Computer Science

Summative Assignment

Module code and title	COMP2221 Programming Paradigms
Academic year	2022/23
Submodule title	Systems Programming
Coursework title	Coursework 2
Coursework credits	10 credits
Lecturer	Amir, Anne
Deadline*	Thursday, January 26, 2023 14:00
Hand in method	Gradescope

Additional coursework files	*langton.h visualiser.c visualiser.h*
Required submission items and formats	*Please list the files that should be submitted, including format*

*** This is the deadline for all submissions except where an approved extension is in place. Late submissions received within 5 working days of the deadline will be capped at 40%. Late submissions received later than 5 days after the deadline will received a mark of 0.**

Note: Please ask any questions you have about the coursework in the message board under Assessment in LearnUltra. This will be used as a sort of FAQ, i.e., please check if your question has already been answered before posting.

Langton's Ant

Langton's ant is a two-dimensional universal Turing machine that can demonstrate complex emergent behaviour based on a very simple set of rules. First created by Chris Langton in 1986, Langton's ant runs on a square lattice of black and white cells. In this coursework, you will implement a generalisation of Langton's ant in C.

Background

The original Langton's ant progresses by colouring squares on a plane either black or white. One square is first determined to be the "ant" arbitrarily. This ant can then travel in any of the four cardinal directions at each iteration (step) of the game. According to the generic definition of the problem, the movement of the ant is controlled by the following simple rules:

- When at a white square, turn 90° clockwise, flip the colour of the square to black and then move forward one unit.
- When at a black square, turn 90° counter-clockwise, flip the colour of the square to white and then move forward one unit.

Following these rules, the ant will move in simple symmetric patterns for the first few hundred iterations of the game, after which chaotic irregular patterns of black and white squares appear. Towards the end, the ant will start steadily moving away from the starting location in a diagonal corridor about 10 cells wide. The ant can then be conceptually moving away infinitely.

An extension to Langton's Ant involves more general *n*-state ants. This is also known as the multi-colour version of Langton's Ant. The squares can be in states 1 through *n* and for each state (sometimes demonstrated with different colours), "L" or "R" is used to indicate whether a left or right turn is taken by the ant. For instance, in LRRRRRLLR, the ant will turn left on visiting squares in states 1, 7 and 8 and will turn right on squares in states 2, 3, 4, 5, 6 and 9.

When the ant arrives at an "L" square, it turns left and when it arrives at an "R" square, it turns right. After the ant has left a square at state *i*, the square's state changes to *i*+1.

Further details of this can be found in the corresponding paper (https://arxiv.org/pdf/math/9501233.pdf). You do not need this paper to solve the coursework, but you may find it interesting.

Assignment

As part of this assignment, you will have to construct a dynamically-linked library **libant.so** which implements Langton's ant, and a program ant that calls the library for all its key functionality.

- Your code should run on Linux. A Docker image of Ubuntu 22.04 with *apt-get install -y make gcc libncurses-dev* will give you an identical environment to the one used for marking. Alternatively, you can use the University Linux system Mira. Failure to do so may result in all marks being lost regardless of whether it compiles and runs on your own windows machine.
- You may **not** use any external libraries aside from ncursesw.
- Some aspects of the marking are automated. Your code should use the functions and file names specified. Do not add additional files or folders. Do not output additional text. This may break the automated system, which will result in you losing marks.

Basic Functionality (50 marks)

The repository already contains functionality for visualising the result using the ncursesw library. In order to use it, you need to link it at compile time using -lncursesw.

All functions relating to visualisation are contained in *visualiser.c* and *visualiser.h*. For the generalisation, you will need to add additional "colours" to the visualiser.

- Pressing any key (except q) will advance the ant and pressing "q" will close the window.
- The predefined variables max_x and max_y define the size of your ant's universe. Ensure that the ant behaves appropriately regardless of the visualisation's window size. If your ant's behaviour changes depending on the size of the visualisation, you will lose marks.

You have been provided with a file called *langton.h* containing function signatures. Create a file *langton.c* that implements those functions. Make sure to use the data structures provided in *langton.h*. Marks will be lost if you do not use **enums** and **structs**.

- Implement the functions turn_left and turn_right. They should take an ant as input and modify its direction so that it turns left or right (8 marks).
- Implement move_forward, this function should move the ant forward by one step in the direction that it is facing (5 marks).
- The function apply_rule should change the colour of the ant's square. For the basic variant this means at a white square turn left and change the colour of the square to black and at a black square turn right and change the colour of the square to white (7 marks).

Since you cannot use infinite memory, you will implement the ant on a finite rectangular grid (the "universe"). Give the grid the topology of a torus: the grid wraps around, so if you go one row higher than the top row, you reach the bottom row, and if you go to the column one row to the leftmost column, you reach the rightmost column, etc. (10 marks).

Create a file called *main.c*. This file should contain only your main loop.

- Next, run the visualisation by creating an ant, starting the visualisation and looping until the user presses "q" to quit. Catch any errors or exceptions. Initially, the ant's universe is all state 0 (15 marks).
- In *langton.h*, write the macro ant_is_at(posy,posx) which checks whether the ant is at a given set of coordinates posx,posy (5 marks). Test it with the visualiser.

Note that cell_at(x,y) is a macro that must be written before the main loop can run. You can also implement cell_at_fct() and might find this useful for the implementation on the torus. It is not required.

Hint: Before you get started, familiarise yourself with the existing functions and data structures provided. Do not try to get the full functionality running in one go.

Advanced Functionality (25 marks)

- Read in command line arguments specifying the type of ant. If none are given run the basic ant variant. For the advanced variant, read in a rule in the format LLR. Handle errors appropriately if the user inputs an unexpected input (5 marks).
- Add the command line input to the struct rule to store the advanced rule (5 marks). Allocate memory appropriately for the length of the rule.
- Write the apply_rule_general with rule struct as input to apply the advanced rule (10 marks). Do not write each rule individually, your function should in principle work for rules of any arbitrary length.

- Modify any components of *visualiser.c* to visualise your new cell types. You may use letters, numbers, colours or a combination in the visualisation, but the cells must be of type **int** (5 marks). Ensure rules of length at least 26 can be visualised. Initially, the ant's universe should be the first state of the system.

Makefile and dynamically linked library (15 marks)

You will need to write a Makefile for the project in Makefile. It should have the following options:

- all: create an executable called **ant** that takes the variant of the game (e.g. LLRR) as the input and runs the game (2 marks)
- library: compile the dynamically linked library libant.so (2 marks)
- clean: remove all executables and library files (1 mark)

You may find this Makefile Tutorial helpful.

Code and documentation (10 marks)

- Clear, readable, well-documented (with sufficient comments) and well-presented program source code with good performance.
- We recommend use of a linter to check for correct code formatting. This might be e.g., splint, cpplint or kwstyle.

Plagiarism

You must not plagiarise your work. Automated software tools may be used to initially detect cases of potential source code plagiarism in this assignment which will include automatic code comparison with any available code online. Attempts to hide plagiarism by simply changing comments/variable names will be detected. You should make yourself aware of the Durham University policy on plagiarism.