# COP290 CLAB Report

Aditya Yadav
Degala Aman
Deepak Sagar

1 March 2025

**idhar vo edge cases ka add krdena**

## 1 Introduction

In this project we are making a spreadsheet which can perform basic arithmetic tasks, scrolling , sleep etc. Broadly there are two parts in this project, parsing the input and then calculating and recalculating the values of the cells. We made use of graphs and avl tree and other data structures to do the recalculations efficiently.

## 2 Flow

### 2.1 Parser

Here we identify the command,take the required info from the command like the cell number and then compute the value accordingly. We then move on to checking whether there is circular error while adding the new constraint and then updating the values and the dependencies of the cells.

### 2.2 Checking for Circular Error

We then find all the nodes that are affected by the change in value in the current node. We do this by first adding the cells that depend on the value of the current cell, then we recursively keep adding the cells that depend on this cells.We add all these cells to an AVL Tree, as all Insert and Find operations take place in O(logn) time in AVL Tree. If in the new constraint, the current cell is dependent on value of some cell which is part of the AVL tree we created, we have a circular error and we reject the input command.

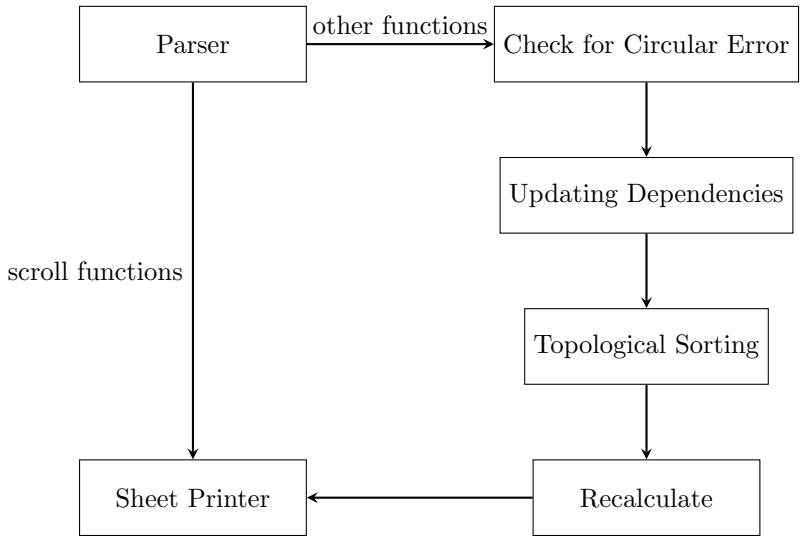


Figure 1: Flow

### 2.3 Updating the dependencies

We then remove the current from the linked lists of the cells the current cell used to depend upon(if any) and add the the current cell to the linked lists of the cells which it depends upon.

### 2.4 Topological Sorting

We store an `indegree` in the node of the AVL tree which indicates the number of cells a cell's value depends on. We then use **Khan's Algorithm** for topological sorting so that we can efficiently recalculate the value of all the cells which directly or indirectly depend on the cell to which we are adding the constraint.If we do recalculations according to the order given by this sorting, we'll be recalculating at each node only once and hence it will be efficient.

### 2.5 Printing the Spreadsheet

We then print all the cells if the output is not disabled while also printing `ERR` in place of cells which have some error.

## 3 Memory

#### 3.0.1 Cell Struct

For each cell, we have to store :-

- integer value stored in it
- constraint assosciated with that cell
- cells which depend on this cell , this is because we need to recalulate those other cells if the value in this cell changes

The obvious way to store these is a struct. The struct is as follows :

```
typedef struct cell{
    int value;
    bool isError;
    char op_code;
    cell_info cell1;
    cell_info cell2;
    dependency_node *dependencies;
} Cell;
```

The `op_code` is a character which indicates the constraint assosciated with that cell. These are the op_code characters and the related constraints :-

| op_code | Constraint Type |
|---|---|
| = | cell |
| + | cell + cell |
| - | cell - cell |
| * | cell * cell |
| / | cell / cell |
| S | SUM |
| m | MIN |
| M | MAX |
| A | AVG |
| D | STDEV |
| Z | sleep(cell) |
| X | const op const, const, sleep(const) |
| p | const + cell or cell + const |
| s | const - cell or cell - const |
| u | const * cell or cell * const |
| d | const / cell or cell / const |

The `cell_info` is a struct which stores the row and column of the cells which appears in the constraints.
`dependices` is basically a linked list storing the `cell_info` structs of the cells which depends on the value of the current cell. We store `(column of cell)*1000 + (row of cell)` as the value in each node in dependency list

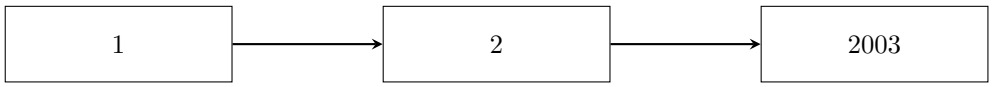


Figure 2: dependency linked list

**Memory used by cell struct**:

- value : 4 bytes
- isError : 1 byte
- op_code : 1 byte
- cell1 : 4 bytes
- cell2 : 4 bytes
- dependencies : 8 bytes

**Total** : 22 bytes

The `dependency_node* dependencies` points to linked list which might keep increasing.

#### 3.0.2 The Spreadsheet

The number of rows in the spreadsheet can range from 1 to 999 and the columns from A to ZZZ, Therefore the total number of cells in the spreadsheet can range from 1 to 1,82,59,722. Allocating such amount of memory in a single malloc statement might cause erorrs if space is not available, so instead we use malloc to allocate cells in each row and hence creating a 2D array.

#### 3.0.3 AVL Tree

Every time a new constraint is added to a cell, we make an AVL tree and immediately free it after using the topological sort.The value stored in each node(using which we compare values while inserting into the tree) is equal to `(column of cell)*1000 + (row of cell)`.

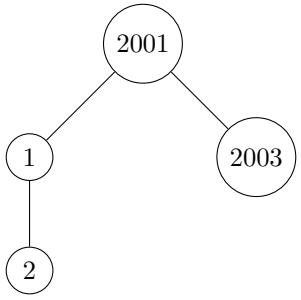


Figure 3: Example AVL Tree

# 4 Error Detection

## 4.1 Invalid input

In the parser, if the input is not of the required format, or if the cell row/column is out of bounds we get an invalid input error and status is changed to `Invalid cmd`

## 4.2 Circular Error

As already discussed, the circular error is detected using the AVL Tree, and the status is changed to `circular error`

## 4.3 Zero Division Error

In case we get a zero divison error, we change the isError field of cell struct to true and accordingly print ERR while printing the sheet.

## 4.4 Cells dependent on cells with Zero Division error

We detect this while computing the value, just like above we change the isError field to true.