COMPSCI 687 Final Project: RL-based Scheduler for Shuttling Quantum Dot Arrays

Anthony Micciche and Henry Lin

December 8th 2023

1 Introduction

There has been recent promising work in the field of spin qubits in silicon for the purposes of efficiency fabricating large-scale high-fidelity quantum computers. Specifically of interest, is the recently demonstrated ability to physically shuttle qubits in a linear array of quantum dots. [Zwe+23] Building upon this notion of qubit shuttling, recent work has conceived of two of these qubit arrays being used as a quantum computer [MK23]. The constraint being that once of these arrays would be fixed with the other being allowed to shift left and right to enable different connectivity between the qubits. As this is a reinforcement learning assignment, the physics of this is unimportant to this report, so let us start the process of abstracting this to a simple reinforcement learning problem, starting with Figure 1.

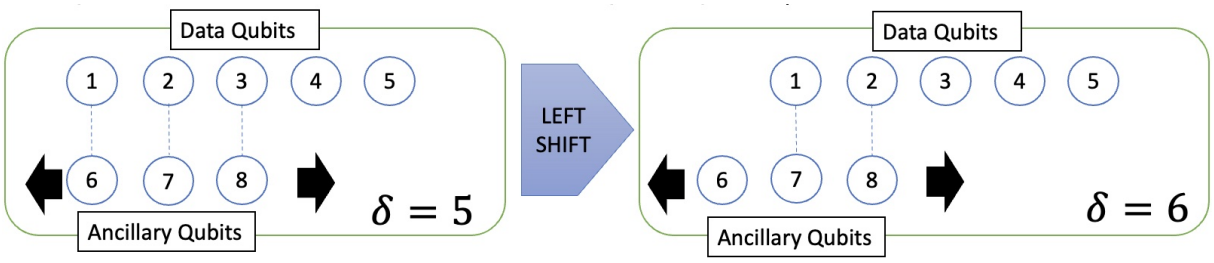

Figure 1: On the left is a possible configuration of this quantum computer, which we can call $\delta = 5$. This represents the offset between the indices of the two rows of qubits (data and ancillary qubits in the case of quantum error correction). This $\delta = 5$ states that connectivity is only permitted between the pairs (1,6), (2,7), and (3,8). If we shift the bottom row to the left, we would then have the right image which has $\delta = 6$ and connectivity between (1,7) and (2,8).

Each time a row of qubits is shifted, an error is introduced into the system simply due to the inescapable fragile nature of quantum systems. Furthermore, being able to execute quantum circuits in fewer shifts thereby increases parallelism within the quantum computer, leading to shorter computation times. In the realm of quantum computation, these shorter computation times are of upmost importance as quantum states exponentially decay into meaninglessness over time. In other words, reducing the number of shifts to execute a quantum circuit (effectively a quantum "program") will reduce errors in the quantum computer in at least these two ways.

Quantum Circuits

To state the problem more concretely, given a quantum circuit, we want to find an equivalent quantum circuit that when mapped to a device like the one in Figure 1, will be executable in a minimal number of

shifts of the architecture. For those unfamiliar with what a quantum circuit it is, Figure 2 will contain the absolute bare minimum description necessary for understanding the problem.

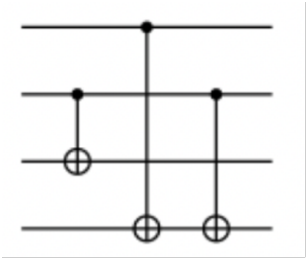

Figure 2: This is a useless quantum circuit, but will illustrate enough to understand our RL problem. The four horizontal lines represent four qubits, and time flows from left to right. The things connecting these lines all represent CNOT gates. Where the open circle represent the target, and the dot represents the control. We can view this circuit as saying: Apply a CNOT from qubit 2 to qubit 3, then a CNOT from 1 to 4, then a CNOT from 2 to 4. There are far more two qubit gates than just CNOT, but for brevity, let's just consider CNOT for this report

Effectively the gates in our circuit will tell us which qubits we want to line up by shifting the architecture, and one way of reducing the number of shifts is to think of clever relabelings of the qubits in the circuit. For this project, we will only consider relabeling the the ancillary qubits (bottom row of the architecture) in some quantum error correcting code's syndrome measuring circuits. These circuits have many nice properties which allow for us to freely permute gates, as well as other things, but we will not be discussing them here. Refer to Figure 3 for an example of how Figure 2 can be transformed to run in fewer shifts.

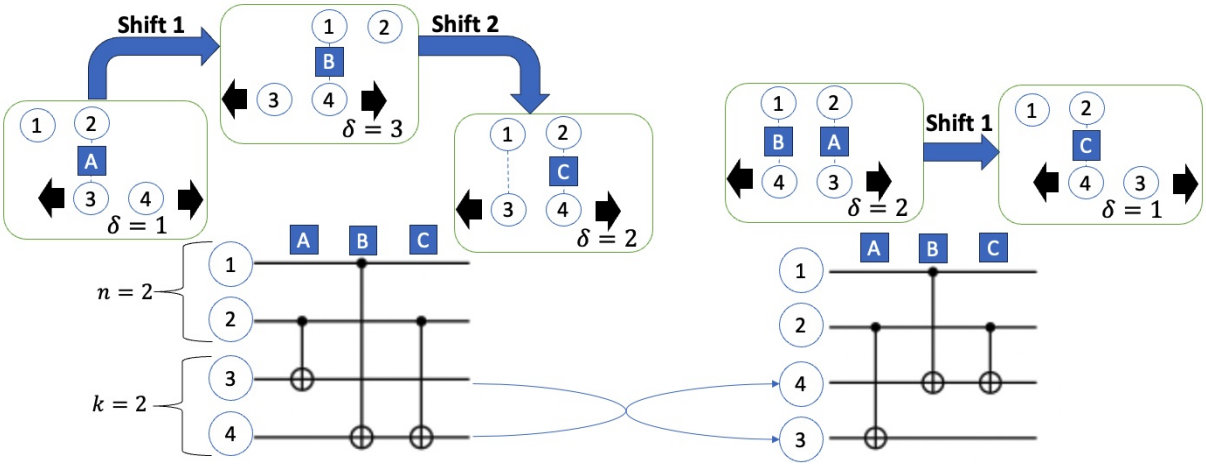

Figure 3: On the bottom-left, we have the original circuit, where we have labeled its three CNOT gates as A,B,C and split its four qubits into two groups, which we can interpret as data and ancillary qubits. Recall that we are only going to think about how to permute the ancillary (bottom) group of qubits. However before we do that, let's refer to the top-left and think about how many shifts this circuit would take. With a default consecutive enumeration of the qubits in the architecture, we need to shift between each gate in order to run this. However, referring to the right, if we simply reindex qubits 3 and 4, we can now run gates A and B in parallel! Thus reducing the number of shifts this architecture must take.

The "Block" Formulation for Ancillary Qubit Reindexing

Before we finally arrive at a somewhat useful problem, let's consider one more useless circuit and an intermediary problem. See Figure 4.

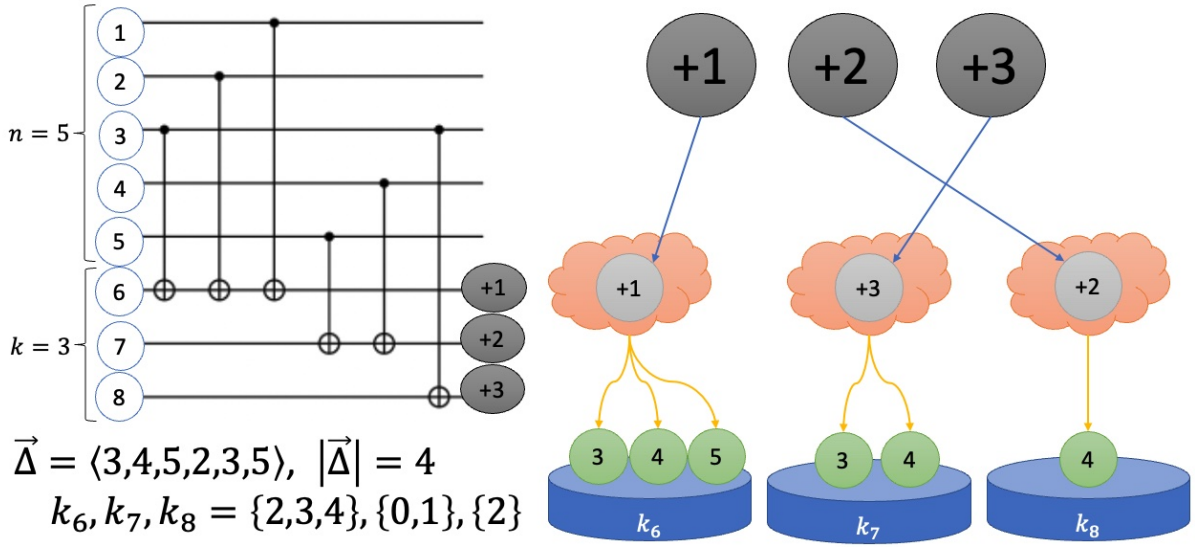

Figure 4: The $\vec{\Delta}$ is a vector of the different offsets between the gates. It can be proved that minimizing the number of distinct elements in this vector will be proportional to the number of shifts required (if we sort the gates by their offsets), hence $|\vec{\Delta}|$ is referring to the number of unique elements in $\vec{\Delta}$. The sets k_6, k_7, k_8 each correspond to their own ancillary qubit's gates' offsets respectively. The idea behind them is that if we swap qubit 6 and 7 in the circuit, all gates connected to qubit 6 will have their offsets increase, while all gates connected to qubit 7 will have their offsets decrease. The sets k_i reflect these values in the nonsensical scenario where all ancillary qubits are at the same position as the last data qubit. From this scenario, placing each "ancillary line of the circuit" back into a valid location will only ever increase values for each k_i. For example $k_6 = \{2, 3, 4\}$ will become $\{3, 4, 5\}$ if placed in position in it's original position of $+1$, $\{4, 5, 6\}$ if placed in $+2$, and so on. From this, we can consider k sets and k modifiers $\{+1, +2, ..., +k\}$. One way of viewing this problem is to find the assignment from the k_i's to the modifiers such that $|\bigcup_k \hat{k}_i|$ is minimized, where $\hat{k}_i$ is k_i after it has been augmented by it's assignment to a modifier. Notice that the above assignment is optimal as it reduces $|\vec{\Delta}|$ from 4 to 3. (Provably optimal because $|k_6| = 3$)

We can reinterpret the assignment problem introduced in Figure 4, by thinking of this as stacking blocks on a staircase. See Figure 5. The idea behind this representation is that now we can more easily visualize how to solve the problem. Furthermore, now our RL agent can more easily be conceived of; our RL agent will start with a set of blocks and a staircase. The RL agent will then need to choose a block to place on the staircase. Now the staircase has a one block on it, and the RL agent needs to choose the next block to place and where. This repeats until there are no longer any blocks left to be placed. If successful, the RL should place these blocks in a way that minimizes the number of columns occupied by at least one blue block.

With this "block" formulation, we can finally discuss the RL problem that we solved. In particular, the general case of this problem is extremely difficult (at least to model as a simple MDP with a reasonably sized finite state space), and seems like it would require some deep RL to solve it well, if it is even possible. Instead we chose one specific historically relevant circuit, the naive syndrome circuit for the Steane 7 qubit code [Ste96], and solved that one simple version of the problem. For brevity, we will merely introduce the block version of that circuit in Figure 6, and not discuss the circuit itself.

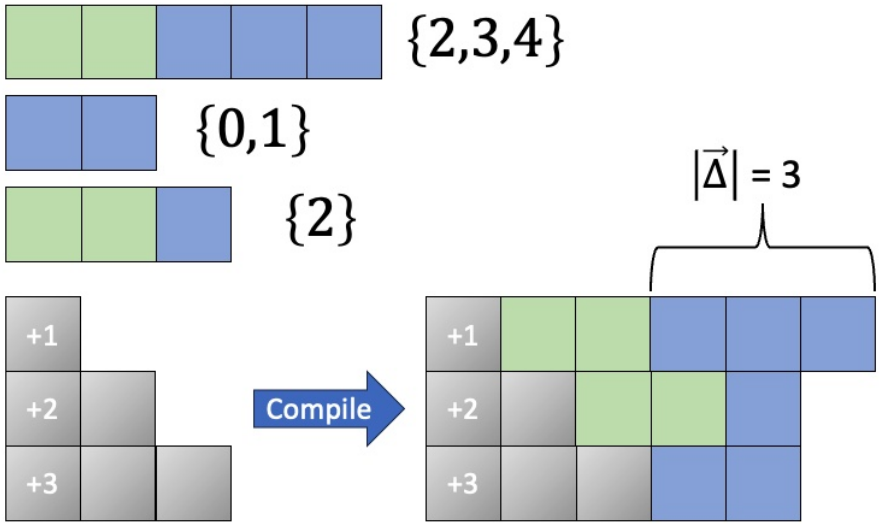

Figure 5: This is the same problem introduced in Figure 4, but we are now representing the k_i's as blocks and the assignment refers to how we are stacking these blocks on the stairs. The green blocks refer to as "air" and don't represent a gate. The blue blocks all represent gates within the quantum circuit, and the goal is to stack the blocks on the staircase such that the number of columns occupied by at least one blue block is minimized. Note: The initial chains of blocks can be viewed as stuck together and therefore must be placed as a single unit on the stairs. In other words, keep in mind the goal of this is to observed the final ordering of block placements and see how the different sets were permuted with respect to each other, as that corresponds to the initial problem of how to permute the ancillary qubit indices. Let us call these chains of blocks "Q-blocks"

Existing Techniques to Solve the Problem

To our knowledge, the only existing methods to solve this problem are currently unpublished [MK23], and those techniques are in fact, quite simple heuristics. The first of which is to sort the Q-blocks (the chains of blocks) by their total length. The other is to sort them by the length of their leading air blocks. Both of these methods break ties by employing the other heuristic. The solution found by those methods on the Steane 7 code uses 9 different columns. This has not been proven to be optimal, however we believe it to be so. If our RL agent can find solutions that are at least equally good to this, then our RL agent worked!

2 MDP Formulation

As mentioned, we will consider the naive syndrome circuit of the Steane 7 qubit code, which has its Q-Block representation shown in Figure 6. Unlike the more simple examples from the previous sections, the Q-blocks in this case will often have air embedded in them. Just like before, we will represent each Q-block by a set, and each element represents the location of each blue block. For example, the indices for the first q-block is {0,1,2,3}, so at positions 0, 1, 2, 3 there are blue blocks. Another example will be the second q-block, {0,1,4,5}. This has blue blocks are located at positions 0,1,4,5, and the position of the green blocks {2,3} are omitted. Below we will now discuss the various parts of the MDP:

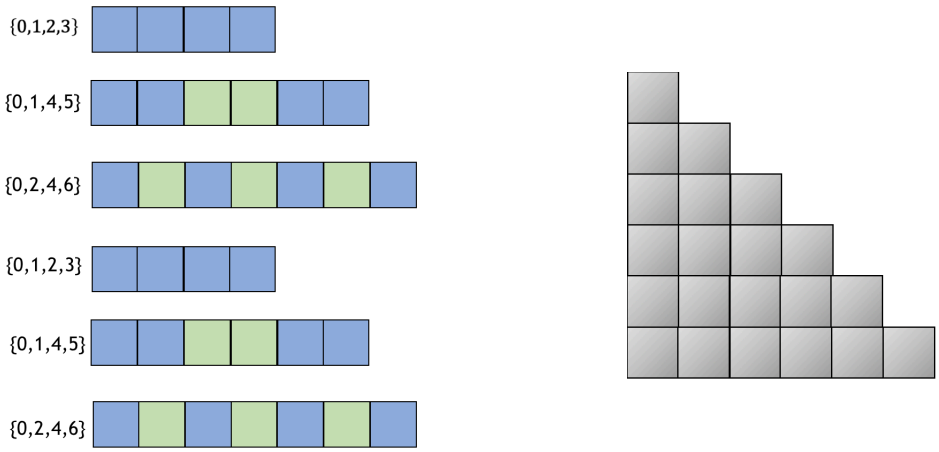

Figure 6: Block formulation of the Stene 7 qubit code's naive syndrome circuit

States

Let's first discuss how we going to naively represent the state using a matrix representation. From Figure 6, we can see there is a staircase 6 rows with different gray block lengths. This can be represented by a matrix and replace the gray block with 0. Now, let's imagine we put the first q-block on the first row of the stairs. Each Q-block will be represented by a binary vector and the indices next to each Q-block will show the position where the 1 will occur. For example, {0,1,2,3} will be represented as [1,1,1,1]. Let's say the Q-block is on the first row of the staircase. This can be represented as

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

In this state representation, both the air(green) block and the stairs themselves are represented by 0. If then placed second block on the second stair, then we would have:

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

While this matrix form was useful for calculating things like reward and the transition function, it was far more expressive than what we needed in order to solve the problem. To address this, we first noticed that in the case of the Steane code, we have exactly two copies of three species of Q-blocks. Realizing that let's us write the state as a 6 element vector which has elements equal to either 0, 1, 2, or 3. Each index of the vector corresponds to a position in the staircase, and the value at those indices hold the following meanings:

- 0 = This position in the staircase is currently empty
- 1 = This position is occupied by a [1, 1, 1, 1] Q-block
- 2 = This position is occupied by a [1, 1, 0, 0, 1, 1] Q-block

- 3 = This position is occupied by a [1, 0, 1, 0, 1, 0, 1] Q-block

So our state can be represented as any 6 element vector that can be interpreted as a submultiset of the multiset $\{1, 1, 2, 2, 3, 3, 0, 0, 0, 0, 0, 0\}$. This allows to calculate the total number of unique possible states as the following:

$$\sum_{a,b,c=0}^{2} \frac{6!}{(a+b+c)!(2-a)!(2-b)!(2-c)!} = 2074$$

Actions

Interestingly, we can express our actions as the set of possible states after the first time step. That is, we have 6 locations to place a block, and 3 species to choose from. So we represent this in the same notation as we do our states, however for actions there can must be exactly one non-zero element in each vector. Some examples of valid actions would be [0,1,0,0,0,0], [2,0,0,0,0,0], and [0,0,0,0,0,3]. There are 18 in total actions in this problem.

Transition Function

The vector representation of both states and actions made calculating $p(s, a, s')$, very simple. We could convert these vectors into matrices and simply add them together in order to observe the next state. However, of course, we needed to implement substantial testing of validity of states and transitions.

Some of the constraints we needed to implement were that taking an action didn't cause previously placed blocks to move, we can only transition from one state to another with one newly placed block, and that the state we transitioned to must match the action we took.

Reward Function

Since we want the agent to learn how to place these q-blocks, the number of aligned blue blocks in each column is maximized. The natural way to describe the reward function is described as the following:

$$R(s, a, s') = 10 * |A \cap B| \tag{1}$$

A is a set that contains the column indices of where there was at least one 1 in the matrix form of the new state (s'). B similarly contains the non-zero corresponding column indices of the matrix form of previous state (s). The reward is calculated to be 10 times the number of elements that are in both A and B. In other words, the agent will get a large reward if it chooses an action that results in a new state with many overlapping columns to its previous state. For example, using the two example states from the "States" subsection, that is we have $s = [1, 0, 0, 0, 0, 0]$ and $s' = [1, 2, 0, 0, 0, 0]$, there are two columns that overlapped, so our reward will be 20.

Other parts of the MDP: γ, d_0, and some notes

We used $\gamma = 1$, as we only cared about the final result of the agent. d_0 is always the initial state $[0, 0, 0, 0, 0, 0]$.

We implemented everything in python3.

3 Simulations

We made simulation of agent learning to play this game by using three different algorithm: Monte Carlo(MC) with epsilon policy, Dynamics Programming(DP), SARSA. For each algorithm, we showed the learning rate and the analysis of each algorithm's behavior.

Convergence Condition

For the following 3 algorithms, the convergence conditions are defined as the max norm of the difference of $v_t(s)$ and $v_{t+1}(s)$. The subscript here indicates the value function at different times. We set θ to **0.001**, the threshold to terminate the training process. In other words, we terminate the training process if:

$$|v_{t+1}(s) - v_t(s)|_\infty < \theta$$

Bellman's operator is a contraction mapping operator which means the convergence is guaranteed. At each iteration, we are applying Bellman's operator to the value/q-function, and $|v_{t+1}(s) - v_t(s)|_\infty$ is guaranteed to decrease. DP updates its value function at every iteration, and it is easy to check if it satisfies the terminate condition. On the other hand, MC and SARSA algorithms update the q-function at every iteration, so we obtained the v-function by the following formula:

$$v(s) = \sum_{a \in A} \pi(s, a) * q(s, a)$$

It is the weighted average of $q(s, a)$ at state s, and the optimal policy $[\pi^*(s)]$ will be a deterministic policy and calculated by the following formula:

$$\pi^*(s) = \underset{a \in A}{\operatorname{argmax}}\, q(s, a)$$

The terminal state (s_∞) of our MDP is defined as every row of the staircases contains a q-block which means the state vector contains all non-zero elements.

Monte Carlo with ϵ policy Methods

We modified Monte Carlo to avoid expensive computations and boost the runtime of the algorithm, so the length of each trajectory is 6.

Let's say an agent picks an action from one of the 18 actions and it is not allowed. For example, the first of the stairs already has a q-block and the agent is going to put another q-block on the first row. This action will not change the state and the agent will get an reward of 0. This is a similar situation to 687 grid world: the agent is next to a wall, but it is trying to get past the wall. The modification that we did is the agent will choose an action that results in the change of the state. This shortens the unnecessary computations in each trajectory, thus speeding up the runtime. This will be a first-visit Monte Carlo because each state will be visited exactly once, and due to the nature of first-visit MC, we have an unbiased estimator.

At every episode only 6 of the $v(s)$ will be changed, therefore, it is desirable to terminate the algorithm when $\delta < \theta$ and δ is not equal to 0 because there is some probability the same converged 6 non-optimal states got selected and other states are not converged yet. It is necessary to introduce that extra condition: δ not equal to 0. Let's consider the following odd situation, the 6 states that converged are bad states, and those same 6 states got selected for two consecutive episodes which will cause the early convergence and result in

a sub-optimal solution. This problem is caused by the length of the trajectory being way smaller than the cardinality of the state space.

For this method, the only hyperparameter that we have is ϵ which is the probability of selecting a non-optimal action. We also modified the ϵ policy algorithm. We started with $\epsilon = 0.9$, but ϵ decreases by a magnitude of 0.9 for each episode. This makes sense because in the beginning the q-function is set arbitrarily and we shouldn't trust the update that much because the q-function is an approximation, so we want a big change in updating the q-function. However, as the number of episodes increases, we should trust the update more because Bellman's operator is a contraction mapping operator so we don't want a big change in the q-functions. We ran this algorithm multiple times and got a consistent optimal result. We found the optimal result empirically, the minimum number of overlapped blue columns is 9 or the maximum possible rewards is 150. The discounted reward γ we set to 1 because it doesn't matter to us if agent can solve it faster or slower, the goal is we want the agent to find the optimal solution. The learning curve for MC algorithm is shown in figure 7.

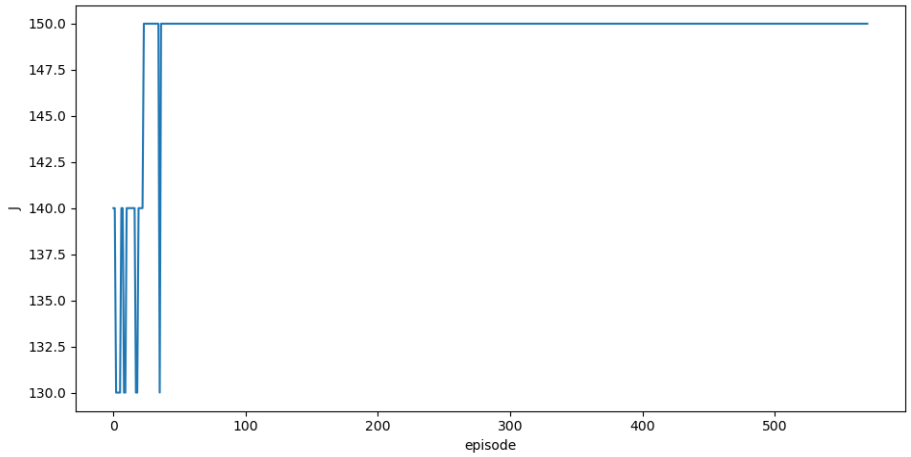

Figure 7: The learning curve for MC algorithm with epsilon policy.

We can see that around 70 episodes, the agent found the optimal solution and consistently getting a reward 150. This is an indication of our agent successfully learned the optimal policy.

We also made a simulation using the optimal q-functions that we found and it is shown in Figure 8.

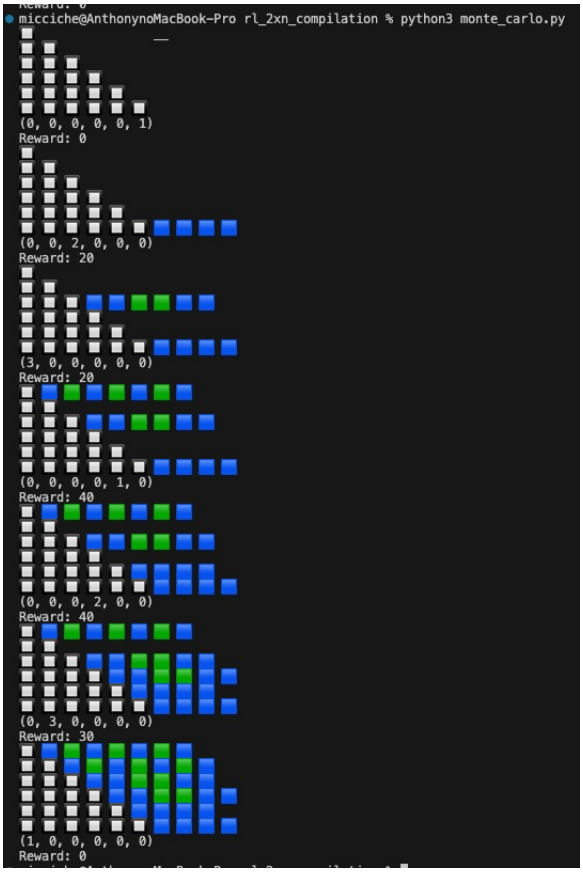

Figure 8: The simulation for MC algorithm. Time progresses from top right to bottom right. Each time step, it shows the staircase, the action that the agent chooses, and the reward that the agent received based on the state and action.

Dynamic Programming

The learning curve for the dynamic programming is shown in figure 9:

It is interesting that dynamic programming only needs 1 iteration to find the optimal solution, and only 3 iterations to converge. However, Monte Carlo needs 70 iterations to find the optimal solution and 550 iterations to converge. This may be affected by the stochastic nature of Monte Carlo. Since the state space is big(2073 states) but the number of states that is changed in each iteration is 6. States that are changed in each iteration are generated by the probability distribution defined by $\pi(s)$ with ϵ policy which means some state will occurred multiple times and it will require more iterations to make all v-values of every state converges. In the case of DP, we are updating all 2073 state-values simultaneously in each iteration and there is no redundancy in updating the converged v-values.

SARSA

SARSA also found an optimal solution, however tuning the hyper-parameters, especially the way in which to decay ϵ was particularly tricky. A setup we found to work was to use $\alpha = 0.3$ and initially $\epsilon = 0.99$. After each episode, we multiplied ϵ by 0.99999 and ran 100,000 episodes. This makes SARSA by far the most computationally intensive of the three methods we tried. Although perhaps there exists a less computational

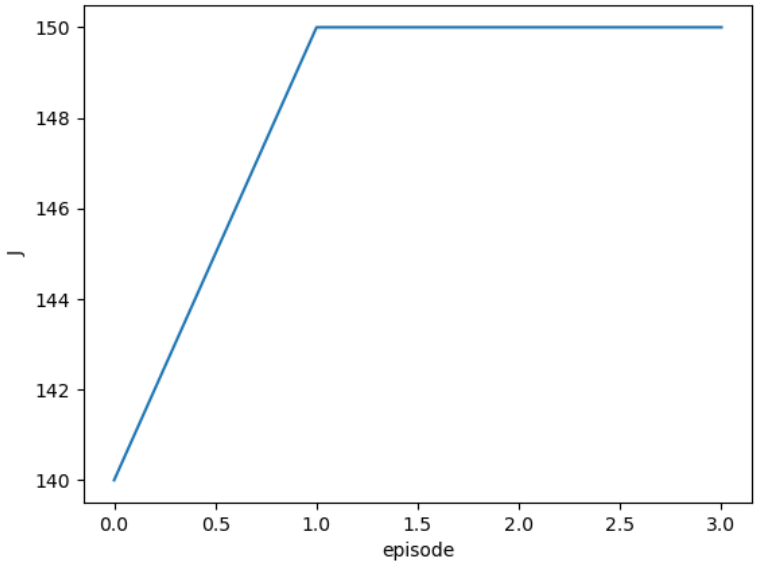

Figure 9: The learning curve for dynamic programming

intensive setup of the SARSA algorithm for our purposes. Refer to Figure 11 for the behavior of our agent following the π^* learned by SARSA and to Figure 10 for a plot of the performance of the agent against the episode number.

4 Conclusion

We implemented three algorithms from scratch, SARSA, MC, and DP, as well as the MDP. All three methods found optimal solutions. It seems that either DP or MC would be the most promising if we were to try to extend this work to other more difficult quantum circuits, or if we were to attempt something that could solve this problem generally.

References

[Ste96] Andrew Steane. "Multiple Particle Interference and Quantum Error Correction". In: *Proceedings of the Royal Society of London. Series A: Mathematical, Physical and Engineering Sciences* 452.1954 (Nov. 1996), pp. 2551–2577. ISSN: 1471-2946. DOI: 10.1098/rspa.1996.0136. URL: http://dx.doi.org/10.1098/rspa.1996.0136.

[MK23] Anthony Micciche and Stefan Krastanov. "Quantum LDPC Error Correcting Codes for use on 1D Quantum Dot Arrays". In: *Unpublished* (2023).

[Zwe+23] A.M.J. Zwerver et al. "Shuttling an Electron Spin through a Silicon Quantum Dot Array". In: *PRX Quantum* 4 (3 July 2023), p. 030303. DOI: 10.1103/PRXQuantum.4.030303. URL: https://link.aps.org/doi/10.1103/PRXQuantum.4.030303.

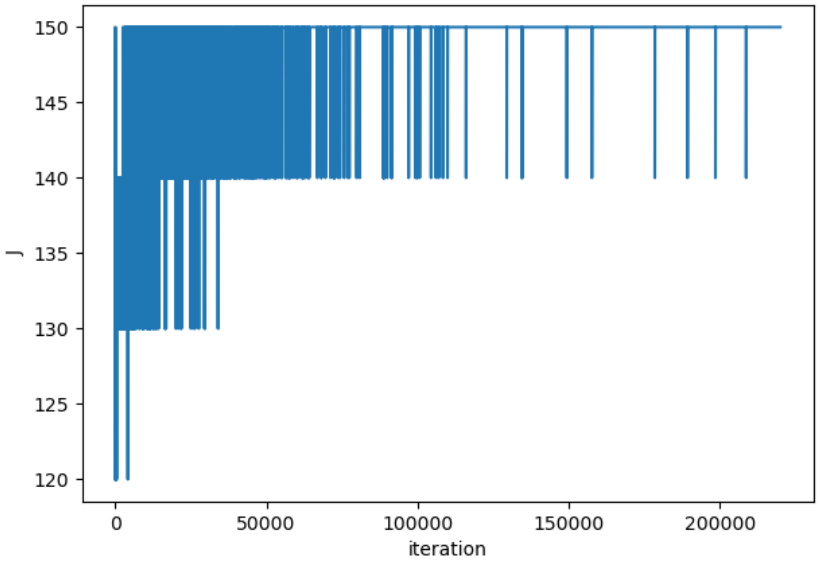

Figure 10: The learning curve for SARSA

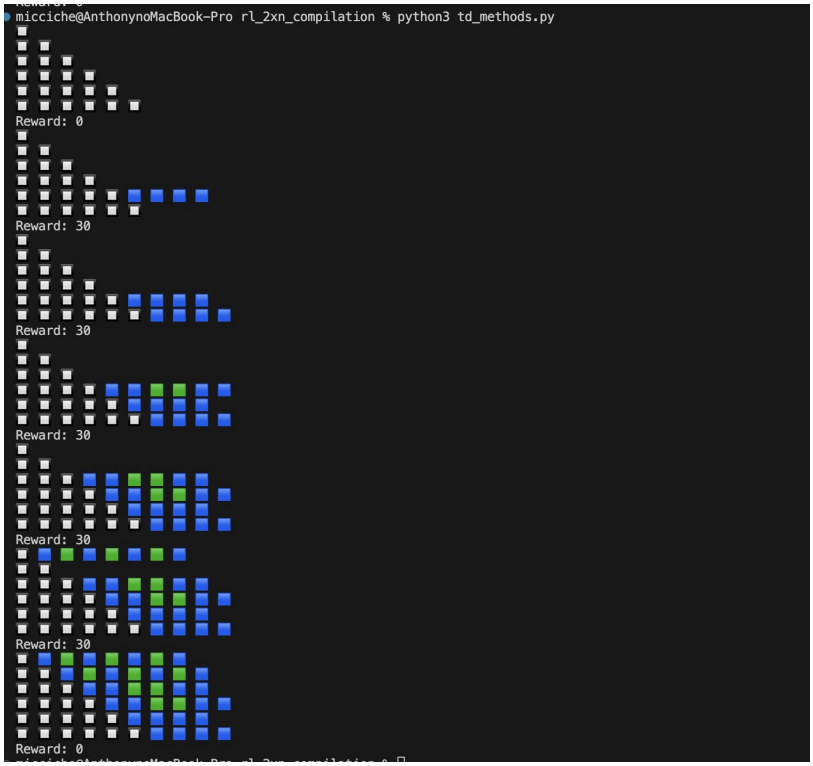

Figure 11: Output of running SARSA on our problem