AUTOMAN: Integrating Human and Silicon Computation

Daniel W. Barowy
University of Massachusetts, Amherst
dbarowy@cs.umass.edu

Emery D. Berger
University of Massachusetts, Amherst
emery@cs.umass.edu

Andrew McGregor
University of Massachusetts, Amherst
mcgregor@cs.umass.edu

Abstract

While the computational power of modern computer systems has increased dramatically, many tasks that humans perform easily remain difficult or impossible for computers. *Crowdsourcing* provides an enticing opportunity to utilize humans as a computational resource for solving these kinds of problems. However, harnessing this power at scale is difficult. We propose a shift to what we call *crowdprogramming*, an approach that simplifies the use of human labor, thus enabling large-scale crowdsourcing applications. We discuss AUTOMAN, a system we are building to explore this idea.

1 Introduction

Humans possess powerful linguistic, visual, and spatial processing capabilities. These skills make them far better than computers at performing tasks like vision, motion planning, and natural language understanding. These tasks fall into a group sometimes referred to as "AI-complete", and many researchers expect them to be beyond the reach of digital computers for the foreseeable future. Crowdsourcing thus seems like a promising approach to solving these problems.

However, efficient scaling of human resources remains a challenge:

- **Non-uniform abstraction.** Existing crowdsourcing systems, particularly Mechanical Turk, address task management in a simplistic manner. This makes incorporating human labor a challenging task for programmers, who must grapple with problems such as task scheduling, network I/O, budget allocation, and error-handling for every new program.
- **Unpredictable quality.** Human-based computations always need to be checked: worker skills and accuracy vary widely, and they have a financial incentive to minimize their effort. Manual checking does not scale, and voting is inadequate, since workers may agree by random chance.
- **Dynamic work environment.** The time that humans take to perform computational tasks is unpredictable, and has high latency. Hiring more workers can reduce latency by increasing the odds of finding faster workers, but this incurs an economic cost.

2 Crowdprogramming

Inspired by the probabilistic human-assisted Turing machine model, crowdprogramming models a human task as a function with some probability of correctness [1]. This abstraction allows programmers to treat human tasks as ordinary function calls in a traditional programming language, freeing them to focus on application logic. Critical tradeoffs between data quality, time, and cost are delegated to the crowdprogramming layer. Human computations may then be freely intermixed with standard programming constructs to build arbitrarily complex applications.

Such a system requires the following facilities:

- **Automatic task management.** In order to provide a clean abstraction, a crowdprogramming system must not only handle basic processing like the posting of tasks and the collection of answers, but also validation and the rescheduling of failed tasks.
- **Quality controls.** The chief concern of a crowdprogramming system should be accurate results, running the computation until the programmer's desired confidence level is achieved (e.g., $\alpha = 0.05$).
- **Performance optimization.** Human computation suffers from high latency, which can be addressed by increasing parallelism. However, increased throughput must be balanced against accuracy and budgetary constraints.

We are developing a prototype crowdprogramming system called AUTOMAN to explore this idea. AUTOMAN is implemented as as an embedded domain-specific language (Figure 1) in Scala, an increasingly popular language built on the Java Virtual Machine [2]. Scala provides native interoperability with Java, allowing the programmer to take advantage of a vast array of existing software.

AUTOMAN currently targets Amazon's Mechanical Turk crowdsourcing platform, however, the details of any particular backend are abstracted, allowing us to plug-in new platforms as needed. An in-depth discussion of the system is available in our tech report [3].

2.1 Automatic Task Management

AUTOMAN expects crowdsourced functions to return with a nonzero probability of incorrectness. The programmer's chosen quality control strategy informs the AUTOMAN runtime's behavior. The role of the runtime is to bound error and re-run calls (Figure 1) until either the error is minimized or the budget is in danger of being exceeded.

2.2 Quality control

In AUTOMAN, programmers construct questions in a canonical form, which allows the system to decompose compound tasks and apply its validation strategies, based on question type, automatically. Users may always override the default validation strategy at compile time in the event that more sophisticated quality control is required.

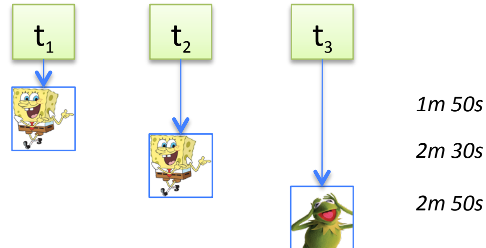
2.2.1 Types of error

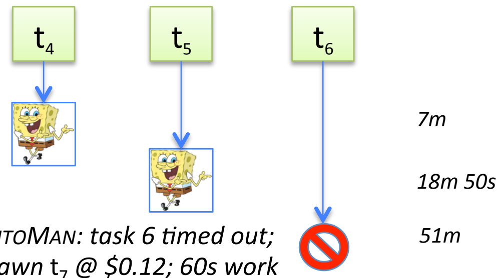
What is a reasonable default strategy given that, as system designers, we know little about the actual task that a programmer may wish to run? Consider the following classification of errors:

1. people who unintentionally answer incorrectly
2. people who intentionally answer incorrectly (i.e., malicious answerers)
3. people who answer randomly

One must also consider that workers are paid for their answers.

```
import edu.umass.cs.automan.core._
import edu.umass.cs.automan.MTurk._

object WhichOneNotBelongSimple {
  def main(args: Array[String]) {

    // AutoMan configuration for MTurk:
    val config = MTurkConfig { c =>
      c.access_key_id     = "XXXX"
      c.secret_access_key = "XXXX"
    }

    // Set up AutoMan parameters.
    val a = Automan { automan =>
      automan.budget = 8.00
      automan.config = config
    }

    // Define a human function.
    val WhichOne = a.Task[String] { t =>
      t.confidence = 0.95
      t.title = "Which one doesn't belong?"
      t.description = t.title
      t.question = a.MultipleChoiceQuestion(
        question_text = t.title,
        selection_texts =
          Map('oscar -> "Oscar the Grouch",
              'kermit -> "Kermit",
              'spongebob -> "Spongebob",
              'cookie -> "Cookie Monster",
              'count -> "The Count")
      )
    }

    // Call the human-based function.
    val fd = WhichOne()

    // Start execution and print result.
    a.run()
    println(fd.value)
  }
}
```





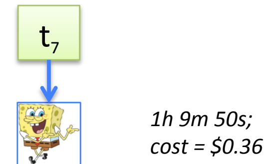


Figure 1: A complete AutoMan program and sample program trace.

Compared to random-answers, malicious answerers are not the most pressing threat. Submitting wrong answers *on purpose* can only be for a single reason: to intentionally thwart a computation, because, on an individual *economic* basis, this strategy is irrational. The same amount of work needs to be done to answer intentionally incorrectly as to answer correctly, but with little possibility for financial gain.

If "easy money" is a worker's desired goal, there exists a better strategy: choosing an answer at random, which is easily automated. People who make mistakes also need to be guarded against, however they are largely indistinguishable from random-answerers. Thus the most important threat is the adversary who attempts to do the least amount of work for the largest amount of gain. Our task is to distinguish random answers from non-random answers.

There is an additional kind of error here, and it's one that we think crowdsourcing is ill-suited to combat: systemic bias. If a particular incorrect belief is widely held by a population, sampling that population will tend to reveal the bias, and increasing the sample size will only tend to confirm that bias. Crowdsourcing cannot fundamentally reveal "truth", rather, it reveals what we call "truthiness."

2.2.2 Default strategy

The default validation strategy employed by our system is based on *preference aggregation*. Tasks are duplicated and an answer is chosen mechanically. The number of replicas is determined with the following procedure.

Given a question with k valid answers, we model the output of n workers as a multinomial distribution where each trial results in one of k possible outcomes and each outcome i has an (unknown) associated probability p_i. By assumption we wish to identify $i^\star = \underset{i}{\text{argmax}}\ p_i$. Our system spawns worker requests until the frequency distribution of the various outcomes is sufficient to determine $i^\star$ with probability at least $1 - \alpha$. If $p_{i^\star}$ is close to 1 we ensure that only a small number of workers are paid. However, if $p_{i^\star}$ is close to $1/k$ we must be careful that the process terminates before spending too much on paying the workers; this limit is presently a configurable paramter.

2.2.3 Combining strategies

Automatic low-level task scheduling allows one to combine strategies in a modular way. For instance, sometimes computational validation of an output is as difficult as producing the output in the first place, thus a programmer may naturally wish to use the crowd to perform the validation. This strategy can be combined with preference aggregation by asking the second set of workers to rate the submissions to the original question. The second set of answers provides data which the runtime can use to automatically select an answer out of the first set.

2.3 Performance

Execution plan abstraction breaks the implicit sequentiality of a program by examining its execution graph, giving control of scheduling a function to the runtime [4]. Only tasks with a functional dependency on other tasks need wait; all other tasks can be executed immediately, and in parallel. Abstracting the execution plan allows for automatic, dynamic error-checking and recovery, as each human "thread" is subjected to validation and possible additional re-computation on an individual basis. Since tasks are parallelized, the runtime is free to perform error recovery while other tasks proceed normally.

3 Related work

TurKit is a scripting system designed to make it easier to manage Mechanical Turk tasks [5]. TurKit adds checkpointing to avoid re-submitting Mechanical Turk tasks if a script fails. TurKit does not fundamentally address quality control, which is the key obstacle to both scalability and efficiently handling the details of human interaction.

CrowdForge is a web tool that wraps a MapReduce-like abstraction on Mechanical Turk tasks [6, 7]. Programmers decompose tasks into *partition* tasks, *map* tasks, and *reduce* tasks. CrowdForge automatically handles distributing tasks to multiple users and collecting the results. Data quality concerns are left to the programmer, and while the MapReduce concept is useful for making parallelism explicit, it is cumbersome to use.

CrowdFlower is a closed-source, commercial web service for crowdsourcing that targets commercial crowdsourcing platforms [8]. As with the approach outlined by Ipeirotis et al., CrowdFlower's approach is to identify likely erroneous workers [9]. Rather than assuming that one can extrapolate future work quality based on a worker's past performance, AUTOMAN addresses work quality directly and with rigorous statistical assurances.

4 Future work

The AUTOMAN prototype supports multiple-choice questions where exactly one of the answers is correct, or when zero or more may be correct. We plan to extend the system to support rankings, free-form questions, and surveys. We also plan to extend the system's support for memoizing intermediate results, ala TurKit, as well as a web service to allow AUTOMAN to monitor a running job. Finally, we plan to enhance Java support for programmers who are more comfortable in a Java-only environment.

References

[1] Dafna Shahaf and Eyal Amir. Towards a Theory of AI Completeness. In *Commonsense 2007: 8th International Symposium on Logical Formalizations of Commonsense Reasoning*. Association for the Advancement of Artificial Intelligence, 2007.

[2] Martin Odersky and Matthias Zenger. Scalable Component Abstractions. In *Proceedings of the 20th annual ACM SIGPLAN conference on Object-oriented programming, systems, languages, and applications*, OOPSLA '05, pages 41–57, New York, NY, USA, 2005. ACM.

[3] Daniel Barowy, Emery Berger, and Andrew McGregor. AutoMan: A Platform for Integrating Human-Based and Digital Computation. Technical Report UM-CS-2011-044, University of Massachusetts, Amherst, 2012. Also available as `http://www.cs.umass.edu/publication/docs/2011/UM-CS-2011-044.pdf`.

[4] Craig Chambers, Ashish Raniwala, Frances Perry, Stephen Adams, Robert R. Henry, Robert Bradshaw, and Nathan Weizenbaum. FlumeJava: Easy, Efficient Data-Parallel Pipelines. In *PLDI*, pages 363–375, 2010.

[5] Greg Little, Lydia B. Chilton, Max Goldman, and Robert C. Miller. TurKit: Human Computation Algorithms on Mechanical Turk. In *UIST*, pages 57–66, 2010.

[6] Aniket Kittur, Boris Smus, and Robert E. Kraut. CrowdForge: Crowdsourcing Complex Work. Technical Report CMU-HCII-11-100, Human-Computer Interaction Institute, School of Computer Science, Carnegie Mellon University, February 2011.

[7] Jeffrey Dean and Sanjay Ghemawat. MapReduce: Simplified Data Processing on Large Clusters. In *OSDI*, pages 137–150, 2004.

[8] Dave Oleson, Vaughn Hester, Alex Sorokin, Greg Laughlin, John Le, and Lukas Biewald. Programmatic Gold: Targeted and Scalable Quality Assurance in Crowdsourcing. In *HCOMP '11: Proceedings of the Third AAAI Human Computation Workshop*. Association for the Advancement of Artificial Intelligence, 2011.

[9] Panagiotis G. Ipeirotis, Foster Provost, and Jing Wang. Quality management on Amazon Mechanical Turk. In *Proceedings of the ACM SIGKDD Workshop on Human Computation*, HCOMP '10, pages 64–67, New York, NY, USA, 2010. ACM.