Project in the course:
Many-particle physics and Quantum Computers
Electrical & Computer Engineering - 9th semester
Professor: Georgios Varelogiannis

Battery Revenue Optimization with QAOA approach

Circuit and Optimizations

Pagonis Alexandros
03116076

School of Electrical and Computer Engineering
National Technical University of Athens
10 - December - 2021

Battery Revenue Optimization with QAOA approach

Circuit and Optimizations

Abstract

In this project, the technique QAOA is used to solve the Optimization Problem: Battery Revenue

Initially, we define the Battery Revenue Optimization Problem and the algorithm QAOA. Then, we set the parameters β and γ used by the unitary operators of QAOA, we express the objective function to be maximized and we construct the quantum circuit that corresponds to the algorithm [1].

Also, we introduce a series of optimizations for the circuit, which reduce its time and cost, while simultaneously increase the algorithm's precision. At last, we simulate the circuit in IBM's simulator and measure the results.

Contents

1 Introduction

1.1 Battery Revenue Optimization Problem

The Battery Revenue Optimization problem is a constrained combinatorial problem and its definition (with an example scenario) is this: A company rents a battery for usage. Two markets M_1 and M_2 are interested in renting the battery for up to n days, but the company has only one battery and thus, must choose where to rent it every day. Market M_1 offers $\lambda_1^{(t)}$ euros to rent the battery at the t-th day and M_2 offers $\lambda_2^{(t)}$. The constraint is that with daily usage the battery is being harmed, and our goal is to use only one battery for the time window of these n days. Additional information is being given by the markets: M_1 will damage the battery by $c_1^{(t)}$ at the t-th day and M_2 by $c_2^{(t)}$. The battery has an endurance of C_{max}. Hence, in a more precise mathematical formulation the problem is expressed thus:

For a choice $z = z_1 z_2 \cdots z_n$, (where $z_t = 0$ means choosing M_1 at day-t and $z_t = 1$ means choosing M_2) we want to find:

$$\begin{aligned} &\underset{z_i \in \{0,1\}^n}{\operatorname{argmax}} \sum_{t=1}^{n} \left[(1 - z_i)\lambda_1^{(t)} + z_i \lambda_2^{(t)} \right] \\ &s.t. \sum_{t=1}^{n} \left[(1 - z_i)c_1^{(t)} + z_i c_2^{(t)} \right] \leq C_{max} \end{aligned} \tag{1}$$

For the t-th day we have these options:

- If $\lambda_1^{(t)} \geq \lambda_2^{(t)}$ and $c_1^{(t)} \leq c_2^{(t)}$, then we choose M_1
- If $\lambda_2^{(t)} \geq \lambda_1^{(t)}$ and $c_2^{(t)} \leq c_1^{(t)}$, then we choose M_2
- If $\lambda_2^{(t)} \geq \lambda_1^{(t)}$ and $c_2^{(t)} \geq c_1^{(t)}$, then our profit is at least $\lambda_1^{(t)}$ and the damage is at least $c_1^{(t)}$. Hence, choosing M_2 gives us the pure profit $\lambda_2^{(t)} - \lambda_1^{(t)}$ and damage $c_2^{(t)} - c_1^{(t)}$.
- If $\lambda_1^{(t)} \geq \lambda_2^{(t)}$ and $c_1^{(t)} \geq c_2^{(t)}$, then our profit is at least $\lambda_2^{(t)}$ and the damage is at least $c_2^{(t)}$. Hence, choosing M_1 gives us the pure profit $\lambda_1^{(t)} - \lambda_2^{(t)}$ and damage $c_1^{(t)} - c_2^{(t)}$.

Cases 1 and 2 are trivial, where 3 and 4 are symmetric. Thus, without loss of generality, we suppose we are always in case 3. This reduced problem is typically known as 0-1 KnapSack, which we will discuss further in the optimization section. What interests us now is the complexity of Battery Revenue Optimization Problem solved by a classical algorithm (which is the same as 0-1 KnapSack's complexity since they are equivalent).

0-1 KnapSack is an NP-Complete problem. We will relax the problem by accepting approximate solutions, meaning: solutions unacceptable (exceeding C_{max}) and solutions acceptable, ideally close to the max acquirable profit. Therefore, our reduction is to the Approximate 0-1 KnapSack, which is solvable by classical algorithms, with the best FPTAS having run-time [2]:

$$O\left(n\log\left(\frac{1}{\epsilon}\right)+\frac{\left(\frac{1}{\epsilon}\right)^{\frac{9}{4}}}{2^{\Omega\left(\sqrt{\log\left(\frac{1}{\epsilon}\right)}\right)}}\right) \tag{2}$$

This result is a bit discouraging, since we want the approximation $1+\varepsilon$ to be pretty small, giving a very big $\log(\frac{1}{\epsilon})$.

Applications of the Battery Revenue Problem (or its equivalent KnapSack) are many and important ones. For example:

- When choosing the most profitable investments under the constraint of a maximum risk.
- In telecommunications, when the sender chooses a compression or a segmentation of data, so that all of them can be included in the biggest packet.

For such an important, yet difficult problem, we need a faster approach, so we incline towards quantum computation.

1.2 Adiabatic Computing?

A first thought for solving the Battery Revenue Optimization Problem would be via Quantum Adiabatic Computing. In this method we construct the Hamiltonian:

$$\hat{H}(t)=(1-t)\,\hat{H}_i+t\cdot\hat{H}_f,\quad t\in[0,1] \tag{3}$$

where $\hat{H}_i$ is the initial state and $\hat{H}_f$ is the (wanted) final state of the system with its eigenstates being the problem's solutions. The idea is that, after some time, the system will have been moved into the state $\hat{H}_f$, waiting for us to fetch the result by simply measuring. However, this technique fails to solve the (equivalent) KnapSack problem, as it has been shown [3]. In addition, this Hamiltonian is very difficult to construct [4]. But, even though this technique is being rejected, its basic idea will help form an educated guess, when choosing the QAOA's parameters β,γ.

1.3 Quantum Approximate Optimization Algorithm

Quantum Approximate Optimization Algorithm is a Quantum Approximate Algorithm that solves a whole group of combinatorial problems. Specifically, it corresponds to a bipartite circuit in which we repeat its two components p-times in order to converge to the desirable optimum solution, the idea being:

- We express the objective function $f(z)$ to be maximized. That is, we seek $z=\underset{z_t\in\{0,1\}^n}{\operatorname{argmax}} f(z)$, where $z_t\in\{0,1\}$ are the n choices we have to make.

- We find an operator C, such that $C|z\rangle = f(z)|z\rangle$ and we construct its corresponding unitary operator $U(C,\gamma) = e^{-i\gamma C}$, with the angle γ being a multiplicative factor in $\gamma \in (0, 2\pi)$ adjusting its importance in the algorithm.
- If σ_t^x is the operator $\begin{pmatrix} 1 & 0 \\ 0 & 1 \end{pmatrix}$ applied to the t-th qubit, we define the operator B as ths sum of all σ^x: $B = \sum_{t=1}^{n} \sigma_t^x$ and its corresponding unitary operator $U(B,\beta) = e^{-i\beta B}$ με $\beta \in (0, \pi)$, similarly to operator C.

Explanation:

Operator C "computes" $f(z)$ when applied to $|z\rangle$, since, by definition, it has f as its eigenstate. This information is then stored in the qubit's phase, not disturbing the qubit's probabilities. That's why we actually use the corresponding unitary operator $U(C,\gamma) = e^{-i\gamma C}$. Now, operator B is used for judging our "solution" by using the information written in the qubit's phase. Of course, we must again, construct the corresponding unitary operator. With these two operators used for convergence, we only need an initial state of equal probabilities for every qubit, which is easily done with Hadamard gates. Operators C and B are being repeated p-times with different parameters (β, γ) and the produced state is:

$$|\beta,\gamma\rangle \equiv U(B,\beta_p)U(C,\gamma_p)\cdots U(B,\beta_1)U(C,\gamma_1)|+\rangle^{\otimes n} \tag{4}$$

C's expected value is $F_p(\beta,\gamma) \equiv \langle\beta,\gamma|C|\beta,\gamma\rangle$. From the Quantum Adiabatic Theorem we have:

$$\lim_{p\to\infty}\left[\max_{(\beta,\gamma)} F_p(\beta,\gamma)\right] = \max_{z\in\{0,1\}^n} C(z) \tag{5}$$

meaning: for infinitely many repetitions and different parameters (β,γ), our algorithm will converge to the optimum solution: $\max_{z\in\{0,1\}^n} C(z)$. Practically, the bigger the p, the better the convergence.

QAOA technique is extremely useful in combinatorial optimization problems, such as MAX-CUT, KnapSack, MAX3-SAT [5] (All of them having NP-Complete complexity when searching for the optimum solutions).

Thus, we choose QAOA to solve Battery Revenue Optimization Problem. Its main advantages compared to the classical algorithms is its **simplicity**, its small **time complexity** $O(pn\log_2(n))$ or even $O(p(\log_2 n)^3)$ with the help of $O(n\log_2(n))$ ancillary qubits [1] and its **versatility** (which we'll examine in the optimization section). Our job is to construct operators $U(C,\gamma)$, $U(B,\beta)$ in circuit.

2 Circuit

For a QAOA problem to be defined we only need to construct the circuits $U(C,\gamma)$ and $U(B,\beta)$. The best choice of β and γ (for faster convergence) is not known [1, 6]. We could search for pairs, but that is time-consuming and data-specific, so a fruitless, hence rejected attempt.

Here (as mentioned in the introduction) we take inspiration from Adiabatic Quantum Computing when choosing there parameters. Initially, our solution is very poor and the convergence must be fast, hence β should be big and γ very small. As repetitions go on, β becomes smaller slowing the convergence and γ becomes larger rewarding our proximity to the solution.

Thus, imitating an adiabatic behaviour we define (for the k-th repetition) [1, 4, 7]:

$$\begin{aligned} \beta_k &= 1 - \frac{k}{p}, \quad \beta_k \overset{k\to p}{\to} 0 \\ \gamma_k &= \frac{k}{p}, \quad \gamma_k \overset{k\to p}{\to} 1 \end{aligned} \tag{6}$$

2.1 Number of Qubits

Our n choices being $|z\rangle = |z_1 z_2 ... z_n\rangle$, we certainly need n qubits for measuring. Apart from them, we also need some qubits to compute the battery's damage. This value will be at most $\sum_t max(c_1^{(t)}, c_2^{(t)})$, so we need at most $d = \left\lceil \log_2 \left(\sum_t max(c_1^{(t)}, c_2^{(t)}) \right) \right\rceil$ ancillary qubits to store this cost. These qubits we will represent with an 1D matrix $A[i], 0 \leq i \leq d-1$. We also need a flag qubit F that checks our solution's viability:

$$F = \left[\left(cost(z) = \sum_{t=1}^{n} (1 - z_t) c_1^{(t)} + z_t c_2^{(t)} \right) \leq C_{max} \right]_{0|1}$$

2.2 Architecture and Hamiltonian

The objective function to be maximized is $f(z) = return(z) + penalty(z)$ (bigger penalty gives unwanted results [1], so we stick to a linear behaviour):

$$return(z = z_1 z_2 ... z_n) = \sum_{t=1}^{n} \left[(1 - z_t) \lambda_1^{(t)} + z_t \lambda_2^{(t)} \right]$$

$$penalty(z = z_1 z_2 ... z_n) = \left\{ \begin{array}{ll} 0 & \text{if } cost(z) \leq C_{max} \\ -a(cost(z) - C_{max}) & \text{if } cost(z) > C_{max} \end{array} \right\}$$

$$f(z) = return(z) + penalty(z)$$

At first, we prepare the state $|z\rangle = |+\rangle^{\otimes n}$, by applying a Hadamard gate to every qubit. By doing so, our initial probability distribution is the uniform 50-50 for the states $|0\rangle$ and $|1\rangle$ of every qubit. As a whole we have $|z\rangle \otimes |\mathbf{0}\rangle \otimes |0\rangle$.

After that, we make $|z\rangle \otimes |cost(z)\rangle \otimes |0\rangle$ and finally we update the Flag qubit $|z\rangle \otimes |cost(z)\rangle \otimes |cost(z) \geq C_{max}\rangle$

If $Flag = 1$, then we impose a penalty to our solution, by imposing a phase shift in the $|cost(z)\rangle$ qubits, making the state $|z\rangle \otimes |cost(z')\rangle \otimes |F\rangle$.

Later, we re-initialize the ancillary qubits, making the state $|z'\rangle \otimes |0\rangle \otimes |0\rangle$. With this reverse process the auxiliary qubits turn back to zero, while the penalty traces back to the choice $|z\rangle$. Idea being: if process A computes $|cost(z)\rangle$ from $|z\rangle$, then process A^{-1} computes $|z'\rangle$ from $|cost(z')\rangle$ (while in the same time, re-initializes the ancillary qubits).

We repeat the parts $U(C, \gamma_k)$ and $U(B, \beta_k)$ p-times and finally we measure the first n qubits.

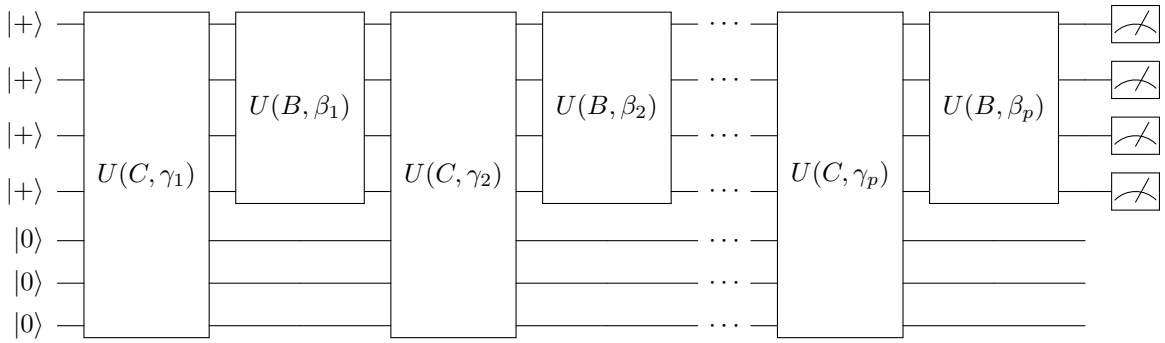

Figure 1: QAOA Circuit overview

The circuit grows linearly with respect to p and creates the state:

$$|\beta, \gamma\rangle \equiv U(B, \beta_p)U(C, \gamma_p) \cdots U(B, \beta_1)U(C, \gamma_1) |+\rangle^{\otimes n}$$

2.3 C Operator

We construct the C operator according to the objective function f:

$$\begin{aligned} U(C,\gamma) &= e^{-i\gamma f(z)} \left|z\right\rangle \\ &= e^{-i\gamma.penalty(z)} e^{-i\gamma.return(z)} \left|z\right\rangle \end{aligned} \tag{7}$$

2.3.1 Profit Calculation

Assuming being in case 3 (see introduction) our profits would be at least $\theta = \sum_{t=1}^{n} \lambda_1^{(t)}$. But, this value does not depend on our choice and, as implied, it's excluded. The pure profit we care about is the extra $(\lambda_2^{(t)} - \lambda_1^{(t)})$ for every t:

$$\begin{aligned} e^{-i\gamma.return(z)} \left|z\right\rangle &= \prod_{t=1}^{n} e^{-i\gamma.return_t(z)} \left|z\right\rangle \\ &= \cancel{e^{i\theta}} \bigotimes_{t=1}^{n} e^{-i\gamma z_t(\lambda_2^{(t)} - \lambda_1^{(t)})} \left|z_t\right\rangle \end{aligned} \tag{8}$$

Terms $e^{-i\gamma(\lambda_2^{(t)} - \lambda_1^{(t)})z_t} \left|z_t\right\rangle$ can be computed in parallel with phase gates $P(\varphi) = \begin{pmatrix} 1 & 0 \\ 0 & e^{i\varphi} \end{pmatrix}$:

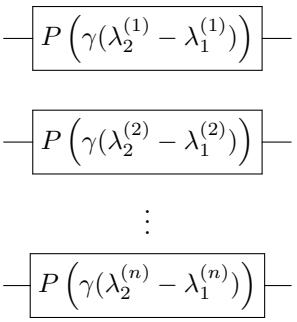

Figure 2: Profit Calculation

2.3.2 Cost Calculation

For every z_i corresponding to the choice $M_{\{1|2\}}$ for day i we have:

- $z_i = 0 \longrightarrow M_1$
- $z_i = 1 \longrightarrow M_2$

And we add $c_2^{(i)}$ or $c_1^{(i)}$ under the constraint of z_i into the A auxiliary array:

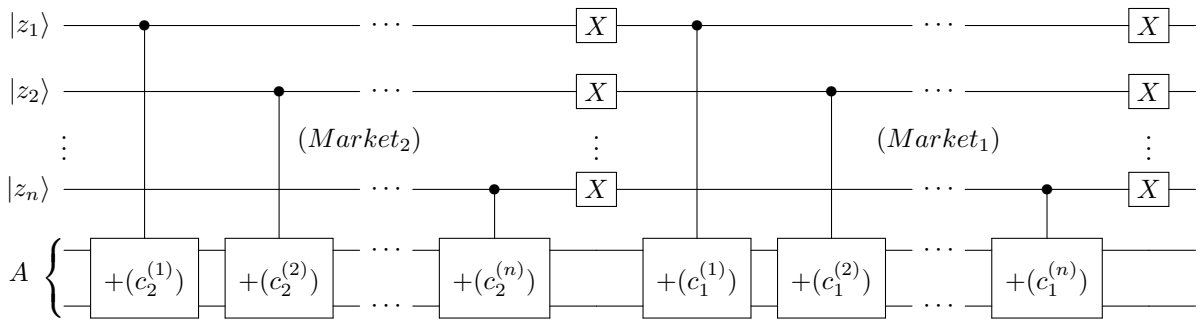

Figure 3: Cost Calculation

Another idea would be to add $c^{(i)}$ in parallel, instead of serially. With a "devide and conquer" technique: instead of n additions in series, we have $n/2$ and $n/2$ additions in parallel and then in $O(1)$ we mix the results. Recursively, $n/4$ then $n/8$ etc. This results in a $log_2(n)$-deep circuit. Ideally better, but must be turned down now, cause it demands too many qubits.

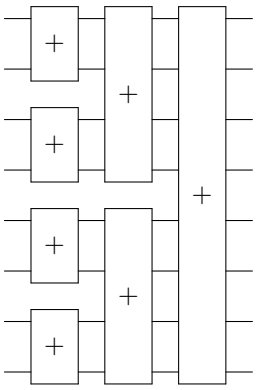

Figure 4: Cost Calculation in Parallel

The two approaches are shown in this table (we prefer less qubits than less time):

Method	Circuit Depth	Ancillary qubits
Serial Adders	$O(nlog_2 n)$	$O(n)$
Parallel Adders	$O((log_2 n)^3)$	$O(nlog_2(n))$

2.3.3 Constraint Checking

Having in binary representation the cost of choice z, we set Flag Qubit to 1, if it exceeds C_{max}. But, C_{max} is an arbitrary number, hence comparing with it is in general difficult. On the contrary, comparing with power of 2 is very easy. So, we add a constant value w on both sides of the inequality, making C_{max} reach its immediately larger power of 2:

$$\begin{aligned} cost(z) \leq C_{max} \quad &\overset{+w}{\Longleftrightarrow} \\ cost(z) + w \leq C_{max} + w = 2^c \end{aligned} \tag{9}$$

Now, we set Flag if all qubits above this power of 2 are zero. This can be done with a Multi-Controlled Not-Gate (which by the way, uses d-c-3 auxiliary qubits).

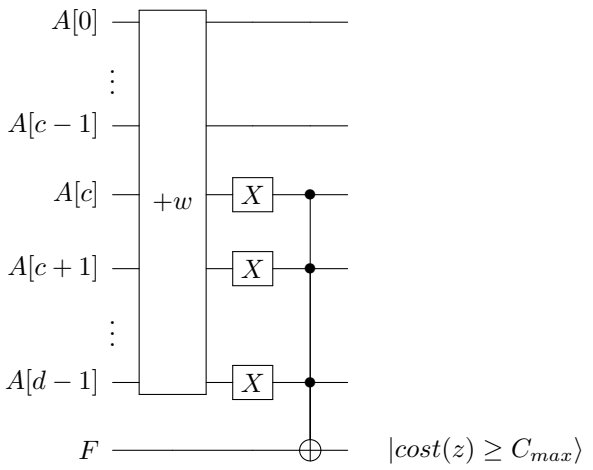

Figure 5: Constraint Checking

2.3.4 Penalty Dephasing

If $F = 1$, then penalty must be imposed:

$$a(cost(z) - C_{max}) = \sum_{j=0}^{d-1} 2^j aA[j] - 2^c a \tag{10}$$

This penalty is being put into the qubits' phase, according to the number's binary representation:

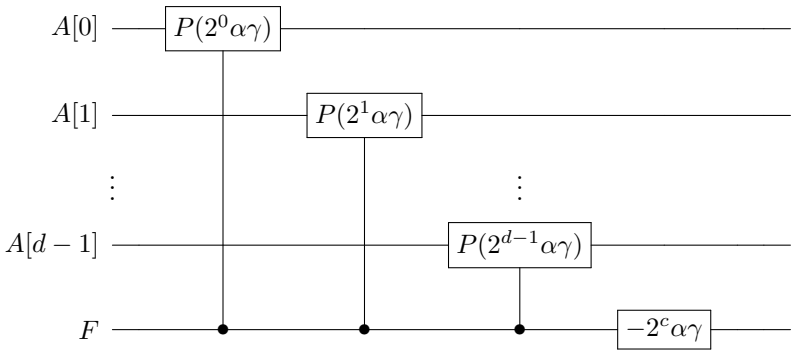

Figure 6: Penalty Dephasing

2.3.5 Re-initialization

To re-nullify the ancillary qubits, without destroying the state, we apply the "constraint checking" and "cost calculation" circuits in reverse. This way, the ancillary qubits go back to $|0\rangle$ while the penalty applied on the $|cost(z)\rangle$ traces back to $|z\rangle$.

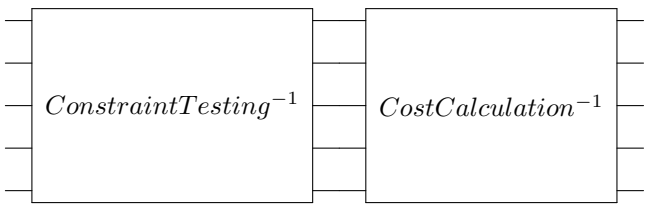

Figure 7: Re-initialization

2.4 B operator

The second part is simpler: For every qubit we must use its phase to "judge" our solution, via a multiplicative factor β. Thus, if σ_t^x is the operator $\begin{pmatrix} 1 & 0 \\ 0 & 1 \end{pmatrix}$ applied to the t-th qubit, we define operator B as the sum of all these σ^x:

$$B = \sum_{t=1}^{n} \sigma_t^x \tag{11}$$

and its corresponding unitary operator:

$$U(B, \beta) = e^{-i\beta B} \tag{12}$$

The unitary operator $U(B, \beta)$ can be implemented with $R_x(2\beta)$ gates upon every qubit.

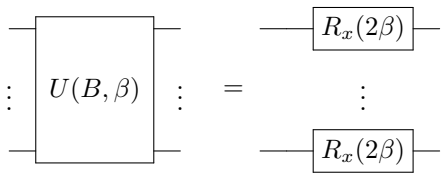

Figure 8: Mixing Operator Circuit

3 Optimizations

3.1 Reduction to Relaxed 0-1 Knapsack

The first optimization will be the reduction of Battery Revenue Optimization Problem to the Relaxed 0-1 KnapSack problem (as mentioned in the introduction).

0-1 Knapsack Problem: Given n items, each with weight w_i and value v_i, we must decide which ones we should put in a bag enduring a weight up to W_{max}, while maximizing the total value.

Typically, the problem has this mathematical formulation:

$$\begin{aligned} &\underset{z_t \in \{0,1\}^n}{\text{argmax}} \sum_{t=1}^{n} z_t v_t \\ &\text{s.t.:} \sum_{t=1}^{n} z_t w_t \leq W_{max} \end{aligned} \tag{13}$$

As explained above, without loss of generality, we choose case 3 of the Battery Revenue Optimization Problem. Easily this problem reduces to 0-1 KnapSack if we set:

$$\begin{aligned} v_t &= (\lambda_2^{(t)} - \lambda_1^{(t)}) \\ w_t &= (c_2^{(t)} - c_1^{(t)}) \\ W_{max} &= C_{max} - \sum_{t=1}^{n} c_1^{(t)} \end{aligned} \tag{14}$$

The basic idea of the reduction is this: We must choose one of the two markets every day, but being in case 3, we have a profit of at least $\lambda_1^{(t)}$ and a cost of at least $c_1^{(t)}$. So, ignoring these values which are implied, we care only for the excessive difference of the two. This way it's like choosing or not choosing Market M_2 every day, similarly to choosing or not choosing an item to put in our KnapSack bag.

By doing this, we reduce our additions to half since we need only add one value per day. Also we avoid using X-gates to toggle between 0-1 of z_i.

This alternation saves us n additions and $2n$ X-gates in each repetition!

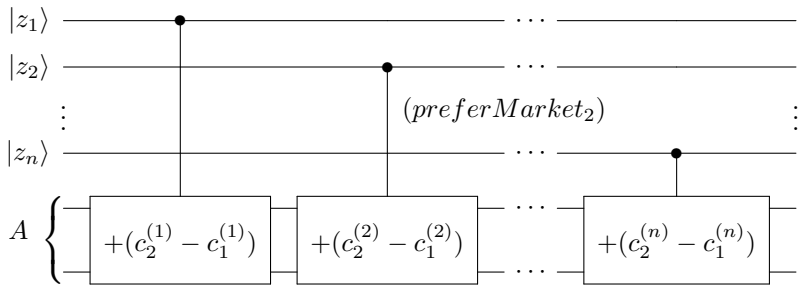

Figure 9: Optimized Cost Calculation

3.2 QFT Adder

There are many quantum circuit implementing a binary adder. Typical among them are:

- Plain adder network [8]
- Ripple carry adder [9]
- QFT adder [10, 11]

We will use the QFT Adder, because it has a very important advantage: we have many additions in series.

QFT Adder adds binary numbers represented by the states $|A\rangle$ and $|B\rangle$. The addition is easily done in **phase space**:

- It applies a **Quantum Fourier Transformation** in the first addend $|A\rangle \overset{QFT}{\longrightarrow} |\mathcal{F}(\mathcal{A})\rangle$
- It applies **Controlled phase gates** with B's qubits as control qubits and A's qubits as target qubits: $|\mathcal{F}(\mathcal{A})\rangle \overset{B_{phase}}{\longrightarrow} |\mathcal{F}(\mathcal{A}+\mathcal{B})\rangle$
- It applies a **Reverse Quantum Fourier Transformation** from the phase space fetching the result in the previously $|A\rangle$ state: $|\mathcal{F}(\mathcal{A}+\mathcal{B})\rangle \overset{IQFT}{\longrightarrow} |A+B\rangle$

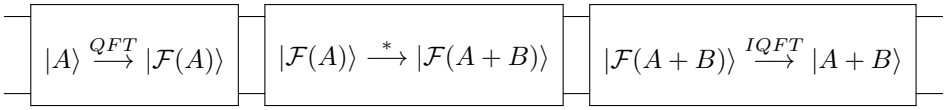

Figure 10: QFT Adder

That is, a specific act of phase gates into phase space corresponds to classical addition of two binary represented numbers. Counter-intuitive as it may seem (cause the price is the many QFTs) it's very fast, cause the QFTs and IQFTs between additions cancel each other out. With a closer look, it's clear that only one QFT and one IQFT are needed. We enter only once the phase space to do all additions and then we exit:

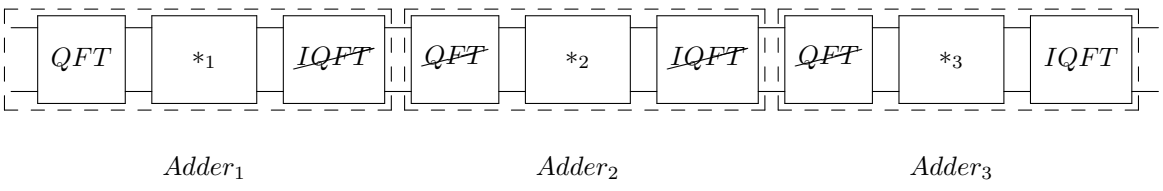

Figure 11: Serial Adder with QFTs

transforms into:

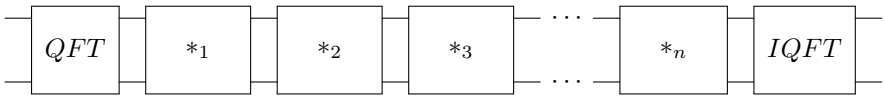

Figure 12: Reduced serial adder with QFTs

The two wanted circuits are constructed thus:

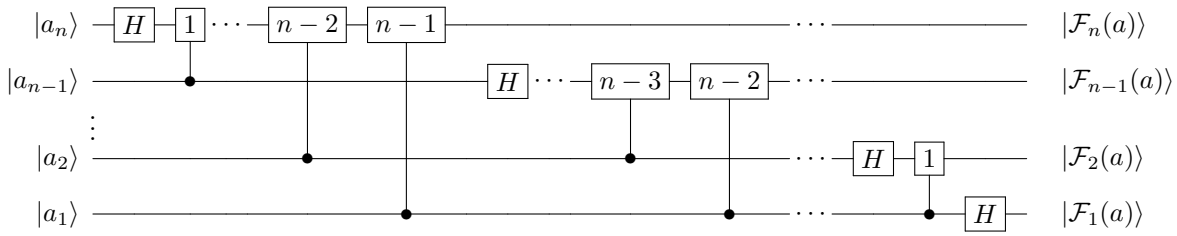

Figure 13: QFT Circuit

where $\boxed{k}$ is the gate corresponding to $P(\frac{\pi}{2^k}) = \begin{pmatrix} 1 & 0 \\ 0 & e^{\frac{i\pi}{2^k}} \end{pmatrix}$.

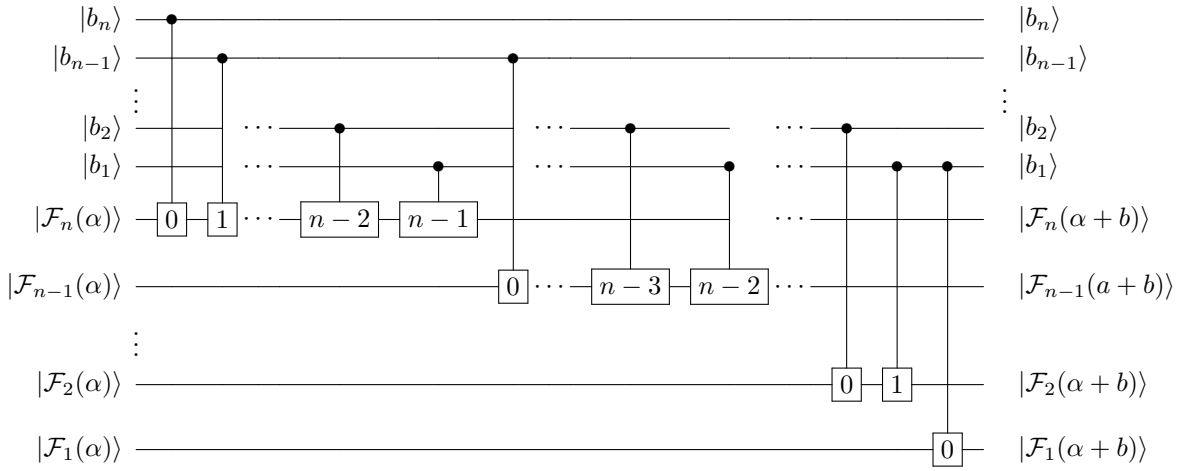

Figure 14: Phase Subroutine

Yet another advantage is that, in our specific case, number $|B\rangle$ is known, hence expressible with classical bits. So, the controlled phase gates can be substituted with simple phase gates (when $b_i = 0$) or no gates at all (when $b_i = 1$).

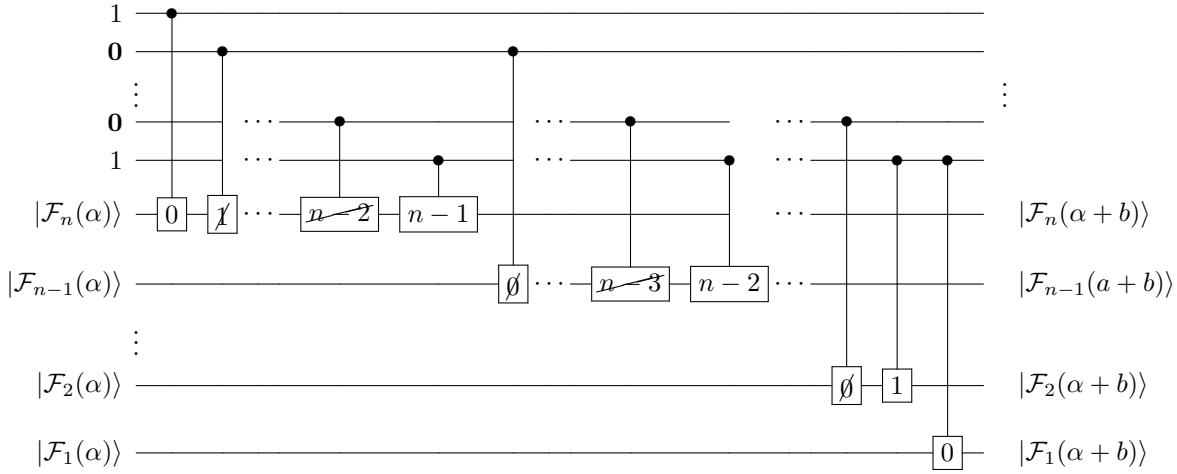

Figure 15: Phase Subroutine with no controlled gates

But for simple phase gates holds: $P(\varphi) * P(\psi) = P(\varphi + \psi)$, therefore all phase gates per qubit condense in one.

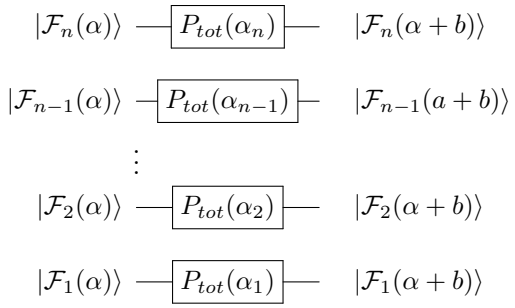

Figure 16: Improved Phase Subroutine

Also for our QFT we can ignore gates corresponding to small angles. In specific, we can add gates with k up to $k \approx log_2(n)$. Thus, our adder which needed $\frac{1}{2}n(n+1) = O(n^2)$ gates, now needs only $O(nlog_2 n)$ [10]. So we choose the Approximate QFT Adder [12].

By choosing the $_{appr}$QFT Adder and our optimizations, we have at last a grid of phase gates for the whole cost calculation (each layer controlled by some z_i, not shown here):

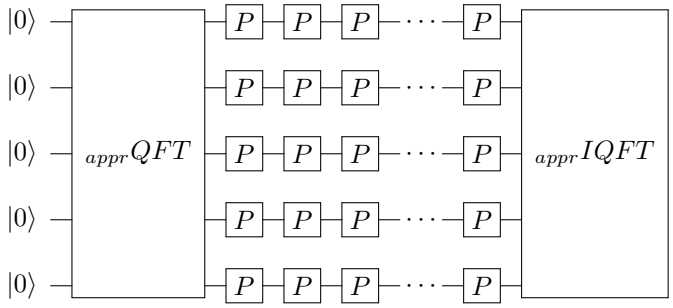

Figure 17: Optimized Cost Calculation Circuit

(Note: We have one more addition in constraint checking. We can postpone the IQFT until we have added w there).

Examining the adder more closely, we notice that the initial QFT is applied into qubits which are in the $|0\rangle$ state. Hence, the phase gates have no meaning, and the QFT degenerates into simple Hadamard gates (optimizing our circuit even further!). At last we have:

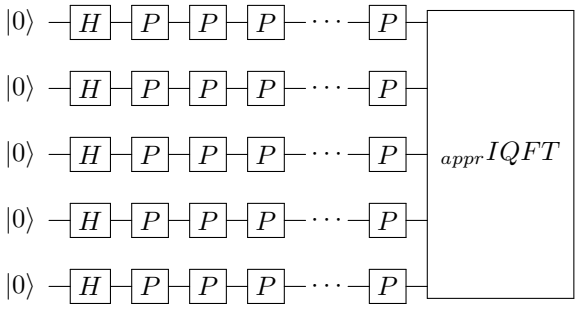

Figure 18: Final Cost Calculation Circuit

3.3 Avoiding Flag Qubit

While checking the constraint we used an extra qubit, namely: Flag Qubit. We added the value w missing from C_{max} to reach its immediately bigger power of 2 (2^c) and then we checked via a multi-controlled X-gate (which used another $d - c - 3$ ancillary qubits).

We propose this improvement: Instead of reaching the next power of 2 ($2^c \geq C_{max}$), we aim for the biggest power of 2, fitting inside the qubits of array A.

That is: $w' = 2^d - C_{max}$.

The idea remains the same: We add the new w' on both sides of the inequality: $cost(z) < C_{max} \Rightarrow cost(z) + w' < C_{max} + w' = 2^d$, so our last qubit acts as the new Flag Qubit and in the same time we discard the multi-CNOT.

Therefore we add w' on our data:

- If the last qubit d became 1, then $cost(z) > C_{max}$
- Otherwise, $cost(z) \leq C_{max}$

We avoid $d - c$ X-gates and $d - c - 3$ auxiliary qubits. Of course, we must modify the penalty dephasing circuit: Gates are controlled from the last qubit (there is no Flag qubit now).

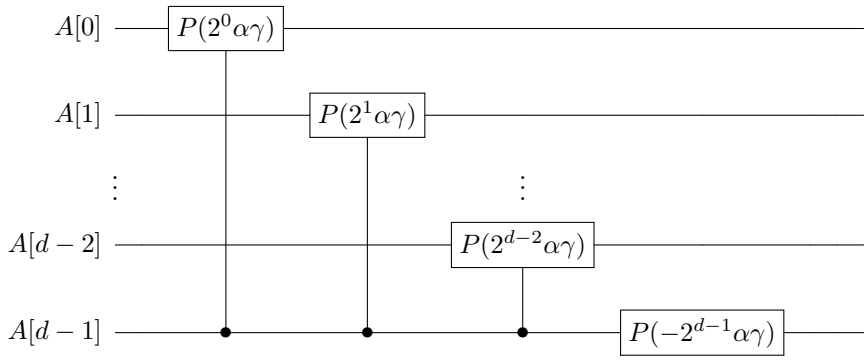

Figure 19: Improved Penalty Dephasing Circuit

3.4 Precision Increase

In our circuit, we initially apply a Hadamard gate in each qubit, meaning we force a uniform 50-50 probability distribution. This choice, though convenient, is completely arbitrary in respect to the data given, and as a result a better ansatz can and should be found.

Of course, choosing another distribution means constructing another mixer, one corresponding to it. For the initial distribution ($|+\rangle^{\otimes n}$) we used X-gates, because this gate has the eigenstates $(|0\rangle + |1\rangle)/\sqrt{2}$ and $(|0\rangle - |1\rangle)/\sqrt{2}$. In general, though, we need a mixer corresponding to an arbitrary probability distribution [6]:

$$
\begin{aligned}
|p_i\rangle &\coloneqq \sqrt{1-p_i}\,|0\rangle + \sqrt{p_i}\,|1\rangle \\
|p_i^{\perp}\rangle &\coloneqq -\sqrt{p_i}\,|0\rangle + \sqrt{1-p_i}\,|1\rangle \\
|p\rangle &= |p_1\rangle \otimes |p_2\rangle \otimes \cdots |p_n\rangle\,.
\end{aligned}
\tag{15}
$$

Therefore we need a mixer $\mathbf{X}$ with eigenstates: $\mathbf{X}_{p_i}|p_i\rangle = -|p_i\rangle$ and $\mathbf{X}_{p_i}|p_i^{\perp}\rangle = +|p_i\rangle$. We notice that Z-gate has the eigenstates $|0\rangle$ and $|1\rangle$, so $\mathbf{X}$ can be written as the linear combination of X and Z (that's why the symbol $\mathbf{X}$ is used, as their overlapping):

$$\begin{aligned}\mathbf{X}_{p_i} &= -(1-2p_i)Z - 2\sqrt{p_i(1-p_i)}X \\ &= -\begin{pmatrix} 1-2p_i & 2\sqrt{p_i(1-p_i)} \\ 2\sqrt{p_i(1-p_i)} & 1-2p_i \end{pmatrix}\end{aligned} \tag{16}$$

Of course, in the extreme cases: $\mathbf{X}_0 = -Z$, $\mathbf{X}_{1/2} = -X$. The only thing remaining is to construct this gate (which, naturally enough, will be made out of rotation gates). Indeed, defining the angles corresponding to the probabilities $\varphi_{p_i} = 2\sin^{-1}(\sqrt{p_i})$ we have the matrix:

$$\begin{aligned}\mathbf{X}_{p_i} &= -\begin{pmatrix} \cos(\varphi_{p_i}) & \sin(\varphi_{p_i}) \\ \sin(\varphi_{p_i}) & -\cos(\varphi_{p_i}) \end{pmatrix} \\ &= -\begin{pmatrix} 1-2\sin^2(\frac{\varphi_{p_i}}{2}) & 2\cos(\frac{\varphi_{p_i}}{2})\sin(\frac{\varphi_{p_i}}{2}) \\ 2\cos(\frac{\varphi_{p_i}}{2})\sin(\frac{\varphi_{p_i}}{2}) & -(1-2\sin^2(\frac{\varphi_{p_i}}{2})) \end{pmatrix}\end{aligned} \tag{17}$$

With further investigation, this matrix can be written as $\mathbf{X}_{p_i} = R_Y(\varphi_{p_i})ZR_Y(\varphi_{p_i})^{\dagger}$ and the corresponding unitary operator: $e^{-i\beta\mathbf{X}_{p_i}} = R_Y(\varphi_{p_i})e^{-i\beta Z}R_Y(\varphi_{p_i})^{\dagger}$. Also, $R_Y(\varphi_{p_i})^{\dagger} = R_Y(-\varphi_{p_i})$. Finally, this mixer can be implemented with this circuit:

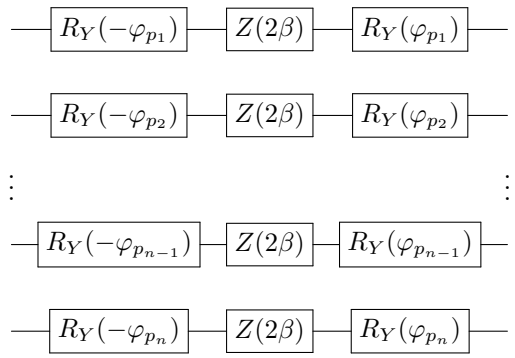

Figure 20: Mixer for the probability distribution p_i

As for the distribution: an initial idea would be to choose a completely constant distribution that has estimated cost C_{max}, bringing us to the edge of "acceptable" solutions from the beginning. To achieve this we set the same probability for all qubits (**Constant Biased State**):

$$\mathbf{Pr}([Q_i = 1]) = \frac{C_{max}}{\sum\limits_t c_i} \tag{18}$$

At the average case we exploit most of the cost C_{max}.

$$\mathbf{E}[cost(z)] = \sum_{t=1}^{n} c_i \cdot p_i = \frac{\sum\limits_{t=1}^{n} c_i \cdot C_{max}}{\sum\limits_{t=1}^{n} c_i} = C_{max} \tag{19}$$

This approach treats all the days as same, giving equal probability to every one of them. The complete opposite would be a totally biased distribution: including surely some choices and excluding others (**Lazy Greedy**).

Thinking this way, we put in descending order all the items with criterion the ratio $r_i = \frac{\lambda_i}{c_i}$ (their "normalized" value) and choosing the first m items until we exceed the limit C_{max}. Linear in time, though it rarely guesses the correct answer. In our terms, the first m items in this scale have a probability $p_i = 1$ and the next have $p_i = 0$, with r_{stop} being the ratio on the limit.

These two distributions are the extreme opposites (one constant and one completely biased) and both are totally greedy. Their behaviour is shown in the figures below.

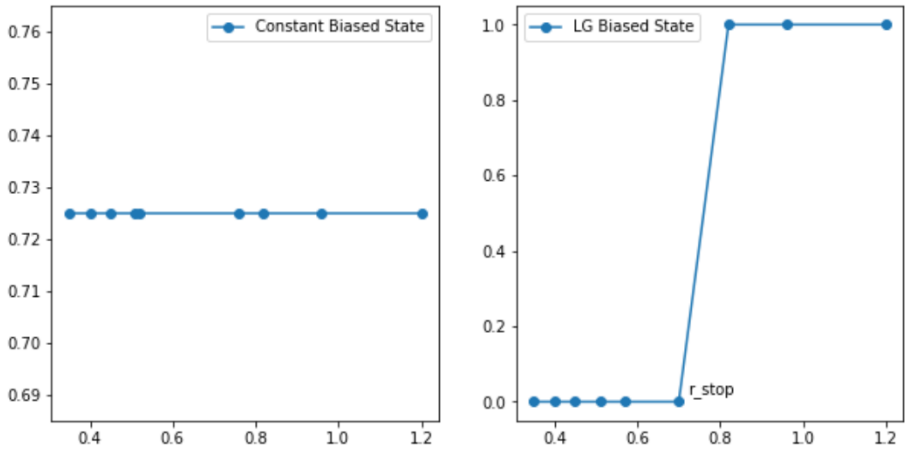

Figure 21: Two extreme probability distributions: **Constant** and **Lazy Greedy**

They will probably fail on our task. That's why we create a distribution in between them: The known **Logistic** [6]. A parameter k is our measure of how much the distribution inclines towards one of the two extremes:

$$p_i = \frac{1}{1 + Ce^{-k(r_i - r_{stop})}} \qquad (20)$$

$$\mu\varepsilon\ C = \frac{\sum_i c_i}{C_{max}} - 1$$

- With $\lim_{k \to 0} p_i$ = Constant Biased State
- With $\lim_{k \to \infty} p_i$ = Lazy Greedy State

A sample of this distribution is shown in this figure (with the limit r_{stop}):

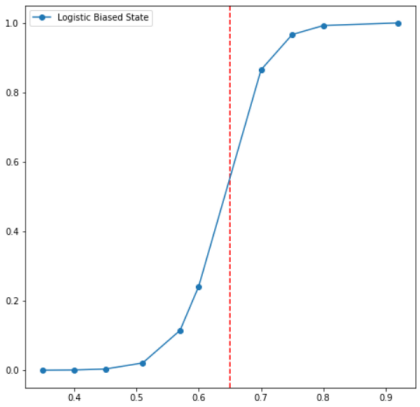

Figure 22: Logistic with a small parameter k

With a few tries we get a good value for k.

4 Results

4.1 Precision measurements

Since our algorithm is approximate, we must measure the precision of our results. The quantum computer gives a different answer every time, and so we must measure how many of them are acceptable ($cost(z) \leq C_{max}$) and how much are these close to the optimum profit λ_{opt}. Hence, as a metric we use the average profit compared to the optimum profit ($R(z) = return(z) =$ profit of choice z):

$$\frac{\sum_z \left[\left(R(z) - \sum_t \lambda_1^{(t)} \right) H(cost(z) \leq C_{max}) \right]}{N_{feasible} \cdot \left(\lambda_{opt} - \sum_t \lambda_1^{(t)} \right)} \tag{21}$$

The precision for the $|+\rangle^{\otimes n}$ and the Logistic (with k=5) distributions are shown below:

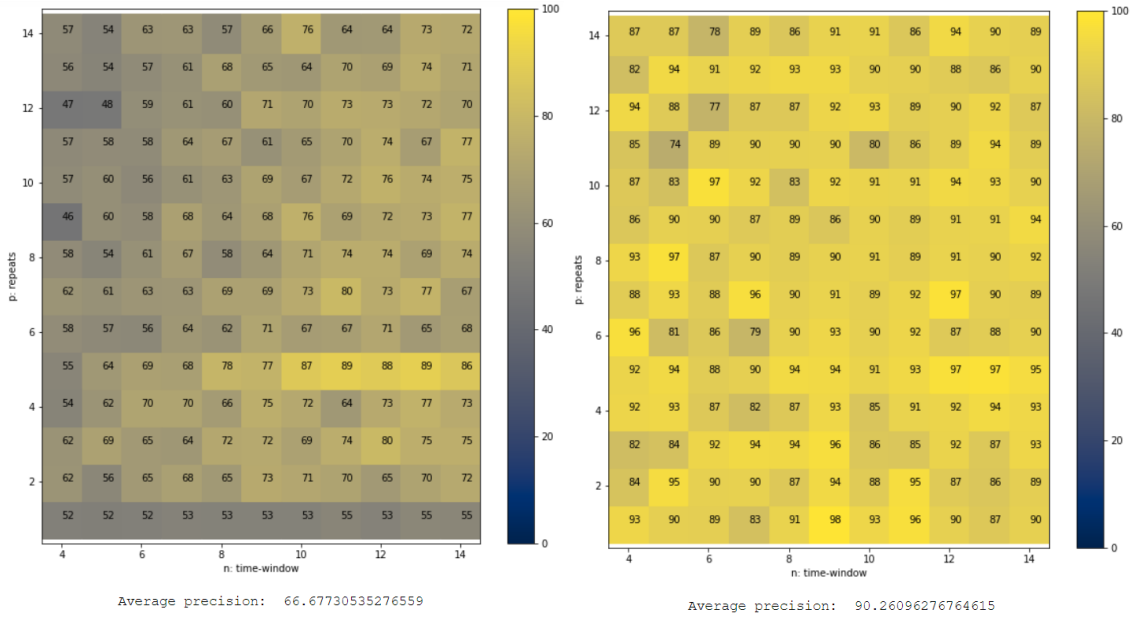

Figure 23: Algorithm precision for distributions $|+\rangle^{\otimes n}$ and **Logistic** ($k = 5$) [x100%]

4.2 Circuit Cost

For the circuit cost we are interested mainly on its depth (which will determine the run-time) and the number of used gates.

Since we use IBM's simulator we will analyse the circuit into the basic gates rz,sx,cx with the instruction transpile(QCircuit, basis_gates=['cx', 'rz','sx']).

The result of the simulations, for different values of n and p is shown in these figures:

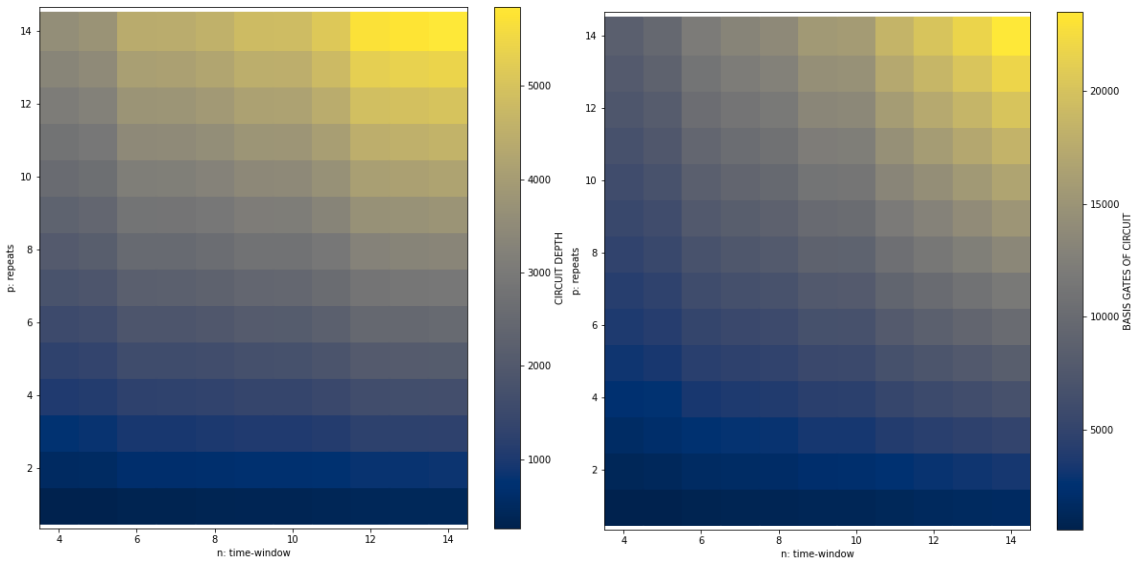

Figure 24: Circuit Depth and Number of Gates

As it seems from the pictures, the cost and run-time of the circuit grow linearly in respect to the length of the input and the algorithm's repetitions (as expected).

5 Epilogue

5.1 Conclusions

As shown in this project, problems equivalent to the Battery Revenue Optimization can be solved (in theory) by a quantum computer with a QAOA approach. With the Logistic distribution we achieved a sufficient precision of 0.9. Although, we notice that the precision does not improve for bigger values of p: this may be due to the parameters (β_k, γ_k), which do not contribute much when $k \to p$. However, for small values of p the precision is enough and the run-time is very low.

5.2 Thoughts for further improvement

To optimize the circuit even more we could try:

- Better **probability distributions**.
- Better **mixers**, preferably many-qubit mixers [6], taking into consideration the joint probability distribution function: when probabilities rise for one qubit, they must decrease for all others (since there is the limit of C_{max}). This approach though, needs to update every qubit for every other qubit, resulting in $O(n^2)$ gates and these updates as operators do not commute!
- Additions in **parallel** [1], which we avoided to use as few qubits as possible.
- Better **adders**.

6 Code

We import the necessary libraries:

```
from qiskit import QuantumCircuit, QuantumRegister
from typing import List, Union
import math
from qiskit import QuantumCircuit, QuantumRegister, ClassicalRegister, assemble
from qiskit.compiler import transpile
from qiskit.circuit import Gate
from qiskit.circuit.library.standard_gates import *
from qiskit.circuit.library import QFT
```

6.1 (Optimized) Circuit

```
# knapSack reduced
def solver_function(L: list, C: list, C_max: int, p: int) -> QuantumCircuit:

    # the number of qubits representing answers
    index_qubits = len(L)


    # data qubits and QFT approximation degree
    max_c = sum(C)
    data_qubits = math.ceil(math.log(max_c, 2)) + 1 if not max_c & (max_c - 1) == 0
        else math.ceil(math.log(max_c, 2)) + 2
    appr = math.floor(math.log(data_qubits, 2))


    # LOGISTIC BIASED STATE
    # sort indexes by ratio in ratios zipped list
    ratios = zip([l/c for (l,c) in zip(L,C)], range(n))
    ratios = sorted(ratios, key = lambda t: t[0], reverse=True)

    # find r_stop
    r_sum , r_stop = 0, 0
    for i in range(index_qubits):
        r_stop = ratios[i][0]
        if r_sum + C[i] > C_max:
            break
        else:
            r_sum += C[i]

    # find phi_i for every p_i
    phi = [0]*index_qubits
    for i in range(index_qubits):
        # parameters
        C_const = sum(C)/C_max - 1
        r_i = L[i] / C[i]
        k = 5

        # p_i and phi_i
        p_i = 1 / ( 1 + C_const * math.e**(-k*(r_i - r_stop)) )
        phi[i] = 2 * math.asin( math.sqrt(p_i) )


    def phase_return(index_qubits: int, gamma: float, L: list, to_gate=True)
        -> Union[Gate, QuantumCircuit]:

        qr_index = QuantumRegister(index_qubits, "index")
        qc = QuantumCircuit(qr_index)

        for i in range(index_qubits):
            qc.p( - gamma * L[i] / 2 , qr_index[i])

        return qc.to_gate(label=" phase return ") if to_gate else qc



    def subroutine_add_const(data_qubits: int, const: int, to_gate=True)
        -> Union[Gate, QuantumCircuit]:
```

```
    qc = QuantumCircuit(data_qubits)

    # accumulate phases per qubit
    phases = [0.0] * data_qubits
    for i in range(data_qubits):
        for j in range(i+1):
            if (const>>j)&1 == 1:
                phases[i] += math.pi / (1<<(i-j))

    for i in range(data_qubits):
        qc.p(phases[i], data_qubits-1-i)

    return qc.to_gate(label=" [+"+str(const)+"] ") if to_gate else qc



def cost_calculation(index_qubits: int, data_qubits: int, C: list, to_gate = True)
    -> Union[Gate, QuantumCircuit]:

    qr_index = QuantumRegister(index_qubits, "index")
    qr_data = QuantumRegister(data_qubits, "data")
    qc = QuantumCircuit(qr_index, qr_data)

    # QFT Once here (+ add + add w) then IQFT
    qc = qc.compose(QFT(data_qubits, appr), qr_data[:])

    for i in range(index_qubits):
        gate = subroutine_add_const(data_qubits, C[i]).control()
        qc.append(gate,  [qr_index[i]] + qr_data[:])

    return qc.to_gate(label=" Cost Calculation ") if to_gate else qc



def constraint_testing(data_qubits: int, C_max: int, to_gate = True)
    -> Union[Gate, QuantumCircuit]:

    qr_data = QuantumRegister(data_qubits, "data")
    qc = QuantumCircuit(qr_data)

    # add w'
    rest = 2**(data_qubits-1) - C_max
    qc.append(subroutine_add_const(data_qubits, rest), qr_data[:])
    # IQFT
    qc = qc.compose(QFT(data_qubits, approximation_degree=appr, inverse=True), qr_data[:])


    return qc.to_gate(label=" Constraint Testing ") if to_gate else qc



def penalty_dephasing(data_qubits: int, alpha: float, gamma: float, to_gate = True)
    -> Union[Gate, QuantumCircuit]:

    qr_data = QuantumRegister(data_qubits, "data")
    qc = QuantumCircuit(qr_data)

    # dephase qubits
    for i in range(data_qubits-1):
        qc.cp((2**i)*alpha*gamma, qr_data[data_qubits-1], qr_data[i])

    # dephase yourself
    phase = 2**(data_qubits-1)
    qc.p(phase*alpha*gamma, qr_data[data_qubits-1])

    return qc.to_gate(label=" Penalty Dephasing ") if to_gate else qc



def reinitialization(index_qubits: int, data_qubits: int, C: list, C_max: int, to_gate = True)
    -> Union[Gate, QuantumCircuit]:

    qr_index = QuantumRegister(index_qubits, "index")
    qr_data = QuantumRegister(data_qubits, "data")
```

```
        qc = QuantumCircuit(qr_index, qr_data)

        # clear ancillary qubits
        qc.append(constraint_testing(data_qubits, C_max).inverse(), qr_data[:])
        qc.append(cost_calculation(index_qubits, data_qubits, C).inverse(), qr_index[:] + qr_data[:])

        return qc.to_gate(label=" Reinitialization ") if to_gate else qc



    def mixing_operator(index_qubits: int, beta: float, to_gate = True)
        -> Union[Gate, QuantumCircuit]:

        qr_index = QuantumRegister(index_qubits, "index")
        qc = QuantumCircuit(qr_index)


        # Biased state mixer Ry(theta) exp(-ibetaZ) Ry(-theta)
        # Ry(w)^dagger = Ry(-w)
        for i in range(index_qubits):
            qc.ry(-phi[i], qr_index[i])
            qc.rz(2*beta, qr_index[i])
            qc.ry( phi[i], qr_index[i])



        return qc.to_gate(label=" Mixing Operator ") if to_gate else qc


        ##############################


    qr_index = QuantumRegister(index_qubits, "index")
    qr_data = QuantumRegister(data_qubits, "data")
    cr_index = ClassicalRegister(index_qubits, "c_index")
    qc = QuantumCircuit(qr_index, qr_data, cr_index)


    alpha = 1 # penalty parameter

    # Logistic initial state
    for i in range(index_qubits):
        qc.ry(phi[i], qr_index[i])



    for i in range(p):

        beta  = 1 - (i + 1) / p
        gamma = (i + 1) / p


        qc.append(phase_return(index_qubits, gamma, L), qr_index)
        qc.append(cost_calculation(index_qubits, data_qubits, C), qr_index[:] + qr_data[:])
        qc.append(constraint_testing(data_qubits, C_max), qr_data[:])
        qc.append(penalty_dephasing(data_qubits, alpha, gamma), qr_data[:])
        qc.append(reinitialization(index_qubits, data_qubits, C, C_max), qr_index[:] + qr_data[:])
        qc.append(mixing_operator(index_qubits, beta), qr_index)

    qc.measure(qr_index, cr_index[::-1])

    return qc
```

6.2 Classical algorithm using DP

To measure precision, we need a classical algorithm to solve 0-1 KnapSack:

```
def knapSack(W, wt, val, n):
    K = [[0 for x in range(W + 1)] for x in range(n + 1)]

    # Build table K[][] in bottom up manner
```

```
    for i in range(n + 1):
        for w in range(W + 1):
            if i == 0 or w == 0:
                K[i][w] = 0
            elif wt[i-1] <= w:
                K[i][w] = max(val[i-1] + K[i-1][w-wt[i-1]],  K[i-1][w])
            else:
                K[i][w] = K[i-1][w]

    return K[n][W]
```

6.3 Precision and Cost Measurements

```
from qiskit import *
import random
import matplotlib.pyplot as plt
import numpy as np


dataX, dataY = [], []
precisions, depths , gates = [], [], []

for n in range(4, 15):
    for p in range(1,15):

        test_cases = 5
        precision = 0
        qgates = 0
        qdepth = 0

        for t in range(test_cases):

            # Random input of length n
            C = [random.randint(1, 3) for i in range(n)]
            L = [random.randint(1, 4) for i in range(n)]
            C_max = random.randint(2, sum(C)-1)

            # solve classicaly with DP
            l_opt = knapSack(C_max, C, L, n)

            # solve in QC
            qc = solver_function(L, C, C_max, p)
            Ns = 1024//test_cases
            counts = execute(qc, Aer.get_backend('qasm_simulator'), shots=Ns).result().get_counts()
            data = sorted(counts.items(), key = lambda x:x[1], reverse=True)

            # evaluate cost or value of your choice
            def evaluate(L: List[int], a: str):
                s = 0
                for i in range(len(L)):
                    if a[i] == '1':
                        s += L[i]
                return s

            Nf , my_sum = 0, 0
            for i in data:
                cost = evaluate(C, i[0])
                val  = evaluate(L, i[0])
                shots = i[1]
                if cost <= C_max:
                    Nf += shots
                    my_sum += val * shots

            # transpile into basis gates
            qc = transpile(qc, basis_gates=['cx', 'rz','sx'])

            # take measurements
            qdepth += qc.depth()
            qgates += len(qc.get_instructions('cx')) + len(qc.get_instructions('rz')) + len(qc.get_instructions('sx'))
            precision += 100 * my_sum / (Nf * l_opt)

        # take avg of measurements
```

```
        dataX.append(n)
        dataY.append(p)

        precisions.append(precision/test_cases)
        gates.append(qgates/test_cases)
        depths.append(qdepth/test_cases)



norm = mpl.colors.Normalize(vmin=0, vmax=100)
plt.scatter(dataX, dataY, s=1600, c=precisions, cmap=plt.cm.get_cmap("cividis"), norm=norm, marker='s')
plt.xlabel("n: time-window")
plt.ylabel("p: repeats")
for i, z in enumerate(precisions):
    plt.annotate(str(int(z)), (dataX[i], dataY[i]))
plt.colorbar(mpl.cm.ScalarMappable(norm=norm, cmap=plt.cm.get_cmap("cividis")))
plt.show()
print("Average precision: ", sum(precisions) / len(precisions))



plt.scatter(dataX, dataY, s=1600, c=gates, cmap=plt.cm.get_cmap("cividis"), marker='s')
plt.xlabel("n: time-window")
plt.ylabel("p: repeats")
plt.colorbar()
plt.show()



norm = mpl.colors.Normalize(vmin=0, vmax=100)
plt.scatter(dataX, dataY, s=1600, c=depths, cmap=plt.cm.get_cmap("cividis"), norm=norm, marker='s')
plt.xlabel("n: time-window")
plt.ylabel("p: repeats")
plt.colorbar()
plt.show()
```

References

[1] Pierre Dupuy de la Grand'rive and Jean-Francois Hullo. "Knapsack problem variants of qaoa for battery revenue optimisation". In: *arXiv preprint arXiv:1908.02210* (2019).

[2] Ce Jin. "An improved FPTAS for 0-1 knapsack". In: *arXiv preprint arXiv:1904.09562* (2019).

[3] Lauren Pusey-Nazzaro et al. "Adiabatic quantum optimization fails to solve the knapsack problem". In: *arXiv preprint arXiv:2008.07456* (2020).

[4] Mark W Coffey. "Adiabatic quantum computing solution of the knapsack problem". In: *arXiv preprint arXiv:1701.05584* (2017).

[5] Edward Farhi, Jeffrey Goldstone, and Sam Gutmann. "A quantum approximate optimization algorithm". In: *arXiv preprint arXiv:1411.4028* (2014).

[6] Wim van Dam et al. "Quantum Optimization Heuristics with an Application to Knapsack Problems". In: *arXiv preprint arXiv:2108.08805* (2021).

[7] Stefan H Sack and Maksym Serbyn. "Quantum annealing initialization of the quantum approximate optimization algorithm". In: *arXiv preprint arXiv:2101.05742* (2021).

[8] Vlatko Vedral, Adriano Barenco, and Artur Ekert. "Quantum networks for elementary arithmetic operations". In: *Physical Review A* 54.1 (1996), p. 147.

[9] Steven A Cuccaro et al. "A new quantum ripple-carry addition circuit". In: *arXiv preprint quant-ph/0410184* (2004).

[10] Thomas G Draper. "Addition on a quantum computer". In: *arXiv preprint quant-ph/0008033* (2000).

[11] Lidia Ruiz-Perez and Juan Carlos Garcia-Escartin. "Quantum arithmetic with the quantum Fourier transform". In: *Quantum Information Processing* 16.6 (2017), p. 152.

[12] Adriano Barenco et al. "Approximate quantum Fourier transform and decoherence". In: *Physical Review A* 54.1 (1996), p. 139.