ECE 459: Programming for Performance
Assignment 1

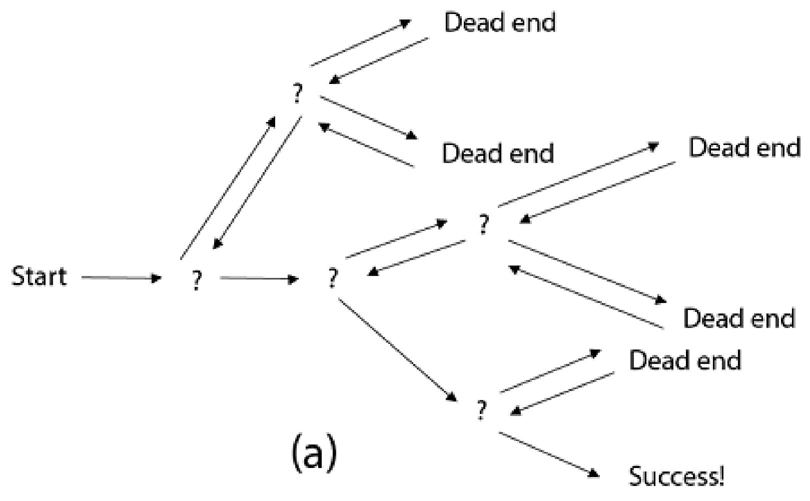
Zahin Mohammad

January 27, 2021

1 Sudoku Solver

1.1 Algorithm

The algorithm used for the sudoku solver is a brute force solution using recursion and back-tracking. The algorithm will do a Depth First Search for all possible input's for all empty squares. If the algorithm gets into a state where there are empty squares left on the board, but there are no more possible options, then the algorithm will traverse back up the call-stack to indicate that that solution path doesn't work, and will try a new input (hence backtracking). Once a solution is found, the algorithms recurses back up the call-stack to the root caller and returns. An example of backtracking can be seen in Figure 1, where each "Dead end" is a puzzle with an incorrect solution, and "Succcess!" is a correct solution. The pseudo code for this algorithim is shown in Listing 1.



Source:
https://www.javatpoint.com/backtracking-introduction

Figure 1: Depth First Search

1.2 Possbile Optimizations

The algorithm described in 1.1 is an un-optimized brute force solution. An alternative strategy is to make use of stochastic search techniques such as Simulated Annealing . By introducing a score, or a measure of how "good" a solution is, we can lead the algorithm down a more focused path instead of attempting all solutions. In the case of the sudoku puzzle, a score that can be employed by the simulated annealing algorithm is the sum of violations in the solution in each row, column and sub-grid, and with this the simulated annealing algorithm would aim to find a solution that minimizes this score. The downside to simulated annealing is the possibility of getting stuck near the end of the search and not resulting in a solution, but the optimization of the optimization will have to be discussed in another report.

Listing 1: Sudoku solver python pseudo code

```
def solve(puzzle, square) -> bool:
    # Loop all valid sudoku square values, values [1,9]
    for guess in range(1,10):
        # DFS by calling solve with new puzzle
        # Back track to caller if this is the correct solution
        puzzle[square] = guess
        if solve(puzzle, square.next):
            return True
        # Revert to last valid puzzle state and try next guess
        puzzle[square] = None
```

2 Nonblocking I/O

The provided code to verify the sudoku solutions used synchronous I/O. An optimization to this was to leverage concurrency to verify multiple solutions at the same time. Using "Hyperfine", the different methods can be quantified, and they are explored in subsequent sections.

2.1 Using Curl Multi

To achieve concurrency, the *curl multi* crate was used. Using *curl multi*, all the puzzles that are to be verified are registered with *curl multi* and then *curl multi* will verify up to a specified amount of puzzles concurrently. Within this lab, 100 puzzles where set to be verified at a time to reduce the load of the sudoku verifier server. The request response handling is done once all the requests have been finished, and *curl multi* no longer has any requests remaining. The rust code to achieve this is present in file `/src/verify.rs`.

2.2 Benchmarking Blocking vs Non-Blocking

During the first benchmarking test, the number of puzzles tested where [1, 10, 100, 200, 300, 400, 500] for both blocking and non-blocking I/O with 100 max connections for the non-blocking case. This test was to highlight the stark difference between non-blocking and blocking I/O. It can be seen in Figure 2 and Table 3 that Non-Blocking I/O is much more preformat than Blocking I/O, almost by a factor of 86 for the 500 puzzles case. The data used to generate the graph is show in Table 1.

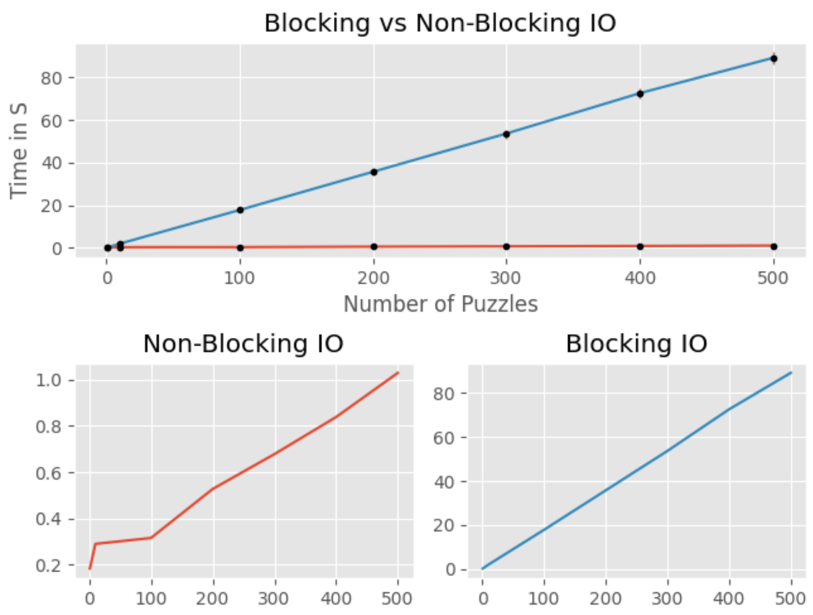


Figure 2: Benchmarking results for blocking vs non-blocking I/O

2.3 Benchmarking Number of Connections

During the second benchmarking test, non-blocking I/O was used with a constant 500 puzzles while the number maximum connections configured for *curl multi* where varied with values [3, 4, 16, 32], and this was compared against the blocking I/O case. The results can be viewed in Figure 3 and Table 2 (the timing for 500 puzzles for blocking I/O value is seen in Table 1). The results seem as expected. Comparing the case of 16 max connections vs 32 we can see that we see the mean time reduced by half, which correlates to the doubling of the maximum connections.

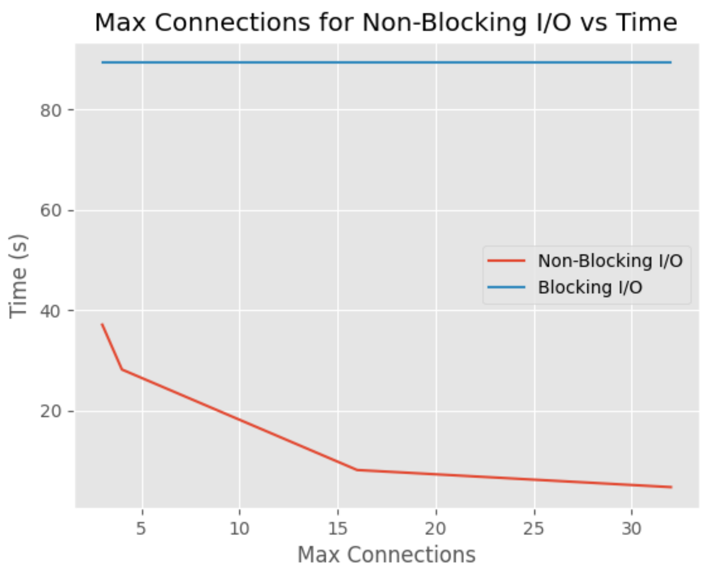


Figure 3: Benchmarking results for non-blocking I/O

An additional test was performed with just the non-blocking I/O with varying maximum connections between 10 to 100 with increments of 10. The results can be seen in Figure 4, and it is

Number of Puzzles	Method	Mean (s)	Min (s)	Max (S)	Standard Deviation
1	Blocking	0.1887	0.0816	0.2595	0.0581
10	Blocking	1.924	1.736	2.168	0.148
100	Blocking	17.76	16.889	18.903	0.632
200	Blocking	35.72	33.921	37.541	1.15
300	Blocking	53.676	51.359	55.29	1.303
400	Blocking	72.627	69.954	75.082	1.542
500	Blocking	89.235	86.081	92.017	1.687
1	Non-Blocking	0.1825	0.1039	0.3073	0.0791
10	Non-Blocking	0.2892	0.2368	0.3098	0.0224
100	Non-Blocking	0.3147	0.3107	0.3202	0.003
200	Non-Blocking	0.5269	0.4986	0.5415	0.0132
300	Non-Blocking	0.6779	0.6536	0.7149	0.017
400	Non-Blocking	0.8387	0.8073	0.8753	0.019
500	Non-Blocking	1.03	0.985	1.317	0.102

Table 1: Benchmarking results for blocking vs non-blocking I/O

Max Connection	Mean	Min	Max
3	37.090501324805	35.912743072505	38.453410431505
4	28.164856773605	27.198395150505	29.169373326505
16	8.143694881105	7.027582561505	8.723201447505
32	4.735995889105	4.100847059505	5.558831242505

Table 2: Benchmarking results for non-blocking I/O compared

Max Connection	Mean (s)	Median (s)	Min (s)	Max (s)
10	12.04420542087	11.76495236617	11.13261502317	13.33893492117
20	6.52144425707	6.53465773117	5.53100086917	6.93016539317
30	4.72316604067	4.73300261067	3.78523025517	5.74956142817
40	3.31534770707	3.06802457117	2.86814998717	4.19138124117
50	3.31214952327	3.30801180867	2.59449317617	3.73973779817
60	3.05853981627	3.10191731517	2.66808086117	3.44505849317
70	2.72070087527	2.73164874617	2.54002154517	2.83269365417
80	2.62072510597	2.62515269467	2.36529386817	2.97473718717
90	2.84280894827	2.81315018817	2.15230128017	3.71023862317
100	2.94937009417	2.95367552867	1.88731003417	3.68511938917

Table 3: Benchmarking results for non-blocking I/O

clear that the larger the maximum number of connections, the faster the program finishes. The theory behind this result is that the more connections there are, the more concurrent requests are being made i.e. more network requests per second. It is to be noted that the rate of increase of performance decreases as the number of connections increase. This indicates that after a certain point, there is no additional value in increasing the number of maximum connections. From Figure 4, the point where the rate of change seems to decrease/slow down is around 40-60 maximum connections.

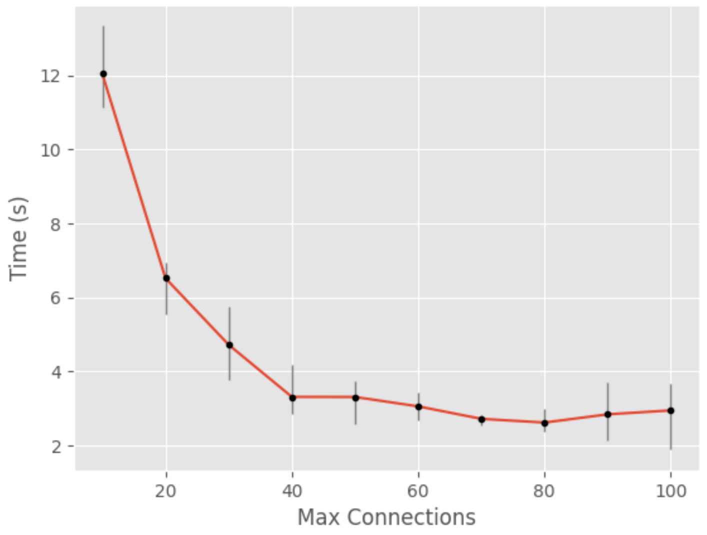


Figure 4: Benchmarking results for non-blocking I/O