COMP 150TW: Paragraph = Issue + Discussion (Samples)

The Engineering Method of Technical Writing

November 8, 2016

Questions:

1. How well does the issue frame the discussion?
   A. Very well
   B. Well enough
   C. Not so well
   D. Not at all well
2. Rank the paragraphs from best to worst
3. About each paragraph: what would you improve?
   - About the issue?
   - About the discussion?

Abby's sample

From Brown et al., "Dis-Function: Learning Distance Functions Interactively," 2012.

> What is needed, then, is a system to bridge the space between the experts and the tools. In this paper we introduce an approach and prototype implementation, which we name Dis-Function, that allows experts to leverage their knowledge about data to define a distance metric. Using our system, an expert interacts directly with a visual representation of the data to define an appropriate distance function, thus avoiding direct manipulation of obtuse model parameters. The system first presents the user with a scatterplot of a projection of the data using an initial distance function. During each subsequent iteration, the expert finds points that are not positioned in accordance with his or her understanding of the data, and moves them interactively. Dis-Function learns a new distance function which incorporates the new interaction and the previous interactions, and then redisplays the data using the updated distance function.

Behnam's sample

Jacobvitz, Hilton, and Sorin, "Multi-Program Benchmark Definition,", ISPASS 2015.

> A notable approach to variable instruction count sampling is the Co-Phase Matrix methodology [19]. This approach is based on determining the behavior of two programs in a co-phase: a period during which their behaviors and contention exhibit a consistent pattern. Assuming two programs paired together, such that the first program has M phases and the second has N phases, one can construct a $M \times N$ co-phase matrix that represents the performance behavior of all co-phases of the two programs. For more than two programs, the dimensionality of the co-phase matrix increases; K programs form a K-dimensional matrix. Each entry in the co-phase matrix is the performance of that co-phase, and it is obtained with detailed simulation. The overall behavior of the multi-program execution is then estimated by a fast analytical simulation which tracks what phase each program is in, looks up the performance characteristics of that co-phase in the co-phase matrix, and then determines how long the co-phase will last (i.e., how long until either program changes phase behavior). The process repeats until the end of the sample of one program is reached, which means this approach is a variable instruction count methodology.

Diogenes's sample

From Amin and Tate, "Java and Scala's Type Systems are Unsound," OOPSLA 2016.

> An entirely different approach is to change the goal of minimization. Whereas traditionally minimization has removed the complex features, one could alternatively minimize to remove the features with the least cross-cutting impact. That is, if a feature is simply another case in a proof, with no effect on the design of the proof, or its guarantees, then remove it. This might significantly reduce the capability of the calculus, but such a reduction does not matter if that capability is not relevant to the question at hand. In this way, the discussion of the work could focus on the most challenging aspects of the language or proof, which the researchers' expertise is most relevant to, rather than the aspects that might easily be recreated by the reader or successor.

Jason's sample

From Batson et al., *Spectral Sparsification of Graphs: Theory and Algorithms*.

> What exactly do we mean by sparse? We would certainly consider a graph sparse if its average degree were less than 10, and we would probably consider a graph sparse if it had one billion vertices and average degree one hundred. We formalize the notion of sparsity in the usual analysis-of- algorithms way by considering infinite families of graphs, and proclaiming sparse those whose average degrees are bounded by some constant, or perhaps by a polynomial in the logarithm of their number of vertices.

Matt's sample

From the book chapter "Beautiful Concurrency," Simon Peyton Jones, 2007.

> But the fundamental shortcoming of lock-based programming is that *locks and condition variables do not support modular programming.* By "modular programming" I mean the process of building large programs by gluing together smaller programs. Locks make this impossible. For example, we could not use our (correct) implementations of `withdraw` and `deposit` unchanged to implement `transfer`; instead we had to expose the locking protocol. Blocking and choice are even less modular. For example suppose we had a version of `withdraw` that blocks if the source account has insufficient funds. Then we would not be able to use `withdraw` directly to withdraw money from A or B (depending on which has sufficient funds), without exposing the blocking condition – and even then it's not easy. This critique is elaborated elsewhere [7,8,4].

Norman's sample

This sample from Don Knuth, with one change, is from page 4 of *The Art of Computer Programming, Volume One: Fundamental Algorithms* (second edition):

> So much for the *form* of algorithms; now let us *understand* one. It should be mentioned immediately that the reader should not expect to read an algorithm as if it were part of a novel; such an attempt would make it pretty difficult to understand what is going on. An algorithm must be seen to be believed, and the best way to learn what an algorithm is all about is to try it. The reader should always take pencil and paper and work through an example of each algorithm immediately upon encountering it in the text. Usually the outline of a worked example will be given, or else the reader can easily conjure one up. This is a simple and painless way to gain an understanding of a given algorithm, and all other approaches are generally unsuccessful.

The original opens slightly differently:

> So much for the *form* of algorithms; now let us *perform* one.

Decide for yourself if you think the paragraph is more about performing algorithms or more about understanding them.

Remy's sample

From "EXE: Automatically Generating Inputs of Death" by Christian Cadar et al., CCS'06.

> The result of these features is that EXE finds bugs in real code, and automatically generates concrete inputs to trigger them. It generates evil packet filters that exploit buffer overruns in the very mature and audited Berkeley Packet Filter (BPF) code as well as its Linux equivalent (§ 5.1). It generates packets that cause invalid memory reads and writes in the udhcpd DHCP server (§ 5.2), and bad regular expressions that compromise the pcre library (§ 5.3), previously audited for security holes. In prior work, it generated raw disk images that, when mounted by a Linux kernel, would crash it or cause a buffer overflow [46].

Xinmeng's sample

From Georgiou et al., "The promise and challenge of high-throughput sequencing of the antibody repertoire," 2014.

> Organism age also influences the antibody repertoire. During early ontogeny, the mammalian adult B-cell repertoire is generated in a predictable developmentally programmed fashion, whereas in advanced age humoral immune responsiveness deteriorates; this phenomenon is referred to as immunosenescence and is thought to be attributable in part to a progressive restriction of the antibody repertoire. For example, among the elderly there is an increased prevalence of autoantibodies and, at the serological level, an increased amount of either a single or a small number of serum immunoglobulins that are produced at a high level by benign outgrowths of one or more plasma cell clones.