Module 7

Ryo Kimura

7 Test Instances

In the next three modules, we will discuss the three major aspects of computational testing: choosing test instances, running experiments, and analyzing the result of experiments. In this module, we will discuss test instance selection in detail.

This module is based on Chapter 8.3 *Test Data Generation* of [2]; refer to the book for a more detailed discussion.

7.1 What are Good Test Instances?

To obtain valid and reliable experimental data about the computational performance of an algorithm, methodology, framework, etc., we need specific *test instances* with realistic parameter values. While the choice of test instances is inevitably influenced by time constraints, availability, convenience, etc., there are several considerations to keep in mind when choosing a set of test instances:

- **Quantity / Purpose**: The *quantity* and types of chosen test instances should always match the *purpose* of the experiment. For example, preliminary testing of an experimental algorithm may only require a few instances, while a systematic assessment of the strengths and weaknesses of a well-known heuristic requires a significantly larger number of instances. Similarly, an experiment for demonstrating the flexibility of a generic solution framework should use a wide variety of test instances, while an experiment for highlighting the practicality of an algorithm should use many realistic test instances.

  In practice, this means **clearly defining the purpose of the experiment** *before* deciding on a set of test instances to use.

- **Comparability / Reproducibility**: The results of algorithmic experiments should be *comparable* to other experiments, and the test instances should be given in enough detail that another researcher could, at least in principle, *reproduce* the results. This is important for the scientific integrity of the results, and aligns with the standards for rigorous empirical experimentation that is expected in other fields like the natural sciences.

  In practice, this means using **test instances that have been used in prior and related works**, instances from a standard test data library, and/or giving detailed explanations for **how exactly the test instances were generated**.

- **Portability**: Whenever possible, test instances should be *portable* and accessible to any researcher who wants to use them. This is closely related to the previous issue since lack of portability hinders reproducibility and comparability greatly.

  In practice, this means **avoiding proprietary instances if possible**, and storing **instance data in a format that is widely accepted** or easily understandable/parsable.

- **Measurability**: There should be a way to *measure* and quantify how good the algorithm performs on a specific instance in terms of the performance metrics of interest. Often there are other metrics besides the time and computational resources used that are important for assessing the performance of an algorithm; for example, for heuristic algorithms the *optimality gap* (i.e., how far the generated solution is from an optimal solution) achieved may be of interest.

  In practice, this means ensuring that the test instances used for the experiments are **capable of providing the performance metrics of interest** (e.g., we may not want to use instances for which the optimal value is not known if optimality gap is an important metric).

- **Variety / Significance / Unbiasedness**: The test instances should include a *variety* of instances in order to ensure the experiment produces *significant*, meaningful results. For example, only using instances that are easy for the algorithm to solve will heavily *bias* the results and conclusions, and they tell us nothing about the limits of the algorithm, cases where the algorithm fails, and the effect of different problem characteristics on algorithm performance.

  In practice, this means ensuring the set of test instances is **chosen in a relatively unbiased manner** (e.g., standard test data libraries, random generation) and includes problems with **different problem characteristics**.

7.2 Types of Test Instances

Broadly speaking, there are four types of test instances used in computational experiments: **real world instances**, **artificial instances**, and **perturbed instances**. We discuss each of these in terms of their advantages and disadvantages relative to the considerations discussed in the previous section.

7.2.1 Real World Instances

Real-world instances originate from real applications, and therefore reflect the ultimate *purpose* of any tested algorithm. Since the goal of a computational experiment is often to evaluate practical usefulness, real-world instances provide valuable assessments of the performance of the algorithm in practice.

On the other hand, some real-world instances are proprietary and not available for public use, hindering *comparability* and *portability*. In addition, it is often difficult or impossible to obtain a sufficient *quantity* of instances, and real-world instances often have properties that are intrinsically tied to practical concerns, resulting in a lack of *variety*.

7.2.2 Artificial Instances

Artificial instances are typically randomly generated by some systematic procedure according to a list of parameters. Often this means that it is very easy to accumulate a very large *quantity* of test instances, and depending on how well the generator program is written, can also produce a wide *variety* of instances. Furthermore, if the generation process is well-documented, they provide an effective means to ensure *reproducibility* and *comparability* of the results.

However, writing a good generator program can be quite difficult, and naive problem generation can produce instances that have *biased* characteristics that do not reflect reality. For example, the

	Advantages	*Disadvantages*
Real World	- representative of real world behavior (*purpose*) - allow assessment of practical usefulness	- only of bounded size (*variety*) - only few available (*quantity*) - sometimes proprietary (*comparability*) - lack of control of characteristics
Artificial	- arbitrary size available (*variety*) - arbitrary number available (*quantity*) - rarely proprietary (*comparability*) - ability to control characteristics	- lack of realism (*purpose*) - difficult to assess real world performance (*purpose*) - susceptible to uninteded correlations and biases (*unbiasedness*)
Perturbed	- better *quantity* than real world instances - more realistic than artificial instances	- *variety* comparable to real world instances - less realistic than real world instances - often hard to identify meaningful perturbation

Table 1: Comparison of Different Types of Test Instances

well-cited paper [1] experimentally shows that hard SAT instances are characterized by the ratio of clauses-to-variables, rather than their size. Similarly, [4] shows that when randomly generating instances of the asymmetric traveling salesman problem, if the number of distinct inter-city distances is fixed, the problems get *easier* as they get bigger,[1] While there is some work [3] that considers generic schemes of hard instance generation,[2] generating realistic instances remains difficut in general.

7.2.3 Perturbed Instances

Perturbed instances start with a real world instance and modify it in a controlled way using a generator program. They attempt to combine the advantages of both real world and artificial instances by enabling the generation of larger *quantities* of instances while maintaining *realistic* problem characteristics. This puts perturbed instances somewhere between real world instances and artificial instances in terms of realism.

However, it is often difficult to identify interesting parameters that have to be changed in order to obtain useful perturbed instances. Furthermore, the size and structure of the perturbed instances is often not much different from the original instance, which imposes a limit on the potential *variety* that is possible with this method.

In short, the different types of instances can be summarized in table 1.

[1] Intuitively, this is because the loewr bound obtained from the assignment relaxation gets tighter as the number of cities increases.

[2] The main idea here is to use PCA to generate an *instance space* that characterizes instance based on their difficulty, then use genetic algorithms to generate instances that "fill in " prominent gaps in the instance space.

7.3 Finding Real World Instances

The best sources of real world instances are from prior works in the literature that have considered similar problems. Many researchers are willing to share their test instances as long as it is not proprietary, or they use established sets of test instances which are referenced in their work and are often freely available. **Google Scholar** (https://scholar.google.com/) is especially helpful for finding potential test instances to use (and for conducting literature reviews more generally).

Many instances are available in various test data libraries:

- **OR-library** is a collection of test data sets for a wide variety of OR problems from corporate structuring to prize-collecting steiner tree to hybrid reentrant shop scheduling (http://people.brunel.ac.uk/~mastjjb/jeb/info.html).
- **MIPLIB** is a large collection of real world instances of mixed integer programs. It is updated every few years, as of April 2019 the most current version is MIPLIB2017 (https://miplib.zib.de/).
- **CSPLib** is a library of test problems for constraint solvers motivated by various areas including bioinformatics, Ramsey numbers, and ridesharing (http://www.csplib.org/).

Some problem specific libraries include:

- **TSPLIB** and University of Waterloo's **TSP Test Data** for traveling salesman problems (http://elib.zib.de/pub/mp-testdata/tsp/tsplib/tsplib.html, http://www.math.uwaterloo.ca/tsp/data/index.html)
- **PSPLIB** for project scheduling problems (http://www.om-db.wi.tum.de/psplib/)
- **SATLIB** for satisfiability problems (https://www.cs.ubc.ca/~hoos/SATLIB/benchm.html)
- **TPTP** for theorem provers (http://www.tptp.org/).

You may also get benchmark/test instances from challenge competitions:

- **DIMACS Implementation Challenge** (http://archive.dimacs.rutgers.edu/Challenges/)
- **MiniZinc Challenge** (https://www.minizinc.org/challenge.html),
- **SAT Challenge** (https://www.satcompetition.org/)

Finally, if what you need is raw data rather than specific instances, the following links for finding datasets can be helpful:

- **Awesome Public Datasets**: list of free publicly available datasets, categorized by topic (https://github.com/awesomedata/awesome-public-datasets)
- **Google Dataset Search**: like Google Scholar, but for datasets (https://toolbox.google.com/datasetsearch)
- **Machine Learning and AI: Find Datasets**: link page of dataset repositories, curated by Huajin Wang as part of CMU's Library Guides (https://guides.library.cmu.edu/machine-learning/datasets)

7.4 Data Wrangling

Data wrangling refers to the process of transforming "raw" data into a form that is easier to use for various purposes, and is often a necessary step when using real world instances or external data sets. Broadly speaking, the process of data wrangling can be separated into three steps: **extract**, **transform**, and **load**.

7.4.1 Extract

Extraction refers to the task of reading in the data from the source system. If the data is in an established format like CSV or JSON, we can use pre-existing libraries and Python modules like `csv` and `json`. But sometimes, we need to implement a *parsing* script to read in the data.

In this case, the typical strategy is to read in the file one line at a time, where each line is split into *tokens* (i.e., words in the line separated by whitespace (or some other character)) which are then stored in data structures. In Python, the code follows a pattern similar to:

```
with open('filename.data') as fo:
    for line in fo:
        tokens = line.strip().split(' ')
        # Do things with tokens
```

while in C++, the code is similar to (after including `fstream` and `string` standard libraries);[3]

```
std::ifstream fo("filename.data");
std::string line;
while (std::getline(infile, line))
{
    std::istringstream ls(line);
        // Do things with ls (e.g., int a; ls >> a)
}
```

Within a string, you can find the positions of characters with the `str.find` function (https://docs.python.org/3/library/stdtypes.html#str.find, https://en.cppreference.com/w/cpp/string/basic_string/find), and extract them as substrings.

For particularly convoluted files, you may need to use *regular expressions* (regexes) using Python's `re` module (https://docs.python.org/3/library/re.html) or C++'s `<regex>` library (https://en.cppreference.com/w/cpp/header/regex).[4] See https://www.regular-expressions.info/quickstart.html for a quick tutorial on regexes, and see https://regex101.com/ for an online tool that helps you construct them.

7.4.2 Transform

Transformation refers to modifying, correcting, and extrapolating from the data to make it more suitable for use. Some common tasks include:

- Remove whitespace: Python's `str.strip` (https://docs.python.org/3/library/stdtypes.html#str.strip) and C++'s `remove_if` (https://en.cppreference.com/w/cpp/algorithm/remove) function are helpful.

[3] Alternatively, you can work directly with the ifstream `fo`.

[4] Python and C++'s implementations of regexes differ slightly; see https://www.regular-expressions.info/refflavors.html for a detailed comparison.

- Select entries satisfying a condition: For Python, list comprehensions can be especially helpful here. For C++, we can iterate through the data structure and copy the relevant entries to a separate data structure.
- Deal with missing data: Entries with missing data can either be removed or filled in, using various methods.

7.4.3 Load

Loading refers to saving the data in a structure that can be used easily by other programs and functions. This step may be implicit if we are immediately using the data after we have read it in. Otherwise, we need to choose a format to store the data in: CSV files are usually sufficient. Pythons' `str.format()` (https://docs.python.org/3/library/stdtypes.html#str.format) and C++'s `<<` operator are particularly useful here.

7.5 Generating Test Instances

Artificial instances are typically generated with scripts. Python is especially recommended since it is very easy to write a simple script but still has enough flexibility if you want to use a more sophisticated instance generation scheme. In general:

1. Spend some time considering what your instance data format should be. If there is an established standard format, follow that. Otherwise, favor formats that are easier and more robust to parse than formats that are easier to read visually.
2. Try to name data files in a unique, consistent manner, preferably using only alphanumeric characters.
3. Ensure the test instances generated fit the purpose of the computational experiments. Do you cover a sufficiently wide range of instance sizes? Does the difficulty scale appropriately?

References

[1] David Mitchell, Bart Selman, and Hector Levesque. "Hard and easy distributions of SAT problems". In: *AAAI*. Vol. 92. 1992, pp. 459–465.

[2] Matthias Müller-Hannemann and Stefan Schirra. "Algorithm Engineering". In: *Noch nicht* (2010).

[3] Kate Smith-Miles and Simon Bowly. "Generating new test instances by evolving in instance space". In: *Computers & Operations Research* 63 (2015), pp. 102–113.

[4] Weixiong Zhang. "Phase transitions and backbones of the asymmetric traveling salesman problem". In: *Journal of Artificial Intelligence Research* 21 (2004), pp. 471–497.