Department of Computer Science

CS2004 Algorithms and their Applications
Description of the Coursework Assessment
Tasks #2
Assessment/Coursework for 2022-2023

TABLE OF CONTENTS

Assessment Title	CS2004 Task #2 (2022-2023)
Module Leader	Dr Mahir Arzoky
Distribution Date	Week 23
Submission Deadline	See below
Feedback by	See below
Contribution to overall module assessment	70% of the coursework component
Indicative student time working on assessment	As per main coursework description
Word or Page Limit (if applicable)	N/A - No limit
Assessment Type (individual or group)	Individual

PURPOSE OF THIS DOCUMENT

This document describes in detail the Task #2 assessment for the CS2004 Algorithms and their Applications (2022- 2023) module, i.e. the more significant programming task. This document should be read in conjunction with the following brief on Brightspace:

"CS2004 Assessment Brief Tasks 1 and 2 (2022-2023) [Provisional].pdf"

The overall objective of this assignment is to produce Java programming code that forms a set of functions that can be used to solve the Eight-Queens problem [described below and in Lecture 12. Further Evolutionary Computation] using a heuristic search algorithm.

DESCRIPTION OF THE ASSESSMENT

This substantial programming assignment will be assessed by **CodeRunner** during the University's (May) examination period. The exact date of the examination has not been set, but it is noted that the examinations period falls between 02/05/2023 and 19/05/2023 (weeks 33-35). More information can be found on Brightspace.

The test questions will be outlined in a subsequent section of this document. As will be seen from the description below, material from all of the assessed worksheets could (and probably will) be very useful to complete this task.

The Eight-Queens problem is a classic problem in computer science and mathematics. It involves placing eight queens on a standard 8×8 chessboard, in a way so that none of them pose a threat to the other queens. As a result, there cannot be more than one queen in a single row, column, or diagonal (see Figure 1). The objective is to find a solution to place the eight queens such that no two threaten each other. The Eight-queens problem is a common illustration of a broader class of problems known as constraint satisfaction problems, where a

solution must satisfy a set of restrictions. The problem has been solved in different ways and has inspired a great deal of research in computer science and mathematics.

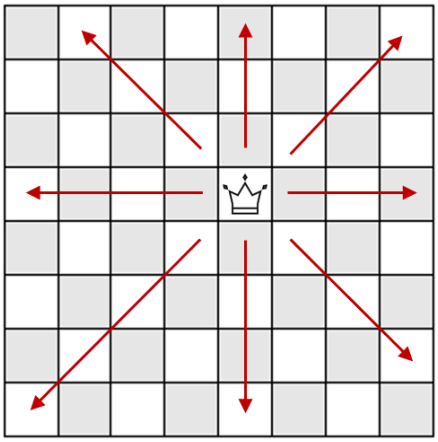

Figure 1 – Eight-queens problem board highlighting the squares that can be attacked [for this queen/position]

Solving the problem can be quite computationally expensive – there are 4,426,165,368 possible arrangements (combinations) of eight queens on an 8×8 chessboard, but only 92 solutions. A heuristic search algorithm can be used to traverse the space of possible solutions using a fitness function to locate the best solution.

The purpose of this assignment is to write Java code that solves the functional requirements listed in Table 1, i.e. develop some or potentially all of the parts of a Heuristic Seach technique to solve the Eight-Queens. ANY single population algorithm can be used. The representation for the Eight-Queens problem must be in binary form, see examples below. A solution will be a 24 length String as described in Lecture 12, Further Evolutionary Computation, where each part (char) of the String will be a '1' or '0' only.

The fitness function is as follows:

- Compute the total number of clashes (both directly and indirectly) i.e. the total number of attacks, where the queens pose threat to other queens
- Compute the maximum number of attacks possible by the eight queens placed on the board (56 = 8×7)
- The fitness function would be the maximum number of attacks minus the total number of clashes, see the equation below.

$$Fitness(Solution) = Total\ Attacks - \sum_{i=1}^{8} Number\ of\ queens\ attacked\ by\ queen\ i$$

Refer to Figure 2 for a **correct** board example – the binary representation of the example board is as follows: 100010111011110000101001, and the fitness function is 56-0 = 56. Refer to Figure 3 for a **valid** board example – the binary representation of the example board is as follows: 011000100110000111000101, and the fitness function is 56-14 = 42. The difference between correct and valid boards is described below.

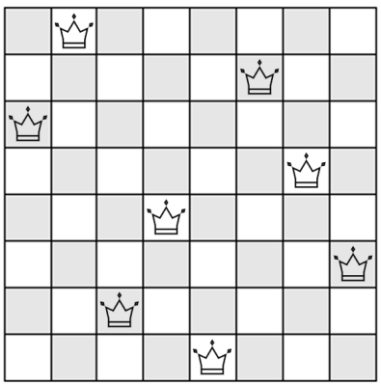

Figure 2 – a correct Eight-Queens board solution

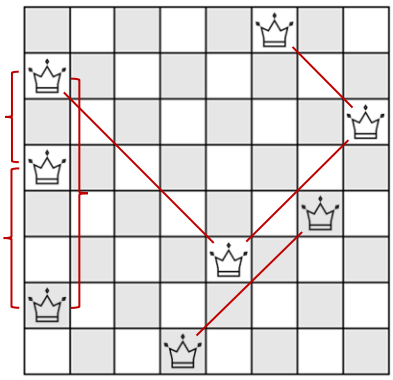

Figure 3 – a valid but incorrect Eight-Queens board solution

When writing your program, please note the following:

1) A chessboard will be represented as an eight (8) row by eight (8) column `Character` array. Note that row and column numbers (parameters – see below) start at one (1), whilst Java arrays start at zero (0).
2) The board (Java representation) can only contain the characters '.' [dot – no queen in a square] and 'Q' [a square has a queen in it]. Lower case letters are invalid. It is assumed that there are no blank (empty/null) squares.
3) If a board is the specified size (8×8), contains valid characters (as per #2 immediately above), and there are exactly eight queens on the board then it is **valid**.
4) A **correct** board has a fitness of 56, i.e. no queens are attacking any other queen.
5) By definition, a **correct** board is also **valid**. However, a **valid** board is not necessarily **correct**.

There will be seven [7] questions during the assessment that will require methods/class to be implemented to perform certain tasks. See Table 1 for details along with the marks for each question. Note that some of the solutions could benefit from calling methods that have been implemented for previous questions (see Appendix B). Also, refer to Appendix C for information on how multiple classes could be used within CodeRunner.

As with all **CodeRunner** tests, your solutions will be tested rigorously [automatically]. For example, invalid input and test data (`null`, empty Arrays, the wrong size, etc..) will be used. Your methods **MUST** be able to cater for this. The exact return conditions [error values] will be specified during the examination.

Note [VERY IMPORTANT]: you are expected to have completed your programming code before the **CodeRunner** examination. Do not turn up to the examination with the preconception that you can complete the assignment on the day, if you do so, it will be highly unlikely that you will be able to pass, given the complexity of the assignment and the duration of the examination.

Method/Feature	Marks	Method/Feature Description
Q1 Check Character	1	Write a method, that returns true or false depending on whether a character (`Character`) [the input] is a valid board square as defined/described above. Note that `null` is not valid.
Q2 Valid Board	2	Write a method that returns true if the input board (`Character` array) is a valid board (as defined/described above), return false otherwise. Note that valid means the board is the correct size, not `null`, **each** square contains a valid character and there are exactly **eight** queens. For this question, you **DO NOT** need to check that the board is **correct**.
Q3 Generate Binary String	3	Write a method to convert an input board (`Character` array) to the corresponding binary String. Assume that the board is valid.
Q4 Initial Starting Point	2	Write a method that creates the initial random starting point (arrangement). The random starting arrangement should be of a binary String format.
Q5 Fitness Function	3	Write a method that takes in a solution to the Eight-Queens problem (a binary String which represents an arrangement of the queens on a board) and returns the fitness value based on the description above.
Q6 Small Change Operator	2	Write a method that takes in a binary String as input, flips a single random bit of the String then returns the changed copy of the arrangement as output. **Make sure that the modified copy of the solution is returned.**
Q7 Solving the Problem	7	Write a class that when given a number of iterations it solves the Eight-Queens problem and return the solution as output. This class will contain the components from the previous questions. Any single population heuristic search algorithm can be used. Note that marks (up to 4 marks) will also be awarded according to a set of tests, i.e. based on the quality of the returned solution. The quality of the solutions (fitness returned for this question) will be

Method/Feature	Marks	Method/Feature Description
		scaled between the maximum possible fitness and the lowest possible fitness (i.e. 56 and 0). Here an overall average fitness of 56 (on the tests) will attain 4 marks, whilst an overall average fitness of 0 would attain 0 marks). **Details to note:** 1) A Java class will need submitting and NOT just a method. Refer to Appendix C and Brightspace for examples on how to prepare your class/classes. The method and class prototype will be specified with the question on the day of the examination. 2) The code for the problem will need to be called through a static method that takes a single parameter as described above – this will be specified/provided with the question. 3) The required number of fitness function calls (iterations) will be an integer, e.g. int, short, Integer or Short. 4) Normal checks on the input parameters will need to be made, e.g. the fitness function calls (iterations) being greater than one [1] etc... Marks will be awarded for this. 5) The method will need to return a solution to the problem, which is specified as a **String** (see above).

Table 1. Functional Requirements for the Task #2 Assessment

FORMAT OF THE ASSESSMENT

Java program code will need to be individually written. You will need to have access to this code during the examination. See **CodeRunner** mock test worksheet. Also see CS2004 Task #1 and #2 Assessment Brief and the Study Guide.

The followings are important additional details to note:

1) Your methods/class just need to answer questions 1-7.
2) You will not need a user interface or any input from any user.
3) None of your methods should display any text to the screen, i.e. there is no need for calling `System.out.println`.
4) The problem will need a heuristic search technique to solve the problem, **ANY single population algorithm** can be used.
5) Do not be tempted to run your code for a long period of time, ignoring the specified number of fitness function calls (iterations). We will check if the run time for a large and small number of fitness function calls is not the same!
6) It is very easy to set up a test-rig to simulate how we are going to implement the **CodeRunner** test, you should try and do this as it will make the actual test easier for you to complete. If you are not sure, ask during the laboratories.
7) Further information will be posted on Brightspace in regard to dealing with imports and accessing/using extra Java classes such as CS2004.java (which will be imported to CodeRunner and can be used).
8) We will be running all submitted code through a plagiarism and similarity checker.

AVOIDING ACADEMIC MISCONDUCT

Before working on and then submitting your coursework, please ensure that you understand the meaning of plagiarism, collusion, and cheating (including contract cheating) and the seriousness of these offences. Academic misconduct is serious and being found guilty of it results in penalties that can reduce the class of your degree and may lead to you being expelled from the University. Information on what constitutes academic misconduct and the potential consequences for students can be found in Senate Regulation 6.

You may also find it useful to read this PowerPoint presentation which explains, in plain English, the different kinds of misconduct, how to avoid (even accidently) committing them, how we detect misconduct, and the common reasons that students give for engaging in such activities.

If you are experiencing difficulties with any part of your studies, remember there is always help available:

- Speak to your personal tutor. If you're not sure who your tutor is, please ask the Taught Programmes Office (TPOcomputerscience@brunel.ac.uk).
- Alternatively, if you prefer to speak to someone outside of the Department you can contact the Student Support and Welfare team.

LATE COURSEWORK

The clear expectation is that you will submit your coursework by the submission deadline stated in the study guide. In line with the University's policy on the late submission of coursework (revised in July 2016), coursework submitted up to 48 hours late will be accepted, but capped at a threshold pass (D- for undergraduate or C- for postgraduate). Work submitted over 48 hours after the stated deadline will automatically be given a fail grade (F).

Please refer to the Computer Science student information pages and the Coursework Submission Procedure pages for information on submitting late work, penalties applied and procedures in the case of Extenuating circumstances.

APPENDIX A: TASK #2 GRADING CRITERIA

The **CodeRunner** test is marked out of 20 marks (see above). This will be converted to a percentage and then a grade point assigned [the full range F – A* for this task] as per Senate Regulations SR2. Table 2 below is a *reduced* version of SR2.

$$\text{Percentage} = \frac{\text{Mark}}{20.0} \times 100\%$$

Task #2	Grade Range
More than 70%	A (A- to A*)
≥60% and <70%	B (B- to B+)
≥50% and <60%	C (C- to C+)
≥40% and <50%	D (D- to D+)
≥30% and <40%	E (E- to E+)
Less than 30%	F

Table 2. Task #2 Grading

APPENDIX B: MULTIPLE METHODS AND CODERUNNER

You may deem it necessary to reuse answers from previous questions, for example, use methods in one question that you wrote for a previous question. You will be able to submit as many methods as you need. However, care must be taken that you only submit the methods and not the class details. For example, if a **CodeRunner** question asked:

Write a public static String method called ExampleCRAnswer that returns the String "Hello World!" (not including the quotation marks).

Then you might have written a class as follows to test the answer:

```
public class ExampleCRQuestion {
        public static void main(String args[]) {
                System.out.println(ExampleCRAnswer());
        }
        public static String ExampleCRAnswer() {
                return(Method1() + " " + Method2());
        }
        public static String Method1() {
                return("Hello");
        }
        public static String Method2() {
                return("World!");
        }
}
```

Here the answer is constructed by calling two extra methods. To submit this, all you would need is the answer method and the methods it calls:

```
public static String ExampleCRAnswer() {
        return(Method1() + " " + Method2());
}
public static String Method1() {
        return("Hello");
}
public static String Method2() {
        return("World!");
}
```

You will need to make sure the correct number of curly brackets [{ and }] are included; using the correct level of indentation within your program will make this easier to get right.

APPENDIX C: CLASSES AND CODERUNNER

You may submit your solution for Question 5 of the CodeRunner examination (Task #2) in any of the following ways:

Single Class

All your work in one class.

Multiple Classes

```
public class MultiClass {
        public static void main(String args[]) {
                ClassOne c1 = new ClassOne(21);
                ClassTwo c2 = new ClassTwo(-600.4);
                System.out.println(c1.GetData() + c2.GetNumber());
        }
}

class ClassOne {
        private int data1 = 0;
        public ClassOne(int x) {
                data1 = x;
        }
        public int GetData() {
                return(data1);
        }
}

class ClassTwo {
        private double data2 = 0;
        public ClassTwo(double y) {
                data2 = y;
        }
        public double GetNumber() {
                return(data2);
        }
}
```

In the multiple classes example above, note the following:

1) The first class should be as normal and has the same name as the Java file
2) You can have as many extra classes listed separately after the main class

Inner Classes

```
public class InnerClass {
        public static void main(String args[]) {
                ClassOne c1 = new ClassOne(21);
                ClassTwo c2 = new ClassTwo(-600.4);
                System.out.println(c1.GetData() + c2.GetNumber());
        }

        static class ClassOne {
                private int data1 = 0;
                public ClassOne(int x) {
                        data1 = x;
                }
                public int GetData() {
                        return(data1);
                }
        }

        static class ClassTwo {
                private double data2 = 0;
                public ClassTwo(double y) {
                        data2 = y;
                }
                public double GetNumber() {
                        return(data2);
                }
        }
}
```

In the inner classes example above, note the following:

1) The first class should be as normal and has the same name as the Java file
2) The inner classes should be static and listed as shown in the example (double check that the brackets are in the right place!)