Chapter 5

Designing Analysis-Friendly Experiments

> Really, the slipshod way we deal with data is a disgrace to civilization. –M. J. Moroney, *Facts from Figures*

Data analysis textbooks are chock full of advice for recognizing and coping with problematic data sets. For example a collection of time measurements may be *censored* if the experimenter halts trials that take longer than a fixed time limit; and special analysis techniques for estimating means can be applied to censored data. The alert data analyst should recognize problems and choose analysis techniques most appropriate to the data set. Chapter 7 discusses data analysis strategies relevant to typical problems in experimental algorithmics.

Most of the textbook advice starts with the premise that the experimenter and the data analyser are two separate people – consultant and client, perhaps – and that the former has little control over the slipshod methods of the latter. This is not the case in algorithmic experiments, where experimental design, data collection, and data analysis is an iterative process carried out by one person (or perhaps a small team). This chapter considers strategies for avoiding problems in data sets *before* the data is collected.

Figure 5.1 illustrates the difference between good and bad data in this context. Panels (a) and (b) show results of two experiments to study a hypothetical cost function that depends on problem size n. In statistics lingo, n is called a *factor* that can be manipulated in the experiment. The observed costs $C_i(n), i = 1 \ldots t$ are shown for t random trials at each of several *levels* of n shown on the x-axis. The dotted lines connect the sample means $\bar{C}(n) = \sum_{i=1}^{t} C_i(n)/t$ at each level. Suppose the purpose of the experiment is to determine whether mean cost grows linearly in n. Which data set would you rather analyze?

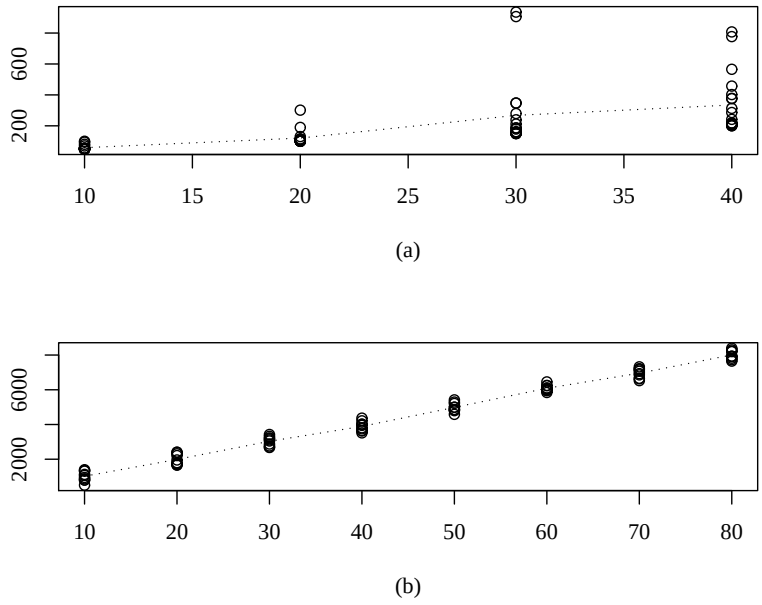

Figure 5.1: Good design vs bad design

You chose (b)of course. The underlying cost function is the same in both panels: $C_i(n) = n + 100 \log_2 n + \epsilon_i$, where the error term ϵ_i is a random uniform in the range $(0, 100)$. But different choices about number and spacing of levels for those factors, and the number of random trials to test, made a difference in the "analyzability" of the results. In general, a good data set shows high *response* (change in mean) with respect to changes in levels, when compared to the *variation* in data within each level.

This chapter surveys ideas for improving the analyzability of results from algorithmic experiments. Of course, the analysis tail must not wag the experimental dog: the experimenter may have good reasons for developing experiments that are challenging to analyze, and much lies beyond the control of the experimenter. Nevertheless, algorithmic experimenters enjoy unusual opportunities for exploiting features of computation to create experimental designs and test programs that produce better data, more efficiently.

The collection of decisions about factors, levels, and trials to use in an experiment is called an *experimental design*. The next section presents an informal survey of insights from the field of *Design of Experiments* (DOE), which is concerned with finding optimal designs to answer specific questions about a given data set under certain formal assumptions. Although it can be

argued that the formal assumptions and questions developed in DOE do not often match the realities and motiviations in algorithmic research, a basic understanding of methods of DOE can be useful.

Section **??** discusses strategies from the field of Simulation, for applying *Variance Reduction Techniques* by adjusting the test program to produce output data with lower variance than would occur in the "obvious" experiment. Section **??** presents a related approach called *Simulation Shortcuts* for generating more data per unit of computation.

No great experiment springs wholly formed in the mind of the scientist. The assumption throughout is that the experimenter already has a general idea of what to expect from a given experiment – how much time it will take, how much variance is in the data, what the output distribution looks like, and so forth. That general idea must be developed from previous experiments, whether reported in the literature or from an earlier round of an ongoingd research project. The best experimental research is iterative, since more knowledge about the subject produces better experimental designs, which in turn produces more knowledge. If your algorithm and your experiment are truly *de novo*, consult Section **??** which describes the pilot study and its uses.

This is by no means a complete treatment of these topics, which would require at least a separate text to cover in detail. See Cohen [3], for an excellent survey of DOE and data analysis, with many examples drawn from computational experiments in Artificial Intelligence. Simulation techniques are discussed by Bratley, Fox, and Schrage [1] Kleijnen [**?**], Law and Kelton [9], and Ripley [10].

5.1 Experimental Design Basics

Factors come in two flavors: *numerical* factors, like $n = (10, 20, 30)$ have levels that can be placed along a number line, while *categorical* factors, like D = *(binary search tree, splay tree, hash table)* have no numerical interpretation. A specific combination of level assignments such as $(n, D) = (30, hash\ \ table)$ is called a *design point* or *sample point*. An algorithmic experiment typically involves some number of random *trials* measuring perfomance at each sample point in the experimental design.

In algorithm research we recognize three categories of parameters that can affect performance:

- *Algorithm parameters* are associated with properties of the algorithm(s) being evaluated. Some are numerical and some are categorical. For example Quicksort may be parameterized by s, the size of the random

sample used for selecting a partition element; or Dijkstra's algorithm may be parameterized by the choice of data structures used in its implementation.

- *Input parameters* are associated with the inputs presented to the algorithm. A set of instances can be described by parameters along two dimensions:
  - *Input class* refers to the source of a group of inputs: they may come from real-world applications, or be randomly generated, or constructed by hand. Input classes are nearly always categorical.
  - *Instance parameters* refer to properties of individual instances. Problem size n (and m or k as appropriate) is nearly always considered to be an important instance parameter. Other parameters may be important to, say, worst-case vs best-case performance within a given problem size. Instance parameters are nearly always numerical.
- Finally, *environment parameters* are associated with properties of the compiler, operating system, or platform, on which the test program is run. Some environment parameters are categorical, like OS = (Linux, Windows); some are numerical, like instruction cycle time or compiler optimization level.

The first step in planning an experiment is to decide which parameters from these categories get promoted to factors, that is, which parameters are to be explicitly manipulated in the experiment. Good experimental design starts with a clearly and precisely stated research question, which naturally suggests some factors. For example the factors in the following questions could be recognized even without the helpful underlining: How does algorithm cost depend on problem size? What is the effect on time and solution quality for my simulated annealing algorithm when I change the stopping criteria and the cooling schedule? How does graph algorithm A respond to changes in graph density and the distribution on edge costs? A good rule of thumb is to select factors that are most important to performance; i.e. choose those parameters that, when manipulated, produce the greatest response in performance indicators.

It is just as important to decide what happens to parameters that do not become factors. As we shall see, sometimes the experiment is too big and the main question is which factors to eliminate from a design; sometimes an interesting factor is simply too hard to control. Two alternatives to manipulating a factor are either to *fix* it, by holding the level constant throughout the experiment, or to treat it as *artifactual* by allowing levels to

vary in some uncontrolled or semi-controlled way from trial to trial.[1] Semi-controlled often means *randomized,* or controlled by another factor together with a random generation scheme. For example, suppose a random graph $G_{n,p}$ is generated with n vertices and probability p that any given edge is present. The actual number of edges m in a particular instance is a random variable – an artifact of the generation scheme – that depends on the factors n an p. If the value of an artifact is not reported during the experiment, it becomes a source of random variation, aka experimental noise.

Note that while it is preferable to control important parameters as much as possible, good information may be gleaned from an experiment when parameters are too difficult to control, and must remain artifactual. If the parameter is randomized, it may be possible during data analysis to exploit properties of the generation method – for example that m in the above example has known expectation. Or, suppose an input testbed contains six instances drawn from some real-world application: instance size n could be a considered a factor if you deliberately chose three small and three large instances from a larger space of possibilities. But if those six are the only instances available, it is difficult to claim much control over n. Whether or not problem size is explicitly controlled, the data analysis problem is the same: to understand the relationship between n and algorithm cost.

The other option is to fix a parameter to a constant level throughout the experiment. Fixing a parameter eliminates a source of experimental noise, and therefore reduces variance, which may have the effect of sharpening the response of performance indicators to the actual factors. For example, it is easier to compare two algorithms A and B using exactly the same (fixed) set of test instances, on exactly the same platform, rather than allowing those parameters to vary between trials.

On the other hand, fixing a parameter narrows the scope of the experiment. This has different consequences depending on parameter type. For example, it is safe to assume that the "reader" of your experimental results, whether practioner, algorithm engineer, theoretician, or fellow experimenter (see Section **??** for their descriptions) has access to a similar algorithm, perhaps even the same code, as the one used in your experiment: they can manipulate the same algorithm parameters and choose the same levels as you. However the reader usually does not have the luxury of selecting input classes and instance parameters to match your experimental design: the

[1] A statistician would use the terms *extraneous variable* to refer to parameters that are not factors, *control variable* instead of "fixed parameter", and *random variable* instead of "artifact." The discussion here uses nonstandard terminology because because the usual statistical terms (especially variable, control, and random) have quite different connotations for computer scientists and are also used (with computer-science meanings) throughout this section.

practitioner, is stuck with inputs from her particular application, and the algorithm engineer cannot tell his clients which inputs to use for best results. Finally, neither you nor the reader has much choice about environmental parameters such as platform and operating system, although you both have some control over parameters like compiler optimization level and system load. In areas where the reader can't match your experiment, it becomes more important to design experiments that support generalization beyond the narrow scope of your sample points.

Some parameters are easier to generalize than others. Categorical parameters are especially problematic: as a general rule, there is no way to extend results for, say, implementation strategies A and B (or input classes A and B, or platforms A and B), to some untested level C. In some cases it may be possible to argue that level A produces, say, worst-case performance, thereby giving a bound on the general case; in some cases it can be argued that the large collection of categorical levels in your experiment spans the set of possibilites and therefore indicates the range of possible outcomes. But the ideal kind of generalization involves, at least, an interpolation line between two levels A and B. For this to be possible, A and B must be placed on a numerical scale.

As the next section illustrates, the problem in some algorithmic studies is how to *reduce* an unwieldy expermental design by eliminating some factors and levels. A good rule of thumb is to choose as factors, those parameters that are most important to performance; fix those parameters that are clearly not important to performance, randomize (and record) any parameter that is too hard to control but is likely important to performance. The last decision about which of remaining parameters to fix and which to make artifacts, depends on the type of research question being asked, and the intended audience for the experimental results, and the resources available to the experimenter.

Avoiding spurious result. We conclude this section with a friendly warning to *beware spurious results.* Spurious results are due to faulty reasoning about cause and effect: you fail to incorporate an important factor into the design, and mistakenly attribute results to other causes, or else you think a factor is important when it is not. While it is conceptually easy in algorithm research to make a list of every possible parameter that could affect performance, even computer scientists can be fooled. Consider:

- *Ceiling and floor effects* occur when the experimental outcome is so close to its maximum (or minimum) possible value that the experiment can't be used to distinguish the effect of different factors or levels. An

example from research on heuristics is to select input classes that are too easy for the algorithm(s) being tested. You may conclude that your sophisticated heuristics all have excellent performance, but fail to notice that a simple algorithm would perform equally well on the instance class. Another example is choosing a stopping time for an iterative-improvement algorithm that is too short to reveal interesting performance – or choosing an initial solution that leaves no room for improvement – and concluding that the algorithm is ineffective.

- *Order effects* occur when you fail to recognize that the order of experimental events affects the outcome. Sometimes the order in which trials are run affects performance measurements: caches can be hot or cold, for example, or the time of day may affect machine load and therefore CPU time (see Section **??** for details). Sometimes the order in which a single input instance is read into the algorithm has a greater effect on performance than different instances. Order effects can be especially problematic in experiments on parallel and distributed systems, since the experimenter has very little control over event order at the scheduling level, which can be critical to performance.

- Not recognizing the difference between performance and the performance indicator. Or between the factor and the controlled variable.

- Inadvertant sampling bias. (Intentional sampling bias can be good.)

- Regression effects: measurements tend toward the middle, away from extremes. Run random algorithm A on instances $I_0 \ldots I_k$. It fails on a subset of instances. Tune a parameter, and run again on the failure set: get a better score. Conclude that the paramter is responsible. No, it could be due to randomness of the algorithm, run a second time turns bad scores into good scores. This is the phenomenon known as regression toward the mean: big numbers are likely to be small next time.

Once the set of factors has been selected, and appropriate decisions made about the dispensation of remaining parameters, the next step is to choose of design points by assigning combinations of levels to factors. A simple *factorial design* is the first and best choice in many scenarios. The following section describes this design, discusses its its merits and some drawbacks, and considers some alternatives.

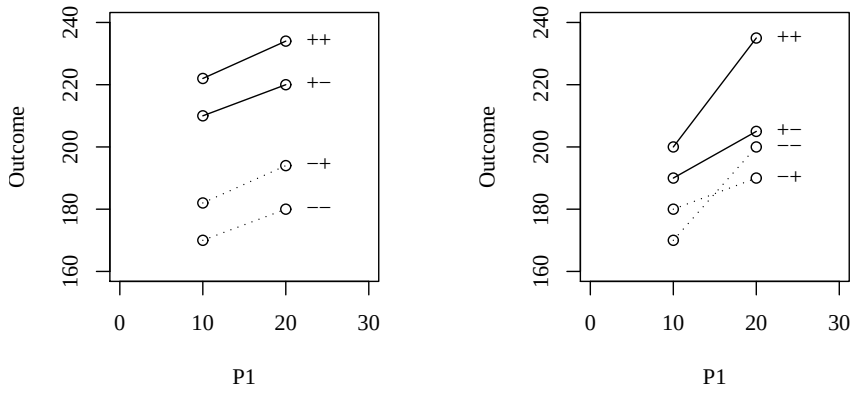

Figure 5.2: Main effects and interaction effects, from a hypothetical experiment.

5.1.1 Factorial designs.

In the classic 2^k full factorial design each of k factors is assigned two levels representing "high" and "low" values, and all 2^k design points, exercising all combinations of levels are measured in the experiment. Following statistical convention, we represent high and low levels with + and -, respectively, and write out the design for $k = 3$ (for a hypothetical algorithm with three parameters), using eight design points, in the table below. (Presumably several random trials would be run at each design point.)

Parameter	Experiments							
	1	2	3	4	5	6	7	8
P_1	-	+	-	+	-	+	-	+
P_2	-	-	+	+	-	-	+	+
$P-3$	-	-	-	-	+	+	+	+

The main advantage of a full factorial design is that is the most efficient way to study the *main effects* of individual factors as well as *interaction effects* of pairs (and triples and higher combinations) of factors.

Figure 5.2 illustrates these two types of effects using data for our hypothetical algorithm. (For simplicity we just show the results of one random trial per design point.) Panel (a) plots the cost $C(P_1, P_2, P_3)$ against P_1 (the values 10 and 20 represent + and - levels). The four lines in each panel are labeled according to P_2 and P_3 levels. We observe that increasing factor P_1 has a positive effect on the outcome C, because every line has positive slope: this is a main effect because the slopes are not much changed by the other parameters. We can also observe that P_2 has a positive effect on outcome because both solid lines are above the dotted lines; this is a main effect due to P_2. The main effect of P_3 appears to be negligible.

Panel (b) shows a different outcome for the same design points. In panel (b), P_1 has generally positive effect on C, but the magnitude of that effect (the slopes) is greater when the level of P_2 matches that of P_3, and smaller when P_2 has different level than P_3. This is an example of an interaction effect.

A less-than-full factorial design must omit some observations: the danger is that the omission will produce incorrect conclusions. For example suppose P_1 has large positive effect when $P_2 = +$ and large negative effect when $P_2 = -$. If P_2 is not a factor, but instead allowed to vary randomly, the interactions will average out to give the appearance of no effect due to P_1: this would correspond to observing only the averages of the solid and dotted lines in panel 5.2 (b). If $P_2 = +$ were fixed then only the solid lines would be observed. Obviously the conclusions drawn from partial designs can produce quite different conclusions about the effect of P_1 on performance.

Another hazard arises from the use of only two levels for each factor. A 2^k design assigns two levels to each parameter on the assumption that measurements at two levels are enough to determine the slope of the *presumed* straight line connecting them. This presumption need not be strictly true, just adequate for rough comparisons of effects. But if the true relationship between a factor and a response is far from linear, the conclusions drawn could be more a reflection of the experimental design than of the underlying relationship.

More generally a 2^3 factorial design reflects an assumption that cost C can be adequately modeled by a linear combination of main and interaction effects, with an error term that is a source of random variation, like this:

$$C = aP_1 + bP_2 + cP_3 + dP_1P_2 + eP_1P_3 + fP_2P_3 + gP_1P_2P_3 + h + err.$$

The full factorial design is intended to be used to analyze the coefficients $a \ldots h$ under formal assumptions, for example that the error terms are normally distributed, independent of levels, and have constant variance througout. For example we can form an estimate for coefficient a by splitting

the design points into those with $P_1 = -$ and those with $P_1 = +$, computing the mean cost in each set, and using the means at those two sets to estimate the main effect of P_1. This is analagous to finding the average slopes of the four lines in panels (a) and (b) of Figure 5.2. When the formal assumptions hold it possible to not only analyze the coefficients, but to analyze the quality of the analysis by producing, for example, error bars.

If it is suspected that a simple line is inadequate to describe some main or interaction effect, it possible to modify the terms of cost function C and to add more levels to factors, as needed. For example if the main effect of P_1 is better described by a quadratic function $aP_1^2 + bP_1 + c$ then three levels of P_1 are needed to find coefficients for that effect. Note that the number of levels depends on the number of *terms* in the function, not necessarily on its degree: the function $an^2 + bn + c$ needs three levels, but the function $an^2 + b$ needs only two levels to find coefficients.

Note that if the real goal of the experiment to pin down that functional relationship, a simple factorial design is probably not the right tool for the job. Section **??** contains advice and references on developing experimental designs for analyzing nonlinear relationships between factors and performance indicators.

The main difficulty with full factorial designs, of course, is that the number of design points grows exponentially with the number of factors. Computational experiments can push the limits by generating masses of experimental data within a few seconds, but nothing really escapes the tyranny of exponential growth. Section **??** discusses strategies for cutting down the number of factors in tests of the IG algorithm.

The remainder of this section considers experimental designs to address two common problems in experimental algorithmics. The first, addressed in Section **??** is how to cope when there are too many potential factors and levels to be accomodated in a single experiment. Next, Section 5.3 considers strategies for experimental design when the goal is to extend results to larger problem sizes n. Both questions are illustrated with a particular heuristic algorithm which is described first in Section 5.3.

Case Study: Iterated Greedy for Graph Coloring. To make the discussion concrete we consider an heuristic for the Graph Coloring Problem called Iterated Greedy. The original algorithm was developed and implemented by Culberson and Luo [5], [4]. The code for IG (and several other graph coloring heuristics) can be downloaded from `www.cs.ualberta.ca/~joe/Coloring/`. This algorithm was one of several evaluated in the the Second DIMACS Challenge on Graph Coloring [**?**], [**?**].

A simplified version of Culberson and Luo's IG algorithm, which elimi-

nates several options available in the original), is available for downloading from *AlgLab* [**?**]. This streamlined implementation has a different user interface and a much less sophisticated back end, intended to make experimentation and code modification easier. The AlgLab implementation (IGA) is described here.

Let $G = (V, E)$ be an undirected unweighted graph containing n vertices and m edges. A *coloring* of G is an assignment of colors to vertices so that no two adjacent vertices have the same color. Colors are numbered $1, 2, \ldots k$. The *color count* is the number of distinct colors in a particular coloring of G. The Graph Coloring Problem is to find a coloring of G with minimum color count. The minimum possible color count for G is called G's *chromatic number* and denoted χ. Graph Coloring is NP-Hard, and arises in many application problems; therefore many approximation algorithms and heuristics have been developed for this problem.

The simple Greedy (GR) algorithm iterates through the vertices, assigning to each vertex $v_i, i = 1 \ldots n$ the lowest-numbered color that does not conflict with previously-colored vertices. It is easy to see that the number of colors used by GR depends on the order in which the vertices are encountered, and that there must exist a vertex order that causes GR to find an optimal coloring.

Figure 5.3 shows pseudocode for the IGA algorithm inspired by Culberson and Luo's IG algorithm: this is an iterative-improvement heuristic that uses GR inside the main loop. Starting with an initial coloring specified by parameter `Pinit`, the algorithm recolors G at each iteration after reordering the vertices according to some reordering rule R. The rule is selected at random from a set of options according to probability distributions specified by the user with parameters `Pcolor` and `Pvertex` (these basically correspond to primary and secondary sort keys, which are selected independently). If the new coloring has the smallest color count found, it is saved as best.

Note that is a property of the reordering rules implemented in IG that color counts can never increase. For each coloring the algorithm also calculates a "color score" that incorporates more information than just the color count; color scores can increase or decrease at each iteration. If `Pretry` iterations have occured with no improvement in the color score, the algorithm reverts to the previous best coloring; if `Pmax` iterations have occurred with no improvement, the algorithm halts. The implementation also allows the user to specify a target color count with `Ptarget`. If the algorithm finds a coloring with this target count, it stops. Figure 5.4 gives a handy list of the parameters that control this algorithm, and their allowed values.

The IGA algorithm bristles with parameters, both categorical and numerical. (Even so, it is simpler than IG which incorporates more parameters

```
G = Input.graph()
G.color(Pinit)
best = G
iter = 1;

while (iter <= Pmax and G.colors >= Ptarget ){

     oldscore = G.score

     R = Select.reorder.rule(Pcolor, Pvertex)
     G.sort.with.rule(R)
     G.recolor()

     if (G.colors < best.colors)
           best = G

     if (G.colorscore() < oldscore)
          icount=0
     else icount++

     if (icount > Pretry) {
           G.color(best)
           icount = 0
     }
}
print (best)    // best coloring found
```

Figure 5.3: Simplified version of the Iterated Greedy algorithm. This version accepts five parameters: Pinit, Ptarget, Pcolor, Pvertex, and Pretry.

String Pinit. (String). Use initial coloring from a named file; if no file is specified use the trivial coloring (each vertex a different color).

int Ptarget. If the target color count is achieved the program terminates.

int Pmax. If Pmax iterations occur with no improvement in score the program terminates.

int[5] Pvertexa, int[3] Pvertexb. There are five *vertex re-order rules* for sorting vertices according to their current color. The user specifies a vector of five weights that determines the probability that each rule is selected. Both primary and

int Pretry. If the specified number of iterations occurs with no improvement in color score, the program reverts to the best coloring previously found.

Figure 5.4: Algorithm parameters, and their types, for IGA.

and more combinations of options.) Let's consider designing an experiment to learn which algorithm parameters are most important to performance. For simplicity let performance be measured by the color count C in the best coloring reported by the algorithm at the end of its run.

Coping with too many factors. We start by considering a *full factorial design*, the first choice of statisticans everywhere. In general full and partial factorial designs can be used to answer a variety of questions that take the form of tests of hypothesis, estimations, or comparisons of two two or more data sets; see Chapter 7 for details. For example: Does the initial coloring `Pinit` have any effect on performance? What is the value of C at various levels of `Pmax`? Which of several combinations of `Pcolor` and `Pvertex` produce the best performance in general? These questions always involve at least one algorithm factor which is assigned to some number of levels.

Of course these questions must be asked with reference to some particular collection of inputs. Even though algorithm parameters are of primary interest, varieties of input classes and instance parameters should be included in the experimental design.

Participants in the DIMACS Graph Coloring Challenge [**?**] created a testbed of 31 graph instances for Graph Coloring algorithms. The testbed contains instances from eight different sources, including:

- (1,2) real-world inputs from two different applications known as *register*

allocation and *school scheduling*;

- (3) instances constructed by hand with known optimal solutions hidden in the graphs;
- (4) instances constructed by hand to be "hard" for certain algorithms;
- (5) simple random graphs;
- (6,7,8) structured random inputs generated by three different methods. One method generates bipartite subgraphs that are connected with random edges; the two others are based on distance cutoffs among vertices placed randomly in the unit square.

In this testbed n ranges between 125 and 4000, and m ranges from 209 to about 4 million. We shall treat n and m as artifacts since these levels are scattered unsystematically within the testbed. Since solution quality does not depend on platform, we do not need to consider environmental factors. Therefore our initial design contains five algorithm factors and an input class factor with eight levels, comprising 31 inputs distributed among those levels.

Unfortunately significant difficulties arise when we try to limit the algorithm factors to just two levels apiece. `Pcolor`, for example, is specified by a vector of weights: referring to the six rules by name as (reverse, random, largest, smallest, increase, decrease) the vector (0, 100, 50, 0,0,0) specifies that the `random` rule is selected with probability 100/150 and `smallest` with probability 50/150 at each iteration of IG. It is impossible to narrow the space of interesting levels to just two. Some strategies for reducing `Pcolor` (and `Pvector` to at least a finite number of levels are listed below:

- When scaled to probabilities, the space of values for `Pcolor` forms the surface of the 6-dimensional unit simplex (because the probabilities sum to 1). We can divide that surface into "regions of interest" by reasoning about algorithm structure. For example, the simplex vertices (like (0,1,0,0,0,)) are interesting because each rule is applied exclusively; edges correspond to rule combinations with some 0 values; and so forth. Choosing one level on each of 6 vertices, 12 edges, and 60 faces of the 6-d simplex creates 78 levels.
- Choose, say, *high=1* and *low=0* levels for each rule independently. When scaled to probabilities, this generates $63 = 2^6 - 1$ levels (since (0,0,0,0,0,0) is not allowed).
- Generate some convenient number L of probability vectors uniformly and independently from the space of all such vectors. (Fix these vectors

beforehand for common use in all trials, rather than generating new vectors in each trial.)

- Rely on previous work suggest on good combinations and ignore uninteresting choices. For example, Culberson and Luo [5] evaluate several choices of `Pcolor` (and `Pvertex`) on five classes of structured random graphs. They found two particular combinations of `Pcolor`, involving (largest, reverse, random) in proportions of (50:50:30), and (70:50:30) that generally worked as well as any they tested. They also point out that assigning nonzero probability to the random rule allows IG to to avoid getting "stuck" on certain patterns, and that the `largest` and `smallest` rules (for `Pvertex`) are generally ineffective. (Of course the potential hazard of adapting their recomendations to new input classes is obvious.)

Suppose we use some combination of the above strategies to reduce `Pcolor` and `Pvertex` to, say, 10 levels each. The four remaining factors `Pinit`, `Pmax`, `Ptarget` and `Pretry` are also problematic.

`Pinit` specifies a file containing an initial coloring; if no file is specified, the default trivial coloring (every vertex a different color) is used. Setting the "hi" level to "default" is straightforward, but how can we specify a second level that is comparably "low" across inputs, and also specific to each input? Also, `Ptarget` specifies a target color count: if the algorithm finds a coloring with this color count the algorithm stops. Meaningful levels for this parameter range from n down to χ, the chromatic number for the graph, but the chromatic number is known for only 10 of the 31 graphs; setting `Ptarget` to something close to n would produce an uninteresting experiment. What levels would you choose? Finally should `Pmax` and `Pretry` be set to constant "hi" and "low" levels throughout the experiment (possibly causing floor or ceiling effects), or should the levels be scaled to, say, instance size?

Even if we could find two levels each for those three factors, a more serious problem looms: this design requires $49600 = 2x2x2x2x10x10x31$ sample points. Assuming an average runtime around 100 seconds per instance on a Sun SPARC 10/40 we might expect to complete one round of trials in about 51 days. Multiply that by the number of random trials per design point, and conclude that this experiment is clearly to big to be practical. We must consider strategies for further reducing the number of levels, or even better, the number of factors in the experimental design.

One strategy is to *merge similar factors:* If two or more factors can be identified that have a similar effect on performance then treat them as one. For example, `Ptarget` and `Pmax` control the number of iterations of the main loop of IGA. They could be merged into a single factor `Piterations` that

controls the total number of iterations. Both `Pmax` and `Ptarget` could be made artifactual if the program were modified to report the iteration count and color score each time the color count decreases: with that data it is possible to infer what values of `Pmax` and `Ptarget` would have stopped the algorithm earlier. (This is an example of a simulation shortcut called trial overloading, discussed further in section **??**.) Furthermore, if this information about the effect of (hypothetical) stopping rules is reported by the test program, there is no need to set `Piteration` to a "low" level.

Merging two rules into one high level produces an experimental design with $12450 = 49800/4$ design points, which corresponds to about two weeks of computation. Additional strategizing is needed: either trim the design along the lines suggested above, or consider using a smaller design that has been developed by the DOE community to meet certain statistical requirements. Two promising approaches are listed below:

- Create a *fractional factorial design* that, for example, concentrates on main effects and two-way interactions, but omits sample points that higher interactions. A 2^{k-p} fractional design is 2^p times smaller than the full design. Designs for given k and p can be constructed by a straightforward procedure or looked up in published tables: see for example [**?**] (Chapter 12) for more information (in their notation, a Resolution III design preserves 1- 2- and 3-way interactions).

- Use *factor-screening strategies* to identify a subset of factors that have the greatest effect on performance (and randomize the ones that don't) Two well-known strategies are called *Plackett-Burman* designs, and the *controlled sequential bifurcation* (CSB). The latter approach, for example, iteratively partitions a collection of factors into "Important" and "Unimportant" according to outcomes of a formal series of tests. See [**?**] (Chapter 6) or [9] (Chapter 12) for details, or [11] for a tutorial discussion of CSB and related techniques.

The reader who expects this section to end by revealing the final "correct" experimental design for IGA will be disappointed. Although the usual goal of DOE is to find an optimal design for a specific research question under a given set of assumptions, the real world demands compromises and lowered expectations: there is no perfect design for this problem. Or rather, the perfect design can only be obtained with perfect information about the performance of IGA: and if we had that information there is no need to experiment.

Rather, the objective here has to encourage a systematic reasoning approach to the question of experimental design. Readers are invited to down-

load IGA from *AlgLab* and complete this process by developing smaller designs that meet specific experimental goals. Some ideas to consider are: (1) further reduce the number of levels for `Pcolor` and/or `Pvector` (perhaps the secondary sort key needs only 2 or 3 levels); (2) cut the input set from 31 to 8 or 16 instances, with one or two from each class; (3) eliminate some input classes that may be redundant or less informative; (4) make `Pretry` artifactual and modify the code to report statistics that factor.

As mentioned earlier, factorial designs start with the assumption of a linear (or near-linear) relationship between factors and performance indicators. However in computer science a very common research problem is to discover the true underlying function (or at least its first term) that describes that relationship. Factorial designs are clearly not appropriate for those kinds of research questions. The next section considers designs for investigating functions and trends in data sets.

5.1.2 Designs for analyzing functions and trends.

In this section we will consider two experiments intended to investigate the shape of the relationship between parameters and performance: that is, instead of comparing or estimating performance at specific design points, our goal is to study how performance changes as a function of parameter values.

First we consider a classic problem in computer science: to discover the functional relationship between instance parameters n and m, and algorithm performance. The subsequent section considers another common problem: how to assess steady-state behavior and analyze convergence properties of some types of iterative algorithms. We continue to use the IGA algorithm described in Section 5.3, to illustrate the concepts developed here.

These types of questions have received little attention in the DOE literature, probably because DOA was originally developed for field experiments that have quite different properties from computational experiments; consequently the discussion in this section relies primarily on common sense and thoughtful prioritization. The development of well-founded formal designs for these questions remains an intriguing open research area.

Designs for fitting and bounding functions. In this section we consider experimental designs for a central research question of algorithm analyisis: to discover the functional relationship between input size and algorithm performance. The total time $\hat{T}_P(n, m)$ for the IGA algorithm depends on the number of iterations I performed by the algorithm, combined with the the cost of the sorting step plus the cost of the recoloring step at each iteration.

```
G: graph with n vertices v, m edges (v,w)

int[]  V[1 ... n]: array of vertices (already sorted) in order to be recolored
int[]  C[1 ... c]: array ov colors (already sorted) in order to be applied

vertptr[]  VerticesWith[1 ... c]:  VerticesWith[i] is a pointer a linked list of vert

colorptr[] ForbiddenTo[1 ... n]:  ForbiddenTo[j] is a pointer to a linked list of col

for (i = 1 to c) VerticesWith[i] = new colorptr (sentinel);  //initialize with sentin
for (i = 1 to n) ForbiddenTo[i]  = new vertexptr(sentinel);

for (i = 1 .. n)  {      // for each vertex
    vertex  = V[i]

    for (j = 1 .. c)  {    // for each color
        color = C[j]
        if ( vertex.notforbidden(color) ) break;

    }
    // valid color has been found for vertex

    vertex.assign.color(color)
    for (w = vertex.nextneighbor()) w.add.forbidden.color (c)
}
```

Figure 5.5: The Graph Recoloring Algorithm

The implementation uses the C system `qsort` function for the sorting step, which is fairly well understood and stable at each iteration, and will not be further considered here.

In this section we consider experimental designs for investigating the cost of recoloring during iteration. This cost depends on the color count for the current coloring, which we shall denote by c. Let $R_P(n, m, c)$ denote the cost of recoloring a graph with n vertices and m edges, with a current coloring containing c colors. The subscript P refers to a small collection of categorical factors described below. We let the cost of the algorithm be denoted by the number of *vertex touches* performed in the main loop of the re-coloring algorithm, which is sketched in Figure **??**.

The calls to the vertex operations are unit cost. The question considered here is, how many vertex operations are performed by this code as a function

of n, m, and c? It may be obvious that an experimental design to study this cost function requires dramatic reduction in the number of other factors (besides those three). Rather than a complicated heuristic we focus attention on a relatively simple algorithm inside the main loop of that heuristic.

Categorical parameters include the reordering rules used to build C and V: although the cost of sorting is not calculated here, the effect of each reordering policy on color count is of interest. The input classes are similarly restricted. As is often the case, the pool of real instances is too small and too limited to allow manipulation of n and m to the extent necessary. Although in some cases it is possible to create real problems of artificial size by, say, eliminating edges at random, or creating pools of subgraphs comprising random subgraphs of the original. But as a general rule we are interested in scaling up, not down. We shall restrict consideration to two types of generated instances: X and Y.

Also note the factor c is difficult to control, since it depends on the initial coloring. If this is going to apply to IGA, we need to sample colorings that would arise during IGA processing. These are likely not the same distribution of colorings as, say, random initial colorings. (From Culberson and Luo) The coloring step takes $O(nc)$ worst case? time, where c is the expected number of colors assigned. For $G_{n,p}$ the expected value of c is known to be $O(n/\log n)$, so the expected running time is $O(n^2/\log n)$. Not much difference between n and $\log n$ in these cases. You need two levels of n to fit a; you need two values of n and two values of m to find b, and you need

We consider the problems of designing an experimental to fit a convenient *descriptive* function for Reorder cost, and then an experiment to *extrapolate* to find an asymptotic bound on Reorder cost.

An alternative to the full factorial design described in the previous is *response surface methodology* approach to choosing parameters and levels. Suppose the experiment involves two parameters x and y. The idea of response surface methodology is to consider the outcome measurements at sample points to represent numerical samples of an unknown surface, and to fit a function – called a metamodel – to the surface. (The metamodel corresponds to the unknown function that gives rise to algorithm performance.)

This approach can be seen as a generalization of a factorial design strategy to non-linear metamodels. The response surface methodology can incorporate more levels and more complicated functions; in particular the function family may be selected after the data is collected, and it may be partially based on what we already know about the problem. This is more appropriate to this algorithm research problem.

The two common designs are factorial designs or regularly spaced grids.

This can be done in an iterative way: take a 2x2 factorial design, then add more points in the area of greatest slope. There are fancier starpoint designs and center models, which mostly don't apply.

Here are some informal rules for design point placement, based on experience with algorithm research, and listed here without statistical authority.

- The success of an asymptotic analysis depends critically on the size of the largest instance size tested. For example, Figure X shows a plateau effect in color count observed by Culberson and Luo. Although this problem is fundamental and can never be completely avoided, it is always wise to *measure the largest input sizes possible.*

- Suppose the goal is to understand the *mean* behavior of cost c as a function of n. Suppose you have selected a convenient function to fit to the data. Then the best strategy is to concentrate sample points at the number of levels necessary to fit to the number of terms in the function. Two points are necessary to fit a straight line, three points to fit a quadratic function, four points to fit a cubic, and so forth. Note that this rule of thumb correspons to the number of terms in the function, not precisely order. If you want to fit two-term functions like $an \log n + b$, or $an^2 + b$, only two points are need.

- Knowing the asymptotic bound on the leading term is not the same as knowing all the terms. If you know Note that it can be a gravely misleading error to try and fit the high-order term to low-order data that has not yet reached asymptotic behavior. Fitting a curve is not the same as bounding a curve. Figure XX shows an example of this type of mistaken analysis.

- Samples may be evenly spaced, incrementally spaced, randomly spaced, or fixed concentrated within their range. Closedly spaced data points may allow you to detect knees, step functions, phase transistions, and cyclic behavior. If the function is not well understood, do not assume these behaviors cannot exist.

- Some data analysis techniques in Chapter 7 begin by transforming the data: for example, taking logarithms of outcomes. This happens, for example, when the y data spans several orders of magnitude, and high values overwhelm low values on the graph. If you know you will be working with log data, it can be easier in some cases o work with log-log data, i.e. logs of both x and y values. In that case, sample points can be spaced in multiplicative rather than additive intervals. This is often the case in asymptotic experiments.

- The next section describes *shortcut experiments*, for getting multiple samples from one experimental trial. Note that in shortcut experiments, you lose the assumption that i.i.d points are taken. This can affect analysis of variance (and error bars) in the data analysis. If you sample far enough apart, they might look independent enough (or at large enough intervals that it seems independent)? Or should the samples be dependent (this is certainly the most efficient time-wise); but should time series analysis be used then?

- A strategy applicable to some problems is to increase the range of levels and the distance between levels. The variance *between* the sample points increases, while the variance *within* each sample point remains constant. (If you are lucky. Variance might of course increase.) Alternatively, since we are concerned with relative variances *within* versus *between* experimental levels, a successful strategy in some contexts is to increase the distance between levels and/or to expand the range of experiments.

  Unlike the first, this second strategy does not come with a general guarantee that data will be easier to analyze. For example, it may not always hold that sample variance stays constant in n as shown here; if within-sample variance increases with n wider intervals may not improve the situation. Also, in general it may not be the case that the unknown cost function $a(n)$ is "anchored" at $a(0) = 0$ (a this property which increases the importance of large-n observations).

  Unfortunately, even when more trials and greater ranges are justified, there may be practical obstacles to their application. The running time of algorithm A may make experiments with large numbers of trials prohibitively expensive; for most algorithms this problem exacerbated at higher input sizes. Both time and space constraints may limit the size of the largest input that can be examined in the experiemnt.

- Decrease variance by increasing the number of controlled parameters. If you can identify a source of variation in your study, control it, and divide the sample into discrete subsamples with (presumably) smaller means. Figure 6.1 shows what happens when each sample set is divided into two groups according to a parameter X.

Steady states, horizons, and stopping rules. Many self-organizing data structures and some iterative algorithms can be analyzed in terms of steady state performance in an average case model. For example suppose we implement a Dictionary ADT on n keys $k = 1 \ldots n$. A sequence of m

Lookup(k) operations is generated by selecting keys ndependently at random according to some probability distribution P_n.

Each "state" s of the data structure (aka configuration) has an average cost $C(s)$ that depends on P_n. Assuming the data structure starts in state S_0, the probability of state s appearing at time $t = 0 \ldots m$ can be described by a state-probability distribution $\hat{P}_t(i)$ that depends on P_n and the initial state. In many cases the data structure can be shown to converge to steady state behavior, which means that the state probabilities approach an asymptotic distribution $P^*(i)$ that does not depend on initial state or on t that is, $\hat{P}(i, t, S_0) = P^*(i)$ as $t \to \infty$. In some cases the probabilities approach the asymptotic distribution only in the limit; in some cases the data structure achieves steady state within a finite time by losing its "memory" of the initial state. Either way the usual goal of analysis is to find the asymptotic mean cost of the data structure, and to analyze the rate of convergence to that asymptote.

Some heuristic algorithms (for NP-hard problems) create a solution by an iterative-improvement process that can be viewed in a similar way, where the state and the cost of the *solution* takes the place of the data structure in the above discussion. Starting with an initial solution S_0 the algorithm steps to another solution S_1 with cost C_1, and so forth, until some final solution C_f is reported by the algorithm. In some cases the costs C_t are monotonically decreasing over time t; more often, however, the costs may be allowed to increase and decrease.

- The algorithm reaches a stopping point that is easy to recognize, but it takes too long to be practical. An exhaustive search algorithm generates all possible solutions, and stops when it finds the best one. But that takes 15 centuries.

- Algorithms with infinite horizons. The algorithm cost approaches some asymptote and takes infinitely long to do it. We know it gets closer to the asymptote, and want to understand the rate of convergence, when to stop. And what the asymptote might be... how close are we?

- Algorithms that reach an asymptotic steady state. These algorithms can be described, for example, by a table of transition probabilities that approaches some infinite table: the probability of each state occuring approaches a constant. The asymptotic "average cost" of this algorithm can be found by taking means over long runs (after steady state), or by taking means over several independent trials. We want to understand mean cost, or perhaps the envelope of min / max cost over time. Sometimes the time to "forget" initial state is known, some-

times it isn't. Move to front, binary search trees with random insertions/deletions.

- Algorithms, especially heuristic algorithms, that can be understood as wandering around a poorly-understood discrete space that can be respresented by an enormous graph (far too large to be represented in memory.). Each node in the space has a cost, and connections to some number of neighbors. The goal is to find a node with minimum cost. The algorithm may find the minimum, but it has no way of knowing that it has done so. There is no steady state, just some region of the space that it may wander into. How to run it long enough to find a good answer, but to know to give up when it is stuck.

If steady state: you can estimate the asymptotic by sampling the finite, but you plan the experiments differently. Need to go long enough to escape the initial conditions, and also need to be aware of cycle times in the steady state, and go long enough for them. If you can, then initial condition is irrelevant, and you don't need to manipulate it. The obvious strategy for steady state is to make one long run and sample (with batch means perhaps) the steady state behavior, using a batch size larger than the cycle size, once the algorithm has reached steady state.

How do you know when you've reached steady state? Put another way, how much of the initial simulation can you discard, and measure the rest. This is called the *startup problem.*

Suppose the goal is to estimate the steady state mean μ. The problem is, the initial states are likely to make the mean $\sum_{i=0}^{T} C(i)$ biased estimator if, say, initial states are consistently high. For example IG has that property. How many initial states s should be discarded?

This question can't be answered without looking at the data. Some procedures have been suggested for particular simulation problems but there is no reason to believe they apply in general. Here is a heuristic based on graphical methods.

You can contemplate estimating steady state behavior if you have reason to believe that the algorithm actually goes through all states more than once in your tests. The state space is small enough that the *distribution* of costs is reasonable to estimate. If you can sample the distribution, that's probabily good enough.

The oscillateions are likely based on either within trial, or or among independent trials. trials, take batched means. Take $R \geq 5$ independent replications $r = 1, 2 \ldots$ measuring at specific times $t = 1, 2, \ldots$ up to time T Report the batched means at each of T $t = t_0, t_1, \ldots t_T$ iteration times. A

batch mean is the mean of observations. This has the effect of smoothing individual oscilations.

Choose a batch size so that you are prepared to discard a number of initial states that is a multiple (not a fraction) of the batch size. Bigger batches: less variance, and any sequential correlation is lessened, fewer data points, easier to compute, but you may discard more initial states than necessary. Batch size may be suggested by the problem.

If the batches look similar between trials, then you can find the "knee" where they appear to reach steady state. Or you can find the point of convergence where batches appear to converge. Discard the initial states before that.

If you can't find that point of convergence, its because the means of batches (across trials) are not smooth enough: too much variation in the sample. Take smoothed means of the batch means: make moving window of b batch means across trials (or batched means), to smooth out the distribution, in successively larger window sizes. This should have a place where the means go from oscillating on the left, to converging on the right.

Now take the batched means for each independent trial, and plot them. If their means tend to jump around, smooth the batched means. If these tend to oscilate, then apply smoothing to the batch means using a rolling window approach, with overlapping values. Use a big enough window that the data points are monotonic.

If this doesn't work, try again: Double the batch sizes in the data you have, or if you think the algorithm hasn't reached asymptotpia,then longer runs.

Schruben has a test for whether a group of observations appears to have initialization bias. Apply it to the smoothed batch means until it passes the test.

It can't be done if you only have one run. Bratley Fox and Schrage describe four heuristic ideas for judging when steady state has been reached– too week to go into here.

You can't do this without looking at the data.

It is a slightly different question to ask when a bunch of independent runs have likely reached steady state, and when an individual run has reached it, testing from inside the run. The second question does not have any good general answer.

1. Used batched means. Batch size b. If b is large you get less bias from initial values, but it takes longer, and more independence in successive batches. Check that the means of successive batches do not appear to be different over time.

2. Take a big long run. Cut in half, smaller pieces, and check that the batches in small pieces have the same general mean as the batch of big pieces.

3. Fixed batch size vs regenerative batch size. Can base MTF on when it loses memory of first test. But notice this can introduce bias because regeration cost is unusual.

4. Try measurements of variance, from many independent trials.

5.

(2) How do you get a good estimate of confidence intervals? A valid confidence interval is based on assumptions of normality and independence. For sequential sampling to work, need enough samples that the central limit theorem applies; any lack of independence is overcome by some mechanisms; initial condition bias The length of startup time may be unrealistic with respect to practice, though, so maybe you do have to shorten up and consider initial states.

An algorithm may not be a steady state type. For example maybe it is a heuristic search algorithm that keeps improving on a solution, converging towards optimal, stopping when some stopping criteria is reached. This is an example of a finite horizon scenario – it eventually stops.

In understhaind stopping criteria. The goal of the experimenter is different from the practitoner. The practitioner needs a rule that stops soon when the algorithm isn't making progress, and doesn't stop when more progress is possible.

An infinite-horizon scenario the algorithm would get closer and closer to an asymptote. Like a numerical estimation of π. There has to be an artificial stopping criteria imposed.

Specially problematic in algorithms with stopping rules. There is necessarily censored data, if it is an infinite horizion situation.

5.2 How many trials?

How many trials per design points? Fixed sample size vs target confidence interval (sequential sampling).

Analysis is simplified if each sample point has the same number of trials. Some trials may be more expensive than others (big n vs small n). Some may have more variance than others: more trials, or try variance reduction technique or transformations to reduce variance in data. Estimation, and comparisons. You have a data set; you want to estimate means, and variance

or dispersion away from means. You want to compare two or more data sets in this way. One sample point, or more with comparisons.

- Increase the number of random trials taken at each level. Chart (b) of Figure 6.1 shows what happens when experiment (a) is performed with 100 trials per level rather than 10: the variation in sample means at each level is reduced considerably, and the linear growth in n is more clearly observed.

  More generally, this approach is guaranteed by the Central Limit Theorem to improve the accuracy of the sample mean estimate. The rule is, variance in the sample mean decreases as the square root of the number of independent trials: to make variance 5 times smaller, take 25 times more trials. Of course, each experiment slows down by a factor of 25 - this appraoch may not be feasible in all cases.

- Statistical analysis for comparison is easier if you use the same sample sizes in both subjects.

- Variance. If variance grows, will need more samples at higher range. Of course, that's terrible because experiments are more costly at high range of n values. On the other hand, transformation can take care of this, so it may not be a problem.

  This resampling idea is not appropriate for hypothesis testing, since the null hypothesis concerns the population, and the test uses the sample, not the population.

- Skew. The performance indicator has an underlying distribution with Counts and amounts will likely need re-expression by logs. Plan ahead and choose levels with sample points that are incrementing by doubling. Log-log analysis will fix skew and homoescadicty.

- Kurtosis. The data is spread wider than the normal distribution: there are fewer observations near the mean, and more at the extremes. The fat tail problem. Tests of normality don't apply: will have to take more sample points than an assumption of normality gives.

- To magnify response, take input parameters larger apart.

- Number of levels depends on the question you want to ask. Bigger, smaller: two levels.

- Censored data. Cutoffs after 1 hour. The value occurs outside the range of a measuring instrument. Left censor, right censor, interval

censor. Avoid floor and ceiling effects? Stop the experiment when a predetermined number of observations have failed. The remaining observations are right-right censored. Type I censoring. Type II censoring.

- Assumptions of normality and independence. Messes up anova, especially if the performance indicators are correlated. Called "statistical assumptions."

Performance indicators: Algorithm parameters: generalization is unnecessary if you assume the practitioner has access to the same (or similar) source code, and if you measure abstract algorithm properties.

Algorithm parameters: generalization is difficult and imperfect if you measure enviornment-dependent properties like CPU times. Need to know something about instruction counts and how environments scale. An alternative to trying to generalize is to treat different environments as (categorical) factors, and sampling them. Otherwise the scope is limited and you should admit it.

Instances: generalization to other instance classes is well-nigh impossible. Worst-case constructed instances can give partitial generalization in the sense of