A Practical Guide to the Empirical Evaluation and Comparison of the Performance of Algorithms Operating on CSPs

Anagh Lal, Praveen Guddetti, Joel Gompert and Berthe Y. Choueiry

Constraint Systems Laboratory
Department of Computer Science and Engineering
University of Nebraska-Lincoln

{alal|vguddeti|jgompert|choueiry}@cse.unl.edu

December 14, 2004 at 6:07 p.m.

Abstract

This report highlights the important steps in experiment design and set-up when conducting experimentations on randomly generated problems.

Contents

List of Tables

1 Generating and storing random problems

In this section we look at how to generate and store random CSPs. We also look at how to determine an appropriate number of samples to generate.

1.1 Generating problems

To generate random problems you can use one of the random generators[1] already developed by the lab or the research community (some of these are listed on the web page of the group under the heading *Resources on Constraints: Generators of random CSP instances*). Typically, the random generators have the following generic usage:

`random-generator` *parameter-list output-file-name*

The parameters are typically $\langle N, a, p, t\rangle$ where N is the number of variables, a is the domain size, p the constraint density and t the tightness of the constraints. The output file is where the random CSP is generated. You will need to understand the output format used by these generators and write code to load it into your algorithms' data-structures.

While the storing of generated CSP instances into files and loading them from these files when conducting experiments require additional programming, the approach has many benefits. Here is a list of reasons to store your random problems:

- When comparing two algorithms, it is prudent to compare them on the *same set* of problems. By generating and storing your random problems *separately* you have the advantage of having pair-wise comparisons of the performance of the two algorithms on each instances and not just a comparison of the averages of the performance values. Moreover, the presence of such paired data is a requirement for paired statistical tests. Therefore, storing your random CSPs is a must if you plan to use statistical tests that pair values to compare algorithms.

- You do not have to run the random generator every time you make a change in your algorithms' implementation. Moreover, it makes it easier for you to compare the effects of the change.

- In case of anomalies in the behavior of your algorithm, peeking into the structure of the problem instance may help in explaining the anomaly.

- In case, for some reason, your experiments are interrupted (e.g., power failure or computer bug), then you know where you restart your experiments from.

- Your successor in the lab will have a problem data-set to evaluate his/her algorithms on, and will be able to make statistical tests to compare with your algorithm. (You might not want to store your problems in case you want to remain supreme and prevent everyone from proving themselves statistically better than you in the future.)

As your experiments grow, carefully managing your data becomes important. A useful way is to create directories representing your parameter values. For example:

- The root of my experiments directory has two directories `N20` and `N30`.

[1]Make sure that the graphs of the problem instances you use are *connected*.

- Each one of these has `a10, a15` and each one the '`a`' has directories representing constraint density and so on.

Such a schema allows you to write scripts or programs to feed data to your algorithms. Another tip is to give each random CSP file a unique name with all its defining parameters in the name followed by the sequence number of the problem. For example, the name:

`N20_a15_p0.5_t0.3_100`

lists all the regular CSP parameters and also indicates that it is the 100^{th} instance in the data set. We found that this convention useful when writing scripts.

1.2 Estimating the number of problems to generate

For a given set of parameter values, how many instances should one generate? This is a tough question to answer. You can use the following steps to have a good idea. For statistical testing to be powerful, a large number of problems or samples is preferred. The larger the number of samples the more robust the results are. This is because a larger number of samples better approximates the set of all possibilities called the 'population,' which allows you to verify whether the data obeys a known distribution. Ensuring that the data follows a known distribution is important for parametric tests, which have distributional assumptions. A large sample size is inconvenient in the sense that testing takes more time and requires managing more data.

To determine the sample size, choose the most representative CSP (i.e., a set of values for the parameters $\langle N, a, p, t\rangle$) and generate a very large number of samples. We used 10'000 instances. Solve these instances and analyze the results as follows. Let us assume the metric being used is CPU time. Plot the *moving average* for all the CPU times. We explain computing the moving average using Table 1.

Table 1: Sample of running averages.

Number of samples	**Mean of some measurement (e.g., CPU time) over samples**
10	90.5
20	100.5
40	110.5
80	105.5
100	100.0
125	106.0
150	105.5
175	104.5
200	105.0
250	105.5
300	105.0
350	104.8
400	105.0
450	105.1
500	104.9
1000	105.0
2000	105.1
3000	104.9

The table shows the mean CPU time for a given number of samples. We observe that the mean varies for smaller sample sizes. For example we can see the mean is 'rather unstable' from 10 to 20 to 80 sample points. Therefore, such sample sizes are not representative. The mean in Table 1 starts stabilizing around 150 sample points. At 250 sample points, it is relatively stable with relatively small variations. This mean value holds up for larger samples too. The example shown here is synthetic, and things may look very different for your data. Figure 1 shows the moving average of the number of constraint checks in experiments carried out by Anagh Lal on dynamic bundling.

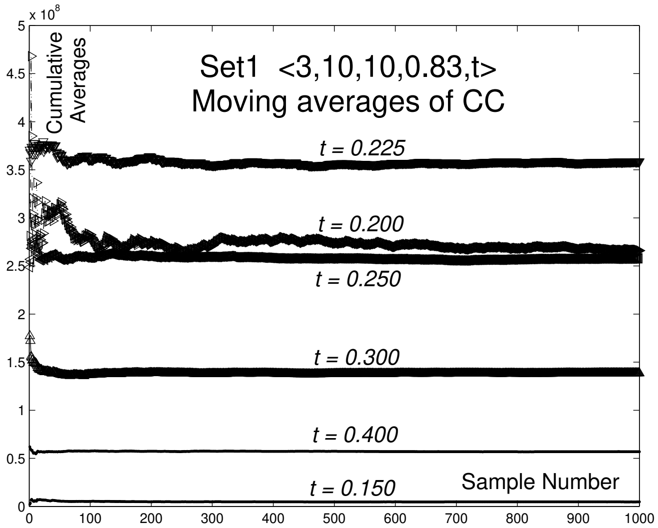

Figure 1: *Moving average of* CC *for DataSet1.*

Figure 2 of the CPU time in experiments carried out by Praveen Guddeti on randomized backtrack search.

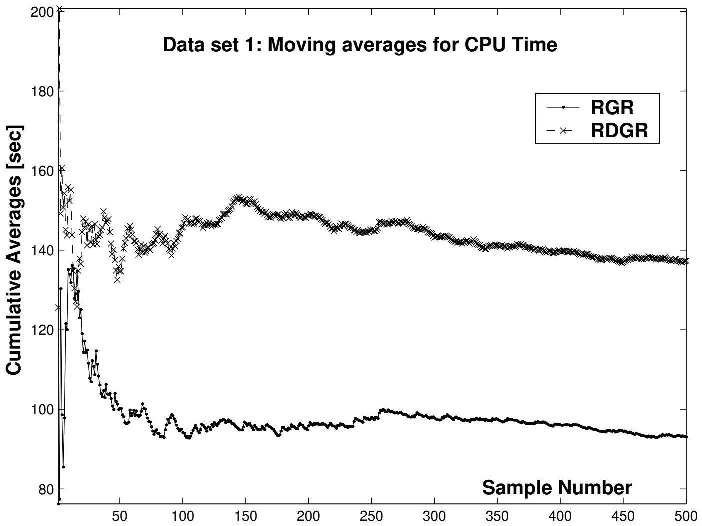

Figure 2: *Moving average for CPU run-times for data set 1.*

In our experience, we saw means stabilize around 1'000 sample points. It resulted in a large amount of data. 1'000 sample points may be an overkill if you are using non-parametric tests (see Sections 2 and 5.2). Consequently, it will be wise to analyze the result of the above experiment and decide what kind of tests you are likely to use. Section 5.2 describes the steps to select the type of statistical test. If you are sure that you will not use parametric tests, then a you can take a sample size that is less than ideal.

2 Terminology

We will use terms that we are not necessarily familiar with. Here are some:

- *Population*: A population is a class of problems over which you are trying to study the performance of your algorithms. For example, this could be random CSPs characterized with given values of $\langle N, a, p, t \rangle$.

- *Sample*: Because you cannot possibly test your algorithms over all the instances of a population (because the population is huge), you generate a sample of a limited size (e.g., 100 instances or 1'000 instances) and assume that the instances in your sample are representative of the entire population. Naturally, if you sampling is not careful, you may end up with a sample that is not representative of the population. For example, to generate connected random CSPs of a given density, it is incorrect to generate a chain (or a star) to guarantee connectedness of the constraint graph, and then randomly add the remaining edges until you get the specified density. The *sample size* is the number of instances in your sample.

  If you are testing a stochastic algorithm (one which does not always yield the same value of the metric across repeated tests), you could either (1) repeat the executions of your algorithm on the same problem instance, or (2) execute your algorithm once on each instance in the sample. The latter constitutes a test over an *ensemble of instances*, see Section 5.2 for the tests appropriate for these two situations.

- *Survival (or failure) time analysis*: This is the type of analysis you are doing when you are measuring the run time of an algorithm until some event occurs. In our studies, the 'failure' is that the problem was solved. In such situations, you want to use a non-parametric test (see below) to compare samples.

- *Censored observations/data*: In the context of a run-time analysis of an incomplete algorithm, we cut off the algorithm after some time has elapsed. In this case, we say that we have 'censored' data points and that we do not have the true run-time for that run. You have to use a test *that takes into account censored data*. Examples of such tests are the Cox-Mantel test and Log-rank test, which are non-parametric tests. Parametric and semi-parametric tests do also exist for censored data. SAS apparently includes functions for doing these tests. Here is a web-site that has some information on the topic:

  `http://www.statsoft.com/textbook/stsurvan.html#comparing`

- *Metric*: The metric is the quantity that you are trying to study and by which you are comparing the performance of your algorithms. It could be the CPU time, the number of nodes visited, the number of constraint checks, the number of solutions found in a backtrack-free manner, the size of the first solution-bundle, etc.

- *Probability distribution*: The probability distribution function is a theoretical construct that corresponds to a population distribution. To empirically assess some performance metric of your algorithm, you run algorithm on the instances of your sample (alternatively, you run the same stochastic algorithm repeatedly on the same instance). Then, you can draw the *probability distribution function*, which counts the number of times a given value of your metric (e.g., CPU time) occurred, and organizes the values in increasing order. Figure 3 shows an example of a *empirical* probability distribution function (PDF).

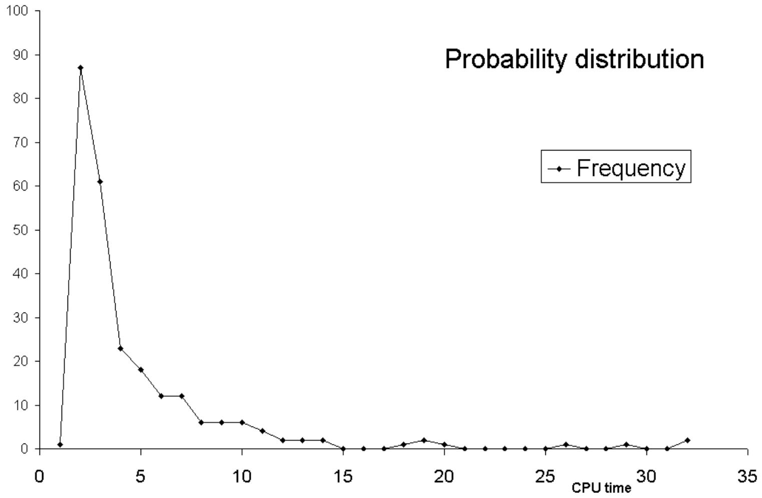

Figure 3: *Empirical probability distribution function.*

- *Cumulative distributions*: Once you have the (empirical) PDF, you can easily draw the cumulative distribution function, which is the integral of the pdf. For a given value of your metric (e.g., CPU time), it says what is the percentage of times your algorithm performed in a value less or equal to the chosen value. Figure 4 shows the *empirical* Cumulative Distribution Function (CDF) of the PDF shown in Figure 3.

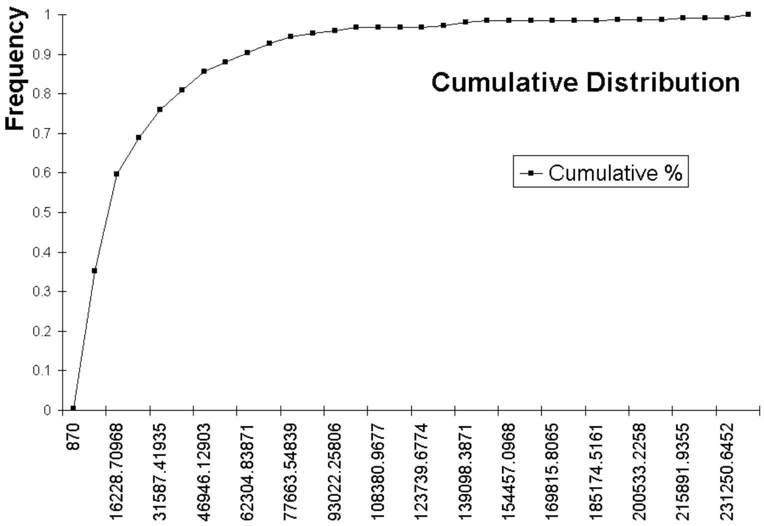

Figure 4: *Empirical cumulative distribution function.*

- *Distribution functions*: Some PDF's are very well studied and often used to model the behavior of some biological, physical, or engineering process. Examples include the normal distribution, the gamma distribution, the heavy-tailed distribution, etc. Section 6.2 provides pointers to some known distributions and example of phenomena they were used to model. In particular, the following site has really cool animations:

  `http://www.stat.vt.edu/ sundar/java/applets/Distributions.html`

  It is a good idea to estimate whether the metric you are measuring (e.g., CPU time and number of problems solved within a give amount of time) follow some kind of distribution[2] (e.g., a normal distribution, a gamma distribution, or a heavy-tailed distribution). In order to decide which distribution to choose, you draw a probability distribution curve. If you find that your metric (or variable) follows a particular distribution, then you should look for a parametric test that is most appropriate for the distribution. Otherwise, you analyze your data using a non-parametric test.

  While it would be nice to have the 'exact' distribution of your data, this is not absolutely necessary. Most tests that assume normality are actually quite robust to violations of this assumption (although your reviewers may strongly object and be impossible to convince). In fact, normality is the least important assumption of normality-based procedures (again do not try to argue with the reviewers). As long as the distribution of the data is basically symmetric, a normal procedure will be fine.. The choice of a test is further explained below.

- *Statistical tests*: Tests take as input the values of your metrics and are used to determine with some confidence levels to what extent the null hypothesis can be rejected (i.e., we try to find evidence to reject it). We strongly recommend that you discuss your experiments with a statistician and decide with him/her which test is the most appropriate test for your context. Keep in mind that: "There are no widely accepted guidelines concerning which test to us in a particular situation" [StatSoft, 1998]. Note that people in AI frown upon those who use the t-test in CSPs given that metrics do not follow the normal distribution, however, there are ways to get over this restriction by chopping the data into samples and computing the averages in each sample and applying the t-test to these average values (because these averages can be safely assumed to follow a normal distribution under the Central Limit theorem..), as it is widely done in engineering.

- *Parameters* are indices. They index (or label) individual distributions within a particular family. For example, there are an infinite number of normal distributions, but each normal distribution is uniquely determined by its mean and standard deviation. If you specify all of the parameters (here, mean and standard deviation), you have specified a unique normal distribution.

- *Parametric tests*, such as the t-test, estimate the value of a parameter (e.g., the mean) of the variable (e.g., CPU time). These tests depend on knowing the distribution of the parameter. Indeed, the t-test assumes normality of the mean of the variable.

- A *non-parametric test* does not rely on parametric assumptions like normality. However, they may have other assumptions, which you still need to check. Some non-parametric tests order the sample values and rank them. (Note: not all non-parametric tests are based on

[2]See Section 6.2 for pointers to various distributions.

ranks.) They use these ranks for deciding whether two sets of readings for a variable come from the same population or not. In other words, non parametric tests sort the CPU time readings and rank each of the CPU time readings.

`http://www.columbia.edu/cu/psychology/courses/S1610q/18Nonparametric.ppt` is a presentation that shows the working of the Mann-Whitney U Test with a small example. Because these tests are independent of the distribution of the variable or the parameters, they are called non-parametric tests.

> Examples of non-parametric tests include:
>
> – the various forms of chi-square tests,
> – the Fisher Exact Probability test,
> – the Mann-Whitney Test,
> – the Wilcoxon Signed-Rank Test,
> – the Kruskal-Wallis Test,
> – and the Friedman Test.
>
> Non-parametric tests are sometimes spoken of as 'distribution-free' tests, although this too is something of a misnomer.
> From Richard Lowry, 1999-2004
> `http://faculty.vassar.edu/lowry/`

3 Posting jobs on PrairieFire and Sandhills

Prairiefire and Sandhills are two machines of the Research Computing Facility (RCF) at CSE (`http://rcf.unl.edu/`) that are available for running computationally intensive jobs[3]. To use these machines you need to follow these steps:

1. Request an account on these machines. You have to make separate requests for each of them. To request an account, go to `http://rcf.unl.edu/newuser/`. It allows you to request for accounts on these two machines.

2. Transfer your code to the your home directory on PrairieFire. If you are using Lisp, delete all the binary files (fasl), and transfer only the source code. Recompile your code.

3. Load your random CSPs on PrairieFire.

4. Write appropriate scripts to post jobs to PrairieFire. We discuss this in detail below.

[3] As we are writing this report, the following machines are available:

- RCF: A 32 processor SGI Origin 300 is RCF's original machine.
- Prairiefire: 128 dual processor AMD machines packed together into a small space make for one hot cluster. It also makes up RCF's beloved Prairiefire.
- Sandhills: Also known as 'the old Prairiefire,' Sandhills is a 40 node dual processor AMD cluster.
- Bugeater: The original cluster, an eight node dual processor Intel based collection of machines.
- Prairieview: A visualization cluster powering RCF's Tiled Display wall for high-resolution images.

It is a good idea to write programs that allow you to solve many CSPs at a time. They take as input the location of the random CSP files and load one CSP at a time and invoke the algorithms that you want to run. Once the CSP instance is solved, this program collect the values of measurements made (such as CPU time, nodes visited, constraint checks, and number of solutions) and append these results to a file of your results. You could invoke all the algorithms you want to compare one after the other for each CSP. This allows you to claim that the problems were solved under similar server loads.

To run jobs on PrairieFire you need to submit your request to the server. You can-not run it from your home directory. If you do, your process is likely to be killed, and once you get disconnected from PrairieFire the process terminates. We now look at how to submit jobs to PrairieFire. The following steps are needed to run jobs on Prairiefire.

1. Login and file transfer.
2. Writing PBS script.
3. Managing jobs.

3.1 Login and file transfer

For security reasons, Prairiefire does not accept `telnet` or `ftp` connections. Only secure login (`ssh`) and secure file transfer (`scp` or `sftp`) connections are allowed.

3.2 Writing PBS script

Do not run jobs directly on the head node, which is the node you log into. Jobs have to be scheduled using the PBS Pro scheduler, which you do by writing a PBS script for the job and entering it into the queue. This goal of this operation is to ensure that the load across the cluster is properly balanced and to prevent time-critical applications such as benchmarking. We give below an example of a PBS script.

```
#PBS -W group_list=choueiry
#PBS -N FC
#PBS -l nodes=1,walltime=72:00:00
#PBS -S /bin/sh
#PBS -q csl_short
#PBS -M root@localhost
#PBS -m abe
#PBS -j oe
cd /home/choueiry/username/RandomCSP/
/home/programs/Allegro_CL6.2/redhat6/acl62/alisp -#! script-FC.lisp > error-FC.txt
```

The only options required to change are: group_list, N, nodes, walltime, q, and the path of the programs.

- Group_list specifies the group the user belongs to, which, in Prairiefire, is the name of the advisor (in the example above it is *choueiry*).
- Option N is used to give a name to the job (FC in the above example).
- Nodes specify the number of CPUs requested.

- Walltime is the CPU time requested for the job. If this CPU time is less than what is needed to finish the job, then the job is killed even if it was not completed.
- Option 'q', specifies which queue the job belongs to.
- Finally, change directory to where the programs/code are stored and the specify the path of the executable/compiler.

In general, jobs are classified in debug, short, long.

- Each debug job is allowed to run for a maximum of 35 minutes.
- Each short job is allowed to run for a maximum of 72 hours.
- Each long job is allowed to run for a maximum of 360 hours.

At the time of writing this report, our group has the following number of jobs:

- debug: 1 job
- csl_short: 28 jobs
- csl_long: 2 jobs2

The more write papers about your research the number of jobs that we can request on PrairieFire increases. For special requirements, you need to negotiate with the system administrator to obtain a custom queue.

If you are using a language that supports parallelization, debug jobs provide access to a maximum of 2 CPs, short jobs to 32 CPUs, and long jobs to 8 CPUs. Allegro Common Lisp (ACL) installed on Prairiefire[4] does not support the parallel execution of a job. Hence, for jobs using LISP, one CPU is used. For other jobs, such as C programs, more than one CPU will speed up their execution.

3.3 Managing jobs

Once the PBS script is ready, the job has to be submitted to the queue.

- To submit a job, use the command `qsub filename`, where filename is the name of the file that contains the PBS script.
- You can check the status of the jobs with the command `qstat` or `qstat -u` *user-name* for a specific user.
- To kill a job, use the command `qdel job-number`.

4 Managing results

Once your jobs are completed on PrairieFire, you must start compiling your results into formats readable by statistical packages. You can do so by generating your results in a tabular format. Assume you have solved 1000 problems with the same CSP parameters using two methods *Method1* and *Method2*. You may generate a result file such as the one shown in Table 2.

[4]Note that there is a Common Lisp implementation that supports multiprocessor systems using real OS threads `http://www.scieneer.com`, which remains to be tested.

Table 2: Sample result-file format.

Problem-Number	Method	CPU time	NV	CC
1	Meth1	100	25	1000
1	Meth2	125	30	1250

Separating entries on each line with a tab or a comma makes it easier to load your result file into statistical packages. MS Excel imports such files directly. From our experience, it is usually preferable to keep the results of the two methods in separate files, which makes it even easier to use statistical packages on the results.

You can do so by generating the results separately, writing them results into separate files, or separating them from a unique file after they have been generated. If the result file has results from two or more methods, you need to write a simple program that reads result file line by line and stores the result of a method in a corresponding result file.

5 Using the R package for data analysis

R is an open-source statistical package. You can down-load it from `http://www.r-project.org`. It is installed on gubbio.unl.edu, which is the PC in the lab. R has a graphical interface very similar to Matlab. It has an interpretive work-environment very similar to that of LISP. It has in-built functions for statistical tests that we discuss below. First, we discuss methods for loading data into R (Section 5.1), then we discuss the functions for performing statistical comparisons, which require the following two steps:

1. *Analyzing your data.* Analyze your data and select an appropriate statistical test. Section 5.2 discusses this step in detail.

2. Use appropriate R function calls to compute test statistics. Section 5.3 discusses this step in detail.

5.1 Loading data

The following command in R allows you to load the data:

```
v1 <-scan(‘‘c:\\MyDataDir\\result_Method1.txt’’, list(0,‘‘’’,0,0,0))
read 1000 records -- output of package
v2 <-scan(‘‘c:\\MyDataDir\\result_Method2.txt’’, list(0,‘‘’’,0,0,0))
read 1000 records -- output of package
```

The first parameter of the above command is the file to be loaded. The second parameter describes the format of the file. To indicate a number use ‘0’ and to indicate a string use double quotes (“”). Thus, the above code reads results for the two methods in two vectors `v1` and `v2` respectively. The code below allows you to access the CPU time of both these vectors:

```
v1_cpu <- v1[3]
v2_cpu <- v2[3]
```

5.2 Selecting the test

Start with one data set, such as the data set that you used to estimate the sample size (see Table 1 and Figures 1 and 2). Plot the distribution of your readings for all the algorithms/methods and check whether they fit with normal distribution (or any other known distribution). *Compute the variance in readings from all tested methods.* We give a brief summary of the tests to check whether the t-test can be used:

1. *Sample size*: Sample sizes of the measurements should be large and equal for the tested algorithms. Although this is not a requirement, unequal sample sizes may affect the test drastically unless the sample sizes are significantly large.

2. *Outliers*: Outliers are anomalous values in the data, for example extremely high or low CPU times. Measured values may not be identically distributed because of the presence of outliers. Outliers tend to increase the estimate of sample variance, thus decreasing the calculated t statistic and lowering the chance of rejecting the null hypothesis. Outliers may be due to recording errors, which may be correctable, or they may be due to the sample not being entirely from the same population. Apparent outliers may also be due to the values being from the same, but non-normal, population. If you encounter outliers in your test results then you need to analyze these specific readings. These anomalies may be due to the random generator inadvertently adding some singularity to the CSP. If this is the case, these anomalous results do not belong to the same population. After consulting with a statistician[5] (and your advisor), you can consider detecting and removing such problem instances from the data set and treat them separately. However, if these anomalies are not due to unique circumstances, the presence of outliers indicates non-normality of your data.

   The t statistic is based on the sample mean and the sample variance, both of which are sensitive to outliers. In other words, neither the sample mean nor the sample variance is resistant to outliers, and consequently, neither is the t statistic. In particular, a large outlier can inflate the sample variance, decreasing the t statistic, and, thus, perhaps eliminating a significant difference. A non-parametric test may be a more powerful test in such a situation.

3. *Normality*: The values in a sample may indeed be from the same population, but not from a population whose metrics (e.g., CPU time) do not follow a normal distribution. Signs of non-normality are skewness (i.e., lack of symmetry), light-tailedness, or heavy-tailedness. However, if there is only a small number of data points, non-normality can be hard to detect. You need to have a large sample size to determine whether or not you have normality.

4. As stated above, you need to compute the variance of your sample. Because it is impossible to compute the true variance of your metric over the entire population of problem instances (with the same CSP characteristics), then you estimate this variance over the instances in your sample. You can do this either informally (including graphically) or by a variance test such as the F test. The F test for checking the equality of variances is very sensitive. However, while the equality of variance is a very important assumption[6] to verify, the t-test is very robust to slightly unequal variances (using the F test might be pessimistic).

   If you are testing the algorithms over samples that have the same size, then the effect of different values of the variances is not as critical, and the the t-test is fairly robust in this

[5] A statistician may be able to help you fix this problem using some kind of transformation on your data, such as taking the logarithm of the values instead of the values themselves. Do not do such a transformation without thoroughly discussing it with the statistician.

[6] Normality is the least important assumption.

situation (assuming the sample sizes are equal). The effect of this difference is most severe when the sample sizes are unequal and the smaller sample has the larger variance. The F test is performed easily using the R package.

```
var.test(v1_cpu, v2_cpu)
```

The test returns a *P-value* of the test for equality of variances. If the two variances are similar you will get a high *P-value* (> 0.05).

If the populations from which the data to be analyzed by the t-test were sampled violate one or more of the t-test assumptions (which are the above four), the results of the analysis may be incorrect or misleading. If the assumption of normality is violated, or outliers are present, then the t-test may not be the most powerful test available, and this could mean that you may miss detecting whether or not the performance measures of your algorithms are truly statistically different.

A non-parametric test or employing a transformation (such as the above-mentioned log transformation) may result in a more powerful test. Unless it is obvious, it is advisable to consult a statistician on selecting the right test to use. Section 6.3 has the details of a statistical help-desk. For example, we, the authors of this report, found the data in our experiments to be non-normal and decided to use a non-parametric test.

The choice of the parametric test depends on your experiment set-up. When your experiment generates a set of random problem instances and solves every problem using each of the methods and collects the metrics, the data you have is said to be matched. Consider the following example to understand what we mean by matched data. Let M1 be the vector representing the CPU time taken to solve a set of problems with the same characteristic and similarly M2 is for the second method. M1[1] is the CPU time for problem instance 1 by algorithm 1 and M2[2] for algorithm 2 and so on. We say M1 and M2 are matched on the problem instance.

In the case of stochastic algorithms, when a single problem instance is solved repeatedly by the methods we cannot generate matched data. This is because there is nothing that we can match or pair the readings with.

In the statistics literature you are likely to come across these three names:

1. Mann-Whitney U Test.

2. Wilcoxon rank-sum test.

3. Wilcoxon matched-pairs signed rank test.

The Wilcoxon matched-pairs signed rank test is applicable when you have matched data. The Mann-Whitney U Test is equivalent to the Wilcoxon rank-sum test and are applicable when the data is not matched. According to Hoos [2004], if you are comparing two stochastic local-search algorithms, the following two non-parametric tests are useful:

- If you are comparing the algorithms by running them on the same problem instance, use the Mann-Whitney U-test to show statistically significant differences in median run-time.

- If you are comparing the algorithms by running them on an ensemble of instances, use the Wilcoxon matched pairs signed-rank test.

Check the following for additional information on non-parametric tests:

`http://www.tufts.edu/ gdallal/npar.htm`

5.3 Running the tests in the package R

The R package provides easy functions to perform t-test, the Wilcoxon matched pairs signed-rank test, the Wilcoxon rank-sum test (a.k.a. Mann-Whitney U-test).

The t-test, which is parametric, can be called from the R package as follows:

```
t.test(v1_cpu, v2_cpu, conf.int = TRUE, paired = TRUE)
```

The Wilcoxon matched pairs signed-rank test, which is non-parametric, can be similarly invoked:

```
wilcoxon.test(v1_cpu, v2_cpu, conf.int = TRUE, paired = TRUE)
```

The parameter `paired` indicates whether or not the data is paired. Setting `paired = FALSE` results in the Wilcoxon rank-sum or the Mann-Whitney test to be performed on your data. The details of these functions can be found in the manuals of R available online and with the R software.

Both these tests test whether the hypothesis (called the *null* hypothesis) that the two input vectors have the same mean value (for the t-test) and the median value (for the Wilcoxon function). These tests return a *P-value*. The *P-value* indicates the probability that of difference of the means (respectively medians) that you have obtained assuming that the null hypothesis is trues.

. A low *P-value* rejects the null hypothesis. By setting parameter `conf.int` to TRUE, the tests give the confidence interval for the mean/median value of the difference for the alternative hypothesis. The default confidence-level is 95%, which is what we typically use. You can change the confidence level by additionally passing the parameter `conf.level = 0.99` for a 99% confidence level.

6 Additional pointers

Below we list a few pointers that you may find useful in your endeavor.

6.1 Software packages for statistical analysis

Fortunately, you do not have to implement your own statistical functions. There are many public-domain and commercial packages available. Below is a non-exhaustive list.:

- The spreadsheet Excel has many statistical functions and an implementation of the t-test.
- SAS is a commercial package that is widely used in the department of Statistics at UNL. We will be acquiring access to it so that we can easily exchange data with the Helpdesk and, importantly, use the various functionalities that they are implementing for us.
- R is a public-domain package, recommended by Dr. Hogler Hoos. It is implemented on `gubbio.unl.edu`, the PC in the lab.
- Matlab is a commercial package, which has a statistical library to be purchased in addition to the basic package. We have purchased a Matlab license for `gubbio.unl.edu` and the department has purchased a Matlab license in `cse.unl.edu`.

6.2 Online pointers

Jeremy Frank recommends the following excellent write-up:

`http://faculty.vassar.edu/lowry/VassarStats.htm`.

It is a good idea to test what is the shape of the time distribution of your algorithms. The more you know about this, the more precise statistic tests you can use. We found the following sites useful to build quickly an 'eye' for the various statistical distributions:

- http://www.tufts.edu/ gdallal/npar.htm
- `http://www.stat.vt.edu/ sundar/java/applets/Distributions.html` (really cool animations).
- `http://www.itl.nist.gov/div898/software/dataplot/`.
- `http://www.itl.nist.gov/div898/software/dataplot/distribu.htm`.
- `http://www.math2.org/math/stat/distributions.htm`.
- `http://astronomy.swin.edu.au/ pbourke/analysis/distributions/`.

6.3 Statistic helpdesk

The Department of Statistics offers statistical assistance through its Help Desk, run by graduate students who consult with their advisors. We strongly encourage you to contact them and discuss with them your experiments and the validity of the statistical tools you are using. They are located in 156 Hardin Hall (33rd & Holdrege), phone: 472-5405. This is located on East Campus. You can find their hours posted on their department web-site, under "Statistical Consulting, Help Desk" `http://statistics.unl.edu`. The advisor assigned to the consulting class is Dr. Erin Blankenship. The graduate student we have been working with is Bradford Danner (bradforddanner@yahoo.com). (During Fall 2004, his hours are Mon/Wed 8:15–9:45 and Wed 11:45-14:45.) As of Fall 2004, they do not have office hours on City Campus. However, they may do so in Spring 2005, and if they do, the sessions will be in Avery Hall.

6.4 Courses at UNL (Department of Statistics)

The statistics department offers the following courses for researchers:

- STAT801 *Statistical Methods in Research.* It is a 4 credit course (3 hours lecture, 2 hours lab). Statistical concepts and statistical methodology useful in the descriptive, experimental, analytical, and interpretative study of biological phenomena. Data summarization, probability and basic distributions, hypothesis testing, t-tests, analysis of variance, regression and analysis of covariance are discussed. Emphasis is placed on application and understanding of statistics and relevance to the biological problem. Offered every semester.

  Prerequisites: STAT 218 or permission.

- STAT802 *Experimental Design.* It is a 4 credit course (3 hours lecture, 2 hours lab). Suitability and efficiency of various designs in conducting experimental investigations in agriculture and related areas and the statistical analysis of the data.

  Prerequisites: STAT 801

6.5 Frequently encountered errors

Here is a list of issues that arose frequently:

- Make sure that the graphs of the CSPs you are testing are all connected.
- It you are using Lisp and running repeated evaluations make sure that:
  1. You are forcing garbage collection before each test (does not hurt to start from a clear slate). In ACL, use the command `(excl:gc t)`.
  2. You are not storing data from one experiment to the next: your data structures may become exceedingly heavy, thus slowing your memory management. 'Flush' the content of your global variables between every two tests.
- When you are running experiments in parallel, be careful not to have all programs try to output results to the same file.
- Sometimes, counting some measurements may slow your algorithm and increase the CPU time. You may need to run your tests twice: one run for collecting the CPU time, and the other run for collecting all other measurements. Try to collect as many measurements as you can, even those that do not seem important, so you do not have to re-run your experiments over and over again in case you find out that you need the values of some metrics that you did not save.

References

[Hoos and Stützle, 2004] H.H. Hoos and T. Stützle. *Stochastic Local Search Foundations and Applications.* Morgan Kaufmann, 2004.

[StatSoft, 1998] StatSoft. Comparing Samples. Web link www.statsoft.com/textbook/stsurvan.html#comparing, 1998.