Assignment 2

Local search

Date Due: 15 October 2018, 11:59pm **Total Marks: 50**

General Instructions

- **This assignment is individual work.** You may discuss questions and problems with anyone, but the work you hand in for this assignment must be your own work.
- Each question indicates what to hand in.
- Do not submit folders, or zip files, even if you think it will help.
- **Assignments must be submitted to Moodle.**

Execution instructions

The markers may or may not wish to run your program, to verify your results. To help them, you should provide brief instructions on what to do to get your program running. Include:

- Programming language used (including version, e.g., Java 8 or Python 3)
- A simple example of compiling and/or running the code from a UNIX shell would be best.
- Ideally, the marker will be able to run your code from a UNIX command-line.

Keep it brief, and name it with the question number as the following example: `a2q1_EXECUTION.txt`. If your assignment uses third party libraries, they have to be included in your submission.

Version History

- **04/10/2018**: Added a clarification about the term "neighbouring state."
- **04/10/2018**: Added a clarification about time and iterations. Just use an iteration counter. Don't worry about checking the time.
- **04/10/2018**: Clarified the approach to be taken, contrasting it with A1. Each of Q1-Q3 now has a specific direction about how to move from state to state.
- **03/10/2018**: released to students

Overview

We start by defining a simple calculating machine. The machine has one register, R, which can store a double precision floating point value (integers will be automatically converted). The machine has the following instructions:

Instruction	Effect
NOP v	R does not change
ADD v	R = $R + v$
SUB v	R = $R - v$
MUL v	R = $R \times v$
DIV v	R = R/v (floating point division)

Notice that the NOP operation effectively tells the machine to ignore the given number. Here's an example script with 5 instructions, listed horizontally for convenience:

```
MUL 1, NOP 2, ADD 3, SUB 4, DIV 5
```

This saves space on the page for examples. We will assume that the register R is initialized to 0.0 (zero) at the start of every script. With this assumption, after executing the above sequence of instructions, the register would contain a floating point value close to -0.2.

Targeted Coding Problem

The **Targeted Coding Problem** is stated as follows. You are given a floating point target value T, and a list of numbers, L, of any length. You are to find a sequence of instructions for our simple machine above that calculates a value as close to T as possible, using all the numbers in L. You must use every number in L, and you cannot change the order. This means that the only thing you can do is figure out which operator to give to each value in the list. For example, given $L = [1, 2, 3, 4, 5]$, you could try:

```
MUL 1, ADD 2, ADD 3, DIV 4, ADD 5
```

or

```
ADD 1, NOP 2, NOP 3, SUB 4, DIV 5
```

You may use any of the 5 operators any number of times. These restrictions are artificial, but they allow us to explore the local search ideas without over-complicating the task.

The objective is to minimize the difference between the target, T, and the result of a sequence of instructions (as described above). To make this precise, let $m(s)$ represent the value in the simple machine's register after executing the instruction sequence s. The **objective function** is defined as follows:

$$f(s, t) = |t - m(s)|$$

where s is any sequence of instructions, and t is the target value.

Programming ideas

The machine It's easy to write a simple interpreter for this machine. In Python it would look something like this:

```
def machine_exec(seq):
    register = 0
    for instruction in seq:
        operator = instruction[0]
        operand = instruction[1]
        if operator == "ADD":
            register += operand
        elif operator == "SUB":
            register -= operand
        elif operator == "MUL":
            register *= operand
        elif operator == "DIV":
            register = register / operand
        elif operator == "NOP":
            pass
        else:
            print('unknown operator', operator)
    return register
```

You can implement the above function in any language. You don't need to parse a textfile containing a script. You can represent a script using lists or arrays. For example, in Python you might use something like this:

```
[('MUL', 1), ('ADD', 2), ('MUL', 3), ('SUB', 4), ('MUL', 5)]
```

For non-Python programmers, 2-dimensional arrays are completely acceptable. Once you are sure you're getting correct results, you can even start using a compiled form, by turning operator strings to integers, to save space and save time.

```
[(1, 1), (0, 2), (1, 3), (2, 4), (1, 5)]
```

It's machine code now, and harder to read. But the first value is the compiled operator, and the second is the operand. It may even be advantageous to split the operators and and operands into two lists (or arrays):

```
operations = [1, 0, 1, 2, 1]
operands = [1, 2, 3, 4, 5]
```

This might be useful because we'll be changing the operations, but not the operands. You'll have to modify your `machine_exec()` program.

Clarification In Assignment 1, we explored a similar problem using a constructive approach: an expression was constructed from an initially empty expression. It's rather like the textbook example of solving the 8-Queens problem starting from an empty board, and placing the queens on the board one at a time. This approach is what AIMA Ch 3 is all about.

In Assignment 2, we want to explore AIMA Ch 4.1. We'll take a *mutative* approach instead. It will be like the textbook example of solving 8-Queens, putting all 8 queens on the board to start, and then solving the problem by moving queens around, one at a time.

To be clear, every question in Assignment 2 should start with a complete script that uses all the numbers in the given order, and has an operator for each one. For example, suppose our target is $T = 40$, and our list is $L = [1, 2, 3, 4, 5]$. You'd start every local search strategy with a random program using the numbers, perhaps:

```
Add 1, Mul 2, Add 3, Nop 4, Div 5
```

All the numbers are used, in the order given; the script is built all at once, not one step at a time as in A1. Your Problem class should have a method to create random scripts like this for lists L of any length.

After that, your search algorithms will explore other complete scripts by various means. In Q1, you'll just create completely random scripts like this. In Q2, you'll make a random change to a scripts (by selecting one of the operations, and changing it at random. In Q3, you'll look at all scripts that are exactly one operation change away, and keep the best one. Etc. Each time you have a complete script, and each script will have a value, namely its score according to the objective function defined earlier.

Software Design As in Assignment 1, it's a good decision to separate the programming into loosely coupled modules.

- A Problem State class. An ADT to contain information that changes as the search algorithm explores the problem space (also called the state space).

  Clarification (04/10/2018) The problem state should contain a complete script, as explained above.

- A Problem class. An ADT to contain particulars about the Problem. It should have a small number of standard methods.

  Clarification (04/10/2018) The actions() result() model for AIMA Ch 3 is probably not the most convenient. Each Question gives a suggestion on what you'll need.

- A module to contain search algorithms. The search algorithms should interact with the Problem only through the standard methods.

Assignment 1 Question 1 confused a lot of people, so I am leaving it out this time. You have to do similar work here, but it's not officially a "question."

Question 1 (6 points):

Purpose: To implement the Random Guessing search strategy, and apply it to the Targeted Coding problem.

Degree of Difficulty: Moderate. This is where you have to get most of the problem class defined, and do some thinking about design. The search algorithm itself will not be difficult.

The Random Guessing search strategy can be described with the following pseudo-code:

```
function random_guessing(problem)
    // Keep generating completely random states
    // Remember the best one

    best_guess = obtain a random state from the problem
    while there's still time
        guess = obtain a random state from the problem
        if guess is better than best_guess:
            best_guess = guess

    return best_guess
```

To control the time spent searching, implement a simple counter, limiting your algorithm to a given number of iterations of the main loop.

Clarification (10/10/2018) Ignore time. Just count iterations. More iterations means more time. An iteration-limit should probably be passed as an argument, meaning you'll need another parameter in the implementation.

Implement the search strategy, and demonstrate that your implementation works, by applying it to the Targeted Coding problem, maybe a few of the simple examples.

Clarification (04/10/2018) For this problem, your problem class should have a method that creates a complete script randomly.

What to Hand In

This assignment is broken into pieces, and the last question asks for all the source code, so you don't have to hand in multiple copies. But pay attention to these requirements:

- A file named `a2q1.txt` containing a demonstration of the output of your program, which is copy/paste from a console, or output file. You only need to show a few examples. If this file is missing, the marker will assume your implementation is incomplete or not working.
- A file named `a2q1.LANG` containing your implementation of the search algorithm. Use the file extension appropriate for your choice of programming language, e.g., `.py` or `.java`. **For this question, only submit the implementation of the search algorithm, not the problem state or problem class implementations.** The markers will not attempt to run the code you submit.

Be sure to include your name, NSID, student number, and course number at the top of all documents.

Evaluation

- 6 marks: Your implementation of Random Guessing is correct, and well-documented.

Question 2 (6 points):

Purpose: To implement the Random Search search strategy, and apply it to the Targeted Coding problem.

Degree of Difficulty: Easy. Hopefully, your problem class definition can be extended easily!

The Random Search search strategy can be described with the following pseudo-code:

```
function random_search(problem)
    // Randomly generate successors from the current state
    // Move to it if it's better than the current state

    best_guess = obtain a random state from the problem
    while there's still time
        guess = obtain a random neighbour of best_guess
        if guess is better than best_guess
            best_guess = guess
    return best_guess
```

To control the time spent searching, implement a simple counter, limiting your algorithm to a given number of iterations of the main loop.

Clarification (10/10/2018) Ignore time. Just count iterations. More iterations means more time. An iteration-limit should probably be passed as an argument, meaning you'll need another parameter in the implementation.

Implement the search strategy, and demonstrate that your implementation works, by applying it to the Targeted Coding problem, maybe a few of the simple examples.

Clarification (04/10/2018) For this problem, your problem class should have a method that creates a complete script by randomly making a change to a given script.

Clarification (10/10/2018) A "neighbour state" is any state you can get to from the current state. In this context, the neighbours if script S are all the scripts that are different from S by exactly one instruction.

What to Hand In

This assignment is broken into pieces, and the last question asks for all the source code, so you don't have to hand in multiple copies. But pay attention to these requirements:

- A file named `a2q2.txt` containing a demonstration of the output of your program, which is copy/paste from a console, or output file. You only need to show a few examples. If this file is missing, the marker will assume your implementation is incomplete or not working.
- A file named `a2q2.LANG` containing your implementation of the search algorithm. Use the file extension appropriate for your choice of programming language, e.g., `.py` or `.java`. **For this question, only submit the implementation of the search algorithm, not the problem state or problem class implementations.** The markers will not attempt to run the code you submit.

Be sure to include your name, NSID, student number, and course number at the top of all documents.

Evaluation

- 6 marks: Your implementation of Random Search is correct, and well-documented.

Question 3 (6 points):

Purpose: To implement the Hill-climbing Search search strategy, and apply it to the Targeted Coding problem.

Degree of Difficulty: Easy. You might need to refactor your code a bit, to make things nice, but the search algorithm is easy.

The Hill-climbing Search search strategy can be described with the following pseudo-code:

```
function hill_climbing_search(problem)
    // Starting from a random state
    // Obtain the state's best neighbour
    // Move to it if it's better than the current state
    // Stop if no neighbour is better

    best_guess = obtain a random state from the problem
    while there's still time
        guess = obtain the best neighbour of best_guess
        if guess is better than best_guess
            best_guess = guess
        else if best_guess is better than guess
            return best_guess

    return best_guess
```

To control the time spent searching, implement a simple counter, limiting your algorithm to a given number of iterations of the main loop.

Clarification (10/10/2018) Ignore time. Just count iterations. More iterations means more time. An iteration-limit should probably be passed as an argument, meaning you'll need another parameter in the implementation.

Implement the search strategy, and demonstrate that your implementation works, by applying it to the Targeted Coding problem, maybe a few of the simple examples.

Clarification (04/10/2018) For this problem, your problem class should have a method that returns the best neighbour of a given script. Here, a neighbour is any script that is exactly one change away from the given script.

Clarification (10/10/2018) A "neighbour state" is any state you can get to from the current state. In this context, the neighbours if script S are all the scripts that are different from S by exactly one instruction.

What to Hand In

This assignment is broken into pieces, and the last question asks for all the source code, so you don't have to hand in multiple copies. But pay attention to these requirements:

- A file named `a2q3.txt` containing a demonstration of the output of your program, which is copy/paste from a console, or output file. You only need to show a few examples. If this file is missing, the marker will assume your implementation is incomplete or not working.
- A file named `a2q3.LANG` containing your implementation of the search algorithm. Use the file extension appropriate for your choice of programming language, e.g., `.py` or `.java`. **For this question, only submit the implementation of the search algorithm, not the problem state or problem class implementations.** The markers will not attempt to run the code you submit.

Be sure to include your name, NSID, student number, and course number at the top of all documents.

Evaluation

- 6 marks: Your implementation of Hill-climbing Search is correct, and well-documented.

Question 4 (6 points):

Purpose: To implement the Random-Restart Hill-climbing Search search strategy, and apply it to the Targeted Coding problem.

Degree of Difficulty: Easy.

The Random-Restart Hill-climbing Search search strategy simply repeats Hill-climbing a number of times. You should control the number of restarts and the number of steps that Hill-climbing is allowed, using separate parameters.

Implement the search strategy, and demonstrate that your implementation works, by applying it to the Targeted Coding problem, maybe a few of the simple examples.

What to Hand In

This assignment is broken into pieces, and the last question asks for all the source code, so you don't have to hand in multiple copies. But pay attention to these requirements:

- A file named `a2q4.txt` containing a demonstration of the output of your program, which is copy/paste from a console, or output file. You only need to show a few examples. If this file is missing, the marker will assume your implementation is incomplete or not working.
- A file named `a2q4.LANG` containing your implementation of the search algorithm. Use the file extension appropriate for your choice of programming language, e.g., `.py` or `.java`. **For this question, only submit the implementation of the search algorithm, not the problem state or problem class implementations.** The markers will not attempt to run the code you submit.

Be sure to include your name, NSID, student number, and course number at the top of all documents.

Evaluation

- 6 marks: Your implementation of Random-Restart Hill-climbing Search is correct, and well-documented.

Question 5 (6 points):

Purpose: To implement the Stochastic Hill-climbing Search search strategy, and apply it to the Targeted Coding problem.

Degree of Difficulty: Easy.

The Stochastic Hill-climbing Search search strategy is a minor variation on Hill-climbing. Instead of requiring the best neighbour of the current state, you get a random choice from the neighbours that are better than the current state. The textbook suggests that the probability of choosing a state should depend on how much better the state is: better states have higher probability of being chosen. But that's a tricky bit of code, so just choose any one of the better neighbours, with equal probability.

Implement the search strategy, and demonstrate that your implementation works, by applying it to the Targeted Coding problem, maybe a few of the simple examples.

What to Hand In

This assignment is broken into pieces, and the last question asks for all the source code, so you don't have to hand in multiple copies. But pay attention to these requirements:

- A file named `a2q5.txt` containing a demonstration of the output of your program, which is copy/paste from a console, or output file. You only need to show a few examples. If this file is missing, the marker will assume your implementation is incomplete or not working.
- A file named `a2q5.LANG` containing your implementation of the search algorithm. Use the file extension appropriate for your choice of programming language, e.g., `.py` or `.java`. **For this question, only submit the implementation of the search algorithm, not the problem state or problem class implementations.** The markers will not attempt to run the code you submit.

Be sure to include your name, NSID, student number, and course number at the top of all documents.

Evaluation

- 6 marks: Your implementation of Stochastic Hill-climbing Search is correct, and well-documented.

Question 6 (20 points):

Purpose: To measure the quality of the search algorithms in terms of error, and time, and make comparisons.

Degree of Difficulty: Moderate

Now that you've got your search algorithms working, we'll get some empirical results to demonstrate their relative performance. We want to assess the general quality of the solutions they provide, and the runtime costs. You should review the Overview section before reading this question.

In order to quantify the quality of a single script s for target t, we'll measure the relative error:

$$err(s,t) = \frac{f(s,t)}{t}$$

This makes use of the objective function for the problem, but the error function is not part of the problem; it's part of the analysis.

To evaluate the quality of a search algorithm, we apply the algorithm to a collection of examples, and measure the average quality. One measure of average quality is called *Root Mean Square Error*. It's calculated as follows:

$$\text{RMSE} = \sqrt{\frac{\sum_{i=1}^{N} (err(s_i,t_i))^2}{N}}$$

where t_i is the target for the ith example problem, s_i is a solution to the ith example problem, and N is the number of example problems. You'll recognize the err function from above. You're basically calculating the squared error for each example in the file, adding it all up, dividing to get an average square error, and then the square root. Using the square of the error emphasizes large errors, and prevents negative errors and positive errors from cancelling each other.

On Moodle, you'll find two data files for this question, containing a number of problem instances, one per line. The target value T is the first value on the line, a floating point value, and then a sequence of integers, which is the list L. Run your search algorithms on all the problems in both files, and produce the following table (one for each file):

Strategy	RMSE	Ave Time
Random Guessing		
Random Search		
Hill-climbing		
Stochastic Hill-climbing		
Random-Restart Hill-climbing (50×20)		
Random-Restart Hill-climbing (10×100)		

Limit your search to 1000 steps. In the case of Random-Restart, try two variations:

1. 50 restarts; limit each hill-climbing attempt to 20 steps $(50 \times 20 = 1000)$.
2. 10 restarts; limit each hill-climbing attempt to 100 steps $(10 \times 100 = 1000)$.

Answer the following questions about your results:

1. Which algorithm had the lowest RMSE?
2. Did random guessing work well? Explain why (or why not)?
3. Compare the two variations of Random-restart Hill-climbing. Is it better to have lots of short runs, or fewer long runs?
4. Do you think the RMSE would get bigger or smaller as the length of L increases?

What to Hand In

- A file named `a2q6_EXECUTION.txt` containing brief instructions for compiling and/or running your code. See page 1.
- A file named `a2q6.txt` containing the tables required, and your answers to the questions above. If this file is missing, the marker will assume your implementation is incomplete or not working.
- A file named `a2q6.LANG` containing your implementation of the search algorithms. Use the file extension appropriate for your choice of programming language, e.g., `.py` or `.java`. You may submit multiple files, provided that the filenames begin with `a2q6_`

Be sure to include your name, NSID, student number, and course number at the top of all documents.

Evaluation

- 8 marks: Your two tables are plausible results from executing the search algorithms.
- 12 marks: Your answers to the questions demonstrate understanding of the issues and concepts.

Extra work for the ambitious

1. Implement the Simulated Annealing search strategy, and demonstrate that your implementation works. Add a row to the table from Question 6 giving your results.
2. Implement the Genetic Algorithm search strategy, and demonstrate that your implementation works. Add a row to the table from Question 6 giving your results.