ECE459 Lab 1

Jackson Guo
j84guo@uwaterloo.ca

1 Solving Sudokus

1.1 Problem Formulation

A 2D 9x9 grid of values can be visualized as a 1D sequence of 81 values. For example, the following Sudoku

5	3			7				
6			1	9	5			
	9	8					6	
8				6				3
4			8		3			1
7				2				6
	6					2	8	
			4	1	9			5
				8			7	9

can be converted into the sequence

5, 3, -, -, 7, -, -, -, -, 6, -, -, 1, 9, 5, -, -, -, …

where we have placed the grid's cells into the sequence starting from the top-left corner, going left-to-right and top-to-bottom (empty cells are shown as a dash).

The problem of solving a Sudoku puzzle can be formulated as taking such a sequence of 81 values and replacing all the dashes with integers. The resulting sequence of 81 integers must satisfy the following constraints:

- all integers between 1 and 9 (replace dashes with valid integers)
- when arranged in grid form:

- no rows have duplicates
- no columns have duplicates
- none of the 9 3x3 sub-grids (drawn with dark black lines above) have duplicates

1.2 Recursive Algorithm

Consider a 1D representation of a Sudoku as shown in section 1.1. As an invariant to the algorithm, we require that any integers in the initial sequence are valid (do not violate Sudoku constraint). We can try to fill in all the dashes (unfilled values) with a recursive algorithm which accepts a Sudoku puzzle and returns true if it can solve it, and false otherwise. Starting from the first value in the 1D sequence:

1. If the value is already an integer, it must be valid. Repeat step 1 starting from the next value (recurse immediately), or return true if this was the last value in the sequence.
2. If the current value is not yet filled, try to use each integer from 1-9 to fill it in. For any integer which can be used as the current value without violating any Sudoku constraints, set the current value to that integer and recurse on the next value.
    a. If the recursive call returns true, we have found a valid Sudoku and can return true as well.
    b. If the recursive call failed, reset the current value to unfilled and try the next possible integer.
        i. This “backtracking” causes the state changes of any recursive call that returns false to be reverted, while those made by any recursive call that returns true are kept (so that the caller sees a valid filled-in Sudoku if the top-level recursive call succeeds).
    c. If no integers fit into the current value, return false; we cannot make a valid Sudoku given the input sequence to this recursive call.
3. Move on to the next value and repeat step 1. (e.g. left-to-right, top-to-bottom).

The Rust-like pseudocode below illustrates the algorithm as described above:

```
// Fill in unfilled values for the sudoku board starting from row r and column c.
// Walk left-to-right and top-to-bottom.
fn solve_sudoku_from(board: &mut Sudoku, r: usize, c: usize) -> bool {
   match board[r][c] {
       // The current value is already a valid integer
       Some(_d) => {
           // Recurse on the next value (left-to-right, top-to-bottom)
           if c + 1 < board[0].len() {
               return solve_sudoku_from(board, r, c + 1);
           } else if r + 1 < board.len() {
               return solve_sudoku_from(board, r + 1, 0);
           }
           // All values filled
           return true;
       },
       // The current value is not yet filled
```

```
            None => {
                // Try each integer from 1 to 9
                for i in 0usize..=8 {
                     // If the integer would violate any Sudoku constraint, skip it
                    if constaints_violated(board, i) {
                        continue;
                    }
                    // Try setting the current value to the integer i
                    board[r][c] = NonZeroU8::new((i + 1) as u8);
                    if c + 1 < board[0].len() {
                        if solve_sudoku_from(board, r, c + 1) {
                            return true;
                        }
                    } else if r + 1 < board.len() {
                        if solve_sudoku_from(board, r + 1, 0) {
                            return true;
                        }
                    } else {
                        // All values filled
                        return true;
                    }
                    // Reset the current value to None since i didn't work
                    board[r][c] = None;
                }
                 // No integer could be filled in, the input puzzle is invalid!
                return false;
            }
        }
    }
```

1.3 Using Data Structures To Improve Runtime

In the pseudocode above, the implementation of `constaints_violated` is not shown. We need to check that the current value's row (`r`), column (`c`) and subgrid (call it `g`) do not already contain the integer we are considering (`i`). Note that each of (`r, c, g`) are integers in [0, 8];

A naive approach would be to iterate through the board each time these constraints need to be checked, incurring the cost of iteration through the 9 values in a row, column and grid for each call to `constaints_violated`. While these iterations are technically bounded by the fixed length of the grid, 9, we can reduce the constant factors of the runtime by using set data structures to store the integers currently filled in each row, column and grid - incurring O(1) lookups, additions and removals.

Each set stores the integers of a distinct row, column or grid, resulting in 9 + 9 + 9 = 27 total sets. This implementation chooses to implement these sets as 3 9x9 matrices of booleans (1 each for the rows, columns and subgrids, respectively), which are looked up by the row/column/subgrid number and the value that we'd like to check (minus 1 since Rust arrays use zero-based indexing).

```
// Sets to lookup the currently filled values in each row, column and subgrid
let mut row_vals = [[false; 9]; 9];
let mut col_vals = [[false; 9]; 9];
let mut grid_vals = [[false; 9]; 9];
```

These sets need to be updated in step 2 of the algorithm description from section 1.2, to reflect our attempts to use each integer (`i`) as the current value.

```
board[r][c] = NonZeroU8::new((i + 1) as u8);
row_vals[r][i] = true;
col_vals[c][i] = true;
grid_vals[g][i] = true;

// Recurse, returning true immediately if successful...

board[r][c] = None;
row_vals[r][i] = false;
col_vals[c][i] = false;
grid_vals[g][i] = false;
```

Finally, how can we compute the subgrid index (that we named `g`)? Since the board's dimensions are constant (9x9) we can statically allocate a mapping from coordinates to subgrid number instead of re-computing them each time we need to access (`g`).

```
fn grid_num(r: usize, c: usize) -> usize {
   // We could compute the grid num each time it's needed, but it's faster to make the
grid nums
   // statically allocated - there are only 9x9=81 of them anyways.
   //
   // If we wanted to compute the grid nums, the formula is:
   // let mut g = r / 3;
   // g *= 3;
   // g += c / 3;
   // return g;
   static GRID_NUMS: [[usize; 9]; 9] = [
        [0, 0, 0, 1, 1, 1, 2, 2, 2],
        [0, 0, 0, 1, 1, 1, 2, 2, 2],
        [0, 0, 0, 1, 1, 1, 2, 2, 2],
        [3, 3, 3, 4, 4, 4, 5, 5, 5],
        [3, 3, 3, 4, 4, 4, 5, 5, 5],
        [3, 3, 3, 4, 4, 4, 5, 5, 5],
        [6, 6, 6, 7, 7, 7, 8, 8, 8],
        [6, 6, 6, 7, 7, 7, 8, 8, 8],
        [6, 6, 6, 7, 7, 7, 8, 8, 8],
   ];
   return GRID_NUMS[r][c];
}
```

Part 2: Nonblocking I/O

2.1 Sudoku Verification

To verify a completed Sudoku, one simply checks that each value is an integer in the range [1, 9] and that no row, column or subgrid has duplicates (sets or arrays can be used to remember the contents of the current row/column/grid when checking the a cell value for duplication). This is performed by the server; it is also implemented for debugging purposes in lib.rs as `check_puzzle()`.

2.2 Event Loop Using libcurl's Multi Interface

Asynchronous I/O is performed by building an event loop using libcurl's Multi interface. We register an Easy2 handle for each puzzle with a Multi handle, since each puzzle is verified using a separate HTTP request (based on the starter code in verify.rs). The Multi is configured to use a maximum number of total connections in its connection pool. Note that HTTP pipelining is enabled for each benchmark scenario.

```
let mut multi = multi::Multi::new();
multi.pipelining(true, true)?;
multi.set_max_total_connections(max_total_connections)?;
```

We then repeatedly call `multi.wait()`, which internally uses the `poll()` system call to make the calling thread wait for any of the underlying non-blocking socket descriptors used by the Easy2 handles to be ready for I/O. At that point, `multi.perform()` performs as much reads/writes as possible for each ready socket without blocking (i.e. on Linux, until the underlying `send` and `recv` return EWOULDBLOCK). The loop repeats until all transfers have finished.

```
// Wait until they're all done
while multi.perform().unwrap() > 0 {
   multi.messages(check_msg); // Panic on connection error
   multi.wait(&mut[], std::time::Duration::from_secs(30)).unwrap();
}
multi.messages(check_msg); // Panic on connection error
```

Once all transfers are complete, we remove all Easy2 handles from the Multi handle and count those which succeeded and received a positive verification from the server.

```
for owned_easy in owned_easies.into_iter() {
   let easy = multi.remove2(owned_easy).unwrap();
   if easy.get_ref().result {
       verified += 1;
   }
}
```

2.3 Benchmark Results

Benchmarks were collected for 100.txt and 10.txt.

2.3.1 100 Puzzles

The benchmarking output below is for running the program in verify mode on the 100 puzzle solutions in solved/100.txt with libcurl set to use varying maximum numbers of concurrent connections.

(1 Connection; no Multi - invoking Easy2::perform() sequentially)

```
Time (mean ± σ):     17.774 s ±  0.593 s    [User: 36.3 ms, System: 32.0 ms]
  Range (min … max):   17.034 s … 18.796 s    10 runs
```

(3 Connections, Multi)

```
Time (mean ± σ):      5.378 s ±  0.261 s    [User: 79.0 ms, System: 17.6 ms]
  Range (min … max):    4.927 s …  5.826 s    10 runs
```

(4 Connections, Multi)

```
Time (mean ± σ):      4.029 s ±  0.217 s    [User: 77.6 ms, System: 11.4 ms]
  Range (min … max):    3.721 s …  4.300 s    10 runs
```

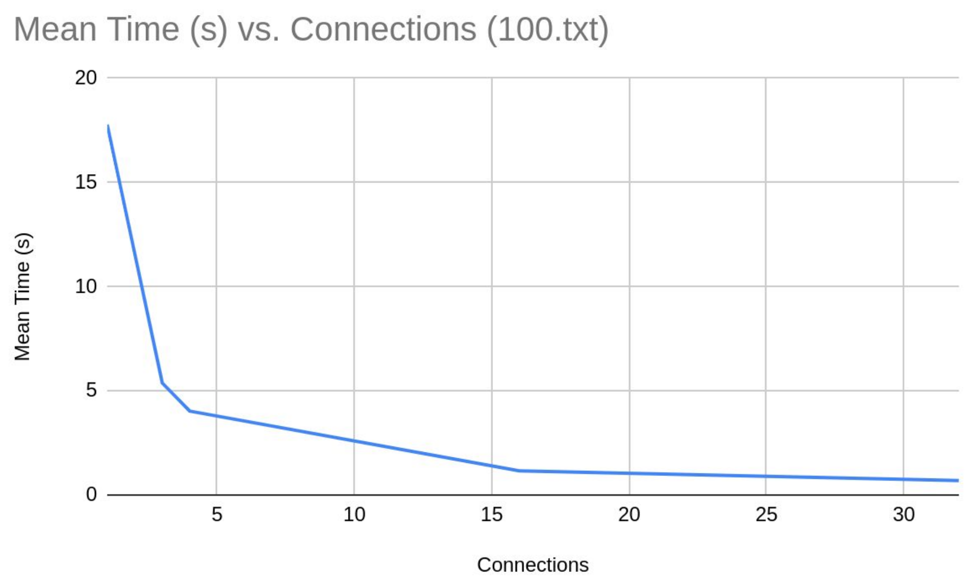
(16 Connections, Multi)

```
Time (mean ± σ):      1.154 s ±  0.054 s    [User: 63.5 ms, System: 19.6 ms]
  Range (min … max):    1.068 s …  1.232 s    10 runs
```

(32 Connections, Multi)

```
Time (mean ± σ):     688.5 ms ±  32.2 ms    [User: 57.1 ms, System: 23.5 ms]
  Range (min … max):   636.7 ms … 735.0 ms    10 runs
```

The line graph below plots mean runtime (10 runs) as the number of connections increases.



Runtime decreases as the number of concurrent connections increases. From 1 connection to 3, runtime drops from 17.774s to 5.378s - approximately 3.3x faster with 3x more connections. From 4 connections to 16, runtime goes down from 4.029 to 1.154 - around 3.49x faster with 4x as many connections. From 16 connections to 32, runtime goes down from 1.068s to 636.7ms - around 1.67x faster with 2x as many connections.

It is clear that multiplying the number of connections by some positive integer N does not cause a corresponding runtime improvement of N; at first, this is the case (1 connection to 3 leads to 3.3x faster), but the runtime improvement factor becomes less than N as the number of connections increases (16 connections to 32 only leads to 1.67x faster). Hence, while the plot looks like an inverse relationship, it is not quite! The improvement factor M becomes less than the connection increase factor N as the connection count increases.

The reason for the runtime improvement factor M becoming less than the connection increase factor N is that there is growing overhead as the event loop handles more concurrent connections. Each call to `multi.wait()` (and hence `poll()`) iterates through each active connection - similarly for `multi.perform()`, which must examine each `struct pollfd` modified by `poll()`.

2.3.1.1 Quick Digression: epoll

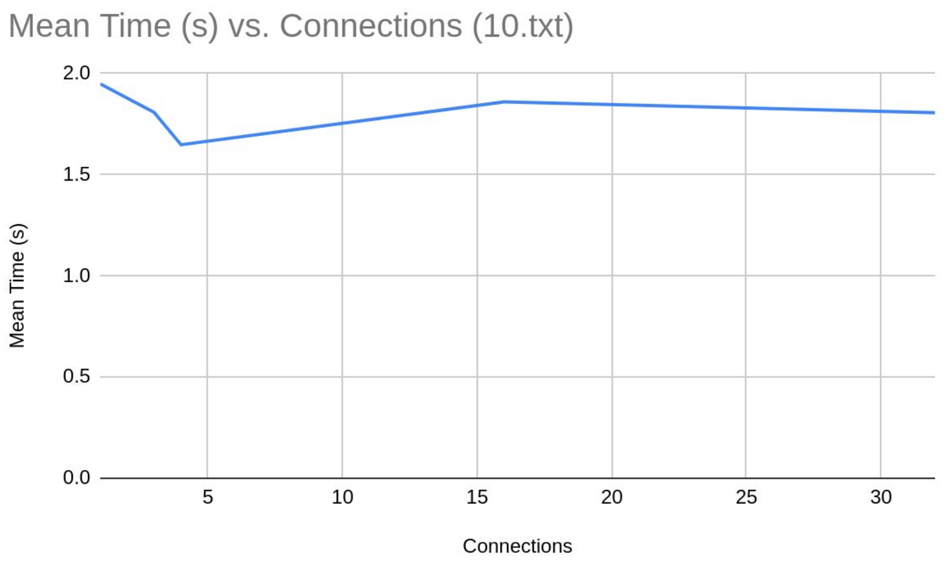
It is possible that the performance plot would look more like an inverse relationship if the system call for multiplexed I/O `epoll()` were used instead of `poll()`; the former does not iterate through each registered socket on each call like the latter.

Similarly, `multi.perform()` could be replaced with a call to `multi.action()` for each ready socket (the ready list returned from `epoll()`) to eliminate the need for iteration over all registered sockets when performing non-blocking I/O on the ready sockets.

2.3.2 10 Puzzles

A similar set of data was collected for 10.txt, the test files containing 10 puzzles (raw benchmark output not shown). Again, each scenario involved 10 runs.

Connections	Mean Time (s)
1	1.949
3	1.808
4	1.648
16	1.860
32	1.806



In this case, some improvement was observed when going from 1 connection to 4 - runtime dropped from 1.949s to 1.648s. However for 16 and 32 connections, not only did performance cease to improve, but it actually regressed - runtime increased back up to 1.860s and 1.806s for 16 connections and 32 respectively.

One explanation for the lack of improvement for 16 and 32 max connections is at that point, there are more connections than requests to send! The extra capacity is simply not used when we only send 10 requests - at most 10 concurrent connections are ever made.

As for the decrease in performance from 4 to 32 connections, that may be due to each request using a new TCP connection. If each connection is used only for a single request, the overhead of establishing all 10 connections may cause the total runtime to be greater than if 3-4 requests were sent down each of 3 connections.