Universiteit Leiden
The Netherlands

# Opleiding Informatica

Hybrid Classical-Quantum Speedups for the Random $k$-SAT Problem using Small Quantum Computers

Miguel Oswaldo Blom

Supervisors:
Mathys Rennela
Vedran Dunjko

BACHELOR THESIS

Leiden Institute of Advanced Computer Science (LIACS)
www.liacs.leidenuniv.nl

30/07/2020

**Abstract**

Real-life instances of SAT problems can be of rather large size, which means that even the most space-efficient known quantum algorithms cannot be implemented on the still size-limited state-of-the-art quantum computers. This thesis fits in a line of work which focuses on achieving possible speedups (over classical computing) through the use of smaller quantum computers. We explore a hybrid classical-quantum approach, the Divide and Quantum approach. In this approach, Divide and Conquer classical algorithms are adapted to work with quantum devices. Specifically, given a quantum computer whose size is a ratio of the size of the input, one substitutes recursive calls to classical algorithms by calls to faster quantum algorithms, whenever the size limitations allow. The question of what the minimal requirements on the quantum computer size to ensure that the Divide and Quantum approach yields a polynomial speedup depends on the problems studied. This thesis focuses on an empirical study of the search trees generated by the classical algorithm DPLL for the random 3-SAT problem, in which Boolean formulas are randomly generated. In other words, we investigate in the context of SAT when the Divide and Quantum approach yields a computational advantage over a classical Divide and Conquer approach. For this purpose, we develop software specifically designed for collecting data about the shape and the size of the search trees generated by DPLL. We analyze this data and determine settings in which the Divide and Quantum approach seems to achieve a polynomial speedup over classical implementations of DPLL. More specifically, we conjecture that the Divide and Quantum approach beats the classical when the ratio of the quantum computer is above 90% in general, and can be significantly smaller in more special cases.

# Contents

# 1 Introduction

Quantum Computers (QCs) are becoming more popular as current research unlocks the possibility for bigger and more reliable machines. They are used for solving certain problems, which are thoroughly researched to this day. Boolean Satisfiability Problem (SAT) is one of the problems in which a lot of research has been done and is still ongoing, because many real life problems can be translated to problem instances of SAT. SAT has many variants, such as random $k$-SAT, which are called *classes*. However SAT is a hard to solve problem, it might take a time exponential in the number of variables, even for the best current algorithms.
There exists a Quantum Algorithm (QA) for which it has been proven that a QC achieves a quadratic speedup on these classical algorithms in specific cases. Solving SAT using a Classical Backtracking (CBT) algorithm results in search trees, these search trees can be explored quadratically faster using a QA. Solving the problem still takes an exponential amount of time, but polynomially less compared to the classical case, which may be of great value in practice. However, we have to deal with the fact that the current QCs are very limited in size, the problem instances do not fit inside the QC in terms of available working memory. For speedups to be attainable, the search trees have to satisfy certain conditions; namely, the subtrees, defined by the size of the available QC, must be of an exponential size in relation to the size of the problem instance itself. These search trees can also be heavily unbalanced, which renders some generally used Quantum Algorithms (QAs) inefficient.
Divide and Quantum (D&Q) is a family of hybrid algorithms with classical and quantum elements. D&Q runs classically on the top of the search tree to reduce the SAT problem, up until a given cutoff point. When the cutoff point is reached, the reduced problem is guaranteed to fit inside the QC for which the cutoff point was determined. D&Q then proceeds to run quantum-wise on the subproblems that were reduced by the classical element and allows the utilization of QCs in solving problems that do not initially fit inside the QC.

Using a smaller QC we can obtain a significant speedup only in specific cases. As we will explain later, this depends on the size of the search trees that the QC operates on, which may prohibit any genuine speedup. This leads to the question of this thesis:

> *"Can we identify classes of instances of random $k$-SAT problems and a ratio of the size of the original problem instance to the size of the smaller quantum computer, for which, on average, a reasonable speedup can be obtained?"*

Whether or not a speedup is obtainable depends on the tree structure and QC size. Here we study DPLL for random $k$-SAT to identify regimes of size, and random $k$-SAT specifications where polynomial speedups are obtainable.

We believe that instances of hard classes of random $k$-SAT generate trees on which much smaller, 50% the size of the problem instance, QCs can still offer a polynomial speedup. We use random $k$-SAT because it allows for an additional parameter $\Delta$, used to tune the hardness of the problem instances that we want to examine, to experiment with different classes of random $k$-SAT with different shapes of search trees. In particular, we focus on random 3-SAT, for which there exist two

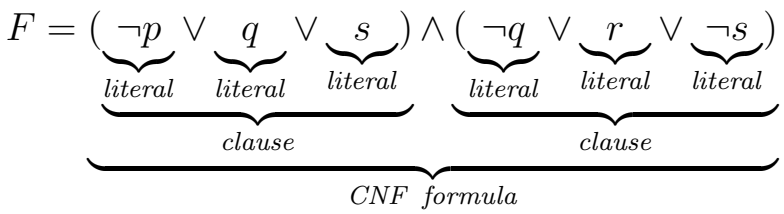



Figure 1: An example of a formula with two clauses and three literals per clause. For example, $p = False$ and $r = True$ would suffice as a satisfying assignment

theoretically proven thresholds[1] [BSW01, PI00, ABM04], which define three different regimes of different hardnesses of corresponding problem instances. For all $\Delta$ in between 3.003 and 4.3 (the *hard regime*) the problem instances are hard; they are easy for any other $\Delta$. Problem instances of random 3-SAT with a $\Delta$ in the hard regime are known to take an exponential amount of time to solve. In our research, we will be looking at speeding up a conventional variant of the Davis-Putnam-Logemann-Loveland (DPLL) family of algorithms, which are Classical Backtracking (CBT) algorithms.

To explore our research question, we will be analyzing the shape and size of the search trees and its subtrees corresponding to runs of DPLL on classes of random $k$-SAT; in particular, we will focus on $\Delta$ in the hard regime. We will report which $\Delta$ yield exponentially sized search trees with exponentially sized subtrees at certain fixed cutoff points. Secondly, we will examine how the hardness of initially hard formulas evolves as we reduce them using DPLL to partial problems. Lastly, we will be examining the effectiveness of a theoretical quadratic speedup on problem instances with different $\Delta$ . These three parts will provide supporting evidence to our hypothesis and show the importance of using QCs for solving SAT problems and, in general, using D&Q. We examine runs of DPLL and collect information about the shape and size of the search trees in order to simulate the effects of a QC with theoretically proven properties.

## 2 SAT Problem

A Conjunctive Normal Form (CNF) formula in Boolean logic [HR04] is constructed by using three logical operators and a given set of variables. A *variable*, such as '$x$', is a unit to which we can assign a value, either $True$ or $False$. The logical *negation*, 'NOT' ($\neg$), negates the value of a Boolean logic expression, it swaps $True$ to $False$ and vice versa. The logical *conjunction*, 'AND' ($\wedge$), results in $True$ if and only if the expressions, which it operates on, result in $True$, otherwise $False$. The logical *disjunction*, 'OR' ($\vee$) results in $True$ if and only if either of the expressions on which it operates results in $True$, otherwise $False$. A logical *literal* is either a non-negated or negated variable ($x$, $\neg x$ respectively). A logical *clause* (constraint) is a disjunction of literals and a CNF formula is a conjunction of multiple clauses. An example of a CNF formula is shown in Figure 1.

The Boolean Satisfiability Problem (SAT) is a problem where the task is to determine whether a problem instance, a Boolean logic formula, is satisfiable. This is the case if there exists a satisfying

[1] A threshold is constant value we define to identify distinct classes by separating instances with values above from those with values below the threshold.

assignment, i.e. a setting of variables, which renders the entire formula $True$. An algorithm that solves SAT should return 'Yes' only for satisfiable formulas and 'No' otherwise. SAT problems are often in CNF, a satisfying assignment would evaluate every clause being $True$.

If a contradiction like $(p) \wedge (\neg p)$ is found, then there exists no satisfiable assignment for the problem instance. As we will discuss later, the best classical algorithms solving SAT have an exponential time complexity; its instances can, in the worst case, take an exponential amount of time to solve, by the best algorithms, in relation to its size.

## 2.1 3-SAT

The SAT problem has many versions, one of which is the $k$-SAT class where our problem instance has a fixed number ($k$) of literals in each clause. In this thesis we will be focusing on 3-SAT, which means that each clause in the formula contains $k = 3$ literals. We use 3-SAT because it is the most challenging case for D&Q.

### 2.1.1 Random 3-SAT

For the class of $k$-SAT we have a subclass of random $k$-SAT (or random 3-SAT in our case). The difference between $k$-SAT and random $k$-SAT is that in random $k$-SAT we specify the generation of problem instances to be a method where the formula consists of a fixed number of clauses, which are chosen uniform at random (i.e. it is not necessarily a worst case instance for our algorithm). We have a concept of a set of all possible clauses that are allowed by the class, defined by the number of literals in a clause as $k$ and a set of variables of size $n$ which may occur in the formula. The total number of combinations of variables in this pool is $c_{lit} = \binom{n}{k}$, the number of possible combinations of literals for each clause is $c_{neg} = 2^k$. Thus the total number of combinations of literals is $c_{tot} = c_{neg} \cdot c_{lit} = 2^k \cdot \binom{n}{k}$. The way we combine clauses in this research is by choosing a number of clauses, with replacement, possibly resulting in having duplicate clauses in a formula.

Specifically in random $k$-SAT, one can characterize each instance with a parameter $\Delta$, which denotes the ratio $\#(Clauses)/n$ and it is known that this parameter can influence the hardness and search tree structure of the corresponding problem instance [CM01]. Higher $\Delta$ results in variables occurring more among clauses. Having variables shared among multiple clauses brings dependencies between these clauses for assignment, thus making a satisfying assignment less likely to exist. Each $\Delta$ together with the number of variables results in a different subclass of random $k$-SAT.

For random 3-SAT two thresholds are defined [CM01], resulting in three regimes; intuitively, when we say that a regime or a formula is hard, we mean that best known algorithms take exponential time (in $n$) to decide if a solution exist. For $\Delta < 3.003$ the problems are known to have many satisfying assignments and therefore it is "easy" to find one. In general, for $3.003 < \Delta < 4.3$ there are very few satisfying assignments, but most often they do exist. To find these, current methods search through many combinations in order to find a satisfying assignment, making this the "hard" regime of random $k$-SAT problems. For $4.3 < \Delta$ we know that there are often no solutions at all. So in our search we can see by the contradictions that our partial solution will not lead to a satisfying assignment early on. This makes this regime "harder" than the easy regime, but not as "hard" as

the hard regime. In the case of our research, we will correlate to the size of the search space we need to explore in order to decide the complexity using backtracking algorithms.

### 2.1.2 Sampling algorithm

In order to generate random $k$-SAT problem instances, it is required to clearly define what our approach is, as there exist a number of variations. Also, randomness in computer science has some notable complications in its definition.

A conventional computer (unlike a quantum computer) is a *deterministic automaton*, meaning that given a state $q$, and an action $a$, the resulting state $r$ from applying $a$ on $q$ ($q \xrightarrow{a} r$) will always be identical for the same $q$ and $a$. Thus given a state of a process (for the generation of random numbers) on the deterministic automaton will result in the same random number each time. We can algorithmically approximate randomness and call this **pseudo randomness** where "pseudo" is often omitted.

To generate random $k$-SAT problem instances, we order all $c_{tot}$ possible clauses by giving each of them a unique identifier. Next we generate $\#(Clauses) = \Delta * n$ random numbers between $0 \leq r_n < c_{tot}$. We use these to pick clauses from the pool by selecting the clauses with the unique identifiers equal to the generated random numbers. A more precise definition of the generation is presented in the appendix, this does not have to be fully understood in order to comprehend our used methods.

## 2.2 Complexity

Complexity is a term that is used to describe the growth of resources required by a procedure as a function of the size of the input ($n$). In complexity, we make use of big-$\mathcal{O}$ notation to determine a set of functions specified by whatever function is within big-$\mathcal{O}$; these functions describe the growth rate $f(n)$ of the required resources as a function of $n$. For example, we describe this behaviour using $\mathcal{O}(g(n))$ where $f(n) \in \mathcal{O}(g(n))$. $f(n) \in O(g(n))$ means that there exist $n_0 \geq 0$ and $c \geq 0$ such that $0 \leq f(n) \leq c \cdot g(n)$ for all $n \geq n_0$. Most often, $g$ will not consists of multiple terms as $\mathcal{O}(g(x)) = \mathcal{O}(g(x) + \text{lower order terms})$. Orders of functions in big-$\mathcal{O}$ notation include: constant ($\mathcal{O}(1)$), logarithmic ($\mathcal{O}(\log(n))$), linear ($\mathcal{O}(n)$), n-log-n ($\mathcal{O}(n \log(n))$), quadratic ($\mathcal{O}(n^2)$), polynomial ($\mathcal{O}(n^m)$), exponential ($\mathcal{O}(exp(n))$) and factorial ($\mathcal{O}(n!)$).

**Time complexity** describes the amount of elementary computational steps[2] resources are needed by a program. We use steps to describe time complexity for it being machine independent. Different machines may have different speeds, this makes measuring the actual time unreliable for describing the computational time complexity of an algorithm. This would become a machine dependent multiplicative term, swallowed by big-$\mathcal{O}$ For example, in SAT if we would use a backtracking algorithm to explore the search space resulting in a binary tree, where each node represents an assignment (step), then in the...

[2]The essential operation in an algorithm which must have a fixed constant time. For example, for an algorithm on solving SAT, the elementary computational step is an assignment of a value to a variable.

1. ... worst case[3], we would perform as many steps as the number of nodes in a full binary tree with a height of the number of variables in the formula. This results in an exponential amount of steps ($\mathcal{O}(2^n - 1)$). All possible assignments were covered until either the very last satisfies to true, or it did not and there exists no satisfying assignment at all.

2. ... best case[4], we would perform as many steps as the number of nodes in a minimally sized binary search tree with a height of the number of variables in the formula. This results in a linear amount of steps ($\mathcal{O}(n)$). Here the very first assignment would satisfy the formula.

**Space complexity** describes the memory resources needed by an algorithm. Suppose we have an algorithm solving SAT, each variable gets either assigned $True$ or $False$, which requires at least exactly one bit per variable for any algorithm in order to solve the problem. If the algorithm requires no extra bits, then its space complexity is thus linear in the size of the problem instance.

The best known classical algorithms for constraint satisfaction problems such as 3-SAT have an exponential time complexity, and their quantum counterparts also have an exponential runtime, although polynomially smaller. In particular, we do not suspect that quantum computers can solve 3-SAT in a polynomial amount of time, but we know they can provide a valuable polynomial speedup for almost all cases. In other words, the complexity will be exponential, but polynomially smaller than in the classical case. [BV97].

## 2.3 Search algorithms

Search algorithms are procedures that explore possible candidates in a search space in order to find a solution to a problem, such as SAT. When comparing search algorithms there are four attributes that can be discussed. A search algorithm can either *terminate* or not, which means that it either does or does not stop running at some point for any finite search space[5]. A search algorithm is *complete* if it obtains a solution if there exists one in the search space. A search algorithm can be *admissible*, which is the same as completeness, but the solution should be optimal[6] in this case. Lastly, space and time *complexity* play a big role in choosing which kind of search algorithm should be used for a specific problem. The algorithms we use to solve SAT should always terminate and must be complete.

### 2.3.1 Brute-force

Brute-force search is a family of exhaustive search algorithms for finding a solution to a problem. The general idea of brute-force is to generate **all**[7] possible solutions and checking them one by one. In SAT, given $n$ variables, there are $2^n$ possible assignments provided that $n > 0$. Thus the time complexity of brute-force on SAT is $\mathcal{O}(2^n)$ in the worst case. SAT problem instances may have a structure that allows for the use of faster algorithms than brute-force, this is the case for 3-SAT.

[3] An upper bound for problem instances with the highest possible number of elementary operations.
[4] A lower bound for problem instances with the lowest possible number of elementary operations.
[5] A finite set of candidate solutions to the problem.
[6] An optimal solution is the best solution as specified by the problem definition. For example, given a minimization problem, a solution to the problem is optimal if there exists no other solution with smaller "costs".
[7] Hence exhaustive.

### 2.3.2 Search space

A search space consists of all (partial) solutions that satisfy all constraints[8]. A search algorithm describes the exploration of this space. We express the exploration for SAT in the form of a tree structure, a search tree. This search tree consists of nodes, corresponding to partial assignments to the original random $k$-SAT formula in the root node. It also consists of edges between parents node and their children (a subformula / subproblem). These edges correspond to the transitions from a formula to a subformula respectively by a reduction by assigning a value of $True$ or $False$ to a variable in the formula. These partial assignments serve as partial solutions to the problem. The leaf nodes (at the endings of the tree) of a fully explored search tree correspond to either a contradiction or a satisfying assignment to the problem. In our problem instances, these search trees can be **unbalanced**, which means that not all leaf nodes lie on the same depth (See Figure 6). Worth mentioning is that the differences in depths can be relatively large to the height of the tree. This means that big parts of the trees may not be explored, due to contradictions or a not fully explored search tree.

### 2.3.3 Backtracking

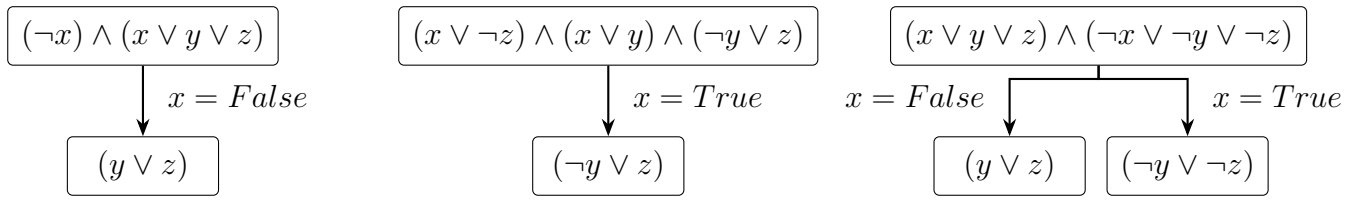
Classical Backtracking (CBT) is a family of classical search algorithms for solving problems that allow evaluation on partial solutions. For SAT we can evaluate partial solutions on the formula to see whether any contradictions occur. In the case of a node in the search tree with a contradiction, the whole subtree of the corresponding node can be ignored; the contradiction will persist in all of its children and it is thus guaranteed to not contain any satisfying assignment. By ignoring the subtree, CBT *prunes* the search tree, resulting in the mentioned imbalance due to contradictions. A CBT algorithm calls itself recursively in order to visit children of the current node in the search tree. In the case of SAT, this is adding the assignment of a value to a variable to the partial assignment. When the algorithm can not or should not visit any child nodes, the algorithm returns to the call on the parent node, ending up back higher in the search tree, until it finds unvisited nodes. This goes on until a satisfying assignment is found, which is returned by the algorithm. The time complexity of a CBT corresponds to the size of the search tree. As mentioned, in the worst case this is $\mathcal{O}(2^n)$ and the best case $\mathcal{O}(n)$.

Brute-force and CBT are approaches of search algorithms that can be generalised for other search problems. Why we prefer CBT over brute-force, is that CBT searches through the search space more strategically. This results in a lower time complexity, in the average case, as a result of pruning disposable parts of the search tree.

## 2.4 DPLL

The Davis-Putnam-Logemann-Loveland (DPLL) [DP60, DLL62, Ouy98] algorithm family consists of backtracking algorithms used to solve SAT problems. A *heuristic* is a type of ruleset that determines which decisions are made based on some evaluation given the original formula and current partial assignment; here the decision is the order in which we explore partial assignments. A partial assignment is a state in the form of a subformula which corresponds to a node in the search

[8]Conditions which a (partial) solution to the problem must satisfy in order to be valid.




(a) Unit propagation (b) Pure literal elimination (c) Branch

Figure 2: Reduction rules and branch operation for DPLL

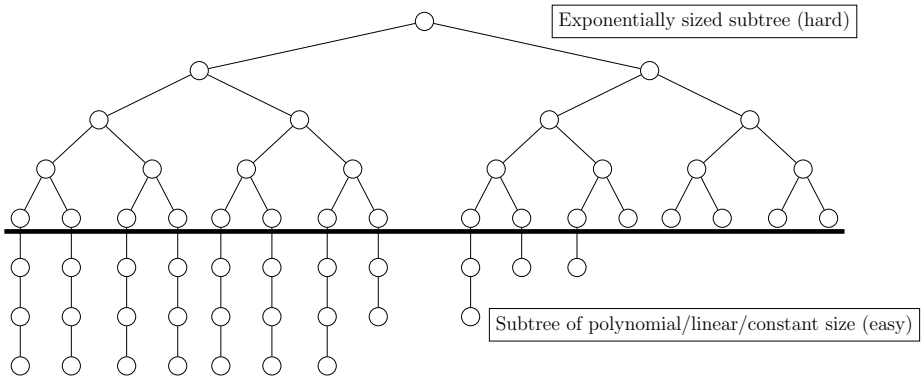



Figure 3: Growth regions of the subtrees of an example search tree of random $k$-SAT formulas.

tree. The heuristics for DPLL can be chosen freely, thus making DPLL a **family** of algorithms. There are two reduction rules that are generally used in DPLL:
Unit Propagation (UP) detects whether a clause with one element exists in the current subformula and assigns a value to the corresponding variable as to render the clause $True$; see Figure 2a.
Pure Literal Elimination (PL) detects whether there exists a variable which occurs negated in all clauses or non-negated, i.e. they occur in one polarity[9] in all clauses. If such variable exists in the current subformula, a value is assigned to it as to render the clauses to be $True$; see Figure 2b.

If neither of the reduction rules can be applied to simplify the formula, then some *branch heuristic* is used. We select the variable corresponding to the label which comes first in the sequence of all used labels (of which their corresponding variables are unassigned) in canonical order. We will call this method NaïveDPLL. The order in which we select a variable, using the heuristic, is critical for performance, hence this is NaïveDPLL. Because the clauses are chosen uniformly at random, the labels of the variables are also at random. By selection using any order we will have random selection. A branch operation is performed as seen in Figure 2c.

A variable is chosen for which both $True$ and $False$ will potentially be evaluated. If a contradiction is found, then the program backtracks higher up to a branch. If no more unexplored regions are found in the search tree, then there exists no satisfying assignment.

### 2.5 Thresholds for random 3-SAT

As mentioned earlier, the thresholds for $\Delta$ in which the hard regime falls are 3.003 and 4.3 [CM01]. These thresholds have been experimentally witnessed in previous work [BFS19, Zho, AP03, MPRT16, Nik05, AHI06]. The whole search trees are likely to be exponential in size in relation to the height of the subtrees, very likely because of its upper region. After some assignments, the subformula is likely to become smaller, such that the bottom region of the search trees often become polynomial, then linear and in the leaves constant in size in relation to the height of the subtrees; see Figure 3. Polynomial sized trees in relation to the height meaning that the function describing the size of the tree in amount of steps is polynomially sized as a function of the amount of variables. This means that a solution for a problem with a polynomially sized search tree can be found in a polynomial amount of time. Also notable is that the height of the tree is a function of the number of variables, because at each step, we eliminate at least one variable. If variables, present in a set of clauses being satisfied in one step, are not present in other clause that remain, then we can eliminate multiple variables at once. A QA, with a theoretical quadratic speedup on problems with an exponential time complexity, would not be effective for problems with a polynomial complexity considering the polynomial overhead needed to use the QC. This is because running the query of the search tree using the QA can actually be slower than the classical algorithm, but the overall number of queries can be much (polynomially) smaller. A search tree that grows exponentially in size in relation to its height and thus number of variables do have an exponential complexity allowing said QA to be effective, even considering the overhead.

## 3 Quantum Computing

Our methods requires an understanding of used concepts from quantum computing [NC09]. QCs are machines built with the goal in mind for computation while exploiting phenomena from quantum mechanics. *Superposition* is the phenomenon where a particle is represented as a linear combination of multiple computational basis states at the same time, until it collapses in only one of the states, upon measurement. *Entanglement* is the phenomenon where the states of all particles in a paired set are much stronger correlated than a classical system allows. For instance, measuring a particle making it collapse also makes its paired particles collapse. *Interference* is the phenomenon where the quantum wave functions of particles interact constructively. A classical computer does not depend on any of these phenomena and makes use of only classical physics. QCs allow for much faster calculations in specific cases relative to classical computers.
The problem remains that QCs are currently resource-sensitive and complex in comparison to classical computers. Current QCs do not have enough working memory in order to solve real-life sized SAT problems. The overhead of using a quantum computer should also be considered while talking about practical speedups; where theoretical speedups might sound good, they might not be good enough in a practical manner. In other words, if a low theoretical speedup is guaranteed for solving specific classes and instances of SAT problems, it might not be valuable enough to outweigh the overhead.

[9] Being negated or non-negated.

## 3.1 Quantum Bits

A Quantum Bit (qubit) is similar to a classical bit in the way that it describes a unit of information. While the state of a classical bit is a discrete value, either 0 or 1, the state of a qubit is represented by a unit vector in a 2-dimensional complex vector space, the Hilbert space. Any valid quantum state can be given with $\alpha|0\rangle + \beta|1\rangle$. The coefficients $\alpha$ and $\beta$ in this expression are called the the *amplitudes* of the state. They describe the superposition of the qubit, In particular, they describe the probabilities of finding a qubit in either the $|0\rangle$ or $|1\rangle$ state upon measurement with probabilities $|\alpha|^2$ and $|\beta|^2$ respectively. The expression $|\alpha|^2 + |\beta|^2 = 1$ should always hold; the probabilities of finding the qubit in either state upon measurement should add up to 1. We can think of the amplitudes as the polarization of a photon, which is a valid physical encoding of a qubit.

Measuring a qubit will collapse its superposition state of $\alpha|0\rangle + \beta|1\rangle$ to either '0' or '1'. In terms of linear algebra, the measurement operation projects a state $|\phi\rangle$ onto either of the vectors $|0\rangle$ or $|1\rangle$ based on the probability given by the amplitudes of the state.
If the system contains multiple Quantum Bits (qubits), the collection of qubits is called a *quantum register*, allowing for quantum entanglement. When having a quantum register of $n$ qubits with for each qubit two different outcomes upon measurement, the state space dimension grows exponentially, $\mathcal{O}(2^n)$. Which means that adding more qubits allow for exponential growth in the number of states we can represent using one quantum register.

## 3.2 Quantum Gates

Quantum gates, like classical gates, perform an operation on the state of the register. In quantum computing, this is not defined by a Boolean logic operation, but rather a linear transformation in a vector space. All quantum gates can be described by using matrices that transform the state of the quantum register. Earlier in this section we mentioned that all quantum operators should be reversible, this is to preserve the amplitudes of the states. The state after applying a unitary gate can be expressed using linear algebra; it is obtained by applying the corresponding matrices onto the vector representing the initial register of the qubit system. Composing operations as a circuit with gates is represented by the tensor product. In other words, any quantum circuit is a representation of an exponentially large unitary transformation.

An example of a gate is the Hadamard gate, which is used to put qubits into superposition, if they were in a computational basis before. The Hadamard matrix is defined by: $H = \frac{1}{\sqrt{2}}\begin{bmatrix} 1 & 1 \\ 1 & -1 \end{bmatrix}$, where for instance applied to $|0\rangle$ and $|1\rangle$ respectively: $H|0\rangle = \frac{|0\rangle+|1\rangle}{\sqrt{2}} = \frac{1}{\sqrt{2}}|0\rangle + \frac{1}{\sqrt{2}}|1\rangle$ and $H|1\rangle = \frac{|0\rangle-|1\rangle}{\sqrt{2}} = \frac{1}{\sqrt{2}}|0\rangle + \frac{1}{\sqrt{2}}|1\rangle$. Taking the amplitudes $\alpha = \beta = \frac{1}{\sqrt{2}}$ from this equation and squaring them gives us the probability of observing either state upon measurement. A probability of $|\frac{1}{\sqrt{2}}|^2 = \frac{1}{2} = 0.5$ to encounter either '0' or '1' upon measurement, thus the Hadamard gate puts these vectors (qubit states) in an equal superposition for the states $|0\rangle$ and $|1\rangle$. This superposition is a common initialization, after which we use interference to amplify "good" branches of solutions. The use here is that given a superposition of all possible states, we apply operations in a way that only viable solutions that respect all conditions can result upon measurement.

## 3.3 Quantum Algorithms

The methods we will be referring to are based on QAs. An algorithm is a description or a recipe that determines a specific sequence of applications of transformations in order to solve a specific problem, given an input, where the input size determines the actual sequence of transformations. Therefore a QA specifies a sequence of quantum gates applied on the input qubits. After the application of the whole circuit, in the final state, a measurement is done on one or more qubits. From this measurement we read the result of the algorithm, given the input. The methods we will be mentioning both make use of superposition. In what follows we will explain a particular QA for the Quantum Search problem, which is the problem of detecting whether a marked element $x^*$[10] is present in a given set of elements $x_0, x_1, ..., x_n$. We will also explain how the QA can be applied to SAT in order to show an approach to solving SAT problems using a QC.

### 3.3.1 Grover's Algorithm

We will now exemplify QAs by showing how quantum (brute-force) search of Grover works [Gro96, NC09]. To describe what kind of properties must be satisfied in this quantum search, we make use of a concept called an *oracle*. An oracle, in general, is a function $f$ that can be treated as a black box. In our case we define a function $f(x) \to 0, 1$ where $x$ is an element in the set of all the possible assignments. $f$ is defined by Equation 1.

$$f(x) = \begin{cases} 1 & x \text{ is marked, a satisfying assignment} \\ 0 & \text{otherwise} \end{cases} \tag{1}$$

Grover's Algorithm (GA) is a quantum search algorithm [NC09] with the objective to find one or more of the $M$ elements that satisfy a criterion given a large unstructured database of size $N$ where $0 \leq M \leq N$; by unstructured we mean that the elements are not necessarily ordered. $N$ here is the number of elements, but we will think of them as indices that can be represented using $n$ bits such that $N = 2^n$. Given this $n$ we can describe the complexity of GA as $\mathcal{O}(\sqrt{n/M})$. GA is an oracular algorithm, which means that it makes use of an oracle. Furthermore GA is a meta-algorithm which shows how to use any such oracle to do its work. The oracle in reality is an explicitly-specified subroutine corresponding to a particular problem. We define a unitary matrix $O_f$ such that $O_f |x\rangle = (-1)^{f(x)} |x\rangle$. In words it flips a bit phase if and only if the evaluation of the oracle is true.

Grover's Algorithm consists of three parts which we will explain on the basis of Figure 4. The qubit register consists of $n$ qubits in a vector $|0\rangle$ and the oracle's working memory in a vector $|1\rangle$.

In the first part we initialize all qubits.

1. Put all the qubits in an equal superposition $|0\rangle^{\otimes n}$ for initialization.

This equal superposition is obtained by applying a Hadamard transform on each qubit, such that the state after initialization is described as $|\psi\rangle = \frac{1}{N^{1/2}} \sum_{x=0}^{N-1} |x\rangle$ where $x$ represents each one of the

[10] An element that satisfies a given set of properties.

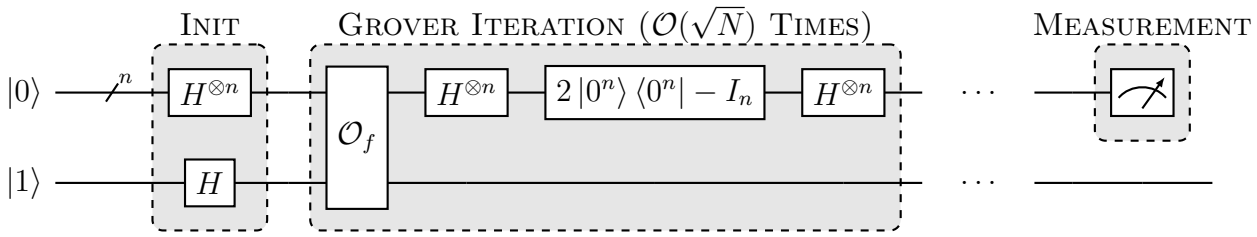
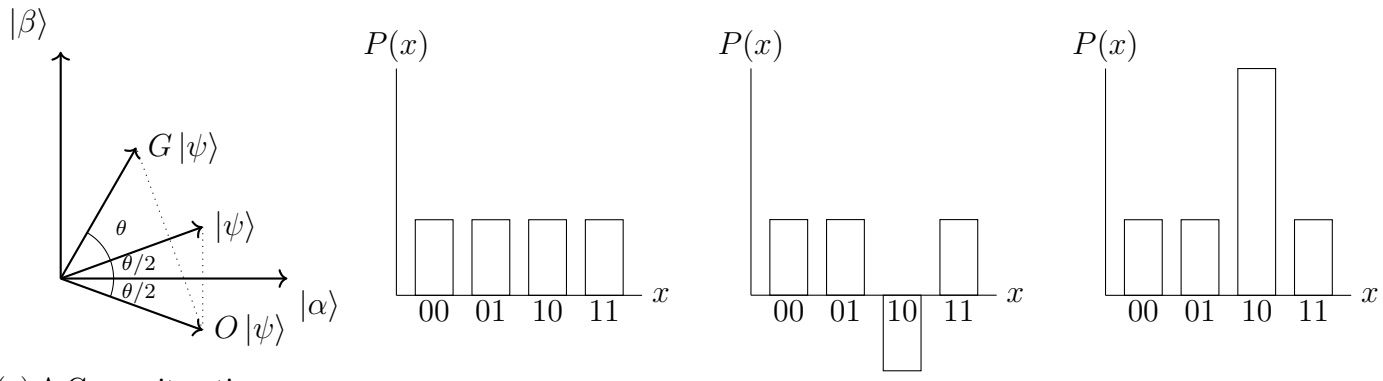



Figure 4: Grover's Algorithm as quantum circuit, consisting of three parts: Initialization, Grover Iteration and Measurement.




(a) A Grover iteration visualized using vectors.

(b) Initial state before Grover iteration

(c) State after applying the oracle

(d) One finished Grover iteration

Figure 5: Grover iteration visualised using vectors and probabilities. In the latter we see the maximization of the probability of finding the candidate solution of '10' upon measurement, where '00', '01' and '11' are not solutions that satisfy the criterion.

$N$ indices.

The second part is to apply the operation G, consisting of a number of quantum gates, $\mathcal{O}(\sqrt{N})$ times. G is a *Grover iteration* as shown in Figure 5a. Here we see a state $|\psi\rangle$ approaching the superposition $|\beta\rangle$, where $|\beta\rangle$ represents a superposition of states that satisfy the criterion as defined by the oracle function. We want to maximize the probability of measuring a solution that satisfies the criterion as shown in Figure 5. We also have the vector $|\alpha\rangle$ which is orthogonal to $|\beta\rangle$, together they span the whole two-dimensional space. A Grover iteration consists of two reflections.

2. Reflect over $|\alpha\rangle$ by applying the unitary matrix $O_f$; as mentioned all elements that satisfy $f(x) = 1$ will be phase flipped. We see in Figure 5b an example of the initial probabilities of the candidate solutions, after applying the oracle we meet the situation as shown in Figure 5c. In Figure 5a, this results in the vector $O\,|\psi\rangle$.

3. Apply *Grover's Diffusion operator*: First, apply a reflection over the initial state $|\psi\rangle$ before applying $O_f$. A Hadamard transform $H^{\otimes n}$ is applied on the qubits, to put all non reflected entries in the computational basis state of $|0\rangle$. Then a conditional phase flip is applied that applies a phase of -1 to all elements except those in the computational basis state of $|0\rangle$.

Finally, apply $H^{\otimes n}$ once again to restore its first application to undo putting the elements in the compuational basis state of $|0\rangle$ (to its original state). In Figure 5a this results in $G|\psi\rangle$, and in Figure 5d we see an examples of the resulting probabilities.

By repeatedly applying Grover iterations, the vector $|\beta\rangle$ will be approached by our transformed $G|\psi\rangle$.

4. Repeat steps 2 and 3 $\mathcal{O}(\sqrt{N})$ times.

The third part of the algorithm is measuring the input qubits. Upon measurement, the superposition state of the input quantum register will collapse in either of the $M$ results that satisfy the criterion.

5. Perform the measurement.

This yields the index of one of the $N$ elements in the database. When running the whole GA multiple times, we can eventually find all of the $M$ possible solutions. Grover proved that applying this $\mathcal{O}(\sqrt{\frac{N}{M}})$ times guarantees a measurement will reveal one of the $M$ solutions with a high probability of $> 0.45$.

### 3.3.2 Applying Grover's Algorithm to SAT

In SAT the criterion of the elements, assignments in this case, we are looking for is satisfying all constraints of the clauses. The database used for solving SAT with GA would be a set of all possible assignments; the $i^{th}$ qubit in the quantum register representing the $i^{th}$ variable of all variables in the formula given an order. Each bit can either be '0' or '1', representing the value assigned to the variable in order to satisfy the formula. In the case of SAT the oracle is a subroutine which evaluates the formula on a given input as defined in Equation 1. Solving SAT using GA is faster than using a classical brute-force algorithm. The complexity of GA here is $\mathcal{O}(\sqrt{n/M})$ in contrast to $\mathcal{O}(2^n - M)$ for brute-force. GA looks at all assignments, which we can map to the leaves of a full search tree as generated by a backtracking algorithm as seen in Figure 6. As mentioned, actual search trees can be heavily unbalanced when a backtracking algorithm like DPLL prunes a large portion of the subtrees. Because pruning is very common, it is likely that DPLL is faster than GA in the vast majority of the cases, despite using a QC and a QA. In Figure 6 we see this happening, as GA will evaluate every assignment at the bottom of the full search tree that would be pruned by DPLL, which is very inefficient.

### 3.3.3 Quantum Backtracking

A quantum walk is similar to a classical random walk through a search tree, but is implemented on a QC. Quantum Phase Estimation (QPE) is a QA that is used to find the eigenvalue given a unitary operator, however in the next description it is solely used to determine whether there exists a marked element inside a search tree when integrated inside a quantum walk. We define an operator which enables the possibility to detect the presence of a marked element within the search tree, as described in Algorithm 2 of [Mon18]. We define a meta-algorithm called *quantum backtracking*, which

- defines a quantum walk operator, which is used to detect a marked element by using Quantum Phase Estimation (QPE).

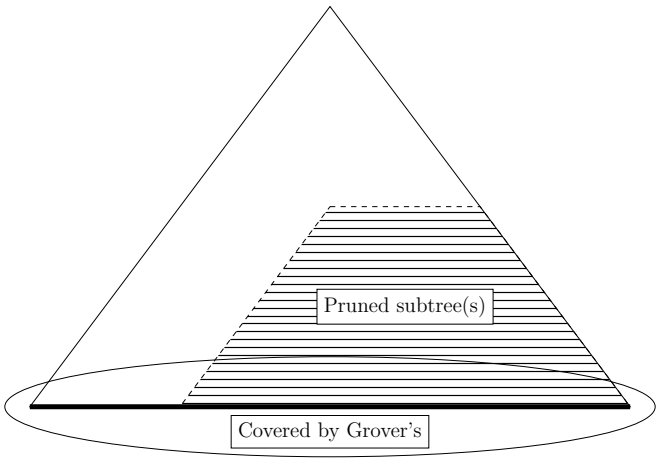



Figure 6: Unbalanced search tree resulting from the pruning of a Classical Backtracking in comparison to Grover's Algorithm evaluating on the same redundant solutions that could be ignored.

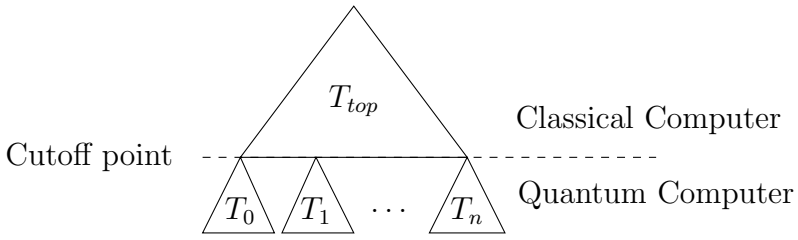



Figure 7: Divide and Quantum (D&Q) division using cutoff point visualised using a search tree. We see the top part of the search tree, annotated with $T_{top}$, which is handled using a classical computer and the cutoff subtrees $T_0, T_1, \ldots, T_n$ at the bottom handled using a QC

- finds marked elements by recursively exploring[11] the search tree using the mentioned detection of a marked element in order to narrow down its location.

as described in [Mon18].

For a given search tree of size $T$, the complexity of a CBT algorithm would be $\mathcal{O}(T)$. Using a quantum backtracking algorithm for solving SAT, one can find a satisfying assignment quadratically faster than a CBT like DPLL. Thus, quantum backtracking allows for a theoretical complexity of $\mathcal{O}(\sqrt{T}n\log(\frac{1}{\delta}))$ (where $\delta$ is a parameter which controls the probability of the algorithm to fail) which is much better than the complexity of GA if $T$ is much bigger than $n$. GA may be worse than DPLL as shown in Figure 6, but quantum backtracking will guarantee a theoretical quadratic speedup, which is why we choose for quantum backtracking over GA.

# 4 Divide and Quantum

Divide and Conquer (D&C) is a family of algorithms that make use of the idea of splitting up a problem instance recursively (divide) until we end up with trivial subproblems, which are solvable

[11] Based on the Classical Backtracking (CBT) routine.

in a polynomial amount of time. The next step is to solve the subproblems and merge them together until we end up solving the original problem instance (conquer).
Divide and Quantum (D&Q) [GD20, DGC18, RLD20] is a Hybrid Divide and Conquer algorithm, that is based on the previous description of D&C, but where we make use of both a classical and a quantum computer. In general for D&Q, if a subproblem reaches a reduced size which fits inside the QC, we can solve it in a polynomial amount of time too. This means that we do not reduce the problem recursively as the case in conventional D&C, but we solve the subproblem immediately using the QC in a polynomial amount of time. In a search tree for a backtracking algorithm, this subtree is not explored classically, but rather by some QA with the aim to reduce the complexity.

Generally the bottlenecks of D&Q are noise and polynomial overheads. We want to find polynomial speedups in order to still obtain a significant speedup that outweighs the polynomial overhead. Given a $m$-sized QC and a $n$-sized problem instance, we can calculate a size ratio $\kappa = m/n$. For the number of steps $T_C(n)$ to solve the problem instance of size $n$ by a classical algorithm and the number of steps $T_H(n)$ a hybrid algorithm takes and a coefficient $\epsilon_\kappa$, we say $T_C(n)$ and $T_H(n)$ are polynomially related by the equation $T_H(n) = \mathcal{O}(T_C(n)^{\epsilon_\kappa})$. If $\epsilon_\kappa$ exists for any $\kappa$, we have a *threshold free* speedup. The idea is that we have two functions $T_C(n)$ and $T_H(n)$ in $n$, and for each $\kappa$, we look for a constant coefficient $\epsilon_\kappa$ such that $T_H(n) \in O(T_C(n)^{\epsilon_k})$.

The essence of the D&Q algorithm of this research will be explained on the basis of Figure 7. A CBT like DPLL is run on the top part $T_{top}$ of the search tree, up to the *cutoff point* where the problem instance has been reduced enough, such that the remaining subtrees $T_0, T_1, \ldots, T_n$ can be handled on a QC, whose size is fixed to be a fraction of the size of the original problem instance. D&Q was developed to beneficially use a QC with a size smaller than that of the considered problem instance, motivated by the fact that near-term QCs are limited in size. The objective is to find the cutoff point, a ratio of the size ($\kappa$) of the instance, where a few conditions are met:

- The problem instances at this cutoff point fit inside a QC with a fixed size; i.e. the number of available qubits of the QC must be sufficient to be able to express the problem instance.
- The search trees of the problem instances at this cutoff point are exponential in size in relation to their height, which is a function of the number of variables. If the complexity of the CBT is exponential, then a theoretical quadratic speedup is guaranteed as shown in [Mon18]. With a polynomial complexity on a CBT, the overhead would cancel out the polynomial speedup gained, making our efforts unvaluable.

Current QCs have very few qubits, not enough to solve big problems with. Say, we can express each variable in a SAT formula using one qubit (which is not the case because of overhead), then the biggest formula we can handle also consists of very few variables. In perspective, real life SAT formulas are often bigger than a couple of hundred variables. However, QAs do have a quadratic speedup [Mon18] on CBT, for problems with exponentially sized search trees. This is the case for the mentioned **hard classes of random $k$-SAT formulas**, which are $\mathcal{NP}$-complete. Meaning that if the problem fits inside a limited sized QC and has an exponential sized search tree, then we can realise a quadratic speedup.

### 4.1 Approach

Given are a formula $\phi$ of size $|\phi|$ which results in an exponentially sized search tree, a fixed sized QC of size $m$ (where $m \leq |\phi|$) and a classical computer, we find a satisfying assignment for $\phi$ using Divide and Quantum (D&Q). First, we reduce $\phi$ using DPLL on the classical computer until we get a subformula $\phi_i$ which fits inside the QC. If $\phi_i$ is still in the exponential region of the search tree, as seen in Figure 3, then we solve the rest of the reduced problem instance using the QC. At this point a quadratic speedup is possible. If the size ratio of the reduced problem instance falls below the cutoff point, so if it is in the polynomial region, then a quadratic speedup can not be guaranteed for this subformula.

Given the search tree of $\phi$ of size $T$, during our run we meet subproblems $\phi_0, \phi_1, ..., \phi_n$ where each $\phi_i$ fits inside the quantum computer: $|\phi_i| \leq m$. Furthermore, there exists no parent of these problem instances for which the same holds, so we consider the largest sub-instances we can. The search trees of these $\phi_i$ are of size $T_i$, for which we take $T'$ steps in the hybrid case, with $T'$ being polynomially smaller than $T$. We define this overall theoretical speedup in the hybrid case as $T' = T + \sum_{i=0}^{n}(\sqrt{T_i} - T_i)$. The task is to determine a cutoff point where subformulas fit inside a limited sized QC **and**, the $\phi_i$ on average, have exponentially sized search trees. If the search trees are not exponential in size, then the polynomial overhead of using the QC in D&Q could cancel out the polynomial speedup gained from using the QA. For certain classes of SAT we suspect the subtrees from a to be determined cutoff point to be exponential in size. We want to identify which classes of  possess this property and up until which cutoff point this property persists as visualized in Figure 3.

## 5 Methods

In this section we explain three different methods we use to investigate whether a QCs will help when used in a D&Q approach to DPLL-based algorithms, if the size of the QC is small. Solving full problem instances with exponential complexity on a QC will always result in a speedup. However we want to investigate whether this holds for D&Q, when the QC size is limited. To find out whether it is the case that there exist classes of SAT, in particular random 3-SAT with $3.003 < \Delta < 4.3$, where D&Q with small QCs helps for wide ranges of ratios, we employ three methods, targeting various criteria for speedups.

**Generation of data on shape and size of search trees**
We have written a python program that generates random $k$-SAT formulas for an experiment, solves the experiment's problem instances using DPLL and stores data about the search trees that were generated during the run. The settings of the experiments are read from configuration files, in which the number of problem instances, a number of variables ($n$), $\Delta$ ($delta$), the number of variables per clause ($k$) and optionally cutoff points of the tree ($onDepth$) at which we will look at nodes are set. For a cutoff point value of $onDepth$, we multiply $onDepth \cdot n$ to obtain some value relative to the absolute number of variables that were available to be used from the start; hence we use the suffix $-OnDepthAbsN$ for some attributes. The generated data, such as sizes of subtrees at different cutoff points, will be accumulated in a CSV file. Whenever the program restarts, the CSV file will be consulted to determine which experiments are already present; this way, we do not

run duplicate experiments when restarting the program.

**Analysis of data**
To analyze the data we generated, another program is created for extracting relevant values from the CSV file. It can categorise certain entries by values of some attribute and plot data on the x- and y-axes of a figure. Next, to evaluate the relation between two attributes in the generated data we need to determine the functions of growth between the two. We perform a curve fitting of constant, linear, multi polynomial and exponential functions on the first 80% of the data (which is thus split up), then we evaluate the accuracy of each function family on the full 100% of the data using $R^2$. The function family with the highest $R^2$ is chosen, of which the function is recalculated on the full setting and plotted. In general, conclusively deciding that a scaling is polynomial or exponential, is a known hard problem.

## 5.1 Subtree size inspection

By direct inspection, we want to demonstrate whether there are classes of random 3-SAT for which, on average, the DPLL algorithm generates exponentially sized search trees with exponentially sized subtrees given their problem instances. Regarding Figure 7, if the roots of the subtrees are be in the exponential region, regarding Figure 3, then the subtrees at this cutoff are, on average, exponentially sized. For classes of random 3-SAT for which its problem instances have exponentially sized subtrees given a cutoff point for D&Q where the instance fits inside a small QC, we can conclude that, regarding the theoretical quadratic speedup, the QC can provide a valuable speedup. The theoretical quadratic speedup over an exponential time complexity will be sufficient to be not cancelled out by any polynomial overhead. To detect exponential scaling, we plot the sizes of generated search trees and its subtrees at fixed cutoff points as a function of the number of variables for different $\Delta$. We make use of curve fitting to get a best fit on our data points to detect exponential growth in the search trees and its subtrees. Two separate parts for the experiments will be defined as follows:

### 5.1.1 Determining $\Delta$ for which problem instances have search trees that grow exponentially in size

We generate data on the size of the search trees as a function of the number of variables for different $\Delta$. Then we perform Curve fitting of polynomial and exponential functions on the data points for each $\Delta$. In order to find the function that describes the data best (best fit), we run a statistical test ($R^2$). The family of the best fit function will either be polynomial or exponential, indicating whether the search trees are either polynomially or exponentially sized. We will assert that the problem instances with $\Delta$ in the hard regime have exponentially sized search trees and also that our data corresponds to that of Figure 3 of [CM01].

### 5.1.2 Determining cutoff points where subtrees still grow exponentially

For $\Delta$ where the growth is determined to be exponential (by performing curve fitting) on the full search tree, plot the average subtree size as a function of n on multiple depths. If these subtrees grow exponentially on specific depths with their height as a function of the number of variables, by curve fitting, then we can determine that the subtrees are, on average, exponential in size.

## 5.2 Subformula structure inspection

In this method we want to investigate which classes of random $k$-SAT formulas remain hard to solve in their subformulas, based on known theoretical bounds ($3.003 < \Delta < 4.3$), and how these amounts evolve for different fixed cutoff points. We know that subformulas with $\Delta$ in the hard regime imply trees of, on average, exponential size. Thus, having subformulas with this property at a cutoff point where it fits inside a small QC, tells us that a polynomial speedup over exponentially sized trees can be acquired for a small QC in D&Q. Regarding Figure 7, if the roots of the subtrees at the cutoff point have an average $\Delta$ in the hard region, then, by theory, their search trees are exponential in size. To determine whether subformulas of hard random $k$-SAT formulas stay hard random $k$-SAT formulas using theoretical bounds, we reduce a formula to subformulas and estimate the $\Delta$ on these subformulas. We count whether these $\Delta$ lay between the theoretical bounds for the hard regime general. If so, the subformulas of hard formulas remain hard to solve.

We generate a formula and reduce it, now for all cutoff points that we are interested in, we determine the $\Delta$ on its subformulas at this cutoff point. Next, we count the number of these subformulas of which the determined $\Delta$ is in between the theoretical bounds and determine the ratio of hard to solve subformulas over the total amount. Finally, we plot this ratio of hard subformulas as a function of the cutoff point and make statements about the results.

## 5.3 Total runtime comparison

We want to determine the overall speedup for a QC with a certain size, on certain classes of Boolean Satisfiability Problem (SAT) formulas, based on the theoretical quadratic speedup of a QC on backtracking algorithms. In order to do so, we compute the runtime of the hybrid algorithm, given a fixed cutoff point, and compare it to a classical algorithm. This gives us a speedup ratio $\sigma$, such that $T' = T^{\sigma}$, where $T$ is the size of the search tree for a classical algorithm and $T'$ the size of the search tree when using D&Q. By expressing the speedup ratio $\sigma$ as a function of the cutoff point at which we switch to a QC with a theoretical quadratic speedup, we can determine whether a QC in D&Q will provide a valuable speedup, despite being small.

We use two distinct sub-methods for our investigation, both require determining $T'$. By solving random 3-SAT formulas using DPLL we generate search trees for the problem instances. Given the search tree, we select nodes that have a fixed ratio of the original number of variables; each of these nodes is the root of a subtree $T_i$. See Figure 7, to calculate $T'$, we take $T_{top} = T - \sum_i T_i$, on which we run DPLL, plus the summation of the square root of each of the subtrees, regarding our theoretical quadratic speedup: $\sum_i \sqrt{T_i}$. We end up with the equation $T' = T_{top} + \sum_i \sqrt{T_i} = T - \sum_i T_i + \sum_i \sqrt{T_i} = T + \sum_i (\sqrt{T_i} - T_i)$, an approximate quadratic speedup, or polynomial to be more exact. For all given cutoff point ratios, we calculate the theoretical number of steps $T'$ that D&Q would take on the original tree of size $T$ using a theoretical quadratic speedup.

### 5.3.1 Estimate $\sigma$ based on the relation following a D&Q speedup

We plot all $T'$ on the fixed cutoff points and $T$ as a function of the number of variables for which we will perform curve fitting of an exponential function. Next we find $\sigma$ such that $T'(n) = T(n)^{\sigma}$,

using the parameters $a$ and $b$ of the exponential function ($f(x) = 2^{a \cdot x} \cdot b$) obtained by the best fit on the number of steps as a function of the number of variables (hence $T'(n)$ and $T(n)$):

$$T'(n) = T(n)^{\sigma} \implies 2^{a' \cdot n} \cdot b' = (2^{a \cdot n} \cdot b)^{\sigma} \implies \log_2(2^{a' \cdot n} \cdot b') = \sigma \cdot \log_2(2^{a \cdot n} \cdot b)$$

Use: $\log_2(2^{a \cdot n} \cdot b) = a \cdot n + \log_2(b)$, provided that $n \neq -\log_2(b)/a$:

$$a' \cdot n + \log_2(b') = \sigma(a \cdot n + \log_2(b)) \implies \sigma = (a' \cdot n + \log_2(b'))/(a \cdot n + \log_2(b))$$

Because we are interested in a constant, we eliminate $n$ using a limit[12]:

$$\sigma = \lim_{n \to \infty} ((a' \cdot n + \log_2(b'))/(a \cdot n + \log_2(b)))$$

This gives an estimation of $\sigma$ based on the exponential functions curve fit on the growth of $T'$ theoretically derived from $T$ for different $n$. We apply this process for each cutoff point and the $\Delta$ we are interested in, then represent the results in a table.

#### 5.3.2 Determining the median $\sigma$ from problem instances with different $n$

We can also derive the same $\sigma$ in a different way. Again, after determining $T'$ for different fixed cutoff points, we calculate $\sigma = \frac{\log_2 T'}{\log_2 T}$ and express it as a function of the cutoff point for different $\Delta$. Check that $0.5 \leq \sigma \leq 1.0$ for ($T' = T^{\sigma}$), where 0.5 is a full quadratic speedup and 1.0 is no speedup. We expect the full quadratic speedup to be at $onDepth = 0.0$ meaning that the quadratic speedup is obtained directly at the root node (we can not do better than this) and no speedup at $onDepth = 1.0$ which means having a quadratic speedup only in the leaf nodes, so no speedup (we can not do worse than this). We then represent the median $\sigma$ over problem instances with different numbers of variables for the different cutoff points and the $\Delta$ we are interested in.

# 6 Results

From our extracted results for each of the methods we can make conjectures that solidify our hypothesis that small QCs are able to help obtain valuable speedups in D&Q. We will use the abbreviations **const** (constant function), **lin** (linear function), **poly**$X$ (polynomial function with degree $X$) and **exp** (exponential function) in these results to describe the function families as a result of curve fitting. Given $onDepth$ and $n$, the cutoff point in our tree is at the nodes with $n \cdot (1 - onDepth)$ variables left. I.e. $onDepth = 0.0$ always corresponds to the full tree, likewise $onDepth = 1.0$ corresponds to only its leaves and anything in between corresponds to nontrivial subtrees.

## 6.1 Results from subtree size inspection

**Conjecture 1** *Subtree size inspection indicates that the D&Q method yields a polynomial speedup on subproblems of problem instances of random 3-SAT with $3.003 \leq \Delta \leq 4.3$ provided that the size of the QC is at least 90% of the size of the input. Exponential growths may be found at smaller subproblems as seen for 70% for $\Delta = 3.7$.*

[12] Otherwise we would end up with a fraction with $n$ in the denominator, and not a simple number.

As described above, we make use of two separate parts for this experiment, one where we determine $\Delta$ with trees with an overall exponential growth; then for those $\Delta$ we examine the growth of their subtrees at different cutoff ratios and connect the results the conjecture above.

### 6.1.1 Determining $\Delta$ for which problem instances have search trees that grow exponentially in size

In Figure 8 we see a graph with the best fit on data points (each point is an average over 100 problem instances) for the number of steps[13] (*steps*) as a function of the number of variables ($n$) for different $\Delta$ (*delta*); as $\Delta$ gets higher, the color gradient shift more from blue to red. In the legend we see which $\Delta$ correspond with exponentially sized search trees according to the curve fitting. For $1.0 \leq \Delta < 3.0$ we find linear shifting to polynomial search tree sizes. For $3.0 \leq \Delta \leq 4.3$ the search tree sizes are exponential. After the second threshold, $4.3 < \Delta$, the average number of steps to solve a problem with $n$ variables takes on a polynomial growth again. We conclude that for $3.003 < \Delta < 4.3$, in the hard regime, the relation between the number of steps and the number of variables is exponential on average. This means that the search trees of those random $k$-SAT problems grow exponential in size in relation to the number of variables overall. For these $\Delta$ it is possible to obtain a quadratic speedup using a QC when running on the whole problem instance.

In Figure 9 we see the number of steps as a function of $\Delta$, where each data point represents the average over 100 problem instances; here the color gradient shifts from blue to red with the number of variables from low to high respectively. We see a spike for which we can successfully assert that our results are similar to that in Figure 3 of [CM01] in regarding the thresholds.

### 6.1.2 Determining cutoff points where subtrees still grow exponentially

In Table 1 the best fit resulting from the curve fitting shown for different $\Delta$, in the rows, and cutoff points (as a ratio of the original number of variables), in the columns. The exponential families that we will cover are shown in light gray. For instance, given a $\Delta$ of 3.7 and a cutoff ratio, *onDepth*, of 0.1, the subtrees are exponential; at this cutoff we have reduced 10% of the problem, for say an original $n = 50$, the subtrees are of size $50 \cdot (1.0 - 0.1) = 45$. We see that for $3.003 < \Delta < 4.3$ the original trees are still exponential on average as concluded previously and, on average, stay exponential at a cutoff ratio of 0.1. For the cutoff point of 0.2 we see only that the same property holds for $\Delta = 3.4$. We see an additional exponential function for $\Delta = 3.7$ and $onDepth = 0.3$, however, a polynomial function is found at $onDepth = 0.2$. As we will mention in the discussion, we can suspect that exponential scalings (better results) exist on deeper levels, which may become clear when working with a higher number of variables. However, it is safer to assume that this is only an outlier.

We can conclude that we can reduce the problems classically up until the mentioned cutoff points where the subtrees are on average still exponential in size, then use a quantum backtracking algorithm with a theoretical quadratic speedup to obtain a valuable speedup. Given a problem instance of random $k$-SAT, with $3.003 < \Delta < 4.3$, which fits inside a small QC at a cutoff ratio of 0.1, we can conclude that a valuable speedup can be provided in the setting of D&Q.

[13] The number of steps it takes to find a satisfying assignment or conclude there are no satisfying assignments.

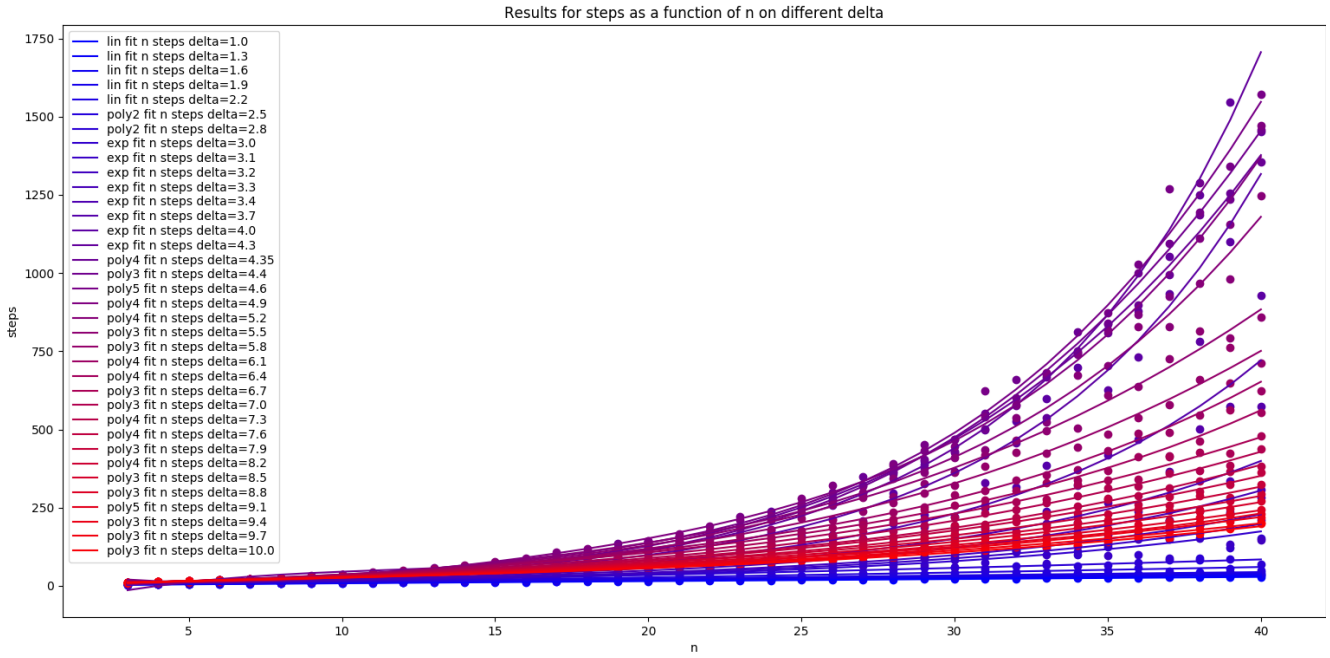



Figure 8: Search tree growth; Size of the search tree as a function of the number of variables of its problem instance for different $\Delta$.

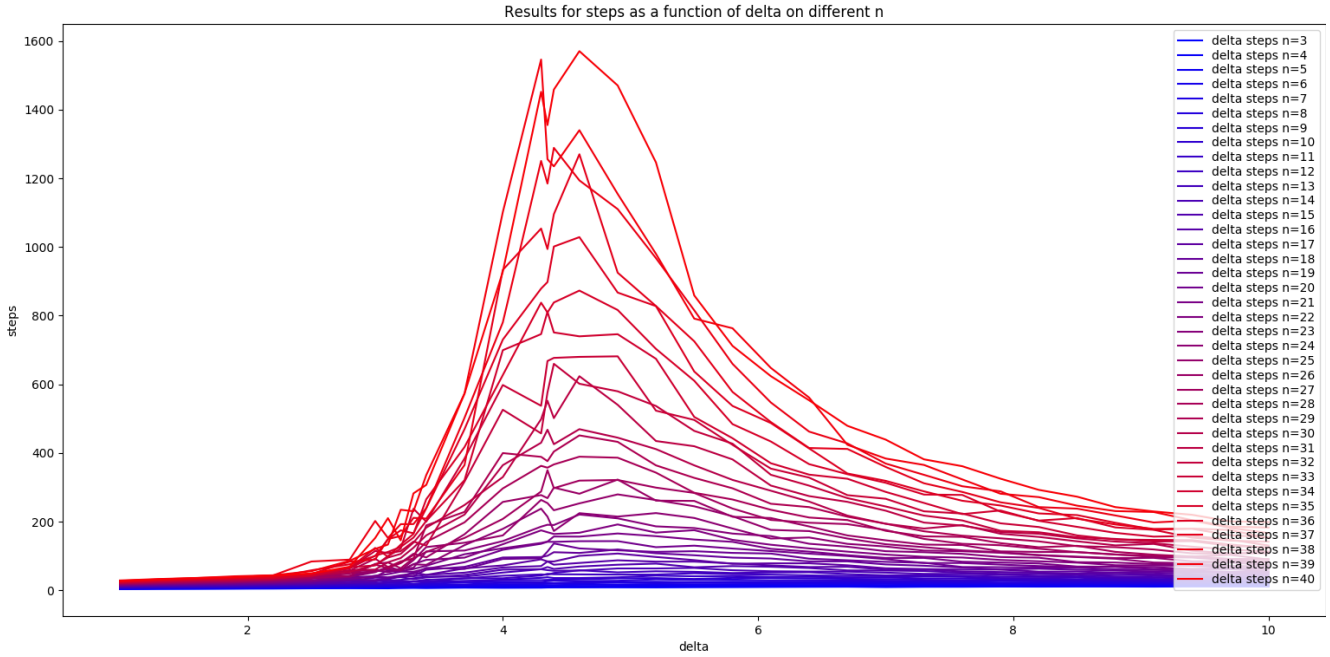



Figure 9: Search tree growth per $\Delta$; The number of steps taken in the search tree of the problem instances as a function of $\Delta$ for different numbers of variables.

| Δ | Function on cutoff point (onDepth) | | | | | | | | | | |
|---|---|---|---|---|---|---|---|---|---|---|---|
| | **0.0** | **0.1** | **0.2** | **0.3** | **0.4** | **0.5** | **0.6** | **0.7** | **0.8** | **0.9** | **1.0** |
| **1.0** | lin | poly3 | lin | poly3 | poly3 | poly4 | poly3 | lin | poly2 | const | const |
| **3.1** | exp | exp | poly2 | poly2 | poly2 | poly3 | lin | lin | lin | poly3 | const |
| **3.4** | exp | exp | exp | poly2 | poly3 | poly5 | poly2 | poly2 | poly2 | poly3 | const |
| **3.7** | exp | exp | poly2 | exp | poly6 | poly3 | poly3 | poly4 | poly2 | poly2 | const |
| **4.0** | exp | exp | poly5 | lin | poly2 | poly4 | poly2 | poly2 | poly2 | poly4 | const |
| **4.3** | exp | exp | lin | poly2 | poly2 | poly3 | poly3 | poly3 | poly3 | lin | const |
| **7.0** | poly3 | lin | poly3 | poly2 | lin | poly5 | poly3 | const | exp | const | const |

Table 1: Subtree growth for different Δ. The subtree size as a function of the number of variables for different cutoff points.

## 6.2 Results from subformula structure inspection

**Conjecture 2** *Subformula structure inspection indicates that the D&Q method yields a polynomial speedup on subproblems of problem instances of random 3-SAT with $3.003 \leq \Delta \leq 4.3$. We get a significant speedup*[14]*, provided that the size ratio of the QC is at least 90% of the size of the input. The ratio gradually decreases to 40% as Δ approaches 4.3.*

In Table 2 we see the ratio of 100 problem instances that have hard subformulas on average for multiple cutoffs and Δ. If the subproblem has an estimated Δ in between 3.003 and 4.3, then it is considered hard. Note that the estimated Δ here is a rough estimate as the subformulas are often not pure random 3-SAT formulas, but rather mixed with random 2-SAT and random 1-SAT clauses.

Given a cutoff point of, say, 0.2 (where we still have 40 variables) and a Δ of 3.7, then 97% of the subformulas are still hard. Therefore, for this case, D&Q seems to provide a polynomial speedup over DPLL provided a QC that is big enough to handle 40 variables. This statement can be generalised to different cutoff ratios and values of Δ, shown in Table 2. For random $k$-SAT problem instances where $3.003 \leq \Delta \leq 4.3$, we can conclude that subformulas with 90% of the original number of variables stay hard, and for $\Delta = 4.3$, this value even drops to 40%. For these subformulas, which remain hard on different cutoff ratios, we can obtain a theoretical quadratic speedup using a small QC in D&Q, as hard problem instances imply exponentially sized search trees.

## 6.3 Results from total runtime comparison

**Conjecture 3** *The total runtime comparison method suggests that the D&Q approach yields a polynomial speedup when given as input instances of random 3-SAT with Δ such that* $3.003 \leq \Delta \leq 4.3$*, provided that the size of the QC is at least 60% of the size of the input*[15]*.*

[14] We take 0.1 as our threshold, in order to find a "significant" fraction of hard subproblems, which is clearly distinguishable from 0.

[15] This is consistent with Conjectures 1 and 2, which put the threshold at 90% (as those methods are more precise than Conjecture 3).

| Δ | onDepth | | | | | | | | | | |
|---|---|---|---|---|---|---|---|---|---|---|---|
| | **0.0** | **0.1** | **0.2** | **0.3** | **0.4** | **0.5** | **0.6** | **0.7** | **0.8** | **0.9** | **1.0** |
| **3.1** | 1.00 | 0.20 | 0.03 | 0.00 | 0.00 | 0.00 | 0.00 | 0.00 | 0.00 | 0.00 | 0.00 |
| **3.4** | 1.00 | 0.99 | 0.58 | 0.12 | 0.02 | 0.00 | 0.00 | 0.00 | 0.00 | 0.00 | 0.00 |
| **3.7** | 1.00 | 1.00 | 0.97 | 0.55 | 0.12 | 0.02 | 0.00 | 0.00 | 0.00 | 0.00 | 0.00 |
| **4.0** | 1.00 | 1.00 | 1.00 | 0.91 | 0.55 | 0.18 | 0.04 | 0.01 | 0.00 | 0.00 | 0.00 |
| **4.3** | 1.00 | 1.00 | 1.00 | 1.00 | 0.87 | 0.43 | 0.13 | 0.04 | 0.01 | 0.01 | 0.00 |

Table 2: Overview of estimated hard problem instances found in subtrees for different $\Delta$ and cutoff points for $n = 50$.

We will investigate the theoretical speedups allowed by a D&Q algorithm by exploring what valuable speedups we can obtain for different cutoff points and $\Delta$. As we will see, for both partial experiments we can conclude that for an $onDepth = 0.2$ a valuable speedup near quadratic is guaranteed for all $\Delta$ in the hard regime. This corresponds to stating that, when applying D&Q, reducing the problem instances to 80% of their original size using a classical computer a valuable speedup can be gained. Thus a small QC with a size of 80% of the original input provides a quadratic speedup in D&Q.

### 6.3.1 Estimate $\sigma$ based on the relation following a D&Q speedup

We see a similar Table 3a to that of the previous part, however here are $\sigma$ listed as calculation from the estimated exponential function as a result of curve fitting. These sigma are not necessarily bounded by $0.5 \leq \sigma \leq 1.0$ because we take an estimation over a performed curve fitting, which is, on itself, already an estimated function. Extrapolating to an $n$ limiting $\infty$ results in these kind of estimation errors. Still, for a $\Delta = 3.7$ and $onDepth = 0.2$ we get a speedup ratio of $\sigma = 0.72$ by theory. This means that D&Q provides a valuable polynomial a speedup $T'(n) = T(n)^{0.71}$ over DPLL.

### 6.3.2 Determining the median $\sigma$ from problem instances with different $n$

In Table 3b we see the medians of $\sigma$, over the averages over 100 problem instances of experiments with different numbers of variables, for multiple $\Delta$ and cutoff ratios. We see that the speedup ratio lies between $0.5 \leq \sigma \leq 1.0$, which follows the theory that a maximum speedup in $T'(n) = T(n)^{\sigma}$ is quadratic (0.5), and the minimal speedup is $T(n)$ itself, with a ratio of 1.0. For instance, for $\Delta = 3.7$ and $onDepth = 0.4$ (we assigned 40% of the variables), a speedup ratio $\sigma = 0.86$, such that $T'(n) = T(n)^{0.71}$, can be obtained according to the theoretical quadratic speedup from [Mon18]. For all $\Delta$ we get a speedup ratio $0.50 \leq \sigma \leq 0.9$ as long as our problem is at least 60% of its original size. This means that using D&Q provides a valuable polynomial a speedup over DPLL.

# 7 Conclusion and Discussion

We have investigated whether that Divide and Quantum using a limited sized Quantum Computer can obtain valuable speed ups over Classical Backtracking algorithms like DPLL for random $k$-SAT

| | | Δ | | | | |
|---|---|---|---|---|---|---|
| | | 3.1 | 3.4 | 3.7 | 4 | 4.3 |
| Depth | 0.0 | 0.43 | 0.44 | 0.45 | 0.46 | 0.48 |
| | 0.1 | 0.48 | 0.49 | 0.56 | 0.63 | 0.68 |
| | 0.2 | 0.56 | 0.62 | 0.72 | 0.78 | 0.83 |
| | 0.3 | 0.68 | 0.77 | 0.85 | 0.90 | 0.93 |
| | 0.4 | 0.81 | 0.89 | 0.94 | 0.98 | 1.00 |
| | 0.5 | 0.90 | 0.96 | 0.99 | 1.01 | 1.02 |
| | 0.6 | 0.97 | 1.00 | 1.02 | 1.02 | 1.03 |
| | 0.7 | 1.00 | 1.01 | 1.02 | 1.02 | 1.02 |
| | 0.8 | 1.01 | 1.01 | 1.01 | 1.01 | 1.01 |
| | 0.9 | 1.00 | 1.00 | 1.00 | 1.00 | 1.00 |
| | 1.0 | 1.00 | 1.00 | 1.00 | 1.00 | 1.00 |

(a) By calculation estimated $\sigma$ from curve fitting exponential functions, represented in a table for all $\Delta$ and cutoff points we are interested in.

| | | Δ | | | | |
|---|---|---|---|---|---|---|
| | | 3.1 | 3.4 | 3.7 | 4 | 4.3 |
| Depth | 0 | 0.50 | 0.50 | 0.50 | 0.50 | 0.50 |
| | 0.1 | 0.62 | 0.62 | 0.61 | 0.62 | 0.64 |
| | 0.2 | 0.71 | 0.71 | 0.71 | 0.73 | 0.75 |
| | 0.3 | 0.78 | 0.79 | 0.80 | 0.82 | 0.84 |
| | 0.4 | 0.84 | 0.85 | 0.86 | 0.88 | 0.90 |
| | 0.5 | 0.89 | 0.90 | 0.92 | 0.93 | 0.94 |
| | 0.6 | 0.93 | 0.94 | 0.95 | 0.96 | 0.97 |
| | 0.7 | 0.96 | 0.96 | 0.97 | 0.98 | 0.99 |
| | 0.8 | 0.98 | 0.98 | 0.99 | 0.99 | 0.99 |
| | 0.9 | 0.99 | 1.00 | 1.00 | 1.00 | 1.00 |
| | 1 | 1.00 | 1.00 | 1.00 | 1.00 | 1.00 |

(b) Median $\sigma$ from experiments with different numbers of variables, represented in a table for different $\Delta$ and cutoff points we are interested in.

Table 3: Determined $\sigma$ represented in a table for different $\Delta$ and cutoff points

problem instances. This has been done in the setting classes of random $k$-SAT with certain $\Delta$[16] for which problem instances take an exponential amount of steps to be solved. The importance of this investigation is showing that small QCs can provide valuable speedups on DPLL for solving SAT problems despite the problem instances being larger than can fit inside the QC.

> **Conclusion:** *Our results seem to indicate that there are settings in which the D&Q method provides a computational advantage (polynomial speedup) over classical implementations of DPLL for the random 3-SAT problem. In particular, our most significant observations are that the D&Q method seems to yield a polynomial speedup in the following cases for $\Delta$ and size ratio $\kappa$:*
>
> - $3.003 \leq \Delta \leq 4.3$ *and* $\kappa = 90\%$ *for all Conjectures;* $\kappa = 60\%$ *for the least precise method (Conjecture 3).*
> - $\Delta = 3.7$ *and* $\kappa = 70\%$ *(Conjecture 1).*
> - $\Delta = 4.3$ *and* $\kappa = 40\%$ *(Conjecture 2).*

First, we showed that problem instances of the hard classes of random $k$-SAT are exponential in time complexity in relation to the number of variables of the instance. Furthermore, these problem instances preserve this property on their subformulas. Here we can define a cutoff point for which we know that if the subformula with this property fits inside a QC, then a theoretical quadratic speedup over DPLL can be obtained for this problem instance. Secondly, we have shown that subformulas of problem instances of random $k$-SAT with a $\Delta$ in the hard regime, will stay hard up until some nontrivial cutoff point. Finally, we have shown what kinds of polynomial speedups D&Q allows us to obtain with limited sized QCs.

[16] A parameter that is used to tune the hardness of problems

## 7.1 Discussion

**Limitations:** We found experimental evidence for polynomial speedups to be obtainable for problem instances of the hard classes of random $k$-SAT where $3.003 \leq \Delta \leq 4.3$. However, our experiments did not provide evidence for much smaller QCs of 50% of the size of the problem instance to offer polynomial speedups. We have used a relatively low number of variables and we did find better results for smaller ratios as we increased the number of variables in our experiments. This makes us believe that better results may emerge from using even higher numbers of variables. We used a relatively low number of variables because our algorithm has an exponential run time, thus running experiments may go up to hours each.
In the subformula structure inspection we estimated a $\Delta$ over subformulas of random 3-SAT problem instances. However, after reducing such formulas, we obtain formulas mixed from random 1-SAT, random 2-SAT and random 3-SAT formulas. The estimation of $\Delta$ has the assumption that this estimation holds for these subformulas as if they were pure random 3-SAT problem instances.

**Future:** In the future, we can look at more techniques of speeding up algorithms for SAT problems. Because we have multiple devices available for calculation in D&Q, we can run them in parallel. We can run the classical computer and the QC(s) at the same time and stop when one solution has been found, this allows for a higher speedup. Given some limited sized quantum computers, we can do much more accurate experiments. If the number of qubits on the QCs grows, then we can obtain even better speedups as shown in our methods (emphatically 1 and 3). Given our cutoff points it is possible to estimate whether solving problem instances of certain classes of random $k$-SAT on bigger quantum computers is worthwhile for certain bigger instances.

It is possible to reduce random $k$-SAT problem instances with $k > 3$ to random 3-SAT problems, we can make statements about our expectations on the speedup that D&Q would grant. We reduce a random 4-SAT problem instance A to a random 3-SAT instance B. Generally, the number of clauses in B would be bigger than that of A in order to describe all dependencies A has. The number of variables stays equal in both instances, so as the number of clauses grows during the reduction, $\Delta$ grows too. Problem instances of A are harder with lower $\Delta$, but it does not mean speedups would be much better. Say we reduce a problem of A to a problem of B, where the resulting B has a $\Delta > 4.3$, then it is not expected that D&Q would provide a valuable speedup on the original problem of A.

# References


[ABM04] D. Achlioptas, P. Beame, and M. Molloy. Exponential bounds for dpll below the satisfiability threshold. In *Proceedings of the Fifteenth Annual ACM-SIAM Symposium on Discrete Algorithms*, page 139–140. ACM/SIAM, 2004.

[AHI06] Michael Alekhnovich, Edward A. Hirsch, and Dmitry Itsykson. Exponential lower bounds for the running time of DPLL algorithms on satisfiable formulas. *Journal of Automated Reasoning*, 35(1-3):51–72, August 2006.

[AP03] Dimitris Achlioptas and Yuval Peres. The threshold for random k-SAT is 2k (ln 2 - o(k)). In *Proceedings of the thirty-fifth ACM symposium on Theory of computing - STOC '03*. ACM Press, 2003.

[BFS19] Thomas Bläsius, Tobias Friedrich, and Andrew M. Sutton. On the empirical time complexity of scale-free 3-SAT at the phase transition. In *Tools and Algorithms for the Construction and Analysis of Systems*, pages 117–134. Springer International Publishing, 2019.

[BSW01] Eli Ben-Sasson and Avi Wigderson. Short proofs are narrow—resolution made simple. *Journal of the ACM*, 48(2):149–169, March 2001.

[BV97] Ethan Bernstein and Umesh Vazirani. Quantum complexity theory. *SIAM Journal on Computing*, 26(5):1411–1473, October 1997.

[CM01] Simona Cocco and Rémi Monasson. Trajectories in phase diagrams, growth processes, and computational complexity: How search algorithms solve the 3-satisfiability problem. *Physical Review Letters*, 86(8):1654–1657, February 2001.

[DGC18] Vedran Dunjko, Yimin Ge, and J. Ignacio Cirac. Computational speedups using small quantum devices. *Physical Review Letters*, 121(25), December 2018.

[DLL62] Martin Davis, George Logemann, and Donald Loveland. A machine program for theorem-proving. *Communications of the ACM*, 5(7):394–397, July 1962.

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

[GD20] Yimin Ge and Vedran Dunjko. A hybrid algorithm framework for small quantum computers with application to finding hamiltonian cycles. *Journal of Mathematical Physics*, 61(1):012201, January 2020.

[Gro96] Lov K. Grover. A fast quantum mechanical algorithm for database search. In *Proceedings of the twenty-eighth annual ACM symposium on Theory of computing - STOC '96*. ACM Press, 1996.

[HR04] Michael Huth and Mark Ryan. *Logic in Computer Science*. Cambridge University Press, August 2004.

[Mon18] Ashley Montanaro. Quantum walk speedup of backtracking algorithms. *Theory of Computing*, 14(1):1–24, 2018.

[MPRT16] Raffaele Marino, Giorgio Parisi, and Federico Ricci-Tersenghi. The backtracking survey propagation algorithm for solving random k-SAT problems. *Nature Communications*, 7(1), October 2016.

[NC09] Michael A. Nielsen and Isaac L. Chuang. *Quantum Computation and Quantum Information*. Cambridge University Press, 2009.

[Nik05] S. I. Nikolenko. Hard satisfiable instances for DPLL-type algorithms. *Journal of Mathematical Sciences*, 126(3):1205–1209, March 2005.

[Ouy98] Ming Ouyang. How good are branching rules in DPLL? *Discrete Applied Mathematics*, 89(1-3):281–286, December 1998.

[PI00] Pavel Pudlák and Russell Impagliazzo. A lower bound for DLL algorithms for $k$-sat (preliminary version). In David B. Shmoys, editor, *Proceedings of the Eleventh Annual ACM-SIAM Symposium on Discrete Algorithms, January 9-11, 2000, San Francisco, CA, USA*, pages 128–136. ACM/SIAM, 2000.

[RLD20] Mathys Rennela, Alfons Laarman, and Vedran Dunjko. Hybrid divide-and-conquer approach for tree search algorithms, 2020.

[Zho] Yi Zhou. *Phase Transition in 3SAT*. http://home.ustc.edu.cn/~zhao03/slides/pt_sat.pdf.

# A Random $k$-SAT generation

To generate random clauses, we can use the chosen parameters to determine the number of total clauses $c_{tot} = c_{tot\ neg} \cdot c_{tot\ lit} = 2^k \cdot \binom{n}{k}$. Next we generate a random number between $0 \leq r_n < c_{tot}$. We use the random number $r_n$ to determine a number that denotes the combination of variables $c_{lit} = r_n / c_{tot\ neg}$ and a number that denotes the sequence of negations $c_{neg} = r_n \mod c_{tot\ neg}$[17].

In Algorithm 1 we see as the first step to generate a row for all k variables that we want to generate, the first row and left border being filled with ones, this row represents one outer layer of the Pascal's triangle (Seen in Figure 10). Then for each cell, we add the left cell and the upper cell together. For the algorithm, we start at the right bottom corner of the part of the used Pascal's triangle, each time the number in the cell is smaller than or equal to our current $c_{lit}$ we shift a cell to the left and increment our counter. Each time this is not the case we shift one cell to the top and save our counter.

Repeating this process gives us a list of numbers that correspond to indices of variables. Afterwards, we apply the negations to our variables. Writing $c_{neg}$ out in binary gives us a 1 bit on all indices of non-negated variables and a 0 on all indices of negated variables. An example of the result of Algorithm 1 is shown in Figure 10. The same for Algorithm 2 and Figure 11. The initial call to generate a formula will be done to Algorithm 3. The examples shown are for the configurations of $k = 3$ and $n = 6$. The random number $rand = 149$ is generated, resulting in the values $c_{lit} = 18$ $c_{neg} = 5$.

**Data:** $n$, $k$
**Result:** Pascal's triangle

$triangle = [[1] * (n - k + 1)]$
**for** $i \in [0, k-2]$ **do**
  $triangle+ = [[1] + [None] * (n - k)]$
**end**

**if** $k > 1$ **then**
  **for** $i \in [1, k-1]$ **do**
    **for** $j \in [1, n-k]$ **do**
      $triangle[i][j] = triangle[i-1][j] + triangle[i][j-1]$
    **end**
  **end**
**end**
**return** *triangle*

**Algorithm 1:** Generation of Pascal's triangle

[17] mod : modulo operator results in the remainder when dividing the left hand side expression by the right hand side expression of the operator.

**Data:** $r_n$, $n$, $k$, $triangle$
**Result:** Random k-SAT clause for given random number, n and k
$c_{neg} = r_n \mod 2^k$
$c_{lit} = r_n / 2^k$
$i = k - 1$
$j = n - k$
$prev = 0$
$A = []$
**while** $i >= 0$ **do**
  $current = prev + 1$
  **if** $c_{lit} > 0$ **then**
    **while** $triangle[i][j] <= c_{lit}$ **do**
      $current = current + 1$
      $c_{lit} = c_{lit} - triangle[i][j]$
      $j = j - 1$
    **end**
  **end**
  $prev = current$
  $A.append(current)$
  $i = i - 1$
**end**
**for** $i \in [0, k-1]$ **do**
  Add literal $A[i]$ with negation of $i^{th}$ bit in $c_{neg}$ to clause.
**end**
**return** *clause*

**Algorithm 2:** Generation of a clause from a random number

**Data:** $k$, $num_{variables}$, $\Delta$
**Result:** Random k-SAT formula for given $\Delta$ and number of variables
$num_{clauses} = \Delta * num_{variables}$
**for** $i \in [0, num_{clauses} - 1]$ **do**
  $r_n = random(2^k * \binom{num_{variables}}{k})$
  $clause = generateClause(r_n)$
  $formula.addClause(clause)$
**end**
**return** *formula*

**Algorithm 3:** Generation of a random k-SAT formula

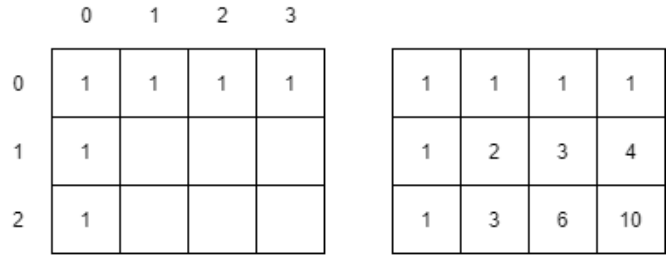



Figure 10: Generation of Pascal's triangle for $k = 3, n = 6$

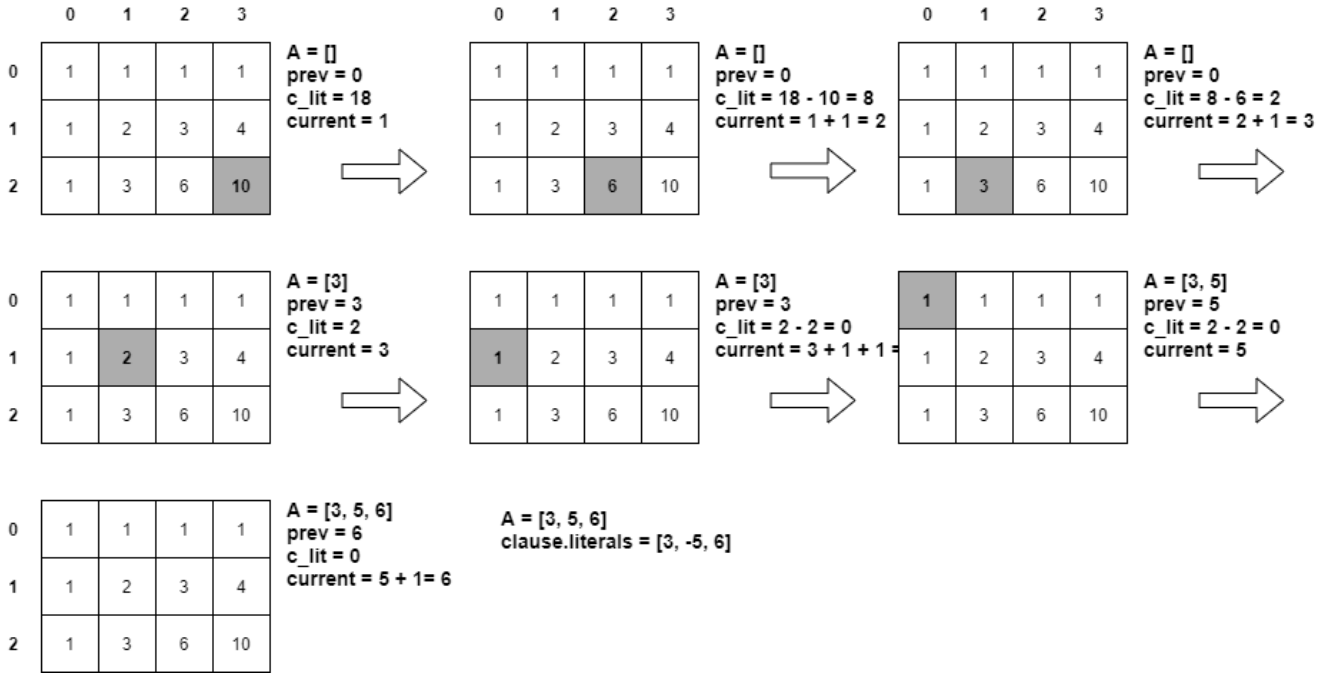



Figure 11: Generation of a clause for $k = 3, n = 6, rand = 149$, thus $c_{lit} = rand$ div $c_{tot\ neg} = 149$ div $8 = 18, c_{neg} = rand \mod c_{tot\ neg} = 149 \mod 8 = 5$

# B Subtree size inspection – exact method

**Part 1**

The exact description is to first determine which $\Delta$ have, on average, exponentially sized trees in relation to their height.

1. For $k = 3$, the number of variables $n \in [3, 4, ..., 39, 40]$ and $\Delta \in [1.0, 1.3, ..., 2.5, 2.8, 3.0, 3.1, 3.2, 3.3, 3.4, 3.7, 4.0, 4.3, 4.35, 4.4, 4.6, 4.9, ..., 9.7, 10.0]$ run an experiment of $p = 100$ problem instances (random 3-SAT formulas) for each combination.

2. Solve these formulas using DPLL and count the number of steps needed to: find a satisfying assignment **or** conclude there are none, for each problem instance.

3. Per experiment, average the number of steps over all $p$ instances.

4. Plot the average number of steps as a function of $n$ for the different $\Delta$.

5. Determine the best fit for constant, linear, polynomials of multiple higher degrees and exponential function families on the data.

6. Find $R^2$ on the best fit for all function families.

7. Select the best fit function with the highest $R^2$ and output it for each $\Delta$.

8. Determine $\Delta$ for which the curve fitting outputs that the trees are overall exponentially sized.

9. Assert that these $\Delta$ match the theoretical bounds given for the hard regime (where we expect exponentially sized trees). Assert that the number of steps as a function of $\Delta$ corresponds to that of [CM01] Figure 3.

**Part 2**
Now that we have found $\Delta$ for which the trees are exponentially sized, find the cutoff points in the number of variables for which the size is still exponential. For the determined $\Delta$, plot and examine the average subtree size for ratios of $n$, as the cutoff point, as a function of $n$ as follows.

1. Run experiments for $k = 3$, $n \in [10, 11, ..., 49, 50]$, $\Delta \in [1.0, 3.1, 3.4, 3.7, 4.0, 4.3, 7.0]$[18] , $onDepth \in [0.0, 0.1, ..., 0.9, 1.0]$ and $p = 100$.

2. Determine the size of the subtrees for the the classes of nodes that have a certain amount of variables. In our case, group all nodes by the number of variables from $n \cdot (1 - onDepth) = [100, 90, ..., 10, 0]$. At a cutoff point where we assigned to a ratio of $onDepth = 0.0$ of the variables, we still have all $n = 100$ variables. The same applies to $onDepth = 0.4, n = 60$ and $onDepth = 1.0, n = 0$.

3. Plot the size of the subtree to $n$ for all chosen cutoff points and curve fit for each cutoff point.

4. Examine and describe the function of the subtree growth at these ratios by looking at the best fit families.

# C Subformula structure inspection – exact method

We want to plot a theoretical upper bound of hard formulas for $\Delta$ in the hard regime, as a function of the ratio of live variables in these formulas as follows:

1. Run experiments for $k = 3$, $n \in [10, 20, 30, 40, 50]$, $\Delta \in [3.1, 3.4, 3.7, 4.0, 4.3]$, $onDepth \in [0.0, 0.1, ..., 0.9, 1.0]$ and $p = 100$.

2. For all node groups from $n \cdot (1 - onDepth)$, determine $n_{new} = n - depth$ as an upper bound of $n$, where $depth$ is the depth of the node in the tree[19]. From this $n_{new}$, estimate an upper bound of $\Delta_{UB} = \left\lceil \frac{n_{live}}{n_{new}} \right\rceil$ where $n_{live}$ is the number of live variables in the formula.

[18] We use $\Delta \in [1.0, 7.0]$ as representatives for their regimes to compare our results with.
[19] Assuming that we eliminate one variable in each step.

3. For each problem instance, average $\Delta$ on all of its subformulas at each cutoff point. Then count the number $\#(hardsubformulas)$ of subformulas with $3.003 < \Delta_{UB} < 4.3$ for all problem instances.

4. For each problem instance, calculate the ratio $\alpha = \frac{\#(hardsubformulas)}{\#(subformulas)}$ for all cutoff points.

5. Average $\alpha$ over all problem instances.

6. Plot $\alpha_{avg}$ as a function of the cutoff point for the given $\Delta$.

7. Examine how $\alpha_{avg}$ relates to the cutoff point and determine where $\alpha_{avg}$ is significantly big for specific depths.

# D Total runtime comparison – exact method

We want to find $\sigma$ as a ratio between the number of steps D&Q would take and the number of steps DPLL would take on a problem instance, using the theoretical quadratic speedup.

1. Run experiments for $k = 3$, $n \in [3, 4, ..., 49, 50]$, $\Delta \in [3.1, 3.4, 3.7, 4.0, 4.3]$, $onDepth \in [0.0, 0.1, ..., 0.9, 1.0]$ and $p = 100$.

2. For all node group the ones at the cutoff points using $n \cdot (1 - onDepth)$

3. For each of the groups, so for each cutoff point calculate $\sigma = \frac{\log_2 T'}{\log_2 T}$, take the median as our result for the median estimation of $\sigma$.

4. Also plot the subtree sizes and find the best fit function using curve fitting.

5. Use $\sigma = \lim_{n\to\infty}((a' \cdot n + \log_2(b'))/(a \cdot n + \log_2(b)))$ in order to calculate $\sigma$ as our second result for the estimation of $\sigma$ by calculation.

# Acronyms

**CBT** Classical Backtracking. 1, 2, 6, 13, 14, 22

**CNF** Conjunctive Normal Form. 2, 3

**D&C** Divide and Conquer. 13, 14

**D&Q** Divide and Quantum. 1–3, 13–19, 21–24, 31

**DPLL** Davis-Putnam-Logemann-Loveland. 1, 2, 6, 7, 12–17, 21–23, 29, 31

**GA** Grover's Algorithm. 3, 10–13

**PL** Pure Literal Elimination. 7, 32

**QA** Quantum Algorithm. 1, 8, 10, 12, 14, 15

**QC** Quantum Computer. 1, 2, 8, 10, 12–19, 21–24

**QPE** Quantum Phase Estimation. 12

**qubit** Quantum Bit. 9–12, 14, 24

**SAT** Boolean Satisfiability Problem. 1–6, 8, 10, 12–15, 17, 23, 24

**UP** Unit Propagation. 7, 32

# Glossary

**3-SAT** A class of SAT problems where each clause has 3 variables. 3, 5, 21

**$k$-SAT** A class of SAT problems where each clause has $k$ variables. 3, 33

**Boolean logic** A system for logic, named after George Boole. The system uses two values, $True$ and $False$, variables and a set of operators to perform manipulations on them. 2, 9

**CSV** A comma-seperated values file is a file format for storing data. The columns of which are seperated by commas. 15, 16

**curve fitting** Curve fitting is a method where we try to find a mathematical expression that describes a function of the data points best. 16–19, 22, 23, 31

**NaïveDPLL** A DPLL algorithm where we use UP and PL and otherwise randomly choose a variable on which we make a binary branch, where randomly choosing defines the Naïve part.. 7

**$R^2$** A statistical method for defining the goodness of a fit. The higher $R^2$, the more accurate a function describes the data points. 16

**random 3-SAT** A class of random $k$-SAT problems where the number of variables per clause equals 3. 1–3, 15–18, 21, 23, 24

**random $k$-SAT** A class of $k$-SAT problems where the clauses are generated uniform at random given a parameter $\Delta$. 1–4, 6, 7, 14, 15, 17, 19, 21–24, 27, 33