Heriot-Watt University

Final Year Dissertation – Deliverable One

Fast Algorithms for Hard Problems

Author:
Joseph Ray Davidson

Supervisor:
Dr. David Corne

January 10, 2011

Abstract

This document contained herein, constitutes the submission for deliverable one of the dissertation “Fast Algorithms for Hard Problems” written by Joseph Davidson. It will introduce the notion of hard problems, outline the objectives of the investigation, strategies and testing methods. It also contains a literature review that aims to present the current state of research in the area. It will close with a projected timetable for the year.

Contents

1 Introduction

In Computer Science 'hard' problems can be considered as those which are not contained within the intersection of the complexity classes P and NP (if P $\neq$ NP). Traditional computers (those consistent with the deterministic model of computation) can, in general, quickly find an exact solution to problems contained in the class P, but cannot do so for those which are outside P. This provides a problem for computer scientists, as the NP problems (defined in equation 2) are very interesting and useful problems that are considered intractable because of the time it would take to solve them [Sip05]. The best we can do for now is approximate an answer for these problems.

This project is not looking to find a fast way to get exact answers for the NP problems but to investigate the usage of greedy algorithms and machine learning to quickly approximate solutions which can be considered 'good enough' for the problem at hand.

To do this, I will be investigating one specific problem: the *vertex cover problem*[1]. This problem has a useful property that allows any fast approximation result to be used for any of the NP problems (see NP-completeness in section 1.3). This property will, if the results of the investigation are favourable, allow me to generalise my approximation algorithm to the entirety of the class NP.

1.1 Definition of NP

First, we will define the complexity class P. For this, will introduce the complexity class DTIME.

Definition 1 (DTIME)**.** ***DTIME(f(n))*** *is the class of decision problems*[2] *solvable by a Turing machine in time $O(f(n))$.*

We can now construct a definition of the class P by using DTIME...

Definition 2 (P)**.** ***P*** *is the class of decision problems that are solvable by a deterministic Turing machine in polynomial time.*

$$\mathrm{P} = \bigcup_k \mathrm{DTIME}(n^k) \tag{1}$$

If P $\neq$ NP then we can view the 'hard' NP problems as those that are not contained within the intersection of P and NP. However – because of the unclosed nature of P vs NP – we shall define the class NP in more concrete terms.

Definition 3 (NTIME)**.** ***NTIME(f(n))*** *is analogous to* DTIME(f(n)) *as it is the class of decision problems solvable by a nondeterministic Turing machine in time $O(f(n))$.*

[1] For a definition of the problem, see definition 4 in section 2.2

[2] This is not strictly true. In complexity theory, Turing machines decide *languages* instead of problems and the complexity classes are classes of languages decidable by certain Turing machines. I consider this a necessary abstraction however, as we are discussing algorithms for deciding these languages so we can refer to them as 'problems'. This will be the convention for the rest of the dissertation.

This definition allows us to state:

Corollary 1.

$$\mathrm{NP} = \bigcup_k \mathrm{NTIME}(n^k) \tag{2}$$

1.2 Why NP problems are intractable

The term NP stands for "Non-Deterministic Polynomial Time" which means that a non-deterministic computer is capable of solving a problem $n \in \mathrm{NP}$ with an asymptotic upper bound time of: $T(n) \in O(n^x)$ where $0 < x$ and x has a reasonable upper bound. This time is referred to as "Polynomial time" and we shall consider problems that can be solved in this time as tractable and problems that take longer as intractable.

Up until very recently, non-deterministic computers were more of a convenient mathematical model to help describe complexity classes. Quantum computers run with non-deterministic principles, but are in the very early stages of research [Ler10]. Consider a computation tree for NP decision problems, which is a tree that has branches for every decision point starting from the starting state at the root, ending with either an accept or reject state at the leaves.

We can picture a deterministic computer D simulating a nondeterministic computer N as performing a breadth first search, expanding each level of the tree until it finds an accepting state. On an input of length n, each branch of the computation tree has a length of at most $t(n)$. Each node in the tree can have at most b children so we can deduce that the number of leaves in the tree is at most $b^{t(n)}$. We can use this to establish an upper bound on the total number of nodes in the tree: $O(b^{t(n)})$ because the total number of nodes in a tree cannot be more than twice the number of leaves. As said before, the time taken to expand a branch of the tree is at most $O(t(n))$ therefore, the running time for the deterministic computer D is $O(t(n)b^{t(n)}) = 2^{O(t(n))}$ which we can generalise to $T(n) \in O(2^n)$ [Sip05].

We can think of a non-deterministic machine using the same computation tree in two ways:

- A Machine that can expand entire tree levels in the same time it takes a deterministic machine to expand one node.
- A Machine that always expands tree nodes on an accepting path.

In the first case, we can see that the non-deterministic machine is $t(n)$ times faster than the deterministic one. In the second, we can think of the non-deterministic machine as the luckiest deterministic machine in the universe because it always expands the correct nodes in the tree that lead to an accepting state [Sip05].

It is simple to see that for a problem space that is large (say $n = 4000$), a deterministic machine would have a worst case running time of: $T(4000) \in O(2^{4000})$ which is unacceptable. Hence the need for fast algorithms for approximating the solution to many of these problems.

1.3 NP–completeness

There is a subset of the NP problems in which every problem in NP can be reduced to one of these problems. This set is called the NP-complete problems, defined in equation 3 below.

$$\text{NPC} = \{x \mid x \in \text{NP} \wedge (\forall y \in \text{NP} : y \prec x)\} \tag{3}$$

Where x and y range over the problems in NP and $y \prec x$ means: "There exists a polynomial Karp reduction from y to x" [BKMT75][Coo71].

The NP-complete problems are a good place to start finding a fast approximation algorithm because of the many-one reducibility property of the problems in the set. If a fast approximation algorithm can be found for one of the problems, that algorithm can be used as a subroutine for solving all of the other NP problems.

The vertex cover problem itself is an NP problem, which is one of the motivations for it being the problem of choice for my investigation. If I were to find an algorithm that could solve the p

2 Project objectives

The main objective of this project is to attempt to determine a method of producing approximations for NP problems – such as the vertex cover problem – very quickly. Because the NP problems are a famous class in the field of complexity, many approaches have been tried to obtain optimal results quickly. These methods have a problem which is that they often depend on stochastic methods which generally do not produce results that are always reproducible, making multiple runs of the programs a necessity. My attempt at this problem will try to use more deterministic methods. The use of greedy algorithms for making fast approximations will be investigated.

2.1 Strategies

As mentioned above, there are many strategies that can be investigated with respect to finding a quick approximation for a vertex cover. The strategies that I wish to focus on in this investigation – as well as reasons for choosing them – are below.

- Greedy algorithms.
- Machine learning techniques – such as Genetic algorithms or ANNs.
- Graph preprocessing.
- A combination of the above.

2.1.1 Greedy algorithms

Greedy algorithms have been used extensively in the investigation into NP problems. Due to their nature, some greedy algorithms are more successful than others – for instance, I present below a very simple greedy algorithm to obtain an approximate vertex cover [FHH+09]

1. Given an undirected Graph $G = (V, E)$.
2. Set $V' = \emptyset$.
3. Repeat:
    - Choose a vertex v_k having the largest degree in G.
    - Set $V' \triangleq V' \cup \{v_k\}$, $V \triangleq V \setminus \{v_k\}$, and $E \triangleq E \setminus \{e \mid e \cap v_k \neq \emptyset\}$.
4. Until G is empty.

This algorithm simply selects the vertex with the highest degree in the graph, adds it to the vertex cover and then removes it from the graph, along with the edges incident with it. The algorithm quickly produces a result in polynomial time, but the result is not very accurate.

It is the eventual aim of this project to produce an algorithm that still quickly produces a result, but that result is also highly accurate. The usage of greedy algorithms is a possibility for achieving this goal, but it appears that there is a speed – accuracy trade off for all such algorithms that attempt to solve NP problems. A balance will have to be struck to make greedy algorithms a viable solution.

2.1.2 Machine learning

Machine Learning techniques have been successfully applied to problems in many fields. Genetic algorithms have designed NASA antennae [LLH+03] and have been applied to solving the n-queens problem (another NP problem) [Cra92]. Artificial Neural Networks have been used to autonomously control vehicles and studies have conducted into the effectiveness of these systems (see ALVINN [BP97]).

If an appropriate learning function can be devised, training these algorithms to produce optimal results could be a possibility. With the right implementation of these algorithms, all that would be needed is a reliable set of training examples. Xu, Boussemart et al. [XBHL05] have devised models that could be used to create hard NP instances with a fixed solution. These problems and answers can form the basis of training the learning algorithms using instance based learning (discussed in section 5.3).

2.1.3 Graph preprocessing

There are methods for preprocessing the input that may improve the chances of obtaining an optimal solution. A technique for this is *Kernelization* [Bod09]. Kernelization works by removing clauses (in the case of the Satisfiability problem) that trivially have to be true in order for there to be an optimal solution. Possessing this knowledge allows us to shrink the input to a smaller equivalent by removing these clauses.

Kernelization techniques were originally envisioned to be used more for programs that use brute force searching to find solutions, but they may have applications to assist greedy algorithms that might get ‘confused’ when searching for solutions in large search spaces.

2.2 Problem selected

As mentioned above, the problem that I have selected is the vertex cover problem which is defined below:

Definition 4 (Vertex Cover Problem)**.** *Given an undirected graph $G = (V, E)$, find a subset $V' \subseteq V$ of minimum cardinality such that for each $e \in E$, $e \cap V' \neq \emptyset$ holds [FHH+09].*

Informally: Given a Graph $G = (V, E)$, we aim to find a set V' of the minimum number of vertices $v \in V$ required such that every edge $e \in E$ is adjacent to at least one vertex in V'.

This problem has been demonstrated to be NP-complete by Karp [BKMT75] using a reduction from the Satisfiability problem, first shown to be NP-complete by Cook [Coo71], to the problem of finding maximal cliques in a graph and then reducing the clique problem to the vertex cover problem.

I have selected this problem primarily because I have previous experience with it and it is also one of the most studied problems in graph theory so there is a vast amount of background research to draw on.

3 Discussion of technical literature

The vertex cover problem is one of the most studied in graph theory. To this end, there is plenty of technical literature on multiple aspects of the problem including general approximation algorithms, type-specific algorithms, the mechanics behind the problem and more. Because of the wide variety, I will focus on the algorithms that have been developed.

3.1 Existing algorithms

The vertex cover problem has had several algorithms developed for it, most of which are sensitive to the topology of the input. For *bipartite graphs*[3] an exact vertex cover can be found in polynomial time using an implementation of the Edmonds–Karp algorithm [EK72].

For general graphs, the process to write an exact algorithm becomes much harder. Bar-Yehuda and Even [BYE81] wrote a 2–approximation algorithm for the *weighted vertex cover*[4] which runs in linear time and gives an approximation ratio that is shown in equation 4 (where $n = \mid V \mid$).

$$2 - \frac{\ln \ln n}{2 \ln n} \tag{4}$$

[3]A bipartite graph is one in which the vertices can be split into two sets, with each edge having vertices in both sets

[4]Weighted cover associates a cost to each vertex and the algorithm attempts to use minimisation techniques with the cost to produce a minimised valid cover.

20 years later, Karakostas [Kar05] improved the original algorithm 2–approximation so that it produced an approximation ratio detailed in equation 5. This ratio is only a very slight improvement over the original one, but it is an improvement nonetheless.

$$2 - \Theta(\frac{1}{\sqrt{\ln n}}) \tag{5}$$

In 2005, Safur and Dinur [DS05] proved that it is hard to approximate a vertex cover within a ratio of 1.3606. . . for sufficiently large graphs unless P = NP.

3.2 Annotated bibliography of relevant technical literature

Due to the pure research nature of this topic, reading previous research papers will be essential. These papers will serve two purposes:

1. Provide essential background knowledge.
2. Reveal new techniques or insights that can improve the speed or optimality.

A list of the documentation that I have identified so far as relevant to the project is below, referenced and annotated.

This list is not meant to be exhaustive, more texts that require reading may be discovered throughout the investigation. If that is the case, they will be added to the main bibliography listing that will be included with the final dissertation submission.

Introduction to the Theory of Computation, Michael Sipser [Sip05].

- This text serves as a course in establishing a basic and essential ground knowledge in the field of computer theory. It is considered a landmark text and is ideal for somebody who wishes to investigate the field further.

Reducibility Among Combinatorial Problems, Richard M. Karp et al. [BKMT75].

- This paper introduced the concept of "Karp's 21 NP-complete problems". It demonstrated that there were 21 distinct combinatorial and graph theory problems which were all members of the NP-complete class. He did this by showing that they were in NP and then reducing the Satisfiability problem to an instance of them.

The complexity of theorem-proving procedures, Stephen A. Cook [Coo71].

- This paper introduced the notion of NP-completeness to the broader audience of computer scientists (although the term was not used in this paper - it was coined later) and generated interest in the notion of NP-complete problems. This spurred other researchers – amongst them, Richard Karp and Leonid Levin – to produce works of their own in the field.

  The paper concerns itself with constructing a Turing machine to calculate the Boolean Satisfiability problem. It shows that this construction will take a time that is greater than polynomial time to calculate a satisfying assignment and that there was no trivial way to speed up this calculation.

A Simple Model to Generate Hard Satisfiable Instances, Ke Xu, Frederic Boussemart et al. [XBHL05].

- This paper describes a method to generate hard instances of the Satisfiability problem using model RB. It also analyses the hardness of problems generated by the models, both with forced and unforced answers.

Machine Learning, Tom M. Mitchell [Mit97].

- This text, whilst perhaps superseded by others, provides a solid foundation in the theory and practise of machine learning techniques. It may not be up to date, but the fundamental principles behind machine learning stay the same and this text explains them thoroughly and allows the reader to go on and learn the more modern techniques found in recently published papers.

Analyses of simple hybrid algorithms for the vertex cover problem, Tobias Friedrich et al. [FHH+09].

- This paper does some small experimentation with creating hybrid algorithms from combining genetic and greedy algorithms together and asymptotically analysing them with respect to the performance of the vanilla algorithms on their own.

Kernelization: New Upper and Lower Bound Techniques, Hans Bodlaender [Bod09].

- This paper is an introduction and summary of the current research into kernelization techniques. It broadly extols the virtues of using kernelization for reducing the search space of a given NP problem and gives examples and citations to other papers of these techniques being used.

Kernels for the vertex cover Problem on the Preferred Attachment Model, Josep Daz et al. [DPT06].

- This paper uses results from Buss [BG93] and Nemhauser & Trotter [NT75] to present algorithms that attempt to kernelize instances of the vertex cover problem. It explains each algorithm and performs a side-by-side comparison of their effectiveness at finding a kernelization for graphs based on the preferred attachment model.

4 System requirements

As this dissertation focuses on a research topic, the resulting system will have few usability requirements. Instead, the requirements focus is more on the criteria associated with the efficient and effective execution of the approximate vertex covering procedure:

1. The system must produce a result quickly – that is, the system must have a running time of no more than a polynomial with regard to the input. An approximation method that is slower than this could be considered intractable and is of little use.
2. The system must produce a valid result. It must produce a result that can be verified as a vertex cover.
3. The system must produce a result that is within a certain percentage of the optimal result. The result should be considered ‘good enough’ for the speed at which it is produced.

5 Initial system design

I have already written a deterministic greedy algorithm that takes a graph and quickly produces an accurate, valid vertex cover – with about $1\% - 2\%$ variance over the optimal when processing a 4000 vertex, 572774 edge graph in about 15 seconds on a modern computer.[5]

I would however, prefer to get a more accurate result using this algorithm combined with other techniques, such as those described above or others. The ideal eventuality would be to have a variance of $< 1\%$ whilst still keeping a very high speed.

This kind of optimality may be unfeasible due to the findings of Safur and Dinur, in which case I will attempt to obtain an estimate as close to this boundary as possible.

5.1 Existing data structure

The current system uses a special structure for graphs that will be detailed below:

- Graph:
  - Contains a 'master' list of vertices and edges which contains all the instances of the vertices and edges.
  - Each vertex and edge is an instance of a structure/class and they are stored in the list by an integer id.
  - Contains other variables that are used to keep track of the state during the covering procedure.
- Vertex:
  - Contains a list of pointers to the edge structures contained in the graph edge list for the adjacent edges.
  - Contains an id, a Boolean variable for determining whether it is part of the cover or not and other variables to record the state of the vertex.
- Edge:
  - Contains two fields for the two vertex ids that the edge spans.
  - Contains other variables to record the state of the edge.

This rather complicated representation for graphs takes up more space, but it allows for very quick access to the structure which prevents time bottlenecks and allows us to evaluate the covering algorithm on its own merits, rather than confusing time taken to perform the covering operation with time taken to retrieve the graph components.

It will most likely be kept for the project as it provides excellent access times and is easily implementable in most languages.

[5]This algorithm was written as part of some self guided independent research in 2008 – during my 2nd year of university.

5.2 Existing algorithm

The algorithm itself processes this special graph structure by the edges which allows it to consider two vertices at a time. Each edge and vertex have a common Boolean variable: *marked*, so that when a vertex has been marked, it is part of the vertex cover and when an edge is marked, it is adjacent to a vertex which is a member of the vertex cover.

Vertices also have another variable *visited* which signifies if a vertex has been considered by the algorithm before. This is the way the program resolves the nondeterministic situation of two vertices having the same number of adjacent unmarked edges: if vertex is visited and the other one is not, the algorithm will always add the already visited vertex to the cover.

This can be considered as one of the major issues with the algorithm. How do we know that by taking one vertex instead of the other, we are not condemning ourselves to a sub-optimal result? We do not, but it is easy to arrange things so that the algorithm will run itself again, selecting the other vertex the next time around every time an issue like that occurs. However for each one of these 'swaps' found, the algorithm will have to *re-run itself again for all the 'swaps' done previously*. Therefore, for n occasions of swapping, the algorithm has to run 2^n times.

For large graphs with many interconnected vertices, n can be massive, which means that the program will have to perform a brute force search over the field of vertex swaps to find the most optimal.

The algorithm has a set of heuristics for each combination of marked and unmarked vertices it will come across. It uses these rules to obtain a greedy cover in a single pass of the graph and then reprocesses the graph by unmarking every vertex that is totally surrounded by marked vertices. This method produces a fast vertex cover that is quite accurate.

5.3 Planned improvements

I will likely produce several programs to solve the vertex cover problem and there are, for now, two methods that I plan to use to improve the optimality and/or speed of the covering algorithm (see Section 2.1).

The first is to add one of the kernelization techniques detailed in [Bod09]. Doing this will reduce the problem space of the input whilst preserving the 'essence' of the graph. One issue with this is that the process requires a factor k that it uses to kernelize the input. I will need to devise a method to work out k which produces a kernelization for the graph and *eliminates only the vertices that must be included in the optimal cover*. Using a statistical method to devise a k which eliminates (say) all the vertices that have an indegree of 2 or more standard deviations above the mean could be an option. More investigation into the topic is required.

The second method will combine the greedy process with machine learning techniques to create a hybrid algorithm. If I introduce more stochastic variables to the program (such as weighting rules with probabilities), I can train the algorithm to hopefully produce more optimal solutions based on the current state and attributes of the graph such as mean indegree, # vertices covered, how connected the graph is, etc.

Of course, training a machine to do this requires a supervisor that knows the answer to the problem. The question is this: *How can we train a program to solve NP-complete problems without knowing the answer to specific problem instances ourselves?*

In theoretical computer science, there are machines that know the answer to such problems: Oracles. In practice, these machines are unfeasible but Xu and Li while researching exact phase transitions, developed a model that can create a *forced satisfiable instance* of the Satisfiability problem [XL06]. This model enables us to create instances of NP-complete problems with varying difficulty and fixed answers. This means that our 'Oracle' can generate instances with a fixed answer, allow the learning algorithm to attempt to solve it, compare the answers and tune the learning algorithms rules.

6 Testing strategies

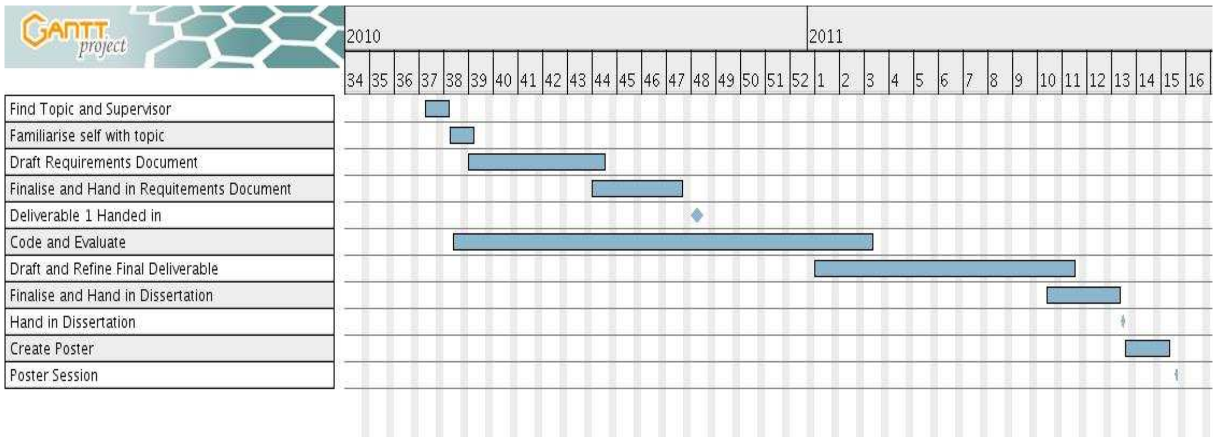
As there is little in the way of interaction with the user, there is no need to test any interface elements. Testing the actual candidate algorithms will involve checking the logic of the code and performing traces on smaller examples of graphs where possible so that I can assert that the code runs correctly and produces an optimal answer for these examples. Once this is done, the code can be evaluated.

Evaluating the algorithm will involve using model RB to generate satisfiable instances with an exact answer. The algorithm will then be run on this set and the time taken and accuracy will be recorded, perhaps some charts will be drawn. The algorithm will be asymptotically analysed and a running time will be derived.

In the case of having multiple algorithms, they will all be individually tested and evaluated and will be compared with each other side by side.

I will also try to discern the approximation ratio formula for the algorithms, although this may be infeasible for the time scheduled because of the complexity of the algorithms and the amount of work I already have to do.

7 Year timetable



Name	Start	End	Milesto
Find Topic and Supervisor	15/09/10	22/09/10	false
Familiarise self with topic	22/09/10	29/09/10	false
Draft Requirements Document	27/09/10	05/11/10	false
Finalise and Hand in Requitements Document	01/11/10	27/11/10	false
Deliverable 1 Handed in	29/11/10	04/12/10	true
Code and Evaluate	23/09/10	20/01/11	false
Draft and Refine Final Deliverable	03/01/11	18/03/11	false
Finalise and Hand in Dissertation	10/03/11	31/03/11	false
Hand in Dissertation	31/03/11	01/04/11	true
Create Poster	01/04/11	14/04/11	false
Poster Session	15/04/11	16/04/11	true

References

[BG93] Jonathan F. Buss and Judy Goldsmith. Nondeterminism within p^*. *SIAM Journal on Computing*, 22(3):560–572, 1993.

[BKMT75] Ronald V. Book, Richard M. Karp, Raymond E. Miller, and James W. Thatcher. Reducibility among combinatorial problems. *The Journal of Symbolic Logic*, 40(4):618+, December 1975.

[Bod09] Hans Bodlaender. Kernelization: New upper and lower bound techniques. In Jianer Chen and Fedor Fomin, editors, *Parameterized and Exact Computation*, volume 5917 of *Lecture Notes in Computer Science*, pages 17–37. Springer Berlin / Heidelberg, 2009. 10.1007/978-3-642-11269-02.

[BP97] S. Baluja and D. A. Pomerleau. Expectation-based selective attention for visual monitoring and control of a robot vehicle. *Robotics and Autonomous Systems*, pages 329–344, December 1997.

[BYE81] R. Bar-Yehuda and S. Even. A linear-time approximation algorithm for the weighted vertex cover problem. *Journal of Algorithms*, 2(2):198 – 203, 1981.

[Coo71] Stephen A. Cook. The complexity of theorem-proving procedures. In *STOC '71: Proceedings of the third annual ACM symposium on Theory of computing*, pages 151–158, New York, NY, USA, 1971. ACM.

[Cra92] Kelly D. Crawford. Solving the n-queens problem using genetic algorithms. In *SAC '92: Proceedings of the 1992 ACM/SIGAPP symposium on Applied computing*, pages 1039–1047, New York, NY, USA, 1992. ACM.

[DPT06] Josep Daz, Jordi Petit, and Dimitrios Thilikos. Kernels for the vertex cover problem on the preferred attachment model. In Carme lvarez and Maria Serna, editors, *Experimental Algorithms*, volume 4007 of *Lecture Notes in Computer Science*, pages 231–240. Springer Berlin / Heidelberg, 2006. 10.1007/1176429821.

[DS05] Dinur, Irit and Safra, Samuel. On the Hardness of Approximating Minimum Vertex Cover, 2005.

[EK72] Jack Edmonds and Richard M. Karp. Theoretical improvements in algorithmic efficiency for network flow problems. *J. ACM*, 19:248–264, April 1972.

[FHH$^+$09] Tobias Friedrich, Jun He, Nils Hebbinghaus, Frank Neumann, and Carsten Witt. Analyses of simple hybrid algorithms for the vertex cover problem*. *Evol. Comput.*, 17(1):3–19, 2009.

[Kar05] George Karakostas. A better approximation ratio for the vertex cover problem. pages 1043–1050, 2005.

[Ler10] Eric J. Lerner. A quantum leap for computing. `http://domino.watson.ibm.com/comm/wwwr_thinkresearch.nsf/pages/quantum4%98.html`, 2010.

[LLH$^+$03] Jason D. Lohn, Derek S. Linden, Gregory S. Hornby, William F. Kraus, and Adaan R. Arroyo. Evolutionary design of an x-band antenna for nasa's space technology 5 mission. In *EH '03: Proceedings of the 2003 NASA/DoD Conference on Evolvable Hardware*, Washington, DC, USA, 2003. IEEE Computer Society.

[Mit97] Tom M. Mitchell. *Machine Learning.* McGraw-Hill Science/Engineering/Math, 1 edition, March 1997.

[NT75] G. L. Nemhauser and L. E. Trotter. Vertex packings: Structural properties and algorithms. *Mathematical Programming*, 8:232–248, 1975. 10.1007/BF01580444.

[Sip05] Michael Sipser. *Introduction to the Theory of Computation.* Course Technology, 2 edition, February 2005.

[XBHL05] Ke Xu, Frederic Boussemart, Fred Hemery, and Christophe Lecoutre. A simple model to generate hard satisfiable instances, Sep 2005.

[XL06] Ke Xu and Wei Li. Many hard examples in exact phase transitions. *Theor. Comput. Sci.*, 355(3):291–302, 2006.