ASSIGNMENT TWO: GENETIC ALGORITHM

ADVANCED PROGRAMMING TECHNIQUES – SEMESTER 2, 2018

SUMMARY

In this assignment, you will use your C programming skills to build a simple **Genetic Algorithm**.

A genetic algorithm (GA) is a heuristic search technique inspired by natural selection. A population of candidate solutions to a problem is iteratively evolved towards an optimal solution (as measured by an evaluation function) by:

- Evaluating the 'fitness' of candidate solutions according to the evaluation function
- Probabilistically selecting 'fitter' candidate solutions from the population to act as 'parents' for the next generation of candidate solutions
- Introducing constrained novelty to the new generation via operations analogous to reproduction and mutation. (Note: the selection function will then tend to remove 'bad' novelty).

The course Canvas assignment 2 page has links to descriptions of the above operations.

Genetic algorithms are used in computing and industry in areas as diverse as hardware design optimization, scheduling and machine learning. The genetic algorithm of this assignment is applied to two problems:

minfn: determine variables to satisfy an equation

pcbmill: minimise the drill head movement (and thus the time taken to drill) of a mill for drilling holes in a Printed Circuit Board (PCB).

The course Canvas assignment 2 page has links to specifications for *minfn* and *pcbmill* and sample data files that they can use.

In the following sections of this document, the details of your assignment are explained. Read them thoroughly and ask your questions from the teaching team. You are expected to understand every requirement explained in this document and implement all of them accordingly.

A start-up code is provided along with this document. Just producing code to implement the requirements is not enough. *You **must** use the start-up code in your assignment, developing and using the given functions in a reasonable way to implement the requirements. You are **not** permitted to change the given function prototypes.* The prototypes have been written in such a way to make you demonstrate your ability to use certain concepts taught in this course. You are required to submit the exact files provided in the start-up code after completing the specified tasks.

SUBMISSION

Total mark: 30% of the final mark

Assignment demo: During tutorial sessions of **week 10** (24th - 27th September 2018).

Final submission: End of week 12 – Friday, 12 October – **10:00 PM**

PLAGIARISM NOTICE

Plagiarism is a very serious offence. The minimum first time penalty for plagiarised assignments is zero marks for that assignment. Please keep in mind that RMIT University uses plagiarism detection software to detect plagiarism and that all assignments will be tested using this software. More information about plagiarism can be found here: https://www.rmit.edu.au/students/student-essentials/rights-and-responsibilities/academic-integrity.

DESCRIPTION OF THE PROGRAM

CORE REQUIREMENTS

REQ1: Command-line arguments

The user of your program must be able to execute it by passing in the following arguments:

```
./ga geneType alleleSize popSize numGen inputFile [outputFile]
```

where:

- `./ga` is the name of the executable
- `geneType` is a string, of either *minfn* or *pcbmill* – indicating which problem the GA is being applied to
- `alleleSize` is a positive non-zero integer specifying the length of a candidate solution (chromosome)
- `popSize` is a positive non-zero integer specifying the number of candidate solutions in the population
- `numGen` is a positive integer specifying the number of generations to run the GA for
- `inputFile` is a string specifying the name of a data file to read
- `outputFile` is an optional string specifying the name of an output file to write

The main program must verify the number of command-line arguments is as expected and ensure they can be converted to valid types and values.

REQ2: Loading the vector input data file

Your program needs to load into memory the data from the input data file specified in the command-line argument. You will need to tokenise this data before you can load it into the system. In this requirement, you can assume the specified input file is completely valid. Dealing with invalid input files is covered in a separate section of the assignment.

You will need to use the `InVTable` structure defined in the start-up code to store your input data. The input data is organised as a series of input *vectors*, one per line. The syntax of a line is:

InputVector:*vector-number***(***vector-element1***,***vector-element2….vector-elementN***)**

Where

- *vector-number* is a positive integer, incrementing sequentially from 0
- The vector is a sequence of at least one (comma-separated) integer *vector-element*
- Each *InputVector* in the file must have the same number of *vector-elements*

For example:

```
InputVector:0(1,3,4,2,40)
InputVector:1(2,7,1,2,9)
```

REQ3: Implementing and calling function *test_minfn*

Function `test_minfn()` tests the operation of functions `gene_create_rand_gene`, `mutate_minfn`, `crossover_minfn`, `gene_print` and `gene_free`. function `test_minfn()` is only compiled and called if macro `DEBUG` is defined. The start-up code includes a description of the function calls necessary to implement `test_minfn()`.

Functions `mutate_minfn`, `crossover_minfn` and `gene_print` also use conditional compilation, including debug statements if macro `DEBUG` is defined. In particular, if `DEBUG` is defined:

- `mutate_minfn` will use a mutation point of chromosome index 2
- `crossover_minfn` will use a crossover point of chromosome index 2

An example of output from a call of `test_minfn()` is:

```
MINFN gene:
chrom:13,16,27,25,23,25 fit: 0.000 raw:  0.000
Mutate: index 2
chrom:13,16,12,25,23,25 fit: 0.000 raw:  0.000
MINFN genes:
chrom: 9, 1, 2, 7,20,19 fit: 0.000 raw:  0.000
chrom:23,16, 0, 6,22,16 fit: 0.000 raw:  0.000
Crossover: index 2
chrom: 9, 1, 2, 6,22,16 fit: 0.000 raw:  0.000
```

Since some functions call *rand()*, output may vary from above.

REQ4: Implementing and calling function *test_pcbmill*

Function `test_pcbmill()` tests the operation of functions `gene_create_rand_gene`, `mutate_pcbmill`, `crossover_pcbmill`, `gene_print` and `gene_free`. function `test_pcbmill()` is only compiled and called if macro `DEBUG` is defined. The start-up code includes a description of the function calls necessary to implement `test_pcbmill()`.

Functions `mutate_pcbmill`, `crossover_pcbmill` and `gene_print` also use conditional compilation, including debug statements if macro `DEBUG` is defined. In particular, if `DEBUG` is defined:

- `mutate_pcbmill` will use a mutation points of chromosome indexes 2 and 4
- `crossover_pcbmill` will use a crossover point of chromosome indexes 2 and 4

An example of output from a call of `test_pcbmill()` is:

```
PCBMILL gene:
chrom: 1, 4, 0, 3, 5, 2 fit: 0.000 raw:  0.000
Mutate: index1 2 index2 4
chrom: 1, 4, 5, 3, 0, 2 fit: 0.000 raw:  0.000
PCBMILL genes:
chrom: 5, 1, 2, 3, 4, 0 fit: 0.000 raw:  0.000
chrom: 0, 1, 2, 3, 5, 4 fit: 0.000 raw:  0.000
Crossover: index1 2 index2 4
chrom: 2, 3, 4, 0, 1, 5 fit: 0.000 raw:  0.000
```

Since some functions call *rand()*, output may vary from above.

REQ5: Conditional compilation

Functions `test_minfn` and `test_pcbmill` should only be compiled and called if the macro DEBUG has been defined. Debug statements in functions `mutate_pcbmill`, `mutate_minfn`, `crossover_pcbmill` and `crossover_minfn` should also be compiled only when the macro `DEBUG` has been defined. This macro should be defined when the user calls `make` with the argument `DEBUG`.

REQ6: Makefile

We should be able to compile your program using a `makefile`, which should be submitted along with the code of your assignment. All compile commands must include the *"-ansi -Wall -pedantic"* compile options. You program should compile cleanly with these options; **no error or warning is acceptable even during the demo.**

Your `makefile` **<u>must</u>** compile your program incrementally. That is, it should use *object* files as an intermediate form of compilation.

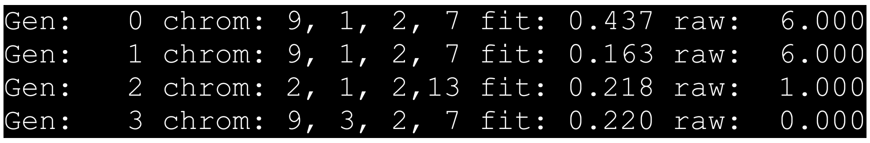
Your `makefile` should permit the defining of macro `DEBUG` from the command line, for use in conditional compilation of `test_pcbmill()` and `test_minfn()`.

You should also include a target called "clean" that deletes unnecessary files from your working directory such as object files, executable files, core dump files etc. This directive should only be executed when the user types "make clean" at the command prompt.

REQ7: Implementing and calling *pop_print_fittest*

Function `pop_print_fittest` should be called each time a population has been evaluated and ready for reproduction. It should cause the the 'fittest' gene in the population to be displayed, along with the number of the current generation. As specified in the start-up code, the function must not be able to access any generation value outside of the function.

A sample output for running *minfn* for 4 generations is:



```
Gen:    0 chrom: 9, 1, 2, 7 fit: 0.437 raw:  6.000
Gen:    1 chrom: 9, 1, 2, 7 fit: 0.163 raw:  6.000
Gen:    2 chrom: 2, 1, 2,13 fit: 0.218 raw:  1.000
Gen:    3 chrom: 9, 3, 2, 7 fit: 0.220 raw:  0.000
```

Since some functions call *rand()*, and different input vector files may be used, output may vary from above.

REQ8: Evolution of the population

The program should correctly create an appropriate initial population and 'evolve' it for a specified number of generations, where each generation is evaluated and reproduced to create the next generation, displaying details of the fittest gene of each generation as described above.

REQ9: Implementation of function pointers

The *Pop_list* data type has three pointers to functions:

- `create_rand_chrom` which can be set to point to a function for creating a random chromosome. Prototypes for two such functions have been provided: `create_pcbmill_chrom` for creating a valid *pcbmill* chromosome, and `create_minfn_chrom` for creating a valid *minfn* chromosome.
- `mutate_gene` which can be set to point to a function for mutating a gene. Prototypes for two such functions have been provided: `mutate_pcbmill` for mutating a *pcbmill* gene, and `mutate_minfn` for mutating a *minfn* gene.
- `crossover_genes` which can be set to point to a function for performing crossover on two genes. Prototypes for two such functions have been provided: `crossover_pcbmill` for performing crossover on *pcbmill* genes, and `crossover_minfn` for performing crossover on *minfn* genes.
- `evaluate_fn` which can be set to point to a function for evaluating the raw fitness of a gene. Prototypes for two such functions have been provided: `eval_pcbmill` for evaluating a *pcbmill* gene, and `eval_minfn` for evaluating a *minfn* gene.

Your program should appropriately set and use these functions pointers.

REQ10: Validating the input file

In the previous requirements, it was assumed that the input vector data file that you have loaded into memory were valid and contained no error. However, in real life this is not necessarily the case and you will be required to detect errors that may arise from reading a corrupt or invalid data file.

Examples of the corrupt and/or invalid data files include the following:

- There are lines of data in the file which does not contain enough fields to constitute a valid input vector;
- There are lines of data in the file which contain too many fields for a valid input vector;
- The number of fields is correct; however, the type of data available for one or more of the fields don't match the expectation of the program;
- The data types match the expectation, but they are out of the acceptable range or have otherwise invalid values;
- The data file is empty;
- The data file doesn't exist

REQ11: Producing the output file

If command-line argument `outputFile` is provided, a file with the same name as the argument should be created/truncated to zero length and all output directed to the file.

REQ12: Memory leaks and abuses

The start-up code requires the use of dynamic memory allocation. Therefore, you will need to check that your program does not contain memory leaks. Use the following *valgrind* command to check for memory leaks on problem *minfn* with a population 20 run for 50 generations and submit the report in a text file named Requirement11a.txt along with the rest of the files in your project.

```
valgrind --leak-check=full --show-reachable=yes <command> <arguments>
```

Another common problem is memory abuses, which refers to situations such as reading from uninitialised memory, writing to memory addresses you should not have access to, and conditional statements that depend on uninitialised values. You can test for these again by using valgrind:

```
valgrind –track-origins=yes <command> <arguments>
```

The report generated by this command must be stored in a new file named Requirement11b.txt and included in your submission. Marks will only be awarded for this requirement if the reports generated by valgrind indicate that there are no memory leaks nor any other memory related problems.

REQ13: Abstract Data Types

In this assignment, you are implementing three Abstract Data Types; gene, population and InVTable. A number of function prototypes for gene have been provided, but the interface for the population and the InVTable are largely for you to develop. For this requirement, you will need to propose a list of interface functions for the population and InVTable ADTs and implement these. All references to these types should be via these interface functions.

REQ14: General requirements

You must read and follow the “Buffer Handling”, “Input Validation” and “Coding Conventions and Practices” requirements written in the “General Requirements” section of this specification.

REQ15: Demo

A demonstration is required for this assignment in order to show your progress. This will occur in your scheduled lab classes on the Unix machines used in this course. Demonstrations will be very brief, and you must be ready to demonstrate your work in a two-minute period when it is your turn. You must also have already uploaded your files to the assignment 2 submission link as proof of your work.

As part of the assignment demo, the tutor may ask you random questions about your source code. You may be asked to open your source files and explain the operation of a randomly picked section. In the event that you will not be able to answer the questions, we will have to make sure that you are the genuine author of the program.

Input validation will not be assessed during demonstrations, so you are advised to incorporate these only after you have implemented the demonstration requirements. Coding conventions and practices too will not be scrutinised, but it is recommended that you adhere to such requirements at all times.

During the demonstration you will be marked for implementing the requirements REQ3, REQ4 and REQ5 which should be completed and functional.

GENERAL REQUIREMENTS

GR1: Buffer handling

Your program must not suffer from buffer overflow if input is larger than expected.

GR2: Coding conventions and practices

Marks are awarded for good coding conventions/practices such as:

- Completing the header comment sections in each source file included in the submission.
- Avoiding global variables. If you still do not know what global variables are, do not assume. Ask the teaching team about it.
- Avoiding *goto* statements.
- Consistent use of spaces or tabs for indentation. Either consistently use tabs or three spaces for indentation. Be careful not to mix tabs and spaces. Each "block" of code should be indented one level.
- Keeping line lengths of code to a reasonable maximum such that they fit into 80 columns.
- Appropriate commenting to explain non-trivial sections of the code.
- Writing header descriptions for all functions.
- Appropriate identifier names.
- Avoiding magic numbers (remember, it is not only about numbers).
- Avoiding platform specific code with *system()*.
- General code readability.

GR3: Functional abstraction

We encourage the use of functional abstraction throughout your code. This is considered to be a good practice when developing software with multiple benefits. Abstracting code into separate functions reduces the possibility of bugs in your project, simplifies programming logic and make the debugging less complicated. We will look for some evidence of functional abstraction throughout your code.

As a rule, if you notice that you have the same or very similar block of code in two different locations of your source, you can abstract this into a separate function. Another easy rule is that you should not have functions with more than approximately 50 lines of code. If you have noticeably longer functions, you are expected to break that function into multiple logical parts and create new functions accordingly.

START-UP CODE

We provide you with a start-up code that you must use in implementing your program. The start-up code includes several files which you must carefully read, understand and complete.

STRUCTURE AND FILES

The start-up code contains the following files:

- `drive.h` and `drive.c`: This is the main header and source file. The main method should be implemented in `drive.c` file. This is also the only place that *srand* should be called.
  `gene.h` and `gene.c`: These files include the declaration and body of the functions related to genes.
- `pop.h` and `pop.c`: These files contain the body of the logic for creating, managing and running a population of genes to implement a genetic algorithm.
- `invector.h` and `invector.c`: These files contain functions related to reading, validating, storing and accessing input vectors.
- `utility.h`: This file is used to declare the Boolean type used by multiple other modules.

In addition to the provided functions in the files, you may need to use functional abstraction to break down the logic of some functions into calling additional smaller functions.

MARKING GUIDE

The following table contains the detailed breakdown of marks allocated for each requirement.

Requirement	**Mark**
REQ1: Command-line arguments	6
REQ2: Loading the vector input data file	6
REQ3: Implementing and calling *test_minfn*	5
REQ4: Implementing and calling *test_pcbmill*	5
REQ5: Conditional compilation	6
REQ6: Makefile	5
REQ7: Implementing and calling *pop_print_fittest*	3
REQ8: Evolution of the population	20
REQ9: Implementation of function pointers	6
REQ10: Validating the input file	3
REQ11: Producing the output file	2
REQ12: Memory leaks and abuses	5
REQ13: Abstract data types	6
REQ14: General requirements	12 (4+4+4)
REQ15: Demonstration in lab	10
Total	**100**

PENALTIES

Marks will be deducted for the following:

- Compile errors and warnings (especially ensure it compiles with the `-ansi` flag!!).
- Fatal run-time errors such as segmentation faults, bus errors, infinite loops, etc.
- Missing files (affected components get zero marks).
- Files incorrectly named, or whole directories submitted.
- Not using start-up code or not filling-in the readme file.

Programs with compile errors that cannot be easily corrected by the marker will result in a maximum possible score of 40% of the total available marks for the assignment.

Any sections of the assignment that cause the program to terminate unexpectedly (i.e., segmentation fault, bus error, infinite loop, etc.) will result in a maximum possible score of 40% of the total available marks for those sections. Any sections that cannot be tested as a result of problems in other sections will also receive a maximum possible score of 40%.

It is not possible to resubmit your assignment after the deadline due to mistakes.

SUBMISSION INFORMATION

Create a zip archive using the following command on *saturn*, *jupiter* or *titan*:

```
zip ${USER}-a2 \
drive.h drive.c \
gene.h gene.c \
invector.h invector.c \
pop.h pop.c \
makefile \
Requirement11a.txt Requirement11b.txt
```

This will create a .zip file named “username-a2.zip” with your username substituted for “username”. Note that **you must test your archive file before submission** to make sure that it is a valid archive, contains the necessary files, and can be extracted without any errors.

Submit the zip file before the submission deadline.

LATE SUBMISSION PENALTY

Late submissions attract a marking deduction of 10% of the maximum mark attainable per day for the first 5 days, including weekdays and weekends. After this time, a 100% deduction is applied.