Software Development: Refactoring and Reflection

B007439

March 30, 2015

Contents

1 Introduction

This report describes the work performed in the Refactoring and Reflection part of the Software Development coursework. The goal of the project was to refactor an existing software in order to improve its usability and maintainability. The software in question is a travelling salesman problem solver, which uses an ant colony optimization algorithm. The code is available at `https://github.com/nethopper/travelling_salesman_refactoring`.

A number of problems with the code were identified and discussed in the Planning and Risks stage of the project. These included:

- Lack of a test suite
- Lack of useful comments
- Unclear names
- Lack of separation of behaviour
- Hardcoded configuration values
- Single, opaque and insecure input and output format
- Extraneous print statements

The current phase consisted of implementing solutions to the above problems. Due to a limited schedule, not all problems could be solved. Therefore, the author has prioritized the fixes in terms of impact and difficulty and carried out as many as was possible given the time available. The software was refactored iteratively, ensuring that a working product was available at each stage. This was done in order to be able to deliver a working solution regardless of when the time budget ran out. This method did mean that some changes were overwritten by later ones, however that drawback was offset by the greatly reduced risk of having to submit broken, work-in-progress code.

2 Changes

This section lists the changes made to the code and its environment in the course of the work. The improvements listed here largely correspond to the work items identified in the initial stage, however some changes have been made. These are described in Section 3. The only improvements that could not be made due to lack of time were adding JSON as an I/O format and enabling pluggable updating rules and stopping criterion.

2.1 Setting up virtualenv

The initial step was to create a self-contained, controlled development environment. This was performed using *virtualenv*, a Python tool which ensures that the project uses its own, separate Python runtime and package directory. This makes the development environment easy to reproduce on different machines, ensuring that the software's dependencies are explicitly known and do not interfere with other projects on the same machine.

2.2 Commenting code

In order to understand the initial state of the software, the author first added comments explaining the functioning of the algorithm as well as the meaning of classes and variables used. This entailed reading through the source code, as well as running the program and observing outputs, noting any information as comments in the relevant sections of the code.

2.3 Adding integration tests

Before any changes to the code were made, there was a need to codify the current behaviour of the program, specifically the solutions it produced for a given set of inputs. Being able to run an automated test after each change was very valuable, as it was a quick and reliable way of finding bugs introduced in the changes. In order to achieve this ability, integration tests were created. These tested the software in an end-to-end fashion, by providing a set input and examining the output. The integration tests proved very useful during later development, as they have notified the programmer of errors multiple times, saving a lot of manual debugging time.

2.4 Renaming classes and variables and restructuring code

The names of many variables were either cryptic or confusing (e.g. 'stuff'), harming readability and making the code difficult to understand, especially for newcomers. In order to make the meaning of the code more self-evident, variables, functions and classes were renamed where appropriate. A small sample of the changes include renaming:

- 'BigGroup' to 'Colony'
- 'Work' to 'Ant'
- 'ntv' to 'nodes_to_visit'
- 'bpc' to 'best_path_cost'

Additionally, classes were moved out to their own files (e.g. Ant to ant.py). This made it easier to find information required, because scrolling was minimized and identifying locations of classes was made straightforward.

2.5 Using logging instead of print output

The software produced a lot of debugging and informational output, e.g. outputting the resulting path, as well as whether 'exploration' was chosen by an ant. The output was unnecessary in most cases, as it provided low-level details of the algorithm, which only served as clutter during normal use. Removing it, however, would not be a good idea, because it could prove useful to some people, such as those looking for bugs in the solver or trying to understand why they are getting specific output. In order to get the best of both worlds, the print statements where replaced with code using the *logging* package. This enabled the output to be suppressed during normal operation, but enabled again when needed. It also meant that the messages could be separated into distinct classes and turned on in groups, rather than all at once. Output describing progress and the best path found was given the level *INFO*, while the more low-level logs, were given the *DEBUG* level. Thanks to this, the user can choose whether to output nothing, only progress and path information, or debugging messages as well.

2.6 Removing cyclic dependency between Ant and Colony

The Planning and Risks phase described the issue of a cyclic dependency between the classes Work (now called Ant) and BigGroup (now Colony). This unnecessarily coupled the two classes together, meaning, for example, that if the Colony class were to change its interface, the Ant class would have to be changed as well. After the fix, the Colony class simply asked the Ant objects to perform their work and updated itself with the results obtained from them. This makes it easier to follow the logic, as one does not need to remember the states of both classes or jump between their code. The coupling of the classes was decreased, while their cohesion increased and it enabled ants to be used by classes other than Colony, if necessary.

2.7 Extracting out methods to better adhere to the Single Level of Abstraction principle

The Single Level of Abstraction principle says that all statements in a method should be at the same level of abstraction. Adherence to this principle results in code that is easier to understand. In the case of this project, when methods were extracted out the code described its function more concisely and read more closely to natural language.

The listings 1 and 2 depict code before and after method extraction. The code could potentially still be improved, however it is already easier to understand the general idea of the loop and the path extension logic is kept together under a name that describes its function.

```python
while not self.end():
    new_node = self.state_transition_rule(self.current_node)
    self.path_cost += self.graph.distance(self.current_node, new_node)
    self.path.append(new_node)
    self.path_matrix[self.current_node][new_node] = 1
    self.local_updating_rule(self.current_node, new_node)
    self.current_node = new_node
self.path_cost += self.graph.distance(self.path[-1], self.path[0])
```

Listing 1: Before method extraction

```python
while not self.end():
    self.extend_path()
self.add_cost_of_return()

def add_node_to_path(self, node):
    self.path_cost += self.graph.distance(self.current_node, node)
    self.path.append(node)
    self.path_matrix[self.current_node][node] = 1

def extend_path(self):
    new_node = self.state_transition_rule(self.current_node)
    self.add_node_to_path(new_node)
    self.local_updating_rule(self.current_node, new_node)
    self.current_node = new_node

def add_cost_of_return(self):
    self.path_cost += self.graph.distance(self.path[-1], self.path[0])
```

Listing 2: After method extraction

2.8 Implementing command-line arguments and CSV I/O

Previously the user did not have any control over the functioning of the algorithm, unless they were able to find and change them in the source code. This has been improved by implementing command-line arguments in the software. They enable the user to specify the files and formats of input and output as well as the parameters of the algorithm, such as *number of ants*, *alpha* or *beta*. This makes it a lot easier to experiment with the program by trying out different combinations of parameters and examining the results.

The input and output formats can also be specified on the command-line. This is useful, as CSV was added for both input and output. The software will also attempt to guess the format by looking at the extension of the input file. The users does not therefore usually need to specify the input format. In addition to files, standard input and output have been added for reading from and writing to. This feature enables users to compose the program with others through Unix pipes.

2.9 Transforming code to use less implicit state

Classes used in the code were data containers not restricting access to the data. Because the methods were all used in the context of the class, they implicitly shared a lot of state, which made it difficult to understand what data a function was operating on at any given time. Additionally, they could not be used on data that did not belong to the class, as they accessed the class's fields, as opposed to their arguments. They were also difficult to test, because they require the entire class to be present in order to be used.

To simplify the code, the classes have been replaced by plain functions operating on data in standard structures, such as dictionaries and lists. This means that the data is no longer closed, but can be used with existing functions that manipulate these structures and the functions are available to be reused with other data, outside class boundaries. These functions are much simpler to reason about and test, as they only operate on their own arguments and do not expect any context apart from that.

2.10 Implementing unit tests

As mentioned above, unit tests were created for the functions that make up the algorithm. For each function, the unit tests were written before the transformation in Subsection 2.9 took place. They made it a lot easier to refacor the code, simplify function, as well as perform the above transformations, thanks to the behaviour being encoded separately, beforehand.

3 Reflection

This section contrasts the expectations of the work set down in the Planning and Risks phase to how the work was actually carried out. The order of improvements was changed by moving implementation of command-line arguments earlier and unit testing later in the process. The former was decided on because it meant that more work could be done before setting out to refactor the architecture, which would be a long and risky process. It was also sensible to incorporate the command-line arguments with the input and output formats and settings, as they were connected to one another.

Unit tests were delayed, as there was great benefit to implementing them together with the architecture transformation. Before that they would be difficult to implement and would have to be rewritten anyway. On the other hand, when written right before their corresponding functions, they were simpler and still provided the same benefits.

The project did not include any major unexpected events that would greatly disturb the plan. However, a problem was encountered with implicit state sharing in ants. There was a plan initially to use immutable, persistent data structure for the graph and ants, which was based on an assumption that they operate independently. This would enable the ants to be trivially parallelised without worrying about race conditions. However, after implementing the graph, the integration tests failed. As it turns out this was because the assumption was false and the ants have in fact relied on sharing a single graph which they mutate in-place. The persistent data structure work had therefore be undone. Apart from lost time, a problem with this situation is the fact that this very implicit state sharing is only encoded in comments and is not visible to the programmer. This will decrease its maintainability, as more time will have to be spent debugging because of invisible connections between pieces of code.

4 Future enhancements

A number of enhancements could be made to the code in order to improve its usability. One example is the implementation of new I/O formats, such as JSON, which would make it easier to compose the software with other programs. Groundwork for this has already been laid, so that for any new format the reading/writing functions only need to be added to the register of available formats.

The software could be extended to be used as a library by other software. This is already possible, but the API should be improved, as it was not designed with this use in mind. The library could allow users to specify their own updating rules or stopping criterions, giving them more control over the functioning of the algorithm.

5 Conclusions

The refactoring has improved the usability and maintainability of the software. Usability was improved by giving the user more control over the algorithm through command-line arguments, as well as through making the software more composable thanks to different input and output means and formats. Maintainability was improved by making the code easier to read and reason about, as well as implementing a test suite, which not only specifies the expected behaviour of the functions, but may also alert the programmer when they have introduced an error into the code. The software could be further improved with new input and output formats, increasing its flexibility. It could also be extended to be a library, giving users even more control over the algorithm.