CSE 326 Analysis and Design of Algorithms

Dr.Walid Gomaa

Name	**ID**
Mohamed Abdelmonem Makram	120220055
Abdelrahman Ahmed Shaheen	120220228
Abdelrhman Mohamed Eldenary	120220253
Anas Ihab Badr	120220360

Computer Science Engineering Department
Egypt-Japan University of Science and Technology

Contents

Schur Numbers Week 14 Report

May 23, 2025

Since this is the final report, we will be summarizing our journey through the Schur numbers problem.

1 The Beginning

In the first couple of weeks, we focused on understanding the problem and the recent literature where we found the **Schur number five** paper published in 2017. This was the current state of the art for this problem where they used massively parallel SAT solver, supported by new heuristics for decision-making and problem partitioning to find $S(5) = 160$ and prove it. However, they used too much computing power about 14 CPU years that was only achieved in 3 days using a Lonestar 5 that features 1252 Cray XC40 compute nodes, each equipped with two 12-core Intel Xeon processors, totaling 30,048 compute cores. Their problem is their solution is hardly scaclable to larger Schur numbers like $S(6)$ due to the exponential growth nature of the problem.

We tried exploring the problem on our own pace without using any supercomputers and with no external solving methods. We tried using multiprocessing, since it was one of the criteria said in the lecture, to have a parallelized solution. We used Python at first with some randomized algorithm, then C++ to try to brute force the problem for smaller numbers. Which worked for $S(3)$ and $S(4)$, but it took $S(4)$ about 5 minutes to solve. Our next step was to try and used CUDA, and the massive parallelization of the GPU to solve the problem.

2 CUDA

2.1 Learning CUDA

Since we had minimal experience with multiprocessing let alone CUDA, we had to learn at least the basics of how CUDA can be used to solve problems. We understood various concepts like how the data is copied from and to the GPU, how the threads are organized in blocks and grids, and how to use shared memory. Our first CUDA program at week 6 was just trying to check the validity of a coloring where each thread would check a color individually. This was not a very big difference, but it was a start.

2.2 Novel encoding approach

After some weeks with no idea what we were doing, a brilliant idea came to us. Traversing the problem as a tree in a DFS manner. We introduced a novel bit mask encoding approach to represent the coloring as 2 64-bit integers for each color such that each bit is turned on in only one color which represents the color of the index of that bit. For example a coloring like this:

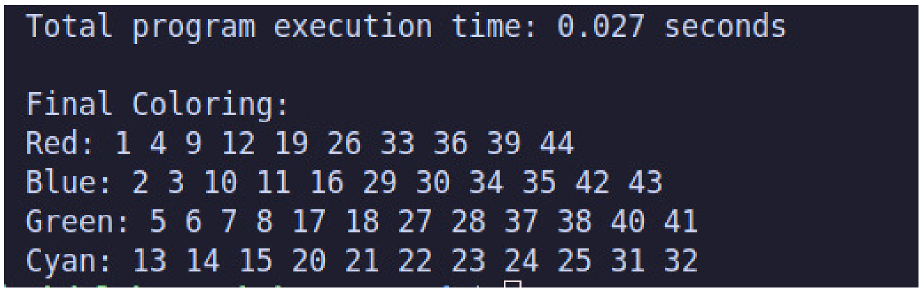
$$
\begin{aligned}
C_0 &= \texttt{0000001001001001}_2 \\
C_1 &= \texttt{0000000100010010}_2 \\
C_2 &= \texttt{0000000010100100}_2
\end{aligned}
$$

would mean that the number 1, 4, 7, 10 are colored using C_0, the number 2, 5, 9 are colored using C_1, and the number 3, 6, 8 are colored using C_2 (We used 2 64-bit integers to represent number from 1 to 128). This allowed for less memory usage, faster access, and faster validation of the coloring. Then for traversing the problem, we would initiate a coloring of length 4. Then our algorithm would be as follows:

1. Iterate through all the current coloring in `current_states` in a parallelized manner. (starting with one coloring)
2. Check if the next number can be colored with any color in the previous coloring.
3. If yes, store the coloring with the new number in `next_states` buffer and continue.

This way we check all the possible colorings while utilizing the GPU threads to check for valid next state colorings. A diagram of the tree is shown in `tex/images/recursionTree.png` (it was too large to fit in the pdf).

The code preformed very well and we could obtain $S(4)$ in 30 ms compared to 5 minutes in Python.



```
Total program execution time: 0.027 seconds

Final Coloring:
Red: 1 4 9 12 19 26 33 36 39 44
Blue: 2 3 10 11 16 29 30 34 35 42 43
Green: 5 6 7 8 17 18 27 28 37 38 40 41
Cyan: 13 14 15 20 21 22 23 24 25 31 32
```

Figure 1: CUDA code output

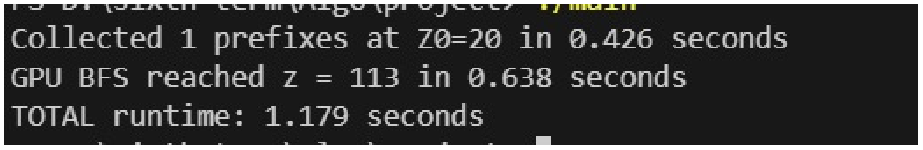
2.3 Scaling the algorithm

2.4 Initial attempt

We then tried to scale the algorithm to $S(5)$, by using 3 64-bit integers to represent number from 1 to 192 (as $S(5)$ is 160). We only reached a coloring of length **81** before we ran out of memory to represent all the states (Over 37 million states). But this was not bad at all. Compared to what took the people from the Schur number five paper, this was quite impressive.

2.5 Randomized Monte Carlo

Since we didn't need all the states from the beginning, we decided to try a randomized approach where we would use Monte Carlo rollouts on the CPU to generate diverse partial colorings up to a fixed depth, and then offloads successful rollouts to the GPU, where we do our breadth-first search normally with fewer states. After that randomized approach, we saw a huge improvement where we could reach a coloring of length **113** before we ran out of memory again. This was a huge improvement, and we were very happy with the results.



```
Collected 1 prefixes at Z0=20 in 0.426 seconds
GPU BFS reached z = 113 in 0.638 seconds
TOTAL runtime: 1.179 seconds
```

Figure 2: Monte Carlo approach

3 Reinforcement Learning and Transformers

Although our parallelized approach was promising, and we could reach a coloring of length 113 with using only our laptops, we had the passion to try and use something that didn't exist in 2017. We wanted to try and use reinforcement learning to solve the problem. We believed that this may be a very good approach, since it was something unexplored in the Schur numbers problem, and that maybe we are on the verge of a breakthrough (we were not).

3.1 Reinforcement Learning

What first came to mind when we draw our `recursionTree` diagram is that it was too similar to what Google DeepMind did with AlphaGo (Figure 3). They used a Monte Carlo tree search to traverse the game tree and then used a neural network to evaluate the states. We though the same logic could be applied to our problem. We could use a Monte Carlo tree search to traverse our states, evaluate the states according to how deep they managed to get the tree to, and design our reward function to be proportional to the depth of the tree.

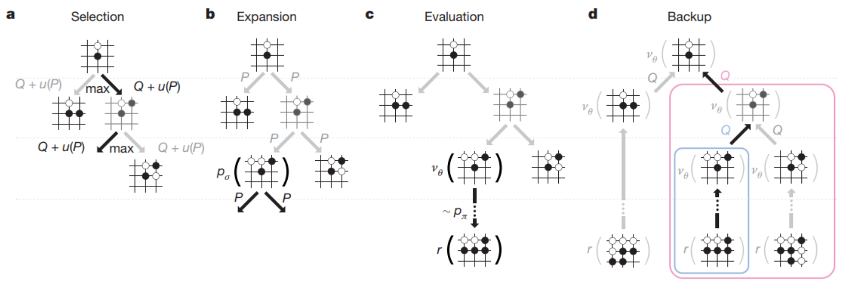


Figure 3: AlphaGo model

Again, we had minimal experience with reinforcement learning, but we tried implementing the mentioned algorithm in a very short time. The results were not that great achieving only colorings of length 28. But to think that it reached there from scratch was no data is still quite impressive.

3.2 Transformers

With a novel idea this time, we thought of treating the problem as a sequence of numbers and colors where the transformers should be able to learn the patterns of the colorings without finding a monochromatic solution to the equation $x + y = z$. This time we generated our own dataset of colorings using the CUDA code we wrote before. We generated 1000 coloring for each depth from 1 to 70 (the maximum our initial CUDA approach could reach) then we initially used an encoder only transformers to try to understand the nature of the sequence and come up with the next color. The transformer processes variable-length sequences by embedding the numbers and their corresponding colors separately, then concatenating them into unified token representations. A learned positional encoding is added to help the model understand sequence order. The core of the architecture is a lightweight multi-head, multi-layer Transformer encoder, and the output corresponding to the last token is passed through a linear layer and softmax to produce a probability distribution over the five possible colors. The model is trained using a standard cross-entropy loss on valid color predictions. Each training example is a prefix of a sequence, and the target is the color of the next number. After training the transformer locally the results were somehow decent. In figure 4 it shows the results of taking the average output length the transformer could reach for 500 trials for each input length from 6 to 70.

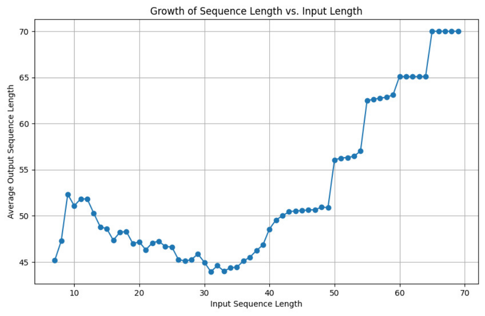


Figure 4: Transformer model results

In this final week we tried to fine tune the transformer that has some good initial predictions using Reinforcement learning to try to improve its results. We used actor-critic method to train the transformer. The actor is the transformer model, and the critic is a neural network that evaluates the action taken by the actor. The critic is trained to predict the expected reward for a given state-action pair, and the actor is trained to maximize the expected reward. The training process involves sampling actions from the actor, evaluating them using the critic, and updating both networks based on the observed rewards. This approach allows the transformer to learn from its own predictions and improve over time.

The results of the fine-tuning methods, shown in figure 5 didn't improve the original transformer model much, but it made it smoother in a way. We believe that this may come from the training of the original transformer that made it overfit the data. The RL fine-tuning was not able to generalize the model enough above the training data.

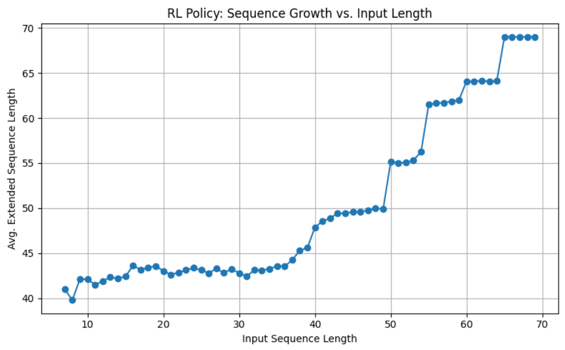


Figure 5: Finetuned Transformer with RL results

3.3 Final Approach

One final approach we are willing to try even after this final report is using a decoder only transformer (like GPT) to generate the sequence iteratively while using causal masking to ensure that the model only attends to previous tokens in the sequence. This would allow the model to generate the sequence in a more natural way, similar to how humans would do it. We believe that this much better than the encoder-only approach since it would be natural to generate the sequence in a left-to-right manner.

4 Conclusion

In conclusion, we have explored various approaches to solve the Schur numbers problem. We started with a brute-force approach using Python, then moved to C++ and CUDA for parallelization. We introduced a novel encoding approach and a randomized Monte Carlo method to improve our results where we achieved our best results of reaching a sequence of length 113. Finally, we explored reinforcement learning and transformers to tackle the problem in a more sophisticated way. Although we did not achieve the desired results using Deep Learning, we learned a lot about the problem and the techniques we used. We hope that our work will inspire others to continue exploring this fascinating problem.