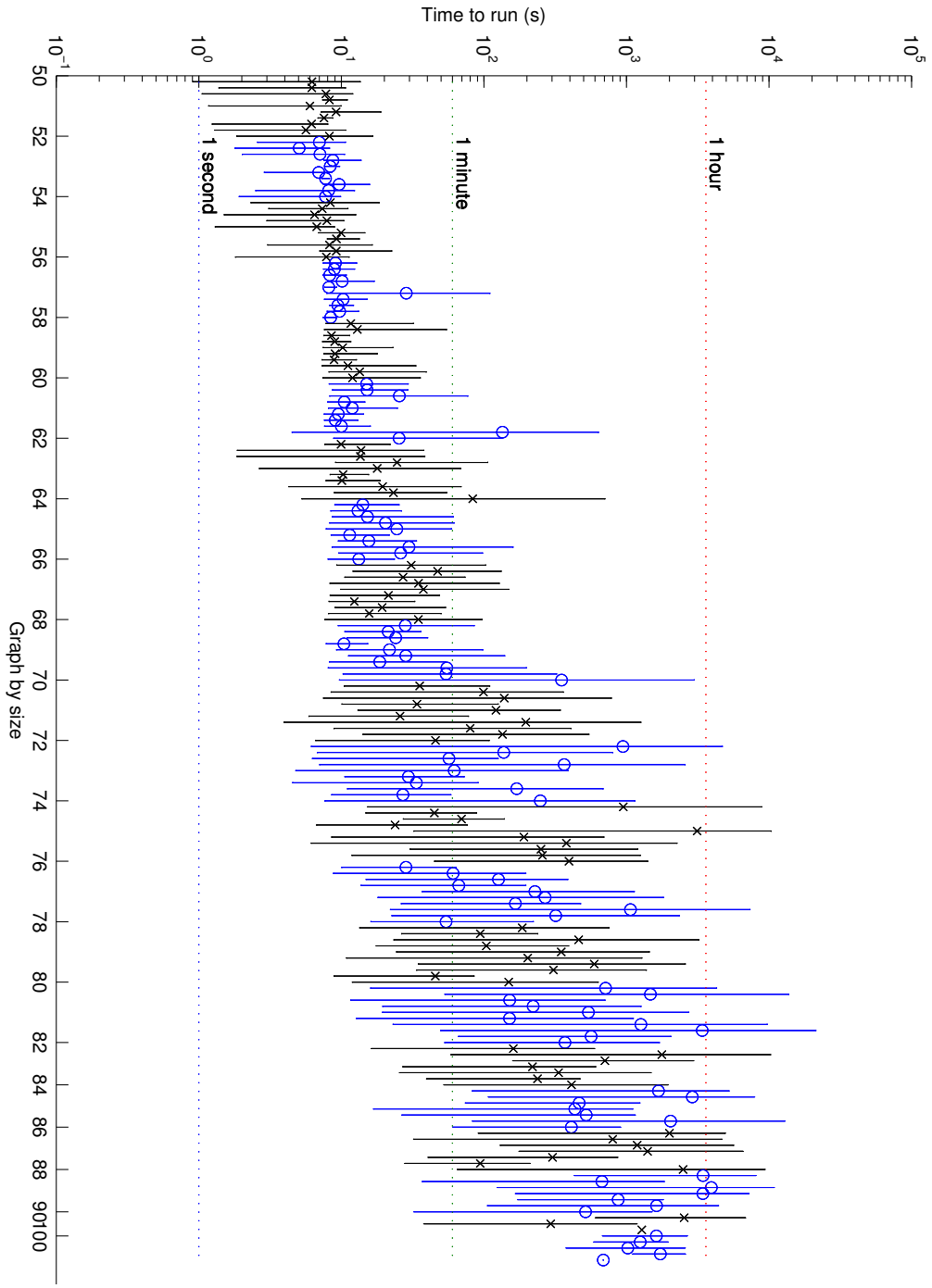

Figure 4.21: CountColors running time by graph

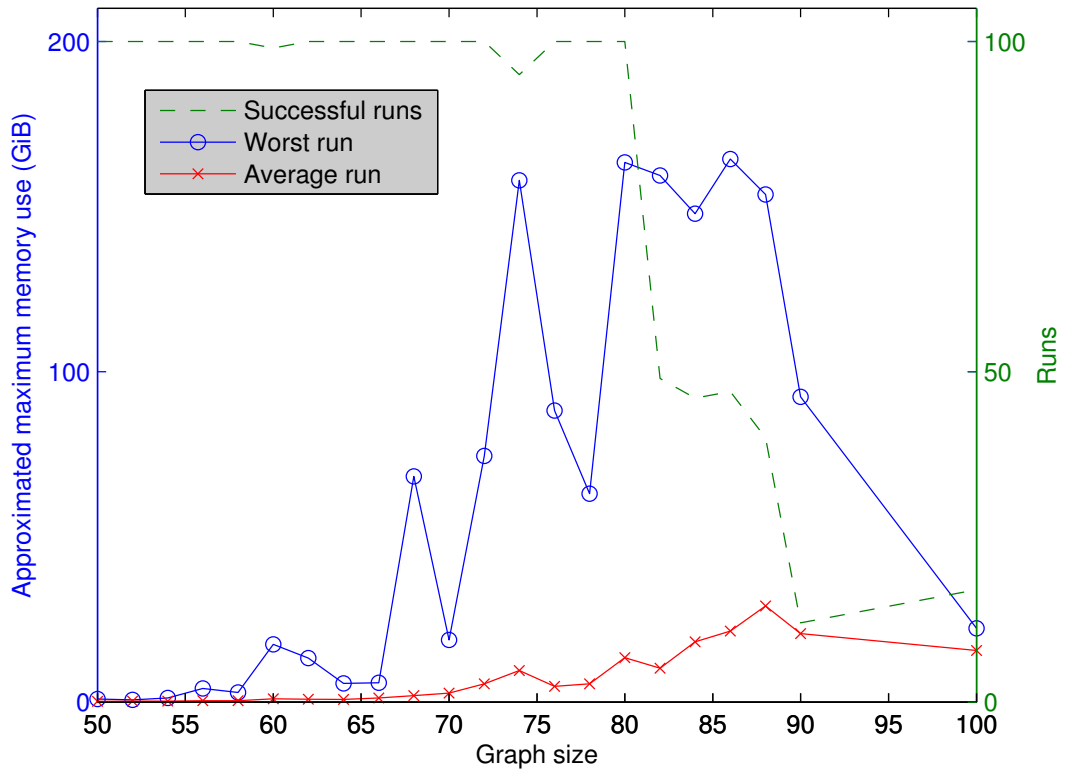

Figure 4.22: CountColors peak memory by graph size

Peak memory use by graph size is given by figure 4.22. The plot also details the number of successful runs for that size. Note the small dip at size 62, which demonstrates that we can run out of memory already for those relatively small graph sizes.

Strictly speaking, this algorithm is not exponential in the size of the graph, but in the width of the path decomposition. It is very relevant to see the efficiency of our aggressive path decomposition search. We plot the values of ϵ_2 per graph size (seen in figure 4.23), and the width (average and range) of the path decompositions found for each graph (figure 4.24).

We are also interested in knowing the memory usages and success rates by path decomposition width. Success rate is shown in 4.26, and memory usage is in 4.25. We see a clear growing memory use (declining success rate) until width 28, where it suddenly drops (shoots back up). We attribute this peculiarity to having too few runs of those higher widths, rather than any general trend that widths greater than 30 are preferable to smaller ones. There were no runs on path decompositions of width 31.

The running times by width exhibit a very interesting phenomenon: one minute running time very nearly corresponds to one GiB peak memory use. This is of course highly dependent on the particular hardware. Average running times is shown in 4.27, and maximum running times in 4.28.

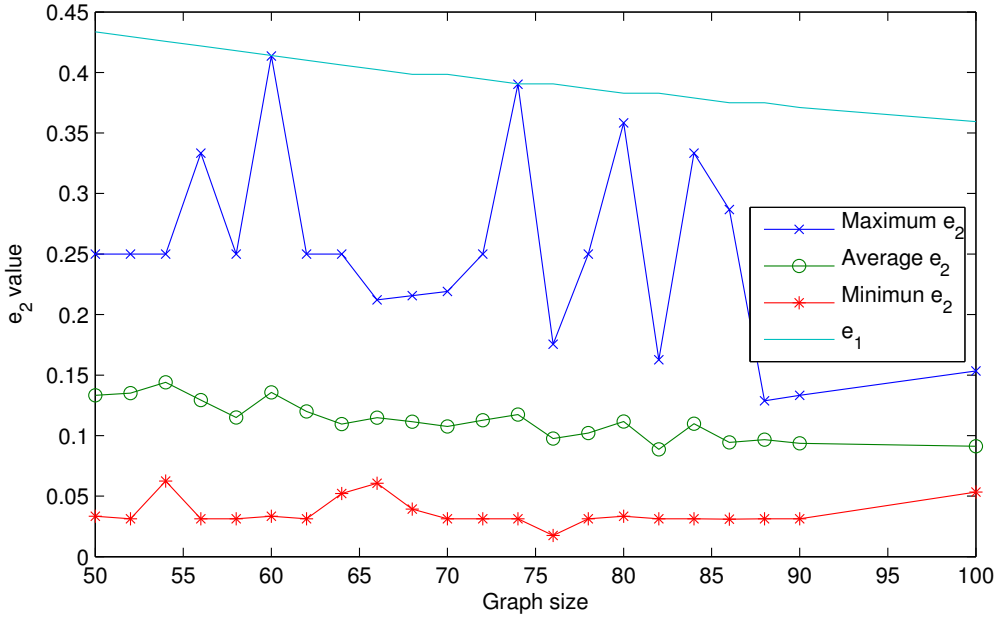

Figure 4.23: ϵ_1 and ϵ_2

4.3.3.2 Parallel version

As we predicted that the Parallel CountColors may require more memory to solve the same problem, we put this to the test using the 64 GiB RAM machine with 8 logical cores (machine 4). The expectation is that the speedup and increased memory use will be visible regardless of which machine performs the tests. By using a computer with less memory available, we illustrate the expected behavior while avoiding the running times of up to several hours observed on the 256 GiB machine. In return, this allows the single-threaded program more time to work on the larger graphs using 256 GiB of memory. Out of our 64 GiB RAM, we dedicate 58 GiB to the JVM heap. We only include the speedups (4.29) and memory increases (4.30) here, as previous tests detailed running times and peak memory usages in absolute terms.

We see that peak memory consumption of `CountColorsParallel` grows to about 140% of that of `CountColors` for the largest graphs, with a hint of a continuing upward trend. There was only a single seeded graph where `CountColors` finished and `CountColorsParallel` crashed: a size 80 graph which used 32 GiB peak memory in its successful run. The gain was that most runs finished 2-3 times faster.

4.3.3.3 Total memory use

We include a plot 4.31 to illustrate memory use for every individual table, and not just the largest one. We tested three different graphs of size 70, and chose the run with the highest peak memory for each of them. The JVM garbage collector is forcibly started between every step. The steep left-most cliff for graph 3 serves well to demonstrate how the memory grows exponentially. Graph 2 peaks at about 22.5 GiB, but the sum over all its steps is 370 GiB, far above what we are able to store.

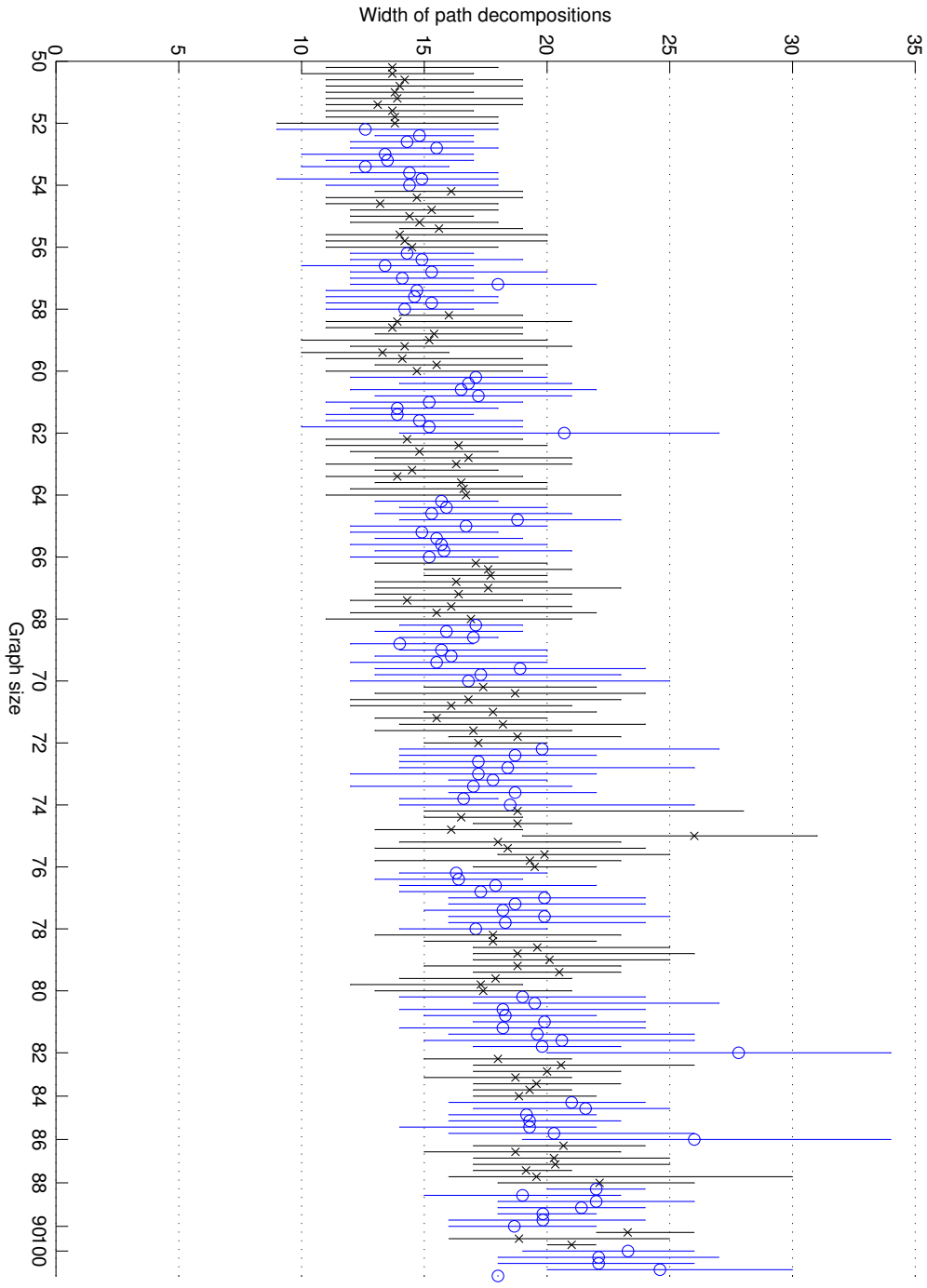

Figure 4.24: Width of path decomposition by graph

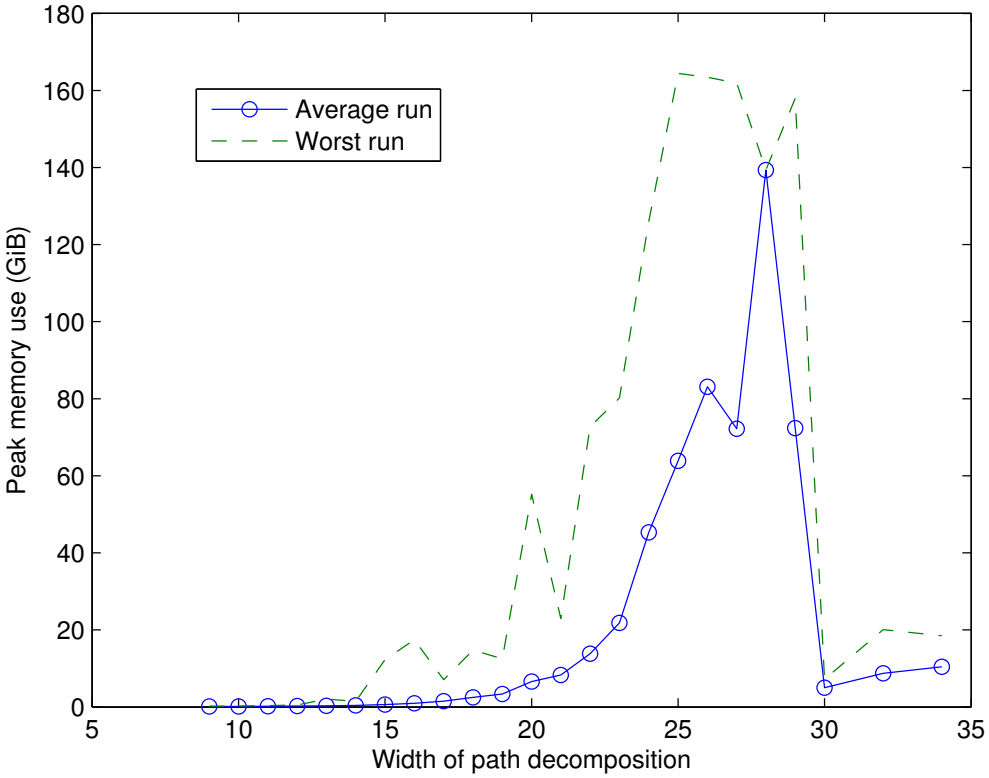

Figure 4.25: Peak memory use by path decomposition width

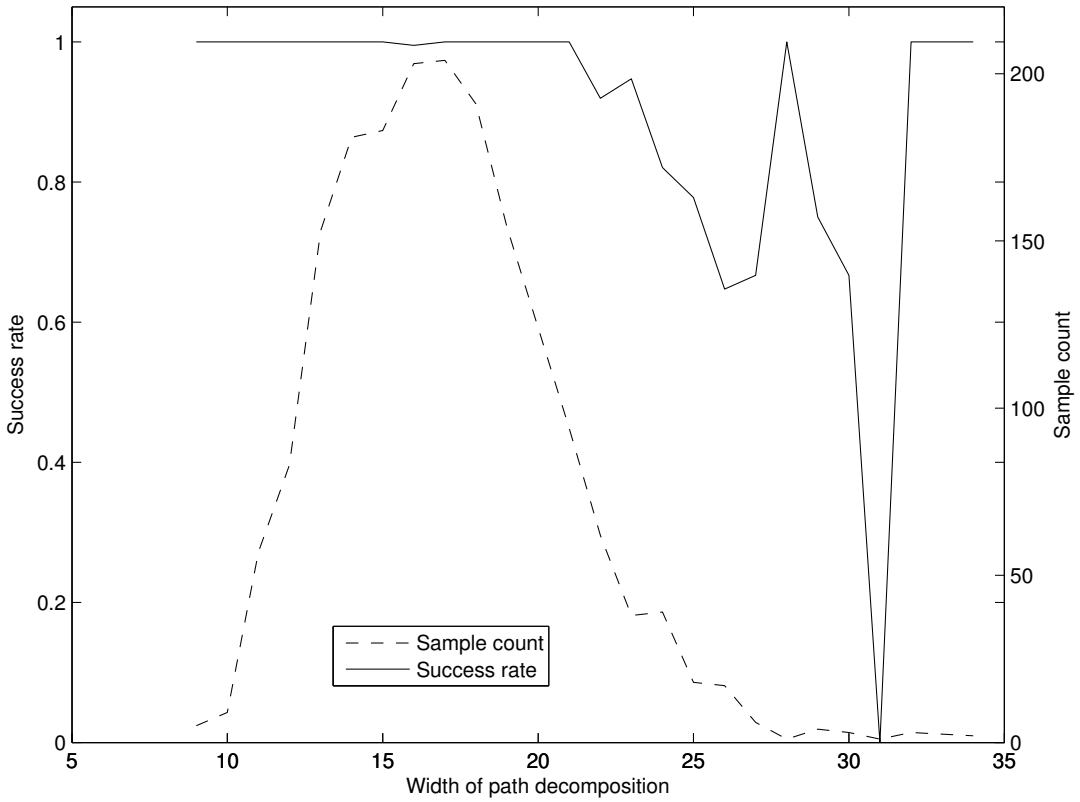

Figure 4.26: Success rate by path decomposition width

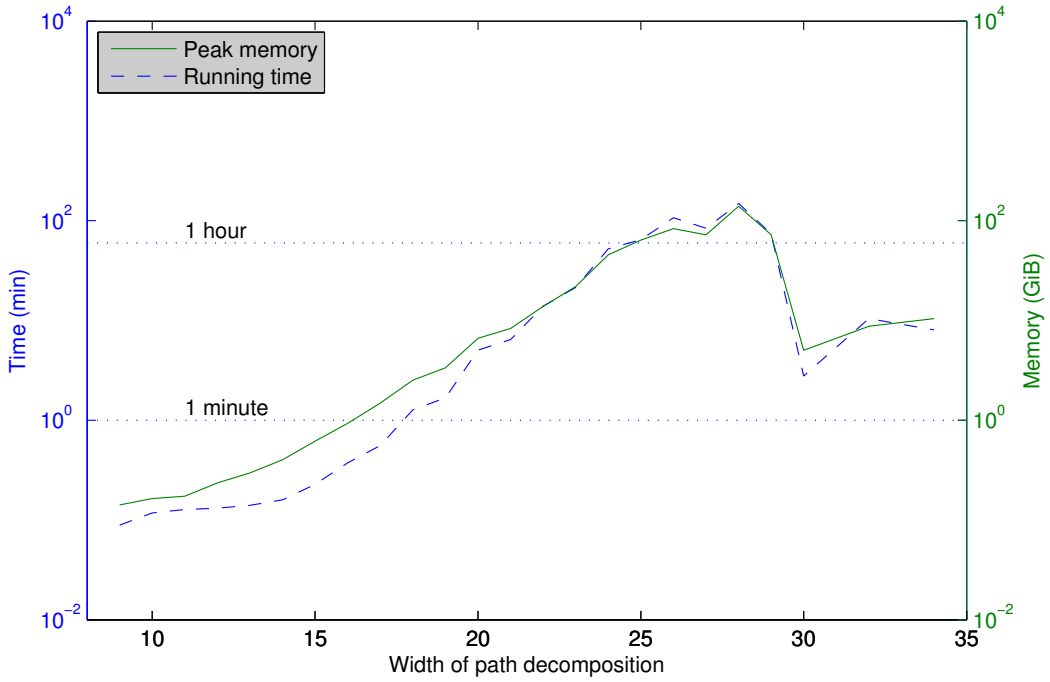

Figure 4.27: Average running time (and peak memory) by path decomposition width

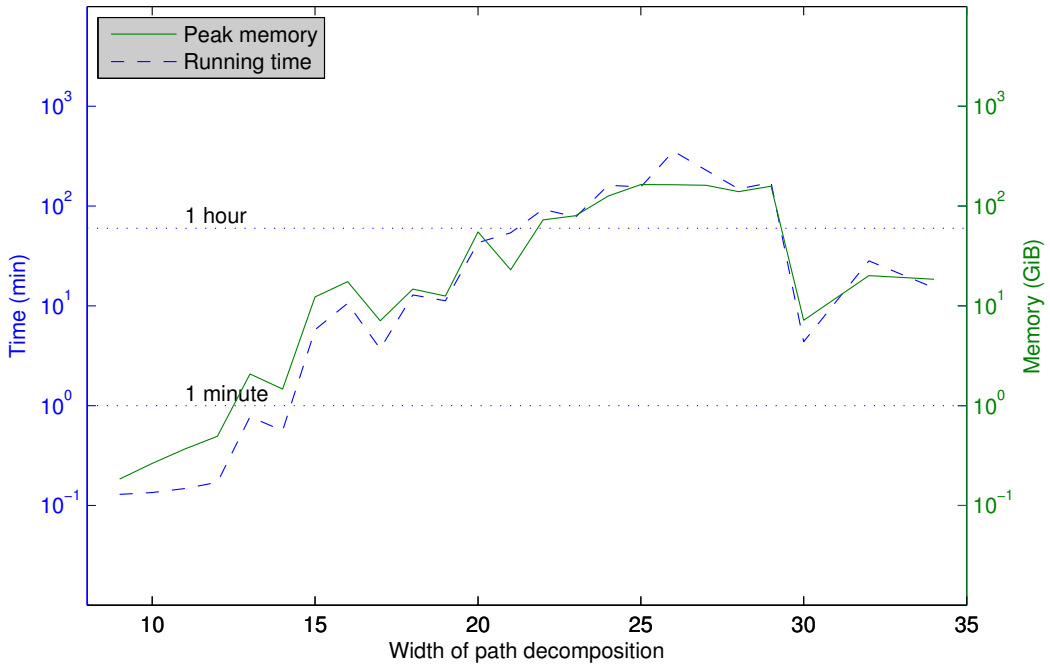

Figure 4.28: Maximum running time (and peak memory) by path decomposition width

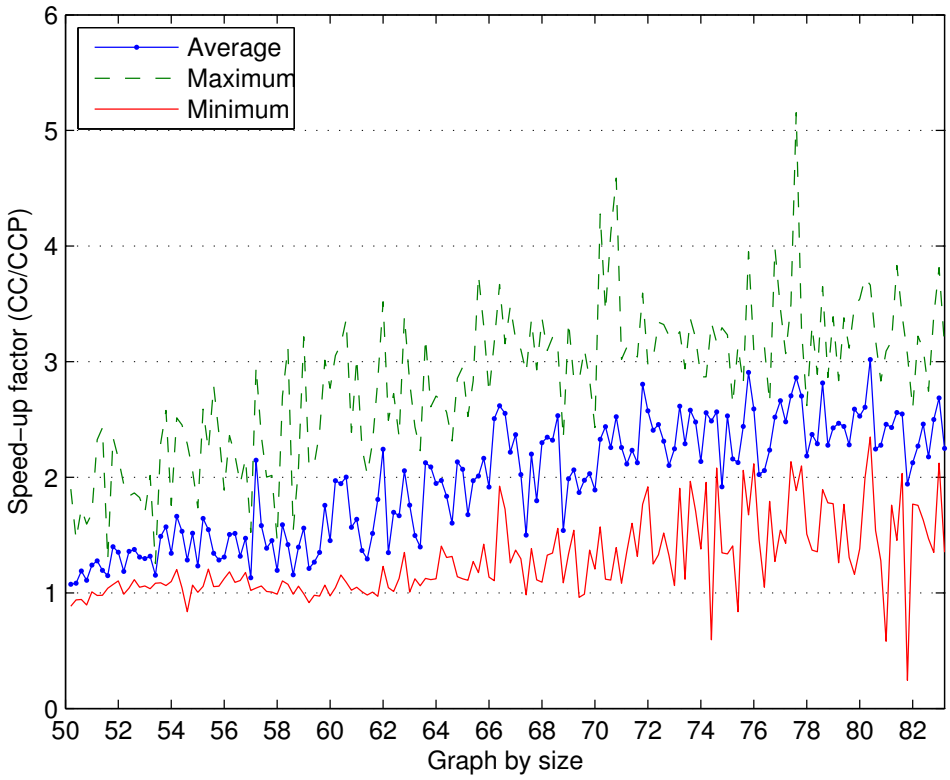

Figure 4.29: Speedup for CountColorsParallel

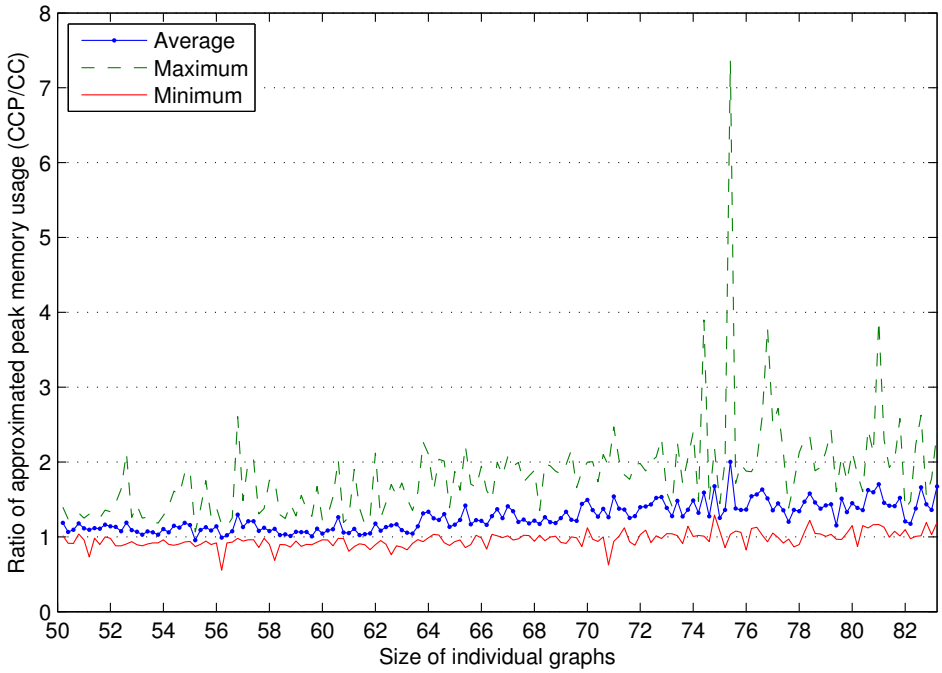

Figure 4.30: Peak memory increase for CountColorsParallel

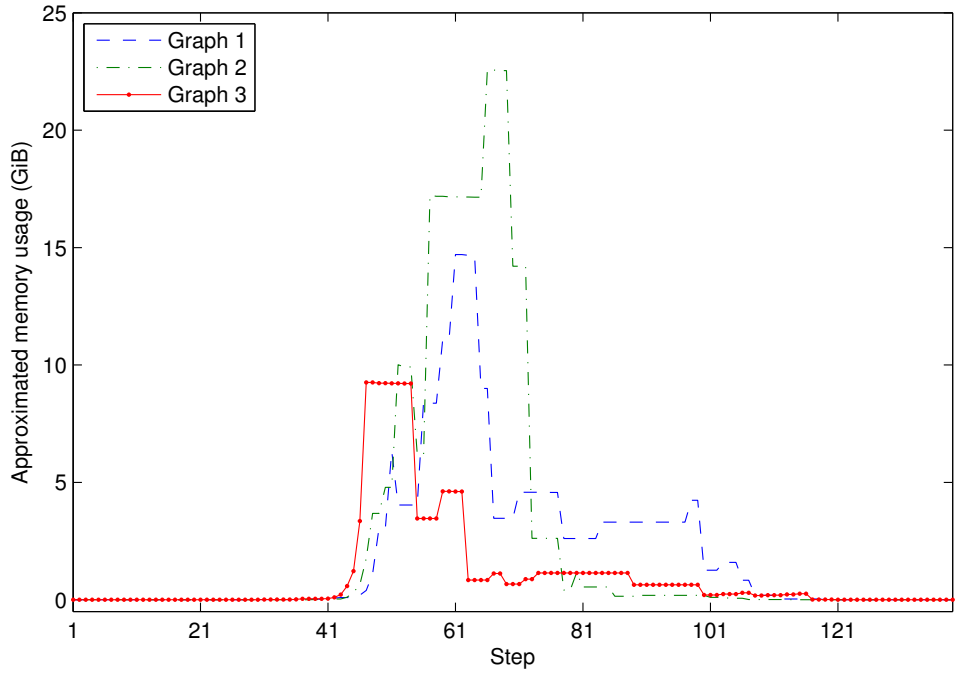

Figure 4.31: Step-wise memory usage for CountColors

4.3.4 Detecting class 2

We now turn to the class 2 set of graphs. Since determining class 2 is equivalent to counting the number of edge colorings and finding that there are none, we run both `EnumColors` and `EnumColorsParallel` (8 threads) since the latter was shown to be somewhat faster in the counting case above. Their test results may be found in figures 4.32 and 4.33 respectively. We also run `Kowalik`, shown in figure 4.34. Finally we run `CountColors` on machine 3; its results are detailed in 4.35.

Kowalik exhibits a remarkable behavior for these graphs. Almost all runs finish in less than a second. But two graphs, one of size 98 and the other of size 110, are notable exceptions which required around 16 minutes and a bit over 2 hours, respectively. A third exception, not visible in the plot, is a size 64 graph which needed 3 seconds. Additionally, Kowalik was very reliably slow for these graphs, getting similar results for every hash seed. For the size 98 graph, it varied between 935,061ms and 1,031,154ms (10% difference), and for the size 110 graph the slowest and fastest runs took 7,363,398ms and 7,972,772ms (8% difference).

At the time this was noticed, we no longer had access to machine 3, but we could still run `EnumColors` on these two graphs. It did not finish within 10 hours for any of the hash seeds for the 110 vertex graph, nor for six of the hash seeds for the 98 vertex graph. We show the running times for the remaining 4 hash seeds in 4.36. We see in two cases, that EnumColors was tremendously faster than Kowalik.

We also include two figures detailing the best 4.37 and worst 4.38 runs of `CountColors`, `EnumColors` and `EnumColorsParallel` for the lower-size graphs. While `Kowalik` was by far the fastest of all these, it is still pertinent to know how the others

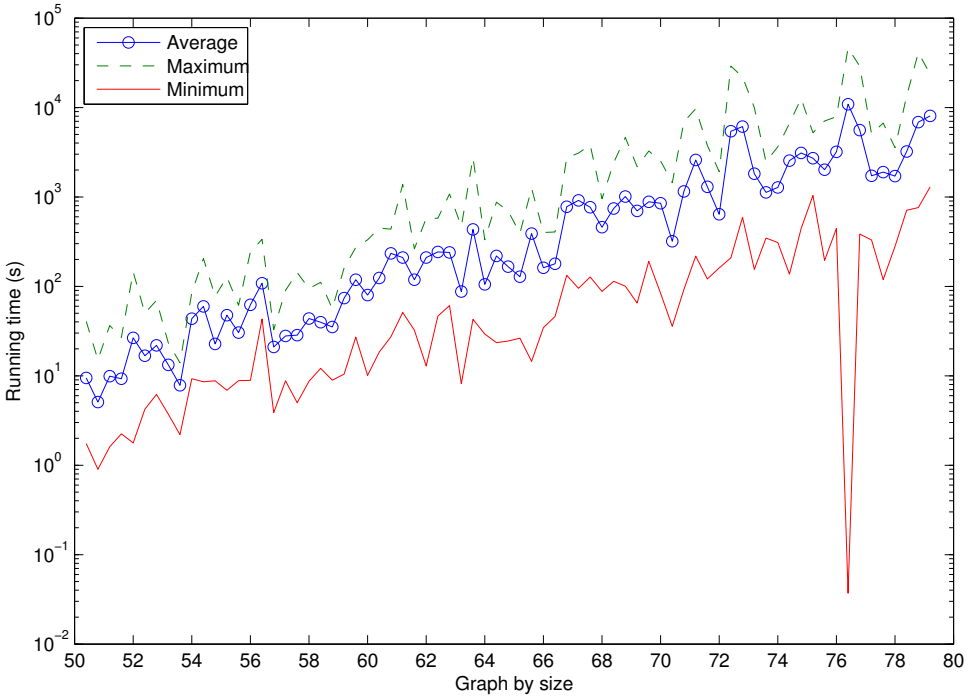

Figure 4.32: EnumColors on class 2 set

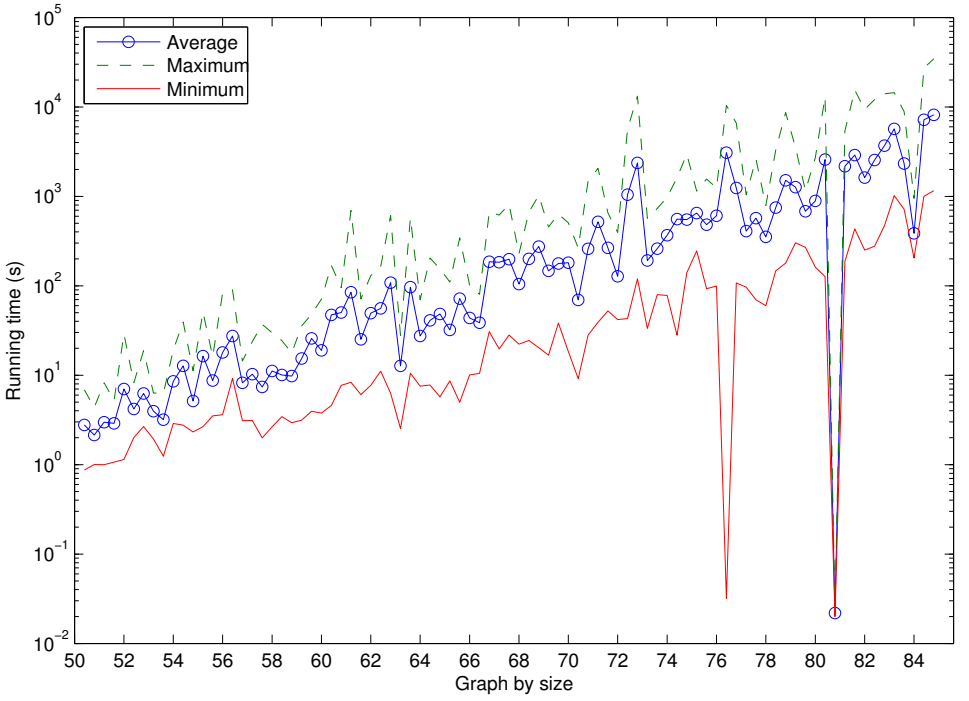

Figure 4.33: EnumColorsParallel on class 2 set

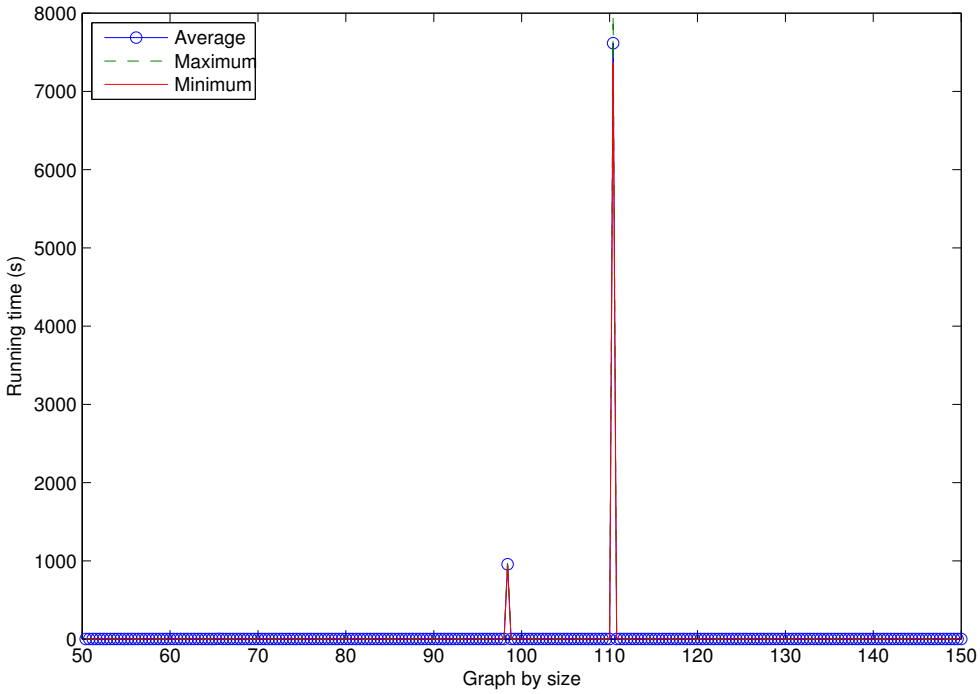

Figure 4.34: Kowalik on class 2 set

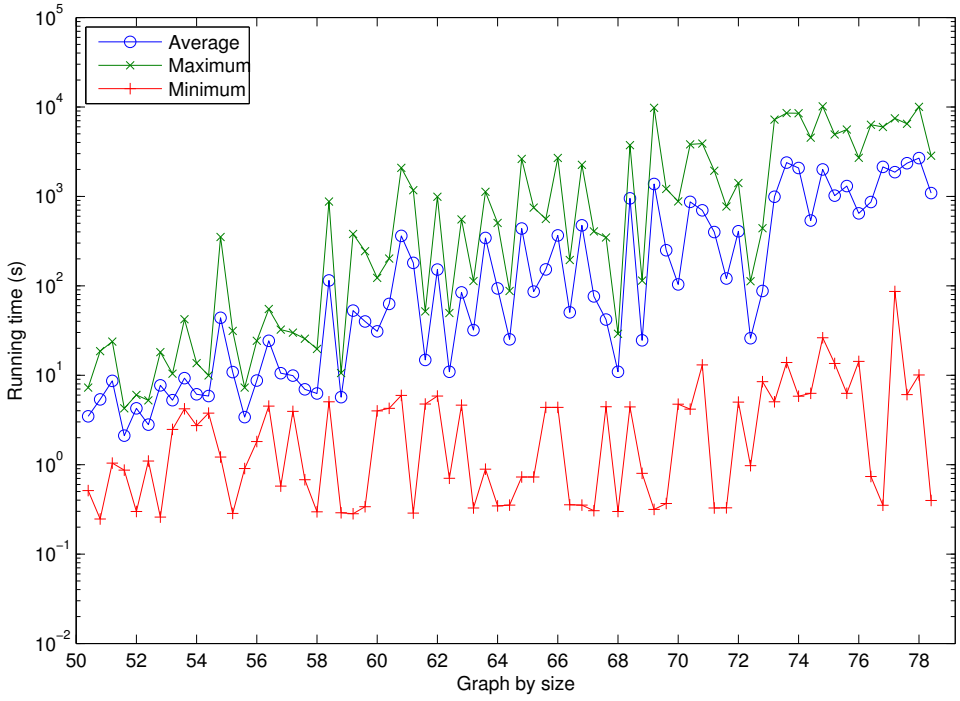

Figure 4.35: CountColors on class 2 set

	Time	Approx. time
Hash seed 2	25092817ms	7 hours
Hash seed 5	44754ms	45 seconds
Hash seed 7	560ms	< 1 second
Hash seed 9	2860259ms	48 minutes

Figure 4.36: Successful runs of EnumColors on size 98 class 2 graph

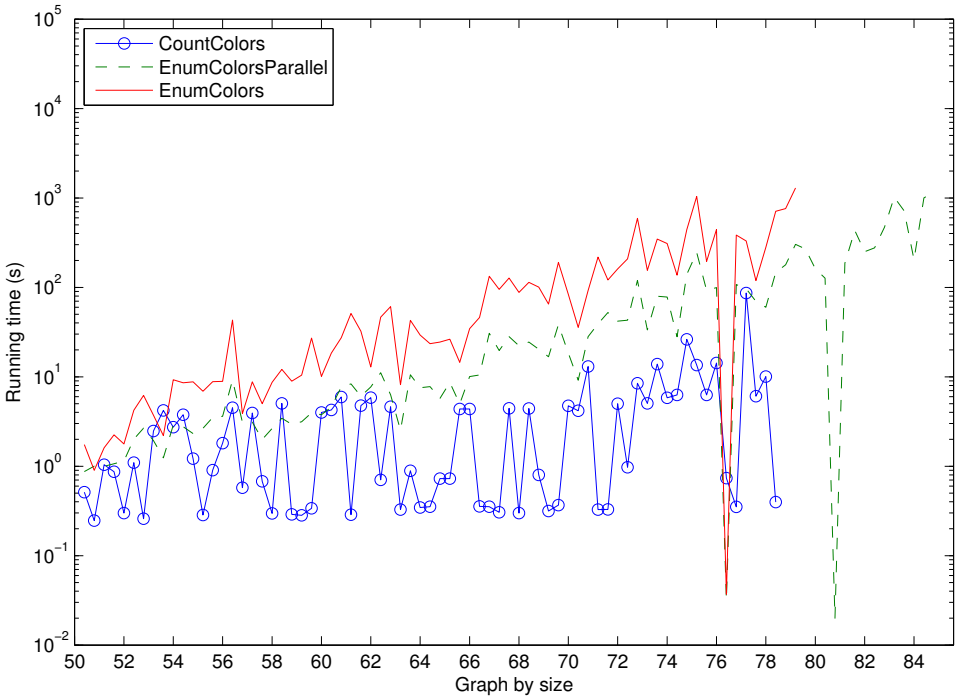

Figure 4.37: EnumColors, EnumColorsParallel and CountColors best class 2 runs

stack up. It appears that `CountColors` typically has the shortest times in the *best runs*, but not the shortest-time worst runs. That honor is shared between `CountColors` and `EnumColorsParallel`, with `EnumColorsParallel` winning out on sizes 74 and above.

4.3.5 Coping with greater sizes

In light of the fact that parallel versions of Kowalik and EnumColors did not provide much benefit for determining class, and the fact that both of them in their single-threaded forms had a low fastest time for most graphs, we devise a different (and simpler) way to utilize the parallel computing power we have access to: run several single-threaded instances concurrently, on the same graph but with different hash seeds. Since they are all calculating the same result, we only need to wait for the first thread to finish. We call these versions

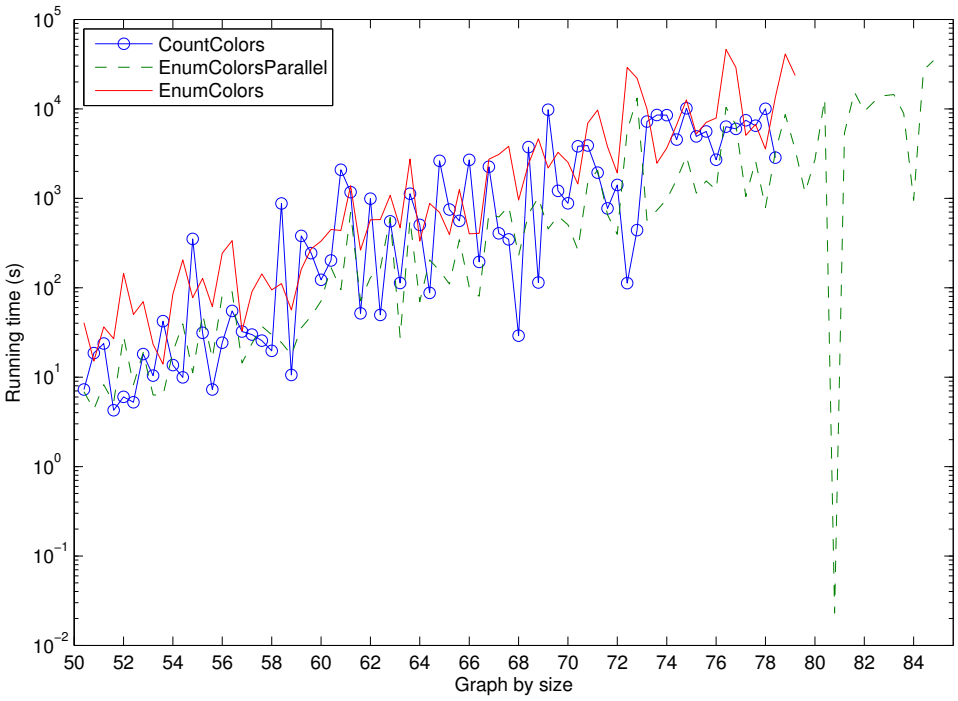

Figure 4.38: EnumColors, EnumColorsParallel and CountColors worst class 2 runs

`ManyKowalik` and `ManyEnumColors`.

In these tests, we run 7 concurrent threads. We do not use the hash values that have been pre-generated, but instead randomize new hash seeds for every thread. Due to time constraints and machine 1 and 2 being busy, these tests were run on machine 4 and given 10 hours to complete and may be found in figure 4.39. They are therefore not directly comparable to the previous Kowalik and EnumColors tests, but we include the averages of those runs anyway for ease of viewing. Re-running them will not necessarily yield the same results as the hash seeds were randomized at runtime and not saved.

`Kowalik` is slightly faster than `ManyKowalik` for small graphs. This is expected, since the times for `ManyKowalik` includes the time to start seven Java threads. `Kowalik` is soon beaten as the time to start threads becomes negligible to the total time.

The runs of `ManyKowalik` revealed another notable fact: every tested graph in the randomly generated large set was of class 1. The test did not finish on the whole large set, but ran on all graphs up to size 1290 and some of size 1300. This may be a hint that most cubic graphs are of class 1.

To see a better time complexity bound than that of Kowalik, CountColors required a small $\epsilon \leq 0.102446$ and was only guaranteed to work for graphs of size $n \geq 8565$ (2.6). We cannot expect to successfully run CountColors for graphs that large, but we can try to generate path decompositions of them and determine the effectiveness of the `aggressive` flag on very large graphs. We tried 10 different size 8000 graphs, which are not large enough to meet the bound but large enough to demonstrate the effect. The values seen in the table 4.40 are averages of 10 runs per graph.

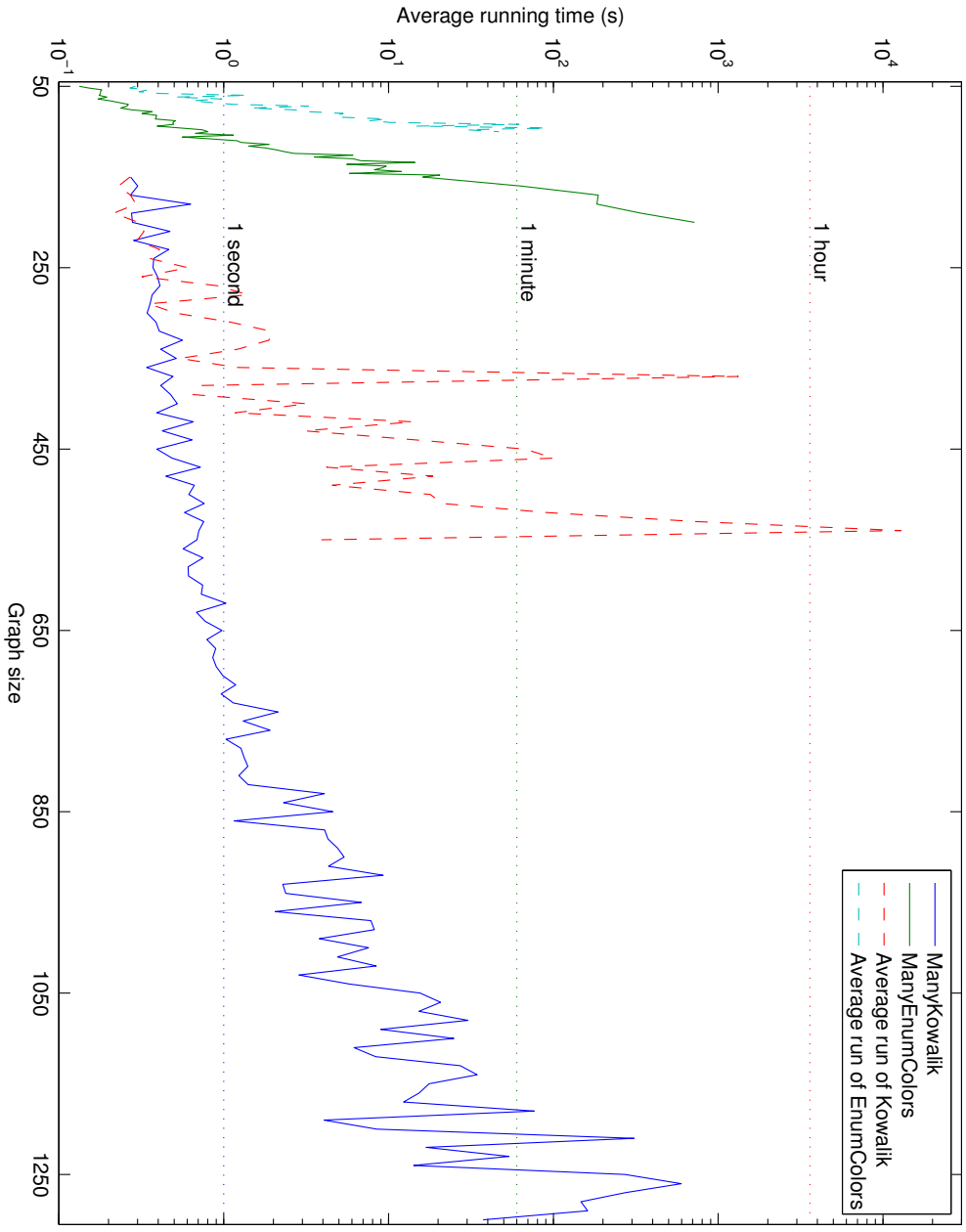

Figure 4.39: ManyKowalik and ManyEnumColors

	Non-aggressive		Aggressive	
	Width	Time (s)	Width	Time (s)
Graph 1	2168	23.13	1185	3142
Graph 2	2168	24.00	1180	3225
Graph 3	2168	23.05	1178	2935
Graph 4	2168	23.86	1181	3605
Graph 5	2168	24.20	1188	3099
Graph 6	2169	22.94	1178	3076
Graph 7	2169	23.40	1176	3190
Graph 8	2168	24.16	1165	3259
Graph 9	2167	23.32	1165	3415
Graph 10	2168	23.76	1185	3061
Average	2168	23.5	1178	3201

Figure 4.40: Efficiency of aggressive flag on very large graphs

We see that our aggressive search was able to almost halve the width of the path decompositions. The process took a little less than an hour on average.

Chapter 5
Discussion

5.1 Determining class

In our tests of small class 1 graphs, Kowalik far outperformed both versions of EnumColors; using the same hardware, EnumColors begins experiencing minute-long average running times at graphs 92 vertices, while even the worst run of Kowalik required less than a second for these and all other graphs up to 150. The standard version of EnumColors did not get past the graphs of size 100 in the original tests. Additionally, while the code modifications made to find an edge coloring took a toll on code readability in `KowalikPrint`, they do not impact running times to any noticeable degree.

Contrary to our expectations, the two parallelized versions of Kowalik do not appear to provide any substantial benefit of the single-threaded version; indeed, they were often noticeably slower. As this fact held for both, we assume that an implementation which combines the parallelization choices of both versions will fare no better. Instead, the best usage of concurrent processing power was to run several instances of `Kowalik` with different hash seeds and stop when either of them finishes. With seven concurrent instances, the program finished in less than a minute on average for all sizes up to 1170, and peaked at 10 minutes average for size 1260 graphs.

For the class 2 set, Kowalik again strongly outperformed both EnumColors and CountColors for every graph on which more than one algorithm ran, except for one hash seed of a size 98 graph. It determined the class of almost all these graphs in under a second, including every graph on which either of the other algorithms finished.

CountColors experienced a greater failure rate for the class 2 graphs than for those of class 1, because it was not as successful in finding a "better" ϵ_2 for the former set. While it finished on almost all graphs up to size 88 and about half the size 100 graphs in the class 1 set, it started to fail at a significant rate for size 74 graphs in the class 2 set. This may seem somewhat counter-intuitive; class 2 have no 3-edge-colorings and therefore should require less memory to store them. However, having no proper edge coloring does not

imply anything about the number of partial edge colorings. Evidently the number was typically larger for the class 2 graphs.

Therefore we expect it would have run out of memory for the mentioned 98 and 110 vertex graphs. Unfortunately, we had no time to verify this, due to the limited access to machine 3. However, one thing to note is that almost all graphs with successful runs had shortest run under 10 seconds. This is in line with our conjecturing that it is often possible to invalidate all partial colorings early, given a path decomposition with fortunate vertex order. As we used only five runs per graph, these path decompositions may be common. If this trend holds, we may also have succeeded on the size 98 and 110 graphs.

Overall, EnumColors performed the worst by far for the class 2 graphs. This is somewhat expected; determining a graph is class 2 with EnumColors is the same as counting all colorings, but unlike CountColors, it lacks a way to "finish early" other than by running into every dead end. It is therefore remarkable that it could finish so quickly on the size 98 graph for some seeds, for which Kowalik was so consistently slow. We do not know what caused this phenomenon.

Therefore, we strongly recommend Kowalik's algorithm for determining the class of a graph, and for finding a single 3-edge-coloring. The aberration that it was slower on the size 98 graph (and possibly the size 110 graph) seems to occur only rarely.

5.2 Finding an edge coloring

While the code modifications made to find an edge coloring took a toll on code readability in `KowalikPrint`, they do not impact running times to any noticeable degree. `KowalikPrint` finished graphs from the up-to-1500 size set at virtually the same speed as regular `Kowalik`. While we did not try to use several simultaneous `KowalikPrint` instances – a "`ManyKowalikPrint`" program – the same trend should hold for the larger graphs. EnumColors already solves the *edge coloring problem* by attempting to find a coloring. Hence, solving the *find* problem in practice is no different than determining the class for either of these two algorithms, so Kowalik is the recommended choice here as well.

5.3 Counting edge colorings

The first time that `CountColors` ran out of memory was when working on a size 62 graph, but afterwards it was mostly fine up to and including size 90 when working on the class 1 graphs. It had greater difficulty for the class 2 graphs, seeing troubling failure rates from size 74. When it did finish, `CountColors` typically outperformed both the `EnumColors` and `EnumColorsParallel` in its best runs. `EnumColors` started seeing average runs at the hour mark for size 70 graphs (size 76 for the fastest parallel version) while `CountColors` would make it past size 80.

Our recommendation for the *count* problem on larger graphs is to use the following strategy. First, ensure that the graph is class 1. Then form path decompositions, using the `aggressive` flag, for very many hash seeds. The time to form a single path decomposition for a size 100 graph grows from 5ms to about 600ms when the `aggressive` flag is

set. In our tests, a fortunate path decomposition of small width had the potential to lower running times from hours to minutes. That makes it an easy choice to spend time in this phase; we not only drastically lower the total running time, but also decrease the risk to run out of memory.

When a suitable path decomposition is found, the peak memory usage of CountColors can be estimated from the width to a reasonable degree. If we expect to have memory to spare, consider using the parallel version; in our tests it could speed up the running time by factor 2-3, at the cost of about 40% extra memory usage. If memory is expected to be scarce, use the single-threaded version.

If our strategy does not yield any path decompositions of usable width, opt to use EnumColors instead. While offering no benefit when finding a single edge coloring, `EnumColorsParallel` did outperform `EnumColors` in the counting case.

5.4 Finding all edge colorings

While we performed no tests specifically for the *find-all* problem, our observations from *count* make us consider it infeasible to use CountColors. A typical run has several tables in a row with near-peak memory usage, of which we currently only store one. Technically we do not need to remember the old *tables*; we can let every characteristic keep a list of pointers to all the immediate predecessors with an S-function that is a subset of their own. This may save some memory in those crucial steps. Still, the memory requirement is going to be tremendous. Furthermore, backtracking every possible path through the surviving chains of characteristics has its own exponential time complexity.

EnumColors therefore seems the only viable option. As the algorithm reports colorings in the rate they are found, I/O capabilities matter for this problem and the trick of running multiple concurrent instances seems counter-productive; we do not want to flood the print stream with duplicated colorings. We may, however, derive a speed-up from using the parallel version.

5.5 CountColors time complexity

Despite having 245 GiB of memory at its disposal — a significant amount by modern standards — CountColors experienced about a 50% failure rate for graphs of size 100. This is a far cry from the minimum size of 8565 (2.6) that is required to achieve a better time complexity than Kowalik. Without any modifications, running our implementation on graphs of that size could increase memory usage of up to a factor of $1.201^{8565-100} \approx 2.14 \cdot 10^{673}$ – a huge number to say the least. The *Cray Titan*, rated as one of the most powerful super computers in 2013 [TOP500], has 710 TiB of memory [ORNL] which is only approximately 2,800 times the amount we possess.

Our `aggressive` flag for path decompositions was efficient for graphs of that magnitude, sacrificing approximately half an hour on average to shrink the path decompositions from widths of about 2170 to about 1150 for size 8000 graphs. As mentioned in section 3.2 we spent no time optimizing this package; the data structure is wasteful and the algorithm actually constructs an entirely new path decomposition for every ϵ_2 rather

than re-using the previous one. With better code and an even more aggressive approach to ϵ_2 (such as bisecting between $-1/6$ and ϵ_1, rather than 0 and ϵ_1), one may potentially be able to efficiently compute path decompositions of even smaller width. Even so, the width is still very large in real terms, so regardless of code optimizations we do not expect to see an implementation of CountColors running with time complexity bounded below that of Kowalik in the foreseeable future unless theoretic improvements are made to the generation of path decompositions.

5.6 Choice of programming language

The decision to use Java was made without fully considering the implications of the exponential space complexity of CountColors. The algorithm fills a table of a large amount of fairly small objects, only to copy them up to six times and modify the copies slightly before their insertion in the next table. The originals are then thrown away. While our code was improved slightly by creating only five copies and reusing the original object as the sixth one, there is still a very large amount of objects discarded soon after their creation.

When memory usage is close to the max heap size, the JVM automatically starts the garbage collector (gc). As described earlier, we may have cases where a table is so large that the memory use is close to that limit, but still successfully create the next table of equal size by quickly removing elements from the former. However, such situations will cause very bad gc behavior: we are constantly at dangerous memory levels, and there are incredibly many objects to inspect, yet very few of these objects are actually able to be reclaimed. A language with explicit memory management, such as C++, could have eliminated this behavior, and at the same time freed up the couple of gigabytes of RAM that was reserved for the gc.

Furthermore, objects in Java carry some overhead that may be avoidable through the use of a different language. That would have allowed us to work with larger graphs, or more accurately, graphs with wider path decompositions. But this can only improve the memory footprint by a constant factor, which in the long run is a poor defense against the exponential space complexity. Realistically, we could probably only have increased the maximum graph size by a small additive constant.

5.7 Further research

Throughout the paper we have identified several topics for further research. We summarize them here for easy reference.

- The random choices, determined by hash seeds in our implementation, greatly impacted running times for Kowalik and EnumColors. Are there rules or heuristics that allow us to predict which choice will be best?
- Kowalik was quick to determine class 2, except for two graphs of size 98 and 110. What in their internal structures made Kowalik fare so poorly regardless of hash seed? Why was EnumColors able to beat Kowalik on the size 98 graph, and can we modify the Kowalik implementation to include this behavior?

- Almost every randomized graphs turned out to be class 1. Is this coincidence or are class 2 cubic graphs rare? We did not test for the properties that always form class 1 graphs (bipartiteness, or planarity plus bridgelessness). Is the perceived rarity of class 2 graphs an artefact of generating mostly graphs with these properties?
- The simple parallel architecture of CountColors is faster but sacrifices memory. How to re-balance the tables to evenly distribute workload across processors and keep memory waste to a minimum?

Bibliography

[BHK] A. Björklund, T. Husfeldt, M. Koivisto
Set Partitioning via Inclusion-Exlusion
SIAM Journal of Computing 39(2), p546-563 (2009)
doi:10.1137/070683933

[BR] H. L. Bodlaender, J. M. M. van Rooij
Exact algorithms for Intervalizing Colored Graphs
Lecture Notes in Computer Science 6595, p45-56 (2011)
doi:10.1007/978-3-642-19754-3_7

[CK] R. Cole, Ł. Kowalik
New Linear-Time Algorithms for Edge-Coloring Planar Graphs
Algorithmica 50(3), p351-368 (2008)
doi:10.1007/s00453-007-9044-3

[COS] R. Cole, K. Ost, S. Schirra
Edge-Coloring Bipartite Multigraphs in O(ElogD) Time
Combinatorica 21, p5-12 (2001)
doi:10.1007/s004930170002

[EST] J. A. Ellis, I. H. Sudburough, J. S. Turner
The Vertex Separation and Search Number of a Graph
Information and computation 113, p50-79 (1994)
doi:10.1006/inco.1994.1064

[FH] F. V. Fomin, K. Høie
Pathwidth of cubic graphs and exact algorithms
Information Processing Letters 97, p191-196 (2006)
doi:10.1016/j.ipl.2005.10.012

[GKC] P. A. Golovach, D. Kratsch, J.-F. Couturier
Colorings with Few Colors: Counting, Enumeration and Combinatorial Bounds
Graph Theoretic Concepts in Computer Science: 36th International Workshop, p39-50 (2010) doi:10.1007/978-3-642-16926-7_6

[Hara] F. Harary
Graph Theory, p85
Addison-Wesley
ISBN:0-201-02787-9

[Holy] I. Holyer
The NP-Completeness of Edge-Colouring
SIAM Journal of Computing 10(4), p718-720 (1981)
doi:10.1137/0210055

[HT] J. Hopcroft, R. E. Tarjan
Efficient Planarity Testing
Journal of the ACM 21(4), p549-568 (1974)
doi:10.1145%2F321850.321852

[Kinn] N. G. Kinnersley
The vertex separation number of a graph equals its path-width
Information Processing Letters 42, p345-350 (1992)
doi:10.1016/0020-0190(92)90234-M

[Konig] D. Kőnig
Über Graphen und ihre Anwendung auf Determinantentheorie und Mengenlehre
Mathematische Annalen 77(4), p453-465 (1916)
doi:10.1007/BF01456961

[Kow] Ł. Kowalik
Improved Edge-Coloring with Three Colors
Graph Theoretic Concepts in Computer Science: 32nd International Workshop, p90-101 (2006)
doi:10.1007/11917496_9

[KT] J. Kleinberg, É. Tardos
Algorithm Design, p94-97
Pearson Addison-Wesley
ISBN: 0-321-37291-3

[Leng] T. Lengauer
Black-white pebbles and graph separation
Acta Informatica 16(4), pp 465-475
doi:10.1007/FBF00264496

[MG] J. Misra, D. Gries
A constructive proof of Vizing's theorem
Information Processing Letters, Volume 41(3), p131-133 (1992)
doi:10.1016/0020-0190(92)90041-S

[MKW] B. D. McKay, N. C. Wormald
Uniform Generation of Random Regular Graphs of Moderate Degree
Journal of Algorithms 11, p52-67 (1990)
doi:10.1016/0196-6774(90)90029-E

[MP] B. Monien, R. Preis
Upper bounds on the bisection width of 3- and 4-regular graphs
Journal of Discrete Algorithms 4, p475-498 (2006)
doi:10.1016/j.jda.2005.12.009

[ORNL] Introducing Titan
Oak Ridge National Laboratory
http://www.olcf.ornl.gov/titan/
Accessed on Jan. 9, 2014

[PARTY] PARTY Partitioning Library
R. Preis
http://www2.cs.uni-paderborn.de/cs/robsy/party.html
Accessed Jan. 9, 2014

[SZ] D. P. Sanders, Y. Zhao
Planar Graphs of Maximum Degree Seven are Class 1
Journal of Combinatorial Theory, Series B 83, 201-212 (2001)
doi:10.1006/jctb.2001.2047

[Tait] P. G. Tait
Remarks on the colourings of maps
Proc. R. Soc. Edinburgh 10, p729 (1880)

[Tarj] R. E. Tarjan
A note on finding the bridges of a graph
Information Processing Letters 2(6), p160-161 (1974)
doi:10.1016/0020-0190(74)90003-9

[TOP500] November 2013
TOP500 Supercomputing Sites
http://www.top500.org/lists/2013/11/
Accessed on Jan. 9, 2014

[Viz64] V. G. Vizing
On an estimate of the chromatic class of a p-graph
Metody Diskret. Analiza 3, p25–30 (1964)

[Viz65] V. G. Vizing
Critical graphs with given chromatic class
Metody Diskret. Analiza 5, p9-17 (1965)

List of Figures