Artificial Intelligence Search

Assignment

This is the assignment for the sub-module *Artificial Intelligence Search* of the module *Software Methodologies*. The hand-out date is Monday 14th October 2019 and it is to be completed and handed in **via DUO** by **2 p.m.** on **Friday 17th January 2020**.

***** It is really important that you read this document *****
***** thoroughly before you start. I'm sorry to appear *****
***** so dramatic with my bold-italic-red script but it *****
***** really is important that you follow the guidelines. *****

Overview

You are to implement ***two different algorithms*** (call them Algorithm A and Algorithm B) in ***Python*** using techniques studied during the lectures, but possibly also other algorithms you have devised or discovered for yourself, to solve the *Travelling Salesman Problem* (*TSP*). Your implementations should seek to obtain the best TSP tours that you can, given 10 collections of cities and their distances that I will supply to you. You will need to hand in the following items:

- between 2 ***and*** 4 ***correct Python programs*** comprising of: at least a basic implementation of each of your chosen algorithms (these are the files necessarily named `AlgAbasic.py` and `AlgBbasic.py`); and possibly an enhanced implementation of each of your chosen algorithms (these are the files necessarily named `AlgAenhanced.py` and `AlgBenhanced.py`)
- for each of the two algorithms that you implement, 10 ***tours***, one for each city-set that I give you, detailing the best tours you have found with any implementation of that algorithm (moreover, each tour needs to be in a specifically named and formatted text file as dictated in Section 4; so this amounts to 20 tour files)
- a ***one-page proforma*** (a pdf document) briefly describing the algorithms you implemented and any enhancements you have made.

The content is as follows: the mark scheme is in Section 1; the FAQs in Section 2; the algorithm tariffs in Section 3; how your tour files should be formatted in Section 4; and finally some hints and tips as regards the core algorithms in Section 5.

1 Mark scheme

The mark scheme is complicated so ***please read it carefully*** as it will determine which algorithms you ultimately choose to implement and help you plan your time. A list of FAQs follows in Section 2.

Marks will be awarded for:

(a) the *sophistication* of the algorithms that you implement

(b) the *correctness* of your basic implementations (`AlgAbasic.py` and `AlgBbasic.py`)

(c) any *enhancements* you have made so as to obtain enhanced implementations of your basic implementations (`AlgAenhanced.py` and `AlgBenhanced.py`)

(d) the *quality* of the tours that you obtain.

I will measure 'sophistication' and 'correctness' as follows.

- 'Sophistication': Each algorithm has a tariff associated with it (see Section 3) where this tariff gives the maximum number of marks that you can secure with your basic implementation of the 'vanilla' version of that particular algorithm (by 'basic' I mean the standard version covered in lectures). The tariff is such that the more technically complex the algorithm, the more marks you can secure.

- 'Correctness': I will run your basic implementations (`AlgAbasic.py` and `AlgBbasic.py`) on 2 secret sets of cities that I will not divulge to you beforehand. One city-set will have 48 cities and the other city-set will have 90 cities.

Your basic implementations (`AlgAbasic.py` and `AlgBbasic.py`) need to be correct.

- By 'correct' I mean that when I run your basic implementations on my secret city-sets, a legal tour is produced for both city-sets.

However, ... your 'sophistication' mark will be the *maximum tariff* from your 'correct' basic implementations (I'll say why in Section 2). Your 'correctness' mark will only be earned if *both* of your basic implementations are correct (I'll say why in Section 2).

- So, if you have only one 'correct' basic implementation then your 'sophistication' mark will be the tariff corresponding to the 'correct' basic implementation but you will not be awarded a 'correctness' mark. (Of course, if neither of your basic implementations is 'correct' then you will not be awarded either a 'sophistication' mark or a 'correctness' mark).

In addition, you can pick up extra 'enhancement' marks by enhancing and experimenting with your basic implementations to obtain two enhanced implementations (`AlgAenhanced.py` and `AlgBenhanced.py`).

- For example, you might try different versions of crossover for a genetic algorithm and this experimentation might secure extra bonus marks (at my discretion).

It is up to you as to whether you wish to try and enhance your basic implementations (though once you have your basic implementations, it is not particularly time-consuming to do a little bit of experimentation). Ideally, you should hand in 2 basic implementations (`AlgAbasic.py` and `AlgBbasic.py`) of two distinct TSP algorithms *as well as* 2 enhanced implementations (`AlgAenhanced.py` and `AlgBenhanced.py`) of *the same* algorithms.

- Note that you are *not allowed* to implement 4 different TSP algorithms!

You will be awarded an 'enhancement' mark for each 'correct' enhanced implementation (with 'correct' defined as above) but these marks will depend upon how innovative I feel your enhancements are.

You will also receive a 'basic quality' mark corresponding to how good the tours are that you have found. For each of the 10 given city-sets, you should return: the best tour you have produced by *any* implementation of your first algorithm (Algorithm A), enhanced or otherwise; and the best tour you have produced by *any* implementation of your second algorithm (Algorithm B), enhanced or otherwise. However, ... the 'basic quality' mark will be determined only by the *best tour* you have found for each of the 10 city-sets (regardless whether this is via an implementation of Algorithm A or of Algorithm B; I'll say why in Section 2).

You could also receive an 'enhanced quality' mark. For each algorithm, I will run your basic implementation and your enhanced implementation on my secret city-sets and depending upon how well your enhanced implementation does in comparison with your basic implementation, I will award extra marks (at my discretion); so, the 'enhanced quality' mark is derived from both of your enhanced implementations.

- When I compare a basic implementation with an enhanced implementation, please ensure that the parameters are identically set, e.g., if your Algorithm A is a genetic algorithm then you should ensure that the number of iterations, mutation probability, size of pupulation, and so on, are identically set in `AlgAbasic.py` and `AlgAenhanced.py` (failure to do so could mean you lose 'enhancement quality' marks).

The possible marks for each category follow:

- 'sophistication': the maximum obtainable is 10 but this will depend upon the tariff of the algorithms implemented

- 'correctness': if both basic implementations are 'correct' then a mark of 4 is awarded, otherwise a mark of 0
- 'enhancement': there is a maximum of 3 marks available for each enhanced implementation; so, the overall 'enhancement' mark is at most 6
- 'quality': the maximum 'basic quality' mark available is 8 and the maximum 'enhanced quality' mark available is 2; so, the overall 'quality' mark is at most 10.

Consequently, the maximum mark achievable is 30.

2 FAQs

- Student: *Why do you insist on Python*?
- Iain: Every student has completed Computational Thinking and so can program in Python; this yields a 'level playing field' when it comes to implementation. Also, restricting to Python-only leads to fairer marking.
- Student: *Why do you take the maximum tariff from two 'correct' implementations as the 'sophistication' mark*?
- Iain: On occasion, a more sophisticated algorithm may give worse results than a more elementary algorithm. I want to reward both the quality of the tours that you find and also your accomplishments in coding up a more difficult algorithm. With things set up as above, I encourage you to code up a more sophisticated algorithm without harming your chances of getting good tours.
- *So I guess that's why you take the best overall tour found when you give the 'basic quality' mark*? *So that we don't harm the quality of the tours we find if we choose to implement a more sophisticated algorithm*?
- Iain: Correct!
- Student: *And I suppose you ask us to implement two algorithms and to experiment so that you can assess both our understanding of the algorithms in the sub-module, through our capacity to code them up, along with our ingenuity and deeper understanding in obtaining good tours*?
- Iain: Correct again! If all I asked you to do was to produce basic implementations of two algorithms and produce some half-decent tours

then there wouldn't be much of a challenge (nor fun) in that. So, I would like you to experiment. I understand that some of you will have more time and inclination for this than others so I ensure that if all you do is produce basic implementations and some half-decent tours then this will suffice to pass the coursework, which I think is fair. But I'm also giving you the opportunity to show me what you can do and be rewarded for it. In the past, many students have enjoyed the challenge of producing good tours.

- Student: *Why do you only give us a 'correctness' mark if both basic implementations give tours on your secret city-sets*?

- Iain: I think it is reasonable that your two basic implementations should both be correct. Don't you agree? My definition of 'correctness' is not exactly challenging!

- Student: *Why do you not just ask us to return the best tour we have found by whatever algorithm for each of the* 10 *city-sets rather than one tour for each algorithm*?

- Iain: I get to see the profile of tours you have produced for each algorithm. This allows me to have confidence that the tours you have supplied to me are indeed produced by the implementations stated, as I know (roughly) how different algorithms should perform. Also, I will run your basic and enhanced implementations on the 10 city-sets to provide me with a sanity check that your codes are what you say they are.

- Student: *What if we have worked hard to fine-tune our implementations to perform really well on the* 10 *city-sets but when you run them on your secret city-sets, they don't do so well*?

- Iain: There is a small chance that this might happen but if your enhanced implementation improves things across many of the 10 city-sets, it is likely that it will improve things on my secret city-sets too. Also, the 'enhanced quality' mark is relatively small.

- Student: *Can you give me some illustrations of the different mark awards depending upon what sort of a student I am*?

- Iain: OK; here are some illustrations.

  - Student A: Maybe you are hard-pushed and will not have the time nor inclination for experimenting but correctly implement a reasonably sophisticated algorithm at tariff 8 and a less complicated algorithm at tariff 6; so, your 'sophistication' mark will be

8 and your 'correctness' mark will be 4. The lack of experimentation means: that you get an 'enhancement' mark of 0; and that you only obtained moderately good tours overall. Perhaps you get a 'basic quality' mark of 5 and so an overall 'quality' mark of 5. You would score 17/30 = 57% (a very solid lower-second mark).

- Student B: Maybe you are struggling and cannot correctly implement two algorithms but get a basic tariff-6 TSP algorithm working, though the tours produced are not very good, resulting in a 'quality' mark of 4. You would receive a 'sophistication' mark of 6, a 'correctness' mark of 0 and an 'enhancement' mark of 0. Your total mark would be 10/30 = 34% (a fail mark).
- Student C: maybe you are like Student A but you are inclined to experiment. Your enhanced impementations are correct and moderately innovative; so you obtain an 'enhancement' mark of 3. The tours of the secret city-sets produced by your enhanced implementations produce a modest improvement over those produced by your basic implementations and you obtain an 'enhanced quality' mark of 1. So, you would score 21/30 = 70% (a first-class mark).

The moral of the story? Show ambition with your implementations and experiment.

- Student: *Is the assignment the same as last year*?
- Iain: No, it has changed. First, the city-sets are different to last year; but more importantly you are being asked to do less work. Last year, students felt that the amount of work required did not match the credit available. This year, students no longer need to supply a 4-page report and I have also supplied to you all of the Python code (`skeleton_code.py`) required for reading and writing data files (you should use this code as directed; it is well commented). I implemented a basic greedy algorithm and a genetic algorithm in an afternoon ... and I can assure you that your programming skills are way better than mine!

3 Algorithm tariff

Brute-force search: As you are doubtless aware, a brute-force search for the TSP will only work for very, very small sets of cities; of course, when it works, the tour obtained is optimal. (Note that it is not combinatorially straightforward to generate all possible tours for checking.) Tariff: 6/10

Basic greedy algorithm: The most obvious greedy algorithm is that obtained by starting at some city and iteratively moving to the nearest city to the last city visited. This is trivial to implement. Tariff: 5/10

Best-first search without heuristic data: Depending on how you choose your evaluation function, you can implement a breadth-first search, a depth-first search and various other algorithms. However, you should bear in mind that a breadth-first search is demanding on memory and a depth-first search runs the risk of not terminating (though this will depend upon how you define your search problem). Of course, you can choose to refine a best-first search by forcing termination under given circumstances. Tariff: 7/10

Iterative deepening: One way of getting round the memory requirements of a breadth-first search is to undertake repeated depth-first searches with bounds on the depth. Tariff: 8/10

Best-first search with heuristic data: You'll have to build your own heuristic functions as none are supplied. Tariff: 9/10

A* search: You'll have to build your own heuristic functions as none are supplied. Also, note that an A* search with a 'good' heuristic is optimal and also that the TSP is **NP**-hard; so, if you don't have a heuristic that gives a good estimation then it is unlikely that you'll get good tours. Of course, you can choose to terminate an A* search under given user-defined circumstances. Tariff: 10/10

Hill-climbing search: This is very easy to implement once you decide upon the representation of the TSP as a search problem. Tariff: 6/10

Simulated annealing: This is moderately easy to implement once you decide upon the representation of TSP as a search problem. Tariff: 7/10

Genetic algorithm: There are many ways in which you can represent the TSP using a genetic algorithm. Tariff: 8/10

Some of the above algorithms lend themselves to experimentation more than others so you might bear this is mind when making your choices.

In the past, many students have implemented algorithms they have found themselves from books and the general literature, such as *Ant Colony Optimisation*, *Recursive Best-First Search* and *Beam Search*. There are lots of nature-inspired algorithms to choose from with new ones regularly being devised. If you wish to implement an algorithm not covered in the course (and I welcome this) then please ***e-mail me beforehand*** and I will supply you with the tariff.

One final thing: I will be executing your implementations automatically and it is possible that you choose parameters so that your implementations (on my secret city-sets) take a long time to execute. When you hand your implementations in, you should ensure that your implementations will take no more than ***two minutes*** on my secret city-sets (of sizes 48 and 90). Typical parameters that you will need to adjust are, for example, the

number of iterations in a genetic algorithm, the temperature function in simulated annealing, and so on. ***If an implementation of yours takes too long then it will be killed and you will run the risk of losing a significant number of marks***.

4 The format of your submission

All submissions should be named using your username as follows. I illustrate using the dummy username abcd12 but you should substitute your own. ***If you don't follow the rules below then you run the risk of getting no marks!***

All files should be in a folder called abcd12. Within this folder there should be:

- 4 Python programs (.py) entitled `AlgAbasic.py`, `AlgBbasic.py`, `AlgAenhanced.py` and `AlgBenhanced.py` which are the basic versions of your two chosen algorithms, Algorithm A and Algorithm B, along with the enhanced versions
- 10 tour-files (.txt) entitled `AlgA012.txt`, `AlgA017.txt`, `AlgA021.txt`, `AlgA026.txt`, `AlgA042.txt`, `AlgA048.txt`, `AlgA058.txt`, `AlgA175.txt`, `AlgA180.txt` and `AlgA535.txt`, with each tour-file containing the best tour you have found using your first chosen algorithm, Algorithm A, on the corresponding city-set
- 10 tour-files (.txt) entitled `AlgB012.txt`, `AlgB017.txt`, `AlgB021.txt`, `AlgB026.txt`, `AlgB042.txt`, `AlgB048.txt`, `AlgB058.txt`, `AlgB175.txt`, `AlgB180.txt` and `AlgB535.txt`, with each tour-file containing the best tour you have found using your first chosen algorithm, Algorithm B, on the corresponding city-set
- one proforma (.pdf) entitled `AISearchProforma.pdf` (I give you the template in Word but please save it and return it in the form of a pdf).

You need to ensure that ***all of your Python implementations can be executed in command-line mode by supplying the city-file, e.g., via the command***

```
python AlgAbasic.py AISearchfile012.txt
```

If I cannot execute your implementations as above then I will not be able to run your codes and you will lose marks. Using `skeleton_code.py` will enable such a command-line execution (and if the input file is omitted then there is a default input file embedded in the code; take a look).

Also, when I run one of your implementations, in your folder abcd12, via a command-line instruction such as

- `python AlgAbasic.py AISearchfile012.txt`

the actual input file `AISearchfile012.txt` is assumed to reside in a folder named city-files that is in the same folder as abcd12. So, the folders city-files, abcd12, xyzp32, gdhn74, ... will all reside in the same folder.

5 The data-files

The actual instances of the Travelling Salesman problem (more precisely, the symmetric Travelling Salesman problem, where the distance from city x to city y, denoted (x, y), is always the same as the distance from city y to city x, that is, (y, x)) will be given in the following form (the cities are always named $1, 2, \ldots, n$).

`NAME = <string =` name-of-the-data-file`>`,
`SIZE = <integer` n `=` the number of cities in the instance`>`,
`<list-of-integers` $d_1, d_2, d_3, \ldots, d_m$`>` where the list consists of
the distances between cities $(1, 2), (1, 3), \ldots, (1, n)$,
then the distances between cities $(2, 3), (2, 4), \ldots, (2, n)$,
...
and finally the distance between the cities $(n - 1, n)$.

Commas ',' are used as delimitters in data-files; carriage returns, end-of-line markers, spaces, etc., should be ignored.

So, for example, the instance with 5 cities where: the city 1 is at the origin; the city 2 is 3 miles north; the city 3 is 4 miles east; the city 4 is 3 miles south; and the city 5 is 4 miles west, and all distances are the Euclidean distances between cities, is encoded as the city-file `AISearchsample.txt`:

```
NAME = AISearchsample,
SIZE = 5,
3, 4, 3, 4,
5, 6, 5,
5, 8,
5
```

With reference to the remark above, re: delimitters, the above city-file could well be presented with no carriage returns or spaces, etc., as simply

```
NAME=AISearchsample,SIZE=5,3,4,3,4,5,6,5,5,8,5
```

Note that in general a given instance need not be based on the Euclidean distances of a collection of cities on the plane. You should assume that a given distance between two cities is a non-negative integer (and so it could well be the case that the distance between two distinct cities is 0).

As stated earlier, I supply you with the (well commented) skeleton of a Python program called `skeleton_code.py`***. The code reads a given input file (in the format above) and supplies the number of cities (*** `num_cities` ***in*** `skeleton_code.py`***) and a two-dimensional symetric array (*** `distance_matrix` ***in*** `skeleton_code.py`***) containing the distances between cities. You should use*** `skeleton_code.py`***.***

The data-files that you will have to execute your code on will be called `AISearchfile012.txt`, `AISearchfile017.txt`, `AISearchfile021.txt`, `AISearchfile026.txt`, `AISearchfile042.txt`, `AISearchfile048.txt`, `AISearchfile058.txt`, `AISearchfile175.txt`, `AISearchfile180.txt` and `AISearchfile535.txt`. The numeric digits denote the number of cities in the particular instance.

When you have computed a legitimate tour of some set of cities, there is code in `skeleton_code.py` ***that checks that this tour is legitimate and writes the tour to an output-file in the correct format. You should supply your tours to me in the file obtained by using this code (after renaming it).***

Some remarks and hints

As mentioned earlier, the following methods are available to you (as studied in the course):

- brute-force search
- basic greedy algorithm ('nearest-neighbour')
- best-first search without heuristic data
- greedy best-first search
- A* search
- hill-climbing search
- simulated annealing
- genetic algorithm.

There is a little bit of work to do as regards the implementation of each method and here is a little bit of help.

Brute-force search

Here is a hint as to how all tours in an n-city Travelling Salesman instance can be generated. Each tour is stored in a 1-dimensional list T of size n, where the elements are all distinct and come from $\{1, 2, \ldots, n\}$. The tour stored in T is $T[1], T[2], \ldots, T[n], T[1]$. In fact, we can similarly store a tour of the $m \leq n$ cities $\{1, 2, \ldots, m\}$ in $T[1], T[2], \ldots, T[m]$, with $T[m+1] = T[m+2] = \ldots = T[n] = 0$.

Our procedure `gen(`T`,`m`)` takes as input a tour of m cities, $x_1, x_2, \ldots, x_m, x_1$, say, held in the list T (as above), and proceeds as follows.

- If $m = n$ then we compare the length of the tour T (of n cities) with the length of the shortest tour found so far and if T is shorter then we remember T and its length.
- If $m < n$ then `gen(`T`,`m`)` generates all tours of the cities $\{1, 2, \ldots, m, m+1\}$ by inserting the value $m+1$ in location 1, then location 2, ..., then location m, then location $m+1$ of T (so that the cities coming after $m+1$ are 'shifted' along the array). Interleaved with generating each tour, which we refer to as T', we recursively call the procedure `gen(`T'`,`$m+1$`)`.

In more detail, `gen(`T`,`m`)` is as follows.

```
if m == n then
   calculate the length of the tour T and if it is shorter
   than the best tour found so far, remember T and its
   length as the best tour found so far
else
   for i = 1 to m+1 do
      T' = T with the city m+1 inserted into location i
      call gen(T',m+1)
      T' = T with the city m+1 removed from location i
fi
```

Thus, the following pseudo-code generates and tests all possible tours, given the distance-file of n cities.

```
T = [1,0,0,...,0]      %initialize tour T as [1]
m = 1                  %initialize number of cities of tour T as 1
call gen(T,m)
output the shortest tour found
```

Although I don't recommend that you implement a brute-force search, brute-force searches are good for checking optimal values in small cases;

also generating all combinatorial possibilities comes up regularly and it is useful to know how to do this. If you do implement a brute-force search then you can always choose to kill an execution and take the best tour you have found up until that point as a guide.

Best-first, greedy best-first and A* search

The Travelling Salesman Problem needs to be realised as a search problem. One way of doing this is to have the set of all lists of distinct cities as the states together with the lists of n distinct cities augmented with the start city (and so a state is a list of between 0 and n cities, or a list of $n+1$ cities where the first n are distinct and the last city equals the first). There is one action with a state t' being a successor of a state t if the list t' is the partial tour t extended with one new city (not appearing in t), or if t has length n and t' is t augmented with the first city of t. The step-cost associated with any transition from state t to state t' is the cost of moving from the final city in the list t to the final city of the list t'. The initial state is the list t_0 consisting of just the start city and a goal state is a list of $n+1$ cities. An optimal solution is thus a path from the initial state to a goal state of minimal cost.

There are a number of heuristic functions available for the Travelling Salesman Problem. One of these is the heuristic h where, given a state $t = x_1, x_2, \ldots, x_r$, $h(t)$ is defined to be the minimal step-cost of moving to a state of the form $t' = x_1, x_2, \ldots, x_r, x_{r+1}$ (that is, always move to the nearest legal city from where you are; if there is no city to move to then $h(t) = 0$).

Another heuristic is as follows. Given some state t, your heuristic function $h(t)$ is the sum of the distance of the closest city c that has not been visited to the last city of the partial tour t plus the distance of any other unvisited city (different from c) to the start city (if there aren't enough unvisited cities to apply this rule then the heuristic value is 0). This heuristic reflects that you want your next city to be visited to be close to the current city but that you don't want to be left with a city that is a long way from the start city.

Yet another heuristic h is as follows. Given a state $t = x_1, x_2, \ldots, x_m$, $h(t)$ is defined to be the minimal step-cost of moving to a state t', where t' is t with some new city inserted somewhere within t, e.g., if $t = 1, 5, 4, 7$ then t' might be $1, 5, 6, 4, 7$. If this heuristic is to be used then the transition function (above) needs to be amended to allow such transitions.

Bear in mind that A* search gives optimal solutions (under mild circumstances) and so unless you have a brilliant heuristic function (which is unlikely) then this method will only work on small instances. I have no idea how the above heuristics will pan out on the instances given!

Hill-climbing search and simulated annealing

Here, the states might be the set of all possible tours of n cities, and one state t' might be a successor of another state t if a swap of the positions of two (or more!) of the cities in the tour t results in the tour t'. The heuristic cost function of a state might be the length of the tour. There are numerous other definitions of a successor function.

Genetic algorithms

In order to formulate the Travelling Salesman Problem for solution by a genetic algorithm, we need to define our population. One way of doing this is to define the population as a set of tours of n cities, represented as strings of length n. The fitness of a member of the population might be just the length of the tour. We now need to come up with a notion of mutation and crossover. One way of defining a mutation is just to randomly swap the positions of two cities within a tour (though there are many others). Defining a notion of crossover is more difficult. However, given two tours $t = x_1, x_2, \ldots, x_n$ and $t' = x'_1, x'_2, \ldots, x'_n$, we could define a new tour as follows.

- Randomly choose some $i \in \{1, 2, \ldots, n-1\}$ and form the strings

  $$s = x_1, x_2, \ldots, x_i, x'_{i+1}, x'_{i+2}, \ldots, x'_n$$

  and

  $$s' = x'_1, x'_2, \ldots, x'_i, x_{i+1}, x_{i+2}, \ldots, x_n$$

  (note that these might not be tours as some cities might be missing and some repeated).

- Scan through s and make a list of the cities not appearing in s and a list of the locations containing repeated cities (these lists have the same length). Replace every repetition with a missing city (according to some user-defined strategy). Do the same for s'.

- Hence, we obtain two tours s and s', and we take the crossover as the shortest one.

For those really interested, there is a paper:

- P. Larranaga, C.M.H. Kuijpers, R.H. Murga, I. Inza and S. Dizdarevic, Genetic Algorithms for the Travelling Salesman Problem: A Review of Representations and Operators, *Artificial Intelligence Review* **13** (1999) 129–170

that discusses genetic algorithms for the TSP.

Please note: the above hints are just suggestions and you might care to come up with your own ideas. Also, it will be up to you to (experimentally) vary parameters (e.g., the different probabilities in a genetic algorithm or a simulated annealing algorithm) to improve your solutions.