ALAN REYNOLDS

GENETIC ALGORITHMS FOR QUANTUM CIRCUIT DISCOVERY: BEYOND THE BASICS

Contents

1 Algorithm overviews

1.1 Introduction

As I took over from Václav as researcher on the quantum circuit discovery project in September 2018, I was bequeathed the C++ code for the optimization algorithm that existed at the time. This code implemented a multi-objective (MO) genetic algorithm (GA) that had been successfully applied to both the quantum Fourier transform problem and to Grover's algorithm. It provided a library of different quantum gates from which the user could select a subset from which quantum circuits would be built. It could exploit quantum simulation code from different sources (quantum++ and QIClib) and used a range of generic and domain specific genetic operators to assist the search.

Given the availability of this successful quantum circuit discovery code, my decision to start again, almost from scratch, may have seemed perverse. However there were a number of good reasons for doing so.

- The performance of a GA can typically be improved many-fold through the exploitation of problem domain specific methods. This usually involves more than merely the use of domain specific genetic operators. The application of circuit simplification methods and numerical optimization techniques means that the GA code is now only a small fraction of the overall code base.

- Solutions were evaluated from scratch. However, for the problem of quantum circuit discovery, it turns out that by applying a little more effort to the analysis of parent solutions, it is possible to greatly improve the efficiency of the evaluation of many possible child solutions — even those obtained through the use of crossover.

- While the code successfully separated circuit simulation from the rest of the GA[-1-], the genetic algorithm code was still more closely tied to quantum computing specific code than was ideal. The genetic algorithm itself was also a somewhat 'home-brewed' mixture of NSGA[-2-] and SPEA 2. Better separation of the GA code from the quantum computing code would enable the simple application of a range of standard and non-standard multi-objective GAs[-3-].

[-1-] This allowed for the use of simulators from more than one quantum computing toolkit, i.e. quantum++ and QIClib.

[-2-] At this time, NSGA is a rather antiquated multi-objective GA, superceded by NSGA II and others.

[-3-] The fact that I already had a code library for a range of multi-objective optimization algorithms, provided added motivation here.

- Given the difficulties associated with *many*-objective optimization, the number of objectives being used was a concern, since the gate count *for each* individual gate type was used as a separate objective[-4-]. Moreover, objective values were stored in an inflexible way, making it difficult to use alternative measures of solution quality.
- Given the breadth of the C++ language, it is unsurprising that I found the style of the code more difficult to work with than my own.

[-4-] For some multi-objective GAs, it is possible to have too *few* objectives. Indeed, the resulting lack of population diversity became an issue more than once during the algorithm development. However, the problems of too few objectives can usually be dealt with by simply choosing an algorithm that places more emphasis on population diversity. The problems of too many objectives is far less tractable.

Viewing these reasons more broadly, the primary motivation for starting from scratch was to produce significant improvements to performance, with a secondary motivation of simplifying for myself the task of adapting the algorithm.

Now we are about to take on a larger challenge. We plan to apply data mining and machine learning methods to detect patterns in small (but successful) circuits, in the hope that these patterns can be used to enhance the discovery of larger quantum circuits, or perhaps even to generalize from small quantum circuits to full quantum algorithms. This document therefore serves to close[-5-] the optimization phase of the project. It provides a fairly detailed description of the optimization algorithm and serves as a source of documentation for the code base.

[-5-] Ok, I am likely to continue looking at further improvements to the optimization algorithm in the future, but this will no longer be the primary focus of the research project.

This chapter continues with high level description of the major algorithm components, including some motivation for the approaches taken. Later chapters willl describe each component in more detail, including descriptions of assumptions and trade-offs made and further improvements that might be of benefit.

1.2 *Two algorithms*

Recent work has resulted in the code base being split into code for two separate algorithms.

The first algorithm emphasizes the goal of obtaining maximum benefit from the circuit evaluations performed, hence minimizing the number of expensive simulations required. No tricks are used to provide short-cuts to the process of evaluating a circuit: evaluation always consists of simulating each gate in turn to the input state and comparing the result with the target output states, so as circuit size increases, so does the computational cost of circuit evaluation. However, when a goal, such as reducing circuit cost through simplification, can be achieved without circuit evaluation, we attempt to do so. When circuits include parameterized gates, the gradient of the primary objective — overall error — can be calculated without too great an increase in computation time. This gradient information is therefore used to further reduce the number of circuit evaluations required, through the use of numerical optimization methods that can tune gate parameters in a few steps, rather than through the meandering search of the GA. Finally, solu-

tion cacheing is used to prevent unnecessary simulations of already evaluated circuits.

The second algorithm emphasizes the efficiency of circuit evaluation. By performing some additional work on parent solutions, it is possible to substantially reduce the costs of evaluation of child solutions, provided the children are generated using suitable genetic operators. Indeed the computational cost of child evaluation becomes independent of the size of the circuit, thus alleviating some[-6-] of the fear of large circuits that we have with the first approach. Thus, by ensuring that we generate many child solutions from relatively few[-7-] adults, we can speed up the optimization process. Circuit simplification is still performed, optimizing circuit cost without the need for any circuit simulations. However, we are more constrained in the type of genetic operators that may be applied. Moreover, numerical optimization of gate parameters can only be performed in a more limited way than in the first algorithm.

[-6-] Parent solutions are still simulated 'in full', so some of this fear remains.

[-7-] The adult population still needs to be large enough to contain sufficient solution diversity to encourage the exploration aspects of the algorithm.

Comparing the two approaches, the first approach:

- Emphasizes evaluation avoidance.
- Is flexible with regards to the types of genetic operators used.
- Permits full numerical optimization of multiple gate angles simultaneously.
- Uses no short cuts in the evaluation of circuits — each circuit is evaluated in full.
- Is expected to be of most benefit when circuits contain many parameterized gates.

In contrast, the second approach:

- Emphasizes evaluation efficiency.
- Is more restrictive in the availability of genetic operators.[-8-]
- Permits only numerical optimization of single gates.[-9-]
- Exploits significantly more efficient evaluation of child circuits.
- Is expected to be of most benefit when circuits have no, or relatively few, parameterized gates.

[-8-] More disruptive operators make it difficult to exploit the efficiency improvements.

[-9-] I am currently considering possible methods for optimizing pairs of parameterized gates, or small sets.

Neither of the two algorithms remains a simple application of a multi-objective GA to a simple quantum circuit discovery problem. While direct application of GAs to a problem are common, they are rarely competitive with algorithms that either explicitly use domain knowledge or use more mathematical optimization methods. Hence applications of GAs often involve hybridization with other techniques, resulting in a more complex algorithm, of which the GA is only a part. This is certainly the case with our two algorithms. To provide some structure to the sections and chapters that follow, where we describe both algorithms in more detail, we note that the algorithms each include a subset of the following major components

- genetic algorithm;
- circuit simplification;
- numerical optimization;
- gate 'reduction';
- cacheing.

Aside from the genetic algorithm itself, these components are either applied during the evaluation of a (partial) solution created through the application of the genetic operators, or used to elide such evaluation. Algorithm 1 includes all of these components, while algorithm 2 used all but the 'cacheing' component.

1.3 *Algorithm 1: overview*

1.3.1 *Circuits vs. circuit structures*

Each of the algorithm components listed above, with the exception of solution cacheing, aims to optimize some objective associated with the generated circuits. This raises two questions:

1. What structure is being optimized by each component of the algorithm?
2. What objective function or fitness measure is being minimized (or maximized)?

The answer to the first of these questions involves making the distinction between circuits and circuit structures.

A circuit is a complete specification of the quantum operations (or gates) to be applied to some input state, with the aim of producing some target state. All information required to determine the output state[-10-] is specified. In particular, if a gate can be thought of as having a parameter, such as a Pauli x-rotation with its angle, the value of that parameter is specified. A circuit *structure* provides the arrangement of the gates and the gate types used, but omits numerical parameters, such as the Pauli x-rotation angle.

The notion of circuits (or circuit structures) being *equivalent* plays an important role in the simplification and reduction components of the algorithm. Further details are deferred to chapter 3.

[-10-] By 'complete specification', we mean the provision of all of the details of the *theoretical* quantum circuit. That is, the circuits being optimized do not suffer from any noise or inaccuracies. The specification does not provide any practical details of how gates might be implemented. The circuits being optimized should be thought of as mathematical objects, rather than physical circuits.

1.3.2 *Genetic algorithm*

The GA provides the top level of the optimization algorithm. The quantum circuit discovery code has been written so as to provide 'Problem' and 'Solution' classes compatible with my pre-existing multi-objective metaheuristic (MOMH) library. This library contains multi-objective genetic algorithms, including a simple MOGA, and the two 'elitist' algorithms, NSGA II and SPEA II, and hybrid algorithms involving both genetic and local search components, referred

to as MOGLS 2 and JMOGLS. Writing the code in this way means that we can apply different algorithms by modifying a single line of code. Moreover, the MOMH library is written using C++ templates and a policy class based approach, meaning that different algorithm components — 'selectors', population 'evaluators', 'breeders' and 'survival' policies — can easily be plugged together in different ways to produce different versions of popular algorithms and hybrids. Hence a wide range of possible genetic algorithms can be applied.

At present, I have focused on using NSGA II. In certain scenarios, the original NSGA II has been found to be *too* elitist[-11-]. As a result, I have also performed some experimentation applying NSGA II but with the survival routine modified to try to encourage population diversity.

Solutions (i.e. circuits) are represented in much the same way as in the original algorithm. The genotype consists of a vector[-12-] of pointers to Gates[-13-], where the Gate class is the base of a hierarchy of more concrete Gate subclasses, such as XGate, Hadamard, YRotation and PiByEight. Calls to functions such as 'mutate' are handled polymorphically, meaning that the precise gate type involved is determined at run time and the correct version of the 'mutate' function is called automatically.

In algorithm 1, the genetic algorithm's primary objective is to optimize circuit structure, rather than all details of the circuit.[-14-] The process of evaluating a circuit structure then modifies the gate parameters, through the use of a numerical optimization algorithm better suited to optimizing these real-valued quantities. We also use fewer objectives than in the original code — typically the overall error of the circuit, the worst case error[-15-] and some measure of the cost or complexity of the circuit. [-16-]

The genetic algorithm also provides the only source (at present) of parallelization in the code, when the population of circuit structures is evaluated.

1.3.3 *Circuit simplification*

The original code provided only the most limited simplification of circuits prior to evaluation. If a gate that is its own inverse, such as an XGate[-17-] occurred twice in succession in the vector of gates, on the same qubit, then they would be cancelled, as shown in figure 1.1. Similarly, two successive PiByEight gates on the same qubit would be simplified to a PhaseGate (see figure 1.2. However, if the two XGates (or two PhaseGates) were separated, in the list of gates, by another gate on a different qubit (as shown in figure 1.3) then no simplification would be made, even though the new gate does not interfere with the XGates (or PhaseGates) in any way.

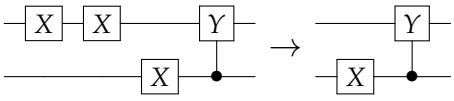


Figure 1.1: A simplification found successfully by the old code. Horizontal placement illustrates the order of the gates in the circuit's gate list.

[-11-] While early multi-objective GAs replaced the population of 'adult' solutions, in its entirety, with the 'child' solutions produced through application of the genetic operators, it was discovered that, for typical multi-objective problems, this resulted in a search with too little 'drive' — exploration of the front of non-dominated solutions was emphasized over improvement of the front. Elitist approaches therefore allow some of the old, adult solution to continue to survive if they are of high enough quality. Indeed, these algorithms typically take just the very best solutions from the combined population of adult and child solutions.

[-12-] For those unfamiliar with C++, a vector is a more sophisticated version of a simple array. (See appendix A.)

[-13-] I use the capitalized 'Gates' to refer to objects of the 'Gate' class in the C++ code, which naturally represent quantum gates.

[-14-] Until recently, the genetic operators would leave gate parameters entirely untouched. However, we have recently reintroduced a genetic operator that adjusts a gate parameter, in order kick the solution out of a local optimum prior to performing numerical optimization.

[-15-] We have performed some experimentation with just two objectives, dropping the worst case error; at present this tends to result in too little diversity in the population (though this is as much the fault of the elitism of NSGA II as the number of objectives). In section 1.3.4 we will also see that the worst case error is treated less seriously as an objective than the overall error, by other components of the overall algorithm.

[-16-] Modifying it to work with the same objectives as the old code, that is the measures of circuit error and the number of each type of gate, would be a trivial matter.

[-17-] Gate types provided by the code are described in section 2.1.1

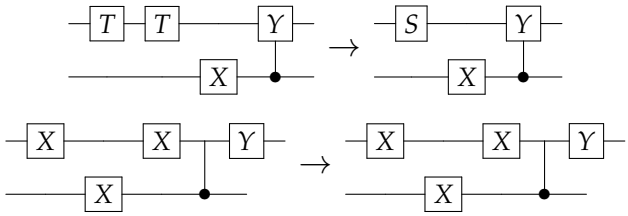


Figure 1.2: Another simplification found successfully by the old code.

Figure 1.3: The old code misses the cancellation. This circuit is the same as that in figure 1.1, except for a different ordering in the gate list.

The case for including more sophisticated circuit simplification is clear: simplification reduces the cost[-18-] of the circuit, without the need to perform expensive quantum simulations. Indeed, throughout this document and the code, the word 'simplification' is taken to mean a change to the circuit that can be shown, without the need for simulation, to reduce circuit cost, with no change to the circuit output[-19-]. In this sense, a simplification may, in some cases, lead to an increase in the total number of gates, provided the new gates cost less than the old.

[-18-] The phrase 'cost of the circuit' will be taken to mean the measure of circuit complexity used as an objective in the search, calculated as a sum of gate costs. (Other measures of circuit cost, such as circuit depth, could be easily included.)

[-19-] Beyond a possible change in overall phase.

While the straightforward application of a genetic algorithm in the old code might be expected to discover circuit simplifications given enough time, the chance, for example, that genetic operators would just happen to eliminate two cancelling XGates is low. Even if this fortuitous operation were to be applied by the algorithm, the new circuit would still be re-evaluated, requiring multiple runs of the expensive circuit simulator. It is much more likely that the genetic algorithm would first make a number of mis-steps, resulting in yet more applications of the quantum simulator. An automated method to perform the simplification without needing additional simulations is clearly preferable.

The current code contains a number of routines for determining whether a pair of gates can be simplified, taking into account gate types, target qubits and control qubits. Unlike in the original code, we consider simplifications where the gates are of different types. For example, if an XGate and a YGate are adjacent and operate on the same target qubit, they will be simplified to a ZGate when there are no control qubits, or a ZGate and a PhaseGate (or PhaseInv) in the presence of matching control qubits[-20-]. Moreover, additional routines determine whether gates may be swapped[-21-]. Hence simplifications of a pair of non-adjacent gates may be performed, provided gate swaps can be used to move the gates next to each other in the gate sequence.

[-20-] All assuming that the replacement gates are cheaper, in total, than the gates being replaced.

[-21-] What we mean by 'swap' here is explained in more detail in chapter 3.

The simplification routines should be thought of as optimizing the *cost* of circuit *structures*. The values of any gate parameters are effectively ignored[-22-]. This means that some circuit operations that might be considered 'simplifications' are not considered by this component of the algorithm. For example, a single qubit XRotation of angle π is equivalent to a (likely cheaper) XGate. This is not considered as a simplification of the circuit structure, since it depends on the value of the gate parameter, but as a gate 'reduction'. Such reductions are described briefly in section 1.3.5 and in more detail in section **??**.

[-22-] This is not entirely true. For example, when simplifying two XRotations, the code does sum the angles of the old gates to produce the angle for the new one. However, subsequent application of numerical optimization will overwrite this value. At present, the start point for the numerical optimization is given by the angles currently in the circuit, so the values of the gate parameters are not entirely irrelevant.

While this approach allows for much greater circuit simplification than previously, there is potential for more. Some simplification scenarios that are not currently handled are listed in chapter 3. Furthermore, the presence of multiple possible simplifications in a circuit raises the possibility that the order in which simplifications are applied may affect the cost of the final circuit. Indeed, there are times when it might be beneficial to temporarily *increase* the cost of the circuit, if this results in new simplifications that can subsequently produce greater cost reductions. The task of circuit simplification can be considered as an optimization problem in its own right, with the potential for the application of more sophisticated algorithms than are currently applied in the code. These ideas, along with a description of the simple heuristics currently used to select simplifications, are described in more detail in chapter 3.

Finally, the simplification routines make use of the notion of a *canonical form* for a circuit structure and provide the means for converting a circuit to this canonical form through the use of the gate swap routines. This provides benefits for other components of the algorithm. For example, the cacheing of objective values to prevent the need to reevaluate revisited circuits would either be less useful or have excessive memory requirements without the ability to use this canonical form. Use of canonical form is also likely to assist in the next stage of the project when we attempt to detect patterns in the circuits discovered. Again, canonical form and the algorithm for converting a circuit structure to this form are detailed in chapter 3.

1.3.4 Numerical optimization

In contrast to the simplification routines, used to minimize the *cost* of the circuit *structure*, we use numerical optimization methods to optimize the values of the *gate angles* so as to minimize the circuit *error*. Genetic algorithms make no use of gradient information to guide the search. In effect, the GA is blind and the mutation of a gate parameter is as likely to be in the wrong direction as the right one. This leads to more circuit evaluations and quantum simulations until the algorithm fortuitously makes the correct adjustments to the angles. Even in situations where gradient information is not available cheaply, alternatives to simply relying on the GA are usually more effective.

The situation for quantum circuit discovery is that gradient information is available, at some extra cost[-23-]. In this case, a more mathematical numerical optimization algorithm allows the gate parameters to be optimized, for a given circuit structure, with far fewer circuit simulations. We currently use the L-BFGS algorithm for this task.

[-23-] Details of the costs and methods for the gradient calculation are detailed in chapter **??**.

The use of the numerical optimization algorithm within the evaluation of circuit structures within a genetic algorithm raises a

number of issues regarding the control of the numerical algorithm. If, for a given circuit structure, the run of the numerical algorithm does not look promising then the algorithm should be aborted. This might happen if, for example, the algorithm is only discovering minor improvements to a low quality circuit when a better (lower error) circuit of lesser cost has already been discovered by the genetic algorithm. In contrast, one also should be aware that, even with only a few gate parameters, it is possible for the numerical algorithm to become stuck at a local minimum, in which case it may be beneficial to restart from different point in parameter space. Additionally, the number of gate parameters and the size of the circuit affect both the number of steps (and hence quantum simulations) required by the numerical algorithm and the expense of each step. Deciding how best to determine when to continue, stop or restart the numerical algorithm is a non-trivial matter.

1.3.5 Gate 'reduction'

The application of numerical optimization may reveal that certain parameterized gates are either unnecessary or may be replaced with a cheaper non-parameterized gate. This is revealed in the value of the angle parameter. For example, an angle very close to zero suggests that the gate can be removed. Such simplification of the circuit is not considered during the 'simplification' phase of solution evaluation, since at this stage it is the circuit *structure* that is being simplified, with angle parameters being ignored. To alleviate potential confusion, we call the type of simplification considered in this section 'gate reduction'.

Of course, such gate reduction may result in simplifications (of the type discussed in section 1.3.3) becoming available. Hence gate reduction is always followed by the reapplication of the circuit simplification routines.

Gates are reduced whenever the angle parameter lies within some tolerance of a special value, such as zero. Since the numerical optimization cannot be expected to produce angles that are precisely optimal, especially in the case where there are multiple parameterized gates that interact, this tolerance should not be too small. However, this tolerance implies that the gate reduction will result in a (small) change to the error values that might not be inconsequential. This raises two questions. Should the circuit be reevaluated after gate reduction? Should the numerical optimization be reapplied? At present, we reevaluate the circuit, since this is not too expensive compared to the numerical optimization, but do not reapply the numerical algorithm.[-24-]

1.3.6 Cacheing

A GA will revisit solutions. In extreme cases, when population diversity is not properly managed, the entire population in a single objective GA can end up being the same solution. In less extreme

[-24-] Since the gate parameters should already indicate a solution close to a (local) minimum, one would expect the reapplication of the numerical application to be relatively cheap compared with the first application. Whether to reapply may, in future, depend on the value of the difference in error values pre- and post-reduction. The answer to this question is likely related to the questions regarding when to stop or restart the numerical algorithm, discussed in section 1.3.4.

cases, the fact that the GA revisits is of little concern, if the evaluation of a solution is cheap compared with the overhead of the GA itself. However, this is certainly not the case in quantum circuit discovery, where it is important to avoid unnecessary circuit simulations. It therefore makes sense to cache information about circuit evaluations, in case the solution is revisted.

For simpler problems, each solution evaluated would have its own associated cached data, which would consist of just the objective values, thus eliminating the need to reevaluate if the solution is revisited. For the quantum circuit discovery problem, the presence of the simplification routine means that multiple different circuit structures may share the same cached data. In addition to the overall and worst case errors, the simplified circuit is also stored[-25-].

[-25-] For implementation reasons, the cached data also contains a flag indicating whether the solution has been evaluated.

In the case where parameterized gates are in use, we envisage cacheing additional information regarding the performance of the numerical search algorithm. To enable the numerical algorithm to be rerun in case the first application led to a local optimum, this could include the points in parameter space visited by the algorithm, so that the rerun can be started from a suitable, unexplored region. It might also be beneficial to resume a previously halted numerical search in the hunt for further improvements, in which case the state of the numerical algorithm at the halt point could be cached. However, at present, numerical search is applied once only to each simplified circuit structure, so the cache is merely used to look up the simplified structure and objective values of a previously visited solution.

The cached data is naturally accessed through the use of a hash table. Details of the data structures and hash function used are provided in chapter **??**.

1.3.7 Evaluation of a circuit structure

Top level pseudocode for the evaluation of a circuit structure is given in figure 1.4. The routine described is a member function of a circuit or solution class, so functions like `makeCanonical()` are applied to that circuit. Most of the interaction with the cache occurs in the `simplify()` routine, which simplifies the circuit but cross-references with the cache. If it finds that the circuit has been evaluated before, it returns a pointer to an object that contains the statistics for that previous evaluation. At present, this means the simplified circuit and the error values, plus a couple of extra items described later. In future this might include more detailed information about the application of the numerical optimization routines. If the `simplify()` routine completes the simplification of the circuit without discovering the circuit in the cache, it creates a new statistics object, containing just the simplified circuit but with spaces for information such as circuit errors, to be filled in later.[-26-]

[-26-] The eagle-eyed reader will notice that if the simplified circuit can, after the application of numerical optimization, be 'reduced', then *two* statistics objects are created: one for the simplified original circuit, and one for the simplified reduced circuit. Therefore, one of the extra pieces of information in the statistics object is a pointer that links the two, allowing either of the two circuits to be retrieved from the cache, as appropriate. The additional book keeping has been omitted from figure 1.4.

```
1: procedure Circuit::evaluateObjectives( )
2:     makeCanonical( )
3:     stats ← simplify(cache)                    ▷ Copies circuit and stats from cache if appropriate
4:     evaluateCost( )                                          ▷ The simple bit — just sum gate costs
5:     if stats→evaluated( ) then                                ▷ Simplified circuit was in the cache
6:         errors ← stats→errors( )
7:     else                                        ▷ Circuit hasn't been visited before - must evaluate
8:         if numParameters( ) > 0 then
9:             optimizeParameters( )
10:        else
11:            evaluateError( )
12:        end if
13:        stats→evaluated ← true
14:        stats→errors ← errors                                 ▷ Copies stats into the cache
15:        if reduce() then
16:            stats ← simplify(cache)
17:            evaluateCost( )
18:            if stats→evaluated( ) then                        ▷ Circuit was in the cache
19:                evaluateError( )                    ▷ We have already applied numerical...
20:                if overallError < stats→overallError then   ▷ ...optimization so we might as well...
21:                    stats→errors ← errors                   ▷ ...check whether current errors are...
22:                else                                        ▷ ...better than those in the cache
23:                    errors ← stats→errors
24:                end if
25:            else
26:                evaluateError( )
27:                stats→evaluated ← true
28:                stats→errors ← errors
29:            end if
30:        end if
31:    end if
32: end procedure
```

Figure 1.4: Top level pseudocode for circuit structure evaluation.

1.3.8 Synergy

From the outset, it was apparent that there was a degree of synergy between the different components of the overall algorithm. Circuit simplification reduces the number of gate parameters that must be optimized by the numerical algorithm, reducing overheads. Effective numerical optimization means that the genetic operators can focus on optimizing circuit *structure*, ignoring gate parameters. It also allows the cacheing of fitness values for circuit *structures*, ignoring the floating point gate parameters that would otherwise complicate this process[-27-] Reducing circuit structures to a canonical form reduces the number of circuits that need to be cached to something manageable, while increasing the effectiveness of the cache at reducing simulations of previously evaluated circuits. The numerical algorithm also reveals further simplifications or 'reductions'.

[-27-] While ignoring gate parameters simplifies the *process* of cacheing data, the fact that numerical optimization does not always find the global optimum in parameter space can complicate the data that is stored.

There is also expected to be a synergy between these methods and the next goal of using pattern detection to aid in the generalization of the circuits discovered to larger inputs. Simplification of the circuits should make pattern detection easier. The use of numerical optimization means that we may usefully discover patterns in just the circuit structure, ignoring the gates' angle parameters[-28-]. Finally, reducing each circuit structure to a canonical form can be thought of as 'unscrambling' the gate sequence, aiding in pattern recognition. For example, for the Fourier problem, ensuring that all of the swap gates occur at the end of the circuit makes the pattern — a YRotation and an ArbitraryPhase on each qubit, with a controlled ArbitraryPhase over each pair — easier to detect. A swap gate in the middle of the circuit scrambles this pattern.

[-28-] We would essentially 'complete the pattern' by performing the numerical optmization.

1.4 Algorithm 2: overview

While algorithm 1 focussed on avoiding unnecessary circuit evaluations and making the best use of those evaluations performed, algorithm 2 focuses on the efficient evaluation of child solutions. Achieved at the cost of additional work on the solutions in the adult population, this approach leads to improvements in algorithm speed when many child circuits are generated from relatively few adult solutions.

1.4.1 Circuit evaluation: adults and children

To evaluate a solution once, one must simulate the circuit once for each input–target state combination. In algorithm 2, when a solution enters the adult population, we effectively choose to evaluate it twice. The first evaluation is performed as usual, with simulations proceeding in the natural direction. The input state is fed to the first gate, which changes the state and hands it to the next gate, and so on until we find the output state of the circuit, which is then compared with the target state to produce a measure of the state

overlap[-29-]. These overlaps are then used to calculate the circuit error. For the second evaluation, the circuit is simulated in reverse. The target state is fed to the inverse of the last gate in the circuit, which feeds the result to the inverse of the second last gate, and so on until we reach the beginning of the circuit, where the resulting state is compared with the desired input state and the value of the overlap is obtained. Note that this overlap should match that obtained from the forward simulation. The key feature of this algorithm is that the intermediate results of each step of the simulations of the adult circuits are stored for use in accelerating the evaluation of child solutions.

When a child solution is created, for example by removing a gate from the parent circuit, the information stored in the parent(s) is used to more efficiently evaluate it. In this case, we already have the state obtained by simulating from the start up to the removed gate. We also already have the state resulting from taking the target state and simulating backwards up to the removed gate. We therefore need only compare the pair of states to calculate the overlap[-30-]. When a typical circuit may contain in excess of 50 gates, the ability to evaluate a child solution by just calculating the overlap of two pre-existing states provides a significant efficiency improvement.

While here we have considered just the removal of a gate from a circuit, child solutions generated by a number of different genetic operators may be more efficiently evaluated in a similar way. A list of the genetic operators we currently use with this approach is given in chapter 9.

1.4.2 *Evaluation short-cuts, circuit structure and gate parameter optimization*

While the use of these evaluation short-cuts can result in much more efficient evaluation of child *circuits*, the same cannot be said for *circuit structures*. Recall that, in the presence of parameterized gates, to find the quality of the circuit *structure* we must find the best values for the gate parameters and evaluate the resulting circuit. Removing a gate, for example, from the circuit structure, may result in significant changes to the values of the best gate parameters. As the parent solutions have only stored the intermediate states for the original set of gate parameters, the stored information cannot be used to expedite circuit structure evaluation.

This means that the evaluation short-cuts cannot be used effectively with the numerical optimization routines of the first algorithm. However, alternative algebraic and numerical optimization routines can be applied, albeit in a more restrictive manner. For example, upon inserting a parameterized gate into a circuit, we can use the information stored in the parent solution to efficiently calculate the overall circuit error symbolically[-31-], i.e. without fixing the angle in advance. The gate angle can then be optimized

[-29-] This 'overlap' is, of course, just the dot product of the two states.

[-30-] This will agree with the overlap that we would have obtained via full simulation — in either direction.

[-31-] The idea of performing symbolic calculations may be extended in future to allow two or more gate parameters to remain unfixed, and subsequently be optimized. However, note that the number of terms such a symbolic calculation would need to keep track of increases exponentially with the number of unfixed gate angles.

algebraically or numerically, as appropriate. Optimizing individual gates in this way as they are added to a circuit[32] provides less thorough optimization than the use of L-BFGS to optimize all gates simultaneously, as in algorithm 1, but it does allow us to optimize gate angles more effectively than the GA alone, while exploiting the information stored in the parent solutions to improve efficiency.

Note, then, that algorithm 2 makes less of a distinction between circuits and circuit structures. We cannot pretend that the algebraic/numerical optimization of a single gate parameter has resulted in the full evaluation of a circuit structure.

[32] Or perhaps modified by a mutation operator.

1.4.3 *How to use circuit simplification*

Improving the efficiency of circuit evaluation puts additional emphasis of the efficiency of the circuit simplification routines. More than this, one of the reasons for simplifying circuits — that it results in more efficient circuit evaluation — no longer holds true. In the first algorithm, where expensive numerical optimization, requiring multiple full-circuit evaluations, is applied to each circuit structure, it made sense to simplify prior to evaluation. In this second algorithm, simplifying does nothing to improve evaluation efficiency. Therefore the evaluation of a child solution occurs first. Simplification is deferred, and may be omitted entirely if the circuit is of low quality with regards to circuit error.[33]

[33] Given that the second algorithm does not (at present) include any numerical optimization (or indeed any parameterized gates), it does not include any notion of circuit reduction akin to that of the first algorithm.

1.4.4 *The genetic algorithm*

Concerns regarding the computational expense of the survival routines of NSGA II led us to try a number of home-brewed multiobjective GAs. It was felt that performing dominance checks for every pair of solutions in the large combined adult-child population would be too expensive, when working on problems with few qubits. However, code profiling revealed that, while the survival routine did take a significant proportion of the computation time, this was due primarily to the destruction of the circuits that do not survive to the next generation, rather than the comparison of solutions. At present, therefore, we use a variant of NSGA II in algorithm 2, in a similar manner to algorithm 1.[34] This variant allows for different child and adult population sizes, places more emphasis on crowding than is usual, and allows for the survival of a few solutions selected entirely at random[35]. The last two changes to NSGA II promote diversity in the adult population, since standard NSGA II loses population diversity on our problems, particularly in the case when we have few objectives.

[34] We do, however, apply algorithm two primarily to two objective problems, where the survival routine of NSGA II can be made more efficient that in the general case.

[35] This last variation has only recently been added and may be of limited benefit.

1.4.5 *Solution cacheing*

Solution cacheing is not a part of algorithm 2. Some preliminary experimentation with algorithm 1 suggested that it was only eliminating around 10% of solution evaluations. Moreover, when algorithm

1 was applied using a gate set with no parameterized gates, the increase in code speed due to the lack of numerical optimization meant that circuit structures were being cached at such a rate as to make excessive memory requirements[-36-].

[-36-] Furthermore, a significant amount of code was required to implement cacheing, which also interacted awkwardly with parallelization.

Note that, at present, algorithm 2 still uses canonical form, but it need not do so, since its primary role in algorithm 1 was to make solution cacheing feasible. The computational cost of using canonical form appears not insignificant, but not great enough for me to have eliminated its use, since canonical form may still be useful in future code developments.

Part I

Algorithm 1

2 Genetic algorithm

The genetic algorithm that we currently apply is just NSGA II, which is adequately described in the literature[-1-]. However, there is still much that may be detailed in this chapter, in particular with regards to the genetic representation of circuits. We start with a description of the permitted gate types, an introduction to the Gate class hierarchy and an analysis of the use of control qubits. We describe how the 'context' for a particularly circuit discovery problem, that is the set of gate types used and their costs, is managed by the code. We then continue with a brief description of circuit representation, including some discussion of the potential use of auxiliary qubits, before examining the genetic operators applied. We conclude with some discussion regarding objective functions.

[-1-] The use of the pre-existing multi-objective metaheuristic library allows simple application of other algorithms such as SPEA 2.

2.1 Gate types, controls and an introduction to the Gate hierarchy

The set of gate types used by the algorithm is the same as those used by Václav's original code, with the exception that the use of the SU2Gate is currently disabled[-2-] and the NotGate has been retired[-3-]. There have been times when I might have been tempted to use a reduced set — the implementation of simplification routines for every *pair* of gate types becomes increasingly taxing as the number of gate types increases. There have also been times when there was the temptation to add some gate types, e.g. the other gate types in the group generated by the PiByEight gate. We list the gate types currently implemented in this section, along with the class names that we use for the gates throughout this document. This is followed by a description of the implementation of control qubits. This section concludes with a simplified description of the Gate class hierarchy used in the code. The full description is postponed to chapter 3.

[-2-] SU2Gate use is disabled merely because the implementation of the simplification routines that involve SU2Gates is not currently of high priority, not because there is any fundamental reason that they can't be used.

[-3-] We have, of course, kept the functionally equivalent XGate. Having both as separate classes merely complicated the code unnecessarily.

2.1.1 Gate types

The gate types currently implemented are as follows.

Hadamard: Parameterless. Applied to a single qubit and may have

controls. Matrix representation:

$$\frac{1}{\sqrt{2}}\begin{pmatrix}1 & 1\\ 1 & -1\end{pmatrix}.$$

PiByEight: Parameterless. Applied to a single qubit and may have controls. Matrix representation:

$$\begin{pmatrix}1 & 0\\ 0 & e^{\pi i/4}\end{pmatrix}.$$

PiByEightInv: Parameterless. Applied to a single qubit and may have controls. Matrix representation:

$$\begin{pmatrix}1 & 0\\ 0 & e^{-\pi i/4}\end{pmatrix}.$$

PhaseGate: Parameterless. Applied to a single qubit and may have controls. Matrix representation:

$$\begin{pmatrix}1 & 0\\ 0 & i\end{pmatrix}.$$

PhaseInv: Parameterless. Applied to a single qubit and may have controls. Matrix representation:

$$\begin{pmatrix}1 & 0\\ 0 & -i\end{pmatrix}.$$

XGate: Also used in place of the old NotGate. Parameterless. Applied to a single qubit and may have controls. Matrix representation:

$$\begin{pmatrix}0 & 1\\ 1 & 0\end{pmatrix}.$$

YGate: Parameterless. Applied to a single qubit and may have controls. Matrix representation:

$$\begin{pmatrix}0 & -i\\ i & 0\end{pmatrix}.$$

ZGate: Parameterless. Applied to a single qubit and may have controls. Matrix representation:

$$\begin{pmatrix}1 & 0\\ 0 & -1\end{pmatrix}.$$

XRotation: Implements the Pauli x-rotation and takes a single angle parameter θ. Applied to a single qubit and may have controls. Matrix representation:

$$\begin{pmatrix}\cos(\theta/2) & i\sin(\theta/2)\\ i\sin(\theta/2) & \cos(\theta/2)\end{pmatrix}.$$

YRotation: Implements the Pauli y-rotation and takes a single angle parameter θ. Applied to a single qubit and may have controls. Matrix representation:

$$\begin{pmatrix} \cos(\theta/2) & \sin(\theta/2) \\ -\sin(\theta/2) & \cos(\theta/2) \end{pmatrix}.$$

ZRotation: Implements the Pauli z-rotation and takes a single angle parameter θ. Applied to a single qubit and may have controls. Matrix representation:

$$\begin{pmatrix} e^{i\theta/2} & 0 \\ 0 & e^{-i\theta/2} \end{pmatrix}.$$

ArbitraryPhase: Phase shifts the basis state $|1\rangle$ by an arbitrary angle θ. Applied to a single qubit and may have controls. Matrix representation:

$$\begin{pmatrix} 1 & 0 \\ 0 & e^{i\theta} \end{pmatrix}.$$

Note that when applied without control qubits, the ZRotation and the ArbitraryPhase are equivalent, but with opposite angles.

SU2Gate: Unavailable at present. Gate simplification code incomplete.

SwapGate: Swaps two qubits. Applied to a pair of qubits and, at present, cannot have control qubits.

Oracle: Applied to the set of all qubits. Multiplies a 'marked' basis state by a phase of -1, leaving all other basis states unchanged.

2.1.2 *Controls*

Gates of the types that permit control qubits hold them in an object of the Controls class, which merely controls access to a vector of boolean values that indicates whether each qubit is a control or not. It also provides a routine that converts the stored data into a form suitable for QIClib.

A key operation on a new Gate object is the selection of a random set of controls for the gate. This is performed by the various Gate classes, at the same time as the target qubit is chosen, rather than the Controls class, since the operation may differ depending both on the type of gate being considered and the gate set permitted by the user. In the description of the various gate types above, gate types were described as either permitting controls, or not. In practice, we need more control than this binary distinction. The user may wish to allow gates of a particular type but only without controls, or with only one control, or up to one control, or with any number of controls, etc. This information, provided by the user, defines part of the 'context' of the circuit optimization problem. The

routine for selecting a random set of control bits for a gate should clearly depend on this context.

The fact that the random selection of target and control bits also depends on the gate type is perhaps less obvious. Gates such as the PhaseGate have the property that the target qubits can be swapped with a control qubit without affecting the operation of the gate in any way. That is, a PhaseGate on qubit 0 with control qubit 1 is precisely the same as a PhaseGate on qubit 1 with control qubit 0. While this is not a problem with a simple application of a GA, for our algorithm this redundancy is undesirable.[-4-] We therefore insist that, for such gates, the target qubit has a lower index than any of the controls. That is, a PhaseGate involving qubits 0 and 1 is always represented as having target qubit 0 and control qubit 1.[-5-]

Naturally it is undesirable to repeat the code for selecting a random set of target and control qubits unnecessarily. We therefore introduce two additional gate classes: SingleTargetGate and PhaseTypeGate. The SingleTargetGate includes all the routines required by the typical quantum gate that has a single target and permits controls. It stores the target qubit, a Controls object and provides a function for randomly selecting the target and controls. This function is overridden in the PhaseTypeGate class. Each of the concrete Gate classes, with the exception of the SwapGate and the Oracle[-6-], then derives from one of these two classes.

[-4-] It either results in revisiting a circuit that has already been evaluated without the algorithm noticing, due to the different labelling of target and controls on a PhaseGate, or requires the cacheing algorithm to cache multiple copies of essentially the same circuit.

[-5-] This is related to the idea of a 'canonical form' for the circuit structure. SwapGates have a similar restriction: the qubits involved are always listed in numerical order.

[-6-] These two are not SingleTargetGates.

2.1.3 *The Gate hierarchy: simplified version.*

A simplified version of the Gate class hierarchy is shown in figure 2.5. Note that the ArbitraryPhase gate is both a RotationGate

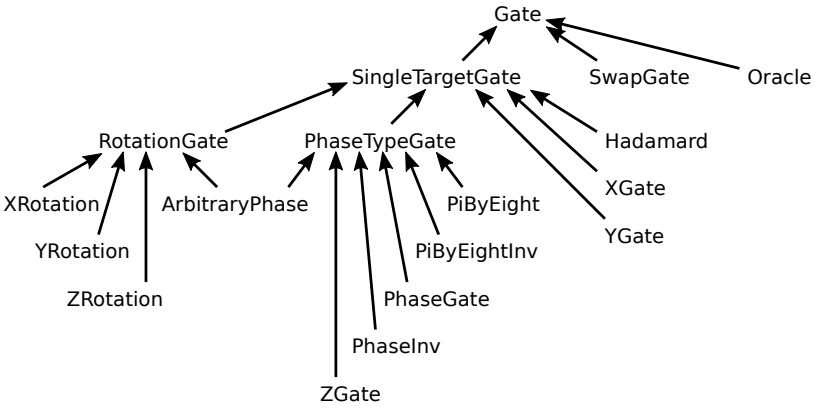


Figure 2.5: Simplified Gate class hierarchy.

and a PhaseTypeGate and that both of these classes inherit from SingleTargetGate. To prevent an ArbitraryPhase gate from having two potentially conflicting sets of data from the SingleTargetGate, both the RotationGate and the PhaseTypeGate classes inherit *virtually* from SingleTargetGate.

This class hierarchy is a little more complex than that in Václav's original code, with the intermediate classes being new. Having all gate classes inherit from the base Gate class is necessary to get the

polymorphic behaviour required. The circuit is represented as a simple vector of pointers to Gates and the code works out as necessary, at run time, which concrete gate type is actually present and which version of each member function to run. The intermediate classes reduce code duplication. In chapter 3, additional intermediate classes will be introduced that aid in the implementation of the simplification routines.

2.2 *The circuit context*

The circuit 'context' class contains the information describing the constraints under which circuits are evolved. This contains and provides access to the number of qubits, information on gate availability, gate costs, a function for creating a random gate, functions that assist in assigning each gate option a unique identifier and information on whether the input for each qubit is always zero, always one or varies[-7-]. Each circuit and each gate created contains a pointer to the circuit context.

[-7-] Useful for circuit simplification routines.

Gate availability and gate costs are both stored as vectors that are first indexed using a type identifier for each gate. Each concrete gate class contains a number from 0 to 14 as a static member and overrides a virtual function, declared in the base Gate class, to provide access to this type identifier. This is then used to access the cost and availability information stored in the circuit context object.[-8-] Gate cost for a particular gate type is stored as an object of the GateCost class, which provides a function that gives the cost of the gate given the number of controls. Gate cost is modelled as a polynomial in the number of controls, with integral[-9-] coefficients. This gives us total flexibility in assigning costs and allows for a more realistic model of gate costs than in the original code.

[-8-] This style of programming, similar to writing '`if (type == ...)`', is generally frowned upon, but seems difficult to avoid in this case.

[-9-] Use of integers here means we avoid the issues that occur when attempting to compare floating point values, without really sacrificing any flexibility.

The availability of a gate type is stored as a vector of three boolean values, indicating whether it is permitted to use a gate with no controls, with one control and with many (meaning two or more) controls respectively. This provides some extra flexibility when compared with the original code, which provided the user with the options 'none', 'one', 'any' and 'atLeastOne'. We now also have 'notMany' (i.e. zero or one), 'many' (i.e. at least two) and 'notOne' as well. While we do not expect a user to explicitly specify these new options very often, being able to handle the random generation of gates in the presence of such availability options has proven useful when it is determined that some gate options permitted by the user are actually redundant, as will be explained below.

The requirement to be able to assign each gate option a unique identifier is due to the use of a hash function by the cacheing component of the algorithm. It is necessary to be able to compute an integer hash value for each circuit structure that is evolved, in such a way that the likelihood that any pair of different circuits share the same hash value is low. By 'gate option', we mean a particular combination of gate type, target qubit and control qubits. The Gate

class declares a `numQbitOptions()` function, defined/overridden in the SingleTargetGate, PhaseTypeGate, SwapGate and Oracle classes. Once the circuit context contains the correct availability data, this function returns the number of different combinations of target and control bits for each particular gate type. The values returned allows the circuit context class to determine a gate option base identifier. For example, if the permitted gate types are PhaseGate, PhaseInv, XGate and YGate and it is determined that there are 5 PhaseGate options, 5 PhaseInv options and 8 XGate options, then the base identifier is 0 for the PhaseGate, 5 for the PhaseInv, 10 for the XGate and 18 for the YGate, i.e. it is the total number of options for the gate types that occur earlier in the list. The Gate class then declares a `calculate_option_id()` function that assigns a unique integer, starting from zero, to each possible combination of target and control bits and then adds this to the value for the base identifier stored in the circuit context. This function is defined by just the SingleTargetGate, PhaseTypeGate, SwapGate and Oracle classes. In this way, each combination of gate type, target bit and control bits is assigned a unique gate option identifier, being an integer in $[0, \text{numOptions})$.

The circuit context class also handles the detection of redundant gate types. For example, if the user were to specify that both uncontrolled ZGates and uncontrolled ArbitraryPhase gates were available, with the ArbitraryPhase gate being no more expensive that the ZGate, then the uncontrolled ZGate would be redundant. The initialization of the circuit structure object detects such redundancies, removing the availability of the uncontrolled ZGate. It is this automatic detection of redundancies that required the increase in flexibility with regards to gate availability specification. The user may only indicate gate availability using the 'none', 'one', 'any' and 'atLeastOne' options, but subsequent detection that, for example, the ZGate with a single control is redundant can lead to the occurrence of the more unusual options such as 'notOne'.

Finally, the circuit context class also contains a double precision real number indicating the mean length of a randomly generated circuit.[-10-]

[-10-] This seems out of place in the CircuitContext class, but is yet to be placed elsewhere.

2.3 *Circuit representation*

As has been indicated already, the circuit is represented by a vector of pointers to Gates, accompanied by a pointer to the CircuitContext object for the optimization problem at hand. The Circuit class also provides a number of functions that allow the various components of the algorithm to manipulate and to extract information about the circuit.

Of course, modern C++ provides a number of options for pointers, beyond the use of a bare pointer such as is found in C. We have chosen to use `std::unique_ptr` from the standard library. This provides automatic memory management with lower over-

head than `std::shared_ptr`. However, it is assumed that the class holding the pointer, i.e. the Circuit class in this case[-11-] *owns* the object that is pointed too. This seems clear in this case. It is also the case that only a single `unique_ptr` can point to an object. As such, `unique_ptrs` cannot be copied; they can only be moved. This has resulted in a few minor difficulties at times.[-12-]

[-11-] One might say that it is the vector that owns the gates referenced by the `unique_ptrs`, but the Circuit owns the vector.

[-12-] It has been a learning experience.

Functions provided by the Circuit class include tests to determine if two circuits are equal, have equivalent structure (i.e. equal up to gate parameter values) and whether one circuit should be sorted before another. Functions are also provided to apply the various genetic operators, described in section 2.4, to a circuit.

The Circuit class also provides a function for extracting the gates' angle parameters as a single vector and for setting these parameters by providing such a vector. These are used to provide straightforward interaction with the numerical optimization code. Functions are provided for converting the circuit into canonical form and for circuit simplification and reduction. A hashing function is provided for use by the cacheing components of the algorithm, as are a number of routines for performing quantum simulation of the circuit[-13-].

[-13-] Along with the basic simulation function, we have included a function that also records the intermediate states and another that not only records intermediate states, but performs a backward simulation, by applying the inverse gates to the target state in reverse order. These are supplied to allow for the efficient calculation of both the error functions and the gradient of the overall error. An additional simulation function that replaces one of the gates with its 'derivative' is also available for calculating components of this gradient, but is currently unused, as this method of calculating the gradient is inefficient.

Thus far, all experiments have been performed without allowing for the use of auxiliary qubits and, at present, the code does not explicitly prepare for their use. However, it should be possible to simply use the current circuit representation, but with a fixed number of additional qubits, to handle this situation. Changes to the objective function code provided as part of the problem (e.g. Fourier, Grover etc.) would be required, while any changes to the Circuit class would likely be restricted to the addition of a single member function that provides a count of the number of auxiliary qubits that are actually used by the circuit.

2.4 *Genetic operators*

The Circuit class provides one form of crossover and six mutation operators, as follows:

Two point crossover: Two cut points are randomly selected in each of the parent solutions. The sections between the cut points are swapped to create the children. Note that, in addition to cutting solutions in the middle, the cut points may also occur at the ends[-14-]. Cut points may also coincide.

[-14-] Thus the operator can perform single point crossover with the correct selection of cut points.

Replace Gate: Replace a gate with a randomly selected one.

Insert Gate: Insert a random gate at a randomly selected location. This location may be in the middle of the gate sequence, or at either end.

Remove Gate: Remove a single, randomly selected gate.

Swap Gates: Swap gates from two randomly selected locations.

Move Gate: Select a gate from the gate sequence at random and move it to a new location.

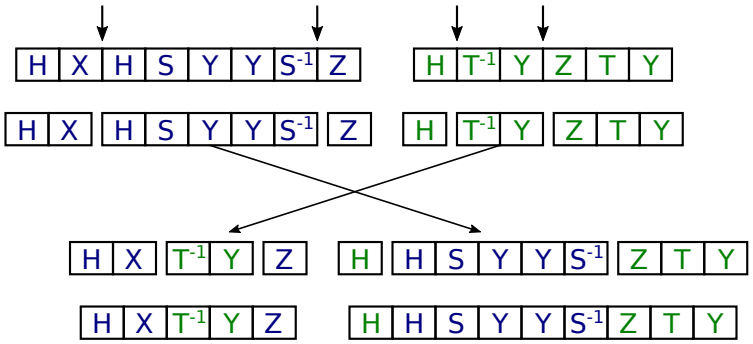


Figure 2.6: Two point crossover.

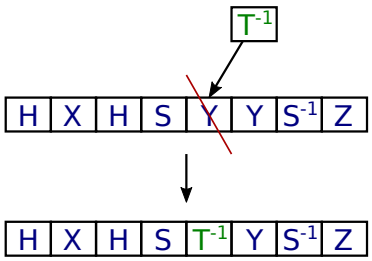


Figure 2.7: Replace gate.

Mutate gate: Select a parameterized[-15-] gate from the gate sequence at random and randomly adjust the gate parameter. At present, this simply means selecting a new angle uniformly at random[-16-]. This genetic operator is provided solely to allow 'adjust and reoptimize' functionality, for when the local search fails to find the global optimum.

[-15-] The Gate class provides the function 'mutatable()' that is overridden by the derived classes and indicates whether the gate can be mutated. At present, a gate is only considered 'mutatable' if it has a parameter such as an angle. ('Mutable' is a keyword in C++ — hence the use of 'mutatable'!))

[-16-] We used to add a normally distributed random variable to the angle.

2.5 *Objective functions*

In Václav's original algorithm, the objective functions were as follows:

Overall error: A measure of the overall or summarized error over all input configurations. Note that this is not necessarily the average of the errors for individual inputs. For example, in the Fourier problem, it is possible to get zero error for each input in the sense that the output matches the target output up to a phase, but the overall error may be non zero since the phases of each of the individual outputs must agree.

Worst case error: The worst deviation of output from target output. As noted above, we may choose to permit the output and the target to differ by an overall phase, without registering this as an error.

Number of gates: The number of gates *of each type* form another set of objectives.

Given a gate set to select from, this results in a large number of objectives. As the number of objectives increases, the chance that a randomly selected solution will dominate another decreases, since it must be better, or at least as good, in *all* objectives. For

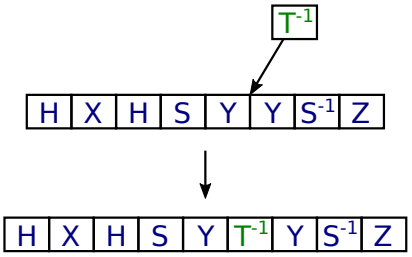


Figure 2.8: Insert gate.

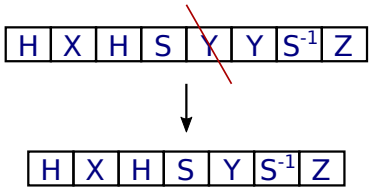


Figure 2.9: Remove gate.

large numbers of objectives, a pair of solutions will typically be such that neither dominates the other. This can lead to a number of difficulties. Perhaps the main difficulty is that if it is highly unlikely that any solution will dominate another, the genetic algorithm obtains very little information to lead it towards better solutions. It therefore loses its 'drive' towards the desired optimal solutions. Furthermore, non-dominated sets of solutions become large. This is seen, to a certain degree, in the non-dominated sets of solutions presented in the paper. Presenting the user with such large sets of solutions, most of which will be of little interest to him/her, is unhelpful.

While a reduced set of objectives would seem preferable, NSGA II is not suited to single objective problems. The highly elitist nature of the algorithms selection and survival components means that the solutions in the population rapidly become mere copies, or near-copies of each other. The resulting lack of population diversity means that the crossover operation merely produces more copies of the same solution and the algorithm fails to search effectively.[-17-]

Typically, NSGA II works well on problems with two or three objectives[-18-]. However, certain circumstances can result in the algorithm performing as if the problem has fewer objectives than it actually has. The most obvious scenario occurs when objectives are correlated. Clearly, when applied to a problem with two perfectly correlated objectives, NSGA II will behave as if applied to a typical one objective problem. A second scenario is when an objective takes only discrete values. Notice that both these scenarios occur in our problem. Circuit costs based on gate counts take only discrete values, while overall and worst case objectives tend to be correlated.

Both of our algorithms use a reduced set of objectives. This is mainly achieved by combining the set of gate count objectives into a single circuit cost. This cost is merely the sum of the costs of the gates that compose the circuit. For many problems, e.g. the Fourier problem, it also seems to make sense to use a single measure of circuit error, i.e. to drop the worst-case error objective. However,

[-17-] An important distinction should be made between the difficulties posed by too many and too few objectives. Too many objectives essentially means that the algorithm has not been provided with enough information to effectively direct the search. There is little that the algorithm designer can do to to mitigate this. Emphasis should be on better distinguishing between better and worse solutions, either through a better (smaller) list of objectives or through other means. The problems that NSGA II has with problems with too few objectives are problems that can be effectively dealt with by the algorithm designer — there are, after all, plenty of single objective GAs!

[-18-] Even on typical 2-objective problems, it has been shown that methods for increasing diversity early in the search, until enough solution diversity is discovered along the non-dominated front, can be beneficial. See some of our previous research.

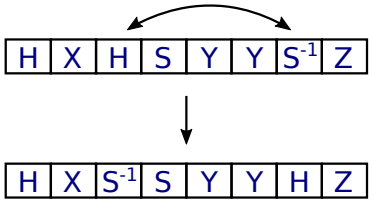


Figure 2.10: Swap gates.

experiments with algorithm 1, which, we recall, uses NSGA II as the base GA, clearly indicate a major lack of population diversity[-19-] when the worst-case error is dropped. We therefore keep the worst-case error as an objective[-20-]. In our experiments, we will also consider other means to further encourage population diversity. (Later we will see that, in our second algorithm, modifications to NSGA II can result in effective optimization for just two objectives.)

[-19-] The population becomes filled with multiple copies of the empty circuit — not useful!

[-20-] We may choose to consider the worst-case error as a true objective, or as merely a means to allow NSGA II to work more effectively.

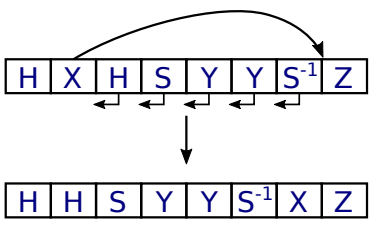


Figure 2.11: Move gate.

3 Circuit Simplification

In this chapter, we examine the circuit simplification component of the algorithm in a bottom up manner, starting with those functions that determine whether pairs of gates may be simplified or swapped and finishing with the top level simplification algorithm, which can be thought of as a simple optimization algorithm in its own right.

Given the unavoidably high cost of simulating a quantum circuit, particularly as the number of qubits increases, some effort should be spent in avoiding unnecessary quantum simulations. Much work can and should be applied to circuit simplification, wherein the gate cost or complexity of the circuit is reduced without affecting circuit accuracy and without the need for circuit simulation. Of course, this can be taken to extremes. We will see that initial attempts to combine circuit simplification and fitness cacheing could easily result in the cost of these processes outweighing those of circuit simulation, at least for few qubits.

The simplification routines implemented at present focus on simplifications of pairs of gates. We will see that this provides the ability to simplify circuits to a much greater degree than the original code, but it does mean that some simplifications that could be detected by a human practitioner slip through the net. At present, however, this code presents our best effort in balancing the benefits of circuit simplification against the costs involved, both in computer time and researcher time.

3.1 Double dispatch

The bottom level circuit simplification and swap routines work with pairs of gates, with the two gates likely being of different types. Furthermore, the first function called will need to accept just two pointers to Gates, i.e. pointers to the *base* class object. The code will need to be able to infer the actual type of both gates at run time. This is achieved using a technique known as *double dispatch* and described in more detail in appendix A.

Note that, at this stage, it would seem that we need 225[-1-] different functions for determining whether a pair of gates can be simplified, another 225 for determining whether a pair of gates can be swapped, and so on. However, the introduction of additional intermediate classes within the gate class hierarchy will be shown

[-1-] With 14 concrete gate types, we have the first function called, taking pointers to two Gates, 14 different possibilities for the first function that is actually run, depending on the actual type of the first Gate, which redirect to 14 different possibilities in the base class, which are then overridden by 14×14 possibilities for the final function that actually performs the test. This makes 225 function declarations and 210 function definitions, 14 of which merely redirect, while 196 do the actual work.

to reduce the number of functions actually required.

3.2 Swaps

For convenience, gate swaps have been categorized into four types.

Simple swaps: Gate swaps that require no change to be made to either gate. This was our original notion of swapping two gates. However, it become convenient to augment these with two other types of 'swap'.

Swap swaps: Gate swaps where at least one of the gates is a SwapGate. While the types of the two gates remain unchanged, the target and control bits of the non-swap gate may be changed. If both gates are SwapGates, the affected qubits of one of the gates changes.

H-Swaps: Gate swaps where one of the gates is a Hadamard and the other is an XGate (or XRotation) that gets changed to a ZGate (or ZRotation), or vice versa. These are only considered as 'swaps' if the change in gate type does not result in a change in cost.

R-Swaps: Gate swaps where one of the gates is a PhaseGate, PhaseInv, XGate, YGate or ZGate and the other is a rotation gate that gets changed in the process. This change may be a reversal of the gate's angle parameter, or a change of rotation type. In the latter case, this is only considered a 'swap' if the change in type does not result in a change in cost.

The simple swaps are managed through a single function[-2-] of the form `bool Gate::canSimplySwap(const Gate&) const`. Swap swaps are managed through a similar function called `canSwapSwap` and two auxiliary functions that indicate any change to the right moving and left moving gates respectively. These two functions return a (unique) pointer to the new gate if a change occurs, or the null pointer otherwise. H-swaps and R-swaps are handled similarly to swap swaps.

3.2.1 Simple swaps

Rather than writing 225 function declarations and 210 function definitions, handling each pair of gate types separately, the implementation of the simple swaps benefits from the presence of intermediate classes in the gate class hierarchy. The first of these is the SingleTargetGate, which we already met in section 2.1.2. The second is the DiagonalGate class[-3-], which is new and includes all single target gate types that have operations represented by a diagonal 2×2 matrix.

A default implementation of `canSimplySwap` is included in each version[-4-] of the function that occurs in the base Gate class. This default version simply returns `false` and only runs if at least one

[-2-] Actually a large number of functions, using double dispatch, but all with the same name. As noted in appendix A, the use of a single name can potentially lead to some confusion and the possibility of erroneous use. We try to help with the name of the function parameter, which is either `prev`, for the previous gate in the circuit, or `next` for the next, but this does not *enforce* correct use, of course.

[-3-] The DiagonalGate class contains almost the same concrete gate types as the PhaseTypeGate class. All PhaseTypeGates are DiagonalGates, while the ZRotation gate is, at present, the only DiagonalGate that is not a PhaseTypeGate. Note, though, that the reasons for introducing the two intermediate classes are very different. The PhaseTypeGate class is introduced due to differences in the representation of such gates and in the selection of random sets of qubits to be manipulated. The DiagonalGate class is introduced solely to aid with the simple swap routines.

[-4-] These versions differ in the type of the parameter, which is a const reference to either a Gate, a SingleTargetGate, a DiagonalGate or an Oracle.

of the gates is neither a SingleTargetGate[-5-] nor an Oracle. With our current gate set, this implies that at least one of the gates is a SwapGate.

These default function definitions are overridden whenever the pair of gate types requires different behaviour from the function, i.e. whenever gate swapping may be permitted. This happens in the cases of SingleTargetGate-SingleTargetGate, SingleTargetGate-DiagonalGate[-6-], DiagonalGate-SingleTargetGate, DiagonalGate-DiagonalGate[-7-], DiagonalGate-Oracle and Oracle-DiagonalGate.[-8-]

Function `SingleTargetGate::canSimplySwap(SingleTargetGate& prev)`[-9-] handles the case where both gates have a single target and optional controls, but neither is represented by a diagonal matrix. In this case, the gates can be swapped and the function returns `true` whenever the target of gate A is not involved in gate B and the target of gate B is not involved, in any way, with gate A. It also returns `true` if the matrices representing the two gates commute and both gates share the same target. Otherwise it returns `false`.

[-5-] Remember that the DiagonalGate is a subtype of the SingleTargetGate.

[-6-] These three override the default behaviour of the base Gate class

[-7-] These two override the SingleTargetGate behaviour.

[-8-] Technically speaking, two Oracles could also be swapped. However, this is not particularly useful, since it results in no change to the circuit and two consecutive Oracles cancel anyway.

[-9-] Omitting the return type and the `consts`.

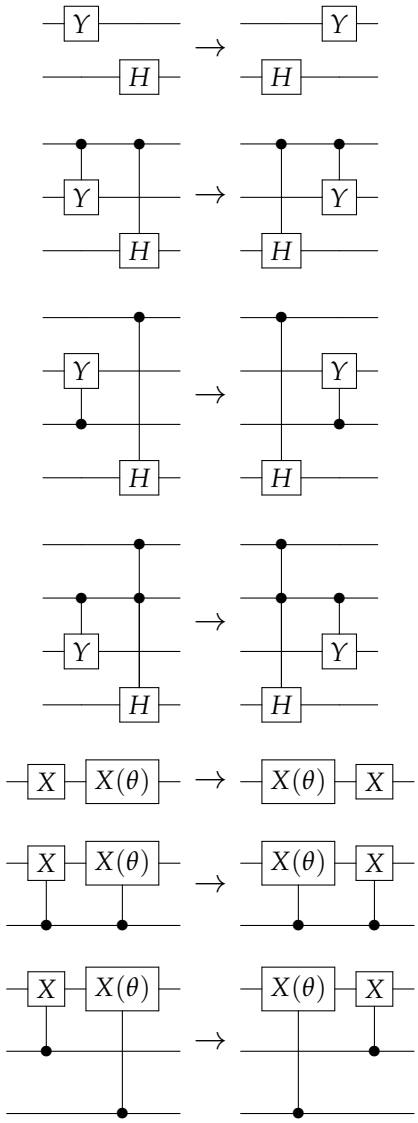


Figure 3.12: Permissible swaps of non-diagonal SingleTargetGates. The Hadamard and YGates have matrices that do not commute with each other, meaning that the gates may only share control qubits. XGates and XRotations have matrices that do commute, meaning that if they may share target bits.

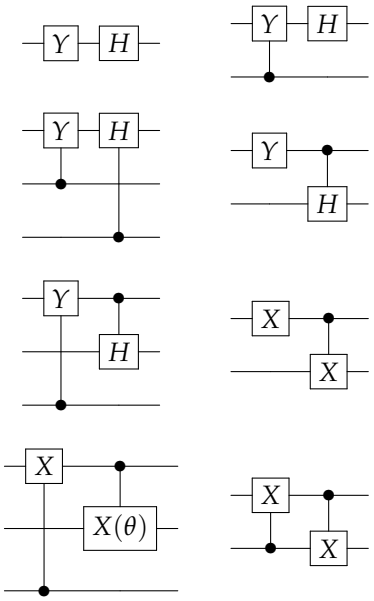


Figure 3.13: Forbidden swaps of non-diagonal SingleTargetGates. If one gate's target bit is a control of the other, then they cannot be swapped in general. If the two gates have matrices that do not commute and the gates share the same target bit, then they cannot be swapped.

Functions `SingleTargetGate::canSimplySwap(DiagonalGate& prev)` and `DiagonalGate::canSimplySwap(SingleTargetGate& prev)` handle the cases where both gates have a single target and optional controls and where precisely one of them is represented by a diagonal matrix. In this case, the gates can be swapped and the function returns `true` if and only if the target of the non-diagonal gate is uninvolved in the diagonal gate. This logic is encapsulated in the first of these two functions; the second merely calls the first via `return prev.canSimplySwap(*this)`.

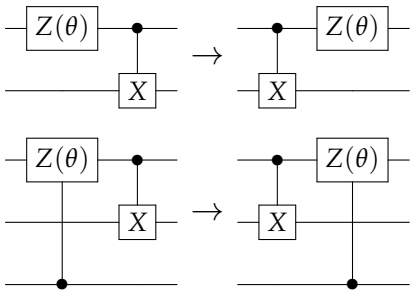


Figure 3.14: Additional permissible swaps of SingleTargetGates when one is also a DiagonalGate. We use the ZRotation gate as an exemplar DiagonalGate, since it does not have the additional properties of PhaseTypeGates. For example, in the second example, if we replace the ZRotation with a ZGate (which is a PhaseTypeGate), we could note that we could swap the target and control qubits of the ZGate and then use the basic SingleTargetGate rules for swapping the gates.

Function `DiagonalGate::canSimplySwap(DiagonalGate& prev)` handles the case where both gates are DiagonalGates. In this case, the gates can always be swapped, so the function simply returns `true`. Finally, functions `Oracle::canSimplySwap(DiagonalGate& prev)` and `DiagonalGate::canSimplySwap(Oracle& prev)` handle the case where one gate is an Oracle and the other is a DiagonalGate. In this case, both gates multiply chosen basis states by complex numbers (phases), without mixing the basis states in any way. Since multiplication of numbers is commutative, these functions also simply return `true`.

While this handles all of the simple swaps considered by the code, notice that when considering swapping two non-diagonal single target gates, the code has to determine whether the matrices

for the two gates commute. This information is found by calling the function `bool SingleTargetGate::matricesCommute(const SingleTargetGate& other) const`. Yet again, this function works by using double dispatch. It is declared, and a default definition is provided in the SingleTargetGate class and is overridden in the DiagonalGate class. It is also overridden in two new intermediate classes: XTypeGate and YTypeGate. The default definition merely checks to see if the gates are of the same concrete type[-10-], returning `true` if so and `false` otherwise. The overrides adjust this rule by noting that two XType gates (of possibly different concrete type) have matrices that commute, as do two YType gates and two DiagonalGates.

[-10-] This is one of the few places in the code that has lines like `return typeid(*this) == typeid(other)`. Avoiding this kind of explicit checking that the types of two objects are the same is usually recommended.

3.2.2 *Swap swaps*

The functions for swap swaps — that is, swaps that involve a swap gate — are structured in a similar manner to the simple swaps. The first function called, `Gate::canSwapSwap(Gate& next)`[-11-] is overridden in the SingleTargetGate and SwapGate classes by `SingleTargetGate::canSwapSwap(Gate& next)` and `SwapGate::canSwapSwap(Gate& next)`. These call `Gate::canSwapSwap(SingleTargetGate& prev)` and `Gate::canSwapSwap(SwapGate& next)` (which provide the default option of returning `false`), which are overridden by `SwapGate::canSwapSwap(SingleTargetGate& prev)`, `SingleTargetGate::canSwapSwap(SwapGate& prev)` and `SwapGate::canSwapSwap(SwapGate& prev)`. Each of these three overrides simply returns `true` to indicate that the gates may be swapped. Note that, in this case, there is no override for `SingleTargetGate::canSwapSwap(SingleTargetGate& prev)` — if the original function is called with two SingleTargetGates, then the function that finally runs is `Gate::canSwapSwap(SingleTargetGate& prev)`, i.e. one of the default options that returns `false`.[-12-]

[-11-] Again, omitting the return type and `const`s.

In addition to the function `canSwapSwap`, the code provides `rightSwapMoverChange` and `leftSwapMoverChange` that indicate whether the gate moving to the right (or left) is changed in the process and return a pointer to the new gate. The gate types do not change, but the qubits involved in the gate may. This is illustrated in figure 3.15 which illustrates possible swaps involving a SwapGate and a SingleTargetGate.

[-12-] A less detailed, less technically correct, but probably more coherent description is that the default function returns `false`, with exceptions when one gate is a SwapGate and the other is a SwapGate or a SingleTargetGate, when the function returns `true`.

In the case where both gates are SwapGates there is a choice as to which of the two gates gets modified, as shown in figure 3.16. However, it turns out only to be necessary to provide the option with the lowest 'sort value', i.e. the option that would come first if the circuits were to be sorted.[-13-] Here this means comparing the two options for the first gate, after swapping, and deciding which should be sorted first. In the illustrated case, assuming that qubits are numbered from 0 from the bottom, the first gate in the first option involves qubits 0 and 1, while in the second option the

[-13-] The notion of sorting gates and circuits is explored in more detail in section 3.4.

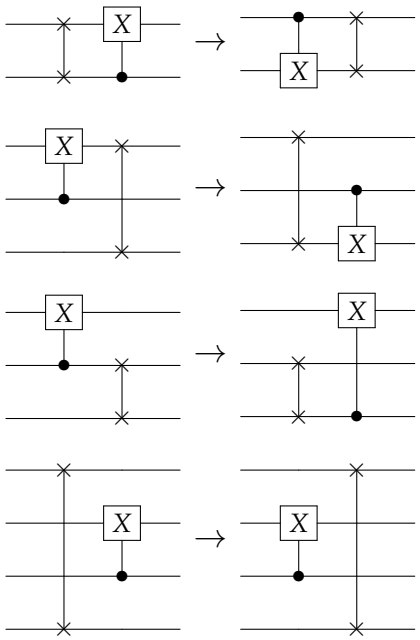


Figure 3.15: Permissible swap swaps involving a SingleTargetGate.

Figure 3.16: Two choices for swapping two SwapGates.

first gate swaps 0 and 2. Using lexicographical sorting, 01 is sorted before 02, so the code selects the first option when swapping these two SwapGates.

3.2.3 *H-swaps*

H-swaps are those gate swaps where exactly one of the gates is a Hadamard and both gates share[-14-] the same target bit. Note that these operations are only considered to be swaps if the overall gate cost of the circuit is not changed. If, for some reason, a ZGate is cheaper than an XGate then swapping an XGate and a Hadamard to produce a Hadamard and a ZGate is considered a *simplification*. The reverse operation, which makes the circuit more expensive, is simply not considered by the code.

[-14-] Or, in the case of a ZGate, can be arranged to share the same target bit.

Having already discussed the double dispatch mechanism for the previous two cases, here we describe only the functions that override those provided in the base Gate class (which naturally all return `false`).

XGate – Hadamard: Gates can be H-swapped provided they share the same target qubit, controls of the Hadamard are also controls of the XGate and a suitable ZGate is available. In moving past the Hadamard, the XGate becomes a ZGate with the same target

and controls[-15-]. In other cases, the function `canHSwap` returns false. Note that this does not necessarily mean that the gates cannot be swapped, though these other cases are handled by `canSimplySwap`.

[-15-] Recall, however, that since a ZGate is a PhaseTypeGate, one may swap the target for one of the controls. Indeed, if the original target qubit does not have the lowest index amongst the involved qubits, the code performs such a swap to get the gate into 'canonical form'.

$$HX = \frac{1}{\sqrt{2}}\begin{pmatrix}1 & 1\\ 1 & -1\end{pmatrix}\begin{pmatrix}0 & 1\\ 1 & 0\end{pmatrix} = \frac{1}{\sqrt{2}}\begin{pmatrix}1 & 1\\ -1 & 1\end{pmatrix}$$
$$= -\frac{1}{\sqrt{2}}\begin{pmatrix}1 & 0\\ 0 & -1\end{pmatrix}\begin{pmatrix}1 & 1\\ 1 & -1\end{pmatrix} = ZH.$$

Hadamard – XGate: As above, though, of course, the identities of the left and right movers are swapped. The functions are implemented so that they simply reroute the call to those for the above case. All other gate pairs that start with the Hadamard are treated similarly.

YGate – Hadamard: Note that

$$HY = \frac{1}{\sqrt{2}}\begin{pmatrix}1 & 1\\ 1 & -1\end{pmatrix}\begin{pmatrix}0 & -i\\ i & 0\end{pmatrix} = \frac{1}{\sqrt{2}}\begin{pmatrix}i & -i\\ -i & -i\end{pmatrix}$$
$$= -\frac{1}{\sqrt{2}}\begin{pmatrix}0 & -i\\ i & 0\end{pmatrix}\begin{pmatrix}1 & 1\\ 1 & -1\end{pmatrix} = -YH,$$

i.e. we get an overall change of phase. This is a problem when the gates have controls, as the change in phase only occurs for basis states where the control qubits are set. However, provided both gates share the same target and neither have controls, the gates may be H-swapped. Again, other possible swaps are covered by the 'simple' swaps.

ZGate – Hadamard: Gates can be H-swapped provided all the qubits involved in the Hadamard are also involved in the ZGate and provided a suitable XGate is available. Note that target qubits may, initially, disagree, but recall that the target qubit of the ZGate may always be swapped with a control qubit without affecting gate operation.

$$HZ = \frac{1}{\sqrt{2}}\begin{pmatrix}1 & 1\\ 1 & -1\end{pmatrix}\begin{pmatrix}1 & 0\\ 0 & -1\end{pmatrix} = \frac{1}{\sqrt{2}}\begin{pmatrix}1 & -1\\ 1 & 1\end{pmatrix}$$
$$= -\frac{1}{\sqrt{2}}\begin{pmatrix}0 & 1\\ 1 & 0\end{pmatrix}\begin{pmatrix}1 & 1\\ 1 & -1\end{pmatrix} = XH,$$

XRotation – Hadamard: As for the first case, but with XGates and ZGates replaced by XRotations and ZRotations.[-16-]

[-16-] One minor difference arises from the fact that a ZRotation is *not* a PhaseTypeGate. Hence there is no need to consider swapping the target bit for one of the controls.

$$HX(\theta) = \frac{1}{\sqrt{2}}\begin{pmatrix}1 & 1\\ 1 & -1\end{pmatrix}\begin{pmatrix}\cos(\theta/2) & i\sin(\theta/2)\\ i\sin(\theta/2) & \cos(\theta/2)\end{pmatrix}$$
$$= \frac{1}{\sqrt{2}}\begin{pmatrix}e^{i\theta/2} & e^{i\theta/2}\\ e^{-i\theta/2} & -e^{-i\theta/2}\end{pmatrix}$$
$$= -\frac{1}{\sqrt{2}}\begin{pmatrix}e^{i\theta/2} & 0\\ 0 & e^{-i\theta/2}\end{pmatrix}\begin{pmatrix}1 & 1\\ 1 & -1\end{pmatrix} = Z(\theta)H,$$

YRotation – Hadamard: Note that

$$HY(\theta) = \frac{1}{\sqrt{2}}\begin{pmatrix}1 & 1\\ 1 & -1\end{pmatrix}\begin{pmatrix}\cos(\theta/2) & \sin(\theta/2)\\ -\sin(\theta/2) & \cos(\theta/2)\end{pmatrix}$$
$$= \frac{1}{\sqrt{2}}\begin{pmatrix}\cos(\theta/2)-\sin(\theta/2) & \cos(\theta/2)+\sin(\theta/2)\\ \cos(\theta/2)+\sin(\theta/2) & -\cos(\theta/2)+\sin(\theta/2)\end{pmatrix}$$

while

$$Y(\theta)H = -\frac{1}{\sqrt{2}}\begin{pmatrix}\cos(\theta/2) & \sin(\theta/2)\\ -\sin(\theta/2) & \cos(\theta/2)\end{pmatrix}\begin{pmatrix}1 & 1\\ 1 & -1\end{pmatrix}$$
$$= \frac{1}{\sqrt{2}}\begin{pmatrix}\cos(\theta/2)+\sin(\theta/2) & \cos(\theta/2)-\sin(\theta/2)\\ \cos(\theta/2)-\sin(\theta/2) & -\cos(\theta/2)-\sin(\theta/2)\end{pmatrix}$$

so that $HY(\theta) = Y(-\theta)H$. So, provided the target qubits agree and all of the Hadamard's control qubits are also controls of the YRotation, the gates can be swapped, but the rotation angle must be multiplied by -1.

ZRotation – Hadamard: Gates can be H-swapped provided the target qubits agree, all of the Hadamard's control qubits are also controls of the ZRotation and a suitable XRotation is available.

$$HZ(\theta) = \frac{1}{\sqrt{2}}\begin{pmatrix}1 & 1\\ 1 & -1\end{pmatrix}\begin{pmatrix}e^{i\theta} & 0\\ 0 & e^{-i\theta}\end{pmatrix} = \frac{1}{\sqrt{2}}\begin{pmatrix}e^{i\theta} & e^{-\theta}\\ e^{i\theta} & -e^{-i\theta}\end{pmatrix}$$
$$= -\frac{1}{\sqrt{2}}\begin{pmatrix}\cos(\theta/2) & i\sin(\theta/2)\\ i\sin(\theta/2) & \cos(\theta/2)\end{pmatrix}\begin{pmatrix}1 & 1\\ 1 & -1\end{pmatrix} = X(\theta)H,$$

Some example circuits where swaps cannot be performed are shown in figure 3.17, while permissible H-swaps are illustrated in figure 3.18.

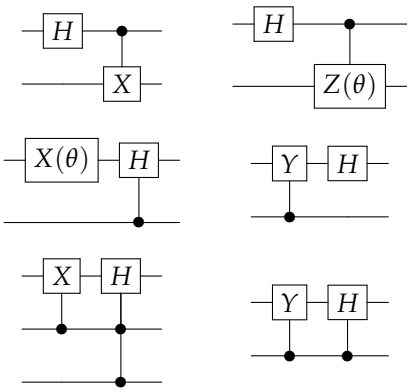


Figure 3.17: Example circuits with no H swaps available.

3.2.4 *R-swaps*

R-swaps are similar to H-swaps, in that one of the gates involved changes identity, either by changing type or through a change in the angle parameter. The changed gate is one of the rotation gates, i.e. XRotation, YRotation, ZRotation or ArbitraryPhase, while the unchanged gate can be a PhaseGate, PhaseInv, XGate, YGate or ZGate. As with H-swaps, these changes are only considered to be swaps if the overall cost of the circuit is unchanged[17].

[17] This will be the usual case. The only possible exception would be if XRotations and YRotations have different costs, which would be odd.

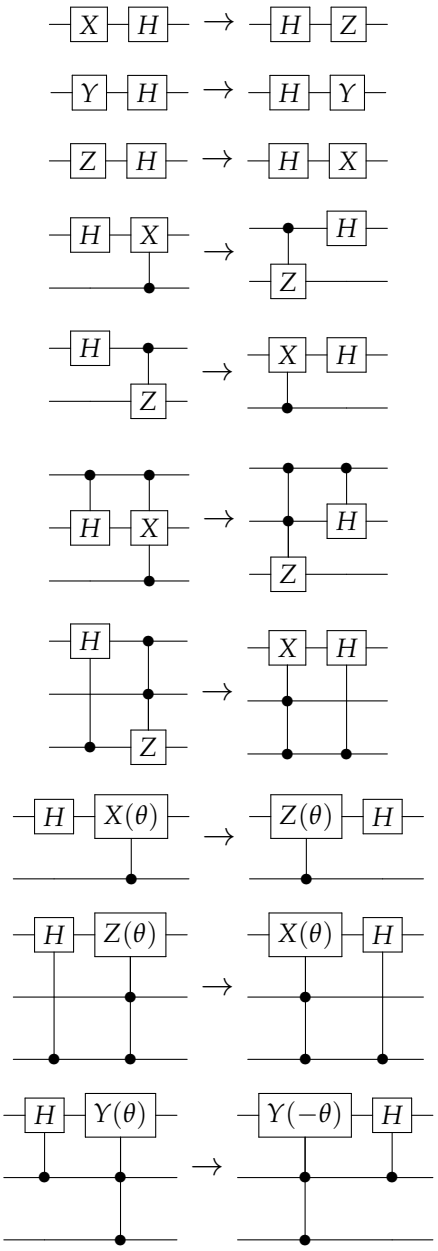


Figure 3.18: Permissible H swaps. Note that here we are careful to ensure that a ZGate's target bit has a lower index than its control bits at all times. (Qubits are indexed from zero, starting at the bottom of the circuit.)

XGate – YRotation: These gates can be R-swapped provided they share the same target qubit and controls of the XGate are also controls of the rotation. In moving past the XGate, the YRotation has its angle reversed. In other cases, the function `canRSwap` returns false. Note that this does not necessarily mean that the gates cannot be swapped: other cases are handled by `canSimplySwap`.

$$\begin{aligned} Y(\theta)X &= \begin{pmatrix} \cos(\theta/2) & \sin(\theta/2) \\ -\sin(\theta/2) & \cos(\theta/2) \end{pmatrix} \begin{pmatrix} 0 & 1 \\ 1 & 0 \end{pmatrix} \\ &= \begin{pmatrix} \sin(\theta/2) & \cos(\theta/2) \\ \cos(\theta/2) & -\sin(\theta/2) \end{pmatrix} \\ &= \begin{pmatrix} 0 & 1 \\ 1 & 0 \end{pmatrix} \begin{pmatrix} \cos(\theta/2) & -\sin(\theta/2) \\ \sin(\theta/2) & \cos(\theta/2) \end{pmatrix} = XY(-\theta). \end{aligned}$$

Note that these are identical — there is no overall phase to concern us when we consider controls. Provided the XGate is 'active' whenever the YRotation is, the swap can be performed. In contrast, if the XGate has a control that is not shared by the YRotation, then it is possible that only the YRotation is operational. As a YRotation with angle θ is different from one with angle $-\theta$, the swap cannot be performed in this case.

The next three cases can now be derived from this case using symmetry arguments.

XGate – ZRotation: These gates can be R-swapped provided they share the same target qubit and controls of the XGate are also controls of the rotation. In moving past the XGate, the ZRotation has its angle reversed.

YGate – XRotation: These gates can be R-swapped provided they share the same target qubit and controls of the YGate are also controls of the rotation. In moving past the YGate, the XRotation has its angle reversed.

YGate – ZRotation: These gates can be R-swapped provided they share the same target qubit and controls of the YGate are also controls of the rotation. In moving past the YGate, the ZRotation has its angle reversed.

ZGate – XRotation: These can be R-swapped provided all the qubits involved in the ZGate are also involved in the rotation. In moving past the ZGate, the XRotation has its angle reversed. This case can also be derived from the XRotation–YGate case using symmetry arguments. The only difference is due to the fact that we can swap the target qubit of the ZGate with any of its controls.

ZGate – YRotation: These can be R-swapped provided all the qubits involved in the ZGate are also involved in the rotation. In moving past the ZGate, the YRotation has its angle reversed.

XGate – ArbitraryPhase: The ArbitraryPhase gate introduces phase shifts that complicate matters when there are controls.

$$\begin{aligned}\Phi(\theta)X &= \begin{pmatrix}1 & 0\\ 0 & e^{i\theta}\end{pmatrix}\begin{pmatrix}0 & 1\\ 1 & 0\end{pmatrix} = \begin{pmatrix}0 & 1\\ e^{i\theta} & 0\end{pmatrix}\\ &= e^{i\theta}\begin{pmatrix}0 & e^{-i\theta}\\ 1 & 0\end{pmatrix} = e^{i\theta}\begin{pmatrix}0 & 1\\ 1 & 0\end{pmatrix}\begin{pmatrix}1 & 0\\ 0 & e^{-i\theta}\end{pmatrix} = e^{i\theta}X\Phi(-\theta).\end{aligned}$$

The phase of $e^{i\theta}$ means that the swap can only be performed if neither gate has controls.

YGate – ArbitraryPhase: As for XGate – ArbitraryPhase.

XRotation – PhaseGate: These can be R-swapped provided they share the same target qubit and controls of the PhaseGate are

also controls of the rotation. In moving past the PhaseGate, the XRotation becomes a YRotation of the same angle. To be considered an R-swap, the replacement YRotation gate must have the same cost as the old XRotation.

$$\begin{aligned} SX(\theta) &= \begin{pmatrix} 1 & 0 \\ 0 & i \end{pmatrix} \begin{pmatrix} \cos(\theta/2) & i\sin(\theta/2) \\ i\sin(\theta/2) & \cos(\theta/2) \end{pmatrix} \\ &= \begin{pmatrix} \cos(\theta/2) & i\sin(\theta/2) \\ -\sin(\theta/2) & i\cos(\theta/2) \end{pmatrix} \\ &= \begin{pmatrix} \cos(\theta/2) & \sin(\theta/2) \\ -\sin(\theta/2) & \cos(\theta/2) \end{pmatrix} \begin{pmatrix} 1 & 0 \\ 0 & i \end{pmatrix} = Y(\theta)S. \end{aligned}$$

PhaseGate – XRotation: These can be R-swapped provided they share the same target qubit and controls of the PhaseGate are also controls of the rotation. In moving past the PhaseGate, the XRotation becomes a YRotation of *opposite* angle. To be considered an R-swap, the replacement YRotation gate must have the same cost as the old XRotation.

$$\begin{aligned} X(\theta)S &= \begin{pmatrix} \cos(\theta/2) & i\sin(\theta/2) \\ i\sin(\theta/2) & \cos(\theta/2) \end{pmatrix} \begin{pmatrix} 1 & 0 \\ 0 & i \end{pmatrix} \\ &= \begin{pmatrix} \cos(\theta/2) & -\sin(\theta/2) \\ i\sin(\theta/2) & i\cos(\theta/2) \end{pmatrix} \\ &= \begin{pmatrix} 1 & 0 \\ 0 & i \end{pmatrix} \begin{pmatrix} \cos(\theta/2) & -\sin(\theta/2) \\ \sin(\theta/2) & \cos(\theta/2) \end{pmatrix} = SY(-\theta). \end{aligned}$$

YRotation – PhaseGate: These can be swapped provided they share the same target qubit and controls of the PhaseGate are also controls of the rotation. In moving past the PhaseGate, the YRotation becomes an XRotation of opposite angle. Note that this is simply the reverse of the PhaseGate – XRotation case.

PhaseGate – YRotation: These can be swapped provided they share the same target qubit and controls of the PhaseGate are also controls of the rotation. In moving past the PhaseGate, the YRotation becomes an XRotation of the same angle. (The reverse of the XRotation – PhaseGate case.)

XRotation – PhaseInv: These can be R-swapped provided they share the same target qubit and controls of the PhaseInv are also controls of the rotation. In moving past the PhaseInv, the XRotation becomes a YRotation of opposite angle.

$$\begin{aligned} S^{-1}X(\theta) &= \begin{pmatrix} 1 & 0 \\ 0 & -i \end{pmatrix} \begin{pmatrix} \cos(\theta/2) & i\sin(\theta/2) \\ i\sin(\theta/2) & \cos(\theta/2) \end{pmatrix} \\ &= \begin{pmatrix} \cos(\theta/2) & i\sin(\theta/2) \\ \sin(\theta/2) & -i\cos(\theta/2) \end{pmatrix} \\ &= \begin{pmatrix} \cos(\theta/2) & -\sin(\theta/2) \\ \sin(\theta/2) & \cos(\theta/2) \end{pmatrix} \begin{pmatrix} 1 & 0 \\ 0 & -i \end{pmatrix} = Y(-\theta)S^{-1}. \end{aligned}$$

PhaseInv – XRotation: These can be R-swapped provided they share the same target qubit and controls of the PhaseInv are also controls of the rotation. In moving past the PhaseInv, the XRotation becomes a YRotation of the same angle.

$$\begin{aligned} X(\theta)S^{-1} &= \begin{pmatrix} \cos(\theta/2) & i\sin(\theta/2) \\ i\sin(\theta/2) & \cos(\theta/2) \end{pmatrix} \begin{pmatrix} 1 & 0 \\ 0 & -i \end{pmatrix} \\ &= \begin{pmatrix} \cos(\theta/2) & \sin(\theta/2) \\ i\sin(\theta/2) & -i\cos(\theta/2) \end{pmatrix} \\ &= \begin{pmatrix} 1 & 0 \\ 0 & -i \end{pmatrix} \begin{pmatrix} \cos(\theta/2) & \sin(\theta/2) \\ -\sin(\theta/2) & \cos(\theta/2) \end{pmatrix} = S^{-1}Y(\theta). \end{aligned}$$

YRotation – PhaseInv: These can be swapped provided they share the same target qubit and controls of the PhaseGate are also controls of the rotation. In moving past the PhaseGate, the YRotation becomes an XRotation of the same angle. (The reverse of the PhaseInv – XRotation case.)

PhaseInv – YRotation: These can be swapped provided they share the same target qubit and controls of the PhaseInv are also controls of the rotation. In moving past the PhaseGate, the YRotation becomes an XRotation of opposite angle. (The reverse of the XRotation – PhaseInv case.)

Some permissible R-swaps are illustrated in figure 3.19.

3.3 *The Gate hierarchy revisited*

Having added the XTypeGate, YTypeGate and DiagonalGate classes, we have completed the Gate class hierarchy. This is illustrated in figure 3.20.

3.4 *Canonical form*

While the use of a single vector of Gates to represent a quantum circuit is straightforward, this representation has a great deal of redundancy. A person's notion of a unique quantum circuit may be represented by a number of different gate sequences, depending on how non-interacting gates are ordered.[-18-] This redundancy causes problems if we wish to cache the results of circuit evaluation. A solution is to convert each circuit, through the use of the swap operations described above, to a 'canonical form'. This results in different gate sequences, for essentially the same circuit, being converted to a single sequence before performing any interaction with the cache.

[-18-] A simply example of this was seen in section 1.3.3 when discussing the limitations of the old simplification routines.

The conversion of circuits to canonical form relies on a notion of circuit equivalence. This document will use several different notions of circuit equivalence at different points, so it is wise to be clear what is meant in each circumstance. Here, two circuits

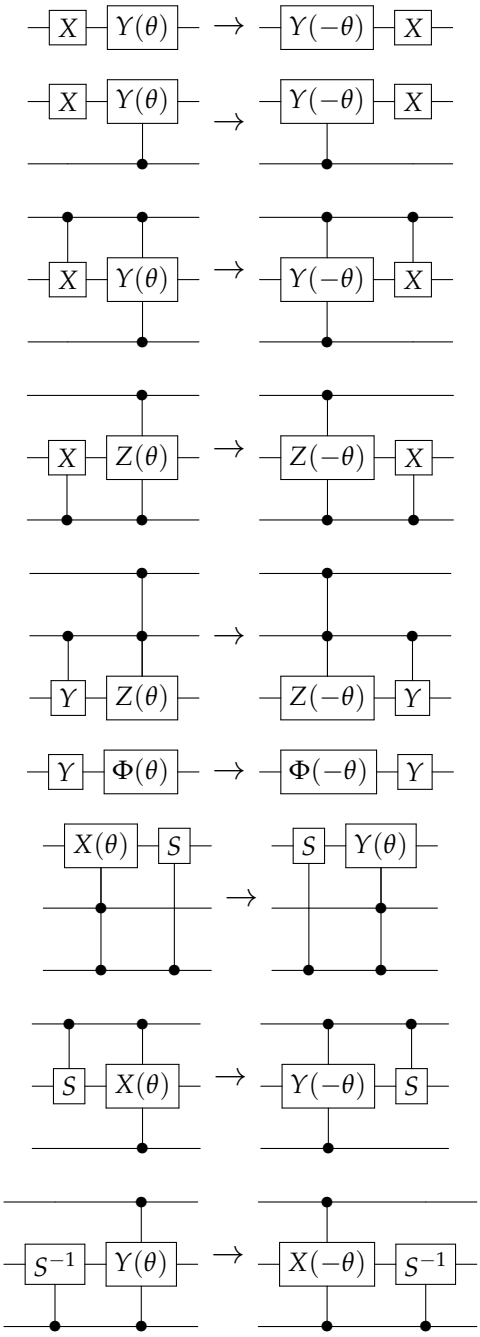


Figure 3.19: Permissible R swaps.

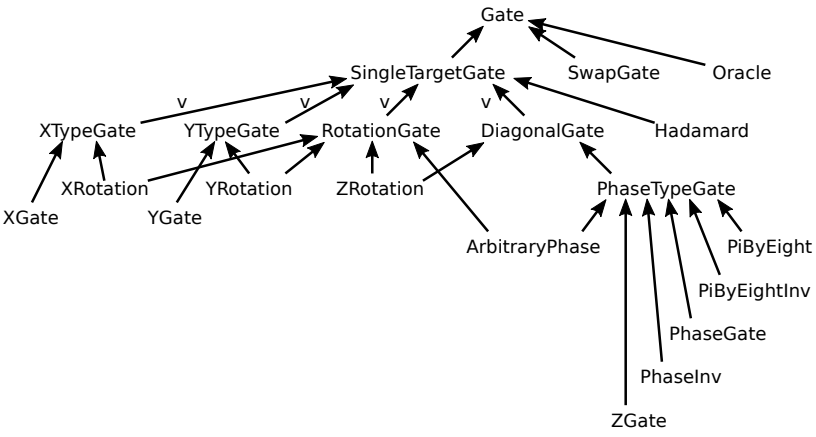


Figure 3.20: The full Gate class hierarchy. Virtual inheritance is marked with the letter 'v'.

represented by two different gate sequences are equivalent if one can be converted to the other through the use of swap operations alone. Here, swap operations are those described in the previous section, i.e. operations that are applied to pairs of adjacent gates, producing a replacement gate pair that is of equal cost, without changing the result of applying the circuit to a state (up to a global phase).

In addition to specifying the means used to convert between equivalent circuit, we need to somehow specify which of many equivalent circuits is in the preferred, canonical form. We do this by assigning a preferred ordering of the various gate options[-19-], by defining a `sortBefore` function. The preferred sequencing follows some simple rules.

1. Gates of different types should be sequenced, if possible, according to the ordering of gate types in section 2.1.1. That is, a Hadamard 'sorts before' a PiByEight, which sorts before a PiByEightInv, and so on. Note that in a circuit without Oracles, swap gates are moved to the end of the circuit, since they can be swapped with and 'sort after' gates of all other types (except Oracles).

2. If the gates are both of the same type, inheriting from SingleTargetGate, then the gate with the least target qubit index is sorted before the other.

3. If target qubits agree, the preferred sequence is determined by comparing the controls.[-20-]

4. If gate types and target qubits agree, the gate parameters are compared, using the 'less than' operator.

5. Given two SwapGates, we first compare the first (i.e. least) qubit for each gate, using the 'less than' operator. If these are equal, we compare the second qubit. Hence a swap of qubits 0 and 2 sorts before a swap of qubits 1 and 2 and after a swap of qubits 0 and 1.

[-19-] We will see later, when we discuss cacheing in more detail, that we actually go further than merely specifying an ordering, by assigning a unique integer to each gate option.

[-20-] In more detail, the vectors of bools used to represent the controls are compared, using the 'less than' operator. Recalling that vectors are compared lexicographically, while `false < true`, we note that, given the same gate type and target qubits, a gate with no controls sorts before one with one or more, while a gate with a single control on qubit 0 sorts *after* a gate with a single control on qubit 1. (In this last case, the first comparison is of the bools stored for qubit 0 and `true > false`.)

This preferred ordering for the gates is then used to define a sorting operator for circuits. First, if two circuits are of different length, the one with the fewest gates is sorted first.[-21-] Then, if the circuits are of the same length, they are sorted using lexicographical comparison.

These definitions of preferred orderings of gates and circuits lead directly to the definition of the canonical representation. The canonical representation is simply the 'least' circuit equivalent to the circuit under consideration, using the definition of equivalence given above. However, there remains the question of how to find this canonical form. While the simulation of circuits is likely to still be the source of greatest computational expense, the algorithm used to place a circuit in canonical form is called many times — after circuit creation and after each step of circuit simplification — so some consideration of efficiency is worthwhile, especially when optimizing small circuits.

At first sight, this looks like a simple sorting problem. However, two complications mean that we cannot simply apply quick sort.

1. The final placement of gates is constrained by the fact that some gate pairs cannot be swapped. For example, if we have an XGate on qubit 1 followed by a controlled Hadamard with target qubit 0 and control qubit 1 then, while the 'sort ordering' defined above would place the Hadamard first, swapping these gates is not permitted.

2. The identity of gates can be changed when gates are swapped, for example when one of the gates is a Hadamard.

We have implemented a modified version of insertion sort, shown in figure 3.21, to perform this task. Each gate is considered in turn. In the first part of the main loop, the code searches for the best place to insert the gate under consideration. This place is the earliest position in the circuit where it is both possible to move the gate, using swaps, and where the resulting circuit is 'better' according to the circuit sorting condition. Unlike in ordinary insertion sort, where we could use a binary search, or search from the first position, here the algorithm searches backwards from the current position of the gate, due to the additional requirement that it be possible, using swaps, to get the gate to the desired position. This part of the algorithm also records any instances where the identity (either the gate type or the gate target and controls) of any gates will be changed by a swap.

The second part of the main loop performs the insertion of the gate into the desired location, moving the displaced gates up and changing them if required. A key difference with plain insertion sort is the addition of lines 26–32. These are included since a previously sorted section of circuit may become unsorted when the gates change identity, so it may be necessary to revisit previously sorted gates.[-22-]

[-21-] This naturally has no influence on the algorithm for putting a circuit into canonical form, since the circuit retains the same number of gates throughout.

[-22-] Using the types of swaps described above and the specified sort order for gates, it can be shown that this algorithm correctly rearranges the gates. Hypothetically, if we were to incorporate additional types of swap, we might find that the algorithm cycles indefinitely.

```
1: procedure Circuit::makeCanonical( )
2:     changedGates[0 to length( ) − 1] ← empty
3:     movingPos ← 0
4:     while movingPos < length( ) do
5:         movingGate ← gates[movingPos]
6:
7:                                                  ▷ Find where to move the moving gate to.
8:         bestPos ← movingPos
9:         candidatePos ← movingPos − 1
10:        candidateGate ← gates[candidatePos]
11:        while candidateGate.canSwap(movingGate) do
12:            changedGates[candidatePos] ← candidateGate.rightMoverChange(movingGate)
13:            movingGate ← candidateGate.leftMoverChange(movingGate)
14:            if movingGate.sortBefore(candidateGate) then
15:                bestPos ← candidatePos
16:                bestGate ← movingGate
17:            end if
18:            candidatePos ← candidatePos − 1
19:            candidateGate ← gates[candidatePos]
20:        end while
21:
22:                                  ▷ Move gate, possibly changing this or other gates in the process.
23:                                          ▷ Changed gates might need to be revisited.
24:        for changePos ← movingPos down to bestPos + 1 do
25:            gates[changePos] ← changedGates[changePos − 1]
26:            if gates[changePos] ≠ gates[changePos − 1] then                ▷ Gate has changed
27:                if changePos − 1 = bestPos then
28:                    movingPos ← changePos                  ▷ Revisit gate after changed gate.
29:                else
30:                    movingPos ← changePos − 1                          ▷ Revisit changed gate.
31:                end if
32:            end if
33:        end for
34:        gates[bestPos] ← bestGate
35:        movingPos ← movingPos + 1
36:    end while
37: end procedure
```

Figure 3.21: Pseudocode for converting a circuit to canonical form. The actual code is more complex, handling the three types of swap differently and making extensive use of pointers. Functions that indicate changes to gates might not return a (pointer to a) gate, but a null pointer to indicate no change. The pseudocode also ignores the possibility that candidatePos becomes negative, in which case we assume that a swap cannot be performed.

3.5 *Simplifications*

During implementation of the functions determining whether a pair of gates can be swapped, we were able to evade the prospect of writing 225 versions of each function, by using intermediate classes within the Gate class hierarchy and implementing general rules for swapping gates. For the simplification of pairs of gates, either through cancellation or the replacement of the gates with a cheaper equivalent gate combination, such general rules cannot be applied. We thankfully still do not have to write 225 versions of each function, since the default option for `canSimplify` — returning false to indicate that the gate pair does not simplify — is often correct. However, writing all of the simplification routines is more of a chore than writing the swap functions.

Before describing the simplification routines in detail, we need to describe a different notion of circuit equivalence than that of the previous section. Moreover, it is also useful to discuss the related, but subtly different notion of circuit *structure* equivalence. When considering the simplification of circuits, two circuits are equivalent if they produce the same output for any input, up to an overall phase that must be the same for all inputs. Since such an overall phase shift is undetectable, two equivalent circuits perform the same task. They may have different costs (in contrast to the previous section) and different numbers of gates. Two circuit *structures* are equivalent if, given any settings for the gate parameters in either of the circuits, settings can be selected in the other circuit so as to result in equivalent circuits. Notice that this means that two equivalent circuits may have inequivalent circuit structures,[-23-] as shown in figure 3.23.

[-23-] And, obviously, we may select gate parameter values to create inequivalent circuits from two equivalent circuit structures, so long as they include parameterized gates. (See figure 3.24.)

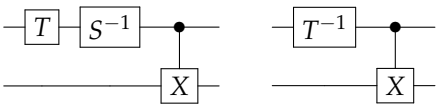


Figure 3.22: Equivalent circuits with equivalent circuit structures.

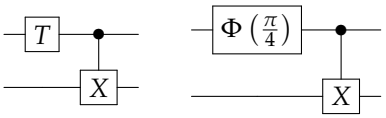


Figure 3.23: Equivalent circuits with inequivalent circuit structures. Φ indicates an ArbitraryPhase gate.

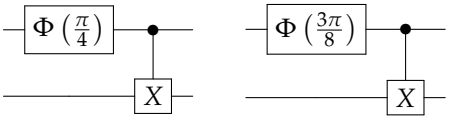


Figure 3.24: Inequivalent circuits with equivalent circuit structures.

The relationship between circuit equivalence and circuit structure equivalence is related to the notion of the number of 'useful degrees of freedom'. This notion captures the number of useful dimensions of the parameter space that is explored, post-simplification, by the numerical optimization routine. Equivalent circuit structures always have the same number of useful degrees of freedom. This number is often, but not always, the same as the number of gate parameters

in the circuit. An exception is shown in figure 3.25, which shows two equivalent circuits, with equivalent structure, but where the left hand circuit includes a 'useless' degree of freedom; the effect of changing the angle of one of the gates can always be mimicked by changing the angle of the other.

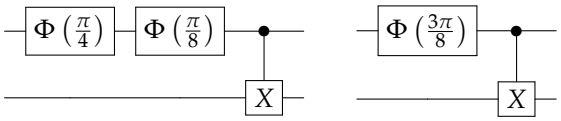


Figure 3.25: Equivalent circuits with equivalent circuit structures. Both have only one useful degree of freedom. For the left hand circuit structure, one can obtain all possible circuits by adjusting the angle parameter of just the first gate.

The simplification routines to be described shortly attempt to convert a circuit into an equivalent *circuit* that is as cheap as possible. Note that the process of simplification may result in an inequivalent circuit *structure*. However, when this is the case, note that the number of useful degrees of freedom only increases, with the result that the new circuit structure is capable of emulating all circuits that can be produced from the old structure, plus additional possibilities. In this situation, the new circuit structure not only has a lower cost, but can be considered to be more flexible.

The simplification routines consist of multiple versions of two basic functions: `pair<int, bool> Gate::canSimplify(const Gate&) const` and `GateSequence Gate::simplification(const Gate&) const`, where `GateSequence` is an alias for `vector<unique_ptr<Gate> >`. The first of these indicates whether a simplification is (potentially) available by returning the cost improvement and a boolean value indicating whether it increases the number of useful degrees of freedom for the numerical algorithm.[-24-] The second produces the replacement gate sequence that produces this reduction in cost.

The function `Gate::canSimplify(Gate& next)`[-25-] is *pure virtual*, meaning that no definition is supplied. This is fine, since every derived concrete class provides an override for this function, each of which simply performs the double dispatch. Default versions of the function called in the second dispatch, i.e. functions of the form `Gate::canSimplify(Hadamard& prev)`, where `Hadamard` may be replaced by any concrete gate type, simply suggest that no simplification can take place. The rest of this section describes those functions that override these default cases.

[-24-] If no simplification can be found, `canSimplify` returns 0 and `false`.

[-25-] Ignoring return type and `const`s again.

Hadamard–Hadamard: Provided both gates have the same target qubits and the same controls, these cancel. Cost improvement is simply the cost of the original gates. The number of useful degrees of freedom is not increased.

Hadamard–XGate: If the two gates share the same target qubits and the controls of the Hadamard are also controls of the XGate, then these gates may be converted into a ZGate–Hadamard, provided the ZGate is available. Note, though, that this is only considered to be a simplification if the ZGate is cheaper than the XGate being replaced. There is also the possibility that the

ZGate is unavailable, but an ArbitraryPhase gate is available and cheaper than the XGate.[-26-] In this case, an ArbitraryPhase gate equivalent to a ZGate is used instead. In this case, a useful extra degree of freedom is produced.

[-26-] Note that, if both the ZGate and the ArbitraryPhase are available, then the ZGate will be cheaper. If the user input indicates that the ArbitraryPhase is cheaper, then the ZGate is considered redundant and eliminated on construction of the CircuitStructure.

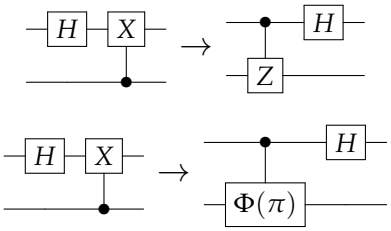


Figure 3.26: Simplifications of Hadamard–XGate, assuming that the replacement for the XGate is cheaper.

Hadamard–ZGate: If all the qubits involved in the Hadamard are also involved in the ZGate, then these gates may be converted into a XGate–Hadamard, provided the XGate is available. This is a simplification if the XGate is cheaper than the ZGate. (See figure 3.18.)

Hadamard–XRotation: See figure 3.18 in section 3.2.3. The illustrated swaps are considered simplifications if the ZRotation is, for some reason, cheaper than the XRotation.

Hadamard–ZRotation: See section 3.2.3.

PiByEight–PiByEight: Provided both gates work with the same set of qubits[-27-], we may replace two PiByEights by a PhaseGate (assuming that a PhaseGate is available and cheaper than two PiByEights). If a PhaseGate is unavailable, but an ArbitraryPhase is available and cheap enough, an ArbitraryPhase of angle $\pi/2$ may be used instead. In this case, the number of useful degrees of freedom is increased.

[-27-] Since PhaseTypeGates are always represented with the target qubit being the 'least' of the involved qubits, this is the same as saying that the targets match and the controls match.

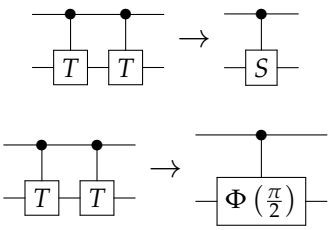


Figure 3.27: Simplifications of PiByEight–PiByEight, assuming that the replacement gate is cheaper than the two gates being replaced.

PiByEight–PiByEightInv: Provided the target and control bits match, these gates (obviously) cancel.

PiByEight–PhaseGate: Provided target and controls match, these may be converted to an ArbitraryPhase of angle $3\pi/4$.

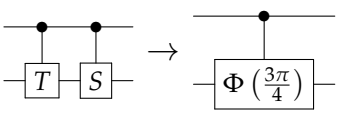


Figure 3.28: Simplifications of PiByEight–PhaseGate, assuming that the ArbitraryPhase is cheaper than the two gates being replaced.

PiByEight–PhaseInv: Given matching targets and controls, these simplify to a PiByEightInv or equivalent.

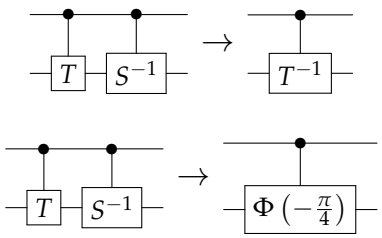


Figure 3.29: Simplifications of PiByEight–PhaseInv.

PiByEight–ZGate: Given matching targets and controls, these simplify to an ArbitraryPhase of angle $-3\pi/4$.

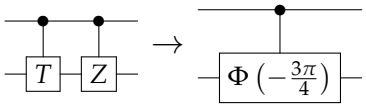


Figure 3.30: Simplifications of PiByEight–ZGate.

PiByEight–ZRotation: If the set of qubits involved in the PiByEight match those for the ZRotation[-28-] then there is the possibility of a simplification. Notice that

$$Z(\theta)T = \begin{pmatrix} e^{i\theta/2} & 0 \\ 0 & e^{-i\theta/2} \end{pmatrix} \begin{pmatrix} 1 & 0 \\ 0 & e^{i\pi/4} \end{pmatrix}$$
$$= e^{i\pi/8} \begin{pmatrix} e^{i(\theta-\pi/4)/2} & 0 \\ 0 & e^{-i(\theta-\pi/4)/2} \end{pmatrix} = e^{i\pi/8} Z(\theta - \pi/4),$$

i.e. $Z(\theta)T$ gives the matrix for a new ZRotation, up to an overall phase. Hence if neither gate has controls, they simplify to a new, single ZRotation gate. However, if the initial gates have controls then, when applied to a basis state, the phase shift affects only those states where the control qubits are set. Hence in this case, an ArbitraryPhase of angle $\pi/8$ must be added across the control bits of the ZRotation.[-29-]

[-28-] Targets and controls need not agree initially — we can always switch the target of the PiByEight for one of the controls in order to get them to match with the ZRotation.

[-29-] Note that in figure 3.31 the PiByEight has 'many' — i.e. more than one — controls, while the ArbitraryPhase has just one control. This means that it is possible that the ArbitraryPhase is cheaper. In contrast, if we were to add an extra control to each gate, then both the PiByEight and the ArbitraryPhase would have 'many' controls. In this situation, if the ArbitraryPhase gate were cheaper then the code would, during construction of the CircuitContext object, determine that the PiByEight with many controls was a redundant gate type and remove it from consideration. Hence this simplification never occurs (at present) if the initial gates have three or more controls.

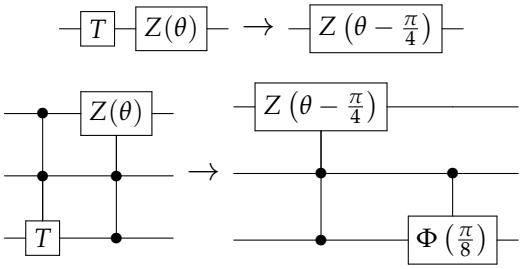


Figure 3.31: Simplifications of PiByEight–ZRotation.

PiByEight–ArbitraryPhase: If targets and control bits agree, these simplify to a new ArbitraryPhase gate.

PiByEightInv–PiByEight: Identical to PiByEight–PiByEightInv. When this situation arises, where the simplification is precisely the same as the case where the gates are reversed, and the reversed case has already been described, the relevant functions are implemented by simply redirecting the call to the previously defined functions.[-30-]

[-30-] As it happens, this redirection and those for a number of the simplifications that follow, is actually the opposite of what the code does. For this case, it is actually the simplification functions for PiByEight–PiByEightInv that redirect to those for PiByEightInv–PiByEight. This confusion has arisen because (due to the use

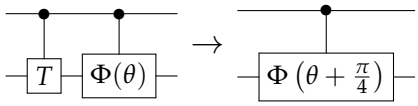


Figure 3.32: Simplifications of PiByEight–ArbitraryPhase.

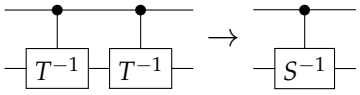


Figure 3.33: Simplifications of PiByEightInv–PiByEightInv.

PiByEightInv–PiByEightInv: If targets and control bits agree, these simplify to a PhaseInv.

PiByEightInv–PhaseGate: As for PiByEight–PhaseInv, but with all gates replaced by their inverses.

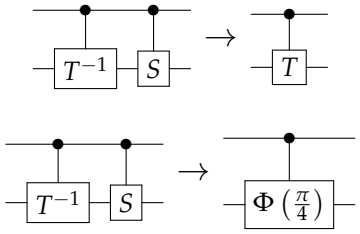


Figure 3.34: Simplifications of PiByEightInv–PhaseGate.

PiByEightInv–PhaseInv: As for PiByEight–PhaseGate, but with all gates replaced by their inverses.

PiByEightInv–ZGate: As for PiByEight–ZGate, but with all gates replaced by their inverses. (The inverse of the ZGate is just the ZGate.)

PiByEightInv–ZRotation: As for PiByEight–ZRotation, except for a change in sign in the change of ZRotation angle and in the additional ArbitraryPhase, if needed.

PiByEightInv–ArbitraryPhase: As for PiByEight–ArbitraryPhase, except for a change in sign in the change of ArbitraryPhase angle.

PhaseGate–PiByEight: Redirects to functions for PiByEight–PhaseGate.

PhaseGate–PiByEightInv: Redirects to functions for PiByEightInv–PhaseGate.

PhaseGate–PhaseGate: Given matching targets and controls, these simplify to a ZGate or equivalent ArbtiraryPhase.

PhaseGate–PhaseInv: Provided targets and controls agree, these gates (obviously) cancel.

PhaseGate–ZGate: Provided the target and control qubits of the two gates agree, these become a PhaseInv or an equivalent ArbitraryPhase[-31-].

PhaseGate–ZRotation: Similar to PiByEight–ZRotation, but with the change to the angle of the ZRotation and the angle of the ArbitraryPhase across the control qubits multiplied by two. This means that the ArbitraryPhase can be replaced by a PiByEight.

[-31-] I guess having PhaseGates available and PhaseInv unavailable or more expensive would be a rather unusual situation.

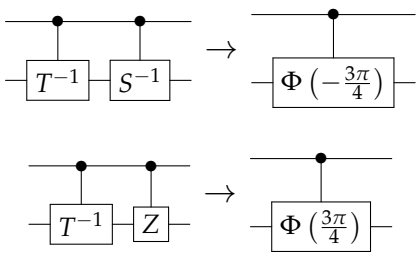


Figure 3.35: Simplifications of PiByEightInv–PhaseInv, assuming that the ArbitraryPhase is cheaper than the two gates being replaced.

Figure 3.36: Simplifications of PiByEightInv–ZGate.

PhaseGate–ArbitraryPhase: If targets and controls bits agree, these simplify to a new ArbitraryPhase gate.

PhaseInv–PiByEight: Redirects to functions for PiByEight–PhaseInv.

PhaseInv–PiByEightInv: Redirects to functions for PiByEightInv–PhaseInv.

PhaseInv–PhaseGate: Redirects to functions for PhaseGate–PhaseInv.

PhaseInv–PhaseInv: Given matching targets and controls, these simplify to a ZGate or equivalent ArbtiraryPhase.

PhaseInv–ZGate: As for PhaseGate–ZGate, but with all gates replaced by their inverses.

PhaseInv–ZRotation: Like PhaseGate–ZRotation, except that the change to the angle of the ZRotation has opposite sign, while any gate added across the control qubits is the inverse of that in the PhaseGate–ZRotation case.

PhaseInv–ArbitraryPhase: As for PhaseGate–ArbitraryPhase, but with the sign of the angle change switched.

XGate–Hadamard: Function `canSimplify` merely redirects to the version for Hadamard–XGate. However, function `simplification` cannot do this, as here we get the sequence Hadamard–ZGate, not ZGate–Hadamard.

XGate–XGate: Provided the targets and controls of the two gates match, these cancel.

XGate–YGate: If both gates have the same target and no controls, then they simplify to a single ZGate or equivalent — the overall phase shift can be ignored. If the gates have matching targets and the same, non-empty set of controls, then a PhaseInv or equivalent must be added across the control bits.

$$YX = \begin{pmatrix} 0 & -i \\ i & 0 \end{pmatrix} \begin{pmatrix} 0 & 1 \\ 1 & 0 \end{pmatrix} = \begin{pmatrix} -i & 0 \\ 0 & i \end{pmatrix} = -iZ.$$

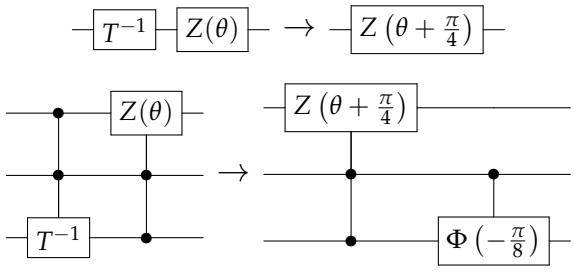


Figure 3.37: Simplifications of PiByEightInv–ZRotation.

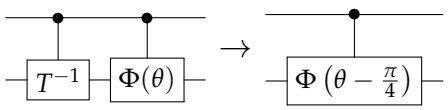


Figure 3.38: Simplifications of PiByEightInv–ArbitraryPhase.

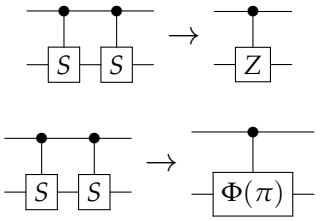


Figure 3.39: Simplifications of PhaseGate–PhaseGate, assuming that the replacement gate is cheaper than the two gates being replaced.

XGate–ZGate: If both gates share the same qubit and have no controls then they simplify to a single YGate. If the gates have controls, then for a simplification to occur, the set of qubits involved in the XGate must match the set involved in the ZGate. Note that targets may initially *disagree* — we can always swap the target of the ZGate with a control to make them match. In this case, the result of the simplification is a YGate and a PhaseGate (or equivalent) across its control qubits.

$$ZX = \begin{pmatrix} 1 & 0 \\ 0 & -1 \end{pmatrix} \begin{pmatrix} 0 & 1 \\ 1 & 0 \end{pmatrix} = \begin{pmatrix} 0 & 1 \\ -1 & 0 \end{pmatrix} = iY.$$

XGate–XRotation: Note that the matrix, X, for the XGate satisfies $X = -iX(\pi) = iX(-\pi)$. So

$$\begin{aligned} X(\theta)X &= -iX(\theta)X(\pi) = -iX(\theta + \pi) \\ &= iX(\theta)X(-\pi) = iX(\theta + \pi). \end{aligned}$$

If both gates have the same target and no controls, they simplify to a single XRotation, up to an overall phase which we ignore. Otherwise, if both target qubits and control qubits agree, the two gates simplify to an XRotation (with the same target and controls) and either a PhaseGate or a PhaseInv across the control qubits. Naturally, if target qubits or control qubits disagree, there is no simplification. The angle of the new XRotation depends on whether we select a PhaseGate or a PhaseInv on the controls. It is possible that neighbouring gates may make one choice better than the other, resulting in subsequent simplifications. However, at present, the code expects only one simplification to be

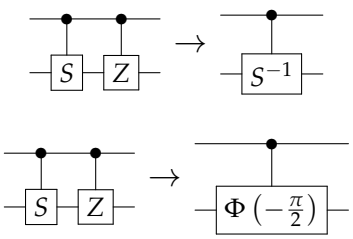


Figure 3.40: Simplifications of PhaseGate–ZGate.

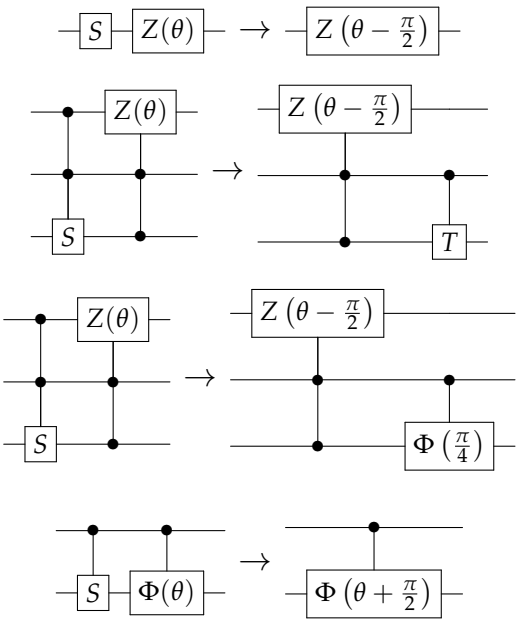


Figure 3.41: Simplifications of PhaseGate–ZRotation.

Figure 3.42: Simplifications of PhaseGate–ArbitraryPhase.

presented. In the (unusual) situation where the PhaseGate is cheaper than the PhaseInv (or vice versa), or if only one is available, then the cheaper (or available) one is chosen. Otherwise the simplification involving creating the PhaseGate is selected. Of course, if neither is available, an ArbitraryPhase may be used instead, in which case one equivalent to a PhaseGate is chosen.

YGate–XGate: As for XGate–YGate, except that the gate across the control bits becomes a PhaseGate rather than a PhaseInv.

$$XY = \begin{pmatrix} 0 & 1 \\ 1 & 0 \end{pmatrix} \begin{pmatrix} 0 & -i \\ i & 0 \end{pmatrix} = \begin{pmatrix} i & 0 \\ 0 & -i \end{pmatrix} = iZ$$

YGate–YGate: Provided the target and controls bits of the two gates agree, these cancel.

YGate–ZGate: If both gates share the same qubit and have no controls then they simplify to a single XGate. If the gates have controls, then for a simplification to occur, the set of gates involved in the YGate must match the set involved in the ZGate. The targets may initially *disagree*, since we can always swap the target of the ZGate with a control to make them match. In this case, the result of the simplification is a XGate and a PhaseInv (or

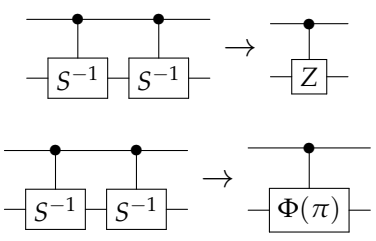
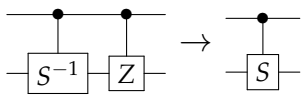


Figure 3.43: Simplifications of PhaseInv–PhaseInv, assuming that the replacement gate is cheaper than the two gates being replaced.



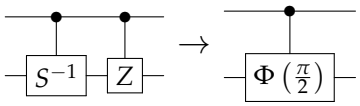
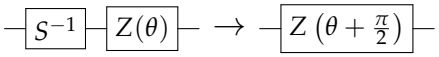
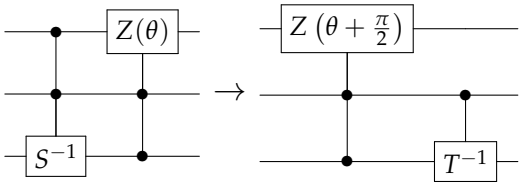


Figure 3.44: Simplifications of PhaseInv–ZGate.





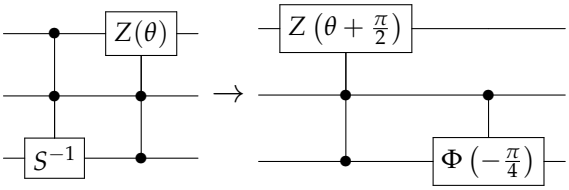


Figure 3.45: Simplifications of PhaseInv–ZRotation.

equivalent) across its control qubits.

$$ZY = \begin{pmatrix} 1 & 0 \\ 0 & -1 \end{pmatrix} \begin{pmatrix} 0 & -i \\ i & 0 \end{pmatrix} = \begin{pmatrix} 0 & -i \\ -i & 0 \end{pmatrix} = -iX.$$

YGate–YRotation: Essentially the same as the XGate–XRotation simplifications, but with X replaced by Y.

ZGate–Hadamard: Function `canSimplify` merely redirects to the version for Hadamard–ZGate. However, function `simplification` cannot do this, as here we get the sequence Hadamard–XGate, not XGate–Hadamard.

ZGate–PiByEight: Functions redirect to versions for PiByEight–ZGate.

ZGate–PiByEightInv: Functions redirect to versions for PiByEightInv–ZGate.

ZGate–PhaseGate: Functions redirect to versions for PhaseGate–ZGate.

ZGate–PhaseInv: Functions redirect to versions for PhaseInv–ZGate.

ZGate–XGate: As for XGate–ZGate, except that the gate across the control bits becomes a PhaseInv rather than a PhaseGate.

$$XZ = \begin{pmatrix} 0 & 1 \\ 1 & 0 \end{pmatrix} \begin{pmatrix} 1 & 0 \\ 0 & -1 \end{pmatrix} = \begin{pmatrix} 0 & -1 \\ 1 & 0 \end{pmatrix} = -iY.$$

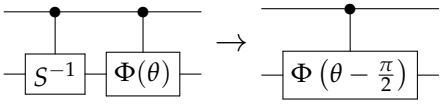


Figure 3.46: Simplifications of PhaseInv–ArbitraryPhase.

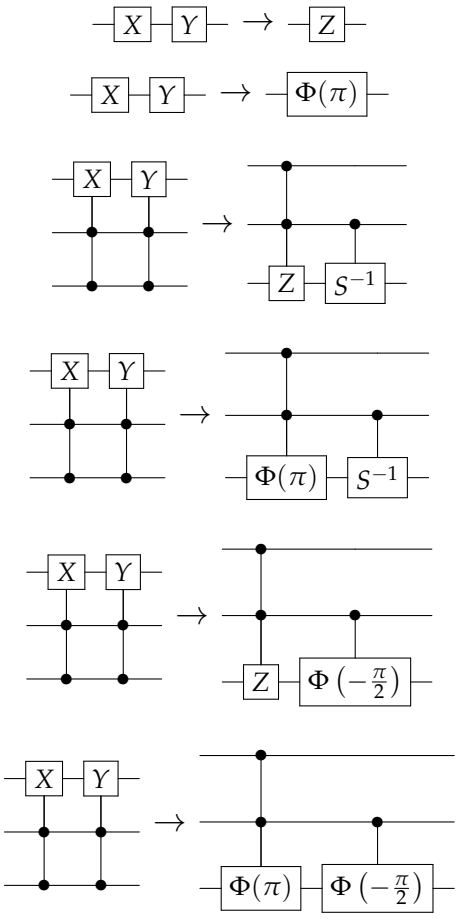


Figure 3.47: Simplifications of XGate–YGate.

ZGate–YGate: As for YGate–ZGate, except that the gate across the control bits becomes a PhaseGate rather than a PhaseInv.

$$YZ = \begin{pmatrix} 0 & -i \\ i & 0 \end{pmatrix} \begin{pmatrix} 1 & 0 \\ 0 & -1 \end{pmatrix} = \begin{pmatrix} 0 & i \\ i & 0 \end{pmatrix} = iX.$$

ZGate–ZGate: Provided the target and control bits of the two gates agree, these cancel.

ZGate–XRotation: If the set of bits involved in the ZGate matches the set of control bits for the XRotation then the ZGate can be eliminated if matrix for the XRotation is multiplied by -1. This is achieved by simply adding 2π to the angle.

ZGate–YRotation: The same as for ZGate–XRotation, with X replaced by Y (of course).

ZGate–ZRotation: These two gates may simplify in two different ways. If both gates operate on the same set of qubits, they simplify as in the XGate–XRotation case, with minor differences due to the fact that ZGate is a PhaseTypeGate. Alternatively, if the ZGate operates on the control bits of the ZRotation, then the simplification is the same as for ZGate–XRotation, with X replaced by Z of course.

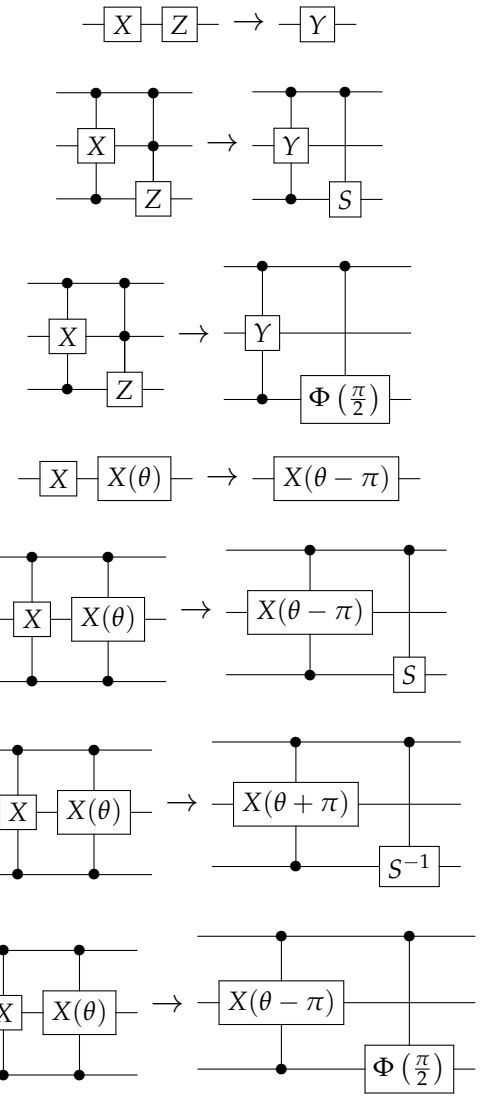


Figure 3.48: Simplifications of XGate–ZGate.

Figure 3.49: Simplifications of XGate–XRotation.

ZGate–ArbitraryPhase: If the two gates share the same targets and controls then the ZGate can be eliminated if the angle for the ArbitraryPhase is increased by π.

XRotation–Hadamard: Function `canSimplify` merely redirects to the version for Hadamard–XRotation. However, function `simplification` cannot do this, as here we get the sequence Hadamard–ZRotation, not ZRotation–Hadamard.

XRotation–XGate: Functions redirect to versions for XGate–XRotation.

XRotation–ZGate: Functions redirect to versions for ZGate–XRotation.

XRotation–XRotation: Provided both target and control qubits for

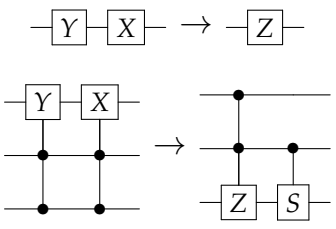


Figure 3.50: Simplifications of YGate–XGate. As with previous examples, alternatives exist where the ZGate or the PhaseGate is replaced with an equivalent ArbitraryPhase gate.

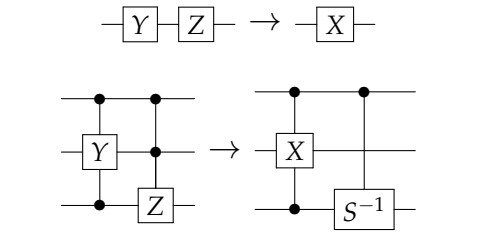


Figure 3.51: Simplifications of YGate–ZGate, omitting the case where the final PhaseInv is replaced by an equivalent ArbitraryPhase.

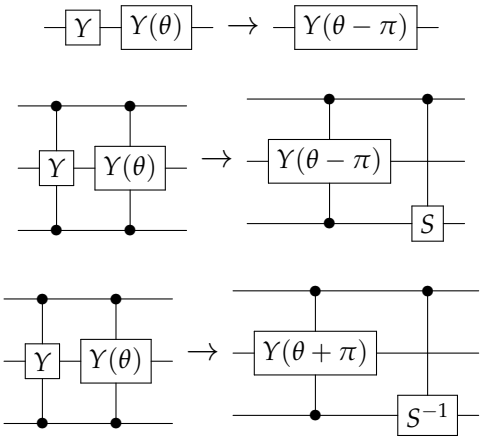


Figure 3.52: Simplifications of YGate–YRotation, omitting the case where an ArbitraryPhase gate is used as a stand-in for the PhaseGate.

the two gates agree, these simplify to a single XRotation — just add the angles.[-32-]

YRotation–YGate: Functions redirect to versions for YGate–YRotation.

YRotation–ZGate: Functions redirect to versions for ZGate–YRotation.

YRotation–YRotation: As for XRotation–XRotation, these simplify to a single rotation if the targets and controls agree.

ZRotation–Hadamard: Function `canSimplify` merely redirects to the version for Hadamard–ZRotation. However, function `simplification` cannot do this, as here we get the sequence Hadamard–XRotation, not XRotation–Hadamard.

ZRotation–PiByEight: Functions redirect to those for PiByEight–ZRotation.

ZRotation–PiByEightInv: Functions redirect to those for PiByEightInv–ZRotation.

ZRotation–PhaseGate: Functions redirect to those for PhaseGate–ZRotation.

ZRotation–PhaseInv: Functions redirect to those for PhaseInv–ZRotation.

ZRotation–ZGate: Functions redirect to those for ZGate–ZRotation.

[-32-] Note that at all times, the stored angle parameters are kept within a range that depends on gate type and the number of controls. With no controls, this range is $[-\pi, \pi]$. The same range is used for ArbitraryPhase gates with controls. For the XRotation, YRotation and ZRotation gates, this range becomes $[-2\pi, 2\pi]$ when the gates have controls. If a simplification operation takes the angle outside of the permitted range, then it is adjusted by adding some multiple of 2π or 4π as appropriate.

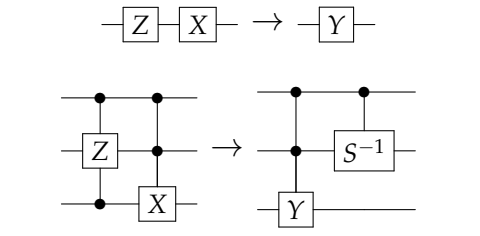


Figure 3.53: Simplifications of ZGate–XGate, ignoring possible use of equivalent ArbitraryPhase gates.

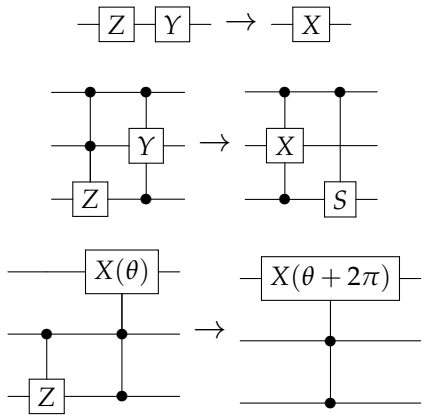


Figure 3.54: Simplifications of ZGate–YGate, ignoring possible use of equivalent ArbitraryPhase gates.

Figure 3.55: Simplifications of ZGate–XRotation.

ZRotation–ZRotation: As for XRotation–XRotation, these simplify to a single rotation if the targets and controls agree.

ZRotation–ArbitraryPhase: These gates will simplify provided the set of qubits involved in the ZRotation matches the set involved in the ArbitraryPhase. Note that the target bits need not agree at first, since the target bit of an ArbitraryPhase gate can always be switched with one of its controls. If neither gate has controls, they simplify to a single ZRotation or a single ArbitraryPhase. If both have controls then an additional ArbitraryPhase is required on the control bits.[-33-]

The choice for the main replacement gate made depends on the relative cost of the gate types, with the cheaper gate selected. Note that when a ZRotation is selected, its target bit must match that of the original ZRotation. The auxiliary ArbitraryPhase gate must be across the control bits of the original ZRotation in either case. The angle for the main replacement gate is the difference between the angles of the original gates, with the ZRotation angle being subtracted from the ArbitraryPhase angle if the replacement is an ArbitraryPhase, and vice versa if the replacement is a ZRotation.

Note that the number of useful degrees of freedom is not changed by this simplification. This contrasts with the other simplifications, described above, where an ArbitraryPhase gate is introduced on the control bits of one of the original gates. An illustration of circuit *structure* equivalence, and hence the equal number of useful degrees of freedom, is shown in figure 3.61.

[-33-] If each gate has only *one* control, then a ZRotation could be used as an alternative to this ArbitraryPhase. Indeed this possibility exists wherever we have added an ArbitraryPhase gate in this manner. At present, the code *does not* implement this possibility. (If the original gates have more than one control, then the notion of using a ZRotation as an alternative to the ArbitraryPhase across the control bits becomes more complicated.)

ArbitraryPhase–PiByEight: Functions redirect to those for PiByEight–ArbitraryPhase.

ArbitraryPhase–PiByEightInv: Functions redirect to those for PiByEightInv–

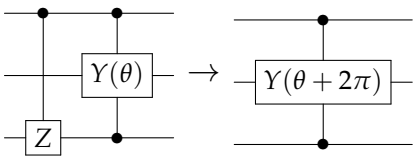


Figure 3.56: Simplifications of ZGate–YRotation.

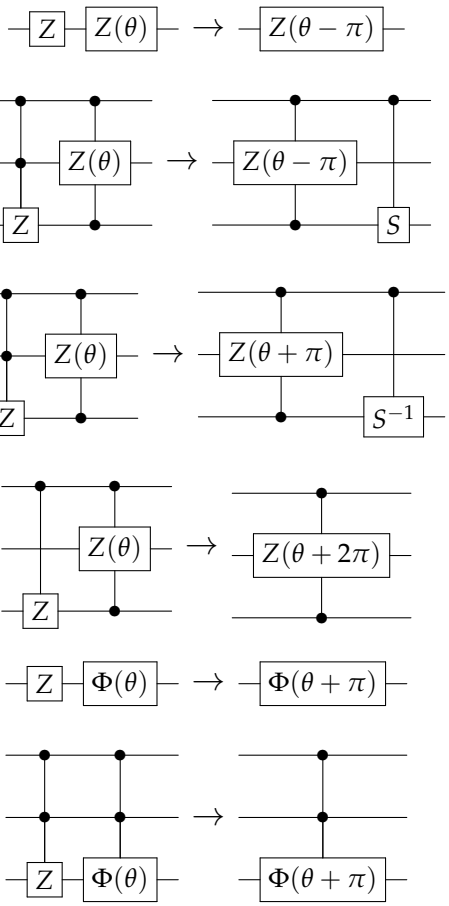


Figure 3.57: Simplifications of ZGate–ZRotation, ignoring possible use of equivalent ArbitraryPhase gates.

Figure 3.58: Simplifications of ZGate–ArbitraryPhase.

ArbitraryPhase.

ArbitraryPhase–PhaseGate: Functions redirect to those for PhaseGate–ArbitraryPhase.

ArbitraryPhase–PhaseInv: Functions redirect to those for PhaseInv–ArbitraryPhase.

ArbitraryPhase–ZGate: Functions redirect to those for ZGate–ArbitraryPhase.

ArbitraryPhase–ZRotation: Functions redirect to those for ZRotation–ArbitraryPhase.

ArbitraryPhase–ArbitraryPhase: As for XRotation–XRotation, we simply add the angles if targets and controls agree.

SwapGate–SwapGate: Two identical swap gates, i.e. gates with matching qubits, cancel.

Oracle–Oracle: Two adjacent Oracle gates cancel.

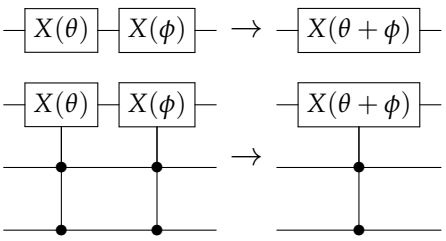


Figure 3.59: Simplifications of XRotation–XRotation.

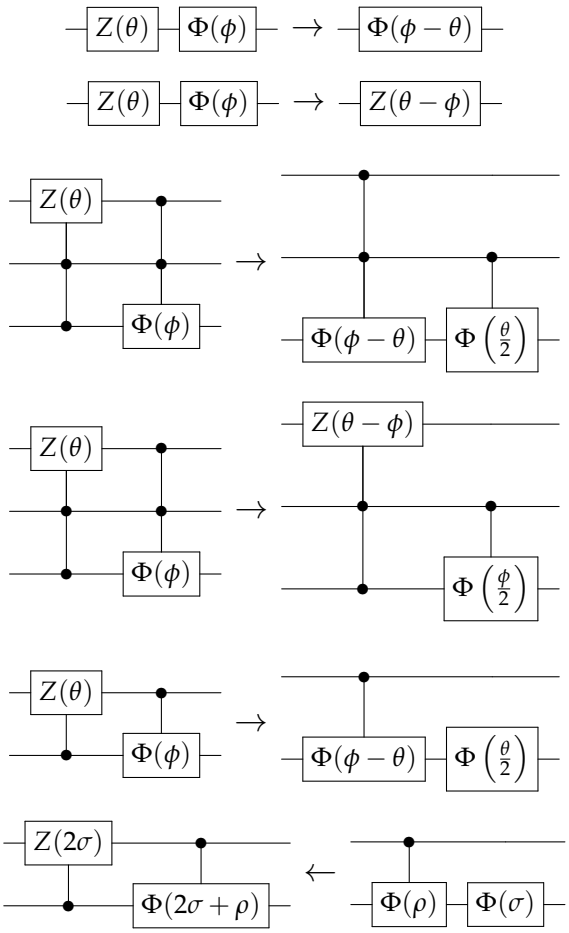


Figure 3.60: Simplifications of ZRotation–ArbitraryPhase.

Figure 3.61: A demonstration of the equivalence of the original circuit *structure* and that for the simplified circuit, in the case of ZRotation–ArbitraryPhase simplification.

3.5.1 *Simplifications at the beginning of the circuit*

In addition to the pairwise simplification of gates described thus far in this section, we may also be able to eliminate individual gates that occur at the beginning of the circuit, if we exploit our knowledge of the types of input the circuit receives.[-34-]

The circuit context object is initialized with information regarding the inputs for each qubit. This information indicates whether an input qubit is always zero, always one or 'varies'[-35-]. For example, for the Fourier problem[-36-], the input for each qubit varies over the input states. In contrast, for the Grover problem, each input qubit is always zero. The Gate class contains a (pure) virtual function, `bool cancelsAtStart()`, which is overridden in the SingleTargetGate, DiagonalGate, PhaseTypeGate, SwapGate and Oracle classes. A description of these overrides follows.

SingleTargetGate: Returns `true` if and only if any of the control qubits is always zero.

DiagonalGate: Returns `true` if any of the control qubits is always zero. Also returns `true` if each of the involved qubits have fixed inputs, i.e. has an input that is either always zero or always one, as this merely multiplies each possible input by the same irrelevant overall phase. Otherwise returns `false`.

[-34-] At present, the code that performs this sort of simplification is fairly crude and can undoubtedly be improved. Some such improvements are described at the end of this section.

[-35-] 'Varies' covers a number of possibilities. Typically it will simply mean that the qubit may be zero or may be one, depending on the simulation being performed. This is the case for the Fourier problem. However, it also covers the possibility that the input for the qubit is actually fixed, but as a superposition of zero and one. It also covers cases where the qubit in question is entangled others, and so cannot be assigned a (pure) state. (Of course, it would be unusual to consider such states as *input* to our circuits).

[-36-] The Fourier and Grover problems will be described in more detail in section **??**.

PhaseTypeGate: Returns `true` for all those cases where the function for a DiagonalGate returns `true` and also when the target qubit is always zero, in which case the gate does nothing.

SwapGate: Returns `true` whenever both qubits are always zero and whenever both qubits are always one. Otherwise returns `false`.

Oracle: Returns `true` whenever all the inputs to the circuit are fixed, i.e. whenever each qubit is either always zero or always one.[-37-]

At present, these routines only eliminate a gate if the gate can be swapped to the beginning of the circuit. This misses possible simplifications. For example, if the input to the circuit is fixed[-38-] and the circuit starts with an XGate on one qubit followed by an Oracle, then the Oracle is redundant, since it always receives the same fixed state[-39-] as input. This will not be discovered by the code, since the Oracle cannot be swapped past the XGate to the beginning of the circuit. Future algorithmic improvements may involve tracing the 'fixedness' of qubits through the circuit, so that we can easily detect such cases.

3.6 Simplification control

Thus far, we have described the routines that determine whether a pair of gates may be swapped or simplified, and the results of taking that action. What remains is to describe how higher level procedures control the application of these routines so as to result in a circuit that is a simple as (reasonably) possible. This aspect of our algorithm has been subject to much change during the development of the code: we are currently at version five. Furthermore, development of the higher level simplification routines has affected the design of the lower level methods, influencing the splitting of circuit adjustments into swaps and simplifications and the inclusion of more complicated operations, i.e. H-swaps, amongst the set of possible swaps.

Recent versions of this code have focussed on the use of simple descent methods[-40-]. At one point, we considered a two-phase method. In the first phase, a steepest descent search was performed. Each pair of adjacent gates was examined for a simplification and the simplification that produced the greatest decrease in cost was performed. This was repeated until no further improvements could be found. Then, in the second phase, a full tree search was performed, making those adjustments to the circuit that neither increased nor decreased the cost. If, at any stage, an improving move was discovered, the search would return to phase one.

This approach fails due to there being far too many zero-improvement adjustments that can be made. This can result in an exponential increase in the number of solutions that must be visited in phase two[-41-]. If one also attempts to add entries to the cache for all of

[-37-] We assume that the evaluation of the circuit involves simulating for each possible Oracle setting and summing over the set of error values produced. We further assume that the the error value for each setting does not change if the output is multiplied by an overall phase. If all input qubits are fixed then the Oracle either does nothing, or multiplies the state by an overall phase of -1 when the Oracle's marked state matches the input. Neither of these will result to a change in the error.

[-38-] Note that saying that the input to a circuit is 'fixed' is not the same as saying that the circuit always received the same state as input. By 'fixed', we mean each qubit is either always zero or always one, i.e. the circuit always receives the same *basis* state as input. If the circuit always received the same *non*-basis state as input, then we do not consider the input to be 'fixed', as some qubits will not be 'always zero' or 'always one'.

[-39-] i.e. basis state

[-40-] Sometimes known as hill-climbing algorithms, though our case we want to go downhill.

[-41-] Consider n non-interacting simplifications, i.e. a set of simplifications such that, if one is taken, the others are still available. Then there would be 2^n alternative circuits produced from just these simplifications.

these solutions (which one would have to do to make the cache worthwhile), then this would also exponentially increase the storage requirements. The failure of this approach is what led to the development of the canonical representation of circuits and the separation of swaps from simplifications. Moreover, it also led to the zero-improvement H-swaps and R-swaps being included amongst the swap operations used when reducing a circuit to canonical form.

The current approach uses only phase one: the steepest descent algorithm. However, rather than considering just those pairs of gates that are adjacent in the circuit representation, all pairs of gates that can be brought together, via application of the swap operations, are considered. There is a concern that the simplification may end at a point on a flat area of the search space, where zero-improvement moves could conceivably result in the discovery of further changes that can reduce circuit cost. Of course, a similar story can be told whenever the search ends in a true local minimum: perhaps making a change that temporarily increases circuit cost will lead us to later, bigger improvements. In any case, we need to limit the amount of effort spent in circuit simplification in some way[-42-]. A second concern is that there may be many equivalent, equal cost circuits, that could be easily found and cached along with the simplified circuit.

Both of these concerns are alleviated somewhat by the fact that the most likely candidate operations for producing zero change in circuit cost, i.e. the swaps, are used to reduce circuits to canonical form. Hence while there may be many equivalent circuits of equal cost, many will reduce to the same canonical circuit. Furthermore, while simplistic methods of assigning costs to gates may leave the distinct possibility of zero-improvement 'simplifications', a careful examination of those simplifications listed above reveals that, with more realistic gate costs, where the costs are more greatly influenced by the number of qubits involved than the type of the underlying gate[-43-], a zero-improvement simplification is highly unlikely[-44-].

Each step of the steepest descent algorithm takes the usual form: examine possible moves; perform the best move. However, in addition, each simplification step is followed by the reduction of the circuit back to canonical form. This enables the algorithm to check, after each step, whether the circuit has been visited before by the algorithm. If it has, then the rest of the process of circuit structure evaluation, i.e. the rest of the simplification, the evaluation of the circuit, the application of numerical optimization, etc., may be circumvented, saving computing time.

We continue by providing a number of blocks of pseudo-code that describe the algorithm in more detail. The first of these, in figure 3.62, is a routine that determines how far to the left a gate may be moved using just swaps and H-swaps[-45-], while the second, in figure 3.63 actually performs such a move. In both cases, similar

[-42-] Whether it would be worthwhile applying a more sophisticated algorithm such as simulated annealing or tabu search is a matter for some later date

[-43-] For example, a controlled-Not being more expensive than any single qubit gate, due to two-qubit gates being more difficult to implement.

[-44-] I am not really convinced by this argument. It is fine if gate cost mimics the *difficulty* of implementing the gate. However, we may decide to use other notions of circuit complexity that capture, for example, the computational complexity, or the likely comprehensibility of a circuit for a human viewer. In these cases, zero-cost 'simplifications' may be more likely, particularly if we use something like circuit depth as our objective, rather than number of gates.

[-45-] It is assumed that the circuit is in canonical form, so all swap gates will be at the end (right) of the circuit, or if there are Oracles present, as far to the right as possible. In either case, swap-swaps do not need to be considered.

In the actual code, there is an additional section that handles R-swaps in the same way as H-swaps are handles here. This has been omitted from figure 3.62 for brevity.

functions are available for moving a gate to the right in a similar manner.

```
1: procedure CIRCUIT::LEFTMOST_SHIFT(int pos, Gate movedGate[0 to length( ) − 1])
2:     pulledPos ← pos
3:     movingGate ← gates[pos]
4:     movedGate[pos] ← movingGate
5:     while pulledPos > 0 do
6:         if gates[pulledPos − 1].canSimplySwap(movingGate) then
7:             pulledPos ← pulledPos − 1
8:             movedGate[pulledPos] ← movingGate
9:         else if gates[pulledPos − 1].canHSwap(movingGate) then
10:            possibleChange ← gates[pulledPos − 1].leftHMoverChange(movingGate)
11:            if possibleChange then          ▷ Actual code uses pointers. No change ⇒ null pointer.
12:                movingGate ← possibleChange
13:            end if
14:            pulledPos ← pulledPos − 1
15:            movedGate[pulledPos] ← movingGate
16:        else
17:            return pulledPos
18:        end if
19:    end while
20:    return pulledPos
21: end procedure
```

Figure 3.62: Pseudocode for determining how far the gate at position *pos* can be moved leftwards, using only swap operations. Since the moving gate may change during such a move, for example when an XGate is moved past a Hadamard, the identity of the moved gate is recorded, at each position, in the 'movedGate' array.

The third piece of pseudocode, in figure 3.64 describes the search for the best simplification. Once the best simplification is found, it is applied by simply moving the gates at `leftPos` and `rightPos` using functions `swap_gate_right` and `swap_gate_left` to new positions `meetPos` and `meetPos + 1` and then replacing them with the simplified gate sequence.

```
1: procedure CIRCUIT::SWAP_GATE_LEFT(int moverPos, int finalPos)
2:     movingGate ← gates[moverPos]
3:     for pos ← moverPos down to finalPos + 1 do
4:         if gates[pos − 1].canSimplySwap(movingGate) then
5:             gates[pos] ← gates[pos − 1]
6:         else                                   ▷ Assumes a H-Swap is available
7:             newJumpedGate ← gates[pos − 1].rightHMoverChange(movingGate)
8:             if newJumpedGate then      ▷ Actual code uses pointers. No change ⇒ null pointer.
9:                 gates[pos] ← newJumpedGate
10:            else
11:                newMovingGate ← gates[pos − 1].leftHMoverChange(movingGate)
12:                if newMovingGate then                  ▷ No change ⇒ null pointer
13:                    movingGate ← newMovingGate
14:                end if
15:                gates[pos] ← gates[pos − 1]
16:            end if
17:        end if
18:    end for
19:    gates[finalPos] ← movingGate
20: end procedure
```

Figure 3.63: Pseudocode for moving a gate from position *moverPos* leftwards to position *finalPos* by swapping with each intermediate gate. This function assumes that this is possible, i.e. that a suitable swap exists at each position. Moreover, it is assumed that the circuit starts in canonical form, meaning that swap swaps are not required — all swap gates are safely out of the way.

```
1: procedure CIRCUIT::BEST_REPLACEMENT( )
2:     leftmost[0 to length( ) − 1] ← empty          ▷ Initialize useful arrays
3:     rightmost[0 to length( ) − 1] ← empty
4:     movedGate[0 to length( ) − 1, 0 to length( ) − 1] ← empty
5:     for pos ← 0 to length( ) − 1 do
6:         leftmost[pos] ← leftmost_shift(pos, movedGate[pos])   ▷ (Passes one row of movedGate)
7:         rightmost[pos] ← rightmost_shift(pos, movedGate[pos])
8:     end for
9:
10:    bestImprovement ← 0                  ▷ Search through possible simplifications
11:    bestReplacement ← empty
12:    for each pair of positions, (leftPos, rightPos) do
13:        if rightmost[leftPos] ≥ leftmost[rightPos] then                ▷ (Gates can meet)
14:            meetPos ← min(rightmost[leftPos], rightPos − 1)
15:            newLeftGate ← movedGate[leftPos, meetPos]                  ▷ (Gates may have changed)
16:            newRightGate ← movedGate[rightPos, meetPos + 1]
17:            improvement ← newLeftGate.canSimplify(newRightGate)
18:            if improvement > bestImprovement then                      ▷ (This simplification is better)
19:                bestImprovement ← improvement
20:                bestReplacement ← (leftPos, rightPos, meetPos)
21:            end if
22:        end if
23:    end for
24:    return bestReplacement
25: end procedure
```

Figure 3.64: Simplified pseudocode for discovering the best available simplification. In determining whether one simplification is better than another, the real code considers not only circuit cost, but whether the number of useful degrees of freedom increases.

4 Numerical optimization

Genetic algorithms may be a suitable approach for combinatorial optimization problems, e.g. circuit discovery when gates are without numerical parameters. However, if a problem involves numeric - in particular, real-valued - decision variables, numerical optimization techniques are often more appropriate. While genetic algorithms, given enough computation time, can be used to solve such problems, algorithms that make greater use of the mathematical structure, e.g. objective gradient information, are typically more effective, requiring fewer solution evaluations. In our case, this implies a reduced requirement for expensive quantum circuit simulation.

When we use a gate set that includes parameterized gates, with numerical parameters such as rotation angle, the optimization problem contains both combinatorial and numeric components. As we wish to continue to perform multi-objective optimization[-1-], it is reasonable to continue to use a population based approach such as a GA for part of the optimization algorithm. However, different methods are more appropriate for the optimization of the gate parameters. This therefore suggests using a hybrid approach, with a genetic algorithm optimizing the circuit structure and numeric algorithms for the optimization of gate angles. We have illustrated this approach in figure 1.4, with the numerical optimization being performed in line 9.

[-1-] Though perhaps with fewer objectives.

4.1 Gradient calculation

The ability to calculate the gradient of the objective function at little extra cost provides additional motivation for the application of numerical optimization algorithms. For our problem, the additional cost of calculating the gradient is not insignificant, but neither is it excessively burdensome. We will show that calculating the cost and gradient for a circuit can be achieved at two to three times the computational cost of calculating the cost alone.

Each application of a gate to an input state may be represented as the multiplication of a matrix, representing the gate, and a vector, representing a state. Each problem that we have considered thus far uses an overall error objective that depends solely on a set of dot products, or 'overlaps', of the output of the circuit, given particular circuit settings and input, and the desired target output.

So given gates represented by the matrices A–G, one such overlap would be $t^\dagger GFEDCBAs$, where s is the input state and t is the target state. If we wish to calculate the derivative of the overall circuit error with respect, for example, to the parameter[-2-] of gate E, we need to get the derivative of the various overlaps. So we need to calculate $t^\dagger GFE'DCBAs$, where E' is the derivative of the matrix E with respect to the gate parameter.

[-2-] Gates with multiple parameters are handled in a similar manner.

At this point, note that if we have already simulated the circuit to obtain the overall error, some of this calculation has already been performed. If we stored the intermediate states during this simulation, then when calculating the derivative we would be able to simply look up the state $i = DCBAs$, eliminating four unnecessary matrix multiplications. If we also, in advance, simulated the circuit in reverse, applying the inverse of gate G to the target state, then the inverse of F to that state, and so on, then we could also simply look up the state $j = F^\dagger G^\dagger t$. The calculation of the derivative, with respect to gate E's angle parameter, would then involve just the product $j^\dagger E' i$.

To calculate all the components of the gradient of the overlap, we need to calculate the derivative with respect to all gate parameters. So counting up the number of times we multiply a state vector by a matrix[-3-], we get ℓ multiplications for the forward simulation of the circuit, ℓ multiplications for the reverse simulation of the circuit and p multiplications to get the derivatives, where ℓ is the number of gates and p is the number of gate parameters. So we have $2\ell + p$ multiplications in total and, assuming that we are not using SU2Gates, we know that $p < \ell$. So the cost of the calculation of the overlap and gradient is no more than 3 times the cost of calculating just the overlap.

[-3-] Ignoring the final dot product of the states.

To complete the calculation of the gradient of the overall error of a circuit, one must calculate overlaps for a set of simulations. In the case of the Fourier problem, each simulation merely uses a different basis state as input, while in the case of the Grover problem, each simulation uses the same input but has the Oracles set to mark a different basis state. The way in which the overlaps combine to produce the overall error is problem dependent, and hence the last steps in calculating the gradient of the overall circuit error are also problem dependent.

Finally, note that applying the 'derivative of a gate', represented in our example by E', to a state, is not directly supported by QIClib. Provided the gate in question has no controls, then one can use QIClibs function `apply_ctrl()`, using the derivative of the 2×2 provided by the Gate object. However, if the gate has controls, then `apply_ctrl()` cannot be used. Given an input state where a control is unset, i.e. has value $|0\rangle$, `apply_ctrl()` passes the input along unchanged as the output. However, when calculating the gradient, the output when a control gate is unset should be zero[-4-]. We have therefore extended QIClib by supplying a modified version of `apply_ctrl()` for use when calculating gradients.

[-4-] Not the zero state, but the zero *vector*.

4.2 Numerical optimization algorithm

We use the algorithm L-BFGS [-5-] to perform the numerical optimization of the gate parameters. All gate parameters are optimized simultaneously. This quasi-Newton method only requires the gradient[-6-], estimating the (inverse) Hessian at each step of the algorithm and using this information to decide on the best direction to search in at each iteration. Each iteration of the overall L-BFGS algorithm includes the application of a line search algorithm, searching along the single line in the search space in the direction selected. It is a 'limited memory' version of the BFGS algorithm[-7-].

After experimenting with both ALGLIB and OptimLib, we have settled[-8-] on an implementation called L-BFGS++. The choice of implementation depended on three factors: the reliability of the implementation, the efficiency and my ability to understand the code. This last factor is important as it is necessary to be able to adjust the code, particularly with regards to termination criteria, in order for it to mesh successfully (and efficiently) with the rest of the algorithm. ALGLIB and OptimLib each failed on at least on of these criteria[-9-]. While the efficiency of L-BFGS++ could be better, it is written in clear, fairly modern and easily modifiable C++.

Regarding the efficiency of L-BFGS++, while the top level of the implementation is fine, the line search procedures available seem rather basic. We have selected a 'bracketing' approach for the lines search, using the `BACKTRACKING_STRONG_WOLFE` option[-10-] to determine when the line search finishes, as this combination appears to give the most reliable results. Note that that this uses interval bisection to search the line. A preferable approach is to use quadratic or cubic regression to choose the next point on the line to evaluate, particularly with our expensive evaluation function.

A fourth implementation of L-BFGS, in the Ceres Solver library, looks promising and may be able to boost the speed of our algorithm, but as yet we have not attempted incorporating it.

[-5-] Jorge Nocedal and Stephen J. Wright. *Numerical Optimization*. Springer, second edition, 2006

[-6-] In addition to the objective function, of course.

[-7-] The BFGS algorithm would probably have also been suitable, as in our case memory is not a limited factor. Indeed, we run L-BFGS with a large value for the parameter m, setting it to 40, meaning that many of our runs of L-BFGS will be the same as those that would be obtained for BFGS. However, it was easier to find suitable implementations of L-BFGS.

[-8-] For now. See 'Ceres' mentioned below.

[-9-] ALGLIB seems efficient, but is coded very much in C, rather than C++, including extensive use of `long_jmp` — very much not my style! I also managed to get it to crash a few times.

[-10-] L-BFGS++ provides a number of options for halting the line search, some of which seem to be incompatible with the overarching approach of L-BFGS. It also provides headers for 'backtracking' and 'bracketing' approaches to performing the line search, in addition to a 'Nocedal and Wright' option — it is a simple matter of changing a template parameter to select one of these three approaches. Some experimentation was required to find the combination that worked reliably enough to run the thousands of times required by our algorithm without crashing, while having reasonable performance.

4.3 Parameter initialization

During early development of algorithm 1, optimization of gate parameters was left exclusively to the numerical optimization routines, with genetic operators leaving gate parameters untouched. The one exception was when the genetic operator inserted a new gate, in which case the gate's parameters were set entirely at random. This has the result that child solutions, prior to numerical optimization, have gate parameters set at the values that worked well for the parents. This could potentially result in an increased tendency for the algorithm to get stuck at local minima.

We therefore provided an option to completely randomize gate parameters before numerical optimization is applied. However, for our early experiments, this tended to result in a significant increase in run time, with at best limited improvement in the quality of the

results. Leaving gate parameters untouched prior to numerical optimization typically reduces the time required by the numerical algorithms, as might be expected.

To reduce any potential tendency for the algorithm to remain in local minima, without producing an undesirable increase in runtime, we therefore reintroduced the 'mutate gate' genetic operator[-11-]. This is applied probabilistically, rather than to every solution, and randomizes a single gate parameter.

[-11-] See section 2.4.

4.4 *Numerical optimization control*

Hybridizing the GA with numerical optimization routines raises questions regarding how the numerical algorithms are controlled. What is the best way to exploit the powers of numerical optimization within a genetic algorithm? It should be clear that simply letting numerical optimization run 'to completion' on every solution generated by the GA is unlikely to be the most effective approach. What is meant by 'to completion' anyway? Numerical algorithms such as L-BFGS typically run until either the gradient is zero, to some precision, or until improvement has stopped (again, to some precision). What precision? Moreover, a single numerical optimization run may end in a local minimum — does this mean we should re-run the optimization? How many times?[-12-]

While numerical optimization is powerful, it is also the most time consuming part of algorithm 1. We therefore put some effort into limiting its use in two situations:

1. When a 'good enough' solution has been found.
2. When the performance of the numerical algorithm looks unpromising.

We added code[-13-] to L-BFGS to cater to halting the search in both of these scenarios. In the first case, the user provides a value for the overall circuit error that is considered 'good enough'. This will usually be a value close to zero — our experiments typically used 0.0001. It is possible for the numerical algorithm to continue optimizing to extremely small values of overall error[-14-], but any computational effort spent in this way is clearly going to produce very limited returns. We have added code to L-BFGS++ to facilitate this termination condition. In the second case, we ensure that the algorithm records the value of overall error for the last p main[-15-] iterations of the L-BFGS algorithm. This allows us to estimate the algorithm's rate of progress. If this rate of progress is such that it looks like the algorithm will not reach some target value t in some quota of iterations, q, then the search is considered unpromising and is halted. Parameter p is currently hard coded to 3, while q is provided by the user[-16-]. The target value is the overall error of the best circuit, of cost no larger than the one under consideration, found thus far.

[-12-] When combined with solution cacheing (see chapter 6.), such questions multiply. If we find that a child solution has already been evaluated and numerically optimized, should we consider re-running the numerical optimization in the hopes of finding a better local minimum? If a numerical optimization is halted due to it looking 'unpromising' (see below), should it be continued again if the circuit structure is revisited?

[-13-] The new termination conditions are in addition to the rule that stops numerical optimization if the norm of the gradient is less than 0.000001.

[-14-] Indeed, due to numerical error, experiments have occasionally produces small *negative* values for error!

[-15-] Here an iteration involves the selection of a search direction, using gradient information and the estimated inverse Hessian, and the application of line search. The line search also, of course, has iterations.

[-16-] Our experiments use 1000.

One other heuristic is used to prevent needless numerical optimization of certain circuits — indeed, to prevent evaluation at all. If a 'good enough' solution of no greater cost than that being considered has already been found, it seems, at first sight, that there is no need to continue circuit evaluation. There is, however, a slight complication. After numerical optimization, it is possible that some parameterized gates have their angles set at values that allows for gate reduction. By this, we mean the removal of the gate entirely, if the angle is sufficiently close to zero, or the replacement of the gate with some cheaper alternative, if the angle is sufficiently close to a special value. Hence it is possible that, for example, a circuit of cost 65 might be numerically optimized, despite the fact that we already have a good enough circuit at that cost, but then subsequently reduced so as to produce a new best circuit of cost, say, 60.

We therefore allow the user to enter a parameter referred to as the 'expected reduction'. Then if the expected reduction is, for example, 4, and the cost of the circuit under consideration is 59, we abort circuit evaluation if a circuit of cost no greater than 55 has already been found that is considered 'good enough'.[-17-]

[-17-] At present, the 'target value' used in the numerical optimization does not use this 'expected reduction'. I wonder whether it should.

5 Circuit reduction

The inclusion of numerical optimization methods within our algorithm raises the possibility of applying further simplifications, of a kind unlike those described in chapter 3. In chapter 3 we always ensured that the simplified circuit structure was always capable of modelling any circuit resulting from the original structure. In other words, whatever values we might choose for the gate parameters of the initial circuit, suitable matching values could always be found for the simplified version. Here, in contrast, we consider simplifications that are *only* possible because of the *specific* values taken by one or more gate parameters. To avoid confusion, we will refer to this new kind of simplification as a 'reduction'.

Note that it is not unreasonable to expect such reductions to be available after numerical optimization. The inclusion of parameterized gates can provide a circuit structure with a great deal of flexibility. Given a circuit structure with many parameterized gates, the numerical optimization can then exploit this flexibility to produce low error circuits. Then, if a parameterized gate is, in fact, surplus to requirements, we may discover that the optimal value for the gate angle was zero. This simply signals that the gate is not required and may be removed.

We therefore apply circuit reduction after numerical optimization. Of course, after circuit reduction, it is possible that new simplifications[-1-] become available, so the circuit simplification routine is called. There are three possible outcomes.

1. The circuit structure simplifies to a form found in the cache, but the result of numerical optimization is an overall cost that is better than that cached. The numerical optimization routine is briefly rerun[-2-], in order to fix any minor loss of quality during the reduction[-3-], after which the cache is updated.
2. The simplified circuit structure is already in the cache and the cached overall error is better than that just obtained via numerical optimization. The circuit[-4-] and error values are copied from the cache.
3. The simplified circuit structure is not found on the cache. In this case, we briefly rerun the numerical optimization and then cache the new circuit.

Note that it is possible that a circuit undergoes multiple reduce-simplify cycles. The simplification part may result, for example, in

[-1-] Of the form described in chapter 3.

[-2-] The run is brief, as the gate parameters should already be close to that for a local minimum.

[-3-] We will see that gates are reduced whenever the gate angle is *close* to a special value. In other words there is some tolerance. This implies that the replacement gate's output is not an exact match to the original gate's. Hence the overall error of the circuit will undergo a small change.

[-4-] Specifically, the *most reduced* version of the circuit is copied. The cached version will have different gate parameter values, which may have led to further gate reductions.

two parameterized gates such as XRotations being merged, with the resulting gate having an angle close to zero. Furthermore, the re-optimizations described above may adjust an angle to a value within the tolerance of a special value.

Having described the use of circuit reduction in the algorithm, we continue by describing the gate reductions that may take place. When we refer to a gate angle being 'close' to a special value, we mean to within a specified tolerance. This angle tolerance is, at present, hard coded to a value of 0.01 radians. Note that, in all cases, the gate reduction is only performed if the replacement is both available and of lower cost.

XRotation: Gate may be reduced if the angle is close to a multiple of π.

Angle close to 0*:* Gate is removed.

Angle close to $\pm 2\pi$*:* The gate *must* have at least one control, as the angle for uncontrolled gates is constrained to be in $[-\pi, \pi]$. The matrix applied to the target bit is just minus the identity matrix, i.e. the gate just applies a phase of -1 when the control qubits are set.

$$X(2\pi) = \begin{pmatrix} \cos\pi & i\sin\pi \\ i\sin\pi & \cos\pi \end{pmatrix} = -\begin{pmatrix} 1 & 0 \\ 0 & 1 \end{pmatrix}.$$

Hence, the gate is replaced by a ZGate across the control qubits. If a ZGate is unavailable, an equivalent ArbitraryPhase may be used instead, provided it is cheaper than the replaced XRotation.[-5-]

[-5-] Note that using an ArbitraryPhase introduces an extra useful degree of freedom, opening up the possibility of further numerical optimization beyond just minor adjustments to the previous result.

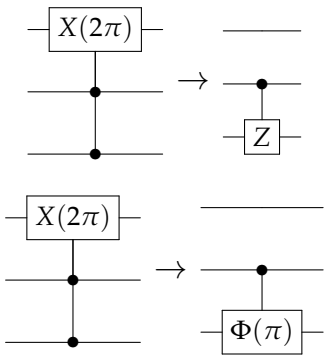


Figure 5.65: Reductions of an XGate of angle $\pm 2\pi$.

Angle close to π*:* If uncontrolled, the gate is replaced by a simple XGate. If the XRotation has controls, the reduction must also include a PhaseGate (or equivalent ArbitraryPhase) across the control qubits, to account for the phase difference of i.

$$X(\pi) = \begin{pmatrix} \cos\frac{\pi}{2} & i\sin\frac{\pi}{2} \\ i\sin\frac{\pi}{2} & \cos\frac{\pi}{2} \end{pmatrix} = i\begin{pmatrix} 0 & 1 \\ 1 & 0 \end{pmatrix} = iX.$$

XRotation - angle close to $-\pi$*:* As for an angle of $+\pi$, but with any PhaseGate replaced by a PhaseInv (or equivalent).

$$X(-\pi) = \begin{pmatrix} \cos\frac{-\pi}{2} & i\sin\frac{-\pi}{2} \\ i\sin\frac{-\pi}{2} & \cos\frac{-\pi}{2} \end{pmatrix} = -i\begin{pmatrix} 0 & 1 \\ 1 & 0 \end{pmatrix} = -iX.$$

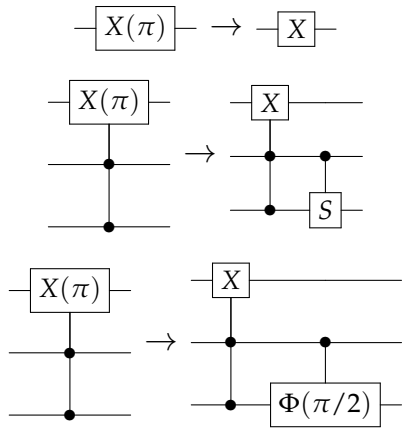


Figure 5.66: Reductions of an XGate of angle π.

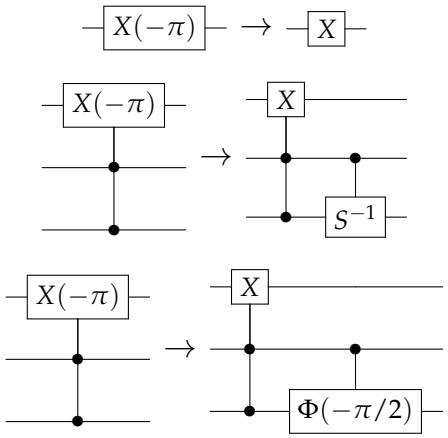


Figure 5.67: Reductions of an XGate of angle $-\pi$.

YRotation: As for the XRotation, but with the following additional options when the angle is close to a multiple of $\pi/2$.

Angle close to $\pi/2$: The gate can be replaced by an XGate followed by a Hadamard, or a Hadamard followed by a ZGate, whichever is cheaper, regardless of the existence of controls, with the new gates having the same controls as the old.[-6-]

$$Y(\pi/2) = \begin{pmatrix} \cos\frac{\pi}{4} & \sin\frac{\pi}{4} \\ -\sin\frac{\pi}{4} & \cos\frac{\pi}{4} \end{pmatrix} = \frac{1}{\sqrt{2}}\begin{pmatrix} 1 & 1 \\ -1 & 1 \end{pmatrix},$$
$$HX = \frac{1}{\sqrt{2}}\begin{pmatrix} 1 & 1 \\ 1 & -1 \end{pmatrix}\begin{pmatrix} 0 & 1 \\ 1 & 0 \end{pmatrix} = \frac{1}{\sqrt{2}}\begin{pmatrix} 1 & 1 \\ -1 & 1 \end{pmatrix},$$
$$ZH = \frac{1}{\sqrt{2}}\begin{pmatrix} 1 & 0 \\ 0 & -1 \end{pmatrix}\begin{pmatrix} 1 & 1 \\ 1 & -1 \end{pmatrix} = \frac{1}{\sqrt{2}}\begin{pmatrix} 1 & 1 \\ -1 & 1 \end{pmatrix}.$$

If the XGate and ZGate have the same cost, the code prefers the ZGate option. This continues to be the case if the ZGate is actually an equivalent ArbitraryPhase, at the same cost as the XGate[-7-].

Angle close to $-\pi/2$: As for the $+\pi/2$ case, but with the ZGate

[-6-] This is due to a fortuitous cancellation of phases.

[-7-] The extra degree of freedom might be useful if we perform more numerical optimization.

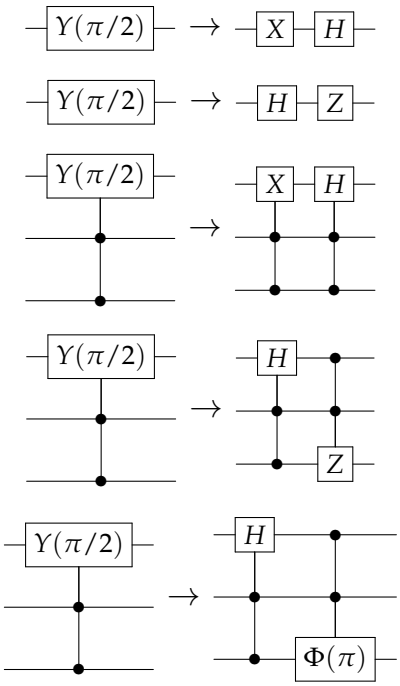


Figure 5.68: Reductions of a YGate of angle $\pi/2$. As usual, PhaseTypeGates such as the ZGate are illustrated operating on the lowest qubit — recall that the target of such gates can always be swapped with a control.

before the Hadamard, or the XGate after.

$$Y(-\pi/2) = \begin{pmatrix} \cos\frac{-\pi}{4} & \sin\frac{-\pi}{4} \\ -\sin\frac{-\pi}{4} & \cos\frac{-\pi}{4} \end{pmatrix} = \frac{1}{\sqrt{2}}\begin{pmatrix} 1 & -1 \\ 1 & 1 \end{pmatrix},$$

$$XH = \frac{1}{\sqrt{2}}\begin{pmatrix} 0 & 1 \\ 1 & 0 \end{pmatrix}\begin{pmatrix} 1 & 1 \\ 1 & -1 \end{pmatrix} = \frac{1}{\sqrt{2}}\begin{pmatrix} 1 & -1 \\ 1 & 1 \end{pmatrix},$$

$$HZ = \frac{1}{\sqrt{2}}\begin{pmatrix} 1 & 1 \\ 1 & -1 \end{pmatrix}\begin{pmatrix} 1 & 0 \\ 0 & -1 \end{pmatrix} = \frac{1}{\sqrt{2}}\begin{pmatrix} 1 & -1 \\ 1 & 1 \end{pmatrix}.$$

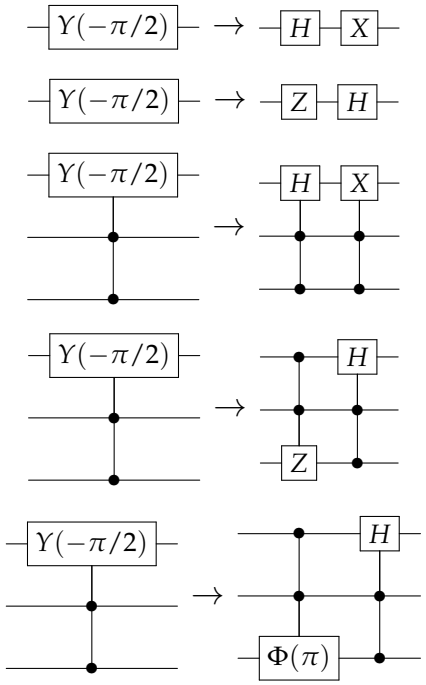


Figure 5.69: Reductions of a YGate of angle $-\pi/2$.

Angle close to $3\pi/2$*:* As for the $-\pi/2$ case, except that an additional ZGate is required on the control qubits to account for a phase difference — the additional overall minus sign in equation 5.1.[-8-]

[-8-] Note that the YRotation must, in this case, have control qubits, since the angle exceeds π. 'Reducing' a single gate to three new gates does little to reduce the conceptual complexity of the circuit! However, it may reduce implementation cost.

$$Y(3\pi/2) = \begin{pmatrix} \cos\frac{3\pi}{4} & \sin\frac{3\pi}{4} \\ -\sin\frac{3\pi}{4} & \cos\frac{3\pi}{4} \end{pmatrix} = -\frac{1}{\sqrt{2}}\begin{pmatrix} 1 & -1 \\ 1 & 1 \end{pmatrix}. \quad (5.1)$$

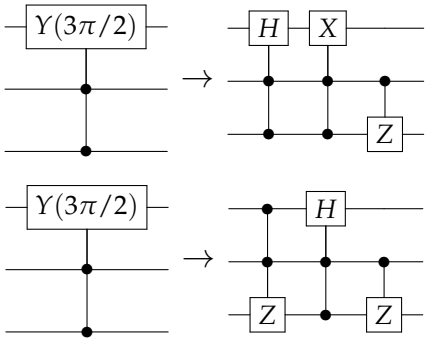


Figure 5.70: Reductions of a YGate of angle $3\pi/2$, ignoring cases where a ZGate is replaced with an equivalent ArbitraryPhase.

Angle close to $-3\pi/2$*:* As for the $+\pi/2$ case, except that an additional ZGate is required on the control qubits to account for a phase difference.[-9-]

[-9-] Again, the YRotation must have control qubits in this case. Figure of reductions in this case are omitted for brevity.

$$Y(-3\pi/2) = \begin{pmatrix} \cos\frac{-3\pi}{4} & \sin\frac{-3\pi}{4} \\ -\sin\frac{-3\pi}{4} & \cos\frac{-3\pi}{4} \end{pmatrix} = -\frac{1}{\sqrt{2}}\begin{pmatrix} 1 & 1 \\ -1 & 1 \end{pmatrix}.$$

ZRotation: Reductions for angles close to a multiple of π take forms similar to those for the XRotation and YRotation gates. A number of reductions for other angles also exist, as follows.

Angle close to $\pi/2$*:* If uncontrolled, this gate may be replaced by a PhaseInv. With controls, and additional PiByEight gate is required on the control qubits.

$$Z(\pi/2) = \begin{pmatrix} e^{i\pi/4} & 0 \\ 0 & e^{-i\pi/4} \end{pmatrix} = e^{i\pi/4}\begin{pmatrix} 1 & 0 \\ 0 & e^{-i\pi/2} \end{pmatrix}$$
$$= e^{i\pi/4}\begin{pmatrix} 1 & 0 \\ 0 & -i \end{pmatrix} = e^{i\pi/4}S^{-1}.$$

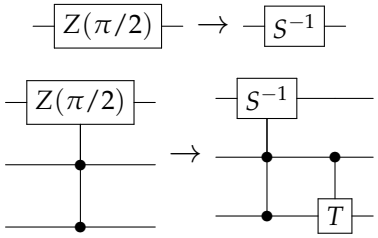


Figure 5.71: Reductions of an ZGate of angle $\pi/2$, omitting variants where one or both of the new gates are unavailable and are instead emulated by ArbitraryPhase gates.

Angle close to $-\pi/2$*:* The inverse of the previous case, so replaced a PhaseGate with a PiByEightInv on the control bits, if present.

Angle close to $3\pi/2$*:* Gate is replaced by a PhaseGate and an ArbitraryPhase, of angle $3\pi/4$ across the control bits, if present.[-10-]

$$Z(3\pi/2) = \begin{pmatrix} e^{3\pi i/4} & 0 \\ 0 & e^{-3\pi i/4} \end{pmatrix} = e^{3\pi i/4} \begin{pmatrix} 1 & 0 \\ 0 & e^{-3\pi i/2} \end{pmatrix} = e^{3\pi i/4} \begin{pmatrix} 1 & 0 \\ 0 & i \end{pmatrix} = e^{3\pi i/4} S.$$

[-10-] This ArbitraryPhase is equivalent to either a PiByEightInv and a ZGate, or a PiByEight and a PhaseGate. The code does not consider splitting the ArbitraryPhase in either of these ways. Perhaps all gates generated by the PiByEight gate, i.e. all gates that shift the phase by a multiple of $\pi/4$, should be included amongst the basic gate types.

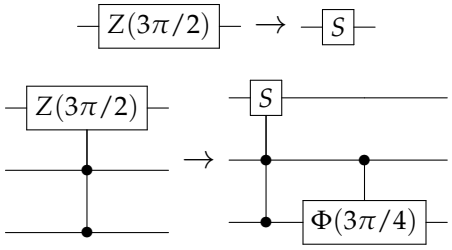


Figure 5.72: Reductions of an ZGate of angle $3\pi/2$, omitting variants where the PhaseGate is unavailable and instead emulated by an ArbitraryPhase gate.

Angle close to $-3\pi/2$*:* The inverse of the previous case.

Angle close to $\pi/4$*:* Replace with a PiByEightInv, with an ArbitraryPhase of angle $\pi/8$ across the control bits, if present.

$$Z(\pi/4) = \begin{pmatrix} e^{\pi i/8} & 0 \\ 0 & e^{-\pi i/8} \end{pmatrix} = e^{\pi i/8} \begin{pmatrix} 1 & 0 \\ 0 & e^{-\pi i/4} \end{pmatrix} = e^{\pi i/8} T^{-1}.$$

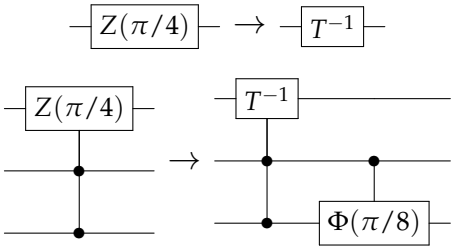


Figure 5.73: Reductions of an ZGate of angle $\pi/4$, omitting variants where the PiByEight is unavailable and instead emulated by an ArbitraryPhase gate.

Angle close to $-\pi/4$*:* The inverse of the previous case.

Angle close to $7\pi/4$*:* Equivalent to a PiByEight and an ArbitraryPhase of angle $7\pi/8$ across the control bits[-11-].

Angle close to $-7\pi/4$*:* The inverse of the previous case.

Angle close to some other multiple of $\pi/4$*:* In these cases, the gate could be reduced to a combination of PhaseTypeGates, with the same target and controls as the original gate, with an ArbitraryPhase of the appropriate angle across the control qubits, if present. These are not currently implemented.[-12-]

[-11-] Again, such a gate will have controls.

[-12-] Such reductions could decrease circuit cost, but also increase the conceptual complexity of the circuit. If we were, in future, to implement the other gates in the group generated by the PiByEight, then these reductions would also be included.

ArbitraryPhase: Reductions are similar to those for the ZRotation, but require no additional gates across the control qubits, as there is no phase difference to correct for.[-13-]

Angle close to zero: Gate is eliminated.

Angle close to $\pm\pi$*:* Reduces to a ZGate.

Angle close to $\pi/2$*:* Reduces to a PhaseGate.

[-13-] Also note that, according to the conventions used in the code, the ZRotation and ArbitraryPhase rotate in opposite directions, that is an uncontrolled ZRotation of angle θ is equivalent to an uncontrolled ArbitraryPhase of angle $-\theta$.

Angle close to $-\pi/2$*:* Reduces to a PhaseInv.

Angle close to $\pi/4$*:* Reduces to a PiByEight.

Angle close to $-\pi/4$*:* Reduces to a PiByEightInv.

Angle close to $3\pi/4$*:* Reduces to a PhaseGate and a PiByEight.[-14-]

Angle close to $-3\pi/4$*:* Reduces to a PhaseInv and a PiByEightInv.

[-14-] These are slightly simpler than the equivalent cases for the ZRotation, so we have included them. Note that we don't need to consider angles close to $\pm 5\pi/4$ or $\pm 7\pi/4$, since the angle of an ArbitraryPhase always lies in $[-\pi, \pi]$, even if the gate has controls.

6 Solution cacheing

Given the propensity of a genetic algorithm to revisit solutions, which in our case are the circuit structures, and the high computation cost of evaluating them, through the use of numerical optimization and quantum simulation, it make some sense to detect when a solution is revisited, retrieving the information regarding the previous evaluation of the circuit structure from a cache of such data, rather than performing the expensive solution evaluation again.

We note in advance that cache implementation is complicated by the use of circuit simplification and gate reduction routines. Which circuits should be cached? What data should be stored and how should the cache be indexed? What costs, both in terms of computing time and memory, are involved in maintaining such a cache? This chapter considers these questions and outlines the work done thus far to implement such a cache, its pros and cons, and possibilities for future developments.

There are also questions regarding how best to *use* the cached information, in particular with regard to the control of numerical optimization. We noted previously that numerical optimization may halt in a local minimum or be halted due to it looking unpromising. So, upon detecting, through use of the cache, that the GA is revisiting a circuit structure, should the algorithm merely copy the data — gate parameter values and objective functions — from the cache, or should it attempt to improve upon the results of the previous numerical optimization? At present, only the former option is provided, with the alternative being to switch off solution cacheing. In future, we hope to provide more nuanced methods for exploiting the cached data.

6.1 Indexing

We first focus on how the cache is indexed and the top level structure of the cache. What data is used to look up the associated stored information and how is this data manipulated to give the memory address[-1-] of the information we are looking for.

The simplest cache for use in circuit structure optimization would allow the algorithm to determine whether a circuit structure had previously been evaluated and, if so, to look up the optimal values for the gate parameters and the overall and worst-case errors

[-1-] Or possibly, a memory location with data that points to another memory location that contains the information we are looking for. Or...

for the resulting circuit. So the cache would be indexed on circuit structure, with each item in the cache containing circuit details (i.e. gate parameters) and objective values. Leaving aside, for now, the details of how to implement such a data structure, we first ask the question: which circuit structure? Should the circuit structure be simplified before searching the cache, or should we be able to index the cache using the circuit structure prior to simplification?

At present, the code uses an approach somewhere between these two extremes. Given that some of our experiments have used very small numbers of qubits, the computational costs of circuit simplification are not insignificant. We would like to avoid these costs where possible. However, given a circuit structure, it is usually possible to find many equivalent structure, of equal cost, simply by applying the simple, H- and R-swaps. Indeed, the number of such circuit structures can grow exponentially. If we merely cached the unsimplified circuit, then we would not be able to use the cached data one when of these equivalent circuits is encountered. If we attempt to find all such equivalent circuits and create a separate cache entry for each, then the computational costs of operating the cache become excessive[-2-]. The solution, of course, is to create an entry in the cache for only the canonical form[-3-] of the circuit structure. We therefore pay the price of putting the circuit into canonical form, in order to make an effective cache that allows us to avoid the simplification and evaluation of solutions that have been visited before.

[-2-] This is true even if each cache entry is merely a link to a single location of cached data.
[-3-] As describe in section 3.4.

We now consider the top-level implementation of the cache. As might easily be predicted, the cache is implemented as a hash table. In C++ this means use of the `std::unordered_map` templated container class, which requires information on

1. the index type;
2. the stored data type;
3. a hash function that produces an integer hash-value for the index data, used to index the top level array;
4. a function that indicates whether two indices are equal.

The index type, in our case, is just the Circuit. We will describe the stored data later. Two circuits are considered equal, as far as the cache is concerned, if they have the same structure - i.e. we index on circuit structure. The remains of this section describe the hash function for circuits[-4-].

[-4-] Or rather, for circuit structures.

Working in bottom-up manner, we start with individual gates. First note that we ignore any gate parameters — we index the cache using circuit *structures*. We wish to assign to each combination of gate type, target qubit and control qubits[-5-], a unique non-negative integer. As an initial step, the Gate class declares a (pure virtual) function called `numQbitOptions()`. This is defined in the SingleTargetGate, PhaseTypeGate, SwapGate and Oracle classes. When called

[-5-] Referred to as a 'gate option'.

on a concrete gate, it returns the number of options for target and controls a gate of that type. For an Oracle, it simply returns one. For a swap gate, it returns

$$\binom{q}{2},$$

where q is the number of qubits. For a single target gate, it returns

$$q\sum_{c\in C}\binom{q-1}{c},$$

where $c \in C$ if and only if a gate of that type is available with c controls. The set C is provided by the circuit context class, which each gate has access to. So, for a five qubit problem and a gate type that is only available with either zero or two control qubits, we would get

$$5\binom{4}{0}+5\binom{4}{2}=5+30=35$$

options, the first five of which are uncontrolled gates while the remaining thirty are those gates with two controls. Finally, for a phase type gate, there is no distinction between target and control bits. Hence the number of options becomes

$$\sum_{c\in C}\binom{q}{c+1}.$$

So, repeating the case of a five qubit problem and a gate that may have zero or two controls, we get

$$\binom{5}{1}+\binom{5}{3}=5+10=15$$

options.

Having determined the number of options for each gate type, a 'base identifier' is determined for each gate type, as described in section 2.2 and described again here. Gate types are listed in a predetermined order. The base identifier for a gate type is simply the total number of options available for all the gate types that occur earlier in this ordering. So, if the permitted gate types are PhaseGate, PhaseInv, XGate and YGate and it its determined that there are 5 PhaseGate options, 5 PhaseInv options and 8 XGate options, then the base identifier for the four gate types are 0, 5, 10 and 18 respectively. The base identifier for each gate type is stored in the circuit context class.

To complete the calculation of the identifier for each gate type, target and control combination, we add the base identifier for the gate type to a number, from zero to the number of gate options, for the particular target/control combination present. This is performed by a function called `calculate_option_id()` when each gate is created, storing the identifier in the Gate object. This function is defined, in different ways, by the SingleTargetGate, PhaseTypeGate, SwapGate and Oracle classes. For an Oracle, we simply

take the base identifier for the Oracle class, i.e. we add nothing, since there is only one option for the Oracle. For a SwapGate, the number added to the base identifier comes in two parts. The first is given by

$$qa - \frac{a(a+1)}{2},$$

where q is the number of qubits and a is the identifier for the first qubit involved in the gate, from 0 to $q-1$. For $q = 5$, this gives 0, 4, 7 or 9, depending on the value of the first qubit[-6-]. The second components is given by

$$b - a - 1.$$

This fills in the gaps, so for $q = 5$, this gives us a number from 0 to 9 to add to the base identifier.

[-6-] We give here values for $a = 0, 1, 2, 3$ respectively. We know that $a < 4$, since 4 is the identifier for the last qubit and the qubits involved in a swap gate are listed in increasing order.

For the remaining gate types, the value added to the base identifier again comes in two parts. The first is simply the number of available options for the gate type that involve fewer qubits. So, if the gate being considered has d controls, this first component is given by

$$q \sum_{c \in C, c<d} \binom{q-1}{c}$$

for a SingleTargetGate, while for a PhaseTypeGate we get

$$\sum_{c \in C, c<d} \binom{q}{c+1}.$$

The second component involves the calculation of the 'combination rank' of a set of numbers. Given a combination of k numbers, $0 \leq n_1 < n_2 < \ldots < n_k$, the rank of the combination is given by

$$\binom{n_1}{1} + \binom{n_2}{2} + \ldots + \binom{n_k}{k}.$$

This assigns a unique number to each of the combinations of k numbers.[-7-] Now in the case of a PhaseTypeGate, since target and controls are interchangeable, the second component is just the combination rank of the qubits involved in the gate. In the case of a SingleTargetGate that is not a PhaseTypeGate, we take the combination rank of the controls[-8-], multiply by q and add the index of the target qubit.

[-7-] Note that this does not require the size of the set from which the numbers are selected. Regardless, when selecting from $\{0, 1, \ldots, n\}$, we get a unique number from 0 to $\binom{n}{k} - 1$. See the Wikipedia article on "Combinatorial Number System" for more details.

[-8-] To account for the fact that the target cannot be a control, we subtract one from the index of any control with an index that exceeds that of the target qubit.

Now that each gate option is assigned a unique integer identifier. the hash function for the circuit is obtained as follows. Start with the value zero. Then considering each gate in turn, multiply the current value by the number of permitted gate options, then add the option identifier for the gate present.

Of course, for larger circuits, this process may lead to overflow. However, we can ignore this, provided we make sure that we use a suitable, unsigned integer type. What is important is that the likelihood that two different circuits hash to the same value should be low. This seems likely using this approach. Note also that the data

type used — `std::unordered_map` — will apply some additional processing to the hash value provided to it, in order to ensure that we get a valid index into the top level array of the hash table.[-9-]

[-9-] This data structure can automatically adjust the size of the top level array, if it deems it necessary. However, such restructuring is likely to be expensive. We can, however, also reserve space for a hash table of some specified size. (At present, we request an array of a million 'buckets'.) In both cases, the hash value provided by my code must be converted into an array index, or 'bucket number'.

6.2 *The stored data*

As already mentioned, the cached data is held in a hash table. In more detail, a hash table is an array, indexed via use of the hash function, where each element of the array is a linked list of items. This caters for the possibility that two different circuits structures may hash to the same value. Provided such clashes are rare, the overall array structure allows for quick access to the stored items.

For our cache, the items stored are simply pointers[-10-] to simple `CachedStats` objects. In turn, a `CachedStats` object contains the following:

[-10-] To be specific, `std::shared_ptr<CachedStats>`, that is a smart pointer to data that may be referenced (shared) by many such pointers.

1. A pointer[-11-] to the simplified circuit. Any gate parameters present are set to the best values discovered by the numerical optimization.
2. The best worst-case and overall errors obtained by the numerical optimization, if required, or simply those values obtained via simulation if the circuit contains no parameterized gates.
3. A pointer[-12-] to the CachedStats for the reduced circuit, if any circuit reduction takes place. If not, the pointer is set to be null.
4. A 'mutex'[-13-] allowing for control of parallel access to the cached data. (See section 6.4.)

[-11-] This time the pointer is a `std::unique_ptr<Circuit>`, i.e. the Circuit is considered as 'owned' by the CachedStats object. This naturally requires the circuit to be *copied*, implying significant time and memory costs.

[-12-] A `shared_ptr` again, this time.

[-13-] A `std::shared_timed_mutex`. All I really wanted was a `std::shared_mutex`, but the compiler on the Mac could not find it!

By storing only pointers in the hash table, we can arrange for different circuits, with different hash values, to index the same CachedStats data. So both the original circuit and the simplified circuit, and indeed any circuits encountered midway through simplification, get their own pointer in the hash table, referring to the same stored data.

Note that the data stored is minimal and allows for only fairly unsophisticated use. At present, the algorithm merely detects whether the circuit has been encountered before and, if so, copies the stored data into the current solution. This approach naturally raises some concerns. Of primary concern is what happens if the numerical optimization applied to a circuit is unsuccessful at finding the global optimum. It would appear that, once the numerical optimization has failed in this way, it is never given a chance to succeed, as the algorithm at all future times merely references the cache, rather than re-optimizing.[-14-] This is naturally worrying. There are a number of possibilities for dealing with this issue, of varying degrees of sophistication. We could, on detection of the circuit structure in the cache, decide, with some low probability, to reoptimize anyway, updating the cache if the obtained results are an improvement. This would keep much of the efficiency boost made available through cacheing while spending a little extra effort

[-14-] In actuality, once numerical optimization has failed on a circuit structure, there is still a chance of finding a better circuit with the same structure. It is possible that numerical optimization may be successfully applied to a larger circuit, which is then subsequently reduced to the circuit structure in question. See chapter 5 for more details.

on ensuring that numerical optimization finds the global minimum. We could arrange for this probability to depend on the complexity of the circuit — numerical optimization is more likely to have been successful on circuits with few gate parameters. We could cache the locations in the search space visited by the numerical optimization algorithm, to ensure that any reoptimization avoids previously visited circuits. Moreover, we could cache the finish status of the numerical optimization. This would allow us to continue numerical optimizations that had previously been considered unpromising. Each of these possibilities are, however, a matter for future work.

6.3 *Interaction with circuit simplification*

As hinted at in the previous section, solution cacheing interacts fairly heavily with the process of circuit structure simplification. Moreover, the desired behaviour of the simplification, numerical optimization and reduction routines impacts upon the structure of the cache.

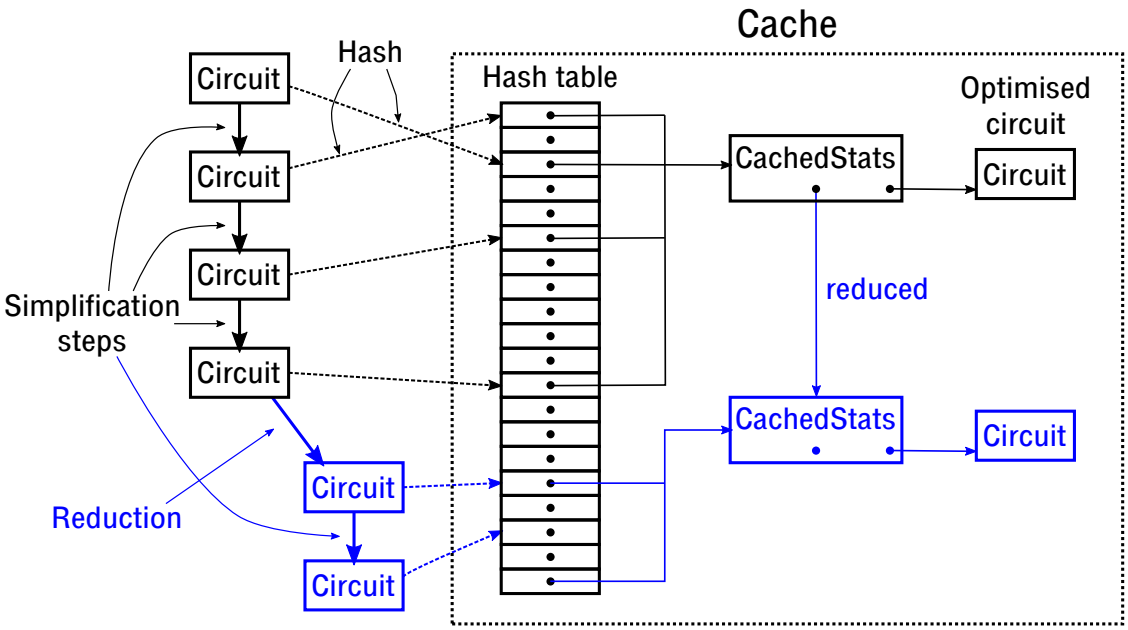


Figure 6.74: Circuit simplification and the cache structure. Here the hash table is portrayed as a simple array. The black parts of the diagram illustrate how a simple simplification run, with no circuit reduction, interacts with the cache. In blue, the figure shows the result if, after numerical optimization, it is discovered that the circuit may be reduced and simplified further.

Figure 6.74 is a simplified illustration of how the cache is structured and used. Three routines most heavily interact with the cache are `Solution::evaluateObjectives`, `Circuit::simplifyCircuit`, `Circuit::simplifyStructure` and `Solution::optimize_and_cache`. `Circuit::simplifyCircuit` and `Solution::optimize_and_cache` only write to the cache, while `Circuit::simplifyStructure` also examines the cache to determine whether the circuit being simplified has been seen before. The results are also used to shortcut subsequent evaluation of the solution. `Solution::evaluateObjectives` calls the other functions, but also writes objective values and information about circuit reductions into the cache, copies circuits from the cache (if a circuit structure is revisited) and compares the

results of numerical optimization with objective values stored in the cache[-15-].

The left hand side of the diagram illustrates the process of circuit simplification (and possible reduction) while the right hand side illustrates how this affects the cache. Consider a single run of the function `Circuit::simplifyCircuit`. A number of simplification steps are performed[-16-], as shown in black on the left. Once circuit simplification is complete, each of these circuits structures is entered into the hash table, together with a link to a `CachedStats` object. This object includes a copy of the simplified circuit structure. If the simplified circuit contains parameterized gates, numerical optimization is applied within the function `Solution::optimize_and_cache`. The optimized gate parameter values are copied to the CachedStats' copy of the Circuit, so that the CachedStats contains the optimized circuit. The `Solution::evaluateObjectives` function also adds objective values to the CachedStats object.

Of course, when simplifying a circuit, the code usually calls `Circuit::simplifyStructure`. This follows the same steps as before, but also checks, after each step, whether the circuit structure is already present in the cache. So, if the initial circuit structure is not in the cache, but after the first simplification, the new circuit structure is found to be present, a new entry for the initial circuit structure is added to the hash table, pointing to the existing `CachedStats` object, while the fully simplified (and numerically optimized) circuit is copied from the cache.[-17-]

The parts of the diagram in blue illustrate what happens if, after numerical optimization is performed, the circuit is found to have parameterized gates suitable for reduction. In this case, the reduced circuit is also simplified. Any additional circuit structure visited is added to the hash table, but note that these circuits get their own `CachedStats` object, which in turn contains a link to the final optimized circuit. The first `CachedStats` object also gets a link to the `CachedStats` object for the reduced circuit.

Giving the reduced circuit its own `CachedStats` facilitates potential future additions to the code that might perform numerical reoptimization. During simplification of a circuit, we may discover the circuit structure in the cache, causing us to access the `CachedStats`. There then might be reason to rerun numerical optimization, or to continue numerical optimization that had previously been halted due to looking 'unpromising'. After the reoptimization, a circuit with lower errors may be discovered, but one which can no longer be reduced in the same way. In this case, the gate parameters in the circuit held by the `CachedStats` would be updated and the link to the `CachedStats` of the reduced circuit severed. However, at present, we do not perform such reoptimization, so currently when a circuit structure is discovered in the cache during simplification, the circuit copied from the cache is obtained by following the 'reduced' links until the most reduced circuit is found.

[-15-] This is done in the case where numerical optimization is applied to a previously unseen circuit structure, resulting in gate parameter values that allow for circuit reduction. If the circuit reduction (and any following simplification) results in a previously seen circuit structure, then there is the possibility that the new numerical optimization has produced a result better than that cached. (This is currently the only way in which the results of a numerical optimization that gets stuck at a local optimum can be overwritten.)

[-16-] Recall that each simplification step is followed by a step that ensures that the new circuit is in canonical form. It is the canonical structure that is used to index the hash table.

[-17-] If `Circuit::simplifyCircuit` is called when some of the circuit structures visited are already in the cache, then the new ones simply aren't added. (I might want to check this. I also need to remind myself why `Circuit::simplifyCircuit` is necessary. (It is currently used after circuit reduction has taken place.))

Note that the hash table in the figure is portrayed as a simple array of pointers. In practice it is a `std::unordered_map`[18], where each each stored element consists of a circuit (which acts as the key) and a pointer to `CachedStats`. These details are omitted for clarity.

[18] Effectively just an array of linked lists, or one would expect from a hash table.

6.4 *The cache and parallelization*

Parallelization of the code is mostly straightforward. Whenever a population of solutions needs to be evaluated, the solutions are placed into a pool representing the work that remains to be done. Threads are created, matching the number of hardware threads supported by the system. Each thread takes solutions from the pool and evaluates them until the pool is emptied, at which point the threads terminate and the code returns to working in serial. Parallelization only takes place at the level of solution evaluation, i.e. the evaluation of circuit structures. Circuit simplification and gate parameter optimization are not parallelized separately.

The cache complicates matters. Without the cache, each thread works on its own solutions, no data is shared, so there is no need for, for example, mutexes and locks to prevent race conditions. The cache, however, is a shared data structure. Class `CircuitCache` therefore contains a mutex[19], as do each of the `CachedStats` objects.

[19] In detail, a `std::shared_timed_mutex`. We only really wanted a `std::shared_mutex`, but the compiler on the Mac couldn't find it!

The `CachedStats` class hides the mutex, but provides two functions, `getReadersLock`[20] and `getWritersLock`[21], leaving the rest of the code to manage parallel access. In contrast `CircuitCache` (the main cache class) manages its own parallel access. Two private member functions, `find_unthreaded` and `add_unthreaded` that perform cache manipulation are called by three public member functions, `find` and two versions of `add`, that include appropriate locking of the mutex for thread-safe cache manipulation.

[20] A `std::shared_lock`

[21] A `std::unique_lock`

The first version of `CircuitCache::add`, used to add a vector of circuits that will refer to an existing `CachedStats` object, is straightforward, locking the mutex[22] and calling `add_unthreaded`. The second version is used to add a vector of circuits with a *new*, empty `CachedStats` object, and is trickier. After obtaining the lock, the cache is searched for the simplified circuit (which is one of the parameters of this `add` function). If it is found, then the empty `CachedStats` is discarded and the function proceeds as with the first version of `add`. Otherwise the empty `CachedStats` object is first linked to the simplified circuit before the call to `add_unthreaded`.

[22] A `std::unique_lock` suitable for when writing to the data structure, that prevents any other thread obtaining a lock.

Checking the cache in this second version of `add` is necessary for two separate reasons. First, it is possible that the circuit was simplified using the function `Circuit::simplifyCircuit`, which does not check the cache first[23]. Second, two threads might discover the same simplified circuit simultaneously. One successfully completes and adds the empty `CachedStats` object to the cache. The second must detect that this has happened so that it can link the hash table elements associated with each circuit to the correct `CachedStats`.

[23] As noted before, I need to remind myself why we used `simplifyCircuit` at all. However, even if we could eliminate use of `simplifyCircuit`, the second `add` function would still need to check the cache for the second reason — a reason that is all about with the parallelization.

This is not a frequent occurrence, but does happen[-24-]

Functions `CachedStats::getReadersLock` and `CachedStats::getWritersLock` are called by two functions, `mostReduced` and `Solution::evaluateObjectives`. Function `mostReduced` merely follows the chain[-25-] of 'reduced' links in the `CachedStats` objects, to find the most reduced circuit. To prevent another thread from writing to the `CachedStats` objects at the same time as `mostReduced` attempts to read them, a 'reader's' lock is applied to each `CachedStats` object in turn. `Solution::evaluateObjectives` applies a 'reader's' lock whenever it discovers a circuit is in the cache and copies the replacement simplified and optimized circuit from the cache. It also applies a 'writer's' lock on any newly created `CachedStats` objects, and on the `CachedStats` object for a reduced circuit, when checking whether the recent numerical optimization has improved the error values.

[-24-] Tests suggest that this occurs in about 10% of runs on 300 generations with a population of 100 circuits.

[-25-] It is possible that after reduction of a circuit, further simplification and numerical optimization may result in further reduction becoming available.

7 Results

7.1 The Fourier problem

We start with the Fourier problem, with the gate set restricted to uncontrolled YRotation gates, ArbitraryPhase gates (with arbitrary controls) and SwapGates. This gate set matches that used by Václav in his experiments, and thus allows for some comparison of algorithm performance. There is, as described above in section 2.5, an important difference in how gate costs are handled. Recall that in Václav's work, the count of gates of each type form a set of objectives. Here, we use a single circuit cost objective (in combination with the error objectives), where SwapGates have a cost of 1 and other gates cost 10.

7.1.1 4 qubits

Ignoring the 3 qubit case, we start with 4 qubits. Algorithm parameters settings[-1-] were as follows:

Population size: 100

Crossover probability: 0.1

Mutation probability: 0.2[-2-]

Mean length of initial random circuits: 50 gates

Maximum circuit length: 100 gates

'Good enough' overall error: 0.0001

Expected reduction after numerical optimization: 30

Numerical optimization iteration quota: 1000

Number of generations: 600

Gate parameters were not randomized prior to each numerical optimization. A solution cache was in use. 100 runs were performed on the linode.

On average, each run took 183 seconds. Note that much of this time was taken by the first 60 generations, which took on average 77 seconds. There are a number of potential contributors to this 'slow start'. I suspect the primary reason was the presence of larger circuits, requiring longer evaluation times[-3-] and more steps in numerical optimization. It is, of course, also true that, early in the

[-1-] Since I haven't finished writing up the full description of the algorithm, some of these parameters haven't yet been described. Ignore them for now.

[-2-] This is the probability of performing each of the six mutation operator, making the application of at least one mutation highly likely.

[-3-] Note how the second algorithm, in comparison, allows us to not fear larger circuits.

search, it is less likely that we will find that the circuit has already been evaluated with results stored in the cache.

Of the 100 runs, all find a zero error solution with the 600 generations. (In fact 120 generations was sufficient.) 96 of the runs find zero error with a circuit cost of 142, i.e. they find a perfect circuit[-4-]. In the remaining four runs, the best cost obtained for a zero error circuit was 152, i.e. a single additional gate was required. In these four runs, the best solution obtained at a cost of 142 had an overall error of 0.0032 and a worst case error of 0.0043. In each case, the ArbitraryPhase gate of the ideal circuit between qubits zero and three was replaced by an ArbitraryPhase between either qubits zero, one and three, or zero, two and three.

If we look at the intermediate results at 60 generations, we find that 99 runs produce zero overall error, with the one remaining run giving an overall error of 0.0029. 18 runs find zero error with the ideal circuit of cost 142. Figure 7.75 shows how the zero cost solutions are reduced in size over the run of the algorithm.

[-4-] Four YRotations, four uncontrolled ArbitraryPhase gates, six controlled ArbitraryPhase gates and two SwapGates, for a total of $10 \times (4 + 4 + 6) + 2 = 142$.

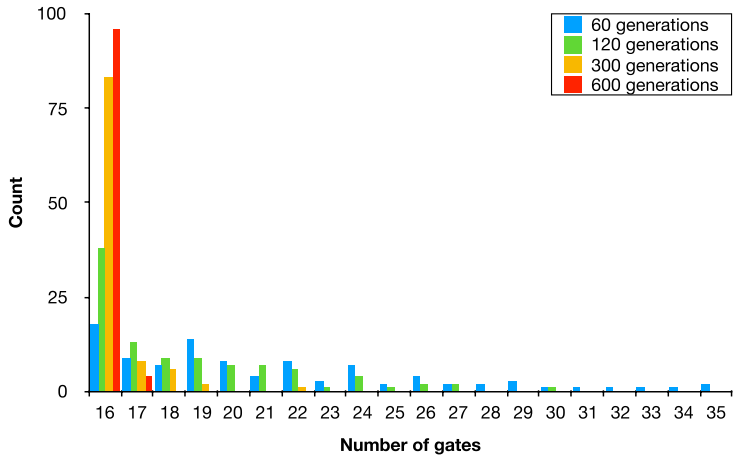


Figure 7.75: Number of runs producing best zero-error circuits of each size. (Plotting against number of gates, rather than circuit cost results in a clearer graph.) Note that, at 60 generations, one run did not produce a zero-error circuit, while another required 51 gates to produce zero error. (These are not plotted.) We see at 60 generations that the best circuit found varies in size fairly widely, while at 600 generations, almost all runs produce the optimal 16 gate circuit.

7.1.2 5 qubits

Experiments were performed again, using the same algorithm parameters, but with 5 qubits. On average, the first 60 generations took 470 seconds, while the full 600 generations required 1217 seconds. Solutions with zero error were found in 96 out of the 100 runs. The remaining four runs achieved overall errors of 0.0072, 0.038, 0.047 and 0.076. 45 of the runs found a perfect 22 gate circuit, with a circuit cost of 202. The distribution of the gate count for the best circuits of these 96 runs is shown in figure 7.76.

7.1.3 6 qubits

For 6 qubits, we performed the algorithm with the same settings, but extending the total number of generations to 1200. On average, each run took 8689 seconds, with the first 600 generations requiring 5808 seconds and the first 60 requiring 1784 seconds. Solutions with

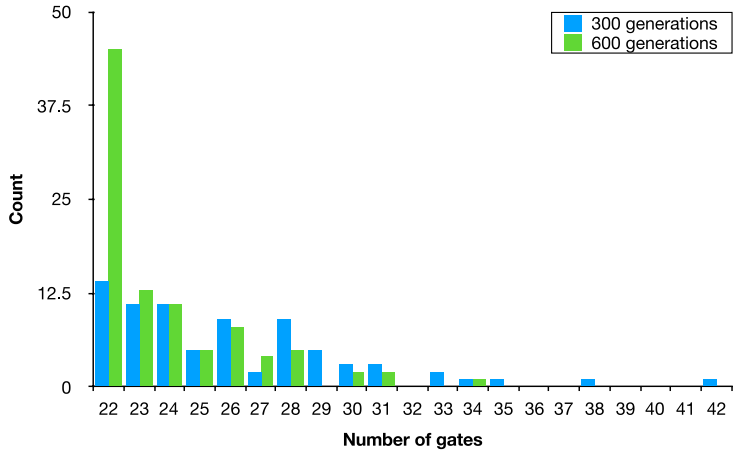


Figure 7.76: Number of runs producing best zero-error circuits of each size.

zero error were found in 84 out of the 100 runs[-5-] Of the remaining 16 runs, 4 produced overall errors of less than 0.001, a further 6 produced errors of less than 0.01. The worst run produced an overall error of 0.0767.

Of the runs that produced zero error, 27 did so using the minimum 30 gates required, giving a cost of 273. The distribution of the gate count for the best circuits of these 84 runs is shown in figure 7.77.

[-5-] Overall error values of less than 0.0001 were taken to be zero.

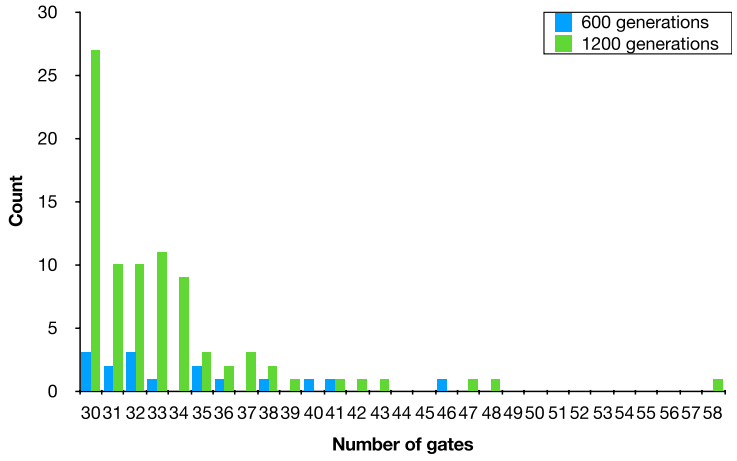


Figure 7.77: Number of runs producing best zero-error circuits of each size.

7.2 *The Grover problem*

We continue with the Grover problem, with the gate set restricted to uncontrolled XRotation gates, ArbitraryPhase gates (with arbitrary controls) and Oracles, as in Václav's experiments, but with the addition of an uncontrolled ZGate. Gate costs were chosen to be 0.9 for the ZGate, 1 for the XRotation and ArbitraryPhase gates and 50 for the Oracle[-6-]. The ZGate was added to the gate set purely to enable circuit simplifications to be found that would not be discovered otherwise[-7-]. The ZGates *do not appear* in the simplified circuits[-8-].

[-6-] Actually, costs are constrained by the code to be integers, so these were in fact set to 9, 10 and 500. Since the final circuits contain no ZGates, it makes sense to scale the results down by a factor of 10.

[-7-] After numerical optimization, an uncontrolled ArbitraryPhase gate may have just the right angle value to make it equivalent to a ZGate. The slightly lower cost of the ZGate ensures that the circuit reduction code makes this substitution. The ZGate may then be swapped with an XRotation on the same qubit, with the proviso that the angle of the XRotation is multiplied by -1.

[-8-] Since ZGates can be swapped with all the other gate types in use, they can be moved to the beginning of the circuit. Since the circuit input is always $|0\rangle$, the ZGates then have no effect on the state and thus are eliminated.

7.2.1 *3 qubits*

Using the same parameters as for the Fourier experiments, 600 generations required, on average, 22.5 seconds, with the first 60 generations taking 5.5 seconds. All runs found zero error circuits. 13 of these also minimize cost, by using only 2 Oracles and 19 gates in total, producing a cost of 1170, while 10 runs came close, scoring 1180. The quality of the best, zero-error solutions from the 100 runs is shown in figure 7.78. Note the preponderance, in figure 7.78,

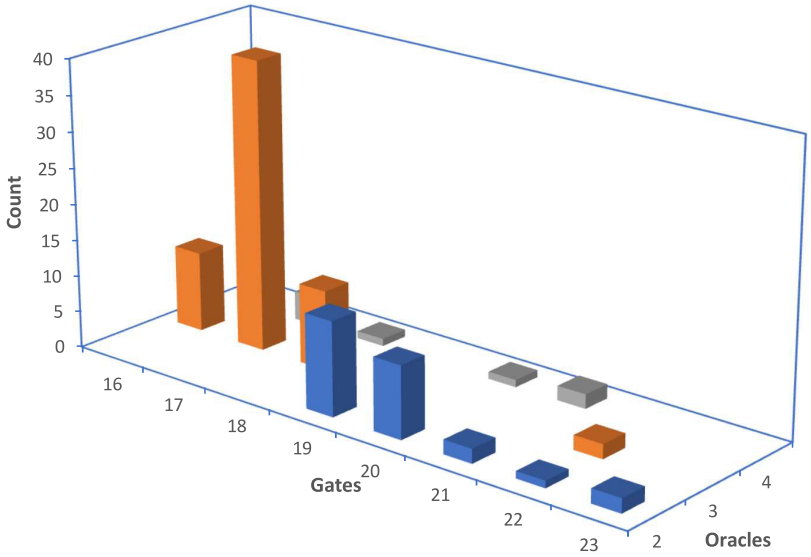


Figure 7.78: The graph indicates the number of runs for which the best circuit has the indicated total number of gates and number of Oracles. Notice how the search seems inclined to favour smaller 16 and 17 gate circuits with an extra Oracle.

of 3 Oracle, zero-error solutions that use fewer gates in total. An example of such a circuit is given in figure 7.79.

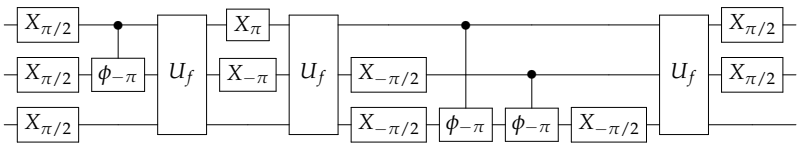


Figure 7.79: One of the shortest solution found for the 3 qubit GROVER problem. Reduction in circuit length is achieved only at the cost of an additional Oracle.

7.2.2 *4 qubits*

Using the same parameters as before, 600 generations required, on average, 658 seconds, with the first 60 generations taking 233 seconds. The increase in runtime, compared with the Fourier problem is due to the increase in the size of the circuits being manipulated, which increases both the amount of time required to simulate the circuit and the number of iterations required by the numerical optimization. Solutions with zero error were found in 98 of the 100 runs. Only three runs succeeded in also minimizing circuit cost, finding the 34 gate, 3 Oracle circuit of cost 1810. Another three runs resulted in zero cost circuits of cost 1820, i.e. incorporating one excess gate. All other runs required additional Oracles. The quality

of the best, zero-error solutions from the 100 runs is shown in figure 7.80. Figure 7.80 also reveals the algorithm's tendency to find

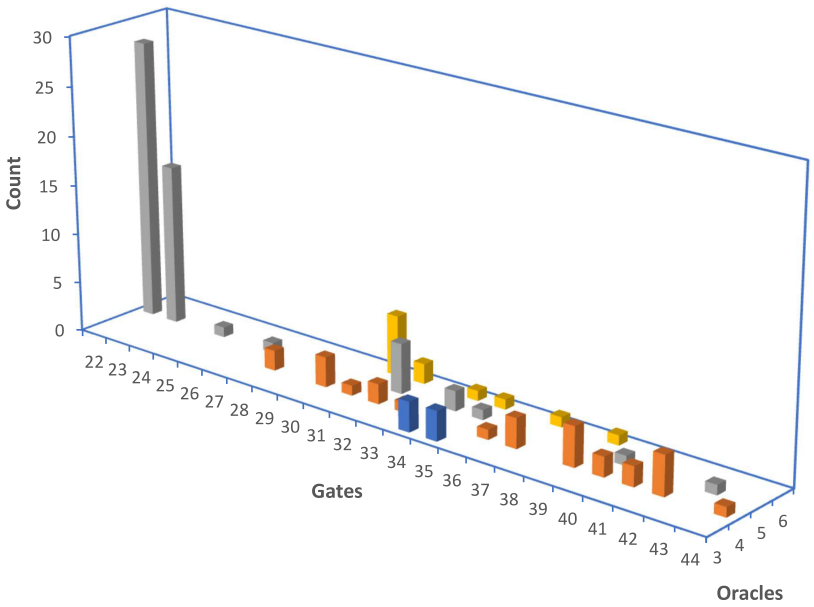


Figure 7.80: The graph indicates the number of runs for which the best circuit has the indicated total number of gates and number of Oracles. Notice how the search seems to favour the smaller 22 gate circuit that requires 5 Oracles.

the alternative 22 gate, 5 Oracle circuit that was also preferred by Václav's code. The smaller number of gates for this circuit make it relatively easy to discover compared with the optimal circuit. Moreover, its dissimilarity with the optimal solution means that its discovery cannot really be considered as a stepping stone towards the optimum.[-9-]

[-9-] I have repeated the 4 qubit Grover experiments, but using modified dominance relations, where, for A to dominate B, A must beat B by at least a certain amount — a 'leeway' — on at least one objective. Leeways of 250 for cost and 0.2 for each of the error objectives were used. However, while this resulted in 15 runs producing zero-error solutions with just 3 oracles, only two of these used the minimum of 34 gates. The 22 gate, 5 Oracle circuit was still favoured, with 34 of the runs producing this as their best solution and a further 13 producing a similar circuit with an extra gate.

7.2.3 *Crowded NSGA II*

The poor quality of the results for Grover's problem outlined above motivated some experimentation with the design of the genetic algorithm component of the algorithm. The most noticeable deficiency with NSGA II, for our work, was a loss of population diversity. This is due primarily to the way in which crowding measures are only implemented to break ties between solutions in the same front. If the non-dominated front contains relatively few unique solutions, but multiple copies of them, these copies will be transferred to the new adult population before any dominated solutions are even considered. While we could simply forbid multiple copies of the same solution from entering the population, this turns out to be insufficient to truly encourage population diversity[-10-] Instead, we forbid circuits within a user-specified distance, in objective space, from an already selected solution, from entering the adult population.

Setting this distance to 0.01, with unscaled objectives[-11-] produced the results shown in figures 7.81 and 7.82.

In the 3 qubit case, the 600 generations took, on average, 36.5 seconds, with the first 60 generations using 9.7 seconds[-12-]. All runs

[-10-] Experimentation suggests that the population, instead of containing multiple copies of the empty circuit, contains multiple single gate circuits with the same objective values.

[-11-] Crowding measures in objective space are typically implemented using scaled objectives. In our case, this would involve multiplying circuit costs by a small factor, to bring them within a range more comparable with the range of the circuit errors, i.e. zero to one. Here, however, the distance of 0.01 with unscaled objectives means that a solution can only crowd out another with the same circuit cost.

[-12-] The increase in computation time is a result of the increase in population diversity, with a greater number of larger circuits being created.

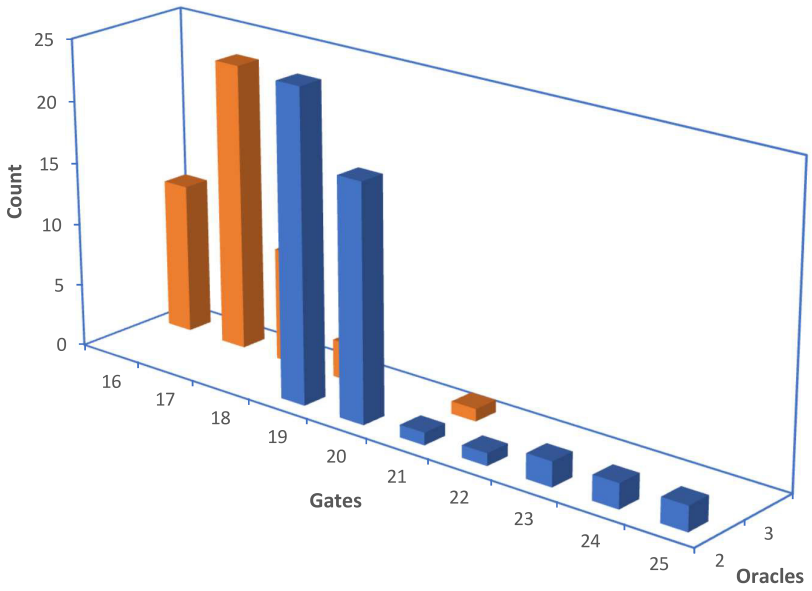


Figure 7.81: The graph indicates the number of runs for which the best circuit has the indicated total number of gates and number of Oracles.

produced zero error circuits, with 25 producing doing so using only 2 oracles and 19 gates in total. A further 19 required an extra gate, while 52 in total required just 2 oracles. In the 4 qubit case, 600 generations required, on average, 849 seconds, with the first 60 taking 355 seconds. All runs produced zero error circuits, with 9 of these requiring only 3 oracles and 34 gates. 17 runs produced zero error circuits with just 3 oracles. While this is an improvement, the search still favours the higher cost 5 oracle, 22 gate solution. We would like to do better.

7.2.4 *Cache and numerical optimization issues*

The wish to improve on these results led me to run the code 10 times again, on the 3 qubit case, and, at the end of each run, look in the cache for the optimal circuit structure. Out of the 10 runs, the optimal circuit[-13-] was found only once, while two oracle solutions were found 7 times. However, the optimal circuit *structure* was *visited* by the search in 7 of the 10 runs. It is just that, in 6 of these cases, the numerical optimization failed to find the global optimum for the gate parameter values.

Given these results, it became apparent that the way in which the solution cache was being used was inappropriate. Using the cache merely to prevent the re-running of circuit simulation and numerical optimization meant that, if the numerical optimization failed to find the global optimum for the parameters on the optimal circuit structure, it would be unlikely[-14-] to find them thereafter. We therefore switched off circuit cacheing and reran the experiments.

We subsequently also made three other changes. Firstly, we reintroduced the *mutate gate* operator, which sets the parameters of a randomly selected gate to values selected uniformly at random. This enables the algorithm to apply 'jump and reoptimize' func-

[-13-] Here, by optimal circuit, I mean the zero error circuit of minimal cost. Given the multi-objective framework we are working in here, I should probably be more careful with my phrasing.

[-14-] Initially it might seem that a failed numerical optimization run would make it *impossible* for the algorithm to find the optimal gate parameter settings later. However, there was always the chance that numerical optimization might be applied to a larger circuit, that is subsequently reduced to the optimal circuit via the reduction techniques of section **??**.

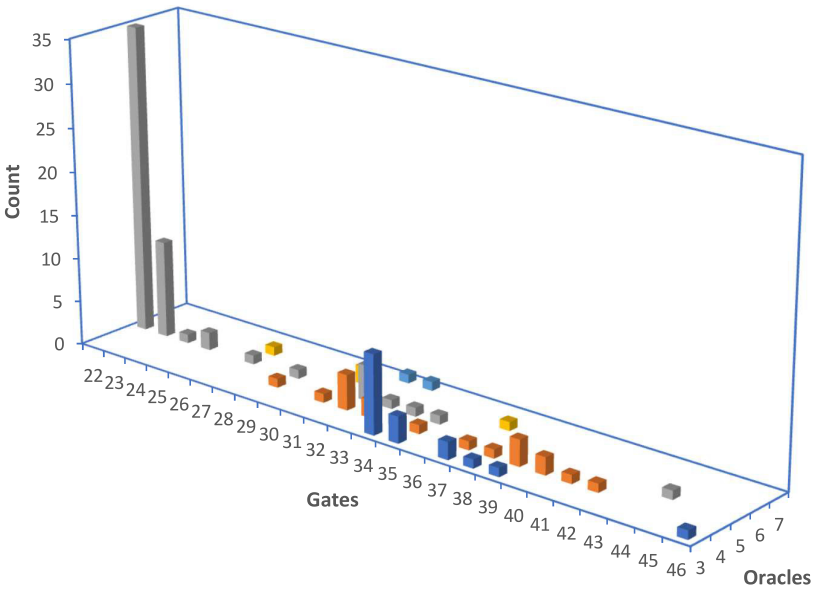


Figure 7.82: The graph indicates the number of runs for which the best circuit has the indicated total number of gates and number of Oracles.

tionality, kicking the solution out of a local optimum in parameter space in the hope of obtaining the global optimum on reoptimization.[-15-] We also increased the spacing distance for the solution survival code to 0.4. Finally, we increased the population size to 200, decreasing the total number of generations to 300 to compensate. Results are illustrated in figures 7.83 and 7.84.

[-15-] We have also experimented with switching on the randomization of all gate parameters before numerical optimization. However, results obtained were not as good. Moreover, this significantly increased the run time of the algorithm.

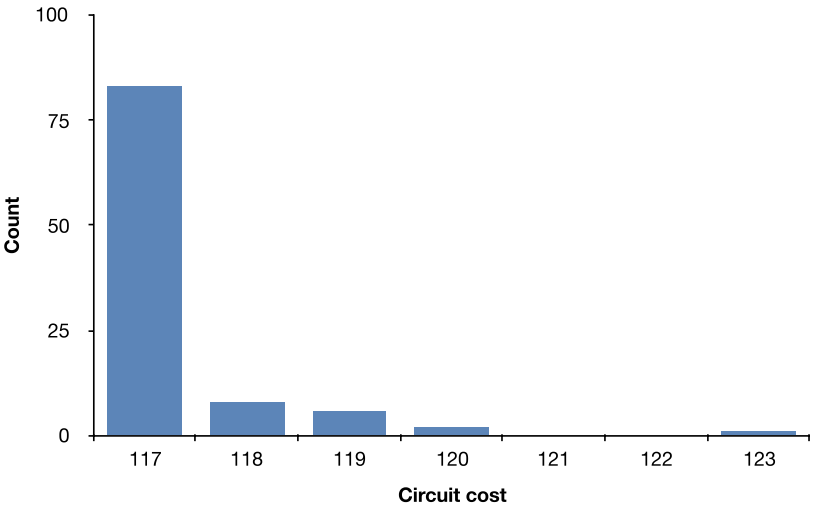


Figure 7.83: The graph indicates the number of runs for which the best circuit has the indicated total gate cost. All runs for 3 qubit Grovers produced circuits of zero cost with just two oracles.

7.2.5 *Fourier reruns*

While we did not expend excessive effort in finding the best algorithm parameter set for the Grover problem, we should be aware that any such efforts to optimize parameters for a single problem are, in a sense, cheating. If we need to experiment extensively with parameter settings for each problem we face, in order to find op-

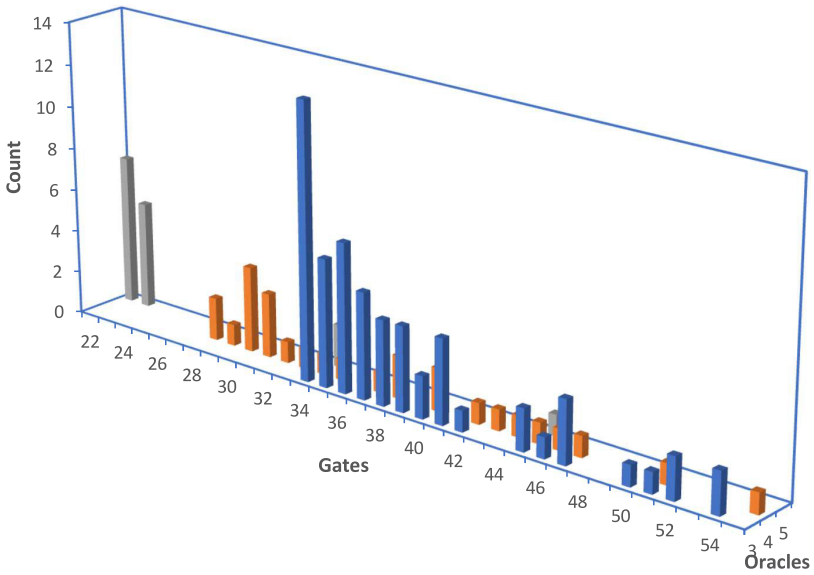


Figure 7.84: The graph indicates the number of runs for which the best circuit has the indicated total number of gates and number of Oracles.

timal solutions, then the time spent in this experimentation needs to be considered when evaluating our overarching approach. In other words, we cannot simply report the run times of the final, optimized, algorithm.

Of course, what we would like to do is find a set of algorithm parameters that work reliably on a range of different problems, with the expectation that this parameter set may work well on unseen problems.[-16-] Having spent some effort in getting the algorithm to work reasonably well on the Grover problem, it makes sense to see how well the modified algorithm, with the chosen parameter settings, works on the Fourier problem.

On the 4 qubit problem, each run took an average of 292 seconds, which is a considerable increase over the original runs. This may be due to the reapplication of numerical optimization to circuit structures that have been seen before.[-17-] The first 60 generations took, on average, 73 seconds, which is slightly quicker than previously. Of the 100 runs, all find a zero error solution with the 300 generations — indeed 30 generations was sufficient. 81 of the runs find zero error with a circuit cost of 142, i.e. they find a perfect circuit. A further 10 found a zero error circuit of cost 152, i.e. using a single additional gate. Of the 19 runs that failed to find a zero error circuit at cost of 142, 15 found circuits of cost 142 or less, with overall error less than 0.005.

Figure 7.85 shows how the zero cost solutions are reduced in size over the run of the algorithm.

On the 5 qubit problem, runs took an average of 1824 seconds, with the first 30 generations requiring 561 seconds. Zero error solutions were found on 95 out of 100 runs. The remaining runs produced overall errors of 0.00056 twice, 0.00067, 0.0014 and 0.0019. 17 runs found a perfect 22 gate circuit. The distribution of the gate count for the 95 runs is shown in figure 7.86.

[-16-] More generally, we wish to find a scheme that allows us to automatically set algorithm parameters based on easily obtainable statistics about the problem being solved, e.g. number of qubits, number of gate types, number of parameterized gate types, etc.

[-17-] This is merely conjecture at present — we would need to rerun experiments, but with the code making more notes about what it is doing.

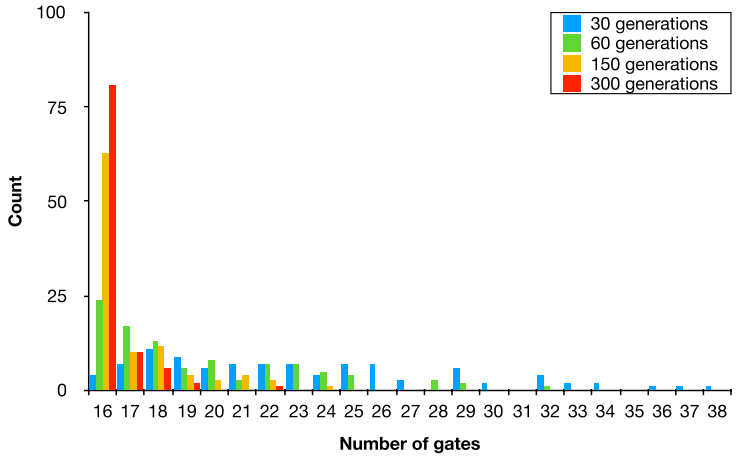


Figure 7.85: Number of runs producing best zero-error circuits of each size. (Plotting against number of gates, rather than circuit cost results in a clearer graph.) Note that, at 30 generations, two runs required 43 and 48 gates respectively — these are not plotted.

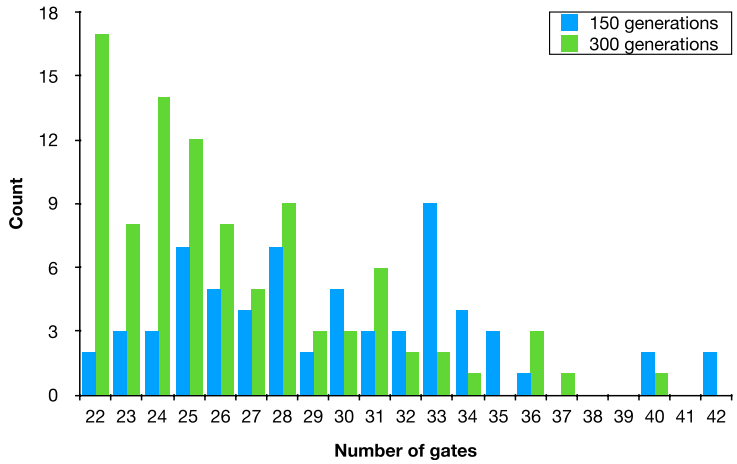


Figure 7.86: Number of runs producing best zero-error circuits of each size.

In summary, it is certainly the case that the Fourier results are degraded somewhat, when using the algorithm settings suited to the Grover problem. However, I would argue that the results are still of good quality.

7.2.6 *The Toffoli gate construction problem*

For the 3 qubit case, runs took an average of 95.1 seconds, with the first 30 generations taking 12.6 seconds. All runs produced zero error circuits with just two oracles. 83 of these runs also used the minimum of 19 gates in total. For the 4 qubit case, runs took an average of 1739 seconds, with the first 30 generations requiring 479 seconds. All runs produced zero error circuits. 58 did so using the minimum of three oracles. Of these, 13 also used only 34 gates in total, with many other runs requiring just a few more gates.

Part II

Algorithm 2

8 Circuit evaluation

The second algorithm is motivated by the existence of methods to accelerate the evaluation of child circuits. This enhancement to the speed of evaluation of child solutions is enabled by applying a little extra work on the parent solutions.

We illustrate these speed-ups using single point crossover as our genetic operator. We use matrix notation in the following[-1-], so that the application of a Gate to a State is equivalent to the application of matrix G to state vector s, i.e. $s \rightarrow Gs$.

If a circuit consists of a sequence of gates represented by the matrices A, B, C, D, E and F, then the result of applying the circuit to initial state s is

$$f = FEDCBAs.$$

In our example problems, this end state is then compared with a target state t to calculate the overlap

$$t^\dagger f = t^\dagger FEDCBAs.$$

The error measures are then determined from the overlaps. Now since the error values can be calculated solely from the overlaps, we are free to calculate these overlaps in whatever way we choose. For example, we could decide to simulate the circuit backwards, starting from the target state and applying the inverse matrices[-2-] in reverse order. This would give us the state

$$b = A^\dagger B^\dagger C^\dagger D^\dagger E^\dagger F^\dagger t.$$

We can then just take the dot produce of this with the start state, to get

$$b^\dagger s = (A^\dagger B^\dagger C^\dagger D^\dagger E^\dagger F^\dagger t)^\dagger s = t^\dagger FEDCBAs,$$

as required. We could equally well simulate forwards from the start state part way, obtaining, for example, $p = CBAs$, simulate backwards from the target state to get $q = D^\dagger E^\dagger F^\dagger t$, and perform the dot product in the middle of the circuit to get

$$q^\dagger p = (D^\dagger E^\dagger F^\dagger t)^\dagger CBAs = t^\dagger FEDCBA$$

as required.

Now consider the application of single point crossover to two circuits. The first is that described above, with gates represented by A–F. Let the second consist of a sequence of gates represented by

[-1-] The use of matrix notation can lead us to over-estimate the computational cost of applying a gate. It would appear that, given n basis states, this requires n^2 multiplications and additions. Since, in our example problems, each circuit must also be evaluated on n different inputs, this means a total of n^3 multiplications and additions. However, in reality the application of a SingleTargetGate to a state takes about $2n$ multiplications and additions in QIClib (and our new replacement code), resulting in $2n^2$ multiplications and additions in total, due to the special structure of the matrix.

[-2-] Note that all matrices are, of course, unitary, so that, for example, $E^{-1} = E^\dagger$.

matrices V, W, X, Y and Z. A child circuit, created by single point crossover might consist of those gates represented by A, B, C, D, X, Y and Z. A full simulation would require us to multiply each input state by each of these gates in turn, before calculating the overlaps required to determine the error. This requires $(2\ell + 1)n^2$ multiplications, where ℓ is the length of the circuit. But if we look at the form of a typical overlap, i.e.

$$t^\dagger ZYXDCBAs,$$

we note that we have done some of this work before. We already calculated the state $DCBAs$ when evaluating the first parent. Furthermore, if we had simulated the second parent *in reverse* we would also have calculated $X^\dagger Y^\dagger Z^\dagger t$. Taking the dot product of these states gives us

$$(X^\dagger Y^\dagger Z^\dagger t)^\dagger DCBAs = t^\dagger ZYXDCBAs,$$

i.e. our desired overlap. Moreover, to calculate all the overlaps in this way requires only n^2 multiplications. This is a significant improvement[-3-].

We can therefore make the evaluation of child solutions considerably more efficient, provided we simulate the parent solutions in *both* directions and store each of the intermediate states. Of course, doubling the cost of evaluating adult solutions makes little sense if we only generate an equal number of child solutions from them, but if we generate large numbers of child solutions from relatively few parents, this provides a mechanism whereby we can greatly improve the efficiency of the genetic algorithm.

[-3-] This also goes some way to assuage our fear of large circuits, since the computational complexity of solution evaluation no longer depends on circuit length. Of course, we also have simplification to consider.

9 Genetic algorithm

Our first algorithm applies an off-the-shelf multi-objective genetic algorithm — NSGA II —, albeit with some modifications to the dominance relations to encourage population diversity. As such, the description of the genetic algorithm, provided by chapter 2, mainly described the representation of circuits, including a description of the circuit context, the gate types supported and the mutation operations. In contrast, NSGA II was found to be unsuitable for our second approach, so a new, less elitist multi-objective genetic algorithm was designed. Since much of the details regarding circuit representation, gate types, circuit context, etc. is unchanged from the first algorithm, the description of the new GA comprises the majority of this chapter.

9.1 Genetic operators

Before describing this genetic algorithm in more detail, we note that the genetic operators for the second algorithm are different from those of the first. Genetic operators are selected to maximize the exploitation of the intermediate states stored in the parent solutions, to accelerate the evaluation of the children. Each of the currently available operators can be considered as consisting of two phases, a *cut* phase and an (optional) insertion phase.

The cut options are

Cut: Take a single parent and cut at a random point, without inserting or deleting anything.

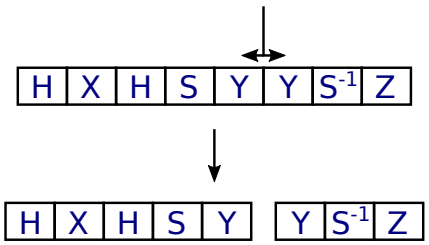


Figure 9.87: Cutting the parent solution. In this and following diagrams, the gap in the final solution indicates the location of any subsequent insertions. Note that the cut point can also be chosen to be either of the two ends of the gate sequence.

Remove gate: Remove a single gate, selected at random.[-1-]

Remove sequence: After choosing two cut points in a single parent at random, remove the sequence of gates from between the cuts.

[-1-] While the remove sequence option can sometimes remove a single gate, we may wish to permit only small removals, or encourage such removals, in the hope that less drastic changes to the circuit are more likely to be accepted by the algorithm.

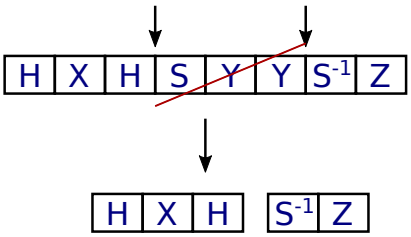


Figure 9.88: Removing a gate subsequence. Note that the removal can also take place at either end of the solution.

Duplicate sequence: After choosing two cut points at random, duplicate the sequence between the cuts. If the parent solution had three subsequences, A, B and C, with B being that between the cut, then the new solution is ABBC. Any subsequent insertion takes place between the two copies of B.

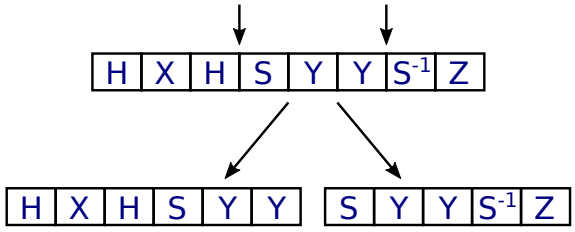


Figure 9.89: Duplicating a subsequence. Subsequent insertions occur between the two copies of the duplicated sequence. Again, the duplicated sequence may occur at either end of the original solution.

Single point crossover: Given two parents, choose cut points in each at random, The new solution then consists of the gates to the left of the cut from the first solution, followed by the gates to the right of the cut from the second. Any subsequence insertion takes place between the two selected subsections.

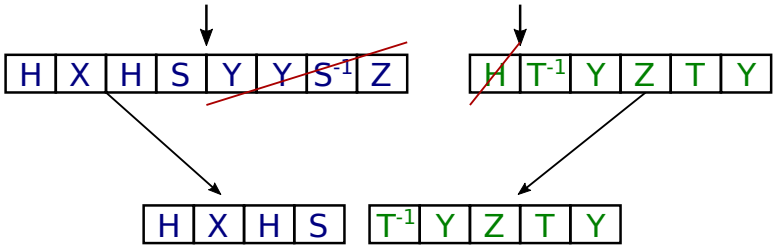


Figure 9.90: Single point crossover. In order to ensure that some of each parent solution is included in the child, the cut point for the first parent must occur after the first gate, while the cut for the second parent must occur before the last. (An obvious exception to this rule is if either parent is the empty circuit.)

Insertion options are

None: This simply reattaches the gate sequence at the insertion point.

One: Insert a single, randomly selected gate.

Many: Insert a pre-selected sequence of gates.

Ignoring the combination of merely cutting the gate sequence and then inserting nothing, which clearly does not change the solution, this gives 14 different genetic operators. There is one additional genetic operator.

Tweak gate: Remove a parameterized gate, then reinsert at the same location.[-2-]

[-2-] This operation is implemented using the same cut and insert approach of the others. Hence cut points are determined as for 'remove gate', and an insertion is defined much like when inserting a new randomly selected gate, but without the randomness.

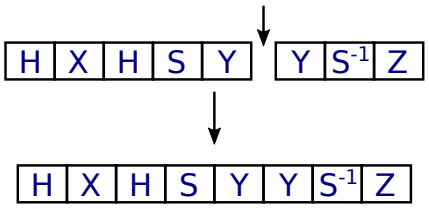


Figure 9.91: Inserting nothing.

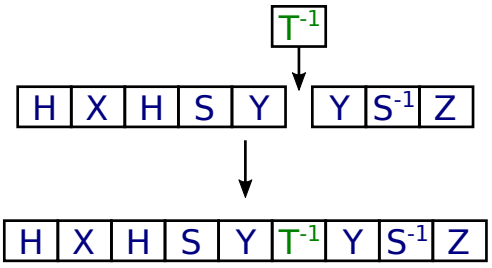


Figure 9.92: Inserting a single gate.

This final option might appear to do nothing. However, we will see in chapter 12 that whenever a single parameterized gate is inserted into a solution, its parameter is either optimized or mutated. Repeated application of this operator is expected to optimize all such gate parameters[-3-]

Note that the computational cost of the genetic operator depends primarily on the choice of insertion. If no insertion is made, then, for a given input and target state, the overlap for the new solution can be calculated as a simple dot product of intermediate states stored in the parent solutions. If there are n basis states and n different inputs to be tested, this requires n^2 multiplications and additions. If a single gate is inserted, one needs to add the cost of applying the gate in addition to the cost of the dot product. For the typical SingleTargetGate this means an additional $2n^2$ multiplications and additions, making $3n^2$ in total.

For the insertion of a sequence of gates, the computational cost depends on how this is implemented. The plan was to generate a small set of short random[-4-] sequences, referred to as 'insertions', at the beginning of each generation of the algorithm. These would be pre-simulated — that is, the transformation matrix for the insertion would be calculated in advance of child generation, to prevent the task being repeated whenever a child solution with the insertion is evaluated[-5-]. Since QIClib provides a function for applying a matrix over a set of qubits, this matrix could be simply applied to a state.

The initial attempt at this pre-simulation approach turned out

[-3-] The mutation option is included in the hope that it will allow the algorithm to escape from local minima (in gate parameter space) or, as noted in chapter 12, from tricky saddle points.

[-4-] Choosing a random subsequence of a random solution in the population would be an alternative way to produce some of these insertions, though I have not yet tried this.

[-5-] The Oracle results in some complication to this approach, since the overall matrix will depend on which state is marked by the Oracle and this will change for different inputs. Oracle gates are handled by requiring insertions to have at most one Oracle, which must occur last in the gate sequence if present. Application of the insertion then consists of application of the pre-calculated matrix for the sequence of non-Oracle gates, followed by application of the Oracle.

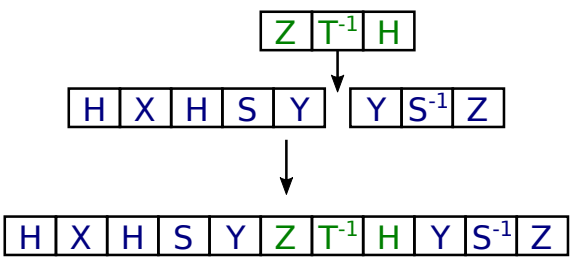


Figure 9.93: Inserting a preselected gate sequence.

often to be less efficient than simply simulating each gate in turn upon child evaluation. The high computational costs[-6-] of applying the transformation matrix resulted in us only using this approach when the insertion contained more than a certain number of gates[-7-]. Otherwise, the code simply applied the gates individually in the usual way.

These efficiency issues resulted from QIClib's representation of the transformation and its implementation of matrix multiplication. QIClib uses a 'dense' representation, where each row of the matrix is simply a list of complex numbers. Given an insertion consisting of only a few gates, the resulting transformation is actually sparse, and so a sparse matrix representation is more appropriate. Having replaced QIClib with our own simulation code and implemented simulators that use sparse matrix multiplication[-8-], pre-simulation and application of the resulting sparse transformation matrix is now performed for all insertions of gate sequences.

Before leaving the subject of genetic operators, note that not only are the genetic operators different for the second algorithm but the way in which they are applied is too. The first algorithm applied genetic operators independently. Each had a fixed probability of being applied, meaning that a single child solution could be the result of crossover and many mutations. This is clearly undesirable for this second algorithm, since the efficiency improvements can only be guaranteed to work if only a single genetic operator from the eleven options is selected[-9-].

To fit within the genetic algorithm framework, we use the code's crossover and mutation functions on *all* child solutions, but note that these functions no longer do quite what one might expect. Child solutions undergo the following steps in sequence:

Creation: A child solution is created from a parent not by copying the parent's gate sequence, but by creating an empty solution with a reference to the parent.

'Crossover': The crossover function (which is always applied) merely adds a reference to a second parent.

'Mutation': The mutation function then tells the child to forget one parent, if crossover is not to be applied. In either case, it then selects cut points in the parents.

The child solution now has enough information to allow it to be evaluated. Note that, thus far, the circuit has not actually been constructed! The copying, cutting, crossover and sequence insertion required to construct the child circuit is only performed after evaluation[-10-].

9.2 *Parent selection and solution survival*

Two considerations led us to choose different approaches to parent selection and solution survival than those taken in the first

[-6-] Given n basis states and n inputs, the matrix implementing the insertion requires n^3 multiplications over the input set.

[-7-] For a 3 qubit problem, an insertion was only pre-simulated if it contains at least four gates. For a 4 qubit problem, this increased to ten gates. For 5 or more qubits, we assumed that the insertion is always short enough to make the application of the pre-simulated matrix inefficient

[-8-] See chapter 11.

[-9-] Yes, some of these options can be thought of as a combination of crossover and mutation. However, here we consider any one of the eleven operators to be a *single* operator.

[-10-] We briefly considered only performing the construction of the circuit if the solution was of sufficient quality. However, this raised complications that outweighed the minor efficiency improvement obtained by eliminating unnecessary circuit construction.

algorithm. Firstly, the survival method needed to be able to work efficiently with large populations. The increased efficiency of circuit evaluation, combined with the need to produce large numbers of child solutions in order to exploit these efficiency improvements, meant that standard methods like that of NSGA II were inappropriate. Secondly, results from experiments with the first algorithm suggested that less emphasis on elitism, and greater focus on maintaining population diversity, would produce better results. This requirement for less elitism turns out to be useful, since it means that it is not necessary to compare all pairs of solutions to find non-dominated fronts — a source of computational expense in NSGA II.

Selection of parents from the adult population is performed uniformly at random, with the proviso that the two parents of a child solution should be different solutions. This is clearly a computationally inexpensive approach, but means that the task of driving the search towards better solutions is solely born by the survival method. The code that determes which solutions should survive to the next generation has three components.

Crowding: Adding a solution to the new adult population prevents the survival of any other solution within a user specified distance[-11-] in objective space.

Elitism: The first m solutions selected for survival, where m is a user specified parameter, are those that are best according to the primary objective[-12-] — in our examples, the overall error.[-13-]

Trials: Finally, candidates are selected at random from the set of unselected solutions to undergo a number of trials, specified by the user. Each trial consists of the selection of a guard solution[-14-] with which it is compared. If the guard dominates the candidate, then the candidate fails the test. If none of the guards dominate the candidate, the candidate is selected for survival.

The survival method described above has evolved to this state over some time. Initially, the notion was just to use the trials to find survivors. However, without some form of crowding mechanism, the adult population quite rapidly fills with multiple copies of the empty circuit, which naturally, given its zero cost, is never dominated. Without the elitist component, one finds that the circuits with lowest error, which tend to be large early in the search and are found comparatively rarely, drop out of the population (by virtue of not being selected as a candidate), while smaller, more frequently generated solutions remain. Further experimentation with the survival class may be of use in future, but for now, the method described above seems to perform well enough.

[-11-] This distance is calculated according to the Manhattan metric, though I expect the choice of metric makes little difference. If, during this process, it is determined that all remaining solutions are crowded out, then the crowding distance is halved. (A small number (10^{-6}) is also substracted from the crowding distance, to ensure that it reaches zero in a finite number of such steps. This handles the rare case that all remaining solutions are crowded out because they are identical to already selected solutions.)

[-12-] We actually add 10^{-5} times the gate cost of the circuit to the overall error. This ensures that if two circuits have near identical error (due to numerical error), the smaller circuit is preferred.

[-13-] This is subject to the crowding rule. Hence, if $m = 2$ and the second best solution is too close to the first in objective space, the third best is considered, and so on, until m solutions are selected.

Note that prioritizing a single objective in this way breaks the pure multi-objective nature of the algorithm. We argue that, early in the search, the need to find a small circuit is outweighed by the need to find a circuit with low error — a circuit that actually comes close to doing the job required of it. Emphasizing small circuits early in the search can prevent the generation of circuits large enough to produce the low errors required.

[-14-] At present, the guard is also chosen from the set of currently unselected solutions. Previously , the guard could have been *any* of the solutions. I am not sure which approach is best — I suspect it makes little difference.

10 Circuit Simplification

While techniques such as numerical optimization and solution cacheing have been removed from the second algorithm, circuit simplification still plays a major role, albeit on a reduced selection of gates. However, the way circuit simplification is incorporated into the overall algorithm has been altered.

10.1 Simplification and the efficiency of circuit evaluation

The primary role of circuit simplification in both algorithms is the reduction of circuit gate cost — one of the objectives being minimized — without the need to perform additional circuit simulations. However, in the first algorithm there is a secondary benefit for circuit simplification: improved efficiency of circuit structure evaluation.

Recall that, in the first algorithm, all circuits are evaluated by performing full simulations of the circuit. Hence the cost of circuit evaluation increases linearly with the size of the circuit. Moreover, the evaluation of a circuit *structure* involves the application of numerical optimization — a computationally expensive task given the number of circuit simulations that might be required. The reduction in the size of the circuit that typically results from circuit simplification can therefore reduce the computational cost of circuit evaluation and numerical optimization. Therefore, circuit simplification is performed first.

Note also that knowledge of the cost of the simplified circuit is also used to determine when to halt the numerical optimization. If the circuit is costly, it makes sense to halt numerical optimization if it does not look likely to produce very good error values. If a perfect circuit, i.e. a circuit with zero error value, or an error value better than some user-provided target, has already been found at a given circuit cost, it may make sense to entirely forgo numerical optimization on more costly circuits. While the post-simplification cost of the circuit may not be the ultimate cost of the circuit, due to possible circuit reductions after numerical optimization, it does provide a better estimate of this ultimate cost than the cost of the unsimplified circuit.

In contrast, in the second algorithm, evaluation of child circuits requires the simulation of a few gates at most, in the case where a small sequence of gates is inserted. For simple crossover or gate

sequence removal, only a dot product is required, of states already stored in the parent solutions. As such, the cost of child evaluation does not increase with circuit length. The motive for simplifying circuits prior to evaluation is therefore eliminated. Moreover, the secret to the faster evaluation of child circuits is the similarity of the child circuit to its parents. The first chunk of the circuit is identical to that of one parent, the final chunk is identical to that of the other (or the same) parent, while the insertion is just a few gates. Simplifying the circuit may significantly alter the child solution. As a result, if we were to attempt to evaluate the *simplified* child circuit, we would not be able to exploit the information stored in the parents.

10.2 Forgoing simplification

The algorithm should still perform circuit simplification at some point, in order to cheaply reduce circuit cost. However, it we delay child circuit simplification until *after* circuit evaluation, we can use the newly calculated error values to decide whether simplification is worthwhile. For circuits with poor error values, we can choose to forgo circuit simplification.

For small problems (e.g. 4 qubits), this can result in useful speed-up of the algorithm. The improved efficiency of circuit evaluation associated with the second algorithm's approach means that, if all circuits are simplified, circuit simplification takes about 36% of the time taken. By only simplifying those circuits that obtain an overall error of less than 0.7 (on Grover's problem), this is reduced to about 5%.

The method used by the algorithm to determine whether a solution is good enough to warrant simplification is precisely that implied by the previous paragraph. The user enters a value of overall error that marks the cut-off between circuits good enough to warrant simplification and those of lower quality. It is easy enough to imagine more sophisticated schemes. For example, we might use the quality of the best solutions found thus far to determine what the cut-off should be, with different values of the cut-off for different values of circuit cost. This would have two advantages. First, one could reduce further the number of circuit simplifications performed, by using a stricter cost-dependent cut-off[-1-]. Second, the ability to adapt the cut-off according to circuit cost, ensures that small, good-for-size, solutions are still simplified, while the simple fixed cut-off runs the risk that the best small circuits are considered 'bad'[-2-].

[-1-] Given that the simple approach already reduces the computational cost of simplification to only 5% of the total cost, this is not a high priority.

[-2-] For the example of Grover's problem, one is unlikely to be that interested in solutions with an overall error worse than 0.7, however, small the circuit is! Hence, there is little motivation for working on improving the cut-off method at present.

10.3 Circuit reduction

Since the second algorithm, at present, does not handle parameterized gates, the circuit reduction methods described for the first algorithm are not applied. However, it is likely that parameterized

gates might be reintroduced in the future. Moreover, numerical optimization might also be reintroduced, though applied differently. If so, the inclusion of circuit reduction will also be reconsidered.

11 Circuit simulation and gate simulators

For reasons discussed in the following sections, we decided to drop the use of QIClib to simulate our circuits, preferring to use our own gate simulation routines. The following sections also describe these new routines and the gate simulator classes used to expedite them.

11.1 Why drop QIClib?

While the efficiency of the routines used for gate simulation is obviously important, there are other considerations. We have already seen, in our description of the numerical optimization component of algorithm 1 in chapter 4, that there is a need to be able to extend the functionality of the simulation code. Ease of use and ease of code maintenance are also important.

The following is a list of some[-1-] of the reasons for dropping QIClib in favour of our own simulation routines.

[-1-] I wrote the heading of this section prior to the end of November 2019. It is now October 2020. Hence I may have forgotten some additional reasons for dropping QIClib.

1. We needed to be able to calculate derivatives in algorithm 1, which required functionality not supplied by QIClib.
2. We wanted to be able to use sparse matrix arithmetic, or at least look into the possibility of obtaining efficiency improvements in this manner.[-2-]
3. QIClib sets up data structures from scratch every time a gate is simulated. Since an individual gate might be simulated multiple times, it makes sense to only set up these structures once, if possible.
4. While our code indexes arrays using a zero-based system, where zero represents the least significant qubit, QIClib uses a one-based system, with one representing the most significant qubit. The conversion between the two different systems resulted in unnecessary complication and headache.
5. While QIClib uses Armadillo[-3-] data structures, the gate simulation code does not use any of the library's linear algebra routines, but performs the required matrix multiplication manually. It is a simple matter to write the simulation code in a manner that does not require the conversion between standard C++ data types and Armadillo types.
6. The QIClib code is almost entirely lacking in comments.

[-2-] This was of particular interest when considering the insertion of a gate sequence into potentially many different circuits, where the matrix for the sequence could be calculated in advanced and stored. However, sparse matrix arithmetic has proven useful even for single gate insertions.

[-3-] A C++ linear algebra package

Given the requirement to rebuild data structures on each gate simulation, and lack of sparse matrix support, we were also concerned about the efficiency of the routines. It turns out that, while our own simulation code is faster than that of QIClib, the efficiency improvement is not that great[-4-]. However, the additional benefits has made the decision to drop QIClib worthwhile.

(The QIClib code is more general in one respect — it can handle qutrits and other generalizations of qubits. Our code only handles qubits. However, it is quite general in other respects, for example, handling multi-target gates.[-5-])

[-4-] As a result, we suspect that it ought to be possible to further improve the efficiency of this simulation code.

[-5-] Obtaining more speed by writing code specialized to single target gates is a possibility.

11.2 *Gate simulators*

As noted above, a gate may appear, unchanged, in a sequence of solutions and hence may need to be simulated multiple times. As such, it made sense to associate with each gate a Simulator object that would not only manage the simulation of the gate, but also manage the data structures associated with gate simulation. This allows the code to initialize these data structures just once but to simulate the gate many times.

We will first consider the construction of such simulators for SingleTargetGates. SwapGates and Oracles are handled differently. Insertions of short gate sequences also are provided with Simulator objects — this case is discussed after the simulation of single gates.

11.2.1 *Single target gates*

Once we began implementation of the `GateSimulator` class, it became apparent that such a class should be split into two. While most of the properties of a Gate remain constant, such as the target qubit[-6-], control qubits etc., the angle parameter might be changed during gate parameter optimization or mutation. The data structure used for simulation that are unchanged by mutation are collected in the SimulatorIndexManager class. The rest[-7-] remains in the GateSimulator class.

[-6-] Gate mutation, in our code, only refers to the modification of the angle parameter in a parameterized gate.

[-7-] Just the 2×2 matrix and a flag. (See following sidenote.) Perhaps, given the smallness of this part, it could be calculated before each simulation.

Single target gates also store *two* simulators, one for forward simulation and the other for reverse simulation, which applies the inverse matrix. These two simulators can share the same SimulatorIndexManager object.

We will consider a 5 qubit problem. Gate simulation requires us to calculate the new state, which consists of a list of coefficients of all the basis states, from the old. While conceptually this is just a single matrix multiplication, it makes little sense to be manipulating 32×32 matrices to implement a simple controlled NotGate. Our code therefore handles just the 2×2 matrices for SingleTargetGates (or $2^t \times 2^t$ matrices for gates with t targets). Each output coefficient is considered in turn, where each coefficient is indexed from 0 to 31. First, using the SimulatorIndexManager, the code determines whether the index corresponds with a basis state where all the

control qubits are set. If not, the value for the output coefficient is merely copied from the input coefficient[-8-]. If all control qubits are set, then the SimulatorIndexManager's member function `row_index` is used to convert the index of the output coefficient into the appropriate row index of the matrix. Then, for each element in the row, the code must figure out which coefficient in the input state it should multiply. This is done by passing the column index[-9-] and the output coefficient index[-10-] to the SimulatorIndexManager's member function `input_index`, which provides the appropriate index into the input state. Then we simply sum the products obtained.

The SimulatorIndexManager performs its tasks by referring to a number of stored bitstrings. Supplied with a list of the target qubits (and a list of the controls), a bitstring, called `in_mask`[-11-], is created with ones for each of the target qubits and another, called `out_mask`, is created with ones for each *non*-target qubit. Bitstrings are also created for each column of the matrix. These are set to zero for each non-target qubit, and to zero or one for the target qubits, depending on the column index. So, for the 5 qubit problem, if the gate in question has two target qubits, 2 and 0, then column zero of the matrix gets the bitstring 00000, column one gets 00100, column two gets 10000 and column three gets 10100. In detail, the column index (from zero to three) is converted to binary and the values of the bits are transferred to the target qubits, qubits 2 and 0, in that order.

Now the input coefficients that affect the value of a particular output coefficient are those that have the same values for the non-target qubits, and all possible combinations of values for the target qubits. These are obtained by taking the output coefficient index (in binary), using the mask to set all target qubits to zero, then using bitwise OR with each of the column bitstrings. This allows us to find the input coefficient that, when multiplied by the matrix element in a specified column, contributes to the specified output coefficient.

[-8-] This is the behaviour we want for a simple simulation of a gate. However, there are times when we want different behaviour — we may wish to set the output coefficient to zero. This happens when we have a parameterized gate, with unspecified angle parameter, and wish to separate out the constant, $\cos\theta$ and $\sin\theta$ parts, prior to optimization of the gate parameter. One part (typically the constant part) is simulated using the usual behaviour, while the other parts are simulated using the alternative. If we were to apply our simulation code to algorithm 1, the calculation of derivatives would also require this alternative behaviour. Which behaviour is required is denoted by the flag briefly mentioned above.

[-9-] Which provides the values of the target qubits.

[-10-] Which provides the values of the non-target qubits.

[-11-] Possibly not necessary.

11.2.2 *Alternative implementations*

The code includes two alternative implementations of the gate simulators, one for dense matrices and the other for sparse. Both work and interact with the SimulatorIndexManager class in the same way, the only difference being how the rows of the gate matrix are stored[-12-]. Naturally, the DenseGateSimulator and SparseGateSimulator classes inherit from a base GateSimulator class.

At present, all single target gates use the sparse gate simulators. Even though the transformation matrices are only 2×2, this still provides a small speed boost.

[-12-] The dense representation is simply a vector of complex numbers, while the sparse representation is a vector (being used as a list) of pairs, with each pair consisting of the column number and the complex number at that location. Locations not lists are assumed to contain zero.

11.2.3 *Other gates*

The code also includes a SwapSimulator class to simulate swap gates. Unlike with the single target gates, swap gates do not create their own simulators. Instead, a SwapSimulator is created whenever a swap gate is simulated. Note, though, that the SwapSimulator does not need to store a bunch of bitstrings. It just contains the identity of the qubits being swapped (and the total number of qubits) and so is relatively inexpensive to create.

The Oracle gate does not have an associated simulator class. So while `SingleTargetGate::applyTo(State& state)` passes the GateSimulator and SimulatorIndexManager to the State's `transform` function, which simply applies the simulator, the function `Oracle::applyTo(State& state)` simply calls the State's `mark` function directly.

Each of the concrete Rotation gate classes[-13-] also contain SparseGateSimulators that simulate just the constant, $\cos\theta$ and $\sin\theta$ parts of the gate matrix. This enables the symbolic computation of the overall error, which allows us to perform algebraic or numerical optimization of the gate parameter. Note that, while the constant part is simulated in the usual way, the other parts handle controls differently, as noted above.

[-13-] Note that it is the *class* that contains the partial simulators, not each individual gate object, since these will be the same for *all* gates of the specified class.

11.2.4 *Gate sequence insertions*

As described in section 9.1, a few gate sequences for insertion are generated *and pre-simulated* every generation, prior to the generation of child solutions. Hence each such Insertion contains a single GateSimulator[-14-], for forward simulation, and the accompanying SimulatorIndexManager. In this case, the use of a SparseGateSimulator is considerably more important, as noted in section 9.1, since the stored matrix is $2^n \times 2^n$, where n is the number of qubits, but typically very sparse, especially when the insertion is only of two gates.

[-14-] Yes, the name of the class is perhaps not ideal — I am considering dropping 'Gate' from the name.

12 Gate parameter optimization

While the second optimization algorithm was initially designed with parameterless quantum gates in mind, one can still use gates with angle parameters. Of course, this raises the question of whether there are more effective ways of optimizing the gate parameters than simply letting the GA wander blindly. Full numerical optimization of the form used in the first algorithm clearly doesn't fit with the ethos of this second approach: L-BFGS modifies multiple gate parameters, from multiple gates, simultaneously, requiring full evaluation of the resulting solutions. However, optimizing a *single* gate's parameter(s) is a possibility, since changing a single gate allows us to continue to use the information from the evaluation of the parent to greatly speed up the evaluation of the child. Hence we may apply simple numerical optimization, or, with a bit of luck, algebraic optimization.

12.1 The algebraic form of the overall error

In each of our problems, the 'overall error' is calculated from the 'overlaps' of the target state and the state obtained from each of the simulations of the circuit. Each of the parameterized gates can be written such that the matrix elements are linear combinations of $\cos\theta$, $\sin\theta$ and 1, where θ is either the angle parameter for the ArbitraryPhase gate, or half the angle parameter for the other rotation gates. Therefore, if we leave one of the parameters unspecified, the overlap obtained is also such a linear combination.

The overall error is different for different problems. Recall that if the output of the circuit is denoted by $|\psi\rangle$ and the desired target by $|\chi\rangle$, then the overall error for the Fourier problem is given by

$$1 - \left| \sum_{i=0}^{n-1} \langle \psi_i | \chi_i \rangle \right| / n, \tag{12.1}$$

while for the Grover problem, it is

$$\sum_{i=0}^{n-1} \left(1 - |\langle \psi_i | \chi_i \rangle|^2 \right) / n,$$

where n is the number of simulations performed[-1-] and i indicates the run number. The first of these requires the calculation of an annoying square root, but minimizing 12.1 can easily be seen to be

[-1-] I.e. the number of basis states.

equivalent to minimizing

$$-\left|\sum_{i=0}^{n-1}\langle\psi_i|\chi_i\rangle\right|^*\left|\sum_{i=0}^{n-1}\langle\psi_i|\chi_i\rangle\right|.$$

Now both objectives are seen to be linear combinations of 1, $\cos\theta$, $\sin\theta$, $\cos^2\theta$, $\cos\theta\sin\theta$ and $\sin^2\theta$.

Now the $\cos^2\theta$ and $\sin^2\theta$ terms can be converted to linear combinations of 1 and $\cos 2\theta$, while the $\sin\theta\cos\theta$ converts to a $\sin 2\theta$ term. Finally, we can convert $a\cos\theta + b\sin\theta$ to $A\sin(\theta+\phi)$ for some value of A and ϕ, and similar for $a\cos 2\theta + b\sin 2\theta$. The result is that each objective can be written in the form

$$A\sin(\theta+\phi) + B\sin(2\theta+\chi),$$

where θ is related to the gate parameter and A, B, ϕ and χ are fixed.

12.2 *Numerical optimization*

Now note that, in the experiments we have performed thus far, we find that either A or B is always zero. As a result, finding θ that minimizes the objective is a simple matter, requiring no numerical optimization at all. However, we have implemented code to handle the case where neither A nor B is zero. This is a straightforward application of Newton's method in one dimension. The only challenge is to find a suitable start point for this method — a location where the second derivative of the objective function is positive, preferably close to the minimum.

In figure 12.94 we have plotted the two harmonic components

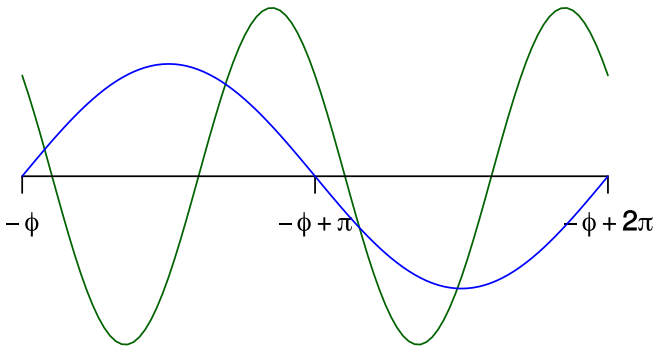


Figure 12.94: The two harmonic components of the error function, plotted against angle parameter θ.

of the error function to be minimized, over a full cycle of angle parameter θ. We have arranged for the slow component to start at zero, by starting the cycle at $\theta = -\phi$. We refer to the two components as the fast component, in green, and the slow component, in blue. As the fast component merely repeats itself in the second half of the graph, the minimum must be in the half of the graph where the slow component is negative. Hence, if $A > 0$, we have $\theta^* \in [-\phi+\pi, -\phi+2\pi]$, where θ^* is the optimal value for θ, while if $A < 0$, we have $\theta^* \in [-\phi, -\phi+\pi]$.

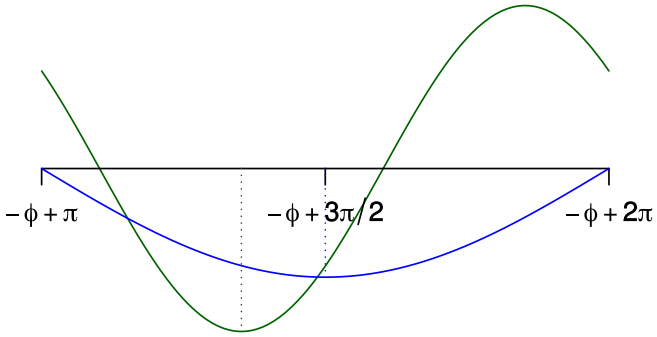


Figure 12.95: Zooming in to the selected half where the slow component is negative.

Zooming in to the selected half, as in figure 12.95, we note that the slow component has only half of a cycle, while the fast component goes through an entire cycle. Suppose the minimum of the fast component occurs early in this half, as illustrated. Then the overall minimum cannot occur to the left of this minimum, as both components will have larger (more positive) values. The minimum also cannot lie between $\theta = -\phi + 3\pi/2$ and the maximum of the fast component, as both components have larger values that those at $\theta = -\phi + 3\pi/2$. Finally, the minimum cannot occur after the maximum of the fast component. To see this, consider a lesser value for θ equidistant from this maximum. It has the same value for the fast component, but a lesser value for the slow.

Hence the overall minimum must lie between the minima of the two components. In figure 12.95, the second derivative happens to be positive in this region, but if we move the two minima apart then this is no longer the case, as shown in figure 12.96. If, however, we

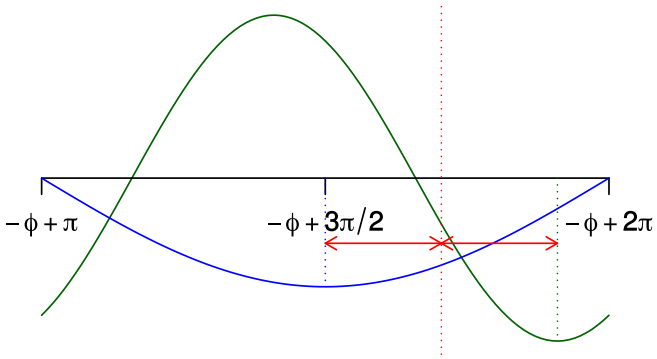


Figure 12.96: Moving the minima apart. Note that just to the right of the minima of the slow component, the total second derivative is negative.

consider the mid-point, then this must be closer to the minimum of the fast component than to the maximum[-2-]. Hence both components are negative and hence both contributions to the second derivative are positive. This is therefore a reasonable starting place for the application of Newton's method.

[-2-] The maximum must be farther than the minimum of the slow component, which is the same distance as the minimum of the fast component.

12.3 *When is gate parameter optimization applied?*

At present, gate parameter optimization may be applied whenever the genetic operator used to create the solution involves the insertion of a single parameterized gate. This includes the 'tweak

gate' operator. The probability that gate parameter optimization is applied is a parameter set by the user[-3-]. When gate parameter optimization is not applied, the gate parameter is mutated, i.e. it is set to a random value[-4-]

[-3-] Default value: 0.9

[-4-] Note that this is only really necessary after application of the 'tweak gate' operator. In other cases, the gate being inserted is already a random gate.

The 'tweak gate' operator therefore allows incremental optimization of a circuit's gate parameters, one gate at a time. It also provides the opportunity to give gate parameters a kick, to escape from local minima in the parameter space, without changing the structure of the circuit. As will be seen in the next section, it may also be necessary in order to escape other stationary points in the search space.

12.4 Issues

The ability to optimizing a single gate parameter is certainly useful and we might hope that repeated such optimizations, applied to different gates, would produce a solution close to that obtained via use of L-BFGS on all gate parameters simultaneously. While one would expect the total number of iterations required by such an approach, *if* numerical optimization is used, to exceed that for L-BFGS, the efficiency improvements attained by using information from the parent solution during each solution's evaluation would be expected to more than compensate. If algebraic optimization can be used, one might expect that repeatedly optimizing individual gate parameters would be the more effective approach.

However, there is a snag. While both approaches may be deceived by local minima, there is the possibility that iterated optimization of single gate parameters might also be deceived by other local stationary points. As a simple example, consider the function $f(a, b) = a^2 + b^2 + 6ab$ and suppose we start our search at $a = 0$, $b = 0$. Then minimization with respect to a is simply minimization of a^2 — a remains at zero. The same story is true for b. However, the origin here is not even a local minimum, but a saddle point. To see this, simply consider $a = -b$. Alternatively, we can define $x = a + b$, $y = a - b$ and we find that $f = 2x^2 - y^2$.

Of course, one may argue that it is unlikely that the search would start precisely at the saddle point, and that if both a and b are merely close to zero, then iterated optimization of single gate parameters will lead[-5-] away from the saddle point. This is true. However, we will see from the results in chapter 13, that the possibility of getting stuck at such saddle points has to be taken seriously.

[-5-] Albeit slowly.

13 Results

14 *Future direction*

Appendix A: C++ technicalities

While this document's primary purpose is to describe, in some detail, the algorithms developed for quantum circuit discovery, it also serves partly as documentation for the code base, covering details of the implementation of the algorithms. As such, some technicalities associated with the language used, C++, unavoidable intrude. To prevent lengthy diversions into such details in the main flow of the text, we have collected such technicalities into this appendix.

Containers

The C++ standard template library (STL) contains a selection of container classes, including vectors, deques (double ended queues), lists (double linked), maps, etc. This saves the typical developer from many of the headaches of memory management associated with the creation of such structures. A corollary is that manually performing such memory management, merely in order to obtain some functionality that is available in the STL container classes, is generally frowned up. In particular, everyday use of 'C-style' arrays is often considered to be 'evil', *especially* if the memory for the array must be manually allocated[-1-].

If the size of an array is known at compile time, the STL provides the array class, `std::array`. However, the usual replacement for a 'C-style' array (at least in my code), is the vector class `std::vector`. More sophisticated than a simple array, vectors handle their own memory management, can grow or shrink as necessary and allow for simple insertions and deletions from the middle of the vector.[-2-] So, when this document describes a "vector of pointers to Gates", feel free to replace 'vector' with 'array'.

[-1-] Of course, in this document I often casually use the words 'array' and even 'list' to refer to data that is actually implemented as a vector.

[-2-] Note, however, that the vector is still implemented, in the background, as a 'C-style' array, so insertions and deletions still incur the computational cost of moving elements up or down the array and, if necessary, the cost of reallocation of memory.

References and pointers

Just as modern C++ sees a reduction in the use of 'C-style' arrays, the use of naked pointers is supplanted (though to a lesser degree) by the use of 'smart' pointer classes. In particular, manual memory management of pointers is greatly reduced in modern C++ code. Alternatives to naked pointers used in our code include the following.

References: A pointer to an integer is clearly and explicitly different from the integer itself. So if `int x = 4`, we define a pointer to this integer via `int *y = &x` (pointer to integer, `y`, is assigned the address of `x`) and to get the value pointed to, we must explicitly dereference `y`, e.g. `*y = 5` This sets what `y` points to, i.e. `x`, equal to 5.

A *reference*, on the other hand, acts more like an alternative name for the same variable. We write `int &z = x`, making `z` a reference to `x`. Then the command `z = 6` sets both `z` and `x` to 6.[-3-] References are mainly used to more tidily implement pass by reference semantics in function calls. So one can define

```
void double(int& x)
{
  x *= 2;
}
```

and then simply call `double(y)` to change the contents of variable `y` to double its original value. More often, pass by reference is used not to allow the value of some external variable to be changed by a function, but to avoid the cost of copying some large object that would be incurred by pass by value. In this case, one passes by *const* reference, using a function declaration like `double totalProfit(const SalesData& salesData)`.

References are hardly new in C++ and most C++ code, including ours, use them extensively.

[-3-] Of course, in the background, this works via the use of pointers. However, references make the use of pointers implicit and simplify code by eliminating the need for dereferencing. There is also no need for any memory management — once a reference is declared to reference `x`, for example, there is no way to get it to reference something else.

Shared pointers: Data manipulated by C++ code falls into two categories: local variables, created on the stack and more persistent data, created on the heap. Any reasonable size project is likely to require the more persistent data and hence requires memory management. Thus C programs can involve a lot of memory allocation and deallocation, with the memory so allocated referenced by pointers. Managing the deallocation of memory, ensuring that it is freed precisely when it is no longer needed, can quickly become tricky. In C++, such manual memory allocation and deallocation should be avoided. Hence smart pointer classes have been developed that handle their own memory management automatically.

The shared pointer, `std::shared_ptr`, is the most flexible option. On allocating memory for an object, the pointer returned can be wrapped in a shared pointer, after which only the shared pointer is used. The shared pointer class keeps a count of the number of pointers pointing to an object. So when a shared pointer is copied, this count is incremented. Whenever a shared pointer goes out of scope, its destructor[-4-] is called with causes the count to be decremented. Once the counter reaches zero, the memory is automatically deallocated.

While one can create a pointer using `new` and then wrap it in a shared pointer manually, one usually uses the function `std::make_shared` instead.

[-4-] Classes typically define special functions called constructors and destructors. Here the destructor is called to destroy the *shared pointer*, not necessarily what the shared pointer points to.

Unique pointers: Shared pointers unavoidably incur some overhead[-5-], both in terms of speed and the size of the pointer object. For this reason, if the pointer isn't going to be shared, a `std::unique_ptr` is probably a better choice. A unique pointer is considered to be the *exclusive owner* of what it points to. As such, `unique_ptr`s cannot be copied; they can only be moved.[-6-]

C++ provides the function `std::make_unique` to tidily manage memory allocation. Memory deallocation happens automatically, of course.

Weak pointers: Completing the list of smart pointers, we have `std::weak_ptr`. These are not currently used in the code.

[-5-] I miss using `boost::intrusive_ptr`, which reduced this overhead, while still having all the functionality of shared pointers that I typically needed. However, I decided this project would be a good chance to explore modern C++, so used shared pointers and unique pointers instead.

[-6-] This has resulted in a few minor difficulties at times: it has been a learning experience!

Classes and objects

Basics

C++ is an object oriented language, in that it supports classes and objects. A class is simply a collection of variables and associated functions, called member functions[-7-], that manipulate these variables. So, for example, an Account [-8-] class might hold the account number, the name of the account owner and the amount of money currently available in the account, and have functions for setting up the account, depositing money, making withdrawals, producing a statement and so on. An instance of a class is called an object. Hence we may have a single Account class, but many Account objects for each of the bank's customers.

[-7-] Other languages refer to such functions as 'methods'.

[-8-] Class names begin with a capital letter in both this document and in the code.

Data members of an object are accessed using the dot notation familiar from `struct`s in C. For example, if `myAccount` is an object of class Account, then `myAccount.account_id` gives the account identifier for `myAccount`[-9-] Member functions are called using the same notation. Hence `myAccount.deposit(1000.00)` should add 1000 to the account balance. Defining a member function (outside of the class body) uses the notation `void Account::deposit(double amount)` — we will also use this notation to describe member functions later in this appendix.

[-9-] If `myAccount` is actually a *pointer* to an account, then we use `myAccount->account_id`.

In effect, the object `myAccount` can be thought of as a fourth parameter of the function `Account::deposit(...)`; the function has access to all the data stored within it. However, there are some important differences between calling the member function of an object[-10-] and passing the object in as a parameter, as we will see in section A.

[-10-] I will usually refer to this as calling the function `deposit` *on* the object `myAccount`

Data hiding

Classes are associated with the notion of *encapsulation*, which covers not only the bundling of data with the functions that manipulate the data, performed by classes, but also the hiding of the data from functions not in the class. Both the data and the member functions of a class may be labelled as being *public*, meaning that outside

functions can access the data directly, or *private*, meaning that outside functions can only manipulate the data through the public functions provided by the class. Public elements form the *interface* of the class, while private elements form part of the *implementation*.

For example, we may have a Duration class, used for storing (for example) marathon times, that holds the number of hours (integer), minutes (integer) and seconds (floating point). We define these as private members of the class and provide member functions for initializing a Duration, adding two Durations, writing a Duration to the screen, etc. Suppose that we output a Duration and it is written as 7 minutes, 60 seconds. Something is wrong. Perhaps the output routine is rounding 7:59.6 up to 7:60, instead of 8:00. Perhaps, instead, the seconds field really is set to 60. Where is the error? By limiting access to the data, we have narrowed our search for the error: one of the class's member functions must be setting the seconds field to 60. If it is the member function for adding two durations, then it is easy enough to supply a fix. If it is the initialization routine, then we can add code that performs sanity checks on the routine's parameters, ensuring that the input falls within the correct bounds. Compare this with the situation when the data is public. Then any part of the code might have set the seconds field to 60. The error could be anywhere.

Inheritance

Suppose that, in addition to basic bank accounts, we also need to model savings accounts. These might need to include the interest rate for the account and have associated functions for calculating and adding interest accrued. While we could create a new class from scratch, this will naturally involve a certain amount of code duplication.

A better approach is to use inheritance. A SavingsAccount class should have all the functionality of the basic Account class, with some additions. We give SavingsAccount the functionality and data of the Account class by (publicly) inheriting from the Account class, and then add the extra bits. In this case, the Account class is the *base* class, while the SavingsAccount class is the *child* class or *sub*-class.

We say that a SavingsAccount 'is an' Account[-11-]. As such, one can always convert a pointer to a SavingsAccount into a point to a basic Account[-12-]. Hence we can imagine a banking application to manipulate a vector of pointers to Accounts, but where each Account is actually a sub-type, i.e. is actually an instance of a sub-class of Account such as SavingsAccount.

[-11-] There are situations where this way of describing inheritance can lead one astray.

[-12-] Provided the Account happens to be a SavingsAccount, one can convert the other way too, but this is riskier — what happens if the account *isn't* a SavingsAccount.

Polymorphism and double dispatch

Since we are describing quantum circuit discovery code, we will now stop talking of Accounts and start talking of Gates and Cir-

cuits. In our code, we have a base Gate class, from which a variety of more specialized classes, e.g. XGate, Oracle, etc., inherit. The circuit is stored as a vector of pointers to Gates. How do we print out the circuit?

Clearly the Gate class requires an 'output' function. Then the Circuit output function could simply call the Gate output function for each Gate in turn. But the base Gate class doesn't contain the information that we would wish to output! This information is only available in the sub-classes. The answer is to use *virtual* functions and *polymorphism*.

A virtual function is a function whose implementation may be overridden by any child classes. In our case, we know that it should be possible to print out any Gate, so we provide the Gate class with a virtual function, `output`. We could give a default implementation of this function, for example, printing out nothing. A better approach, in our case, is to explicitly give no implementation in the Gate class. Such functions are called *pure* virtual functions and pure virtual function make a class an *abstract* base class[13]. Classes that have no pure virtual functions are referred to as *concrete* classes. We then supply each sub-class with their own implementation of `output` that *overrides* that of Gate.

The magic of virtual functions is called *polymorphism*. Given a pointer or a reference to a Gate, we may call Gate's output function. However, at run time, the program works out precisely what type of Gate the pointer actually points to and runs the appropriate sub-class's version of the function.

When it comes to circuit simplification routines, we need a function that can take a *pair* of pointers to Gates and perform the appropriate simplification, or determine whether the Gates can swap positions in the circuit. Just like the case above, the Gate class doesn't contain the information required to work out what simplification can be performed, or whether the Gates can be swapped — this information is only available in the concrete sub-classes. So can we use virtual functions here?

The problem here is that, for example, the `simplifies` function is only called *on* one of the gates, with a pointer or reference to the other gate passed in as an ordinary parameter. So we might write `firstGate.simplified(secondGate)`, i.e. we call the function `bool Gate::simplifies(Gate& next)` on `firstGate`. If `canSwap` is a virtual function, the code infers the type `firstGate`. So, if `firstGate` is actually a Hadamard, the function that is actually run is `bool Hadamard::simplifies(Gate& next)`. Notice that C++ does absolutely nothing to the parameter, `next`, which stubbornly remains a reference to base class Gate.

So how do we get the code to infer the type of the second gate? We use a technique known as *double dispatch*. The function `bool Hadamard::simplifies(Gate& next)` calls a second function, called something like isSimplifiedBy[14] where the roles of the two gates are reversed, using the line `return next.isSimplifiedBy(*this)`[15].

[13] Naturally, one cannot create objects of an abstract base class, such as Gate. One must create objects of the concrete sub-classes, such as XGate and Hadamard, that provide implementations for all of the member functions.

[14] In our actual code, the name of the second function is actually the same as that of the first, which is perfectly legal. However, when describing double dispatch, it is clearer are first if we take extra care to differentiate between the two functions.

[15] Member functions have access to a special pointer, `this`, which points to the object being operated on.

Here, the second function is `bool Gate::isSimplifiedBy(Hadamard& prev)`. Now if this function is also declared to be virtual, the code infers the type of the second gate and actually runs a function like `bool XGate::isSimplifiedBy(Hadamard& prev)`. So at the expense of an extra function call[-16-], the code can automatically determine the types of both gates, at run time, and run the desired function.

[-16-] And virtual table lookup

This technique is not a perfect solution to the problem, by any means. Whenever we add a new type of gate, we need to add an extra function, `void Gate::isSimplifiedBy(NewGate& prev)`, to the Gate class, breaking encapsulation. Controlling access to the functions is problematic, meaning that some functions in the base class that probably ought to be 'private' must be declared as 'public'[-17-]. Determining which function actually gets to run also becomes trickier when there are intermediate classes such as those present in our code.

[-17-] This is particularly worrying when the functions all share the same name. A more detailed description of my concerns is provided in a comment at the top of `gate.h`.

Appendix B: Glossary

We have attached very specific meanings to a number of, possibly vague-sounding, phrases. These phrases and their meanings are collected in this appendix, along with other technical terminology that might be unknown to the reader.

Circuit context: The gate types made available for a run of the optimization algorithm, with their costs, and the number of qubits. Also contains information about whether a qubits input value is always zero, always one or varies. Using this information, the `CircuitContext` class also provides access to functions that determine the number of possible gate options, provides a randomly generated gate (subject to availability), etc.

Circuit structure: The list of gate types, in sequence, and the qubits to which each gate is applied. The circuit structure *does not* include the values assigned to any gate's angle parameters.

Double dispatch: A technique in C++ extending polymorphism, where on calling a function involving, for example, *two* (pointers or references to) Gates, *both* concrete Gate classes are inferred and the correct function implementation selected for execution.

Gate option: A combination of gate type, target qubit and control qubits. Given two qubits and two types of gate, XGate and ZGate, with each gate permitted to have zero or one control qubits, then there are 7 available gate options.[-1-]

Phase type gate: A gate that leaves the $|0\rangle$ qubit alone, but multiplies $|1\rangle$ by a phase. Represented by a diagonal matrix with the top left element equal to 1. Given a controlled phase type gate, the target qubit can be swapped with any of the controls without changing the effect of the gate.

Polymorphism: Calling a (virtual) member function of a base class, with the result that a different function is executed depending on the concrete class of the object on which it is called. Hence the Gate class provides an `apply` function, but when called, the function that is executed is that defined in the XGate or Oracle or Hadamard class. The precise type of the Gate is inferred at run time.

[-1-] 4 XGate options, 2 with a control qubit and 2 without, and 3 ZGate options, 1 with a control qubit and 2 without. (Remember that we can switch the target qubit of a ZGate with any of the control qubits without changing the effect of the gate. Hence a ZGate on qubit zero with qubit one acting as a control is the same gate option as a ZGate on qubit one with qubit zero acting as a control.)

Reduction: The replacement of a parameterized gate with a cheaper set of gates that perform the same task, but only because the original gate's parameter is set to a special value. For example, an XRotation can be eliminated entirely if the angle parameter is zero. An ArbitraryPhase of angle $\pi/2$ reduces to a PhaseGate. Circuit reduction refers to the application of gate reduction across all the parameterized gates in the circuit. Reduction leads to an equivalent *circuit*, but an inequivalent circuit *structure*.

Simplification: Circuit simplification is an attempt to reduce circuit cost, without changing circuit behaviour[-2-]. A simplification is a small step in this process, perhaps replacing a pair of gates with some other sequence of gates, a single gate or just eliminating the gate pair. Usually a simplification will reduce circuit cost, though at some points we consider 'zero improvement' simplifications.[-3-]

Useful degrees of freedom: The total number of gate parameters for the circuit, ignoring any that are redundant. For example, given two consecutive, uncontrolled XRotations on a qubit, there is only one useful degree of freedom: only the total angle matters.

[-2-] If circuit cost accurately characterizes circuit complexity, then the result is a simpler circuit. In other cases, circuit 'simplification' may increase the gate count, producing a more complex (but cheaper) circuit. However, we continue to use the word.

[-3-] I plan to change this, perhaps by using the word 'adjustment' to describe minor changes to the circuit that may, or may not, result in cost reduction.

Appendix C: Bibliography

[1] Jorge Nocedal and Stephen J. Wright. *Numerical Optimization*. Springer, second edition, 2006.