SHTEM 2023

Designing the Most Efficient Recombination Process by Classical and Quantum Algorithms

Aden Lee, Allan Jiang, Kim-Nga Shenoy, Vihaan Kodeboyina

Mentors: Junjie Luo, Kepler Boyce

Abstract

With the development of synthetic biology, to achieve highly specific and accurate control of living organisms, or to construct complex metabolic pathways, it is often desirable to create genetic circuits with multiple genetic elements. Traditional approaches involve docking these genetic elements on different chromosomes or integrating them at different loci far apart on the same chromosome and then recombining them. Because the traditional genetic approaches are constrained by the fundamental laws of genetics, the turnover time increases linearly with the number of genetic elements in the circuit. And the cost of maintaining all the genetic elements in the genetic circuit increases dramatically with the number of genetic elements.

Dr. Schnitzer Lab developed a recombination tool that can recombine two transgenes at the same docking site. This approach greatly accelerates the construction of intricate gene circuits and allows for the synthesis of biological strains with numerous genetic elements, leading to the efficient attainment of complex functionalities.

Based on the newest version of the Super Recombination system, SuRe 3.0, which uses 3 orthogonal adaptor pairs to sufficiently recombine any number of genes, we created a computational program that finds the quickest process to recombine multiple genetic elements. The turnover time for the recombination is proportional to the logarithm of the number of transgenes to be recombined. Our application initially assesses whether genes possess recombination capabilities. If recombination is possible, our application determines the shortest and quickest recombination tree by finding the shortest path. Our application allows researchers in the biology field to design the optimized recombination process with a computer automatically.

Introduction

The origins of synthetic biology can be traced back to the incorporation of rudimentary genetic circuits into viral genomes. Early experimentation focused on modifying viral genetic material to control viral replication, stability, and host specificity. This phase marked the first steps towards engineering biological entities for desired outcomes, setting the stage for more complex genetic circuit design (Meng & Ellis, 2020).

Biosynthesis started with gene circuits in virus genomes, where the first genetic pathways were incorporated into the mouse genome (Kodumal et al., 2004). The first experiments focused on the manipulation of bacterial genes in order to control viral morphology, complexity, and specific pathogen production. This phase served as the first step toward engineering organisms to achieve desired outcomes and it set the stage for the development of more robust genetic pathways.

As biology advanced, researchers turned their attention to prokaryotic organisms, and Escherichia coli (E. coli) served as a model system especially the simplicity and well-defined genetic makeup of this bacterium made it a suitable candidate for genetic circuit manipulation. The basic genetic mechanisms built in E. coli, such as toggle switches and oscillators, laid the foundation for more complex circuit design.

With the development of prokaryotic systems, the search for genetic circuits spanning the eukaryotic organism Saccharomyces cerevisiae, commonly known as yeast, emerged as a platform important for synthetic biology due to its ease of handling and similarity to higher eukaryotes (Xiong et al., 2008). Yeast can be used as the host organism to collect and test artificial embryos containing specific parts of an animal's genome. This process is part of artificial genomics, where scientists create standardized DNA sequences with complete chromosomes to study the function of genes, and can engineer new organisms. They often use yeast because it is a eukaryotic organism and shares certain cellular features with animals.

Synthetic biology and the increased capacity to recombine genes led to a need for more efficient approaches. Dr. Schnitzer Lab developed the Super recombination (SuRe) system. In the newest version of the SuRe system, Luo et al. proved that three orthogonal adaptor pairs can facilitate the recombination of any number of genes (Luo et al., 2022). The recombination process using SuRe 3.0 can be described by a recombination family tree (**Figures 1** and **2**). It has led to a need for a computer program that finds the shortest recombination family tree. Obtaining the shortest and most time-efficient recombination tree will speed up scientific advancements.

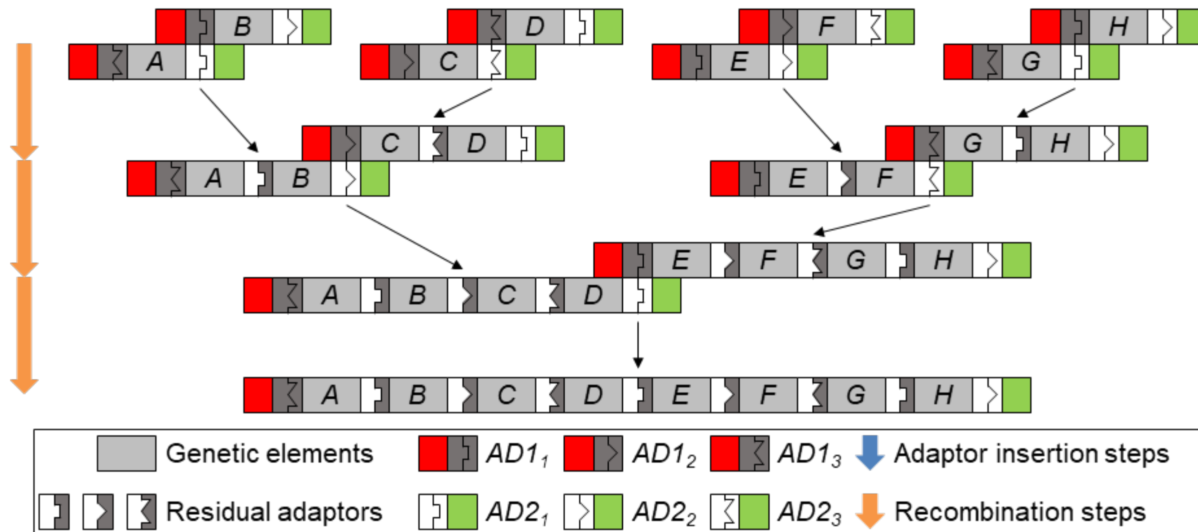

Figure 1. *The recombination of 8 genes by SuRe 3.0.*

Methods

Figure 2 shows the two steps to determine the algorithms for optimizing the recombination process. First, we check whether we have strain stocks. And that determines whether or not we use the quantum algorithm or the classical algorithms. Then we check whether we constrain the order of the genes, and that determines whether or not we are working on a gene que or a gene set. We found the classical algorithms with polynomial complexity for minimizing the height of the recombination family tree. But the classical algorithm for minimizing the number of strains has exponential complexity. So we tried using the quantum algorithm to make it more efficient.

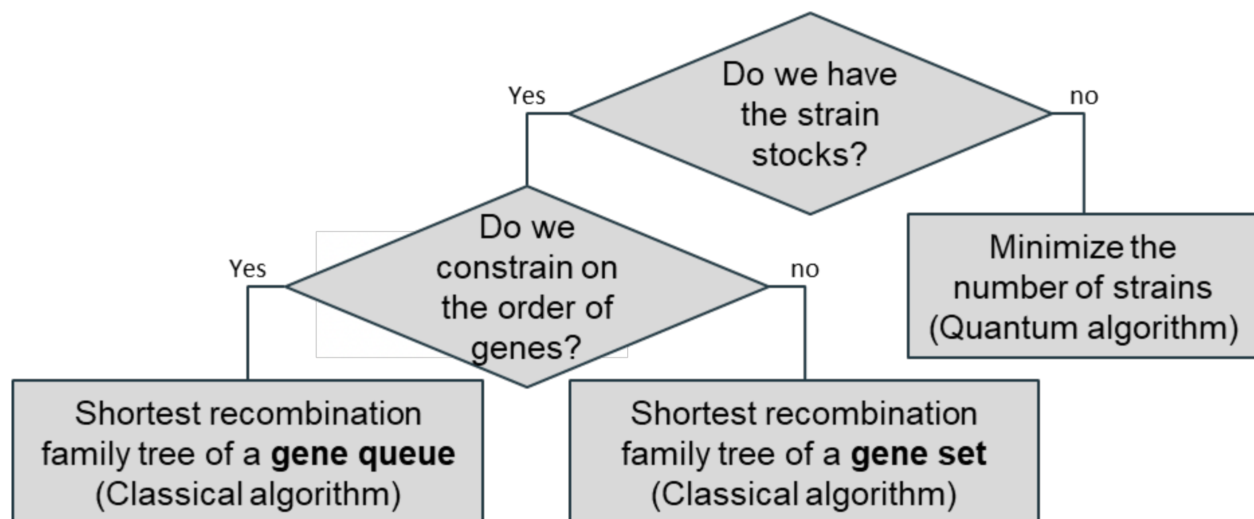

Figure 2. *The flowchart to select the algorithms*

Classical Algorithms:

Recombining genes is an expensive and laborious process. Developing an efficient algorithm for combining genes is highly valued among geneticists. By developing an efficient algorithm, we can construct more intricate gene circuits. The recombination family trees of **Figure 3A** and **Figure 3B** produce the same recombination results, but the tree in **Figure 3B** is shorter, which means this recombination process takes less time to create the product. The recombination product of the recombination family tree in **Figure 3C** contains the same genes as the product in **Figure 3B.** Because the order of genes in **Figure 3C** is not constrained, it is possible to find a recombination family tree shorter than the one in **Figure 3B**.

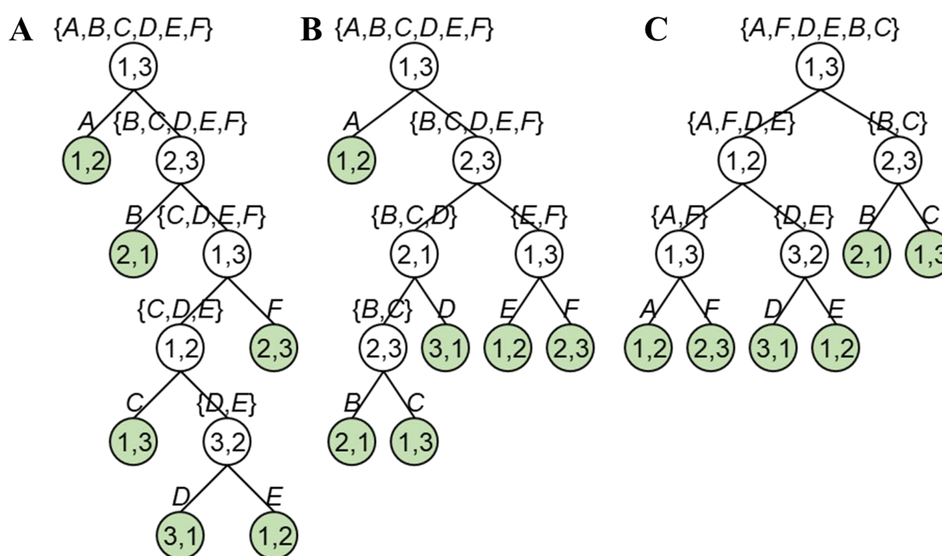

Figure 3. *Examples of recombination family trees*

Once the algorithm finishes reading the input, it validates whether or not the gene queue can be recombined by checking if the downstream adaptor of the original gene matches with the upstream adaptors of the next gene, neighboring adaptors are different, and there are three types of adaptors. If the input doesn't meet the required conditions, the program displays an error message letting the user know about the invalidity of their input. An example of such a message can be referred to in the Results section of this paper.

Once our input has been validated, the program utilizes dynamic programming to yield the shortest height and shortest recombination tree of the gene queue. The algorithm calculates the minimum height for increasingly larger trees and stores it in a NumPy array. It utilizes the stored information to combine sub-trees to create larger trees, avoiding the need for recomputation when calculating the height of trees. This allows for our algorithm to be more efficient and save

time. Finally, the program uses the result of the algorithm to draw the final tree and display the result to the user. An example of such results can be seen in the Results section. The time complexity of the algorithm for the shortest recombination tree given a gene queue was $O(n^3)$.

A gene queue is restrictive in the sense that it limits the number of combinations due to its inability to change its order. If the genes don't need to follow a particular order, however, we can define them as a gene set. The algorithm for a gene set goes through multiple combinations of different gene orders. Similar to the program for a gene queue, the program to find the shortest recombination tree of a gene set takes in input in the format of a csv file, uses a discriminant function to validate the input, and uses dynamic programming to find the shortest recombination family tree. The algorithm for a gene set starts by organizing a grid to keep track of the shortest height for each number of genes. Each cell in the grid stores the frequency of each set of upstream and downstream adaptors as a way of remembering the shortest recombination tree. **Figure 4** depicts an illustration of a grid.

.

	1	2	3	4	…
1	[1,0,0,0,0,0]	[]	[]	[]	…
2	[]	[0,0,0,1,0,1]	[]	[]	…
3	[]	[]	[1,1,0,0,0,1]	[2,1,1,0,0,0]	…
…	…	…	…	…	…

Figure 4. *Grid with columns being numbers of genes and rows being the minimum height of the tree*

The information stored in the grid is used to calculate the minimum height for the next column by combining sub-trees. Finally, the tree is drawn and displayed to the user, as shown in the Results section of the paper. The time complexity of the algorithm for the shortest recombination tree given a gene set was $O(n^8)$.

Quantum Algorithm:

If we do not have the strains to be recombined, we need to synthesize them. If some gene appears in the gene queue multiple times, we may find a recombination family tree that makes the repeated genes have the same adaptor pattern (**Figure 5**). Though both trees have an equal number of steps, the tree on the right is more efficient since the amount of unique gene adaptor

patterns on the right is less than that for the tree on the left. This minimizes the number of strains we need to prepare for the recombination.

In order to more efficiently design a solution to this optimization problem, we encoded the adapter pattern of each lower gene by 0 and 1 (**Figure 5**). As shown in the example, adaptor patterns (1,2), (2,3), and (3,1) were assigned 0, while adaptor patterns (2,1), (3,2), and (1,3) were assigned 1. Based on the rule that a gene's downstream adaptor should match the next gene's upstream adapter, we can compute the adaptor pattern of each gene from the bit string. If we need to recombine n genes, the number of possible bit strings is 2^n. If we use a classical algorithm to iterate all the bit strings, its complexity is exponential. Therefore, we tried to design a quantum algorithm to accelerate the solving of this problem.

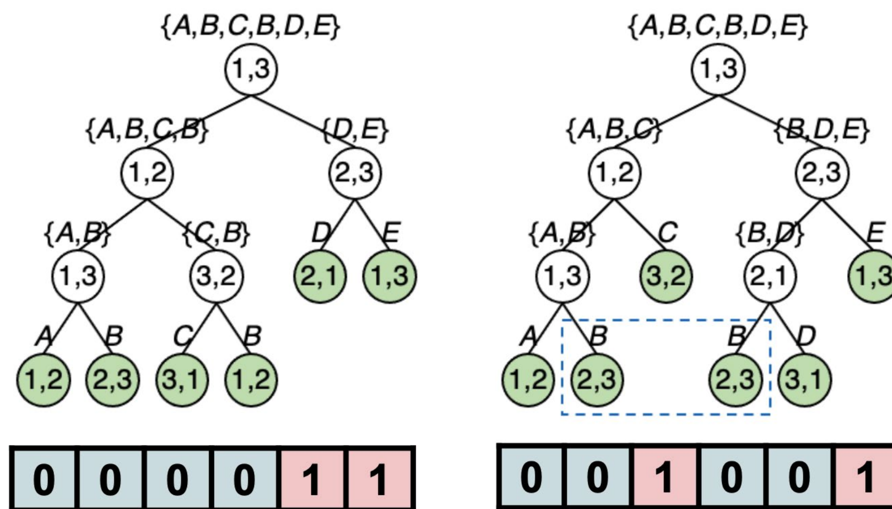

Figure 5*. An example in which the number of strains can be minimized.*

Quantum algorithms use qubits to store the bit strings, which is the quantum computation counterpart of the bit. Qubits are basic units of quantum information that can represent probabilities of 0 and 1. The quantum approximate optimization algorithm (QAOA) is a viable solution to this optimization issue. QAOA is a quantum algorithm that produces results for combinatorial-optimization problems (Zhou et al., 2020). We designed quantum circuits to encode the number of unique gene adaptor patterns and used QAOA to minimize it, in turn improving recombination efficiency. We used the IBM Quantum Platform to simulate these quantum circuits and run these quantum circuits on quantum computers.

Results

Classical Algorithms:

We created a working GUI that used Python backend code combined with Python's GUI package tkinter (**Figure 6**). The GUI is able to take the input files of a user's strain stock file and gene list file and outputs the shortest recombination tree and the number of steps it will take. The GUI gives the user a visual reference of the tree, giving them a clear set of steps to recombine the genes. There are different outputs that can happen after the user inputs their gene list and presses the "Gene Queue" or "Gene Set" button. The most sought-after output is the one that simply outputs the correct recombination tree and the number of steps. This means the data was correct, and everything worked accordingly. However, if there is something incorrect with the data inputted, the GUI will output an error code. The error code identifies the problem for the user, helping them fix any issues.

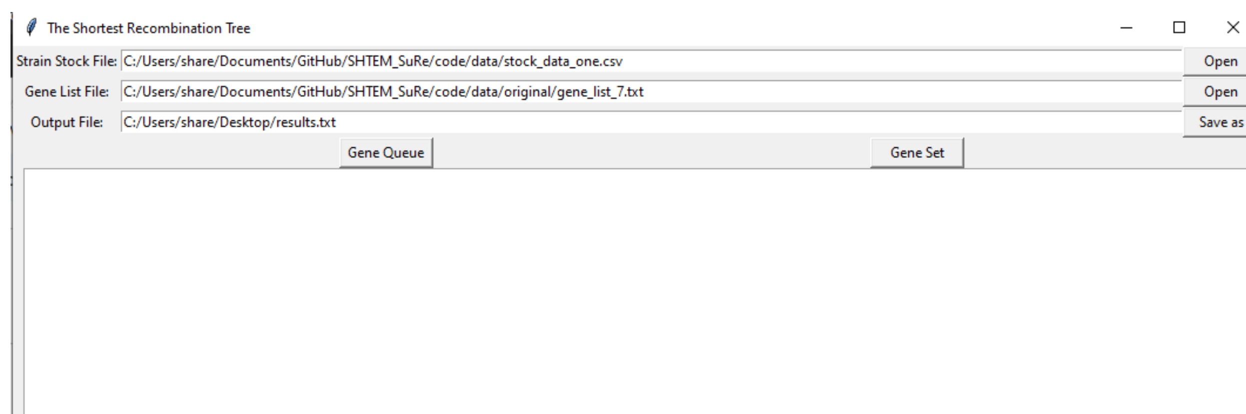

Figure 6. *GUI of our program using tkinter and Python*

When using our algorithm for a gene queue, we start by inputting the file of strain stock and the file of gene list (**Figure 6**). The strain stock has the genes and which adaptor pattern it corresponds to (**Figure 7A**). The gene list can be interpreted as the letters corresponding to each gene (**Figure 7B**).

A

1	Gene name	(1,2)	(2,3)	(3,1)	(2,1)	(3,2)	(1,3)
2	A	0	1	1	0	1	0
3	B	0	1	1	0	0	0
4	C	1	1	0	1	0	0
5	D	1	1	0	0	0	0
6	E	0	0	1	1	1	1
7	F	1	0	1	1	0	0
8	G	0	1	0	0	1	1
9	H	1	1	0	1	1	0
10	I	1	0	1	0	1	0

B

Y
U
F
G
O
H
D
B
I
L
P
N
S
T

Figure 7. *Examples of the input files. A) The csv file of the stock stain list. B) The txt file of the gene list.*

The GUI has two output options: gene queue and gene set (**Figures 8** and **9**). The only distinct difference between these two options is that the gene queue has a constraint on the order for the output, while the gene set doesn't. For example, looking at the images below, one can see that with the same inputs, two outputs can arise. With a closer look, anyone can realize that the order of the output for the gene queue goes from letter A-F in alphabetical order, while in the gene set output, it's not in the set A-F order. A reader and user will only have to understand that the only difference between a gene queue and a gene set is the constraint on the order of genes.

```
The gene queue can be recombined in 4 steps:
```

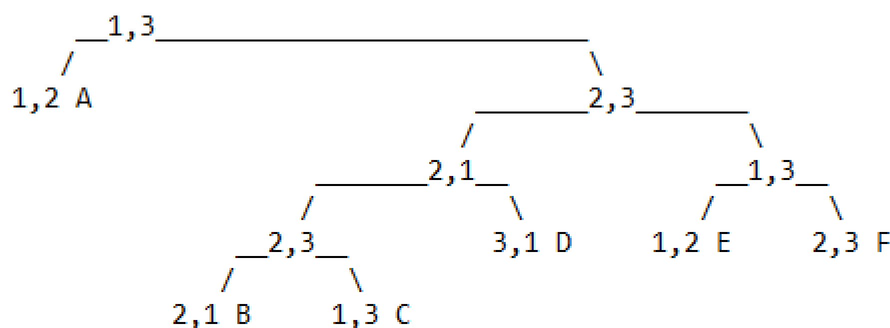

Figure 8*. The example output of a gene queue.*

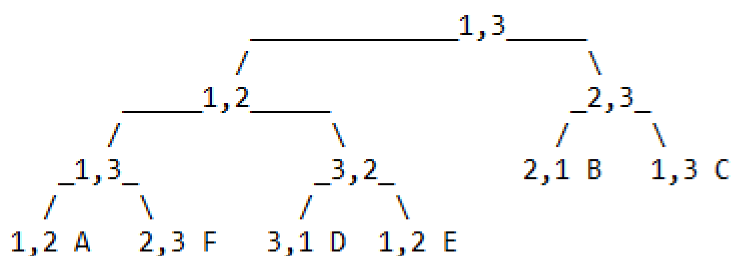

Figure 9*. The example output of a gene set.*

There are three error conditions of a gene queue and one error condition of a gene set: 1). Some genes in the list cannot be found in the stock. This means that a gene that was supposed to be in the strain stock file is missing, and the recombination cannot happen. The error code identifies what genes need to be created by the user and then put into the strain stock. 2). The upstream adaptor of a gene ahead in the queue doesn't match the downstream adaptor of the gene before it. This means that recombination is impossible unless the adaptors match. The error message identifies which genes have mismatching adaptors, helping the user quickly fix the problem of recombination. 3). The upstream adaptor of the first gene matches the downstream adaptor of the last gene. This will be impossible for recombination trees because the structure will form a circle, meaning that users won't be able to recombine towards specific genes they are looking for.

There are two gene set errors that can happen: 1). some genes in the list cannot be found in the stock. This error has the same meaning as the first type of error of gene queues. 2). If the gene set cannot be recombined, the GUI will show the numbers of genes with adaptor patterns (1,2), (2,3), (3,1), (2,1), (3,2), or (1,3). The users can figure out for the adapter pattern how many genes should be added or removed from the gene set to make the set can be recombined.

Quantum Algorithm:

We designed two quantum circuits to minimize the number of unique adaptor patterns using QAOA (**Figure 10**). Each quantum circuit consists of several qubits, essentially the quantum counterpart to the bit, and several gates, which allow the qubits to interact with each other and perform operations on them. The inputs of these quantum circuits are bit strings from the genes first B, C, and second B. Qubits 0, 1, and 2 are assigned to these three genes, which are either 0 or 1. Because the genes upstream of the first B and the genes downstream of the second B are

unique, their bit strings do not influence the number of strains. To save qubits, we do not input them into the quantum circuit.

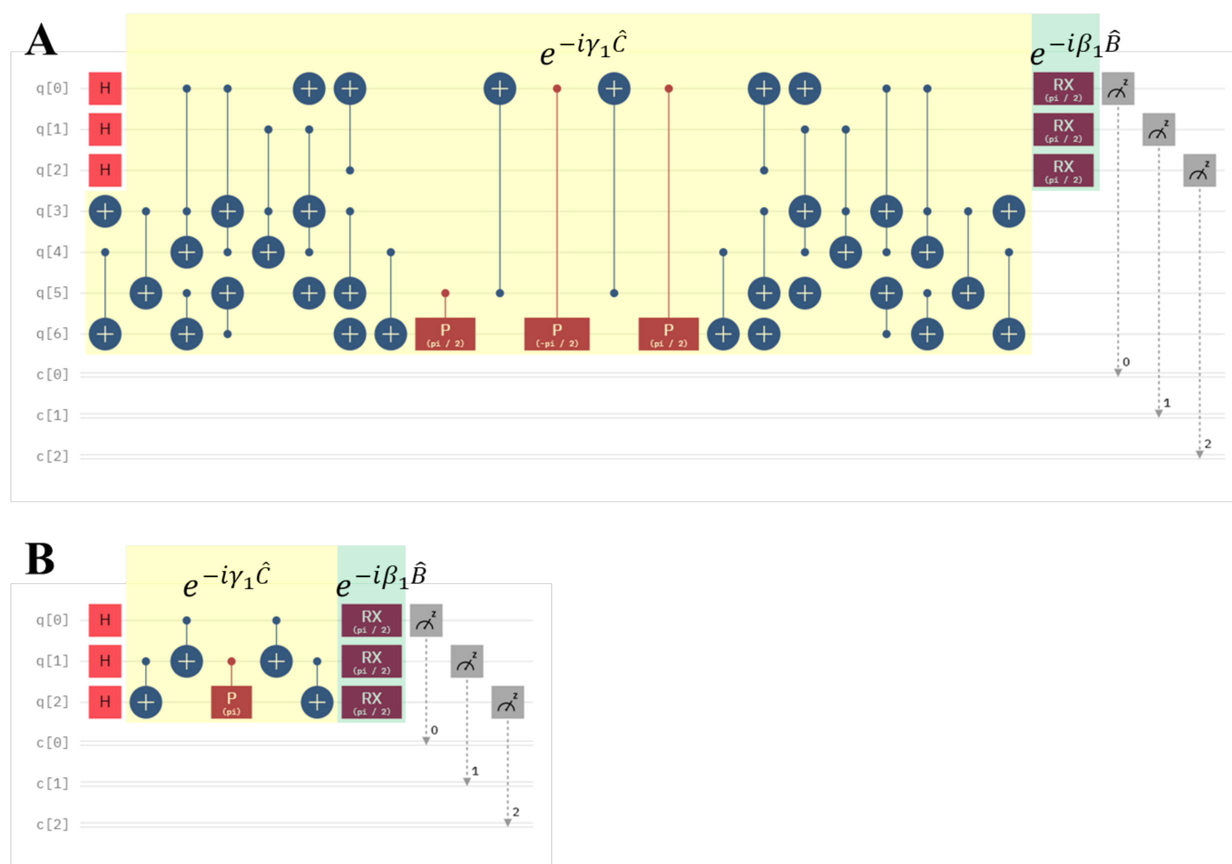

Figure 10. *The QAOA circuit to minimize the number of strains. A). The QAOA quantum circuit automatically generated by a computer program. B). The manually simplified QAOA quantum circuit.*

The four graphs below are the results of these two circuits (**Figure 11**). Results from the automatically generated circuit are shown under *Circuit **1** Results*, and results from the manually simplified circuit are shown under *Circuit **2** Results*. The graphs listed under *Simulation* are results from quantum computer simulators, and the graphs listed under *Quantum Computer* are results from real quantum computers.

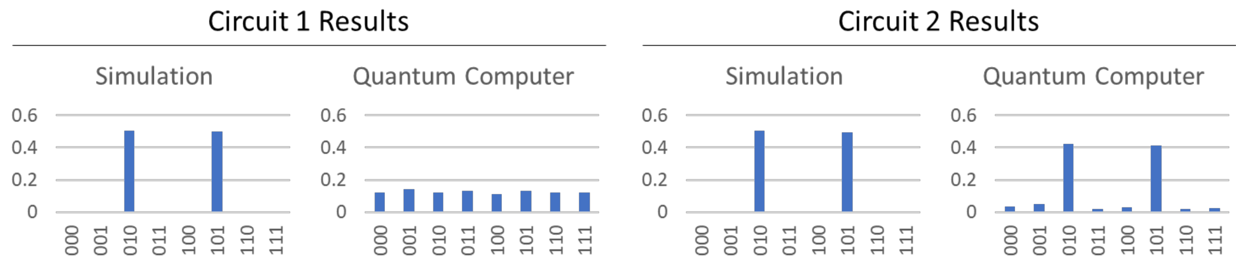

***Figure 11**. Results from Circuits 1 and 2.*

Preferable results are 0.5 probability for 010 and 0.5 probability for 101, shown by the simulation results. However, the real quantum computer contains noise, results from both circuits are not perfect.

The actual results from the first quantum circuit are inadequate and inferior to the results from the second circuit. This is due to a large amount of noise produced by the size and complexity of Circuit 1. The more gates and operations involved in a circuit, the higher the chance of noise and errors for the results produced by the circuit. In fact, results from the real quantum computer for Circuit 1 are almost random. Circuit 2, meanwhile, has been manually simplified and contains fewer gates than Circuit 1. This results in more simple and more efficient computation, leading to ~85% chance to output the correct solutions.

Conclusion

The main goal of our UI has been to create a recombination tree program that can reach scientists all around the world. The GUI is now in a state of being usable by people around the world. However there are still some updates that we are looking forward to implementing. Some key examples of the updates we are planning is window resizing. Our UI currently is unable to resize to different windows, so smaller devices like tablets or mobile phones might be unable to easily use the UI. Another addition we are thinking of implementing is some sort of progress bar or percentage to tell the user how much time they might have left. These fixes and many more in the future could make the UI more compatible for a number of users and help in achieving our goal of helping scientists around the world.

The quantum algorithm we designed shows the potential that it is able to minimize the number of strains. With the development of quantum computers, we expect quantum computers will have more qubits and lower noise. That will enable us to optimize a large-scale gene recombination problem with the quantum algorithm and demonstrate quantum primacy.

References

Kodumal, S. J., Patel, K. G., Reid, R., Menzella, H. G., Welch, M., & Santi, D. V. (2004). Total synthesis of long DNA sequences: synthesis of a contiguous 32-kb polyketide synthase gene cluster. *Proceedings of the National Academy of Sciences of the United States of America*, *101*(44), 15573–15578.

Luo, J., Schnitzer, M. J., & Huang, C. (2022). Genetic tools for recombining transgenes at the same locus (WIPO Patent No. 2022082225:A2). In *World Patent* (No. 2022082225:A2). https://patentimages.storage.googleapis.com/31/41/f8/8ee25bf241ff92/WO2022082225A2.pdf

Meng, F., & Ellis, T. (2020). The second decade of synthetic biology: 2010–2020. *Nature Communications*, *11*(1), 1–4.

Xiong, A.-S., Peng, R.-H., Zhuang, J., Gao, F., Li, Y., Cheng, Z.-M., & Yao, Q.-H. (2008). Chemical gene synthesis: strategies, softwares, error corrections, and applications. *FEMS Microbiology Reviews*, *32*(3), 522–540.

Zhou, L., Wang, S.-T., Choi, S., Pichler, H., & Lukin, M. D. (2020). Quantum Approximate Optimization Algorithm: Performance, Mechanism, and Implementation on Near-Term Devices. *Physical Review X*, *10*(2), 021067.