COP290
C-LAB

Submitted by:

Savya Goel	Harshit Kansal	Varun Subramaniam
2023CS50115	2023CS10498	2023CS50497

Department of Computer Science and Engineering
IIT Delhi

---

1 Design Decisions

1.1 Data Structures and Algorithms

Step 1: Sheet Representation and Dependency Tracking
The spreadsheet is represented using a 2D dynamic array. For tracking dependencies among cells, an adjacency structure based on linked lists is used. Each cell maintains its dependencies in a separate linked list. The linked list implementation allows for insertion and deletion at the head with a time complexity of $O(n)$(due to checking for duplicates before insertion), though lookups and traversals require $O(n)$, where n is the number of dependencies for a cell. Additionally, we used bitfields to store the operation type along with the dependent cell coordinates in a memory-efficient way.

Step 2: Recalculation Strategy
To propagate updates efficiently, a topological sort-based recalculation is implemented. Functions like `toposort` and `recalculate` traverse the dependency graph ensuring that cells are updated in the correct order – i.e., every cell is recalculated only after all cells it depends on have been updated.

2 Challenges Faced

The main challenge was handling large range operations. For example, the command `A1=SUM(A2:ZZZ999)` involves almost 18000000 cells. The challenge arose due to the large number of memory allocations and linked list operations required, as well as the cycle detection in the dependency graph. Additionally, we had to convert recursive implementations to iterative ones to prevent stack memory overflow and program crashes, which required significant debugging effort.

3 Structure of the Program

3.1 Modules and Interactions

The program is modularized into the following key files:

- **main.c**: Controls the program flow; initializes the sheet; reads input from the user; times execution and prints output; and calls other modules for processing.
- **input_process.c**: Parses user input and distinguishes among control commands, value assignments, arithmetic expressions, and function assignments.
- **sheet.c**: Manages the spreadsheet state including coordinate conversions. It provides routines to print the sheet and executes commands.
- **functions.c**: Implements various range functions (e.g., `MIN`, `MAX`, `SUM`, `AVG`, `STDEV`) as well as sleep functionality.
- **recalculations.c**: Handles dependency management using linked lists, cycle detection (`has_cycle`), and the propagation of updates via topological sorting. All updates to dependent cells are scheduled here.

Execution Flow:
The execution flow begins with input read by `main.c`, moves to parsing in `input_process.c`, and then branches to processing commands in either `sheet.c` or `functions.c`. Any changes that affect other cells trigger dependency updates in `recalculations.c`, and finally, the updated sheet is printed as output.

4 Test Suite

The test suite in `unittest.c` validates the spreadsheet program across `input_process.c`, `sheet.c`, `functions.c`, and `recalculations.c`, using targeted test functions with normal, boundary, and error scenarios to ensure robustness. Below, we summarize the coverage and edge cases.

- **Input Parsing (`test_process_input`, 15 cases):**
  - *Normal Cases:* Assignments (e.g., "ZZZ999=10"), arithmetic (e.g., "A1=10+20"), functions (e.g., "A1=MIN(A1:A3)"), and commands (e.g., "enable_output").
  - *Edge Cases:* Invalid inputs (e.g., "A1=10+20+30"), incomplete expressions (e.g., "A1=10+"), maximum cell references (e.g., "ZZZ999=100").
- **Assignment Handling (`test_process_assign_input`, 5 cases):**

  - *Normal Cases:* Constants (e.g., "A1=10"), overwrites (e.g., "A1=5"), references (e.g., "B2=A1").
  - *Edge Cases:* Circular dependencies (e.g., "A1=A1", expected to fail), ensuring cycle detection prevents infinite loops.

- **Arithmetic Operations (`test_process_arith_expr`, 7 cases):**
  - *Normal Cases:* Basic operations (e.g., "A1=5+3"), references (e.g., "A1=B2-C3"), mixed inputs (e.g., "A1=2*C3").
  - *Edge Cases:* Division by zero (e.g., "A1=B2/0", expecting error), cycles (e.g., "A1=B2+1", "B2=A1+1", expected to fail).

- **Function Processing (`test_process_functions`, 10 + 4 large-range cases):**
  - *Normal Cases:* Operations like "SUM(A1:D10)", "AVG(A1:D5)", "STDEV(A1:D1)", and "SLEEP(2)" (sets value to 2 after delay).
  - *Edge Cases:* Invalid ranges (e.g., "SUM(A10:A1)", expected to fail), circular dependencies (e.g., "A1=SUM(A1:A1)"), single-cell ranges (e.g., "SUM(A1:A1)"), and negative SLEEP inputs (e.g., "SLEEP(-3)", assigning -3 without delay).
  - *Large-Range Cases:* Extensive ranges (e.g., "SUM(F1:ZZZ999)"), ensuring scalability.

- **Cycle Detection (`test_has_cycle`, 6 cases; `test_range_has_cycle`, 5 cases):**
  - *Normal Cases:* Direct self-references (e.g., A1=A1), indirect cycles (e.g., A1=B1, B1=A1), range cycles (e.g., "B2=MIN(A1:C3)", "C3=MIN(B2:A1)").
  - *Edge Cases:* Sheet-edge cycles (e.g., "E5=E5"), single-cell range references (e.g., "D4=MIN(A1:A1)").

- **Dependency Management (`test_add_delete_dependencies`, 5 cases):**
  - *Normal Cases:* Single (e.g., A1 impacts B1) and range dependencies (e.g., A1:B2 impacts C3).
  - *Edge Cases:* Self-referential dependencies (e.g., C3 impacts C3).

- **Recalculation (`test_recalculate`, 7 cases):**
  - *Normal Cases:* Propagation (e.g., B1=A1+5), references (e.g., C3=B2), arithmetic (e.g., C1=A1+B1), range sums (e.g., F6=SUM(F1:J5)).
  - *Edge Cases:* Division by zero (e.g., B1=A1/0), cycles (e.g., A1=B1, B1=A1), error propagation through ranges (e.g., C1=MIN(A1:B1) with B1 in error).

- **Topological Sort and Memory (`test_toposort`, 3 cases; `test_free_dependencies`, 1 case):**

– *Normal Cases:* Linear (A1→B1→C1), branching (A1→B1, A1→C1) dependencies, isolated nodes; memory cleanup after dependency additions.

The suite ensures robustness by addressing boundary cell references, integer overflows, large ranges, and error propagation, handling both typical and exceptional scenarios reliably.

5 Diagram of Software

A high-level flowchart of the program is outlined below:

Input Block	`main.c` reads user commands and sends them to `process_input`
Processing Block	`input_process.c` splits the command into control, assignment, arithmetic, or function types and calls the respective module
Dependency Block	`recalculations.c` updates the dependency graph (linked lists) and recalculates affected cells using topological sort
Output Block	`sheet.c` prints the updated sheet via `print_sheet`

This modular structure and clear separation of responsibilities enhance both the maintainability and scalability of the program.

6 Conclusion

In summary, the program implements a spreadsheet application with arithmetic and range operations. We implemented the sheet using dynamic arrays, dependency management using linked lists, and topological sort for recalculation to achieve efficiency and scalability even for very large inputs. While there were issues with performance and memory management during large operations, they were resolved using iterative implementations and the use of explicit stacks.

7 Links

- **GitHub Repository:** `https://github.com/username/spreadsheet-project`
- **Demo Video:** `https://youtu.be/sample-video-id`