Module 8

Ryo Kimura

8 Designing and Running Experiments

In this module, we will discuss how to design and run computational experiments that provide meaningful results and data suitable for analysis.

8.1 Why Do Computational Experiments?

Computational experiments are not the only method for evaluating algorithm performance; in particular, *theoretical analyses* often provide deep and powerful insights about the fundamental limits and nature of broad classes of algorithms that can often be extended and generalized beyond a single algorithm or a single problem instance.

However, relying solely on theoretical analysis can paint a misleading picture of algorithm performance. For example, the simplex algorithm for linear programming has exponential worst case running time [6] but often performs well in practice, while Robertson and Seymour's polytime algorithm for detecting graph minors [9, 10] is completely impractical due to a hidden constant on the order of 10^{150}. Furthermore, many heuristic methods (e.g., simulated annealing, tabu search) and widely used policies (e.g., join-shortest-queue scheduling) cannot be analyzed theoretically, and thus computational experiments must be used to perform principled analyses of their performance. Finally, computational experiments provide the most direct evidence for justifying performance of the algorithm in practice, especially when test instances with realistic parameters are used.

8.2 Designing Computational Experiments

(This section is based on sections 8.1 and 8.2 of [8].)

There are three main questions to consider when designing computational experiments:

1. **What are the goals of the experiment?**
2. **What factors do we want explore?**
3. **How should we measure performance?**

8.2.1 Defining Goals

Computational experiments should always be designed so that they are able to satisfy the goals of conducting the experiment. **Choosing an appropriate research goal/hypothesis is the most important aspect of desigining computational experiments.** Common goals include:

- Proof-of-concept of a novel framework/methodology/model/algorithm

- Analyzing an algorithm/problem's behavior to better understand it
- Comparing the performance of competing algorithms
- Showing the relevance of an algorithm for a specific application
- Brute-force/numerical validation of a theoretical conjecture

It is important that the defined goals be *testable* (i.e., Will running experiments answer the question?), *significant* (i.e., Why should anyone care about the results?), and *insightful* (i.e., What implications do the results have beyond the current experiment?).

Alternatively, we may think about what *hypothesis* we want to confirm/reject with the experiment. Some examples are:

- "VRP is easier to solve when the underlying graph has low tree-width."
- "Branching rules that minimize the pseudocost result in smaller branch-and-bound trees."
- "For stochastic LPs, an iterative method is better suited for larger problems, while a reformulation method is better suited for smaller problems."

8.2.2 Determining Factors

The *factors* of a computational experiment are akin to independent variables in statistics and predictors/features in machine learning; that is, the goal of an experiment is to see how differences in experimental factors translate to differences in various performance metrics. The most common factors include:

- *Algorithm parameters* (e.g., maximum number of cuts/iterations, target error/approximation ratio, branching/node selection rules, etc.)
- *Instance size* (e.g., number of customers/items/jobs/etc.)
- *Instance characteristics* (e.g., tree-width of a graph, number of nonzero entries, etc.)

We also need to define *levels* for each factor in order to test the effect of differences (e.g., big/medium/small instances, $\alpha = 0.01, 0.1, 0.5$, etc.).

8.2.3 Choosing Performance Metrics

The most common performance metrics in use are *total running time*, *space consumption*, and *solution quality*, since they are often the most relevant in practice. However, focusing solely on these metrics can lead to an overly *competitive* approach to computational testing, an approach that was once common in research on heuristic algorithms (in the 1990s) and has since been criticized [3, 11] for producing papers of dubious scientific quality contributing little to the general theory of heuristics.

Some other classes of performance metrics of potential interest include:

- *Extensions of running time* (e.g., time to find a feasible solution, time to certify optimality, total time spent running heuristic/subproblem, minimum/average/maximum amount of time spent on one iteration)

- *Structural measures* (e.g., number of nodes in branch-and-bound tree, number of iterations, number of memory references)
- *Bottleneck operation counts* (e.g., number of oracle calls, number of subproblems solved, number of cuts generated)
- *Meta-metrics* (e.g., robustness of algorithm, ease of implementation)

As with factors, performance metrics should be chosen to best illustrate the goal of the experiment.

8.3 Running Computational Experiments

8.3.1 Advice from the Literature

The following is some advice from the literature on running computational experiments:

Johnson, "A theoretician's guide to the experimental analysis of algorithms" [4]:

- *Perform newsworthy experiments.*
  - **Think before you compute.**
  - Don't over-analyze the results of one or two instances. It's better to test a variety of instances in a systematic way.
  - **Don't run full experiments until you've (a) confirmed that your implementation is correct, (b) finished making your implementation efficient, and (c) decided what data to collect.** Otherwise, you will have to re-run the experiments, resulting in a huge waste of time.
  - Use "exploratory experimentation" to find good questions, but don't get stuck in an endless loop.
- *Tie your paper to the literature.*
  - **Do your homework. Know what is the state-of-the-art.** What has already been done, what is still unknown, what is missing from the literature?
  - Make comparisons to previous experiments. What is the "standard" approach? Are there similar or alternative approaches?
  - Don't waste your time comparing with "dominated algorithms", unless that's the point (e.g., popular algorithm A is actually really bad at problem X).
- *Use instance testbeds that can support general conclusions.*
  - Don't use unstructured random instances. It's hard to generalize the results.
  - Don't only run experiments on easy test instances (e.g., solve in less than second, have already been solved). The algorithm may perform differently on hard instances.
  - Don't only run experiments on test instances that are favorable for your algorithm. The algorithm may perform differently on other types of instances.
  - **In general, aim for a healthy range/variety of test instances.**
- *Use efficient and effective experimental designs.*

- Use variance reduction techniques (e.g., common random numbers, control variates, batch means [7, 2]) when investigating the average performance of an algorithm.
- Use self-documenting programs.

- ***Use reasonably efficient implementations.***
  - **Make fair comparisons with existing approaches.** E.g., exact algorithms, approximation methods, and heuristics are all designed with different tradeoffs in mind.
  - Inadequate programming time/ability is NOT an excuse.
  - That being said, avoid too much code tuning (unless, of course, that is specifically what you are testing).

- *Ensure reproducibility.*
  - **Make sure the code you implement matches the paper's description of it.**
  - Ensure your reported performance metrics are reproducible.
  - Don't use running time as a stopping criterion (e.g., stop the algorithm after one hour); it's not reproducible since different machines have different speeds. Use combinatorial limits instead (e.g., limit on number of nodes in branch-and-bound tree; limit on number of constraint failures).
  - Don't use parameters that aren't known in practice (e.g., optimal value of a problem, modulus of strong convexity) for tuning; only use them for evaluation.
  - Algorithm parameters should be tuned systematically, not by hand.
  - Don't base all of your analysis on a single run.

- *Ensure comparability.*
  - **Report environmental details (e.g., software versions, processor speed, amount of memory, number of CPU cores)**
  - Ensure test instances used are available.
  - Backup everything you need to reproduce the experiment (especially test instances, source code, and input data), so you can provide it when you're asked for it later.

Journal of Heuristics Policies on Heuristic Search Research [5]:

- **If an experiment shows that a procedure performs better than its competitors, it must provide *insight* as to *why* it performs so well.**

- **Customized methods should always be compared with general-purpose, "off-the-shelf" optimizers.**

- Statements about performance should be backed up by statistically valid experiments [1].

- Procedures that require parameter tuning must (a) separate tuning data from testing data, and (b) demonstrate generalizability (i.e., in machine learning terms, **training data should be separate from testing data**, and the model should avoid overfitting).

Müller-Hannemann and Schirra, *Algorithm Engineering*, Section 8.5 [8]:

- Use reasonably efficient implementations
- **Check your input data**
- **When in doubt, record it (logging/output)**
- Use version control software
- Use scripting to automate repetitive tasks
- Keep notes on experiment details
- Change only one thing at a time
- Run it often enough and with appropriate data sets
- **Check that the output/results make sense**
- Do unusual things, test the trivial/extreme/corner cases
- Set appropriate time/resource limits

8.3.2 Getting/Recording Solver Information

Information about code that we have written is easy to record because we have full control over the process. However, to record information related to the MIP solver we need to use functions from the corresonding API.

- **Gurobi**: Most information can be accessed through *Attributes*, which are associated to the relevant objects; for example, the number of nodes in the branch-and-bound tree can be obtained via the `NodeCount` attribute of `Model` objects:
  - **Python**: if `m` is a `Model` object (e.g., `m = Model("myModel")`), then we can access its `NodeCount` attribute via

    ```
    m.getAttr(GRB.Attr.NodeCount)
    m.NodeCount
    ```

    Note that object attributes in Gurobi's Python interface do not distinguish case, so `m.nodecount` also works (but `m.getAttr(GRB.Attr.nodecount)` does NOT work.)
  - **C++**: if `m` is a `GRBModel` object (e.g., `GRBModel m = GRBModel(env);`), then we can access its `NodeCount` attribute (which is a double attribute) via

    ```
    m.get(GRB_DoubleAttr_NodeCount);
    ```

  See http://www.gurobi.com/documentation/current/refman/attributes.html for a list of the different attributes with short descriptions. See http://www.gurobi.com/documentation/current/refman/attribute_examples.html for more details on how to access attributes.
- **CPLEX**:

- **C++**: Use getter methods of the objects. Most information that is not associated with a specific variable (`IloIntVar`) or constraint (`IloConstraint`) is associated with the `IloCplex` instance. If `cplex` is an `IloCplex` instance associated with an `IloModel` object `model` (e.g., `IloModel model = IloModel(); IloCplex cplex(model)`), then

  ```
  cplex.getNnodes();
  ```

8.3.3 Getting Optimization Status

8.3.4 Measuing runtime

8.3.5 Using scripts

8.3.6 Recording output

References

[1] Marie Coffin and Matthew J Saltzman. "Statistical analysis of computational tests of algorithms and heuristics". In: *INFORMS Journal on Computing* 12.1 (2000), pp. 24–44.

[2] George S Fishman and L Stephen Yarberry. "An implementation of the batch means method". In: *INFORMS Journal on Computing* 9.3 (1997), pp. 296–310.

[3] John N Hooker. "Testing heuristics: We have it all wrong". In: *Journal of heuristics* 1.1 (1995), pp. 33–42.

[4] David S Johnson. "A theoretician's guide to the experimental analysis of algorithms". In: *Data structures, near neighbor searches, and methodology: fifth and sixth DIMACS implementation challenges* 59 (2002), pp. 215–250.

[5] *Journal of heuristic policies on heuristic search research*. http://www.springer.com/cda/content/document/cda_downloaddocument/Journal+of+Heuristic+Policies+on+Heuristic+Search.pdf?SGWID=0-0-45-1483502-p35487524. See "Policy Statement (pdf, 451kB)" link on https://www.springer.com/mathematics/journal/10732.

[6] Victor Klee and George J Minty. *How good is the simplex algorithm*. Tech. rep. Washington Univ Seattle Dept of Mathematics, 1970.

[7] Catherine McGeoch. "Analyzing algorithms by simulation: variance reduction techniques and simulation speedups". In: *ACM Computing Surveys (CSUR)* 24.2 (1992), pp. 195–212.

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

[9] Neil Robertson and Paul D Seymour. "Graph minors. XIII. The disjoint paths problem". In: *Journal of combinatorial theory, Series B* 63.1 (1995), pp. 65–110.

[10] Neil Robertson and Paul D Seymour. "Graph minors. XX. Wagner's conjecture". In: *Journal of Combinatorial Theory, Series B* 92.2 (2004), pp. 325–357.

[11] Kenneth Sörensen. "Metaheuristics–the metaphor exposed". In: *International Transactions in Operational Research* 22.1 (2015), pp. 3–18.