School of Computing and Information Systems

comp10002 Foundations of Algorithms
Semester 2, 2025
Assignment 2

Learning Outcomes

In this project, you will demonstrate your understanding of dynamic memory (Chapter 10) and extend your program design, testing, and debugging skills. You will explore sparse matrix representations, implement core operations for manipulating sparse two-dimensional matrices, and apply these operations to perform sequences of matrix manipulations.

Background

In this assignment, you are asked to implement a program that transforms a given initial two-dimensional matrix into a target matrix by applying a sequence of matrix manipulations. This sequence can be viewed as an attempt to solve a puzzle, where the objective is to reach the target matrix with as few manipulations as possible.

For example, the initial matrix in Figure 1a can be transformed into the target matrix in Figure 1f through the following sequence of manipulations: first, set the value of the cell at row 1 and column 1 to 1 (Figure 1b); next, set the value of the cell at row 4 and column 4 to 0 (Figure 1c); then, multiply all matrix entries by two (Figure 1d); afterwards, copy the values from row 1 into row 7 (Figure 1e); and finally, add one to all nonzero entries (Figure 1f). In Figure 1, the numbers shown along the left and right margins indicate row indices, while the numbers displayed above the matrices denote column indices (counting from zero).

```
     012345   012345   012345   012345   012345   012345

0    000000   000000   000000   000000   000000   000000   0
1    002200   012200   012200   024400   024400   035500   1
2    010030   010030   010030   020060   020060   030070   2
3    010030   010030   010030   020060   020060   030070   3
4    012230   012230   012200   024400   024400   035500   4
5    010030   010030   010030   020060   020060   030070   5
6    010030   010030   010030   020060   020060   030070   6
7    010030   010030   010030   020060   024400   035500   7
8    000000   000000   000000   000000   000000   000000   8
       (a)      (b)      (c)      (d)      (e)      (f)
```

Figure 1: (a) Initial matrix, (f) target matrix, and (b)–(e) intermediate matrices constructed when transforming the initial matrix into the target matrix; nonzero values are highlighted with a yellow or green background, and values updated by the last manipulation are highlighted with a green background.

In general, the matrices can be large, making it infeasible to store them as two-dimensional arrays in memory. Fortunately, such large matrices are typically sparse, meaning that most of their entries are zero. Your implementation should therefore make use of this observation and rely on sparse matrix representations. One option is to store only the row and column indices of the nonzero values, along with the values themselves, in arrays that get dynamically reallocated to accommodate all nonzero entries. Alternatively, one may use an array of linked lists, where each linked list stores nonzero values and their corresponding column indices from a single row of the matrix. You can choose which representation to implement. We do not require you to design the most efficient structure for all supported matrix manipulations. However, your program should achieve a reasonable runtime, typically under one second for each test input.

Input Data

Your program should read input from `stdin` and write output to `stdout`. The input will list instructions, one per input line, each terminating with "`\n`", that is, one newline character. The input specifies how to configure the initial and target matrices and lists the manipulations to apply to the initial matrix. The following file `test0.txt` provides an example input containing 42 lines.

```
1   9x6
2   1,2,2
3   1,3,2
4   2,1,1
5   2,4,3
6   3,1,1
7   3,4,3
8   4,1,1
9   4,4,3
10  4,2,2
11  4,3,2
12  5,1,1
13  5,4,3
14  6,1,1
15  6,4,3
16  7,1,1
17  7,4,3
18  #
19  1,1,3
20  1,2,5
21  1,3,5
22  2,1,3
23  2,4,7
24  3,1,3
25  3,4,7
26  4,1,3
27  4,2,5
28  4,3,5
29  5,1,3
30  5,4,7
31  6,1,3
32  6,4,7
33  7,1,3
34  7,2,5
35  7,3,5
36  #
37  s:1,1,1
38  s:4,4,0
39  m:2
40  r:1,7
41  a:1
42
```

Line 1 specifies the dimensions of the initial and target matrices in the "rxc" format, where r is the number of rows and c is the number of columns. Each subsequent line, up to the first line containing only the "#" character (line 18 in `test0.txt`), sets values in the initial matrix. Each such line follows the format "r,c,v", which sets the entry in row r and column c to the integer value v. All other entries in the initial matrix are set to zeros. Lines following the first "#" and up to the second occurrence of "#" (lines 19 to 35) set the values of the target matrix. These lines use the same "r,c,v" format as above. All unspecified entries in the target matrix are set to zeros. Lines 1 to 36 in `test0.txt` define the initial and target matrices shown in Figure 1a and Figure 1f, respectively.

Each non-empty line following the second occurrence of # (lines 37 to 41 in `test0.txt`) specifies a manipulation to apply to the initial matrix. These manipulations must be executed in the order in which they appear in the input. Each manipulation is given in the format "t:p", where t denotes the type of manipulation and p specifies its parameters. The manipulation type is captured by a single character, and its parameters are given as a comma-separated list, with the number of parameters depending on the manipulation type.

The input will always follow the proposed format. You can make your program robust by handling inputs that deviate from this format. Extensions to the program that handle wrong inputs will not be tested.

Output

The output your program should generate for the `test0.txt` input file is provided in the `test0-out.txt` file. This file contains 133 lines in total. A subset of these lines is shown below.

```
1   ==STAGE 0============================
2   Initial matrix: 9x6, nnz=16
3   [        ]
4   [   22   ]
5   [  1  3  ]
6   [  1  3  ]
7   [  1223  ]
8   [  1  3  ]
9   [  1  3  ]
10  [  1  3  ]
11  [        ]
12  -------------------------------------
13  Target matrix: 9x6, nnz=17
14  [        ]
15  [  355   ]
16  [  3  7  ]
17  [  3  7  ]
18  [  355   ]
19  [  3  7  ]
20  [  3  7  ]
21  [  355   ]
22  [        ]
23  ==STAGE 1============================
24  INSTRUCTION s:1,1,1
25  Current matrix: 9x6, nnz=17
26  [        ]
27  [  122   ]
28  [  1  3  ]
29  [  1  3  ]
30  [  1223  ]
31  [  1  3  ]
32  [  1  3  ]
33  [  1  3  ]
34  [        ]
35  Target matrix: 9x6, nnz=17
36  [        ]
37  [  355   ]
38  [  3  7  ]
39  [  3  7  ]
40  [  355   ]
41  [  3  7  ]
42  [  3  7  ]
43  [  355   ]
44  [        ]
        ... lines 45–86 skipped ...

87  ==STAGE 2============================
88  INSTRUCTION r:1,7

        ... lines 89–129 skipped ...

130 -------------------------------------
131 TA-DAA!!! SOLVED IN 5 STEP(S)!
132 ==THE END============================
133
```

Stage 0 – Reading, Analyzing, and Printing Initial and Target Matrices (10/20 marks)

The first version of your program should read the initial and target matrices from the input and print basic information about them to the output. The first 22 lines of the `test0-out.txt` file show the output your program is expected to produce in Stage 0 based on the first 36 lines of input in `test0.txt`. Line 1 of the output prints the Stage 0 header. Lines 2–11 print the initial matrix along with basic information about its dimensions (see `9x6` in line 2, indicating nine rows and six columns) and the number of nonzero entries (see `nnz=16`). Line 12 prints a delimiter, which is followed by the output describing the target matrix (lines 13–22).

The number of rows and columns in matrices can range from one up to the maximum of 5,000,000 (five million). To handle large matrices, your program should rely on dynamic memory management and appropriate data structures, such as dynamic arrays or linked lists, and use a sparse matrix representation for storage.

A matrix with up to 35 rows and columns and all entries between zero and nine inclusive should be printed as a sequence of rows, one per output line. Each row is enclosed in square brackets "`[ ]`". Nonzero entries are printed as digits (1–9) in their corresponding column positions. Zero entries are represented by blank spaces. Hence, the width of each row matches the number of matrix columns. Row and column indices start from zero, with row 0 and column 0 located at the top-left corner of the matrix (row-major zero-based indexing).

If a matrix has more than 35 rows or more than 35 columns or contains at least one entry outside the 0–9 range, it should be printed as a list of all its nonzero entries, one per line. Each nonzero entry must be printed in the "`(`r`,`c`)=`v" format, where r, c, and v denote the row index, column index, and entry value, respectively. Entries should be printed in the row-major order, that is, matrix entries should be processed row by row, from top to bottom, and within each row, from left to right. Refer to the supplied output file for the `test2.txt` input file for an example of this matrix printing format.

Stage 1 – Performing Basic Matrix Manipulations (16/20 marks)

In Stage 1, your program should read from the input and apply to the initial matrix the four basic manipulations listed in the top part of Table 1. The manipulation of type "`s`" sets the value at row r and column c to integer v (which can be negative). The manipulation of type "`S`" swaps the value at row r_1 and column c_1 with the value at row r_2 and column c_2. Finally, the manipulations of types "`m`" and "`a`" multiply every value in the matrix by v and add v to every nonzero value in the matrix.

The output of Stage 1 should start with the corresponding header; see line 23 in the `test0-out.txt` file. Lines 24–86 then report the manipulations applied to the initial matrix. For each manipulation, the output first shows a summary of the instruction (line 24), followed by the state of the transformed matrix after the manipulation and the target matrix (lines 25–44). If, after applying a manipulation, the resulting matrix matches the target matrix, your program should print a delimiter line (see line 130 in `test0-out.txt`), print the "`TA-DAA!!!`" message reporting the total number of manipulations used to transform the initial matrix into the target matrix (line 131), print the end message (line 132), and terminate execution.

You should not assume a fixed upper limit on the number of manipulations requested to be applied in Stage 1. Processing of manipulations should continue until either an unrecognized manipulation type is encountered or parameters of the currently processed manipulation fall outside the bounds of the matrix.

Stage 2 – Extending Supported Matrix Manipulations (20/20 marks)

In Stage 2, your program should read from the input and apply to the initial matrix four further manipulations listed in Table 1. The manipulation of type "`r`" (type "`c`") copies values from row r_1 (column c_1) to row r_2 (column c_2). The manipulation of type "`R`" (type "`C`") swaps values in row r_1 (column c_1) with the values in row r_2 (column c_2). Stage 2 of your program should begin as soon as the first manipulation of this stage is encountered in the input. The output of Stage 2 should start with the corresponding header; see line 87 in the `test0-out.txt` file, followed by information about the performed manipulations (lines 88–129), presented in

Table 1: Matrix manipulation instructions; cell (r, c) refers to the entry in the matrix at row r and column c.

Stage	Type	Parameters	Description
1	s	r,c,v	Set the value of cell (r, c) to v.
1	S	r_1,c_1,r_2,c_2	Swap values in cells (r_1, c_1) and (r_2, c_2).
1	m	v	Multiply all values in the matrix by v.
1	a	v	Add v to all *nonzero* values in the matrix.
2	r	r_1,r_2	Copy all values from row r_1 to row r_2.
2	c	c_1,c_2	Copy all values from column c_1 to column c_2.
2	R	r_1,r_2	Swap the values in rows r_1 and r_2.
2	C	c_1,c_2	Swap the values in columns c_1 and c_2.

the same format as in Stage 1. Again, the processing of manipulations should terminate once an unrecognized manipulation type is encountered or parameters of the currently processed manipulation fall outside the bounds of the matrix. Make no assumptions about the number of instructions that can be supplied in Stage 2. If, after a manipulation, the transformed matrix is identical to the target matrix, print the delimiter line (like line 130), the "`TA-DAA!!!`" message (line 131), and terminate the program after printing the end message (line 132).

The input in the `test0.txt` file specifies five manipulations (three to be performed in Stage 1 and two in Stage 2). Figures 1a through 1f illustrate the intermediate matrices obtained by applying these manipulations while transforming the initial matrix into the target one.

Stage 3 – Having Fun (no marks for fun as fun is your reward)

Extend your program into a game that loads an initial and a target matrix from an input file, applies matrix manipulations entered by the player via standard input (`stdin`), and records the shortest transformation sequence found so far into an output file. Furthermore, implement strategies for searching optimal transformation sequences and add support for new matrix manipulations of your choice.

Pay Attention!

In total, three test inputs and outputs are provided. The outputs generated by your program should be *exactly the same* as the sample outputs for the corresponding inputs. Use `malloc` and dynamic data structures of your choice to store matrices. Before your program terminates, all the `malloc`'ed memory must be `free`'d.

Important...

This project is worth 20% of your final mark, and is due at **6:00pm on Friday 17 October**.

Submissions that are made after the deadline will incur penalty marks at the rate of two marks per (part or all) calendar day. For example, submissions made after 6:00pm Friday and before 6:00pm Saturday will lose two marks. Multiple submissions may be made; only the last submission that you make before the deadline will be marked. If you make any late submission at all, your on-time submissions will be ignored, and if you have not been granted an extension, the late penalty will be applied.

A rubric explaining the marking expectations is linked from the LMS, and you should study it carefully. Marks and feedback will be provided approximately two weeks after submissions close.

You need to submit your program for assessment **via the link to Gradescope at the bottom of the LMS Assignment page**. Submission is not possible through Ed Lessons.

Academic Honesty: You may discuss your work during your workshop, and with others in the class, but what gets typed into your program must be individual work, not copied from anyone else. So, do **not** give hard copy or soft copy of your work to anyone else; do **not** have any "accidents" that allow others to access your work; and do **not** ask others to give you their programs "just so that I can take a look and get some ideas, I won't copy, honest". The best way to help your friends in this regard is to say a very firm "**no**" if they ask to see your program, pointing out that your "**no**", and their acceptance of that decision, are the only way to preserve your friendship. See `https://academicintegrity.unimelb.edu.au` for more information. Note also that solicitation of solutions via posts to online forums, whether or not there is payment involved, is also Academic Misconduct. In the past students have had their enrolment terminated for such behavior.

The assignment page contains a link to a program skeleton that includes an Authorship Declaration that you must "sign" and include at the top of your submitted program. Marks will be deducted (see the rubric) if you do not include the declaration, or do not sign it, or do not comply with its expectations. A sophisticated program that undertakes deep structural analysis of C code identifying regions of similarity will be run over all submissions. Students whose programs are identified as containing significant overlaps will have substantial mark penalties applied, or be referred to the Student Center for possible disciplinary action.

Nor should you post your code to any public location (`github`, `codeshare.io`, etc) while the assignment is active or prior to the release of the assignment marks.

Special Consideration: Students seeking extensions for medical or other "outside my control" reasons must lodge a request following the FEIT process linked from the LMS page, and do so as soon as possible after those circumstances arise. If you attend a GP or other health care service as a result of illness, be sure to obtain a letter from them that describes your illness and their recommendation for treatment.