Department of Computer Science

# CS2004 Algorithms and their Applications
# Description of the Coursework Assessment
# Tasks #2
## Assessment/Coursework for 2021-2022

**TABLE OF CONTENTS**



| | |
|---|---|
| **Assessment Title** | CS2004 Task #2 (2021-2022) |
| **Module Leader** | Dr Mahir Arzoky |
| **Distribution Date** | As per main coursework description |
| **Submission Deadline** | As per main coursework description |
| **Feedback by** | As per main coursework description |
| **Contribution to overall module assessment** | As per main coursework description |
| **Indicative student time working on assessment** | As per main coursework description |
| **Word or Page Limit (if applicable)** | As per main coursework description |
| **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 (2021- 2022) module, i.e. the more significant programming task. This document should be read in conjunction with the following document on Blackboard:

***"CS2004 Assessment Brief Tasks 1 and 2 (2021-2022) [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 software module clustering problem [described below] using a heuristic search algorithm.

## DESCRIPTION OF THE ASSESSMENT

This substantial programming assignment will be assessed by **CodeRunner**, followed by a **verbal viva** (assessment) in Week 26. You must pass the verbal viva to pass the CodeRunner assessment. Refer to Appendix E for more details. The dates of the assessment can be found in Table 1. More information can be found on Blackboard.

| Task #2 Assessment | Format | Date | Feedback |
|---|---|---|---|
| CodeRunner | CodeRunner | Monday, Week 26 | Immediate / 3 weeks |
| Viva | Pass/Fail | Wednesday, Week 26 | 3 weeks from the viva date |

Table 1. Task #2 assessment and format



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.

Software module clustering is the problem of automatically partitioning the structure of a software system using low-level dependencies in the source code to understand and improve the system's architecture. With the tendency for systems to decay over time and generate associated maintenance problems, software module clustering has significant resonance for developers and project managers alike. It is widely believed that software systems that were modularised are easier to develop and be maintained. The software module clustering problem involves finding good quality software modules clusters based on the relationships amongst the modules. Closely related modules are grouped into clusters that are loosely connected to other clusters.

Components such as modules, classes or subroutines of a system are represented as nodes and the interrelationships between the components are represented as edges. Such graphs are referred to as a **Module Dependency Graph (MDG)**. An MDG can be represented as follows: If there are *n* nodes to represent, for an *n* by *n* matrix *M*, a non-zero value of $M_{ij}$ (ith row, jth column of *M*) means there is an edge between node *i* and *j*. The matrices produced are symmetrical. Figure 1 illustrates how the matrix is represented. As the dataset is non-weighted, $M_{ij}$ is either one for an edge or zero for no edge i.e. one for a relationship and zero for no relationship.

$$\begin{bmatrix} 0 & 1 & 0 & 0 & 1 \\ 1 & 0 & 1 & 0 & 0 \\ 0 & 1 & 0 & 1 & 1 \\ 0 & 0 & 1 & 0 & 0 \\ 1 & 0 & 1 & 0 & 0 \end{bmatrix}$$

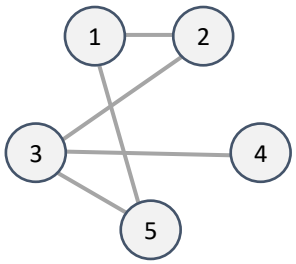


Figure 1 – Matrix representation of an MDG and its corresponding graph

**Munch** is a software clustering tool that can be used to modularise source code software components (referred to as modularisation). A heuristic algorithm is used to traverse the space of possible solutions using a fitness function to locate the best solution. Thus, the problem will need a heuristic search technique to solve the task at hand, **ANY single population algorithm** can be used. It uses an MDG as input and produces a partition of the MDG as output. It partitions the system into clusters. A cluster is a set of the modules in each partition of the clustering. A **cluster** will be represented as a vector *C* where $c_i$=*j* means that object *i* is in cluster *j.* For example *C* = [1,2,3,1,2,3] (Number of clusters, *k*=3), Figure 2 shows the graphical representation of this clustering.

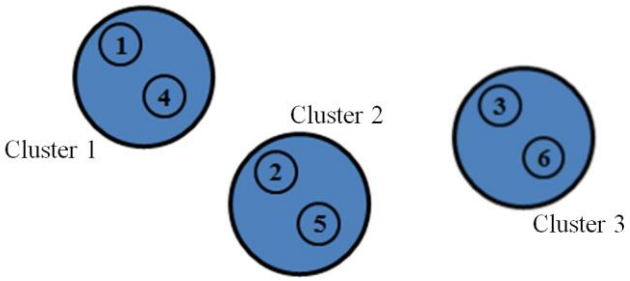


Figure 2 – A graphical representation of clustering

More information on the modularisation problem and the Munch clustering algorithm can be found here: https://ieeexplore.ieee.org/abstract/document/5954442.

There will be five [5] questions during the assessment that will require methods/class to be implemented to perform certain tasks. See Table 2 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 ArrayList, 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 Is Valid MDG | 3 | Write a method that takes in as input an MDG and returns true or false depending on whether the MDG is valid, as defined/described above. The MDG should be represented as a 2-dimensional array of integers. |
| Q2 Initial Starting Point | 2 | Write a method that creates the initial starting clustering point (arrangement). The starting clustering arrangement, which should be in vector format [as described above] i.e. ArrayList of integers, should assume every element/variable/module is in its own cluster. It assumes that all modules are independent and there is no relationship between the modules. |
| Q3 Fitness Function | 5 | Write a method that takes in a clustering arrangement (of type ArrayList of integers) and the MDG, and returns the fitness value based on the EVM fitness function described in Appendix A. |
| Q4 Small Change Operator | 4 | Write a method that changes the value of a single random element/variable of the clustering arrangement (of type ArrayList of integers) to another random value, where the two values are not the same and the new value is less than the size of the clustering arrangement. It mimics moving an element from one cluster to another. The method should take the clustering arrangement as input and returns the changed clustering arrangement as output. |
| Q5 Solving the problem | 6 | Write a class that when given the MDG and the number of iterations it applies the Munch clustering algorithm and return the clustering arrangement as output. This class will contain the components from the previous questions. Appendix A contains the pseudocode of the Munch clustering algorithm using the Random Mutation Hill Climbing algorithm. However, any single population heuristic search algorithm can be used.<br><br>**Details to note:**<br>1) A Java class will need submitting and NOT just a method. Refer to Appendix C and Blackboard for examples on how to prepare your class/classes. The method and class prototype will be specified with the question.<br>2) The code for the problem will need to be called through a **static** method that takes two parameters as described above – this will be specified/provided with the question.<br>3) The required number of fitness function calls will be an integer, e.g. int, short, Integer or Short.<br>4) Normal checks on the input parameters will need to be made, e.g. the fitness function calls being less than one [1] etc... Marks will be awarded for this.<br>5) The method will need to return a solution to the problem, which will be specified as an ArrayList of integers.<br>6) Bonus marks will be awarded according to a set of test datasets i.e. based on the quality of the returned solution (see below). |

Table 2. Functional Requirements for the Task #2 Assessment



**Bonus Five (5) Marks**: marks will be awarded for the accuracy of the results produced from the Munch clustering algorithm i.e. the quality of the clustering arrangements produced from running the algorithm. There will be up to five (5) marks that could be attained for this element. The quality of the solutions (from Q5) that will be produced from the whole cohort will be ranked and scaled. Student(s) that produce the best solution will attain the full 5 marks, student(s) that produce the worst solution will attain zero marks, the rest of the solutions from the cohort will be ranked and scaled between 5 and 0, as appropriate.

As per Appendix D, this assignment will still be assessed out of 20 marks, hence if you attain more than 20 marks (given it could now be out of 25) you will get 20 marks (A*).

## Additional Details To Note

1) Your methods/class just need to answer questions 1-5 (and perhaps the bonus question).
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. 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 Blackboard 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.

## 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 coursework description and study guide.

## 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: THE MUNCH CLUSTERING ALGORITHM AND THE FITNESS FUNCTION

### Munch Clustering Algorithm

```
Algorithm 1. Munch(ITER,M)
Input: ITER is the number of iterations (runs)
       M is an MDG
1) Let C be a random clustering arrangement
2) Let F = Fitness Function
3) For i = 1 to ITER
4)    Choose a random variable, x = UI(1,n)
5)    Set C[x] to UI(1,n) (not the same value)
6)    Let F'= Fitness Function
7)    If F' is worse than F Then
8)       Undo change
9)    Else
10)       Let F = F'
11)    End If
12) End For
Output: C - a modularisation of M
```

UI is a uniformly distributed random number generator function, part of the CS2004 Java class provided alongside some of the laboratory worksheets.

### EVM Fitness Function

For the following formal definition of *EVM*, a clustering arrangement *C* of *n* items is defined as vector *C*, where $c_i$=*j* means that object *i* is in cluster *j* . For example, C = [1, 2, 3, 4, 1, 1, 1, 1, 1] represents the partition {{1, 5, 6, 7, 8, 9}, {2}, {3}, {4}}. Let MDG *M* be an *n* by *n* matrix, where a one at row *i* and column *j* ($M_{ij}$) indicates a relationship between variable *i* and *j*, and zero indicates that there is no relationship.

The Pseudocode of the EVM fitness function is as follows:

```
Algorithm 2. Fitness(C,M)
Input: C is an ArrayList of integers
       M is a 2-D numerical array of size n>0
1) Let EVM = 0
3) For j = 0 to n-1
4)    For k = j+1 to n
5)       Let c1 = C[j]
6)       Let c2 = C[k]
7)       If c1 = c2 then
8)          EVM = EVM + 2*M[j][k] - 1
9)       End if
10)    End For
11) End For
12) Return EVM
Output: EVM, the fitness function value
```



## 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!)



## APPENDIX D: 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 3 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 3. Task #2 Grading



## APPENDIX E: VERBAL VIVA

The format of the verbal viva assessment is as follows:

1) The viva will be held in-person on Wednesday Week 26. The location is Tower A 407/408. More information will be provided on Blackboard
2) You must attend your allocated time slot on the day, details will be sent to you soon
3) An examiner will go through the Java code that you've submitted through the CodeRunner assessment with you, and ask questions about the code as appropriate
4) The viva will take approximately 15 minutes
5) Make sure that you bring your student ID card with you on the day
6) You will need to pass the verbal viva to pass the CodeRunner assessment