Computation Structures
Project 1
Perfect maze in β-assembly

October 03, 2017

General information

- **Deadline: October 29, 2017, 23:59**.
- Project must be done by **teams of two students**.
- There will be a penalty for late submission.
- Questions will no longer be answered 24 hours before the deadline.
- English is strongly encouraged.
- Contact: `r.mormont@ulg.ac.be`, Office I.127 (B28).

1 Introduction

The goal of this project is to make you more familiar with the β-assembly language by getting your hands dirty and writing code. More precisely, you will implement an algorithm to draw a perfect maze in the machine's dynamic memory.

1.1 Perfect maze

Let a maze M with R rows and C columns be a set of cells c_i with $i \in \{0, ..., (R \times C) - 1\}$. Thus, if the rows are numbered 0 to $R-1$ and the columns 0 to $C-1$, the cell in row k and column l is $c_{(k \times C + l)}$. A cell c_i has a connection $e(i,j)$ with all its direct existing neighbours:

- Top neighbour: $e(i, i-C)$ if $i \geq C$
- Bottom neighbour: $e(i, i+C)$ if $i < C(R-1)$
- Left neighbour: $e(i, i-1)$ if $(i \mod C) \neq 0$
- Right neighbour: $e(i, i+1)$ if $(i \mod C) \neq C-1$

A connection $e(i,j)$ can be either opened (i.e. $e(i,j) = 1$) or closed (i.e. $e(i,j) = 0$). For instance, in Figure 1, the connection $e(0,1)$ is opened while $e(14,5)$ is closed. Note that $e(i,j) = e(j,i)$. A path $p(k,l)$ is a sequence of **opened** connections $e(k,j_1)$, $e(j_1,j_2)$,..., $e(j_{n-1},j_n)$, $e(j_n,l)$ linking cells c_k and c_l. A **perfect maze** is such that, for all pairs of cells c_k and c_l with $l \neq k$, there is one and only one path $p(k,l)$ linking them. In other words, there should be no cycle in the maze and every cell can be reached from any other cell.

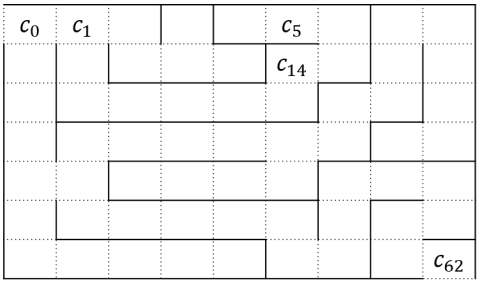

Figure 1: A perfect maze

1.2 Algorithm

The following algorithm PerfectMaze creates a random perfect maze with R rows and C columns.

PerfectMaze(R, C)
1 $M =$ "*a maze with R rows and C columns with all connections closed*"
2 $nbCells = R \times C$
3 // pick random cell c_s to start building the perfect maze
4 s = Rand() % $nbCells - 1$
5 // V[i] contains 1 if there is a path between cell c_i and c_s, 0 otherwise
6 // in other words, it contains 1 if the cell c_i was already connected to the maze being built
7 $V =$ "*array of zeros of size nbCells*"
8 **return** PerfectMazeAux(M, V, R, C, s)

The algorithm PerfectMazeAux builds a random perfect maze M starting at cell c_c. Taking them in a random order, it will connect each unvisited neighbour cell c_n of c_c to c_c and build a random perfect maze from c_n recursively.

PerfectMazeAux(M, R, C, V, c)
1 V[c] = 1
2 $N =$ "*create array with c_c neighbours indexes*"
3 $N =$ RandomShuffle(N)
4 **for** $n \in N$
5 **if** V[n] == 0
6 Connect(M, c, n)
7 $M =$ PerfectMazeAux(M, V, R, C, n)
8 **return** M

The procedure Connect(M, i, j) opens the connection between the cells c_i and c_j of M (i.e. it makes $e(i, j) = 1$).

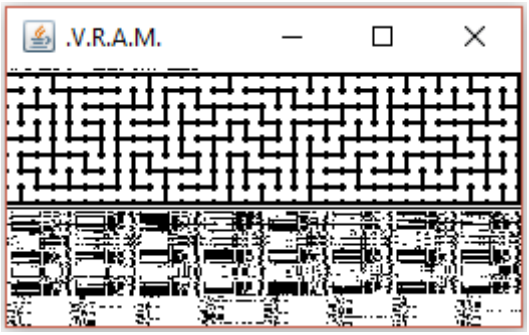

Figure 2: The memory view showing the first 1024 words of the dynamic memory.

2 Memory view of the β-simulator

The β-simulator (βsim) [1] presented during the tutorials features a memory view window allowing you to visualize the 1024 first words of the dynamic memory (see Figure 2). In this view, each pixel corresponds to 1 bit: a black pixel means this bit is set to 1 while a white pixel means it is set to 0. Each line of pixels corresponds to eight consecutive words of the memory. This window is where you will draw the maze. **Be careful**: while the words are drawn in the window following the order of addresses (i.e. word addresses increase when moving from left to right or down), the bits within a 32 bit word are displayed with the most significant bit on the right.

3 Project

You task is to implement a `perfect_maze` procedure in a file called `perfect_maze.asm` ready to be included in the file `main.asm` that is provided. This procedure should implement the recursive procedure `PerfectMazeAux` given in Section 1.2 and modify the memory to draw the resulting random perfect maze[2]. This procedure should be callable using a label `perfect_maze` and should have five parameters:

1. `maze`: a pointer to the first word containing the memory image of the maze
2. `nb_rows` : the number of rows in the maze
3. `nb_cols` : the number of columns in the maze
4. `visited` : a pointer to a bitmap containing the information whether each cell was visited or not. The bit b_{ij} (i.e. i^{th} bit of the j^{th} word of the bitmap) is set to 1 if the cell $c_{32\times j+i}$ has already been attached to the maze, 0 otherwise.
5. `curr_cell` : the cell from which the maze should be constructed

[1] The βsim program can be found on the page `http://www.montefiore.ulg.ac.be/~rmormont/?rpath=/info0012`.

[2] The memory should be updated such that the maze can be seen in the memory view of the β-simulator, see Figure 2.

You are provided with a file `perfect_maze.c`, a C implementation of the `perfect_maze` procedure, that you can use as basis for your implementation. While you are **not** bound to translate all implementation details contained in this file in your assembly code, you are still expected to implement the recursive algorithm presented in Section 1.2 and implemented in `perfect_maze.c`.

In addition to the file `perfect_maze.c`, you are provided with two other assembly files:

- `beta.uasm`: the definition of the β-assembly. You can check this file to see which macros you can use in your own code. Especially, the macro `RANDOM()` might be useful. `;)`
- `main.asm`: this file contains the main program that will be used to test your procedure. It does the following:
  - write a fully closed maze in the memory
  - allocate the `visited` array
  - initialize the stack and relevant pointers (`SP` and `BP`)
  - pick a random cell to start building the maze
  - call the `perfect_maze` procedure with relevant parameters

4 Additional guidelines

4.1 Practical organization

In order to learn β-assembly effectively, **this project will be done by teams of two students**. A report of **maximum two pages** may be provided if you want to explain things that are not easy to understand by just looking at the code and comments. Providing a report does not necessarily mean that you will earn a better grade; it should be provided only if it brings something that is not mentioned clearly elsewhere (e.g. in the comments).

Plagiarism is of course not allowed and severely punished. Any detected attempt will result in the grade 0/20 for all who have participated in this practice.

You will include your completed `perfect_maze.asm` and your (optional) report (PDF only) in a ZIP archive named `sXXXXXX_NAME1_sYYYYYY_NAME2.zip` where (`sXXXXXX`, `NAME1`) and (`sYYYYYY`, `NAME2`) are the student `ID` and family name in uppercase of the two team members. Insert your `perfect_maze.asm` in a ZIP archive even if you do not provide a report. Naming your files differently or submitting other files **will result in a penalty**.

Submit your archive to the **Montefiore Submission Platform**[3], after having created an account if necessary. If you encounter any problem with the platform, let me know. However problems that unexpectedly and mysteriously appear five minutes before the deadline will not be considered. **Do not send your work by e-mail; it will not be read**.

[3] `http://submit.montefiore.ulg.ac.be/`

4.2 Code guidelines

Choose a coding style and stick to it. You are advised to use the coding style used in `main.asm`. The goal here is not to write code which is as compact and efficient as possible, but to learn the concepts of β-assembly. However, your code should not be unreasonably long and inefficient: minimize the number of registers you use in your recursive procedures, as it impacts the growth rate of your stack.

4.3 Documentation

One of the challenges when writing assembly code is to write a program which is relatively easy to understand. Thus, the second most important element taken into account for your grade (after correctness) will be your code's readability. Use comments extensively (your comments can be larger than your code), but don't be verbose : explain the non obvious, not the immediately apparent.

In addition to the comments written alongside your code, all your procedures should be documented using pre- and post-conditions:

- Arguments have to be properly defined
- Any return value must be documented
- Any side effect (e.g. modification of the dynamic memory) must be documented

Your are free (and advised) to use macros to reduce the redundancy in your code. Those macros should be documented like your procedures (arguments and operation performed).

Good luck and have fun !