Programming with Human Computation

by

Greg Little

B.S., Arizona State University (2005)
S.M., Massachusetts Institute of Technology (2007)

MASSACHUSETTS INSTITUTE OF TECHNOLOGY
JUN 17 2011
LIBRARIES

Submitted to the Department of Electrical Engineering and Computer Science
in partial fulfillment of the requirements for the degree of

ARCHIVES

Doctor of Philosophy

at the

MASSACHUSETTS INSTITUTE OF TECHNOLOGY

June 2011

© Massachusetts Institute of Technology 2011. All rights reserved.

Author . . . .
Department of Electrical Engineering and Computer Science
May 20, 2011

Certified by . . . .
Robert C. Miller
NBX Career Development Associate Professor of Computer Science and Engineering
Thesis Supervisor

Accepted by . . . .
Leslie A. Kolodziejski
Professor of Electrical Engineering
Chair of the Committee on Graduate Students

Programming with Human Computation

by

Greg Little

Submitted to the Department of Electrical Engineering and Computer Science
on May 20, 2011, in partial fulfillment of the
requirements for the degree of
Doctor of Philosophy

Abstract

Amazon's Mechanical Turk provides a programmatically accessible micro-task market, allowing a program to hire human workers. This has opened the door to a rich field of research in human computation where programs orchestrate the efforts of humans to help solve problems. This thesis explores challenges that programmers face in this space: both technical challenges like managing high-latency, as well as psychological challenges like designing effective interfaces for human workers. We offer tools and experiments to overcome these challenges in an effort to help future researchers better understand and harness the power of human computation.

The main tool this thesis offers is the *crash-and-rerun* programming model for managing high-latency tasks on MTurk, along with the TurKit toolkit which implements crash-and-rerun. TurKit provides a straightforward imperative programming environment where MTurk is abstracted as a function call. Based on our experience using TurKit, we propose a simple model of human computation algorithms involving creation and decision tasks. These tasks suggest two natural workflows: iterative and parallel, where iterative tasks build on each other and parallel tasks do not. We run a series of experiments comparing the merits of each workflow, where iteration appears to increase quality, but has limitations like reducing the variety of responses and getting stuck in local maxima. Next we build a larger system composed of several iterative and parallel workflows to solve a real world problem, that of transcribing medical forms, and report our experience. The thesis ends with a discussion of the current state-of-the-art of human computation, and suggests directions for future work.

Thesis Supervisor: Robert C. Miller
Title: NBX Career Development Associate Professor of Computer Science and Engineering

Acknowledgments

My ideas are not my own.

Image by Charles Dy. Gears turned with help from Rob Miller, Tom Malone, David Karger, Maria Rebelo, Yu An Sun, Naveen Sharma, Lydia Chilton, Max Goldman, Michael Bernstein, John Horton, Janet Fischer, Chen-Hsiang Yu, Katrina Panovich, Adam Marcus, Tsung-Hsiang Chang, Juho Kim, Jeff Bigham, Haym Hirsh, Michael Sipser, Dave Gifford, Vann McGee, Irving Singer, Mishari Almishari, Matthew Webber, Jessica Russell, Max Van Kleek, Michael Toomim, Niki Kittur, Björn Hartmann, Alex Sorokin, Paul André, Krzysztof Gajos, Liz Gerber, Ian McKenzie, Sian Kleindienst, Philip Guo, Sarah Bates, Dan Roy, Amanda Zangari, David Dalrymple, Athena Dalrymple, Scott Dalrymple, Arghavan Safavi-Naini, Will Leight, Olga Sapho, Terry Orlando, Ann Orlando, Yafim Landa, Chen Xiao, Allen Cypher, Tessa Lau, Jeff Nichols, Neha Gupta, David Alan Grier, Lukas Biewald, Sethuraman Panchanathan, John Black, Goran Konjevod, Sreekar Krishna, Leonard Faltz, Michael Astrauskas, Mark Astrauskas, Meg Howell, Mr. Canaday, my parents, my family, over a thousand Mechanical Turk workers, and many others I forgot to name. Thank you.

This work was supported in part by the National Science Foundation under award number IIS-0447800, by Quanta Computer as part of the TParty project, and by Xerox. Any opinions, findings, conclusions or recommendations expressed in this publication are those of the authors and do not necessarily reflect the views of the sponsors, or of the people mentioned above.

Contents

1 Introduction **19**

1.1 Human Computation 21

1.2 Human Computation Market 22

1.3 Motivation to Program with Human Computation 23

1.3.1 Why Hasn't This Been Tried? 25

1.4 Contributions 26

1.4.1 Programming with Human Computation 27

1.4.2 Modeling and Experimenting with Human Computation Algorithms 29

1.4.3 Putting It All Together 30

1.4.4 Thesis Statement 31

1.5 Outline 32

2 Related Work **33**

2.1 Human Computation 33

2.1.1 Human-based Genetic Algorithms 34

2.1.2 Games With a Purpose 35

2.1.3 Programming with Human Computation 36

2.2 Collective Intelligence 38

2.2.1 User-Generated Content Sites 39

2.2.2 Open Source Software 40

2.3 Human Computation Markets 40

2.3.1 Question/Answer Sites 41

2.3.2 Prediction Markets 41
2.3.3 Outsourcing Intermediaries 42
2.3.4 Micro-task Markets 43
2.4 Programming Model 45

3 Tools **47**
3.1 TurKit Layers 49
3.1.1 JavaScript on Java 49
3.1.2 JavaScript API to MTurk 49
3.1.3 Database 53
3.1.4 Memoization 55
3.1.5 Syntactic sugar for MTurk API 57
3.1.6 Crash-and-Rerun 57
3.1.7 Parallelism 58
3.1.8 Higher-level Utilities 59
3.1.9 External HITs 61
3.1.10 User interface 63
3.2 Using TurKit 65
3.2.1 Iterative Improvement 68
3.3 External Uses of TurKit 69
3.3.1 Decision Theory Experimentation 69
3.3.2 Psychophysics Experimentation 69
3.3.3 Soylent 70
3.3.4 VizWiz 70
3.4 Performance Evaluation 71
3.5 Discussion 72
3.5.1 Limitations 73
3.5.2 Benefits 78
3.6 Conclusion 79

4 **Experiments** **81**
4.1 MODEL 82
4.1.1 Creation Tasks 83
4.1.2 Decision Tasks 83
4.1.3 Combining Tasks: Iterative and Parallel 84
4.2 EXPERIMENTS 85
4.2.1 Writing Image Descriptions 85
4.2.2 Brainstorming 96
4.2.3 Blurry Text Recognition 104
4.3 Discussion 112
4.3.1 Tradeoff between Average and Best 112
4.3.2 Not Leading Turkers Astray 113
4.4 Conclusion 113

5 **Case Study** **115**
5.1 Problem 115
5.2 Solution 116
5.2.1 Phase 1: Drawing Rectangles 116
5.2.2 Phase 2: Labeling Rectangles 118
5.2.3 Phase 3: Human OCR 119
5.3 Experiment 119
5.4 Results and Discussion 120
5.4.1 Phase 1: Drawing Rectangles 120
5.4.2 Phase 2: Labeling Rectangles 121
5.4.3 Phase 3: Human OCR 122
5.5 Discussion 123
5.5.1 Independent Agreement 123
5.5.2 Dynamic Partitioning 124
5.5.3 Risk of Piecework 124
5.5.4 Ambiguous Instructions 124

5.6 Conclusion 125

6 Discussion and Future Work **127**

6.1 State-of-the-Art 127

6.1.1 Quality Control 127

6.1.2 Task Interfaces 130

6.1.3 Basic Operations and Algorithms 131

6.1.4 Human Modeling 134

6.1.5 Low Overhead 135

6.1.6 Bottom-up versus Top-down 135

6.2 Axes of Good and Evil 138

6.2.1 Globalization versus Low Wages 138

6.2.2 Learning versus Cheating 139

6.2.3 Personal Autonomy versus Dehumanization 140

6.2.4 Transparency versus Hidden Agendas 141

6.2.5 Hyper-specialization versus Deskilling 142

6.3 The Future 143

6.3.1 Experts Integrating Human Computation 143

6.3.2 Meta-Experts 144

6.3.3 New Self-Expression 144

6.3.4 Human Computation Algorithms 145

7 Conclusion **147**

List of Figures

1-1 A passage of poor handwriting is transcribed using a human computation algorithm. Workers put parentheses around words they are unsure about. 19

3-1 A TurKit script for a human computation algorithm with two simple steps: generating five ideas for things to see in New York City, and sorting the list by getting workers to vote between ideas. 47

3-2 (a) The TurKit stand-alone interface allows editing of a single JavaScript file, as well as running that file, and viewing the output. (b) The TurKit web user interface is an online IDE for writing and running TurKit scripts. 64

3-3 Time until the first assignment is completed for 2648 HITs with 1 cent reward. Five completed within 10 seconds. 71

3-4 Time and space requirements for 20 TurKit scripts, given the number of HITs created by each script. 72

4-1 This creation task asks for a descriptive paragraph, showing a description written by a previous worker as a starting point. 86

4-2 This decision task asks a worker to compare two image descriptions, and pick the one they think is better. The choices are randomized for each worker. 87

4-3 This decision task asks a worker to rate the quality of an image description on a scale from 1 to 10. 88

4-4 This figure shows all six descriptions generated in both the iterative and parallel processes for two images, along with various statistics including: the number of votes in favor of each description over the previous best description, the average rating for each description, and the number of seconds a turker spent crafting each description. The green highlights show the winning description for each process, i.e., the output description. 90

4-5 This figure shows the output descriptions for both the iterative and parallel process on six of the thirty images in this experiment. 91

4-6 The average image description rating after n iterations (iterative process blue, and parallel process red). Error bars show standard error. As we run each process for additional iterations, the gap between the two seems to enlarge in favor of iteration, where the gap is statistically significant after six iterations (7.9 vs. 7.4, paired t-test $T_{29} = 2.1$, $p = 0.04$). 92

4-7 Descriptions are plotted according to their length and rating, where iterative descriptions are blue, and parallel descriptions are red. A linear regression shows a positive correlation ($R^2 = 0.2981$, $N = 360$, $\beta = 0.005$, $p < 0.0001$). The two circled outliers represent instances of text copied from the internet. 92

4-8 (a) Time spent working on descriptions for both the iterative (blue) and parallel (red) processes. The distributions of times seem remarkably similar for each process. Note that the y-axis uses a logarithmic scale. (b) A histogram showing how many workers spent various amounts of time on task. The distribution appears normally distributed over a log scale, suggesting that work times are log-normally distributed. 94

4-9 A total of 391 unique turkers participated in the image description writing experiment, where the most prolific turker completed 142 tasks. 97

4-10 Turkers are asked to generate five new company names given the company description. Turkers in the iterative condition are shown names suggested so far. 98

4-11 This figure shows all six sets of names generated in both the iterative and parallel processes for two fake company descriptions, along with various statistics including: the rating of each name, and the number of seconds a turker spent generating the names. 99

4-12 This figure shows the highest, middle and lowest rated names generated in both the iterative and parallel process for four fake company descriptions. 100

4-13 Blue bars show average ratings given to names generated in each of the six iterations of the iterative brainstorming processes. Error bars show standard error. The red stripe indicates the average rating and standard error of names generated in the parallel brainstorming processes. (See the text for a discussion of iteration 4, which appears below the red line.) 101

4-14 Gaussian distributions modeling the probability of generating names with various ratings in the iterative (blue) and parallel (red) processes. The mean and variance of each curve is estimated from the data. The iterative process has a higher average, but the parallel process has more variance (i.e. the curve is shorter and wider). Note that in this model, the parallel distribution has a higher probability of generating names rated over 8.04. 102

4-15 Turkers are shown a passage of blurry text with a textbox beneath each word. Turkers in the iterative condition are shown guesses made for each word from previous turkers. 104

4-16 This figure shows all sixteen transcriptions generated in both the iterative and parallel process for the first part of a passage. The original passage is shown at the top, and incorrect guesses are struck out in red. The "combined" row shows the result of applying the merging algorithm. Note the word "Aeroplane" in the very last row. One of the turkers in this parallel process chose the correct word "entertaining", but the merging algorithm chose randomly from among the choices "Aeroplane", "entertaining" and "advertising" in this column. 106

4-17 This figure shows all sixteen transcriptions generated in both the iterative and parallel process for the first part of a passage. The original passage is shown at the top, and incorrect guesses are struck out in red. The "combined" row shows the result of applying the merging algorithm. Note that turkers seem reluctant to change guesses in the iterative case, causing it to hit a local maximum, while the parallel algorithm does very well. 107

4-18 This figure shows the results for the complete passage from the previous two figures, as well as a couple other passages. These examples show cases where both iteration and the parallel algorithm do well, as well as instances where one or both algorithms do poorly. 108

4-19 Blue bars show the accuracy after n iterations of the iterative text recognition process. Red bars show accuracy for the parallel process with n submissions. Error bars show standard error. The overlap suggests that the processes may not be different, or that we do not have enough power in our experiment to see a statistically significant difference. 109

5-1 This is a phase 1 HIT where a turker is instructed to draw rectangles over a form marking areas where handwritten information would appear. Four turkers have already completed a similar HIT, and the fifth turker sees the rectangles that the previous workers have agreed on. . 117

5-2 This is a phase 2 HIT where a turker is asked to label the highlighted rectangle. Subsequent turkers will be shown this worker's label as a suggestion. 118

5-3 This is a phase 3 HIT where a turker is asked to transcribe the given form field. They are also shown the label of the field from the previous phase, in case this helps give context to the transcription. 119

List of Tables

4.1 Time spent working on various tasks, in seconds. All differences are statistically significant, except between iterative and parallel description writing. 95

4.2 Time spent waiting for someone to accept various task on MTurk, in seconds. The notation "first of X" refers to the time spent waiting for the first turker to accept a task in a HIT with X assignments. The starred (*) items form a group with no pair-wise statistically significant differences. All other differences are statistically significant. 95

4.3 Number of unique turkers participating in different task types for this experiment, along with the number of tasks completed by the most prolific turker in each category. 96

4.4 Time spent working on various tasks, in seconds. All differences are statistically significant, except between iterative and parallel brainstorming (two-sample $T_{70} = 1.47$, $p = 0.15$). 103

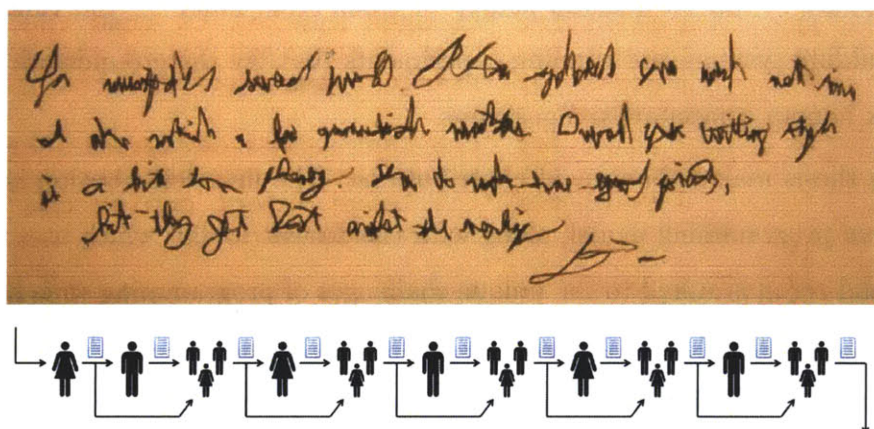
4.5 Number of unique turkers participating in different task types for this experiment, along with the number of tasks completed by the most prolific turker in each category. 104

4.6 Time spent working on the iterative and parallel transcription tasks, in seconds. The difference is not significant (two-sample $T_{382} = 1.26$, $p = 0.21$). 111

4.7 Number of unique turkers participating in different task types for this experiment, along with the number of tasks completed by the most prolific turker in each category. 112

Chapter 1

Introduction



"You (misspelled) (several) (words). Please spellcheck your work next time. I also notice a few grammatical mistakes. Overall your writing style is a bit too phoney. You do make some good (points), but they got lost amidst the (writing). (signature)"

Highlighted words should be: "flowery", "get", "verbiage", and "B-".

Figure 1-1: A passage of poor handwriting is transcribed using a human computation algorithm. Workers put parentheses around words they are unsure about.

This thesis is about programming with human computation. The sequence in Figure 1-1 is an example. The example starts with a passage of practically indecipherable handwriting. This passage is fed into a program to decipher the text. This program is special — it can hire people. That is, it uses *human* computation. The program hires a series of workers in a *human computation market*, namely Amazon's

Mechanical Turk, to make guesses about the words in the passage. The program hires additional workers to vote on the guesses made by other workers. This process, an automated collaboration between human and machine, correctly deciphers 90% of the words.

The human computation algorithm above is just one possible algorithm solving one possible problem. Similar algorithms may solve a wide range of problems. In fact, it may be possible to write sophisticated algorithms that can perform knowledge work tasks like writing or programming as well as experts working alone. This thesis will not go that far, because programming with human computation presents a unique set of programming challenges that need to be understood and overcome first. However, given MTurk — an on-demand source of small-scale labor — the time is ripe to start building systems and running experiments that lay the groundwork for writing complex human computation algorithms.

This thesis makes three main contributions. The first contribution is the *crash-and-rerun* programming model, along with the TurKit toolkit which uses this model. Crash-and-rerun is suited to the unique challenges of programming with human computation. In particular, hiring humans introduces latency. A simple process like the one above can take hours to complete. This is a problem for developers writing a new algorithm, since they may need to rerun the algorithm many times to debug it and figure out how it should work. The crash-and-rerun model helps by memoizing the results from human workers so that they can be replayed without cost or delay the next time a program runs. We implement the crash-and-rerun programming model inside TurKit, and offer an API for programming on Mechanical Turk using this programming model.

The second contribution of this thesis is a set of experiments that explore the tradeoffs between two basic human computation algorithms: *iterative* and *parallel*. Figure 1-1 demonstrates the iterative algorithm, where a series of workers iteratively improve on each other's work. In the parallel version of this algorithm, we would show each worker a blank slate upon which to start transcribing the passage, and we would use the same voting tasks to determine the best transcription. This thesis applies each

algorithm to several problem domains, including describing images, brainstorming company names, and transcribing blurry text. We find that iteration increases the average quality of responses from each worker, but also decreases the variance. We also find that iteration can get stuck in a local maxima, similar to a gradient ascent algorithm.

The third contribution is a case study of a human computation process designed to solve a real-world problem, that of digitizing handwritten forms. We present the design of the process which involves three phases. Each phase is a simple iterative or parallel algorithm. The first phase locates fields on a blank form; the second phase produces labels for each field; and the third phase transcribes the handwritten information in each field of each form. We execute the process on a sample of real-world data, and discuss the results. We also make suggestions for researchers or practitioners working on end-to-end systems of this sort.

The contributions above are motivated by a grander vision of where human computation can take us. It may be possible to take many knowledge work tasks that are traditionally carried out by single experts, and break them down into algorithms which orchestrate the efforts of many individuals. These algorithms may perform tasks more efficiently and reliably than experts, doing for knowledge work what the industrial revolution did for manufacturing. In a sense, each of the contributions of this thesis is a stepping stone toward exploring this possibility.

1.1 Human Computation

When we think of the physical building blocks of computation today, we might think of transistors in a microprocessor, or assembly instructions executed by a microprocessor. In either case the set of basic operations is small, but theorists believe they are sufficient to express all complex and useful algorithms. Of course, we do not always know the right algorithm to use, and many algorithms are too inefficient to be practical. Hence, humans remain better than computers at many tasks, including handwriting recognition.

We can think of human computation as adding an additional instruction: a human procedure call. That is, we add the ability of a program to delegate work or computation to a human. For instance, the algorithm above delegates the task of making guesses about the words in a passage, and the task of assessing the quality of guesses made by other humans. The resulting algorithm may recognize the handwriting more efficiently than either a computer or a human working alone, at least until handwriting recognition software improves significantly.

Of course, delegating tasks to humans is very general. To limit our scope, this thesis will focus on human computation systems where the human tasks are short and well-defined. Most tasks will take the form of web forms that can be completed within 10 minutes. In Figure 1-1, the task of generating guesses involves a web form displaying the handwriting, along with a textbox for typing one's guess. Users are not expected to do it perfectly, and are told to put parentheses around words they are unsure about.

1.2 Human Computation Market

A program is just an idea until we find a computer to execute it on, along with some power to run the computer. If all we want to do is perform simple arithmetic, we might get by with Burroughs' 1892 adding machine, powered by the mechanical energy derived from pulling a crank. These days, computers take the form of microprocessors powered by electricity. Microprocessors are very general computational devices, capable of executing a wide variety of programs.

Human computation programs also need a computer to execute on, e.g., some mechanism to pair humans with tasks, along with some motivation for humans to do tasks. If we want to create an encyclopedia, we might pair humans with tasks using a website that lets anyone add or modify articles, like Wikipedia[1]. People may be motivated to contribute out of an altruistic desire to share their knowledge with the

[1] http://wikipedia.org

world. Unfortunately, altruism is not a very general motivation — not every problem we want to solve directly benefits mankind as a whole.

Luis von Ahn proposed an alternative approach: games with a purpose (GWAP) [56], where people are motivated by enjoyment. For example, the ESP Game [57] generates tags for images as a byproduct of people playing a fun game. In the game, each player is paired with a random unknown partner. Next, both players are shown an image, and asked to guess what word the other player would associate with that image. If they both guess the same word, then they earn points. As a byproduct, the system may use this word as a good tag for the image. Luis von Ahn shows how a number of problems may be decomposed into games [58], [59], but it is not always easy to express a problem as a game. Even if we express our problem as a game, it is not always easy to get people to play our game.

Amazon's Mechanical Turk[2] attempts to solve the generality problem with money. MTurk is a programmatically accessible small-scale labor market. Requesters post short human intelligence tasks (HITs) that they want completed. Workers on MTurk, known as *turkers*, get paid small amounts of money to complete HITs of their choice. Typical tasks pay between $0.01 and $0.10, and include image labeling, audio transcription, and writing product reviews.

This thesis uses MTurk as a human computation market — our analogy to the microprocessor on which to run human computation programs. In Figure 1-1, the writing and voting tasks were expressed as web forms, and posted on MTurk as HITs. The writing HITs offered a reward of $0.05, and the voting HITs offered $0.01.

1.3 Motivation to Program with Human Computation

Given MTurk and similar labor markets, the time is ripe to learn how to program with human computation. Interesting and useful algorithms may be written using small chunks of human effort. In particular, it may be possible to design and execute

[2]https://www.mturk.com/

human computation algorithms that perform certain tasks more efficiently, in terms of time, money and quality, than the current practice of hiring a single expert to do them alone. The algorithm in Figure 1-1, for deciphering poor handwriting, is an example. The handwriting is very difficult to read, and it may be that nobody can read it alone, at least not cheaply.

On the other hand, perhaps we cannot decompose knowledge work tasks in this way. It may be that society has already found the optimal size of task to give people, and that most knowledge work tasks are simply too interconnected to decompose into bite-sized chunks. However, there are several reasons to believe it may be possible.

First is the principle of "many eyes." Eric Raymond proposes that "given enough eyeballs, all bugs are shallow" [44], known as Linus' Law. This refers to the idea that people notice different things about a program, and so any given bug will be obvious to a person looking at the aspects of the program related to that bug. Similarly, some grammar mistakes are more obvious to some people than others — everyone has a set of mistakes that are particularly annoying to them and those mistakes will present themselves more clearly to those people. Hence, it helps to have many people look over a document for mistakes, rather than allocating the same number of human-hours to a single person looking for mistakes. A human computation algorithm can effectively manage the use of many eyes, hiring many people to look at different aspects of a project as it evolves to get a list of possible bugs or mistakes that need to be fixed.

Second is "wisdom of crowds". James Surowiecki argues that the average guess of a group of people is often more accurate than any particular member of the group [51]. The principles of statistics predict this will be true as long as the guesses are all made independently, and the group does not have a systemic bias about the answer. This is a tool that human computation algorithms may be able to leverage for making decisions. For instance, an algorithm may have many people vote between two plans to see which is more likely to result in a better end product, or between two outlines to see which is likely to result in a better article.

Note that wisdom of crowds is different from many eyes. Wisdom of crowds says that nobody knows the answer, but the average guess is correct, whereas many eyes

says that only a few people know the answer, so it's worth finding those people. Both concepts are useful in different contexts.

A third reason is division of labor, or the industrial revolution of knowledge work. The industrial revolution increases the efficiency of physical production by breaking down the process of constructing a product into steps that can be performed on an assembly line. Likewise, it may be possible to break down large knowledge-work projects, like writing an article, into small well-defined chunks, with a big efficiency gain to be had by having people hyper-specialize in the various bite-sized chunks. Breaking knowledge work tasks apart in this way, assuming it is possible, may also get around Fred Brooks' "mythical man-month" [8], allowing additional manpower to speed up a project linearly.

Another benefit tied with breaking down projects originally done by a single expert is quality control. If a single expert is responsible for a large knowledge work task, they may jeopardize the entire task if they have a bad day. On the other hand, if a task is decomposed into many bits, no single worker can harm it too much, especially if workers check each other's work. Also, by breaking a process down, it is possible to add redundancy to sub processes without adding overhead to other parts that may not need as much redundancy. It is also possible to apply statistical process control (SPC) measures attributed to Walter Shewhart for manufacturing processes [47].

Finally, breaking down processes allows for more quantitative experimentation. This allows systematic optimization techniques to be applied to a process. These sorts of optimizations can be difficult to apply at the level of a single individual, because it can be hard to convince an individual expert to vary their protocol in a tightly controlled way. However, when more of the process is encoded as a human computation algorithm, it may be easier to experiment with variations of the protocol, and test them.

1.3.1 Why Hasn't This Been Tried?

Perhaps you are convinced that there is some value in learning to orchestrate small contributions from many humans. Why hasn't this been done already? The short

answer is: it has been. In “When Computers Were Human”, David Alan Grier cites a number of historical cases where organized groups of human computers perform calculations with schemes for dividing labor into small chunks and detecting errors [25]. One recent example he cites is the Mathematical Tables Project, which was started by the Works Progress Administration (WPA) in 1938, and lasted for ten years.

Human computers were ultimately replaced by digital computers. However, it is interesting to note that the modern notion of “programming” in computer science has its roots in production engineers designing steps for manufacturing processes involving human effort [24]. In fact, people have thought for ages about how to break down large problems into manageable chunks which can be assigned to different people. The pyramids are an ancient example. Companies are modern examples. Modern companies even make use of formal languages to describe business processes, like the business process execution language (BPEL) [1].

However, there is a difference today. High-speed telecommunications over the web and micro-payment technologies have combined to enable something that was not possible before: a reliable on-demand general-purpose source of human computation. Today, anyone who wants to can post a question on MTurk asking one hundred different people around the world to name their favorite color for 1 cent each, and expect to get results within a day. This was not possible even ten years ago. This is the essential difference that we want to take advantage of and explore in this thesis.

1.4 Contributions

Someday we want to learn how to break down complex knowledge work tasks into efficient human computation algorithms, but this is still a distant goal. This thesis takes several steps in the direction of this goal, laying groundwork for other researchers to build upon. We begin with a toolkit for writing human computation algorithms. Next, we articulate a model for common building blocks within human computation algorithms, and run a series of experiments exploring the tradeoffs of a couple of

algorithmic approaches. We end by putting this knowledge together to build an end-to-end system, and describe what we learned.

1.4.1 Programming with Human Computation

Even with a source of human computation like MTurk, it turns out to be tricky to write programs that use human computation. Let us set aside for the moment the design challenges of knowing how to organize small bits of human effort to do something useful. Instead, we focus on more basic challenges associated with the process of writing and debugging human computation code. The problems stem from the high latency associated with posting tasks and waiting for responses. An algorithm with a moderately long sequence of tasks may take hours or days to complete.

One issue this causes is the simple technical annoyance of maintaining a long running process. Experiments can be ruined if a server reboots or a laptop is closed for the night. Even if the computer keeps running, a process may crash due to a bug in the code. Developers often need to rerun algorithms many times before working out all the bugs. This problem exists in other programming domains as well, for instance running an algorithm over a very large data set. One common approach to this problem is to develop an algorithm on a subset of the data, so that it runs very quickly, and then run it on the entire dataset once the algorithm is debugged and working. Unfortunately this approach does not work in human computation, since even a single human procedure call may take longer than a developer wants to wait, and will cost money.

One natural answer to these challenges is to use an event driven program backed with a database, like a database backed web server. Such a web server can handle a machine shutting down because its state is persisted on disk. A human computation algorithm could run on such a server as a web application. It would just proactively make requests where a typical web application would respond to requests.

The problem with this approach is that the event driven programming model is a cumbersome model for writing algorithms. Programmers often express algorithms in an imperative or functional programming language, using constructs like for-loops and

recursive function calls. These constructs are implemented by a runtime environment using state variables like an instruction pointer and a call stack. If a programmer wishes to take an imperative or functional program and express it in an event driven model, they need to figure out how to store this state information themselves in a database, which can be cumbersome.

We propose an alternative approach called *crash-and-rerun*. The design goal of the crash-and-rerun programming model is to allow users to continue using the language of their choice, while adding the ability to persist a program's state on disk. The key insight of crash-and-rerun is that we can return a program to a given state by memoizing the results of all the non-deterministic actions leading up to that state. If we memoize results in this fashion, then when a program crashes, we can rerun it to get back to the point of failure.

We introduce a toolkit for writing human computation algorithms on MTurk called TurKit, which uses the crash-and-rerun model. When a TurKit script would have to wait for a task to complete on MTurk, it simply stops running (crashing itself). If a user reruns the script after the task is complete, the script will get back to where it left off and successfully read the results. Since these results are non-deterministic, they are memoized for future reruns of the program. If the script crashes due to a bug, the user can fix the bug and rerun the script up to the point of the bug without waiting for any MTurk tasks to complete. The downtime between reruns also provides a convenient opportunity for a programmer to modify or append code to their script, without interfering with memorized results, in order to iteratively develop an algorithm.

TurKit has been available as an open-source tool for over a year, and this thesis discusses insights gained from our own use of the tool, as well as use from other people. Some limitations include usability issues with TurKit's parallel programming model, and scalability issues with the crash-and-rerun programming model in general. Some benefits include retro-active print-line debugging and easy experimental replication.

1.4.2 Modeling and Experimenting with Human Computation Algorithms

Once we have a suitable programming model and toolkit, we need knowledge about how to write effective programs that use human computation. The analogous knowledge for conventional programming tells us that sorting is a common subproblem, and that quicksort is more efficient than bubblesort for solving this problem.

We identify two common types of tasks in human computation: *creation* and *decision* tasks. Creation tasks may ask a worker to solve of a problem or generate an idea. They tend to be open ended. In the handwriting transcription algorithm at the beginning of this chapter, the tasks that ask workers to make guesses about the words in the passage are creation tasks. The results of creation tasks may be too difficult for a computer to process directly. Decision tasks ask workers to evaluate, rate or compare work artifacts. Decision tasks are used to bring results from creation tasks down to a level the computer can understand and make decisions about, e.g., down to a boolean value or number that is easy to write an if-statement around.

A common subproblem in human computation involves hiring many workers to solve a single problem, either because it is too difficult for any single person to solve, or as a means of quality control in case some workers do not produce high quality output. A natural question to ask is whether it is more efficient for workers to work independently, or whether it would help to show workers the results obtained from other workers. This suggests two patterns: *iterative* and *parallel.* The iterative pattern shows each worker results obtained from prior workers, in the hope that they can use them as stepping stones to improve upon. The parallel pattern has workers work independently, with the idea that seeing other people's work may stifle their creativity, or cause people to make only minor incremental improvements.

We perform a series of three experiments to examine the tradeoffs between iterative and parallel algorithms. Each experiment compares each algorithm in a different problem domain. The problem domains include writing image descriptions, brainstorming company names, and transcribing blurry text. In each experiment, we hold

money constant, and use the same number of creation and decision tasks. We measure quality as an output. In each domain, we find that iteration increases quality, with statistically significant differences in the image description writing and brainstorming domains. However, we also find that iteration tends to push workers in a particular direction, reducing the variance of individual workers. Hence, the most promising results may be more likely to come from the parallel process. We also find that the iterative approach can get stuck in local maxima, where the prior work presented to new workers pushes them in the wrong direction. This suggests that a hybrid approach, running several iterative processes in parallel, may work better.

1.4.3 Putting It All Together

After we have a toolkit, and we know how to write some simple processes, it is natural to ask how these can be put together to perform a larger real-world task. We chose the task of digitizing handwritten form data. We wanted an end-to-end system that would start with a set of scanned handwritten medical forms, and output a spreadsheet representing the data on the forms. The spreadsheet would contain a row for each scanned form, a column for each form field, and a cell for each transcription of one handwritten field for a given scanned form.

We were also interested in privacy control, since we would be exposing potentially sensitive form data to random people on the internet through MTurk. Our idea for privacy control was to limit how much of a form any given worker could see — if a worker could see the text "HIV", but not see who it was associated with, then this might be ok. Although this idea motivated our approach, we did not run tests to validate the privacy aspects of the system.

The process involves three phases. The first phase identifies form fields by having workers draw rectangles over fields on a blank form. This phase uses an iterative algorithm where workers see rectangles drawn by previous workers. This lets us ask for a small number of rectangles from each worker, while still covering the entire form. If we did not show rectangles drawn by previous workers, most people would

contribute rectangles in the upper left of the form. Showing rectangles drawn by other workers encourages people to draw rectangles elsewhere on the form.

The second phase generates labels for each form field for use as column headers in the output spreadsheet. This phase also uses an iterative process to generate labels. We chose iteration in this case because many good labels were possible for different fields, and we wanted to facilitate faster convergence on a label that was "good enough". This backfired a little because of the way we displayed prior suggestions, which caused some workers to think they were meant to come up with novel suggestions rather than copying the best prior suggestion.

The third phase generates a different task for each form field on each scanned form. In each task, a worker sees a scanned form cropped down to a particular field. We also display the label for this field to give the worker context for deciphering the handwriting. We use a parallel process in this phase, since we want higher accuracy, so we don't want to let workers copy suggestion from prior workers that seem "good enough". Although the accuracy was high, the fact that we cropped the form created a number of issues. For instance, a worker might see the word "same" written in a zip code field, but not be able to see the other zip code field that the word "same" is referring to.

Overall, this process explores the tradeoffs between iterative and parallel approaches in the context of a real-world system. It also exposes issues that can crop up in a larger system, some of which are most relevant to the specific example of transcribing forms, but some of which may generalize to developing other human computation systems.

1.4.4 Thesis Statement

The crash-and-rerun programming model allows developers to prototype human computation algorithms given a human computation market (e.g. MTurk). Two common building blocks for human computation algorithms are creation and decision tasks, which can be organized into iterative and parallel algorithms. Both of these

algorithms are beneficial for achieving different goals in larger human computation systems.

1.5 Outline

We will now progress as follows. In Chapter 2, we talk about related work, positioning this work in the greater context of human computation and collective intelligence. Chapter 3 introduces TurKit, a toolkit for writing human computation algorithms. Chapter 4 presents some experiments we ran comparing iterative and parallel algorithms. Chapter 5 discusses the development of a larger human computation system composed of several sub-processes strung together. Chapter 6 coalesces our experience into a high-level discussion of the current state-of-the-art of human computation, outlining the challenges we face now, and suggesting directions for future work. Chapter 7 closes the thesis with a few brief concluding remarks.

Chapter 2

Related Work

2.1 Human Computation

Quinn and Bederson suggest that modern use of the term *human computation* originates with Luis von Ahn's thesis, which is titled "Human Computation" [55]. Quinn and Bederson also aggregate definitions in the literature since von Ahn's thesis and come up with two primary criteria for human computation. First, human computation deals with problems that "fit the general paradigm of computation..." So far the definition fits our metaphor of thinking about human computation as adding a human procedure call to an ordinary microprocessor. Their definition goes on to add "...and as such might someday be solvable by computers." This part seems more philosophical, implying that there is some inherent difference between what humans and computers are capable of solving. This thesis is going to deal with some human computation algorithms that attempt to do things like write a paragraph describing an image. Writing is generally considered a creative task, and at the far reaches of what a computer might someday do. Hence, we will allow human computation to include the delegation of any sort of tasks to humans, as long as the delegation is performed as part of an algorithm. This falls in line with the second part of Quinn's definition: "human participation is directed by the computational system or process."

2.1.1 Human-based Genetic Algorithms

One branch of literature conceptualizes human computation in terms of genetic algorithms. An early example is the Biomorphs program accompanying a book by Dawkins in 1986 [18]. This program is a genetic algorithm designed to evolve images, but it outsources the fitness function to humans. This is called an interactive genetic algorithm (IGA). A more general concept is interactive evolutionary computation (IEC), which is described in [52] as a computer guided search through "psychological space".

Kosorukoff proposes a human-based genetic algorithm (HBGA) where both the selection and recombination steps of a genetic algorithm are performed by humans [34]. Kosorukoff also proposes a classification scheme of algorithms involving both humans and computers based on the division of labor between selection and recombination [34]. In this work, Kosorukoff describes the Free Knowledge Exchange[1] which uses a human-based genetic algorithm to evolve answers to questions.

This thesis takes a broader view of human computation, including algorithms and workflows that do not look like genetic algorithms. With that said, many of the workflows discussed in this thesis can nevertheless be modeled in terms of a genetic algorithm. For instance, the algorithm at the beginning of this thesis may be viewed as a genetic algorithm with a population of one or two genomes. The voting step may be used as the fitness function, which selects one of the genomes to survive and "reproduce" in the next generation. There isn't any recombination since only one genome survives, but this genome produces a mutated offspring when a new MTurk worker makes changes to the guesses in the text.

Additionally, as we will discuss a little later, this thesis notes two primary types of tasks involved with human computation, namely generation and decision tasks. Both of these task types map easily into the genetic algorithm model of recombination and selection. Hence, it is easy to see how some people view human computation through the lens of genetic algorithms.

[1] http://3form.org/

2.1.2 Games With a Purpose

One view of human computation is a system where humans follow rules, as in a game. Of course, games are not ordinarily thought of as performing computation, so much as entertaining the participants. However, just as computer programs can be written to perform useless tasks, so to can games be designed with useful byproducts.

Consider the ESP Game designed by von Ahn and Dabbish [57]. Roughly speaking, the game works as follows: two players see the same image, but not each other. Each player guesses a word, based on the image, and the players win if they guess the same word. If the players win, then the word becomes a useful byproduct, namely a label for the image.

Luis von Ahn et al. have designed a number of similar games with other useful byproducts, and have generalized the idea as the notion of games with a purpose (GWAP) [56]. One GWAP worth mentioning is Phetch [58], which is a game designed to create natural language descriptions of images. This is a related alternative to the iterative and parallel algorithms we will experiment with in chapter 4 for writing image descriptions.

Turing Test

Alan Turing proposed a sort of game with a purpose called the Turing test [54]. In the game, a human tries to distinguish which of two unseen agents is human, just by talking to them. The purpose of the game is to decide whether an agent has human-like intelligence.

Luis von Ahn created a sort of Turing test for deciding whether visitors to websites are human, called a CAPTCHA [3]. You have likely done many of these yourself — they involve deciphering obscured letters and numbers. The brilliant insight about CAPTCHAs is the idea that it is possible for a machine to create a puzzle that humans are better at solving than machine. It is easy for a machine to display obscured text, but it is not easy for a machine to read obscured text.

Von Ahn went on to create reCAPTCHA [60], which combines elements of traditional GWAPs as well as CAPTCHAs. ReCAPTCHAs have two byproducts: first is evidence that an agent is human, and second is the transcription of a difficult word in an OCR process. In this case, the puzzles are not computer generated, since the computer does not know the answer. The trick is to show humans two words, where one word is already known (based on previous reCAPTCHA instances), and the other is not. The human must enter both words, not knowing which is already known, and thereby providing a guess about the unknown word.

2.1.3 Programming with Human Computation

We conceptualize human computation as an extension to a Turing machine. Of course, Turing machines are theoretical constructs. In theory, Turing machines can do anything we think of as computation. In practice, people do not write programs using the primitive operations of a Turing machine. Instead, people write programs using higher level instructions and idioms. Similarly, human computation algorithms may be capable of doing great things in theory, but what instructions and idioms will help us write these algorithms in practice?

This thesis proposes two general types of human instructions: *create* and *decide*. Creation tasks ask users to generate new content, usually in a form the computer cannot understand. Decision tasks assess content in some computer-understandable way, so the computer can decide what to do next. Malone et al. also make a distinction between create and decide. For example, Threadless[2] uses a creation task to generate a bunch of designs, and then a decision task to pick a design. This thesis uses the same names, but applies them at a lower level, e.g., generation task for us refers to a single worker generating a single design. The generative/decision distinction is also made by Kosorukoff [34] using names like recombination or innovation and selection. As we mentioned before, these names come from thinking of human computation as a genetic algorithm.

[2] http://www.threadless.com/

Another construct examined in this thesis is the distinction between "parallel" and "iterative" workflows, where parallel tasks are done without knowledge of what other workers are doing, while iterative tasks allow workers to build on prior work. Malone et al. [39] also make a distinction between workers working dependently and independently, which roughly map to our iterative and parallel processes. Quinn and Bederson [42] propose an aggregation dimension to human computation, of which "parallel" and "iterative" would be two examples.

Concurrent with and after our research, other researchers have explored a number of other human computation algorithmic paradigms. Kittur et al. [32] explore the map-reduce paradigm for human computation in CrowdForge. They apply this model to a couple of problems: writing an article, and making a decision. In each case, they also include a partition step before the map and reduce steps. In the article writing case, the partition step generates a list of subjects for each paragraph, the map step writes the paragraphs, and the reduce step aggregates the paragraphs into a single article. They also suggest the idea of recursively applying this paradigm. For instance, in this example, one could imagine that each paragraph is written with a recursive application of partition to generate supporting points, map to turn each point into a sentence, and reduce to aggregate the sentences into a paragraph.

Kulkarni et al. also propose a recursive task-decomposition framework for human computation in Turkomatic [35]. Kulkarni takes the idea to a meta-level by hiring worker to do the task-decomposition themselves, rather than requiring a human computation programmer to decompose the task themselves. For each task, workers can either do the task, or propose a way to sub-divide the task into simpler pieces which will result in new tasks.

Bernstein et al. build on the TurKit framework proposed in this thesis and propose a high-level design pattern specific to human computation called *find-fix-verify* [5]. This pattern is used in Soylent, an add-in for Microsoft Word that performs a number of word processing tasks. The Soylent proofreading algorithm uses a *find* step to locate errors in a paragraph, a *fix* step to propose solutions for each error, and a *verify* step to filter out bad solutions. The find-fix-verify pattern is designed to balance a

number of cost and quality concerns, given the way different people approach human computation tasks.

Dai et al. advocate an AI backed approach for designing human computation algorithms, where steps in an algorithm are adjusted based on *a priori* models of human behavior, and adjusted dynamically at runtime as more information is acquired [16]. For example, consider the algorithm proposed in Figure 1-1. The algorithm hard-codes the number of iterations, as well as the number of votes between each iteration. However, Dai shows how the quality of output may be improved if less votes are used for early iterations, and more votes are used in later iterations, since those iterations are likely to be more difficult to distinguish in terms of quality. In fact, the algorithm can work even better if the number of votes used at each iteration is adjusted dynamically based on the results of prior votes, as well as the number of iterations. The TurKit toolkit proposed in this thesis is general purpose enough to allow a programmer to implement these improvements to the algorithm.

2.2 Collective Intelligence

Collective intelligence is a broader concept than human computation. It is what we get when we relax the requirement that a system be governed by clearly defined rules. As Malone puts it, the only requirement of such a system is that it involve "groups of individuals doing things collectively that seem intelligent" [39].

However, even though collective intelligent systems need not have clearly defined rules, they nevertheless often have structure. Malone et al. identify recurring structural themes that map the "genome of collective intelligence." [39] Using this scheme, websites like Wikipedia can be broken down into structural elements, or "genes". For instance, Wikipedia may be broken down into two primary tasks, creating articles, and editing articles, each of which may be expressed as a sequence of "create" and "decide" genes. This characterization is very similar to the sequence of create and decide steps in the algorithm at the beginning of this thesis, only a little more loosely managed and enforced.

2.2.1 User-Generated Content Sites

Wikipedia is a well known example of a user-generated content (UGC) website, where another popular example is YouTube [3]. These sites are so popular that they have received academic attention in their own right. Bryant [9] makes observations about how people begin contributing to Wikipedia, and what tools expert contributors use to manage and coordinate their work. Cosley [15] presents an interesting suggestion for Wikipedia users which is an automated system for routing tasks to users. Kittur [31] observe how different coordination strategies affect the quality of Wikipedia articles. An extensive study of YouTube is done in [11]. This study compares UGC video sites to non-UGC video sites. It also studies various usage patterns among contributors. Insights gained from these studies seem relevant to designing human computation algorithms, because if we understand successful coordination patterns that lead to well-used and popular UGC sites, it may shed insight on how to specify these coordination patterns with rules that can be used within human computation algorithms.

In addition to studying popular existing sites, some researchers have tried to create their own site and make it popular enough to run studies. This is essentially the approach taken by von Ahn to entice users to a project by making it appealing to a broad audience, e.g., as a video game. Researchers at University of Minnesota created MovieLens [4], which allows users to rate movies. The site has been used to conduct controlled studies of usage patterns given user interface changes [43]. Having control of such a site is a powerful tool for researching collective intelligence, but is not something most academic researchers have at their disposal. This is part of the appeal of micro-task labor markets like MTurk, which we will discuss further in the Human Computation Markets section below.

---

[3] http://www.youtube.com

[4] http://movielens.umn.edu/login

2.2.2 Open Source Software

Open source software is another form of collective intelligence that has garnered a fair amount of attention, especially with the popularity of the Linux operating system. Raymond gives a good overview of the open source process in [44]. One key principle he states is "given enough eyeballs, all bugs are shallow." I mentioned before that I believe human computation algorithms stand to benefit from this same principle. A lot of academic research studies the phenomena of open source software, including the motivations of workers [36], [28], [61], and the communication channels underlying open source development [26]. Communication issues in commercial development project are covered in [10]. The communication question may be especially relevant to human computation algorithms, since the algorithms themselves may need to act as communication mediators, letting each contributor know what has been done, and what next steps need to be done. This is especially important if we want to start creating algorithms that operate on artifacts that are too large to show each individual worker, like a software project, or even a large document, like a book.

2.3 Human Computation Markets

We want to think about human computation like regular computation, with the added ability to outsource bits of work to humans. One boon to the study of computation is the advent of computers, which make it cheap and easy for researchers to run their programs. However, it is historically very difficult to run a human computation program, because it is difficult to find willing humans to participate in the computation. Hence, the human computation analog to a computer is a human computation market — a place where a computer can easily hire human workers.

Luis von Ahn essentially built his own human computation market for the ESP Game, where players trade effort for entertainment. However, much of the effort involved in setting up and marketing such a system is tangential to study of different human computation algorithms, just as much of the effort involved with building a computer from scratch is tangential to the study of conventional algorithms.

Fortunately, the internet has seen a rise in a variety of marketplaces for human effort. We will discuss a few, ending with Mechanical Turk, which we will use throughout this thesis.

2.3.1 Question/Answer Sites

Question/Answer websites like Stack Overflow[5], Quora[6], Experts Exchange[7] and Yahoo Answers[8] allow users to post and answer questions. Different sites have different systems for motivating contributions, and cater to different communities of users. Yahoo Answers is perhaps the largest and most studied of these systems. Bian [6] studies searching Yahoo Answers for factual answers. Another interesting study of Yahoo Answers is [2], which looks at differences in the number of answers a question will receive, and how lengthy those answers are, as a function of the type of question. This study also looks at programming questions in particular—apparently it is hard to get answers for "deep" programming questions. Stack Overflow is probably a better place to get answers to difficult programming problems, however, question/answer sites are still not completely general. For a question to be answered, it must typically provide some opportunity for the answerer to demonstrate their unique knowledge or insight. This does not happen if a question has already been answered elsewhere on the site, or if the question is something anyone could answer. Some work that a human computation algorithm wishes to outsource may not meet these criteria. In fact, some work may not even be textual, e.g., a task that asks workers to draw a picture using a web-based drawing application.

2.3.2 Prediction Markets

Prediction markets may be viewed as another form of human computation market. A typical prediction market allows people to buy and sell a kind of good who's value depends on the outcome of some future event. The current price of such a good

[5]http://stackoverflow.com/
[6]http://www.quora.com
[7]http://www.experts-exchange.com/
[8]http://answers.yahoo.com/

reflects the market's belief about the future outcome. Prediction markets have also been studied extensively. Wolfers [62] gives a good overview of prediction markets, and describes the different kinds. A comparison between prediction markets and polls used for predicting election results is covered by [4]. Also, [45] gives some reason to believe that money may not be as important as trade volume for prediction market accuracy. Like question/answer sites, prediction markets are not completely general, meaning it is not possible to outsource any sort of work to a prediction market. However, prediction markets are highly relevant to human computation, since the particular mechanics within a prediction market may be viewed as a human computation algorithm in itself.

2.3.3 Outsourcing Intermediaries

Outsourcing Intermediaries like Elance[9], oDesk[10], Freelancer[11] and a host of others are good examples of human computation markets based on money. These systems typically work as follows: a buyer posts a project; people post bids for the project; the buyer selects a bidder to complete the project; once the project is complete, the buyer pays the agreed amount. These projects can last days, weeks or months, with prices ranging from tens to thousands of dollars.

These sites are much more general than question/answer sites or prediction markets, however, they are not typically used to hire someone for just 10 minutes of their time. In principle, this is possible, but the overhead involved with bidding makes it impractical for requesters as well as workers. Also, not all of these sites allow the hiring process to be completely automated with a computer, though Freelancer does offer a full hiring API.

[9] http://www.elance.com/
[10] http://www.odesk.com
[11] http://www.freelancer.com

2.3.4 Micro-task Markets

Amazon's Mechanical Turk[12] may be the first general purpose micro-task market. In many ways, MTurk is like any other online labor market, with some key differences. First, there is no bidding process. The first qualified worker to accept a task is given that task, and nobody else can take it from them unless they return it. Second, workers are anonymous — each worker is assigned an opaque ID. Third, the final decision about whether to pay a worker lies entirely with the requester of the work. There is no appeal process for unfairly rejected work. These factors reduce the overhead of posting and completing tasks enough to accommodate micro-tasks paying as little as $0.01. The site is popular enough that tasks are accepted fairly quickly, often within 30 minutes.

MTurk has experienced an explosion of interest within the academic community, both as a tool, and as an artifact to be studied in itself. The two primary benefits of MTurk as a tool for academics is first as a forum in which to conduct user studies, and second as way to annotate large datasets for machine learning algorithms. Kittur studied it as a platform for conducting user studies in [30]. This work makes some useful suggestions about how to design tasks in order to avoid people gaming the system. The primary rule of thumb here is to design tasks which are as hard to do poorly as they are to do well. Heer et al. explore the use of MTurk for assessing visual design [27]. One form of user study particularly relevant to human computation is economic game theory experiments, where games may involve relatively complex sequences of actions by multiple players. Chilton et al. describe a system for designing and running game theory experiments in [13]. Sorokin et al. examine the use of MTurk as a tool for annotating data sets in [50]. This has proven to be a very popular and effective use of MTurk.

MTurk has also been studied as an artifact in itself. Chilton et al. [12] study the way turkers search for tasks given the MTurk interface, where the primary search method is a dropdown box with 6 sorting dimensions. HITs which are at the extreme ends of these dimensions are more likely to be found and accepted, where the

[12]https://www.mturk.com/

creation date dimension is the most important. Mason and Watts [40] examine the relationship between reward and quality, finding that higher pay has no effect on quality, but does increase the quantity of tasks a worker is willing to do. A related study, not conducted on MTurk is [23], which notes that paying people to do work, as opposed to not paying them, can decrease the quality of their work if they are not paid enough. The quality of work on MTurk is also studied in [49], which finds high agreement between MTurk workers and expert annotators for a natural language task. In general, many researchers have found MTurk quality to be remarkably high given the small price. Other researchers have also worried about the small price in itself as a labor ethics issue, including Silberman et al. who explore issued faced by workers on MTurk in [48].

Other Micro-task Markets

Although MTurk has been the primary target of academic research in the space of micro-task markets, a number of alternatives are starting to surface. CrowdFlower[13] began as Dolores Labs, which ran a number of interesting studies on MTurk, in addition to consulting for businesses interested in using MTurk. CrowdFlower has now expanded their labor pool beyond MTurk, and has an API of their own which can even be used to access MTurk indirectly. SamaSource[14] is a non-profit that focuses on providing work to people in developing countries. CroudClowd[15] is a labor pool where people login using their Facebook accounts. Some other services exist which employ crowds, but which are less general purpose, including LiveOps[16], if we consider individual phone conversations as micro-tasks, and Microtask[17], which focuses on document processing.

[13]http://crowdflower.com/
[14]http://www.samasource.org/
[15]http://www.cloudcrowd.com/
[16]http://www.liveops.com/
[17]http://www.microtask.com

2.4 Programming Model

One contribution of this thesis is a programming model suited to programming with human computation. The *crash-and-rerun* programming model involves re-running scripts without re-running costly side-effecting functions. This programming model is novel, mainly because the design constraints of programming with human computation are odd enough that the model seems to have found no use up until now.

However, it is related to programming languages that store execution state, and resume execution in response to events. IGOR [22] supports reversible execution, where a debugger can step backward through steps in a program. The Java Whyline records an execution trace of a program in order to answer causality questions about a program after it has already executed [33]. Similarly, the crash-and-rerun model can be used to inspect and debug a program which has already executed. However, our implementation is more light-weight, and does not require instrumenting a virtual machine. Crash-and-rerun programming is also similar to web application programming. Web servers typically generate HTML for the user and then "crash" (forget their state) until the next request. The server preserves state between requests in a database. The difference is that crash-and-rerun programming uses an imperative programming model, whereas web applications are generally written using an event-driven state-machine model.

Some innovative web application frameworks allow for an imperative model, including Struts Flow[18] and Stateful Django[19]. These and similar systems serialize continuations between requests in order to preserve state. However, this approach typically does not support modifications to the program before restarting a continuation. This is less of an issue for web services since the preserved state generally deals with a single user over a small time-span, whereas human computation algorithms may involve hundreds of people over several days, where it is more important to be able to modify a script between re-runs in order to fix bugs.

[18] http://struts.apache.org/struts-sandbox/struts-flow/index.html
[19] http://code.google.com/p/django-stateful/

Crash-and-rerun programming is also related to scripting languages for distributed systems, since they all need to deal with computation stretched over time. Twisted[20] uses "deferred computation" objects. SALSA[21] uses an event-driven model. Swarm[22] uses Scala[23], which implements portable continuations. The crash-and-rerun approach is similar to storing a continuation, except that instead of storing the current state of the program, it memoizes the trace leading up to the current state. This approach is not suitable for many distributed applications because it consumes more memory over time. However, this simple implementation leads directly to the ability to modify programs between re-runs, which turns out to be useful for prototyping human computation algorithms.

[20]http://twistedmatrix.com/
[21]http://wcl.cs.rpi.edu/salsa/
[22]http://code.google.com/p/swarm-dpl/
[23]http://www.scala-lang.org/

Chapter 3

Tools

```
var ideas = []
for (var i = 0; i < 5; i++) {
    ideas.push(mt.promptMemo(
        "What's fun to see in New York City?"))
}

ideas.sort(function (a, b) {
    return mt.voteMemo(
        "Which is better?", [a, b]) == a ? -1 : 1
})
```

Figure 3-1: A TurKit script for a human computation algorithm with two simple steps: generating five ideas for things to see in New York City, and sorting the list by getting workers to vote between ideas.

This chapter introduces TurKit, a toolkit for writing human computation algorithms[1]. The goal of TurKit is to provide an abstraction that allows programmers to incorporate human computation as an ordinary function, with inputs and outputs, inside a typical imperative program (see Figure 3-1).

TurKit is built on top of MTurk. Many programming tools are built on MTurk, but TurKit caters to the design constraints of prototyping algorithmic human com-

[1]Parts of this chapter are published in a conference paper at UIST 2010 titled "TurKit: Human Computation Algorithms on Mechanical Turk." [38]

putation programs, where tasks build on other tasks. MTurk is also often used for *independent* tasks, where requesters post a group of HITs that can be done in parallel, such as labeling 1000 images. Tool support for these processes is pretty good. Amazon even provides a web interface for designing HIT templates, and posting batches of similar HITs.

There are also programming APIs for making MTurk calls in a variety of languages, including Java and Python. However, there are some difficulties if we want to write an algorithm. The main problem is that MTurk calls have high latency. There are two primary ways to deal with this. First is to poll. That is, we write a while loop that continually checks MTurk to see if a given HIT is done yet. The problem with this approach, as we'll see below, is that it makes the iterative design cycle of writing a program too long and costly. Consider a program that spends 30 minutes waiting on the results of several HITs, and then throws an exception on the very next line. Once we fix the bug, we need to wait another 30 minutes for the program to return to that line of code, as well as spend more money re-posting the HITs.

The second method is to use an event driven style, similar to GUI or web applications. In this style, we generate events when HITs are completed on MTurk, and the program responds to these events. The problem with this approach is that the state of the program needs to be stored in some data structure between events, and this can be cumbersome to do in the case of certain algorithms, especially recursive algorithms, where the data structure needs to manually represent a stack of recursive calls.

TurKit incorporates a unique *crash-and-rerun* programming model to help overcome these systems challenges. In this model, a program can be executed many times, without repeating costly work. Crash-and-rerun allows the programmer to write imperative programs in which calls to MTurk appear as ordinary function calls, so that programmers can leverage their existing programming skills.

This chapter will explain TurKit in layers, where each layer adds additional functionality to the lower layers. We will then present a performance evaluation of TurKit, and discuss how people have used TurKit for a variety of human computation algo-

rithms. We'll end with a discussion of pros and cons we have discovered in the programming model.

3.1 TurKit Layers

3.1.1 JavaScript on Java

A TurKit program is written in JavaScript. This JavaScript is executed using Rhino[2], which is an open source JavaScript interpreter written in Java. Rhino provides easy access to Java objects and methods from JavaScript code, e.g., scriptable java[3]. We will use this feature to expose useful methods to JavaScript that are not native to the JavaScript language. For instance, JavaScript does not have a `print` function, but we can create it as follows:

```
function print(s) {
    Packages.java.lang.System.out.println(s)
}
print("Hello World")
```

3.1.2 JavaScript API to MTurk

The first layer of functionality provided by TurKit is a JavaScript API for accessing MTurk. Most of this API is implemented in JavaScript itself, though it does require a Java function to make web requests, since Rhino does not provide the XmlHttpRequest object provided by most web browsers.

Users initialize the JavaScript API by calling `mt.init`. This function takes five parameters. Only the first two parameters are required, which are strings indicating the user's Amazon Web Service (AWS) access key id and secret access key. These credentials are needed for all MTurk API calls. The third parameter is a string with the value "real" or "sandbox". Passing "sandbox" tells the API to use the MTurk

[2]http://www.mozilla.org/rhino/

[3]This is fuel for the fire of confusion about the name JavaScript.

sandbox, which is a functionally complete clone of MTurk that uses fake money, and is used for testing purposes. The default is "real", which tells the API to use the real-money version of MTurk.

The user may also pass two optional safety parameters: a maximum money to spend, and a maximum number of HITs to create. TurKit will throw an exception if the script spends more money or creates more HITs than these numbers indicate. This guards the user against accidentally creating HITs in an infinite loop. Users can also guard against spending too much money by limiting how much money they put into their MTurk account — this is different from other AWS services such as S3, for which the user does not pay upfront, but rather gets a bill at the end of the month.

The rest of the JavaScript API essentially wraps the MTurk SOAP API[4], and tries to make it a bit simpler to use.

For instance, consider that we want to create a HIT where the title and description are both "What is your favorite color?". We want to offer $0.01 for this HIT, and the web form itself is hosted at "http://example.com". The MTurk SOAP request for this operation looks like this:

```
<?xml version="1.0" encoding="UTF-8" ?>
<soapenv:Envelope
    xmlns:soapenv="http://schemas.xmlsoap.org/soap/envelope/"
    xmlns:xsd="http://www.w3.org/2001/XMLSchema"
    xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance">
  <soapenv:Body>
    <CreateHIT xmlns="http://mechanicalturk.amazonaws.com/
        AWSMechanicalTurkRequester/2008-08-02">
      <AWSAccessKeyId>ACCESS_KEY_ID</AWSAccessKeyId>
      <Timestamp>2011-03-24T03:53:34Z</Timestamp>
      <Signature>OsaSiMMMoj+n+iKvE+adg3Yq7AY=</Signature>
      <Request>
        <Title>What is your favorite color?</Title>
        <Description>What is your favorite color?</Description>
        <Question>&lt;ExternalQuestion xmlns=&quot;http://mechanicalturk.
          amazonaws.com/AWSMechanicalTurkDataSchemas/2006-07-14/
          ExternalQuestion.xsd&quot;&gt;&lt;ExternalURL&gt;
          http://example.com&lt;/ExternalURL&gt;&lt;
          FrameHeight&gt;600&lt;/FrameHeight&gt;&lt;/
          ExternalQuestion&gt;</Question>
        <AssignmentDurationInSeconds>3600</AssignmentDurationInSeconds>
        <LifetimeInSeconds>604800</LifetimeInSeconds>
```

[4]http://docs.amazonwebservices.com/AWSMechTurk/latest/AWSMturkAPI/

```
        <MaxAssignments>1</MaxAssignments>
        <Reward>
          <Amount>0.01</Amount>
          <CurrencyCode>USD</CurrencyCode>
        </Reward>
      </Request>
    </CreateHIT>
  </soapenv:Body>
</soapenv:Envelope>
```

Note that the website "http://example.com" appears inside the `Question` element, embedded within a block of escaped XML representing an `ExternalQuestion`.

The response from MTurk for this request looks like this:

```
<CreateHITResponse>
  <OperationRequest>
    <RequestId>ece2785b-6292-4b12-a60e-4c34847a7916</RequestId>
  </OperationRequest>
  <HIT>
    <Request>
      <IsValid>True</IsValid>
    </Request>
    <HITId>GBHZVQX3EHXZ2AYDY2T0</HITId>
    <HITTypeId>NYVZTQ1QVKJZXCYZCZVZ</HITTypeId>
  </HIT>
</CreateHITResponse>
```

The TurKit wrapper looks like this:

```
mt.init("ACCESS_KEY_ID", "SECRET_ACCESS_KEY", "sandbox")
mt.createHIT({
     title: "What is your favorite color?",
     description: "What is your favorite color?",
     url: "http://example.com",
     reward: 0.01
})
```

This function shields the user from writing cumbersome XML, in addition to supplying useful defaults. For instance, the `CurrencyCode` defaults to `USD`. This function also takes care of signing the request, which involves taking a SHA-1 hash

of certain elements in a certain order, and Base64 encoding the result into the `Signature` element. The function also repeats the request if it fails due to a "service busy" error, exponentially backing off the request interval. The function looks for `<Request><IsValid>True</IsValid></Request>` and throws an exception if it is not. If the request succeeds, then the function returns just the HIT Id, since this is all that is required to retrieve the results of the HIT.

Most defaults for `mt.createHIT` can be overridden by suppling additional properties, but some cannot, like the `CurrencyCode`. If the user needs more control over the SOAP request, they can generate the XML inside the `Request` element of the SOAP request and pass it to `mt.soapRequest`, along with the operation name, e.g., "CreateHIT". This method will add all the additional SOAP XML, as well as properly sign the request. They would then need to parse the resulting XML.

Incidentally, parsing XML is not too bad in Rhino's JavaScript, since it supports ECMAScript for XML (E4X), which allows XML literals in JavaScript code. Parsing XML can be done by simply calling `eval(x)`, where `x` is a string of XML, or passing the string to the native XML object constructor, e.g., `new XML(x)`.

Retrieving Results

TurKit also simplifies the process of retrieving results for a HIT. The process can involve several API calls to MTurk. One call is needed to get the status of the HIT. Next, since HITs may have multiple assignments, and MTurk only allows 100 assignments to be retrieved with a single API call, multiple calls may need to be made to retrieve all the assignments. Finally, the results for each assignment are stored as a string of escaped XML within this XML. This requires parsing the XML of an assignment, fetching the `Answer` element, and parsing it as XML again.

The TurKit `mt.getHIT` function does all this work, and aggregates the information about a HIT and its assignments into a single JavaScript object. The assignments are stored inside an array called `assignments`. Each assignment contains an `answer` object with the results submitted for that assignment. For instance, if a HIT uses a webpage with a text input named "color", and the first turker to accept an assignment

for the HIT enters "green" into this input field, then the following code will print "green":

```
print(mt.getHIT(hitId).assignments[0].answer.color)
```

3.1.3 Database

After we use TurKit to create a HIT on MTurk, we need to wait until it completes before we can do anything with the results. The naive solution is to use a busy-wait, like this:

```
mt.init(...)
hitId = mt.createHIT(...)
do {
    result = mt.getHIT(hitId)
} while (!result.done)
print(resutl.assignments[0].answer)
```

This is conceptually elegant, and doesn't consume too much CPU time if we sleep for a minute between each iteration of the loop. However, it does have a couple of drawbacks. First, a script like this can be a bit annoying if it is running on a laptop, and we want to shut it down over night. Second, and more important, what happens when the while-loop completes, and the program encounters a silly error on the very next line, namely, I mistyped "result" as "resutl". We may have waited for an hour for someone on MTurk to complete our HIT, only to get a null pointer exception on a line of code that we haven't debugged yet, because the script has never executed to that point before.

If we rerun the code, it will create *another* HIT on MTurk, spending more money, and worse, spending more time. In this case, the time is more painful to wait because there actually is a complcted HIT out there, our program just forgot the Id for it.

Our solution is to persist information between runs of the program. In particular, we set aside a variable called `db`, and say that this variable will be serialized as JSON

between runs of the program. Hence, running the following code for the first time will print "1", whereas it prints "2" on the second run.

```
if (!db.a) {
    db.a = 1
} else {
    db.a++
}
print(db.a)
```

After each run, the database contains the JSON for db, which looks like this after the second run:

```
{
    "a" : 2
}
```

We may now use the database to cache or memoize the HIT Id from MTurk as follows:

```
mt.init(...)
if (!db.hitId) {
    db.hitId = mt.createHIT(...)
}
do {
    result = mt.getHIT(db.hitId)
} while (!result.done)
print(resutl.assignments[0].answer)
```

This time, when our program throws an exception on the line containing "resutl", we can repair the script to say "result", and simply rerun it. When it runs again, the line `!db.hitId` will be false, so `mt.createHIT(...)` will not be called again, and `db.hitId` will still contain the HIT Id created on the first run of the program.

We can make our program a little more efficient on subsequent runs by also memoizing the result of `getHIT` once the HIT is done, like this:

```
mt.init(...)
if (!db.hitId) {
    db.hitId = mt.createHIT(...)
}
if (!db.result) {
    do {
        result = mt.getHIT(db.hitId)
    } while (!result.done)
    db.result = result
}
print(db.result.assignments[0].answer)
```

3.1.4 Memoization

At this point, our code becomes a little cluttered with memoization if-statements. Worse, since `db` is a global variable, the programmer needs to find a unique location inside `db` to store each statement. This is a little annoying if the memoization is happening in a loop, where the programmer must create an array inside `db` to store each iteration, e.g., `db.hitIds[0]`, `db.hitIds[1]`. It is even more annoying in a recursive algorithm, where the programmer needs to maintain a recursive data structure inside `db` with entries like `db.recurse.hitId` and `db.recurse.recurse.hitId`. This is complicated further if the recursive algorithm is invoked multiple times. In general, all functions that use memoization internally and can be called multiple times would need to accept a unique namespace as an argument.

We use a simple solution to solve both the clutter and namespace issues. We will explain the technique in two steps. First, we introduce a new function called `memo`, short for memoize, which we can think about as syntactic sugar for the style of memoization used in our previous example. This function will take three arguments (for now). The first argument is the global variable used to store the memoized value. The second argument is a function to memoize, where the third argument is actually a variable number of arguments to pass to this function. If there is no value stored

in the global variable, then the function is called with its arguments, and its return value is memoized in the global variable for next time. This gives us:

```
mt.init(...)
hitId = memo("db.hitId", mt.createHIT, ...)
result = memo("db.result", function () {
    do {
        result = mt.getHIT(hitId)
    } while (!result.done)
    return result
})
print(result.assignments[0].answer)
```

Next, we want to remove the first argument to `memo`. We do this by automatically generating names behind the scenes, inside of the `memo` function. We essentially increment an integer each time `memo` is called to generate the next name, giving us "name1", "name2", and so on. What happens in the case of recursive calls? Nothing special, we continue to dole out names sequentially. This works fine as long as the script is deterministic.

Hence, the script must be deterministic. Non-deterministic code must be made deterministic by calling it using `memo`. This works because the non-deterministic results from the first time the code executes will be memoized for future runs of the program.

One obvious source of non-determinism is the `Math.random()` function. To shield the user from needing to wrap each call to `Math.random()` inside a `memo` function, we memoize the random seed to `Math.random()` in the database, so that random numbers are generated the same way each time. When the database is reset, a new random seed is generated and memoized, which avoids the problem of everyone's TurKit scripts generating the same sequences of random numbers.

The other big source of non-determinism in TurKit programs is MTurk. Users are motivated to memoize calls to MTurk because they cost time and money. Next, we'll discuss how we make this task a little easier.

3.1.5 Syntactic sugar for MTurk API

Since most calls to MTurk need to be memoized, we introduce a set of API calls that include the memoization as part of the process. These calls all have the word "Memo" appended to the usual name of the function, e.g., `createHITMemo`. Applying this to our running example yields:

```
mt.init(...)
hitId = mt.createHITMemo(...)
result = mt.waitForHITMemo(hitId)
print(result.assignments[0].answer)
```

Now we cheated a bit. The `waitForHIT` function was not part of the original MTurk API. Hopefully the meaning is still clear — it waits until the HIT is done before memoizing the result and returning. In fact, because the idiom above is so common, we add a utility function called `mt.createHITAndWaitMemo`, which takes the `createHIT` parameters, and returns the results after the HIT has been completed.

3.1.6 Crash-and-Rerun

It turns out that throwing an exception in this environment is not too harmful, since we have recorded all the costly operations in a database, which will allow us to execute up to the point of failure again without repeating any costly work.

Hence, rather than busy-waiting for a HIT to be done, we can simply throw an exception if it is not done, from within `waitForHIT`. We can see if the HIT is done later by rerunning the program. We call this style of programming *crash-and-rerun*.

One advantage of crash-and-rerun programming is that it can be used in environments where code is not allowed to busy-wait forever, like Google App Engine, where scripts are only allowed to run in 30 seconds intervals. Another advantage, of removing busy-waits in particular, is that it allows a unique form of parallelism, which we will discuss next.

3.1.7 Parallelism

Although TurKit is single-threaded, we do provide a mechanism for simulating parallelism. This is done using `fork`, which does two things. First, it creates a nested namespace for all the `memo` calls made within the fork, so that they can be called out of order with respect to `memo` calls inside other forks. This works recursively as well, in case `fork` is called within a fork. The second thing `fork` does is catch the exception thrown by `waitForHIT`. This exception is called "crash", and can be generated elsewhere in TurKit scripts by calling the `crash` function. By catching the "crash" exception, `fork` allows execution to resume after the call to `fork`.

The `fork` function is useful in cases where a user wants to run several processes in parallel. For instance, they may want to post multiple HITs on MTurk at the same time, and have the script make progress on whichever path gets a result first. For example, consider the following code:

```
a = createHITAndWaitMemo(...)         // HIT A
b = createHITAndWaitMemo(...a...)     // HIT B
c = createHITAndWaitMemo(...)         // HIT C
```

Currently, HITs A and B must complete before HIT C is created, even though HIT C does not depend on the results from HITs A or B. We can instead create HIT A and C on the first run of the script using `fork` as follows:

```
fork(function () {
    a = createHITAndWaitMemo(...)         // HIT A
    b = createHITAndWaitMemo(...a...)     // HIT B
})
fork(function () {
    c = createHITAndWaitMemo(...)         // HIT C
})
```

The first time around, TurKit would get to the first `fork`, create HIT A, and try to wait for it. It would not be done, so it would crash that forked branch (rather

than actually waiting), and then the next `fork` would create HIT C. So on the first run of the script, HITs A and C will be created, and all subsequent runs will check each HIT to see if it is done.

TurKit also provides a `join` function, which ensures that a series of forks have all finished. The `join` function ensures that all the previous forks along the current path did not terminate prematurely. If any of them crashed, then `join` itself crashes the current path. In our example above, we would use `join` if we had an additional HIT D that required results from both HITs B and C:

```
fork(function () {
    a = createHITAndWaitMemo(...)         // HIT A
    b = createHITAndWaitMemo(...a...)     // HIT B
})
fork(function () {
    c = createHITAndWaitMemo(...)         // HIT C
})
join()
D = createHITAndWait(...b...c...)         // HIT D
```

3.1.8 Higher-level Utilities

In TurKit, it turns out to be a little cumbersome to create user interfaces, since they generally need to be web pages, or else they need to use Amazon's XML syntax for creating questions, which can be a little cumbersome itself. For instance, here is Amazon's XML syntax for asking a one-line question "What is your favorite color?", with a free-text response:

```
<QuestionForm xmlns="http://mechanicalturk.amazonaws.com/
        AWSMechanicalTurkDataSchemas/2005-10-01/QuestionForm.xsd">
    <Question>
        <QuestionIdentifier>response</QuestionIdentifier>
        <IsRequired>true</IsRequired>
        <QuestionContent>What is your favorite color?</QuestionContent>
        <AnswerSpecification>
            <FreeTextAnswer>
                <Constraints>
                    <Length minLength="1" maxLength="20"></Length>
```

```
                </Constraints>
                <DefaultText></DefaultText>
                <NumberOfLinesSuggestion>1</NumberOfLinesSuggestion>
            </FreeTextAnswer>
        </AnswerSpecification>
    </Question>
</QuestionForm>
```

Not to mention the code to create the HIT itself, and wait for a response:

```
result = mt.createHITAndWaitMemo({
    title: "What is your favorite color?",
    description: "What is your favorite color?",
    question: ...the XML from above...
    reward: 0.01
})
```

There is a still a little more work needed to find the answer from within the `result` object returned from MTurk (made a little easier since we convert the result to a JavaScript object, rather than XML):

```
favoriteColor = result.assignments[0].answer.response
```

Note that `response` above comes from the `QuestionIdentifier` element in the `QuestionForm` XML.

This is a lot of code to do something conceptually simple, like ask a user on MTurk for their favorite color. If we were writing JavaScript inside a webpage, we could ask the user for their favorite color using the `prompt` function, which takes a string as input, and returns a string as output. Following this example, we create an MTurk `prompt` function that wraps all the code above into a single function call:

```
favoriteColor = mt.promptMemo("What is your favorite color?")
```

We also include a utility roughly analogous to JavaScript's `confirm` function, called `vote`. While `confirm` provides the user with the options "Ok" or "Cancel", `vote` takes an argument which is an array of possible options to present to the turker. Also, `vote` will ask for more than one answer, which is usually what we want on MTurk, since some people may cheat and choose randomly.

```
favoriteColor = mt.voteMemo("Which color is best?",
                            ["Blue", "Green", "Red"])
```

The `vote` function returns the item with the plurality of votes, and guarantees that this item has at least 2 votes. The function tries to be efficient (in terms of money) by asking for the fewest number of votes that could theoretically achieve the desired winning conditions, and only asks for more votes if necessary. For example, if there are two choices, the process will ask for 2 votes, and only ask for a third vote if the first two disagree.

3.1.9 External HITs

The high level functions described above use MTurk's custom XML language for creating interfaces for turkers. However, more complicated UIs involving JavaScript or CSS require custom webpages, which MTurk will display to turkers in an iframe. These are called *external HITs*, since the interface is hosted on a several external to MTurk itself.

TurKit provides methods for generating webpages and hosting them either on S3 or Google App Engine. Users may create webpages from raw HTML like this:

```
url = web.createPageMemo('<form>What is your favorite color?<br>
                          <input type="text" name="color" /><br>
                          <input type="submit" /></form>')
```

The `url` returned from `web.createPageMemo` can now be passed as an argument to `mt.createHITMemo`. It should also be passed as an argument to `web.removePageMemo` after the HIT is done, to free up space wherever the webpage was stored.

Unfortunately the simple webpage shown in the example above will not work on MTurk. In particular, the form action has not been set appropriately. Also, when a form is submitted to MTurk, there needs to be a hidden form field indicating the assignment Id of the HIT. On top of that, there is a subtle issue related to the way MTurk lets workers preview tasks before accepting them. The preview of a task uses the same iframe as the normal task, causing some users to mistakenly complete a task before accepting it. To help prevent this error, it is common for HITs to disable all the form fields when in preview mode. TurKit provides a template that achieves all of these behaviors.

```
url = web.createPageFromHITTemplateMemo(
                              'What is your favorite color?<br>
                              <input type="text" name="color" /><br>
                              <input type="submit" />')
```

The string passed to `createPageFromHITTemplateMemo` is embedded within a larger HTML page that uses jQuery to make all the necessary adjustments dynamically when the page loads, including setting the form action url, adding a hidden form element with the assignment Id, and disabling all the input elements if the HIT is being previewed.

This function also provides an optional second parameter, which is a list of worker Ids to block from doing this HIT. This is useful when an algorithm wants to prevent turkers who generated content from voting on that content. This feature works by comparing the worker Id of the user doing the HIT with the list. If the worker is on the list, then the page dynamically turns into a page telling the worker that they are not allowed to do the HIT, and that they need to return it. Note that this approach is not perfect. Turkers will still see HITs that they are not allowed to do in their list of available HITs. Also, they cannot tell when they preview a HIT that they are not

allowed to do it. Our method can only inform a turker that they are not allowed to do a HIT after they accept it. We hope that Amazon adds a way to restrict specific users from specific HITs as part of the MTurk API, perhaps extending the notion of qualifications to include *dis*qualifications.

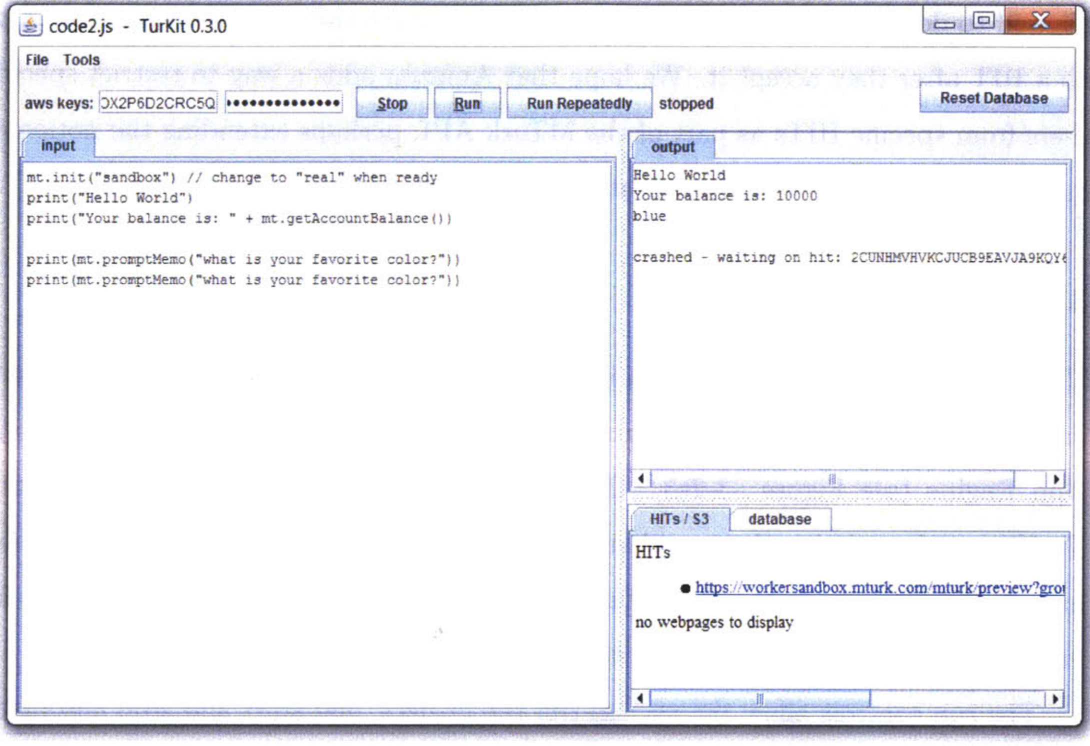
3.1.10 User interface

The final layer of the TurKit onion is the GUI. TurKit has two GUIs: a stand-alone Java GUI written in Swing (see Figure 3-2a), and an online GUI running on Google App Engine (see Figure 3-2b). Both interfaces are essentially IDEs for writing TurKit scripts, running them, and automatically rerunning them. The interfaces also have facilities for managing the database file associated with a TurKit script, and for viewing and deleting HITs created by a script.

The run controls allow the user to run the project, and start and stop automatic rerunning of the script. This is necessary since the script is likely to crash the first time it runs, after creating a HIT and seeing that the results for the HIT are not ready yet. Starting automatic rerunning of the script will periodically run the script, effectively polling MTurk until the results are ready.

There is also a control for resetting the memoization database. Typically, users will debug their script using the MTurk sandbox before letting it run unattended on MTurk. After the script appears to be working in the sandbox, the programmer may reset the database. Resetting the database clears the memoized execution trace, and also deletes any outstanding HITs or webpages created by the script. The user may then run the script in normal mode by omitting the "sandbox" parameter to `mt.init`, and TurKit will create HITs again on the real MTurk without any memory of having done so in the sandbox. Reseting the database is also useful after correcting major errors in the script that invalidate the memoized execution trace.

The *execution trace* panel shows a tree view representing the memoized actions in previous runs of the script, since the last time the database was reset. Note that calling `fork` creates a new branch in this tree. Some items are links, allowing the user to see the results for certain actions. In particular, `mt.createHITMemo` has a link to



(a)

(b)

Figure 3-2: (a) The TurKit stand-alone interface allows editing of a single JavaScript file, as well as running that file, and viewing the output. (b) The TurKit web user interface is an online IDE for writing and running TurKit scripts.

the MTurk webpage for the HIT, and the `web.createPageMemo` function has a link to the public webpage that was created. Currently, these links are only distinguishable by the order in which they appear in the tree. If the user has many HITs or webpages, it may be useful to add a feature whereby users can specify a unique name to display in this interface.

The online version has a couple of extra features. First is a project management sidebar, similar to the package explorer in Eclipse[5]. This is necessary, since the project files are stored online, and the user has no other way to add and delete project files. In the stand-alone version, the project files are stored on the user's machine, allowing them to manage the files directly using the interfaces provided by their operating system.

Also in the online version, new users can get started by cloning a project from the panel in the lower-right. These projects demonstrate many common programming idioms in TurKit. Users may modify their cloned version of these projects to suit their own needs. There is also a link to the TurKit API for reference. Note that the API is the same for the offline version.

3.2 Using TurKit

The idea of TurKit is to abstract human computation away as a regular function call within a program. Let's look at an example of TurKit in action, using the simple toy scenario at the beginning of this chapter, namely, we want to generate a sorted list of five neat things to do in New York City. Our algorithm might begin by soliciting five things to see in New York as follows:

```
mt.init(...)
ideas = []
for (var i = 0; i < 5; i++) {
    idea = mt.promptMemo("What's fun to see in New York City?")
    ideas.push(idea)
}
```

[5]http://www.eclipse.org

We can guard against multiple people generating the same idea by showing each worker the ideas that we have so far like this:

```
idea = mt.promptMemo("What's fun to see in New York City?" +
                     "Ideas so far: " + ideas.join(", "))
```

Now JavaScript has a sorting function that lets us supply our own comparator. Hence, we can sort our list as follows (note that the comparator function returns -1 if the first argument should be sorted before the second argument):

```
ideas.sort(function (a, b) {
    var better = mt.voteMemo("Which is better to see?", [a, b])
    return better == a ? -1 : 1
})
```

This code will work, but it may take longer than it needs to, because it will ask for all of the comparisons sequentially. Many sorting algorithms allow some of the comparisons to be done in parallel, e.g., quicksort. Here is code for quicksort, and we'll see next how to parallelize it:

```
function quicksort(a, comparator) {
    if (a.length == 0) return []
    var pivot = Math.floor(a.length * Math.random())
    var left = [], right = []
    for (var i = 0; i < a.length; i++) {
        if (i == pivot) continue
        if (comparator(a[i], a[pivot]) < 0)
            right.push(a[i])
        else
            left.push(a[i])
    }
    left = quicksort(left)
    right = quicksort(right)
    return left.concat([a[pivot]]).concat(right)
}
```

We can now make this a parallel quicksort by adding **fork** in a couple of places. First, we want to perform all the comparisons with a given pivot in parallel, since these do not depend on each other's outcomes. Then, we can perform quicksort of the left and right subsequences in parallel as well, since these do not depend on each other. This gives us:

```
function quicksort(a, comparator) {
    if (a.length == 0) return []
    var pivot = Math.floor(a.length * Math.random())
    var left = [], right = []
    for (var i = 0; i < a.length; i++) {
        if (i == pivot) continue
        fork(function () {
            if (comparator(a[i], a[pivot]) < 0)
                right.push(a[i])
            else
                left.push(a[i])
        })
    }
    join()
    fork(function () { left = quicksort(left) })
    fork(function () { right = quicksort(right) })
    join()
    return left.concat([a[pivot]]).concat(right)
}
```

Note the use of `join` in this example. We need to join after we compare everything with the pivot to ensure that `left` and `right` are in their final states before passing them recursively to quicksort, otherwise subsequent runs of the program may pass a different version of `left` or `right`, depending on what has been completed on MTurk. This would introduce non-determinism in the program. To see this, note that the pivot selection is based on the length, which may be different each time if we do not first `join`. Likewise, we need to `join` before concatenating the results and returning them, otherwise this too would be a source of non-determinism.

On one hand, this example shows how easy it is to incorporate human computation calls into an otherwise ordinary imperative program. On the other hand, it is important to note that these abstractions create a programming model with some subtle peculiarities. These issues arise due to the crash-and-rerun programming

paradigm, and the parallel programming abstractions. We will discuss these issues in the discussion section at this end of this chapter.

3.2.1 Iterative Improvement

Let's look at a less contrived example of a TurKit program. One of the initial uses of TurKit was iteratively improving some artifact, like a paragraph of text. The workflow begins by asking one turker to write a paragraph with some goal. The workflow then shows the paragraph to another person, and asks them to improve it. The workflow also has people vote between iterations, so that we eliminate contributions that don't actually improve the paragraph. This process is run for some number of iterations. Here is a simple version of such an algorithm:

```
// generate a description of X
// and iterate it N times
var text = ""
for (var i = 0; i < N; i++) {
    // generate new text
    var newText = mt.promptMemo(
        "Please write/improve this paragraph describing " +
        X + ": " + text)

    // decide whether to keep it
    if (mt.voteMemo("Which describes " + X + " better?",
        [text, newText]) == newText) {
        text = newText
    }
}
```

The high-level utility methods make this code fairly simple, though in practice, our requirements often deviate from the features provided by the utility methods, and we need to write custom web pages. For instance, we may want to generate a description of an image, but the current `prompt` function does not support images. Also, we may want to randomize the order in which the voting options are presented, but this is not supported by the current `vote` function. These particular improvements are

probably worth adding to the utility methods directly, and we are working to do so. In the meantime, TurKit programmers can get maximum control over their interfaces by creating web pages, as discussed previously.

3.3 External Uses of TurKit

3.3.1 Decision Theory Experimentation

TurKit has been used to coordinate a user study in a Master's thesis outside our lab by Manal Dia: "On Decision Making in Tandem Networks" [19]. The thesis presents a decision problem where each person in a sequence must make a decision given information about the decision made by the previous person in the sequence. Dia wanted to test how well humans matched the theoretical optimal strategies for this decision problem. Dia used TurKit to simulate the problem using real humans on MTurk, and ran 50 trials of the problem for two conditions: with and without an option of "I don't know". The first condition replicated the findings of prior results that used classroom studies. The second condition, which had not been attempted in prior work, found some interesting deviations in human behavior from the theoretical optimal strategy.

Dia found TurKit helpful for coordinating the iterative nature of these experiments. However, she did not discover the parallelization features of the tool, which would have made her tasks even easier. She was using an early version of TurKit where the parallelization feature was called `attempt` rather than `fork`. This and other reasons discussed toward the end of this chapter may have made parallelization difficult to discover and use.

3.3.2 Psychophysics Experimentation

Phillip Isola, a PhD student in Brain and Cognitive Science, used TurKit to explore psychophysics. He was interested in having turkers collaboratively sort, compare, and classify various stimuli, in order to uncover salient dimensions in those stimuli.

For instance, if turkers naturally sort a set of images from lightest to darkest, then we might guess that brightness is a salient dimension for classifying images. This work is related to the staircase-method in psychophysics, where experimenters may iteratively adjust stimuli until it is on the threshold of what a subject can perceive [14].

His experiments involved using TurKit to run genetic algorithms where humans perform both the mutation and selection steps. For instance, he evolved pleasant color palettes by having some turkers change various colors in randomly generated palettes, and other turkers select the best from a small set of color palettes.

Isola found TurKit to be the right tool for these tasks, since he needed to embed calls to MTurk in a larger algorithm. However, he too found the parallelization features difficult to discover in the old version of TurKit where `fork` was named `attempt`.

3.3.3 Soylent

Michael Bernstein, a member of our group, used TurKit to create Soylent [5]. Soylent is a plugin for Microsoft Word that allows people to outsource various tasks directly from Word. These tasks include summarizing a passage of text, proofreading some text, and performing a macro on each sentence or paragraph in some text (i.e. having people convert each sentence to present tense). Each of these processes uses a *find-fix-verify* algorithm, which consists of three phases. This algorithm is expressed as a TurKit script, which runs in a separate process from Word.

3.3.4 VizWiz

Jeffery Bigham, while working at our lab, started work on VizWiz [7], a system that allows blind users to take a picture of an object with their iPhone, and have several turkers identify the object. His initial prototypes of the system used TurKit, though mostly as a convenient API for accessing MTurk. Jeff ultimately created a branch of TurKit called quickTurKit, which is designed to get tasks done very quickly by

having turkers working on tasks all the time, usually old tasks for which the answers are already known, but if a new task comes in, then it is inserted into the flow of a worker who is already doing tasks.

3.4 Performance Evaluation

This thesis claims that the crash-and-rerun programming model is good for prototyping algorithmic tasks on MTurk, and that it sacrifices efficiency for programming usability. One question to ask is whether the overhead is really as inconsequential as we claim, and where it breaks down.

We consider a corpus of 20 TurKit experiments run over the span of a year, including: iterative writing, blurry text recognition, website clustering, brainstorming, and photo sorting. These experiments paid turkers a total of $364.85 for 29,731 assignments across 3,829 HITs.

Figure 3-3 shows the round-trip time-to-completion for a subset of these HITs. These include 2648 HITs, each with a 1-cent reward, which tend to be faster than our higher paying tasks. The average is 4 minutes, where 82% take between 30 seconds and 5 minutes. Five complete within 10 seconds. The fastest is 7 seconds.

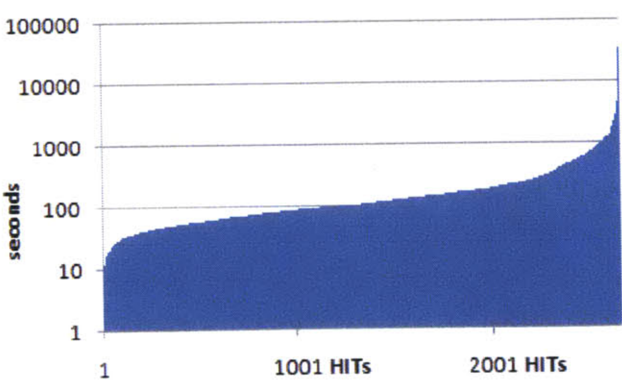


Figure 3-3: Time until the first assignment is completed for 2648 HITs with 1 cent reward. Five completed within 10 seconds.

Figure 3-4 gives a sense for how long TurKit scripts take to rerun given a fully recorded execution trace, in addition to how much memory they consume. Both of

these charts are in terms of the number of HITs created by a script. Note that for every HIT created, there is an average of 6 calls to `memo`, and 7.8 assignments created. The largest script in our corpus creates 956 HITs. It takes 10.6 seconds to rerun a full trace, and the database file is 7.1MBs. It takes Rhino 0.91 seconds to parse and load the database into memory, where the database expands to 25.8MBs.

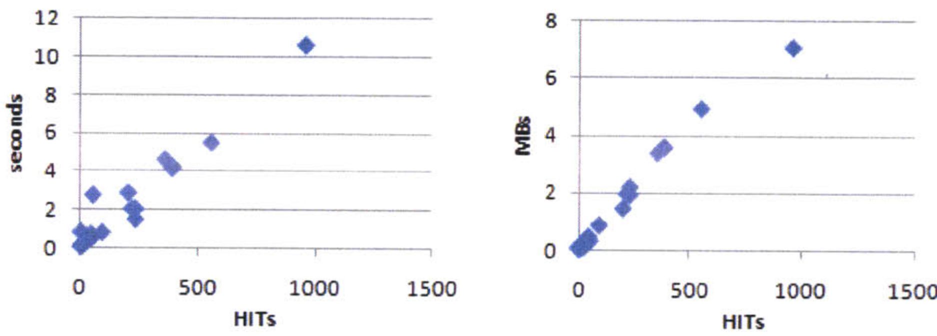


Figure 3-4: Time and space requirements for 20 TurKit scripts, given the number of HITs created by each script.

This means that waiting for a single human takes an order of magnitude longer than running most of our scripts, which suggests that crash-and-rerun is suitable for many applications. The slowest script is faster than 99% of our hit-completion times. Note that making the script 10x slower would still be faster than 70% of hit-completion times. For such a slow script, it may be worth investigating options beyond crash-and-rereun. Even in those cases, however, crash-and-rerun may be a good way to prototype an algorithm before implementing the final algorithm using another paradigm.

3.5 Discussion

We have iterated on TurKit for over a year, and received feedback from a number of users, including four in our group, and two outside our group. This section discusses what we've learned, including some limitations of TurKit, as well as unanticipated benefits.

3.5.1 Limitations

Fixable Limitations

TurKit tries to make it easy to write simple human computation scripts, but users have uncovered a number of usability issues over the course of its development. First, even when users know that a script will be rerun many times, it is not obvious that it needs to be deterministic. In particular, it is not clear that `Math.random` must generally be wrapped in `memo`. This led us to memoize the random seed to `Math.random`, so that all calls to `Math.random` are effectively memoized.

Users were also often unclear about which aspects of a TurKit script were memoized, and which parts could be modified or re-ordered. This was due primarily to the fact that many functions in TurKit would call `memo` internally, such as `createHIT` and `waitForHIT`. We mitigated this problem by adding a view of the execution trace to the GUI, trying to make clear which aspects of the script were recorded. More recently, we also appended the word "memo" to all functions that automatically memoize their results, giving us functions like `createHITMemo` and `waitForHITMemo`.

Finally, many early TurKit users did not know about the parallel features of TurKit. Multiple users asked to be able to create multiple HITs in a single run, and were surprised to learn that they already could. The relevant function used to be called `attempt`, a poor naming choice based on implementation details, rather than the user's mental model. We renamed this function to `fork`. We also added `join`, since most uses of the original `attempt` function would employ code to check that all of the attempts had been successful before moving on.

Inherent Limitations

TurKit's crash-and-rerun programming model tries to make the script look like an ordinary imperative program. The abstractions all suggest to the user that they can think about their program as if it only runs once. However, there are subtle ways in which the crash-and-rerun programming model yields different results than a normal

imperative program would. In particular, you can modify the program between runs, but you cannot change the order that things happen in. For instance, if you have:

```
a = mt.createHITMemo(...)
b = mt.createHITMemo(...)
```

you cannot change this to:

```
b = mt.createHITMemo(...)
a = mt.createHITMemo(...)
```

and expect a and b to have the same values as before, since the memoization is based on the order in which the `mt.createHITMemo`'s are encountered, and not on the variable that the result is being assigned to.

Note that the reordering issue can crop up in tricky ways. For instance, in our New York example we created a list of five items, and then tried to sort them. If we instead decided to collect ten items, we could not simply increase the iteration count to ten, because the 6th iteration would be confused with the first HIT in the sorting part of the program. Note that we could in fact raise the iteration count to ten as long as the sorting part of the program had not yet begun.

TurKit does try to detect cases where memoized items have been reordered, based on the function being memoized, but this method does not work for catching the error above, since in both cases, it is the `mt.promptMemo` function being memoized. However, it could in theory catch it since the arguments are different, i.e., the second call to `mt.promptMemo` includes the idea generated from the first call. We plan to add this feature to TurKit, however it is not enough to cover all cases. For instance, users could pass implicit arguments to a function through a global variable. Also, users may call `mt.promptMemo` with exactly the same arguments, but care what *time* each HIT was placed on MTurk. In these cases, the user just needs to understand how the system works, and be careful.

Parallel Programming Model

The crash-and-rerun parallel programming model is interesting. In some ways, it seems safer than true multi-threading, since variables are guaranteed not to change between accessing them and setting them to a new value during a given run of the program. However, since crash-and-rerun programs execute many times, many of the same race conditions present in truly multi-threaded environments arise in crash-and-rerun programs as well, with different consequences. Consider the following program:

```
x = 0
fork(function () {
    x = mt.promptMemo("enter a number:")
})
if (x % 2 == 0) { ...A... } else { ...B... }
```

In a truly multithreaded program, it is unclear whether the fork will finish before the condition of the if-statement is evaluated. Hence, the program could enter branch `A` with an odd value of `x`, if `x` is updated right after the condition is evaluated. This cannot happen in a crash-and-rerun program — `x` will always be even inside branch `A`, and odd in branch `B`.

However, crash-and-rerun parallelism suffers from a different problem in this case. The program might execute *both* branches `A` and `B`, on different runs of the program. The first time the program runs, it will post a HIT on MTurk asking someone to enter a number. It will then probably throw an exception, since nobody has entered a number yet. The `fork` will catch the exception, and the program will continue execution after the `fork` where it will enter branch `A`. The next time the program runs, someone may have entered an odd number, in which case the program will enter branch `B`. This might cause problems if both branches `A` and `B` execute code with side-effects, like posting HITs on MTurk.

This is a contrived example, and is easy to fix in both the truly multi-threaded and crash-and-rerun models. The most sensible fix is probably to remove the `fork` entirely, or to add a `join` after it. However, let's look at a more realistic example.

Consider a parallel sorting algorithm that is even more efficient than the parallel quicksort we described above: a parallel tree sort. We'll use a simple implementation of tree sort, without a self-balancing tree, relying on non-determinism within how tasks are chosen on MTurk to keep things random. Here is an attempt at coding this algorithm in TurKit:

```
function insertIntoTree(e, tree, comparator) {
    if (!tree.value) {
        tree. value = e
    } else if (comparator(e, tree.value) < 0) {
        if (!tree.left) tree.left = {}
        insertIntoTree(e, tree.left, comparator)
    } else {
        if (!tree.right) tree.right = {}
        insertIntoTree(e, tree.right, comparator)
    }
}
function treesort(a, comparator) {
    var tree = {}
    for (var i = 0; i < a.length; i++) {
        fork(function () {
            insertIntoTree(a[i], tree, comparator)
        })
    }
    return inorderTraversal(tree)
}
```

Unfortunately this does not work. The problem is that the `tree` data structure is initialized to be empty each time the program runs, which means that modifications made to the tree by later `forks` will not be visible to earlier forks when the program reruns. A similar problem would arise with true multi-threading, since multiple threads would be accessing the same tree data structure. The solution there would involve the use of critical sections. As far as I know, the crash-and-rerun model has no simple solution using just `fork` and `join`, meaning that this parallelism model is not completely general. It is possible to get around this in TurKit by persisting values in `db` directly, but this breaks outside the simple abstraction we were hoping to provide.

Another possibility involves representing forks with coroutines, and memoizing the order that the coroutines make progress, so that the runtime system could reproduce this order when the program reruns. This option should be a small matter of coding, but is left to future work.

Scalability

The crash-and-rerun model favors usability over efficiency, but does so at an inherent cost in scalability. Consider the following program:

```
a = "starting sentence"
while (true) {
    a = mt.promptMemo("what does this make you think of?: " + a)
    print(a)
}
```

This program prints an infinite sequence of thoughts based on other thoughts. However, the crash-and-rerun programming model will need to rerun all previous iterations of the loop every time it re-executes, and eventually the space required to store this list of actions in the database will be too large. Alternatively, the time it takes to replay all of these actions will grow longer than the time it takes to wait for a HIT to complete, in which case it may be better to poll inside the script, rather than rerun it.

Possible Fix To Programming Model

One way to overcome many of these problems, including the parallel programming complexity and the scalability issue, is to serialize continuations between runs of the program, rather than memoizing specific actions. Coroutines could be used to achieve parallelism, as we mentioned before. Rhino supports first-class continuations, which provide the ability to save and serialize the state of a running script, even along multiple paths of execution. Essentially, the "crash" exception could be replaced with a "yield" method.

This programming model is conceptually elegant, and does keep TurKit's advantage of being able to run in environments where continuous execution is not possible, e.g., a laptop that is turned off over night, or Google App Engine where all execution must happen within 30 seconds chunks. However, it gives up some of the advantages that we will discuss in the next section, in particular, the ability to modify a program between runs, which turns out to be a huge advantage for debugging.

3.5.2 Benefits

Despite the limitations, crash-and-rerun programming has some nice advantages for prototyping human computation algorithms. First is incremental programming. When a crash-and-rerun program crashes, it is unloaded from the runtime system. This provides a convenient opportunity to modify the program before it is executed again, as long as the modifications do not change the order of important operations that have already executed. TurKit programmers can take advantage of this fact to write the first part of an algorithm, run it, view the results, and then decide what the rest of the program should do with the results. This allows a programmer to incrementally develop a program.

The second benefit is that crash-and-rerun is easy-to-implement. It does not require any special runtime system, language support, threads or synchronization. All that is required is a database to store a sequence of results from calls to `memo`. In fact, we are aware of at least one other lab implementing the crash-and-rerun model in Python.

The final benefit is retroactive print-line-debugging. In addition to adding code to the end of a program, it is also possible to add code to parts of a program which have already executed. This is true because only expensive or non-deterministic operations are recorded. Innocuous operations, like printing debugging information, are not recorded, since it is easy enough to simply re-perform these operations on subsequent runs of the program. This provides a cheap and convenient means of debugging in which the programmer adds print-line statements to a program which has already executed, in order to understand where it went wrong. This technique

can also be used to retroactively extract data from an experiment that a programmer forgot to print out on the first run.

Experimental Replication

The crash-and-rerun model offers a couple of interesting benefits for experimental replication using MTurk. First, it is possible to give someone the source code for a completed experiment, along with the database file. This allows them to rerun the experiment without actually making calls to MTurk. In this way, people can investigate the methodology of an experiment in great detail, and even introduce print-line statements retroactively to reveal more information.

Second, users can use the source code alone to rerun the experiment. This provides an exciting potential for experimental replication where human subjects are involved, since the experimental design is encoded as a program. Most of the experiments discussed in the next chapter are posted on the Deneme[6] blog, along with the TurKit code and database needed to rerun them.

3.6 Conclusion

While the TurKit programming model has some subtle complexities, we believe that the pros and cons yield a net benefit over more straight forward approaches to writing human computation algorithms. It is worth exploring alternative models as well, but the one benefit we would advocate for any model is the ability to reuse human computation results between runs of a program. This ability seems to dramatically improve the iterative design cycle of debugging and refining a human computation algorithm.

In fact, it may be nice to have more fine grained control over which parts of the human computation are reused. For instance, it may be that the first part of an algorithm yields good results, but then the next part of the algorithm does not.

[6]http://groups.csail.mit.edu/uid/deneme/

We may want to change the second half of the algorithm and rerun that part fresh, without forgetting the results from the first part.

This is technically possible in TurKit, if the programmer knows how to modify the JSON database file. In fact, I have made use of this ability in some of my programming, as has a labmate. It may be nice, as future work, to add a GUI interface for programmers to remove or reorder items in the execution trace.

Chapter 4

Experiments

We see many examples on the web of people contributing small bits of work to achieve something great, like Wikipedia. It would be nice to understand these processes well enough to encode them in human computation algorithms. One difference we notice between human computation processes on the web is between *iterative* and *parallel*[1]. For instance, if we look at how an article is written on Wikipedia, we often see an iterative process. One person starts an article, and then other people iteratively improve it by looking at what people did before them and adding information, correcting grammar or creating a consistent style. On the other hand, designs on Threadless, a t-shirt design site, are generally created in parallel. Many people submit ideas independently, and then people vote to determine the best ideas that will be printed on a t-shirt.

We are interested in understanding the tradeoffs between each approach. The key difference is that the iterative process shows each worker results from previous workers, whereas the parallel processes asks each worker to solve a problem alone. The parallel process can parallelized because workers do not depend on the results of other workers to solve problems, whereas the iterative process must solicit contributions serially.

[1]Parts of this chapter are published in a workshop paper at HCOMP 2010 titled "Exploring Iterative and Parallel Human Computation Processes." [37]

To study the tradeoffs between each approach, we create a simple model wherein we can formally express the iterative and parallel processes. We then run a series of controlled experiments, applying each process to three diverse problem domains: writing image descriptions, brainstorming company names, and transcribing blurry text. These experiments are run using MTurk as a source of on-demand labor. This allows us to run the experiments multiple times, increasing statistical validity. Each process is coordinated automatically using TurKit, described in the previous chapter.

Our results show that iteration improves average response quality in the writing and brainstorming domains. We observe an increase in the transcription domain as well, but it is not statistically significant, and may be due to other factors. In the case of writing, the increase in quality comes from the ability of workers to collaboratively write longer descriptions. We also observe a few downsides for iteration. In the case of brainstorming, workers riff on good ideas that they see to create other good names, but the very best names seem to be generated by workers working alone. In transcription, showing workers guesses from other workers can lead them astray, especially if the guesses are self-consistent, but wrong.

Overall, prototyping and testing these processes on MTurk provides insights that can inform the design of new processes. This chapter proceeds by offering a model and simple design patterns for human computation processes that coordinate small paid contributions from many humans to achieve larger goals. We then present a series of experiments that compare the efficacy of parallel and iterative design patterns in a variety of problem domains. We end with a discussion of the tradeoffs observed between the iterative and parallel approaches.

4.1 MODEL

We are interested in human computation workflows that coordinate small contributions from many humans to achieve larger goals. For example, a workflow might coordinate many workers to write a description for an image. All of the creative and

problem solving power in these workflows will come from humans, e.g., humans will do the writing.

Typical user generated content, like image descriptions, comes in a form that the computer does not understand. In order for the human computation workflow to make decisions, it will need to ask humans to evaluate, rate, or compare content such that the result is a number, boolean, or other data representation that a computer can readily process. This suggests a breakdown of the domain into *creation* and *decision* tasks.

4.1.1 Creation Tasks

When a worker writes a description for an image, this is a creation task. Creation tasks solicit new content: writing, ideas, imagery, solutions. Tasks of this form tend to have few constraints on worker inputs to the system, in part because the computer is not expected to understand the input. The goal of a creation task is to produce new high quality content, e.g., a well written and informative description for an image.

Many factors affect the quality of results. We are interested in exploring the potential benefits of iteration, where each worker is shown content generated by previous workers. The hope is that this content will serve as inspiration for workers, and ultimately increase the quality of their results.

4.1.2 Decision Tasks

If we have two descriptions for the same image, we can use a decision task to let the process know which is best. Decision tasks solicit opinions about existing content. Tasks of this form tend to impose greater constraints on worker contributions, since

the computer will need to understand the contributions. The goal of a decision task is to solicit an accurate response. Toward this end, decision tasks may ask for multiple responses, and use the aggregate. In our experiments, we use two decision tasks: comparisons and ratings. The comparison tasks show a worker two items, and ask them to select the item of higher quality. The order of items is always randomized in our comparison tasks.

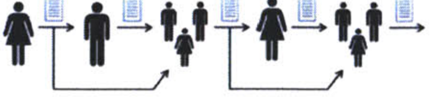
The rating tasks show a worker some content, and ask them to rate the quality of the content, with respect to some goal, e.g., "Please rate the quality of this text as a description of the image." All rating tasks in this paper use a rating scale from 1 to 10.

4.1.3 Combining Tasks: Iterative and Parallel

Human computation processes combine basic tasks in certain patterns. At this point, the patterns we explore are quite simple. All of the processes in this chapter follow one of two patterns: iterative or parallel.

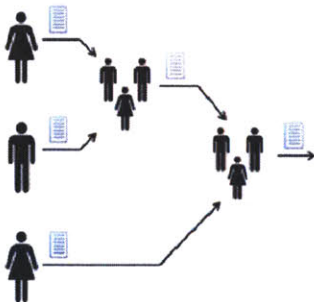
The iterative pattern consists of a sequence of creation tasks, where the result of each task feeds into the next one:

The goal is to explore the benefits of showing workers content generated by previous workers. If it is not easy to merge the results of creation tasks, as is the case for image descriptions, then we place a comparison task between creation tasks:



This comparison lets the computer make a decision about which content to feed into the next task. In our experiments, we want to keep track of the best item created so far, and feed that into each new creation task.

The parallel pattern consists of a set of creation tasks executed in parallel. The parallel pattern acts as a control in our experiments, using the same number of tasks as the iterative pattern, but workers are not shown any work created by others. The results from all workers are merged together using a simple algorithm. If an iterative pattern uses comparison tasks between iterations, then the corresponding parallel algorithm uses the same sequence of comparisons to find the best result after all the results have been generated in parallel:



Our experiments also feed all subjective contributions into rating tasks in order to compare the effectiveness of each process.

4.2 EXPERIMENTS

In this section, we describe three experiments run on MTurk which compare the parallel and iterative processes in three different problem domains: writing, brainstorming, and transcription. Each domain was chosen because it solves a useful problem, and we wanted the combination to span a number of problem solving dimensions. For instance, the problems impose different levels of constraints: the transcription task has a definite answer, whereas the writing and brainstorming tasks are more open ended. The domains also vary in the size of each unit of work, ranging from single words to whole paragraphs.

4.2.1 Writing Image Descriptions

This experiment compares the parallel and iterative processes in the context of writing image descriptions. The experiment is inspired by Phetch [58], a game where humans

write and validate image descriptions in order to make images on the web more accessible to people who are blind. This is a different approach to the same problem that may be applicable to a greater variety of writing contexts, since it does not rely on the writing being fun.

Each process has six creation tasks, each paying 2 cents. Five comparison tasks are used in each process to evaluate the winning description. Each comparison task solicits five votes, each for 1 cent. A creation task is shown in Figure 4-1. The task asks a turker to describe the image factually in at most 500 characters. A character counter is updated continuously as the user types. The "Submit" button only activates when the content of the text area has changed from the initial text and there are at most 500 characters. Note that the instruction about "using the provided text" appears only in creation tasks that have text from a previous iteration to show. This instruction is omitted in all the parallel tasks.

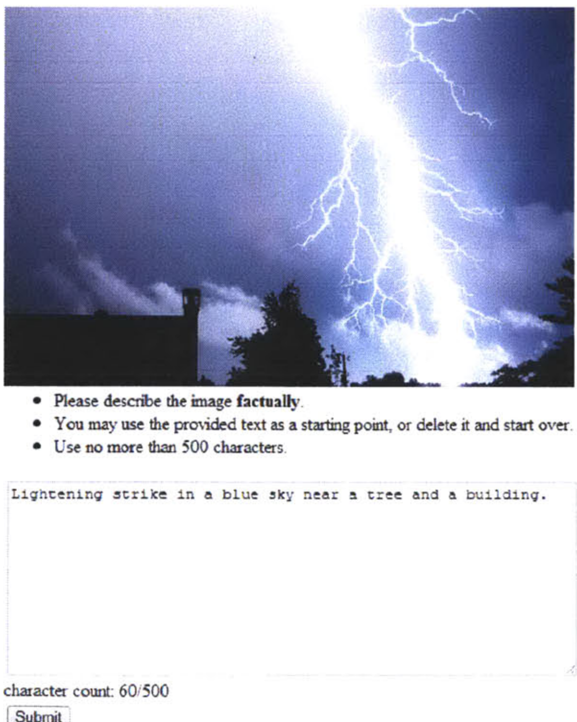


Figure 4-1: This creation task asks for a descriptive paragraph, showing a description written by a previous worker as a starting point.

A comparison task in shown in Figure 4-2. This task presents a worker with two paragraphs. The presentation order is randomized for each worker, and the differences between the paragraphs are highlighted. We highlight the differences in case they are hard to find, e.g., if a worker in the iterative process makes a very small change to a description. To be consistent, the comparison tasks in the parallel condition also highlight the differences between descriptions, even though they are unlikely to have much in common.

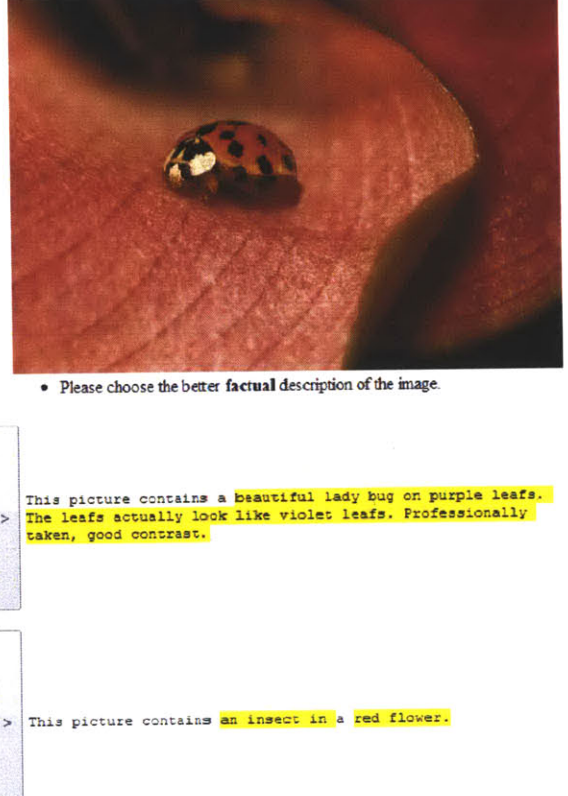


Figure 4-2: This decision task asks a worker to compare two image descriptions, and pick the one they think is better. The choices are randomized for each worker.

To compare the iterative and parallel processes, we selected 30 images from www.publicdomainpictures.net. Images were selected based on having interesting content, i.e., something to describe. We then ran both the parallel and iterative process on each image. For a random half of the images, we ran the parallel process first, and for the other half, we ran the iterative process first.

In order to compare the results from the two processes, we created a rating task. Turkers were shown an image and a description, and asked to rate the quality of the description as a factual description of the image, on a scale of 1 to 10. This task is shown in Figure 4-3. We obtained 10 ratings for each image description to compute an average rating.

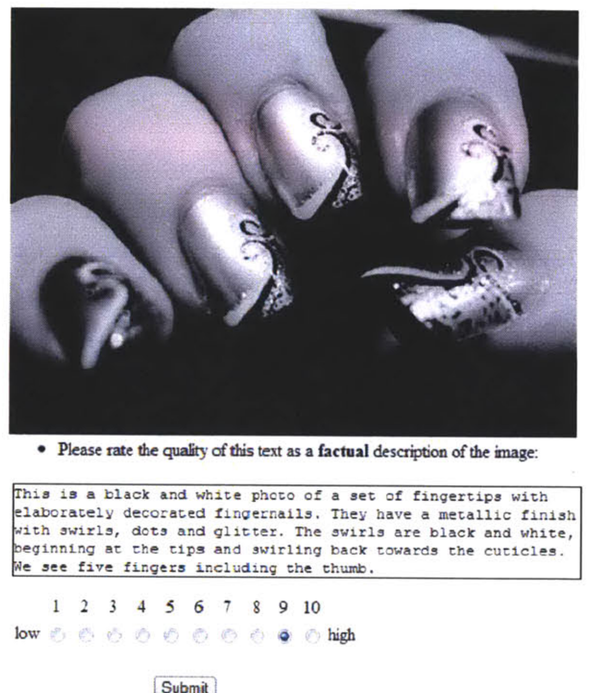


Figure 4-3: This decision task asks a worker to rate the quality of an image description on a scale from 1 to 10.

Turkers were not allowed to participate in both processes for a single image. They were also not allowed to rate descriptions for images that they contributed any writing to. However, turkers were allowed to contribute to multiple images, as well as rate multiple descriptions for the same image.

Our hypothesis was that the iterative process would produce better results. We reasoned that workers would be willing to spend a constant amount of time writing a description, and they could do more with that time if they had a description to start from.

Results & Discussion

Figures 4-4 and 4-5 show a sample of raw results from the experiment. The first figure shows all six iterations of both the iterative and parallel processes for two images. The next figure zooms out a bit and shows the output description for six of the thirty images in the experiment.

In most of these examples, the ratings are higher for the iterative process. In fact, if we average the ratings of resulting descriptions in each process for all 30 images, we get a small but statistically significant difference in favor of iteration (7.9 vs. 7.4, paired t-test $T_{29} = 2.1$, $p = 0.04$). Figure 4-6 shows what the result would have been if we had run the process for n iterations. Note that the two processes are identical when we use only one iteration.

It is worth noting that there is a correlation between description length and rating: longer descriptions are given higher ratings, accounting for about 30% of the variability in ratings according to the linear regression in Figure 4-7 ($\mathrm{R}^2 = 0.2981$, $N = 360$, $\beta = 0.005$, $p < 0.0001$). This makes sense since we asked for "factual descriptions", and longer descriptions can hold more facts and details. The two circled outliers indicate cases of text copied from the internet that was only superficially related to the image.

This correlation is relevant because the iterative process produces longer descriptions, about 336 characters on average compared with 241 characters in the parallel process. This difference is enough to explain the difference we see in ratings. We can see this with a multiple regression model using two input variables: an indicator variable for the process type (iterative or parallel), and length. The dependent variable is the rating. When we train the model, length is a significant predictor of the rating ($p < 0.001$) while process type is not ($p = 0.52$).

However, note that the final descriptions outputted by each process do appear to be better than length alone would predict. We can see this with a similar multiple regression model from before, except over all the descriptions, and not just the final output descriptions. Again, we will use two input variables: an indicator variable for

Iterative
Lightening strike in a blue sky near a tree and a building. *(rating 6.9, 53 seconds)*
The image depicts a strike of fork lightening, striking a blue sky over a silhoutted building and trees. *(4/5 votes, rating 7.9, 112 seconds)*
The image depicts a strike of fork lightning, against a blue sky with a few white clouds over a silhouetted building and trees. *(5/5 votes, rating 8.6, 112 seconds)*
The image depicts a strike of fork lightning, against a blue sky- wonderful capture of the nature. *(1/5 votes, rating 6.8, 74 seconds)*
This image shows a large white strike of lightning coming down from a blue sky with the tops of the trees and rooftop peaking from the bottom. *(3/5 votes, rating 8.4, 183 seconds)*
This image shows a large white strike of lightning coming down from a blue sky with the silhouettes of tops of the trees and rooftop peeking from the bottom. The sky is a dark blue and the lightening is a contrasting bright white. The lightening has many arms of electricity coming off of it. *(winner, 4/5 votes, rating 8.7, 80 seconds)*
Parallel
A lightning in the sky. *(rating 5.8, 89 seconds)*
A branching bolt of lightening in front of a blue, partly-cloudy sky. The roof of a house with a small chimney and the tops of a few trees are silhouetted against the sky. *(4/5 votes, rating 8.3, 129 seconds)*
A white thunder in the sky with the cloudy sky and the darknest of the ground. *(1/5 votes, rating 6.1, 155 seconds)*
White lightning n a root-like formation shown against a slightly wispy clouded, blue sky, flashing from top to bottom. Bottom fifth of image shows silhouette of trees and a building. *(winner, 3/5 votes, rating 7.2, 334 seconds)*
picture with a bolt of lightning lighting up the sky. *(2/5 votes, rating 6.6, 112 seconds)*
very realistic.great brightness contrast. *(0/5 votes, rating 3, 227 seconds)*

Iterative
This picture contains an insect in a red flower. *(rating 5.6, 207 seconds)*
This picture contains a beautiful lady bug on purple leafs. The leafs actually look like violet leafs. Professionally taken, good contrast. *(2/5 votes, rating 6.2, 241 seconds)*
This picture contains a close-up shot of a single ladybug (red with black spots) on the petal of a pinkish-red flower. There is another petal in the background. *(winner, 5/5 votes, rating 8.8, 223 seconds)*
A red and black ladybug crawling across two pink flower petals. *(0/5 votes, rating 7.5, 77 seconds)*
There is an orange-red ladybug with black spots on a pinkish, lined surface that may be a leaf of some kind, that curls up at the edge. A darker red "leaf" rests below and to the right. *(1/5 votes, rating 7.2, 258 seconds)*
it is a ladybug or some sort of insect eating a leaf or something on a red leaf.it is not that interesting of a picture though. *(1/5 votes, rating 5.2, 72 seconds)*
Parallel
ladybug on red flower leave *(rating 6, 32 seconds)*
Ladybug on red leave *(1/5 votes, rating 5.3, 65 seconds)*
There are 2 red colored leaves with prominent veins. There is a lady bug on one of the leaves. This leaf is slightly bent in one section. *(winner, 5/5 votes, rating 8.3, 163 seconds)*
A bug in a leaf *(0/5 votes, rating 5.5, 105 seconds)*
The small bug on the top of the red petal of a flower. *(1/5 votes, rating 6.9, 76 seconds)*
very nice picture.ladybird on some red flower. *(0/5 votes, rating 5.9, 233 seconds)*

Figure 4-4: This figure shows all six descriptions generated in both the iterative and parallel processes for two images, along with various statistics including: the number of votes in favor of each description over the previous best description, the average rating for each description, and the number of seconds a turker spent crafting each description. The green highlights show the winning description for each process, i.e., the output description.

	Iterative	Parallel
	A photo of the white marble statue of Abraham Lincoln seated in a large chair at the Lincoln Memorial in Washington, D.C. At the foot of the statue is a row of wreathes. Two people in uniform are standing to the left next to the wreaths. There is a smooth stone wall behind the statue, containing an inscription above Mr. Lincoln's head. *(rating 8.8, 297 seconds)*	The Lincoln monument, a large white marble statue of a sitting Abraham Lincoln, surrounded at the base by numerous colorful floral wreaths and two park service guards or interpreters. *(rating 7.8, 68 seconds)*
	A cobbled street with half timbered buildings along one side of it. There are plants growing in front of the buildings. To the left is a ivy creeper clad structure. The street slopes steeply upwards and the pavements are narrow and uneven. Outside two of the buildings are small advertising blackboards giving details of food and drink available inside. Red flowers hang from the window ledges. Perhaps jolly old England or another quaint European town. (rating 8.8, 91 seconds)	I see a narrow cobblestone street. The homes seem close together and most are made of brick. I can't see any lawns, but there are some bushes and small trees around the homes. Two have flowering plants. On the left side of the street, I can see the back corner of a car. (rating 8.5, 239 seconds)
	This is a black and white photo of a set of fingertips with elaborately decorated fingernails. They have a metallic finish with swirls, dots and glitter. The swirls are black and white, beginning at the tips and swirling back towards the cuticles. We see five fingers including the thumb. (rating 8.3, 107 seconds)	A black and white photograph of a woman's long fingernails. They have been painted a metallic color and are artfully airbrushed with glitter and black and white swirls. (rating 7.6, 55 seconds)
	Ten, red, shiny, three-dimensional, two-piece question marks are arranged carefully on a shiny white surface. Their reflections are seen on this surface. Eight of the question marks are standing in four groups of two with differing configurations and diminishing size as the distance increases from the focal point. Focus is on the largest and central question mark which stands alone with another in the left foreground that appears to have fallen over or has not yet been placed. (rating 8.3, 1001 seconds)	I see a picture using iconic red question marks across it. There are 10 question marks against a white background. One stands prominately in the center of the screen. On each side of this center question mark are two groups of four question marks. One lone question mark lays on its side towards the bottom of the image. (rating 7.7, 361 seconds)
	We see a cemetery with very old headstones. One headstone has fallen and leans against another. The ground is green and grassy. Trees are in the distance, and are just starting to change color. The sun is starting to go down and is casting long shadows. *(rating 8.1, 230 seconds)*	In this cemetery, there are various sized markers and monuments, most appearing to be rather weathered. Some of the markers are large and and are capped with urns and crosses. Others are small and sit in the shadows of the larger ones. This appears to be a northern setting, with conifers and deciduous trees in the background. *(rating 9.2, 224 seconds)*
	A brown eyed young woman cradles an acoustic guitar in front of her chest. Her head rests against the neck of the guitar. She has long, brown hair and wears a slouchy brown hat, a gray knit long sleeved top, and two silver rings. She is either Caucasian or of Hispanic descnet. Her shirt sleeves extend past her wrists. Her hands are holding the neck of the guitar. *(rating 9.1, 89 seconds)*	A young girl poses with her acoustic guitar. She resembles a young Sandra Bullock with dark eyes, full lips and long dark hair. She wears a newsboy-type cap a bit askew on her head, and a long-sleeve gray top; the sleeves are a bit long and extend past her wrists. *(rating 7.6, 250 seconds)*

Figure 4-5: This figure shows the output descriptions for both the iterative and parallel process on six of the thirty images in this experiment.

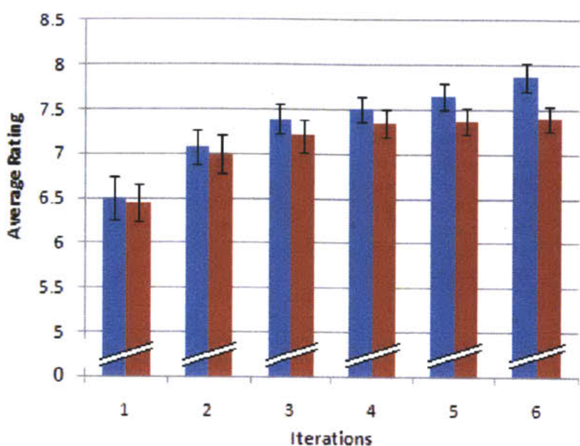


Figure 4-6: The average image description rating after n iterations (iterative process blue, and parallel process red). Error bars show standard error. As we run each process for additional iterations, the gap between the two seems to enlarge in favor of iteration, where the gap is statistically significant after six iterations (7.9 vs. 7.4, paired t-test $T_{29} = 2.1$, $p = 0.04$).

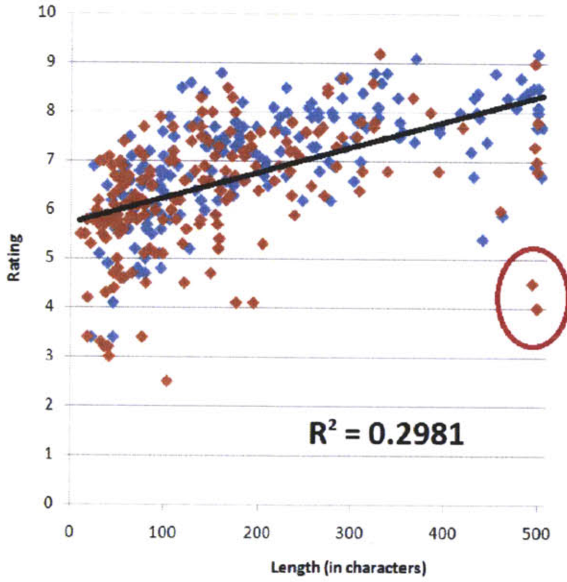


Figure 4-7: Descriptions are plotted according to their length and rating, where iterative descriptions are blue, and parallel descriptions are red. A linear regression shows a positive correlation ($R^2 = 0.2981$, $N = 360$, $\beta = 0.005$, $p < 0.0001$). The two circled outliers represent instances of text copied from the internet.

whether a description is a final description, and length. The dependent variable is still the rating. When we train the model, both the indicator variable and length are significant predictors of the rating ($p < 0.001$). This may be attributable to the fact that we have people vote between descriptions, rather than simply comparing their lengths.

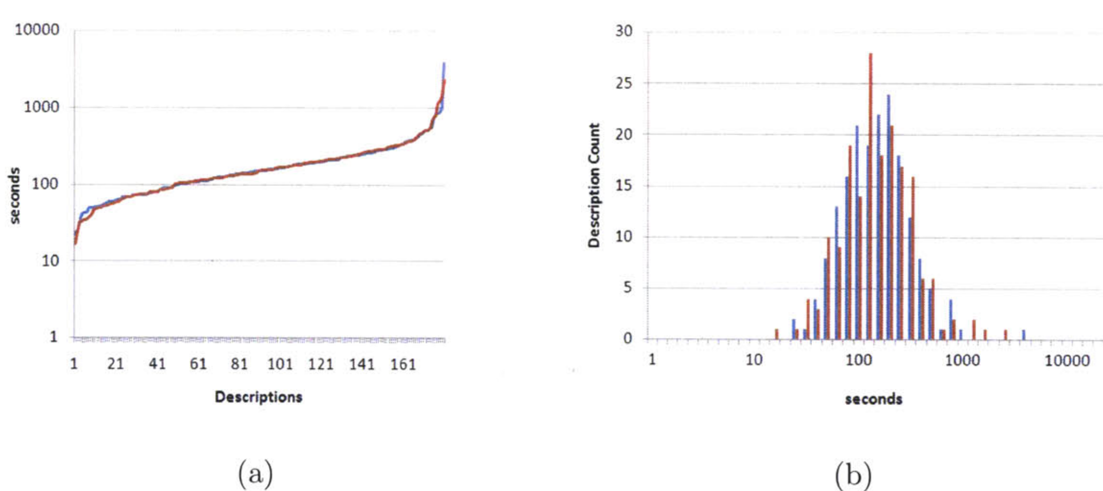
One simple model for what is happening in the iterative process is that it starts with a description of a certain length, and then subsequent turkers add more content. On average, turkers add about 25 characters in each iteration after the initial description. However, the standard deviation is very large (160), suggesting that turkers often remove characters as well. If we look more closely at each of these instances, we can roughly classify their modifications as follows:

- 31% mainly append content at the end, and make only minor modifications (if any) to existing content;
- 27% modify/expand existing content, but it is evident that they use the provided description as a basis;
- 17% seem to ignore the provided description entirely and start over;
- 13% mostly trim or remove content;
- 11% make very small changes (adding a word, fixing a misspelling, etc);
- and 1% copy-paste superficially related content found on the internet.

Time

Note that most modifications (82%) keep some of the existing content and structure, suggesting that these turkers may be doing less work. However, Figure 4-8 shows that the distribution of time spent by turkers writing or improving descriptions is almost identical. Note: when we talk about work time, we mean the time between a worker accepting a task and submitting a result. We do not actually record what turkers are doing.

This figure also suggests that work times are log-normally distributed. Hence, we will report times using geometric means and geometric standard deviations. Arithmetic means are not reported because they are highly sensitive to the presence of large data points. Note that a geometric standard deviation of x means that 68.2% of the data falls within 1/x and x times the geometric mean, and 95.4% of the data falls within $1/x^2$ and x^2 times the geometric mean. All statistical tests will be reported on log-transformed times.



(a) (b)

Figure 4-8: (a) Time spent working on descriptions for both the iterative (blue) and parallel (red) processes. The distributions of times seem remarkably similar for each process. Note that the y-axis uses a logarithmic scale. (b) A histogram showing how many workers spent various amounts of time on task. The distribution appears normally distributed over a log scale, suggesting that work times are log-normally distributed.

Table 4.1 includes work times for voting and rating tasks. All differences involving voting and rating times are statistically significant in this chart. As may be expected, the voting tasks are faster than the writing tasks. The rating tasks are faster still. We would expect them to be faster based solely on the fact that raters only need to read one description instead of two. However, we might also guess that raters would spend more time deliberating, since they have more options to choose from: ten possible ratings, compared with two possible descriptions.

Another time to consider in these processes is how long the actual process takes to run end-to-end. We do not have good data on this, since the crash-and-rerun

task	geometric mean	geo std dev	N
writing descriptions (iterative)	150.3	2.11	180
writing descriptions (parallel)	150.0	2.18	180
voting	57.6	2.41	1500
rating	45.7	2.31	3600

Table 4.1: Time spent working on various tasks, in seconds. All differences are statistically significant, except between iterative and parallel description writing.

program was rerun at irregular intervals during these experiments. Also, the votes for the parallel processes were done well after the initial descriptions were written. Given these caveats, each iterative process took an average of 6.2 hours to run, where the longest ran in 11.3 hours.

We also have data on how long we waited for different types of tasks to be accepted by workers on MTurk, see Table 4.2. There are a couple of interesting things to notice in this chart. First, the parallel tasks waited much longer to be picked up. This is a common phenomenon on MTurk. What is happening is that when we post a parallel task, we post it as a single HIT with six assignments. The first assignment is accepted in roughly the same time as an iterative task would be accepted, as expected, but then the HIT starts to fall down in the list of most-recently-added HITs, so fewer turkers see it. The other thing to note is that the voting tasks are picked up quicker than the rating tasks, even though we saw above that the voting tasks take longer to complete. This may mean that voting tasks are *perceived* as easier to complete.

task	geometric mean	geo std dev	N
writing descriptions (iterative)*	387.4	5.75	180
writing descriptions (parallel)	1825.4	3.92	180
voting	74.8	4.67	1500
rating*	330.1	3.61	3600
writing descriptions (parallel), first of 6*	371.1	4.27	30
voting, first of 5	18.54	5.50	300
rating, first of 10	48.81	5.22	360

Table 4.2: Time spent waiting for someone to accept various task on MTurk, in seconds. The notation "first of X" refers to the time spent waiting for the first turker to accept a task in a HIT with X assignments. The starred (*) items form a group with no pair-wise statistically significant differences. All other differences are statistically significant.

Participation

Although we impose several restrictions on turkers in this experiment, e.g., a turker is not allowed to vote on a description that they wrote, we do allow turkers to complete multiple tasks. Table 4.3 shows that 391 unique turkers participated in this experiment, where the most active turker completed 142 tasks. Figure 4-9 shows the distribution of how many times each turker participates. The number of tasks done by each turker follows a power law distribution, where about 73% of tasks are completed by 27% of the workers, roughly approximating the standard "80-20" rule. We see a power law again if we zoom in to any set of tasks in our experiment.

There were 48 turkers who participated in both the iterative and parallel processes, and 46 turkers who participated in only one. Both processes had exactly 71 unique turkers, but as far as we can tell, this is a coincidence. The most prolific turker in the iterative condition contributed to 16 image descriptions, versus 7 in the parallel condition. It is possible that this difference is meaningful, based on the way tasks are found by turkers. We discussed above that HITs in our parallel condition take longer to be found, and it could be that the long tail of turkers are the ones who find them. However, we did not see a similar difference in the experiments that follow.

task	unique turkers	tasks done by most active turker
overall	391	142
writing descriptions	94	18
writing descriptions (iterative)	71	16
writing descriptions (parallel)	71	7
voting	263	35
rating	182	142

Table 4.3: Number of unique turkers participating in different task types for this experiment, along with the number of tasks completed by the most prolific turker in each category.

4.2.2 Brainstorming

This experiment compares the iterative and parallel processes in a different domain, namely brainstorming company names. Brainstorming is a popular process whereby

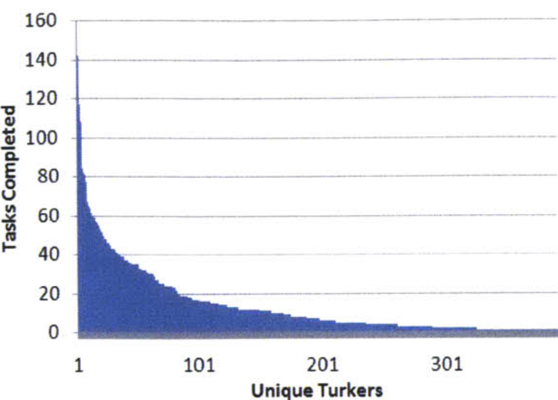


Figure 4-9: A total of 391 unique turkers participated in the image description writing experiment, where the most prolific turker completed 142 tasks.

many people generate ideas, either individually, or in a group. This process is well studied, and Taylor, et. al. [53] suggest that combining the results of individual brainstorms is more effective than having people brainstorm in a group. While group brainstorms typically generate fewer unique names, we try to mitigate this effect by programmatically enforcing that each worker contribute the same number of unique names in each process.

Each process has six creation tasks, each paying 2 cents. The instructions for these tasks are shown in Figure 4-10. The instructions ask a worker to generate five new company name ideas based on the provided company description. The "Submit" button only becomes active when there is text in each of the five input fields. The section that lists "Names suggested so far" only exists in the iterative condition. This list contains all names suggested in all previous iterations for a given company.

We fabricated descriptions for six companies. We then ran both the iterative and parallel process on each company description. As with the previous experiment, we ran the parallel variation first for half of the companies, and the iterative first for the other half. No turkers were allowed to contribute to both the iterative and parallel process of a single company description.

In order to compare the results of these processes, we used the rating technique discussed in the previous experiment to rate each generated company name. Again, we

- **Company details:** Our company sells headphones. There are many types and styles of headphones available, useful in different circumstances, and our site helps users assess their needs, and get the pair of headphones that are right for them.

- Please supply 5 **new company name** ideas for this company.

Submit

Names suggested so far:

- Easy hearer
- Least noisy hearer
- Hearer of silence
- Silence's hearer
- Sharp hearer

Figure 4-10: Turkers are asked to generate five new company names given the company description. Turkers in the iterative condition are shown names suggested so far.

solicited 10 ratings for each company name, and averaged the ratings. Our hypothesis was that the iterative process would produce higher quality company names, since turkers could see the names suggested by other people, and build on their ideas.

Results & Discussion

Figures 4-12 and 4-12 show a sample of raw results from the experiment. The first figure shows all six iterations of both the iterative and parallel processes for two fake companies. The next figure zooms out a bit and shows the highest, middle and lowest rated names for each of the remaining four fake companies.

No turkers in the parallel condition suggested duplicate names, resulting in 180 unique names for this condition. We removed 14 duplicate names in the iterative condition from our data. They could have been prevented in JavaScript, and 13 of the duplicates names came from the last three iterations for a single company. This may have been due to a group of turkers working together who collectively misunderstood the directions, or tried to cheat (unfortunately we did not record IP addresses, so we do not know if these turkers were collocated).

description: Cooking can be difficult for young adults moving out on their own, or starting college. Our site offers real-time consulting services over the internet. People can bring a laptop into the kitchen and chat live with our staff as they cook, asking anything they need, even how to turn on the stove.		**description:** Our company sells headphones. There are many types and styles of headphones available, useful in different circumstances, and our site helps users assess their needs, and get the pair of headphones that are right for them.	
Iterative	**Parallel**	**Iterative**	**Parallel**
Cooking Made Simple (7.1) Cooking Online (7.2) Surfing, Cooking & Browsing (6.1) Chat About Cooking (6.5) Answers 4 Food (7) (291 seconds)	Cooking Concultants (4.4) Ask-a-Chef (6.5) Dial-a-Chef (7.2) Cooking School (6.3) Live Chef (7.6) (167 seconds)	Easy hearer (6.2) Least noisy hearer (5.1) Hearer of silence (5.4) Silence's hearer (5.8) Sharp hearer (6.8) (182 seconds)	hear da music (6.3) head phones r us (4.2) headphones helper (6.4) headphone style (5.6) music brain (8.3) (190 seconds)
Cooking 4 dummies (5.3) Cooking 4 idiots (5.4) Cooking Newbies (5.8) Newbies in Food (7.1) New Cooks helping hand (4.9) (120 seconds)	CookingSimply (6.4) SimpleChef (7.1) YourGourmet (6.2) EasyRecipeasy (5.4) CookingGuide (6.6) (159 seconds)	Music Trend (6.9) Music Maker (5.4) Rocking Music (6.3) Music Tracker (6) Music Explorer (7.1) (245 seconds)	HedPhone (5.8) AheadPhone (6.1) The Right Headphone (6) Headphones Ahead (6.4) Smart Headphones (6.4) (142 seconds)
Easy Cooking (6.3) Cooking Revolution (5.6) Cooks to Chef (5.4) Cook Helper (5.9) Instant Cook (7.1) (222 seconds)	Cooking Live (6.9) Learn to Cook in Your Kitchen (5.9) Cooking Classes Live to Your House (6) Cooking Basics (5.8) Beginners Kitchen Basics and Cooking Live (5.9) (125 seconds)	Easy Listener (6.7) Helpful Headphones (6.2) Music Enhancers (5.9) Right Hearing Headphones (6.1) Sounding Ease (6.7) (192 seconds)	Headshop (7) Nice Ears (5.7) Speak To Me (6.3) Headphone House (7.4) Talkie (6.8) (152 seconds)
the galloping coed (5) stick a fork in me (4.7) foogle (4.7) your just dessert (6.1) the student chef (5) (401 seconds)	Cooking for College Dummies (5.8) Dorm Recipes (5) College Cooking (6) Simple College Meals (5.5) Cooking for Lazy People (5.9) (153 seconds)	Headphone Store (6.8) Headphones R Here (6.7) Styly Headphones (6.5) Shop Headphone (4.8) Headphony (4.9) (144 seconds)	headphones (5.3) different circumstances (3.7) user needs (4.7) company sell (4.3) styles of headphones (5) (132 seconds)
ECOOKING (5.9) WEBCOOKING (5.9) KITCHEN HELPER (5.7) WWW FOOD (4.6) COOK ONLINE (7.3) (238 seconds)	Cook Online (6.3) Now You're Cooking (6.1) Let's Cook (5.8) Internet Cooks (5.9) Cook Like Mom (6.1) (129 seconds)	Accurate Headphone (5.2) Right Choice Headphone (7.1) Perfect Hearing (6.6) Great Sound Headphone (7) Headphony Bazaar (6.5) (300 seconds)	Hitfulphones (5.6) Giftaphone (5.6) Joyphone (5.5) Kaetkaphone (4.3) Thalaphone (5.5) (221 seconds)
iChef (6.3) eCooking Helper (7) CookinGuide (6.3) Help Me Cook (6.6) LiveCooking (6.4) (216 seconds)	Campus Cookers (6.7) Enlightened Self-Cooking (6.9) CookCare (6.7) Make Me Make Dinner (6.4) Dinner Helpers (8) (78 seconds)	Easy on the Ears (7.3) Music Candy (7) Easy Listening (7.1) Rockin the Ears (6.8) Stick It In Your Ear (5.8) (163 seconds)	Koss Corporation (4.8) Ultrasone (5.6) Skullcandy (6) Audio-Technica (6.4) Precide (5.1) (321 seconds)
Iterative Sorted	**Parallel Sorted**	**Iterative Sorted**	**Parallel Sorted**
COOK ONLINE (7.3) Cooking Online (7.2) ... your just dessert (6.1) Cook Helper (5.9) ... stick a fork in me (4.7) WWW FOOD (4.6)	Dinner Helpers (8) Live Chef (7.6) ... YourGourmet (6.2) Cook Like Mom (6.1) ... Dorm Recipes (5) Cooking Concultants (4.4)	Easy on the Ears (7.3) Easy Listening (7.1) ... Styly Headphones (6.5) Headphony Bazaar (6.5) ... Headphony (4.9) Shop Headphone (4.8)	music brain (8.3) Headphone House (7.4) ... Nice Ears (5.7) Ultrasone (5.6) ... head phones r us (4.2) different circumstances (3.7)

Figure 4-11: This figure shows all six sets of names generated in both the iterative and parallel processes for two fake company descriptions, along with various statistics including: the rating of each name, and the number of seconds a turker spent generating the names.

description: We sells computer books. Our website allows users to specify the technology goals of their project in a novel interface, and our website suggests the most appropriate books to cover all the technology involved. Our site caters to both novice and expert users alike.

Iterative Sorted	Parallel Sorted
Technical Times (8.1)	Computers & Internet Books (7.4)
Tech Read (8)	comp-e book (7.3)
Geek Books (7.7)	Technifind (7.2)
...	...
book guru (6.9)	tech-go-pro book (6.1)
Technical Tomes (6.6)	computer books (6)
the reading mouse (6.5)	Cyberspace, my cyberspace (6)
...	...
How to Work Computer (5.2)	c++ (4.1)
pages for pages (4.9)	windows (3.8)
Program Running Books (4.8)	c (2.7)

description: Our website allows users to design a custom lamp, and have it created and delivered to them. Our interface allows users to choose from a variety of lamp types and styles, combine different options, and even create new parts from scratch, for a completely unique lamp.

Iterative Sorted	Parallel Sorted
HomeStyle Lamps (7.5)	Touch of Light (7.7)
Lighting Up your Day (7.4)	Bright Eyes (7.4)
Bright Lighting (7.3)	Brights Ideas (7.4)
...	...
	design-a-lamp (6.6)
Customized Lamps Central (6.8)	Lamps R Us (6.6)
Brighten Up My Life (6.8)	HangZhou JingYing Electric Appliance Co., Ltd. (6.5)
My Light (6.7)	...
...	lamped (6)
Uniquely Yours Lamps (6.1)	Leave It to the Lava (5.5)
Lamp of the Day (5.9)	buildalamp (play on build-a-bear and lavalamp) (5.3)
Lights For u (5.3)	

description: Our websites offers a number of anonymised chat forums/support groups for people to discuss delicate issues, or get support for embarrassing or uncomfortable problems. All the chat forums are moderated, to prevent people from intruding with spam comments, or hurtful remarks (since the purpose of the forums is to provide mutual support).

Iterative Sorted	Parallel Sorted
CareChat (7.1)	A Friend In Need (7.5)
securechat (7)	trust talk (7.4)
chatcentral (6.8)	Friends Through Everything (7.3)
...	...
chatperial (6)	Chat Forum (6.2)
chatdelicate (5.8)	molehole (6)
Chat in Private (5.7)	Private Sessions (6)
...	...
mschat (5.5)	I'm Listening (5)
quitchat (5.4)	Support Tool (4.6)
impchat (5.4)	Chatting (4.5)

description: Our site helps users keep track of their calorie and nutrition intake. Our competitive advantage is that we offer a simple natural language interface for users to record what they eat. They can write things like "3 gummy bears", and our system will resolve this into nutrition information.

Iterative Sorted	Parallel Sorted
Total Intake Calculator (8.1)	CalorieRef (8.1)
Nutrition Station (8)	EZCalorieCalculator (7.9)
Complete Count (7.9)	Nutrition Tracker (7.8)
...	...
healthy diet (6.9)	Personal Plate (7.1)
Calorie Adjusters (6.8)	Nutrition Intake (7.1)
tummy info (6.7)	Calorie Looker (7)
...	...
The Eatery (5)	infonutri (5.7)
calorie (4.3)	healtylife (5.7)
care take food (4.1)	behealthy (5.6)

Figure 4-12: This figure shows the highest, middle and lowest rated names generated in both the iterative and parallel process for four fake company descriptions.

The parallel process generated the best rated name for 4 out of the 6 fake companies. However, the *average* name generated in the iterative process is rated higher (6.4 vs. 6.2). The significance of iteration becomes clear in Figure 4-13, where we show the average rating of names generated in each iteration of the iterative process. The red line indicates the average rating of names in the parallel process. The iterative process is close to this line in the first iteration, where turkers are not shown any example names. The average rating seems to steadily increase as turkers are shown more and more examples (except for iteration four, discussed next). The last iteration is statistically significantly higher than the parallel process (6.7 vs. 6.2, two-sample $T_{203} = 2.3$, $p = 0.02$).

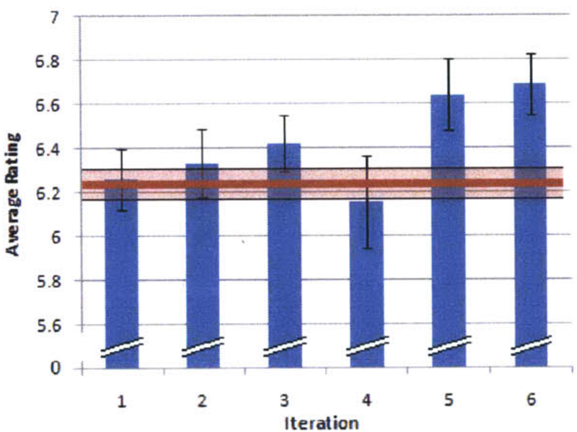


Figure 4-13: Blue bars show average ratings given to names generated in each of the six iterations of the iterative brainstorming processes. Error bars show standard error. The red stripe indicates the average rating and standard error of names generated in the parallel brainstorming processes. (See the text for a discussion of iteration 4, which appears below the red line.)

Iteration four breaks the pattern, but this appears to be a coincidence. Three of the contributions in this iteration were considerably below average. Two of these contributions were made by the same turker (for different companies). A number of their suggestions appear to have been marked down for being grammatically awkward: "How to Work Computer", and "Shop Headphone". The other turker suggested names that could be considered offensive: "the galloping coed" and "stick a fork in me".

Getting the Best Names

Although iteration seems to increase the average rating of new names, it is not clear that iteration is the right choice for generating the *best* rated names (recall that the parallel process generated the best rated name for 4 of the 6 fake companies). This may be because the iterative process has a lower variance: 0.68 compared with 0.9 for the parallel process (F-test, $F_{180,165} = 1.32$, $p = 0.036$). The higher variance of the parallel distribution means that the tail of the distribution does not shrink as quickly, and must eventually surpass the iterative process. Figure 4-14 illustrates this phenomenon, and shows the crossover point at a rating of 8.04. This suggests that the parallel process is more likely than the iterative process to generate names rated 8.04 and above, assuming that the names are normally distributed according to this model.

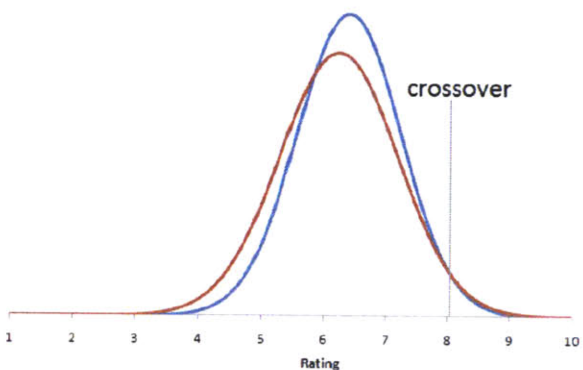


Figure 4-14: Gaussian distributions modeling the probability of generating names with various ratings in the iterative (blue) and parallel (red) processes. The mean and variance of each curve is estimated from the data. The iterative process has a higher average, but the parallel process has more variance (i.e. the curve is shorter and wider). Note that in this model, the parallel distribution has a higher probability of generating names rated over 8.04.

Note that this model may not tell the whole story. The maximum possible rating is 10, which suggests that Gaussian curves are not the best model (since they do not have any bounds). A Beta distribution may be more appropriate. This does not mean that the effect we are seeing isn't real, but it is worth further investigation.

One high level interpretation of this is that showing turkers suggestions may cause them to riff on the best ideas they see, but makes them unlikely to think too far afield from those ideas. We did see some anecdotal evidence of people's ideas being heavily influenced by suggested names. For an online chat company, the first turker suggested five names that all incorporated the word "chat". The next two turkers also supplied names that all incorporated the word "chat". The corresponding iterations in the parallel process only used the word chat three out of fifteen times. For another company, a turker used the word "tech" in all their names, and subsequent turkers used the word "tech" seven times, compared with only once in the corresponding parallel iterations.

This suggests some interesting questions to explore in future work: does showing people name suggestions inhibit their ability to generate the very best new name ideas? If so, is there anything we can do in an iterative process that *will* help generate the best names? Alternatively, is there any way we can increase the variance in the parallel process, to have an even better chance of generating the best names?

Time & Participation

Like the previous task, we did not observe a statistically significant difference in the time that turkers spent generating names in each process, see Table 4.4. We also observed similar participation behavior in this experiment, this time with about 72% of the work done by 28% of the workers. Table 4.5 shows that 142 turkers participated in this experiment, with the most prolific turker completing 165 tasks (all rating tasks).

task	geometric mean	geo std dev	N
brainstorming (iterative)	184.2	1.54	36
brainstorming (parallel)	156.4	1.67	36
rating	29.1	2.19	3450

Table 4.4: Time spent working on various tasks, in seconds. All differences are statistically significant, except between iterative and parallel brainstorming (two-sample $T_{70} = 1.47$, $p = 0.15$).

task	unique turkers	tasks done by most active turker
overall	142	165
brainstorming	41	5
brainstorming (iterative)	29	3
brainstorming (parallel)	28	4
rating	112	165

Table 4.5: Number of unique turkers participating in different task types for this experiment, along with the number of tasks completed by the most prolific turker in each category.

4.2.3 Blurry Text Recognition

The final experiment compares the iterative and parallel processes in the transcription domain. The task is essentially human OCR, inspired by reCAPTCHA [60]. We considered other puzzle possibilities, but were concerned that they might be too fun, which could have the side effects discussed in [40].

Figure 4-15 shows an example blurry text recognition task. The instructions are simply to transcribe as many words as possible, and we place a textbox beneath each word for this purpose. In the iterative condition, these textboxes contain the most recent guess for each word. We also ask workers to put a "*" in front of words that they are unsure about. This is meant as a cue to future workers that a word requires more attention. Note that this instruction appears in the parallel tasks as well, even though no workers see any other workers' stars.

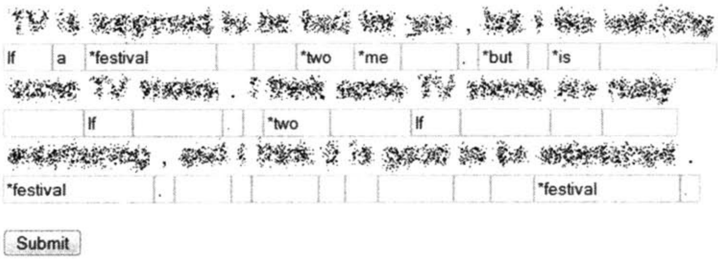


Figure 4-15: Turkers are shown a passage of blurry text with a textbox beneath each word. Turkers in the iterative condition are shown guesses made for each word from previous turkers.

We composed twelve original passages. It was important to use original text, rather than text obtained from the web, since turkers could conceivably find those passages by searching with some of the keywords they had deciphered.

We then ran each passage through an image filter. The filter works on a pixel level. Each pixel in the new image is created by randomly choosing a pixel near that location in the old image, according to a 2-dimensional Gaussian. This appears to be identical to the lossy "blur" tool in the GIMP[2]. The result is blurry text that is very difficult to decipher. Some words appear to be entirely illegible on their own. The hope is that by seeing the entire passage in context, turkers will be able to work out the words.

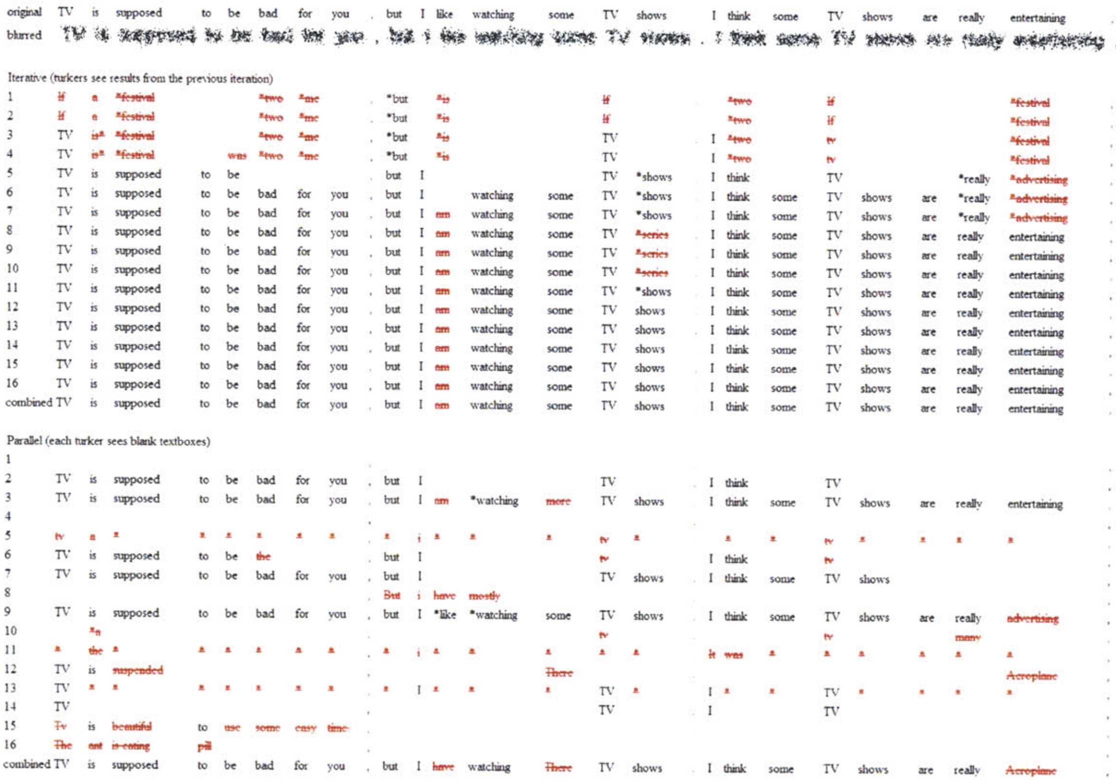
Unlike the previous experiments, we use sixteen creation tasks in both the iterative and parallel processes, each task paying 5 cents. Note that we did not use any comparison tasks, because we thought workers would primarily contribute to different parts of the transcription, and these contributions would be easy to merge automatically. We also didn't need to pay any workers to rate the results, since we could assess the accuracy of the results automatically using the ground truth text from which the blurry images were created. We will discuss these decisions more in the results section.

We applied both the iterative and parallel process to each passage. As before, each process was run first for a random half of the passages, and no turkers were allowed to participate in both processes for a single passage. The final transcription of the blurry text is derived by merging the transcriptions within a process on a word by word basis. We look at all the guesses for a single word in all sixteen transcriptions. If a particular word is guessed a plurality of times, then we choose it. Otherwise, we pick randomly from all the words that tied for the plurality. We also tried choosing the final iteration in the iterative condition, but it did not make much difference, and it seemed more fair to apply the same merging algorithm to each process. Note that other merging algorithms are possible, and we will have more to say about this below.

[2]GIMP is the GNU Image Manipulation Program: http://www.gimp.org/

Our hypothesis was that the iterative process would have a higher probability of deciphering each passage, since turkers would be able to use other people's guesses as context for their own guesses. The analogy would be solving a crossword puzzle that other people have already started working on.

Results & Discussion



original	TV	is	supposed	to	be	bad	for	you	,	but	I	like	watching	some	TV	shows	.	I	think	some	TV	shows	are	really	entertaining	.
blurred									,								.									,
Iterative (turkers see results from the previous iteration)																										
1	~~If~~	~~a~~	~~*festival~~			~~*two~~	~~*me~~		,	*but		~~*is~~			~~If~~				~~*two~~		~~If~~				~~*festival~~	.
2	~~If~~	~~a~~	~~*festival~~			~~*two~~	~~*me~~		,	*but		~~*is~~			~~If~~				~~*two~~		~~If~~				~~*festival~~	.
3	TV	~~is*~~	~~*festival~~			~~*two~~	~~*me~~		,	*but		~~*is~~			TV			I	~~*two~~		~~tv~~				~~*festival~~	.
4	TV	~~is*~~	~~*festival~~		~~was~~	~~*two~~	~~*me~~		,	*but		~~*is~~			TV			I	~~*two~~		~~tv~~				~~*festival~~	.
5	TV	is	supposed	to	be				,	but	I				TV	*shows		I	think		TV			*really	~~*advertising~~	.
6	TV	is	supposed	to	be	bad	for	you	,	but	I		watching	some	TV	*shows		I	think	some	TV	shows	are	*really	~~*advertising~~	.
7	TV	is	supposed	to	be	bad	for	you	,	but	I	~~am~~	watching	some	TV	*shows		I	think	some	TV	shows	are	*really	~~*advertising~~	.
8	TV	is	supposed	to	be	bad	for	you	,	but	I	~~am~~	watching	some	TV	~~*series~~	.	I	think	some	TV	shows	are	really	entertaining	.
9	TV	is	supposed	to	be	bad	for	you	,	but	I	~~am~~	watching	some	TV	~~*series~~	.	I	think	some	TV	shows	are	really	entertaining	.
10	TV	is	supposed	to	be	bad	for	you	,	but	I	~~am~~	watching	some	TV	~~*series~~	.	I	think	some	TV	shows	are	really	entertaining	.
11	TV	is	supposed	to	be	bad	for	you	,	but	I	~~am~~	watching	some	TV	*shows	.	I	think	some	TV	shows	are	really	entertaining	.
12	TV	is	supposed	to	be	bad	for	you	,	but	I	~~am~~	watching	some	TV	shows	.	I	think	some	TV	shows	are	really	entertaining	.
13	TV	is	supposed	to	be	bad	for	you	,	but	I	~~am~~	watching	some	TV	shows	.	I	think	some	TV	shows	are	really	entertaining	.
14	TV	is	supposed	to	be	bad	for	you	,	but	I	~~am~~	watching	some	TV	shows	.	I	think	some	TV	shows	are	really	entertaining	.
15	TV	is	supposed	to	be	bad	for	you	,	but	I	~~am~~	watching	some	TV	shows	.	I	think	some	TV	shows	are	really	entertaining	.
16	TV	is	supposed	to	be	bad	for	you	,	but	I	~~am~~	watching	some	TV	shows	.	I	think	some	TV	shows	are	really	entertaining	.
combined	TV	is	supposed	to	be	bad	for	you	,	but	I	~~am~~	watching	some	TV	shows	.	I	think	some	TV	shows	are	really	entertaining	.
Parallel (each turker sees blank textboxes)																										
1																										
2	TV	is	supposed	to	be	bad	for	you	,	but	I				TV		.	I	think		TV					.
3	TV	is	supposed	to	be	bad	for	you	,	but	I	~~am~~	*watching	~~more~~	TV	shows	.	I	think	some	TV	shows	are	really	entertaining	.
4									,																	.
5	~~tv~~	~~a~~	~~*~~	~~*~~	~~*~~	~~*~~	~~*~~	~~*~~	,	~~*~~	~~i~~	~~*~~	~~*~~	~~*~~	~~tv~~	~~*~~	.		~~*~~	~~*~~	~~tv~~	~~*~~	~~*~~	~~*~~	~~*~~	.
6	TV	is	supposed	to	be	~~the~~			,	but	I				~~tv~~			I	think		~~tv~~					.
7	TV	is	supposed	to	be	bad	for	you	,	but	I				TV	shows	.	I	think	some	TV	shows				.
8									,	~~But~~	~~i~~	~~have~~	~~mostly~~													.
9	TV	is	supposed	to	be	bad	for	you	,	but	I	*like	*watching	some	TV	shows	.	I	think	some	TV	shows	are	really	~~advertising~~	.
10		~~*a~~							,						~~tv~~						~~tv~~			~~many~~		.
11	~~*~~	~~the~~	~~*~~	~~*~~	~~*~~	~~*~~	~~*~~	~~*~~	,	~~*~~	~~i~~	~~*~~	~~*~~	~~*~~	~~*~~	~~*~~		~~It~~	~~was~~	~~*~~	~~*~~	~~*~~	~~*~~	~~*~~	~~*~~	.
12	TV	is	~~suspended~~						,					~~There~~											~~Aeroplane~~	.
13	TV	~~*~~	~~*~~	~~*~~	~~*~~	~~*~~	~~*~~	~~*~~	,	~~*~~	I	~~*~~	~~*~~	~~*~~	TV	~~*~~		I	~~*~~	~~*~~	TV	~~*~~	~~*~~	~~*~~	~~*~~	.
14	TV								,						TV			I			TV					.
15	~~Tv~~	is	~~beautiful~~	to	~~use~~	~~some~~	~~easy~~	~~time~~	,																	.
16	~~The~~	~~ant~~	~~is eating~~	~~pill~~					,																	.
combined	TV	is	supposed	to	be	bad	for	you	,	but	I	~~have~~	watching	~~There~~	TV	shows	.	I	think	some	TV	shows	are	really	~~Aeroplane~~	.

Figure 4-16: This figure shows all sixteen transcriptions generated in both the iterative and parallel process for the first part of a passage. The original passage is shown at the top, and incorrect guesses are struck out in red. The "combined" row shows the result of applying the merging algorithm. Note the word "Aeroplane" in the very last row. One of the turkers in this parallel process chose the correct word "entertaining", but the merging algorithm chose randomly from among the choices "Aeroplane", "entertaining" and "advertising" in this column.

Figures 4-18, 4-18 and 4-18 show a sample of raw results from the experiment. Figures 4-18 and 4-18 show all sixteen iterations of both the iterative and parallel processes for two passages, including the passage from Figure 4-15, though some

original	Please	do	not	touch	anything	in	this	house	.	Everything	is	very	old	,	and	very	expensive	,
blurred									.					,				,
Iterative (turkers see results from the previous iteration)																		
1									.					,				,
2						~~to~~			.		is			,				,
3	Please	~~*~~	~~*~~	~~*~~	anything	~~to~~	~~*~~	~~*~~	.	Everything	is	~~*~~	~~*~~	,	~~*~~	~~*~~	~~*~~	,
4	Please	~~*~~	~~ask~~	~~*~~	anything	~~to~~	~~*~~	~~*~~	.	Everything	is	~~*~~	~~*~~	,	~~*~~	~~*~~	~~*~~	,
5	Please	~~*~~	~~ask~~	~~*~~	anything	~~to~~	~~*~~	~~*~~	.	Everything	is	~~*~~	~~*~~	,	~~*~~	~~*~~	~~*~~	,
6	Please	~~*~~	~~ask~~	~~*~~	anything	~~to~~	~~*~~	~~*~~	.	Everything	is	~~*~~	~~*~~	,	~~*~~	~~*~~	~~*~~	,
7	Please	~~*~~	~~ask~~	~~*~~	anything	~~to~~	~~*for~~	~~*me~~	.	Everything	is	~~*~~	~~*~~	,	~~*~~	~~*~~	~~*~~	,
8	Please	do	~~ask~~	~~*~~	anything	~~you~~	~~need~~	~~*me~~	.	Everything	is	~~going~~	~~fine~~	,	~~there~~	~~*~~	~~*~~	,
9	Please	do	~~ask~~	~~*~~	anything	~~you~~	~~need~~	~~*me~~	.	Everything	is	~~going~~	~~fine~~	,	~~there~~	~~*~~	~~*~~	,
10	Please	do	~~ask~~	~~*~~	anything	~~you~~	~~need~~	~~*me~~	.	Everything	is	~~going~~	~~fine~~	,	~~there~~	~~*~~	~~*~~	,
11	Please	do	~~ask~~	~~*about~~	anything	~~you~~	~~need~~	~~*me~~	.	Everything	is	~~going~~	~~fine~~	,	~~there~~	~~*was~~	~~*~~	,
12	Please	do	~~ask~~	~~*about~~	anything	~~you~~	~~need~~	~~*me~~	.	Everything	is	~~going~~	~~fine~~	,	~~there~~	~~*was~~	~~*~~	,
13	Please	do	~~ask~~	~~*about~~	anything	~~you~~	~~need~~	~~*me~~	.	Everything	is	~~going~~	~~fine~~	,	~~there~~	~~*were~~	~~*~~	,
14	Please	do	~~ask~~	~~*about~~	anything	~~you~~	~~need~~	~~*me~~	.	Everything	is	~~going~~	~~fine~~	,	~~there~~	~~*were~~	~~*~~	,
15	Please	do	~~ask~~	~~*about~~	anything	~~you~~	~~need~~	~~*me~~	.	Everything	is	~~going~~	~~fine~~	,	~~there~~	~~*were~~	~~*~~	,
16	Please	do	~~ask~~	~~*about~~	anything	~~you~~	~~need~~	~~*me~~	.	Everything	is	~~going~~	~~fine~~	,	~~there~~	~~*were~~	~~*~~	,
combined	Please	do	~~ask~~	~~about~~	anything	~~you~~	~~need~~	~~me~~	.	Everything	is	~~going~~	~~fine~~	,	~~there~~	~~were~~		,
Parallel (each turker sees blank textboxes)																		
1	*Please	do	not	touch	anything	in	~~*the~~	house	.	Everything	is	~~nery~~	~~*nice~~	,	and	very	expensive	,
2					anything	in	~~the~~	*house	.	Everything	is			,	and	*very		,
3	Please	do	not	touch	anything	in	this		.	Everything	is	very	old	,	and	very		,
4		~~was~~							.		~~an~~			,				,
5	~~P~~		~~*fish~~		*anything	*in	*this		.	Everything	is	~~**y~~		,		*very	~~*impressive~~	,
6	Please	do	not	touch	anything	in	~~the~~		.	Everything	is	very		,				,
7	Please	do	not	touch	anything	in	this	house	.	Everything	is	very	old	,	and	very	expensive	,
8	Please	do	not	touch	anything	in	~~the~~	house	.	Everything	is	very	old	,	and	very	~~ewpensive~~	,
9						in	~~the~~		.	Everything	is	~~*only~~		,				,
10	~~*~~	~~*to~~	~~the~~	~~*~~	anything	~~is~~	~~*~~	~~*needed~~	.	~~*Transponding~~	~~to~~	~~many~~	~~*~~	,	~~*~~	~~*~~	~~*~~	,
11	~~*~~	~~*~~	~~*~~	~~*~~	~~*~~	~~*~~	~~*~~	~~*~~	.	~~*~~	~~*~~	~~*~~	~~*~~	,	~~*~~	~~*~~	~~*~~	,
12	*Please	do	not	~~write~~	anything	~~*at~~	this	~~*point~~	.	Everything	is			,		*very		,
13	Please	do	not	touch	anything	in	~~the~~	house	.	Everything	is	very	old	,	and	very	expensive	,
14	~~*~~	~~*~~	~~*~~	~~*~~	anything	~~at~~	this	~~*~~	.	~~everything~~	~~*~~	~~*~~	~~*~~	,	~~*~~	~~*~~	~~*~~	,
15		~~is~~	not		anything	in	this		.	~~Everyday~~	is			,				,
16	~~They~~	~~he~~	~~the~~						.			~~They~~	~~also~~	,				,
combined	Please	do	not	touch	anything	in	this	house	.	Everything	is	very	old	,	and	very	expensive	,

Figure 4-17: This figure shows all sixteen transcriptions generated in both the iterative and parallel process for the first part of a passage. The original passage is shown at the top, and incorrect guesses are struck out in red. The "combined" row shows the result of applying the merging algorithm. Note that turkers seem reluctant to change guesses in the iterative case, causing it to hit a local maximum, while the parallel algorithm does very well.

[illegible]	[illegible]
Iterative: TV is supposed to be bad for you , but I ~~am~~ like watching some TV shows . I think some TV shows are really entertaining , and I think it is good to be entertained . **Parallel:** TV is supposed to be bad for you , but I like watching ~~more~~ some TV shows . I think some TV shows are really ~~Aeroplane~~ entertaining , and I think it is good to be entertained .	**Iterative:** Please do ~~ask about~~ not touch anything ~~you need me~~ in this house . Everything is ~~going fine~~ very old , ~~there were~~ and very expensive , ~~show me~~ and you ~~then bring you~~ will probably break anything you ~~desire~~ touch . **Parallel:** Please do not touch anything in this house . Everything is very old , and very expensive , and you will probably break anything you touch .
[illegible]	[illegible]
Iterative: Killer whales are beautiful animals . I remember seeing these huge , ~~beautiful~~ smooth , black and white creatures jumping high into the air at Sea World , as a kid . **Parallel:** ~~There Have~~ Killer whales are beautiful ~~dont~~ animals . ~~I sometimes~~ I remember seeing ~~They them~~ these huge , ~~solvent~~ smooth , black and white creatures ~~are bottle structure~~ jumping ~~They have was is the~~ high into the air at Sea World , ~~a*~~ as a kid .	**Iterative:** ~~Lion is strongest wild animal.~~ Hot air balloons are beautiful . It is always a ~~king~~ treat ~~of and i like it. butterflies~~ to see a hot air balloon flying in the distance . ~~Understandably this~~ Unfortunately I've never ~~good~~ been in one . **Parallel:** ~~The way domastic~~ Hot air balloons are ~~complete~~ beautiful . ~~I~~ It is always a treat to see a ~~best of~~ hot air balloon flying in the distance . Unfortunately ~~this They~~ I've never been in one .

Figure 4-18: This figure shows the results for the complete passage from the previous two figures, as well as a couple other passages. These examples show cases where both iteration and the parallel algorithm do well, as well as instances where one or both algorithms do poorly.

of the passage is cropped off for space. Figure 4-18 zooms out a bit and shows the resulting transcriptions for both of these passages, as well as two other passages. These examples show the range of results, where sometimes iteration does better than the parallel process, sometimes the parallel process does better, and sometimes neither process does very well.

These results show cases where each algorithm does better, i.e., guesses a higher percentage of the words correctly. When we average over all 12 passages, we fail to see a statistically significant difference between each process (iterative 65% vs. parallel 62%, paired t-test $T_{11} = 0.4$, $p = 0.73$).

If we look at the accuracy of each process after n iterations in Figure 4-19, we see that both processes gain accuracy over the first eight iterations, and then seem to level off. Note that the iterative process appears to be above the parallel process pretty consistently after the fourth iteration. The difference is greatest after eight iterations, but is never statistically significant.

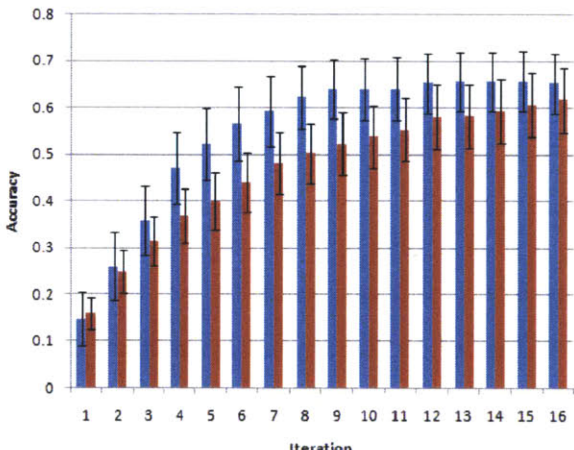


Figure 4-19: Blue bars show the accuracy after n iterations of the iterative text recognition process. Red bars show accuracy for the parallel process with n submissions. Error bars show standard error. The overlap suggests that the processes may not be different, or that we do not have enough power in our experiment to see a statistically significant difference.

The results suggest that iteration may be helpful for this task. However, it is also worth noting that iteration sometimes appears to get stuck due to poor guesses early

in the process. For instance, the iterative process in Figure XXX1 ended up with 30% accuracy after sixteen iterations. When looking at the progression of transcriptions, we see that turkers are reluctant to change existing guesses. By iteration 11, the process has more-or-less hit a local maximum:

> **16th iteration:** "Please do ~~ask *about~~ anything ~~you need *me~~. Everything is ~~going fine, there *were~~ * , show me then ~~*bring~~ * anything you ~~desire~~."

These guesses all have a gist of "ask for anything you need, and I'll give it to you." The fact that they make a sort of sense may have made the guesses all the more difficult to change. Note that multiple turkers deciphered this entire passage almost perfectly in the parallel process, suggesting that progress was hampered by poor guesses rather than by unreadable text. In fact, one turker left a comment alluding to this possibility in one of the iterative passages: "It's distracting with the words filled in—it's hard to see anything else but the words already there, so I don't feel I can really give a good "guess" ".

Merging Algorithm

Although iteration appears to be marginally better, some of the benefit may have to do with the algorithm we use to merge all the guesses into a final answer. In the current merging process, we treat each transcription for a particular word as a vote for that transcription, picking the transcription with the plurality of votes. However, if we rigged the system so that if anyone guessed the correct transcription for a word, we would pick it, then the parallel process would do better: 76% versus 71% for the iterative process. This difference is not significant ($p = 0.64$), and only serves to cast greater doubt on the benefit of iteration for this task.

This suggests that any benefit we see from iteration may have to do with the fact that it has an implicit mechanism for multiple turkers to vote on a single turker's guess. If a turker leaves a guess alone, then they are implicitly voting for it. In the

parallel process, a word can only get multiple votes if multiple turkers arrive at the same guess independently.

One way the parallel process could be improved would be using other information to help pick out the best guesses. For instance, a word may be more likely to be correct if it came from a turker who made guesses for surrounding words as well, since the word may be satisfying more constraints.

It may be possible to improve the iterative process as well by randomly hiding guesses for certain words, in order to solicit more options for those words. Also, we could show people multiple options for a word, rather than just the most recent guess.

Although it is interesting to think of ways to improve the algorithm for this process, our real hope was that this task would be so difficult that a single turker could not accomplish it on their own. In future work, it would be nice to explore a task that had a higher level of difficulty, or impose a time limit to simulate higher difficulty.

Time & Participation

Yet again, we did not observe a statistically significant difference in the time that turkers spent transcribing text in each process, see Table 4.6. Note that [38] reports a significant difference between these conditions, but that comparison does not take into account that the data is log-normally distributed.

We also observed similar participation behavior in this experiment to the previous experiments, this time with about 69% of the work done by 31% of the workers. Note that the power law distribution was capped at 12, meaning that no worker could do more than 12 tasks. Six workers did 12 tasks, but no worker did all 12 in the same condition. We had 94 turkers participate overall, see Table 4.7.

task	geometric mean	geo std dev	N
transcription (iterative)	96.0	2.31	192
transcription (parallel)	107.4	2.46	192

Table 4.6: Time spent working on the iterative and parallel transcription tasks, in seconds. The difference is not significant (two-sample $T_{382} = 1.26$, $p = 0.21$).

task	unique turkers	tasks done by most active turker
overall	94	12
transcription (iterative)	65	9
transcription (parallel)	74	8

Table 4.7: Number of unique turkers participating in different task types for this experiment, along with the number of tasks completed by the most prolific turker in each category.

4.3 Discussion

All of these experiments demonstrate success in performing high-level creative and problem solving tasks, using processes that orchestrate the efforts of multiple workers. We also see that the breakdown into creation and decision tasks is applicable to a diverse set of problem domains.

4.3.1 Tradeoff between Average and Best

In the brainstorming task, we saw a tradeoff between increasing average response quality, and increasing the probability of the best responses. Showing prior work increased the average quality of responses, but reduced the variance enough that the highest quality responses were still more likely to come from turkers not shown any prior suggestions.

There is a sense in which this tradeoff exists in the other two tasks as well. The writing task generates descriptions with a higher average rating, but lower variance, when turkers are shown prior work. Also, the transcription task increases the average frequency of correct guesses for each word when turkers are shown a prior guess, but it decreases the variety of guesses for each word.

However, an alternate explanation for the reduced variance is simply that there is a maximum quality, so pushing the average toward this barrier must eventually reduce the variance (e.g., if the average rating reaches 10, then the variance must be 0). The real question is whether the reduction in variance is enough to exhibit the tradeoff we seem to observe in the brainstorming case. We will need to employ more

robust mathematical models to make more progress on this front, since Gaussian distributions are limited in how well they can model a bounded random variable.

Investigating this tradeoff further may be worthwhile, because if it exists, then it implies two different alternatives for achieving quality responses, depending on our target quality. If our target quality is not too high, then we can increase the average, whereas if it is very high, then we may do better to increase the variance and find some method of detecting the high quality responses when they arrive.

Note that we may be able to proactively increase the variance in the brainstorming task, perhaps by showing people completely random words (which may have nothing to do with the topic), in order to get them thinking outside the proverbial box.

4.3.2 Not Leading Turkers Astray

In our experiments, showing prior work can have a negative effect on quality by leading future workers down the wrong path. This effect is most pronounced in the blurry text recognition task, and may be an issue in other tasks of this form where puzzle elements build on each other (like words in a crossword puzzle). Turkers take suggestions from previous turkers, and try to make the best guesses they can, but backtracking seems more rare.

This suggests that a hybrid approach may be better, where multiple iterative processes are executed in parallel to start with, and then further iteration is performed on the best branch.

4.4 Conclusion

This chapter compares iterative and parallel human computation processes. In the iterative algorithm, each worker sees the results from the previous worker. In the parallel process, workers work alone. We apply each algorithm to a variety of problem domains, including writing, brainstorming, and transcription. We use MTurk and TurKit to run several instances of each process in each domain. We discover that iteration increases the average quality of responses in the writing and brainstorming

domains, but that the best results in the brainstorming and transcription domains may come from the parallel process, because it yields a greater variety of responses. We also see that providing guesses for words in the transcription domain can lead workers down the wrong path.

A couple of recent studies by Dow et al. [21] [20] compare iterative and parallel prototyping processes and find strong evidence to support the parallel approach, casting further doubt on iteration. One important difference in their experimental setup is that they study individuals iterating on their own work given feedback, whereas our experiments involve people iterating on other people's work. Also, their parallel condition involves *some* iteration, which we might simulate by having a parallel process feed into an iterative process. Still, the experiments have a lot in common, and our own iterative process exhibits some of the same problems cited by Dow, including a lower variety of responses. Hence, there may be much to learn by drawing connections between these experiments in future work. In particular, it is worth exploring processes that combine aspects of the iterative and parallel processes together, as well as processes that allow individual workers to participate multiple times.

Chapter 5

Case Study

In this chapter we present a case study of a relatively complicated human computation process for performing human OCR on hand-completed forms, and discuss design issues we encountered while building and evaluating the system. This process was designed, built and tested while at an internship with Xerox.

The design challenges in this space include issues like: What sorts of tasks are humans willing to do at affordable prices, given relatively common skills? How do we decompose a given problem into such tasks? How do we communicate our tasks as efficiently as possible (since we are essentially paying workers to read our instructions as well as do our tasks)? How do we validate the work that people do? We tackle each of these issues in the design of our human OCR system.

This chapter will describe our problem, our solution, and an experiment we ran on our solution. Then we spend the rest of the chapter discussing the results, including sharing some high level insights that came from this work.

5.1 Problem

We want to perform OCR on handwritten forms. In particular, we want to take a set of scanned forms, and generate a spreadsheet of the data on the forms. The output spreadsheet should have a row for each scanned form, and a column for each field of

data (e.g. a "first name" column, and a "zip code" column, if there are fields for first name and zip code).

Unfortunately, computers are not yet sufficiently good at recognizing handwriting, which underlies part of the reason CAPTCHAs [3] are so effective, and why reCAPTCHA [60] is being used to help transcribe books. Humans are much better at recognizing handwriting, so we want to use human workers to help. However, our particular scenario is transcribing medical forms, which adds a privacy concern: we do not want to leak sensitive information from the forms to the workers helping perform the OCR.

5.2 Solution

Our basic approach involves showing human workers small cropped windows of each form, each containing a small chunk of information that should be useless on its own (e.g. the word "HIV", without seeing who has it). Since our approach involves showing people cropped views of a form, we will also need some way to discover the location and size of each view. To support this task, we add an additional input to the system, which is a blank template form. Workers may view the entire template form without revealing anyone's private information. The template form is used in the first phase of a three phase process, described next.

5.2.1 Phase 1: Drawing Rectangles

The first phase has people draw rectangles indicating fields on the blank template form. Workers are instructed to "click-and-drag to draw rectangles where writing would go", and they are shown some positive and negative example rectangles over an example form, as shown in Figure 5-2.

We only keep rectangles that multiple people draw in roughly the same place. To determine whether two rectangles are "roughly the same", we see whether each border on the first rectangle is within some tolerance of the corresponding border on the second rectangle. The horizontal tolerance is equal to 10% of the average width

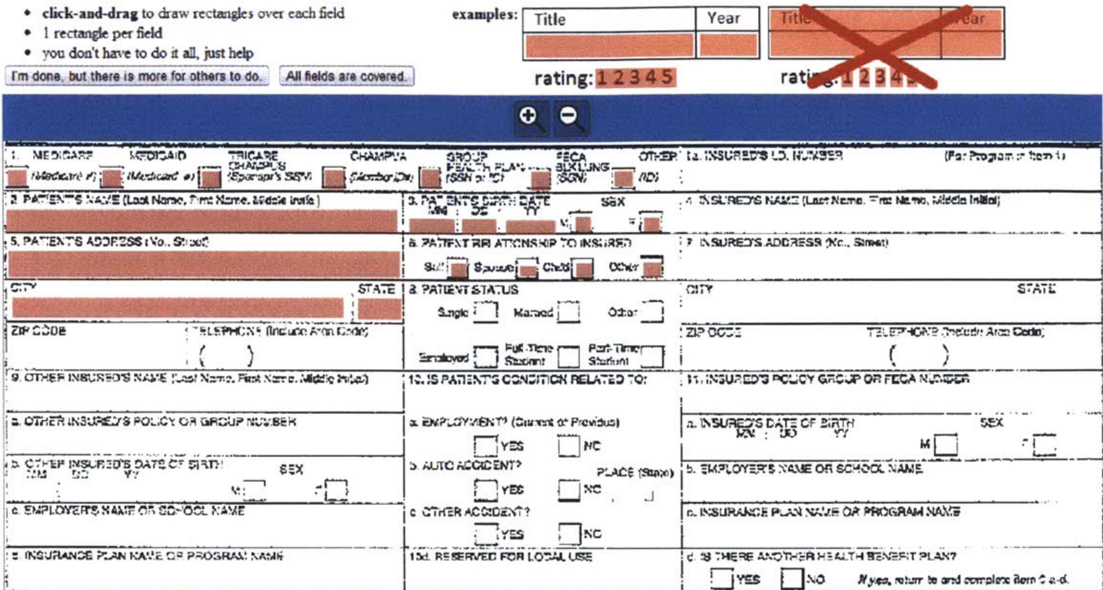


Figure 5-1: This is a phase 1 HIT where a turker is instructed to draw rectangles over a form marking areas where handwritten information would appear. Four turkers have already completed a similar HIT, and the fifth turker sees the rectangles that the previous workers have agreed on.

of the two rectangles. Similarly, the vertical tolerance is 10% of the average height. If either tolerance is less than 10 pixels, then we make it 10 pixels.

Because the form may have many fields, we tell workers "you don't have to do it all, just help", and we show workers all the rectangles that previous workers have agreed upon. These rectangles act as extra examples of what to do, and also encourage workers to draw rectangles elsewhere on the form, so that everyone doesn't draw rectangles for the top few fields. In case all the rectangles are already drawn, we include a submit button labeled "All the fields are covered".

We pay 5 cents for each worker, and keep asking workers to add more rectangles for a set number of iterations (10 in our case). One could also imagine a stopping condition based on the "All the fields are covered" button.

5.2.2 Phase 2: Labeling Rectangles

The first phase provides a set of rectangles covering the form. These rectangles mark all of the chunks of information that will appear in the columns of the final spreadsheet. What we need next is a heading for each column, i.e., a label for each rectangle. The interface for this task is fairly straightforward: we show a worker the form with a single rectangle highlighted on it. Then we ask for a label of the given rectangle. A sample HIT is shown in Figure 5-2.

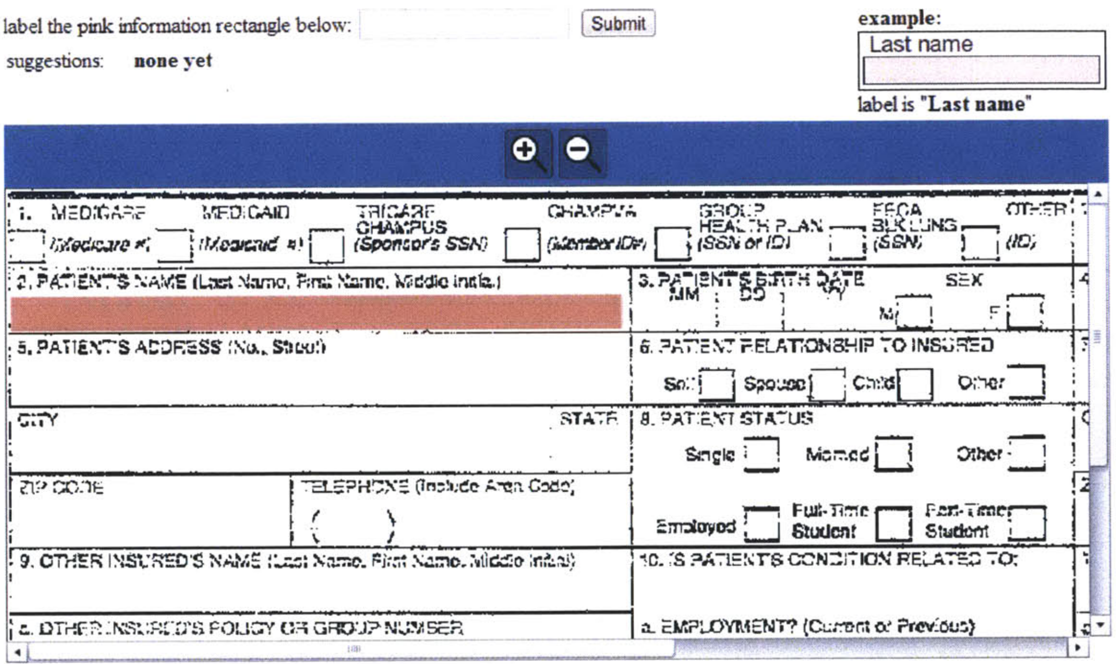


Figure 5-2: This is a phase 2 HIT where a turker is asked to label the highlighted rectangle. Subsequent turkers will be shown this worker's label as a suggestion.

We iteratively ask for labels (1 cent each) until two workers submit exactly the same label. We were concerned that this might take too long, since the space of possible labels might be quite large given different abbreviations for a long label, and different ways to capitalize it. To foster convergence, we show each worker all the labels that previous workers have provided, so that they can simply copy one of these if it seems good enough.

5.2.3 Phase 3: Human OCR

The final phase of the process is the most straightforward. For each rectangle, on each scanned form, we show workers a cropped view of that form where the rectangle is, and prompt them to write what is written there. This interface is shown in Figure 5-3.

Figure 5-3: This is a phase 3 HIT where a turker is asked to transcribe the given form field. They are also shown the label of the field from the previous phase, in case this helps give context to the transcription.

We also show people the label for the rectangle (e.g. "Zip Code"), so that they have some context to help them decipher the handwriting (e.g. these must be five numbers). We also expand the rectangle a little bit, in case people write outside the lines. Note that the instruction: "just write 'check' for check marks" was added based on pilot experiments were workers were unsure what to write when they saw a check mark.

For this task, we do not show workers the results from previous workers, since we want *independent* agreement about the correct answer. Hence, we keep asking for more submissions (1 cent each) until two workers submit exactly the same response.

5.3 Experiment

We iteratively developed the solution above by testing each phase on a set of artificially created data, including real handwriting that we procured by hiring MTurk workers to print a fake form, fill it out with fake information, and scan it (50 cents each).

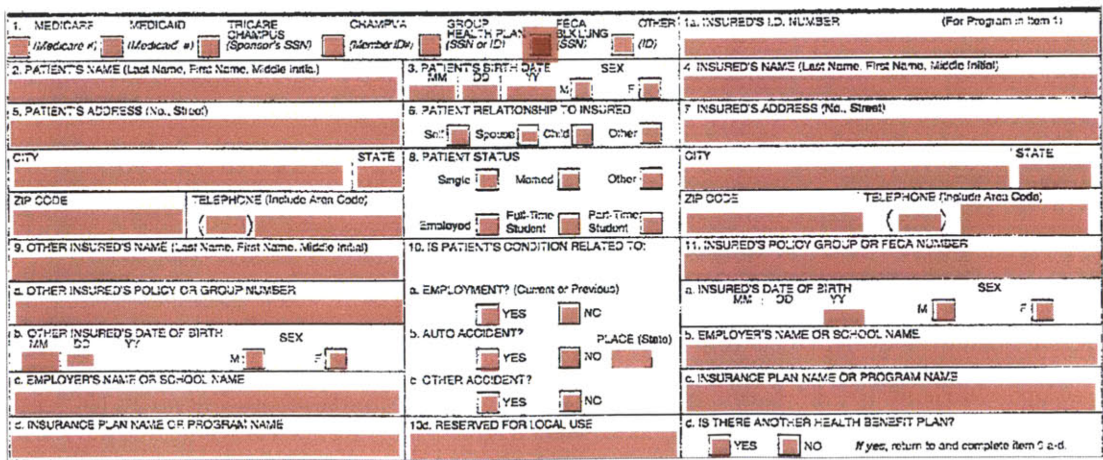
We tested the final process on a set of real forms with real data. We started with 30 scanned medical insurance claim forms, along with a blank template form. First, we removed all identifying information, like names and telephone numbers (we were not yet ready to test the privacy aspect of the system). Then, we cropped the forms, to remove form elements that our system was not prepared to deal with. For instance, the form had a list where users could enter one or more items, and this did not fit our key-value spreadsheet model for the data. (This form element would be like a cell in a spreadsheet containing a nested spreadsheet, which we will leave to future work.) Finally, we manually aligned all the scanned forms with the template form. Aligning the forms was necessary so that the rectangle coordinates found in Phase 1 would be applicable in Phase 3. We imagine that the alignment process could be automated with image processing.

5.4 Results and Discussion

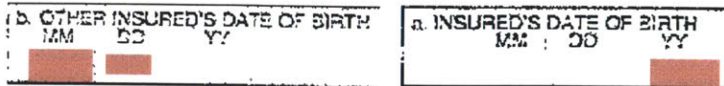
It is easiest to gauge the performance of the process by looking at the results for each phase in turn.

5.4.1 Phase 1: Drawing Rectangles

Workers agreed on rectangles for nearly every form element that we were interested in (61/64, or 95%), and we had one duplicate rectangle:


1. MEDICARE (Medicare #) MEDICAID (Medicaid #) TRICARE CHAMPUS (Sponsor's SSN) CHAMPVA (Member ID#) GROUP HEALTH PLAN (SSN or ID) FECA BLK LUNG (SSN) OTHER (ID)
1a. INSURED'S I.D. NUMBER (For Program in Item 1)
2. PATIENT'S NAME (Last Name, First Name, Middle Initial)
3. PATIENT'S BIRTH DATE MM DD YY SEX M F
4. INSURED'S NAME (Last Name, First Name, Middle Initial)
5. PATIENT'S ADDRESS (No., Street)
6. PATIENT RELATIONSHIP TO INSURED Self Spouse Child Other
7. INSURED'S ADDRESS (No., Street)
CITY STATE
8. PATIENT STATUS Single Married Other
CITY STATE
ZIP CODE TELEPHONE (Include Area Code) ( )
Employed Full-Time Student Part-Time Student
ZIP CODE TELEPHONE (Include Area Code) ( )
9. OTHER INSURED'S NAME (Last Name, First Name, Middle Initial)
10. IS PATIENT'S CONDITION RELATED TO:
11. INSURED'S POLICY GROUP OR FECA NUMBER
a. OTHER INSURED'S POLICY OR GROUP NUMBER
a. EMPLOYMENT? (Current or Previous) YES NO
a. INSURED'S DATE OF BIRTH MM DD YY SEX M F
b. OTHER INSURED'S DATE OF BIRTH MM DD YY SEX M F
b. AUTO ACCIDENT? YES NO PLACE (State)
b. EMPLOYER'S NAME OR SCHOOL NAME
c. EMPLOYER'S NAME OR SCHOOL NAME
c. OTHER ACCIDENT? YES NO
c. INSURANCE PLAN NAME OR PROGRAM NAME
d. INSURANCE PLAN NAME OR PROGRAM NAME
10d. RESERVED FOR LOCAL USE
d. IS THERE ANOTHER HEALTH BENEFIT PLAN? YES NO If yes, return to and complete item 9 a-d.

The missing form elements come from these two fields:



We did not anticipate the wide horizontal variance that workers submitted for these form elements, which threw off our fuzzy rectangle matching algorithm, which was based on a fixed tolerance of 10% of the rectangle's width. This tolerance also allowed for a duplicate rectangle (note the dark red inner rectangle):

The extra rectangle actually exhibits two problems. First, we hoped that the algorithm would consider these rectangles to be the same, since they indicate the same form element. Second, the larger rectangle is actually the result of a single worker drawing two larger rectangles, and the algorithm counting this as inter-worker agreement, even though it was really a worker agreeing with themselves.

5.4.2 Phase 2: Labeling Rectangles

Most of the labels acquired in Phase 2 were correct, and unambiguous (41/62 or 66%), though they were not always consistent. For instance, the following two labels are for the "yes" and "no" checkboxes of the same question, but one puts the word "yes" at the beginning, and the other puts "no" at the end:

- Yes (Employment current employer or previous)
- employment(current or previous)-no

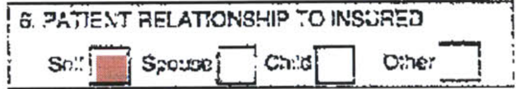
The next 31% (19/62) were ambiguous, meaning that the same label could apply equally well to more than one form element. For instance, "YES", since four checkboxes had this label. Note that our instructions simply asked workers to label the

rectangle, so this is not a case of cheating, but rather underspecified instructions. It is reassuring, however, than many workers provided unique labels anyway.

The final 3 labels were wrong. Two were instances of a checkbox in the middle of a long sequence of checkbox-label-checkbox-label:

In this case, workers sometimes gave the left label when they should have given the right label. It is possible to determine which label is correct only by looking at the checkboxes on the extreme ends. Note that this is a symptom of poor form design, suggesting that our process may used to detect usability errors of this sort.

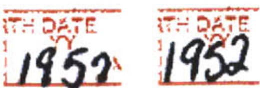
The final mistake came labeling the rectangle below:



We can infer from context that the label is a type of biological relationship, and the first letter is "s", but the scan quality is poor. The incorrect winning label was "SON", whereas one worker suggested the better label "Patient Relationship to Insured — Self".

5.4.3 Phase 3: Human OCR

Overall the OCR process seems to work extremely well, although we don't have the ground truth for these forms, so all we really know is that MTurk workers agree with the author's own guesses about people's handwriting. We do know that workers made at least one mistake:

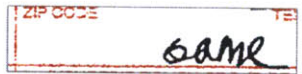


We know these dates indicate the birth-year of a single individual, so they should be the same, however, workers converged on "1957" for the first date, and "1952" for

the second. However, this seems understandable, since the "2" in the first date has many qualities of a "7".

Note that we included a special instruction to deal with checkboxes: "just write 'check' for check marks." Out of 261 instances of a check in a box, workers converged on the guess "check" 229 times (88%). Only four times did workers converge on "x" as the answer, where the other 28 times people agreed on a value like "M", which was the label of the checkbox.

We encountered another interesting problem on forms: people using words like "same" to refer to information on other parts of the form:



This may be difficult to overcome while preserving privacy, but maybe not. If people can see a blank form, they may be able to provide a useful guess about which other form element the "same" is likely to refer to and recommend that another worker look at both fields together.

5.5 Discussion

This work generated insights that may be applicable in broader contexts. We discuss each in turn.

5.5.1 Independent Agreement

Bernstein et al. [5] cite independent agreement—multiple workers generating the same response, independently—as a good sign that a response is correct. Our results corroborate this insight. For instance, in Phase 1, we only keep rectangles that multiple workers draw in the same place, and these tend to be correct. In fact, in one pilot experiment, we had multiple workers draw all the rectangles for a form, without seeing any other worker's submissions, and while no single worker got them all correct, the set of independently agreed upon rectangles was correct.

5.5.2 Dynamic Partitioning

Drawing many rectangles on a form presents an interesting challenge: how do we partition the task so that no worker needs to do all the work? The problem is that the form is difficult to break into pieces that can be processed independently. For instance, showing people a cropped window of the form might result in fields straddling the edge of that window. Our solution is dynamic partitioning, where we show each worker what has been done so far, so that they can decide for themselves where to make the next contribution. This approach may generalize to other problems which are difficult to break into clearly defined pieces. Note that this approach is parallel in the sense that each part of the problem is processed by a subset of the workers. However, it is not parallel in the sense of having workers work simultaneously, since we must wait for a worker to complete their work before showing their results to the next worker.

5.5.3 Risk of Piecework

When we divide a task into small pieces, there is a risk that workers may make mistakes which could be avoided with global knowledge about other pieces. For instance, in Phase 2, workers labeling a rectangle do not know what labels are being given to other rectangles, so they have no way to establish a consistent naming convention. (Contrast this with Phase 1, where rectangles from previous workers act as examples to subsequent workers). Also, in Phase 3, the example of the word "same" appearing in a box shows how obscuring the rest of the form prevents the worker from locating the intended value for the given field.

5.5.4 Ambiguous Instructions

When humans encounter tasks for which the instructions are poorly specified or ambiguous, they often try to make a good faith effort to solve the problem. For instance, in Phase 2, labelers would often write things like "Other Accident? — yes", instead of just "Yes", even though the instructions did not say that the label needed to be

unique. Also, in Phase 3, some check marks were OCRed as the label next to the checkbox, e.g., "M", which would actually be necessary if we had a single column for "SEX" (rather than our separate column for "M" and "F"). Of course, in this case, they were disobeying our instruction to write "check" when they see a check. However, it is interesting that these workers would have done something reasonable even without that instruction. These findings suggest that it may be possible to write better, more succinct instructions, if we have a better understanding of how humans will naturally handle underspecified instructions.

5.6 Conclusion

This chapter presents a system for performing human OCR on a set of hand-completed forms, without revealing too much information from the forms to the human workers. We decompose the problem into three phases: the first phase identifies information regions on the form, the second phase provides labels for each region, and the final phase has humans transcribe what is written in each region. An evaluation of the system affirms the effectiveness of independent agreement and dynamic partitioning, and reveals risks associated with hiding global information from workers, as well as notes the ability of humans to handle edge case exceptions given ambiguous instructions. There are many directions for future work within the area of processing handwritten forms, though this thesis is more concerned with human computation algorithms in general. This case study has hopefully shed insights that are applicable to other researchers and practitioners building larger human computation systems.

Chapter 6

Discussion and Future Work

This chapter takes a step back and reflects on the field of human computation today, and where it is going. We will begin by looking at current bottlenecks, and suggesting research questions to help progress the field. We will then discuss the broader impact of human computation on society and the world, and end with some guesses of what the future will look like.

6.1 State-of-the-Art

This section discusses current issues faced by human computation researchers. These issues may be viewed as research problems that need to be solved before we can make more progress in the field.

6.1.1 Quality Control

Our goal is to build very complicated algorithms, and it is hard to build them on a shaky foundation. Bernstein et al. [5] note that about 30% of their results for open-ended tasks on MTurk are poor. I have heard this number anecdotally from other sources as well. These numbers are shaky enough to merit a firm layer of abstraction.

Current Methods

Researchers and practitioners have already explored many ways of dealing with quality control, many of which we covered earlier in this thesis. Kittur [30] suggest making cheating on a task as difficult as doing the task well. Repetition is another common strategy, though "Babbage's Law" warns about the tendency for multiple people to make the same mistakes when doing a task the same way [25]. Randomization can help with this in the case of multiple choice questions.

Another strategy is "gold standard data", which CrowdFlower[1] makes good use of. The idea is to present workers with some tasks for which the answer is known. If a worker gets such a task wrong, then this provides some evidence that they may have done poorly on other tasks. One drawback of this approach is that it requires keeping track of workers over time, or presenting multiple questions in a single task.

Bernstein et al. [5] mention the power of multiple independent agreement as another quality control measure. This is most effective for open-ended tasks where the space of possibilities is very large, so that the probability of multiple workers arriving at the same answer by chance is very small. Babbage's Law may apply here as well, but the open-ended nature of the task increases the chance that different people will approach the task differently of their own accord.

Of course, some open-ended tasks do not have correct answers at all, like writing an image description. In these cases, independent agreement will not work, since it is extremely unlikely that two individual workers will write exactly the same thing. We deal with this problem by having a second task to verify the quality of the open-ended task, where this second task can use other quality control strategies. This is similar to the "Verify" step in Find-Fix-Verify [5].

Why Isn't This Good Enough?

All of the approaches above are good. Unfortunately many of these strategies are design patterns rather than abstractions. They do not shield the user from needing

[1]http://www.crowdflower.com

to think about quality control. For instance, Kittur recommends making tasks as difficult to do correctly as incorrectly, but it is not clear how to achieve this goal for any given task. Similarly for Babbage's Law, it is not always clear how to create a set of tasks that push users along different paths toward the same answer.

Users of these quality control strategies also need to think about the efficiency needs of their particular task. For instance, when using repetition, how many repetitions are necessary? More repetitions are needed as the difficulty of a task increases, but nobody has a formula for deriving the optimal number of repetitions for any given task. Dai et al. [16] propose using artificial intelligence to discover this number by modeling workers over time. However, this brings us to the next problem: some of these methods require keeping track of workers over time. This puts requesters who do not post many tasks on MTurk at a disadvantage, since they are less likely to have prior data on each worker who does their tasks.

Finally, the strategies above do not cover all types of tasks for which we may want to control quality. One large class of tasks for which we do not know how to control for quality is subjective assessment. If I ask someone to rate the aesthetic quality of a picture on a scale of 1 to 10, how do I know how well they did? There is no ground truth — different answers will be correct for different people — so there is no way to use independent agreement or gold standard data. Nevertheless, tasks of this form are important for human computation, making this an important area for future research.

Suggestions

In the near term, I think the most powerful improvement in quality control will come from finding ways to share and aggregate more sophisticated worker quality data. MTurk already does this a little bit. For instance, when a worker has a task rejected on MTurk, this information updates a rating that can be used by all requesters in the system. However, this is a relatively weak quality signal, since many poor quality workers have high approval ratings. This happens in part because some requesters

approve all their assignments, sometimes because it is difficult to automatically detect fraudulent work, and it is too difficult to look over all the results by hand.

CrowdFlower has a more sophisticated quality signal in the form of answers to gold standard questions. Gold standard questions are great for problem domains where at least some of the tasks have unambiguous answers. It would be useful to have similar quality control signals for open-ended tasks like writing an image description. This could possibly be done if the image description is later rated or compared with another description written by someone else. In this case, the results of that rating or comparison could be used as a quality signal for the worker who wrote the description.

Building a system to share and aggregate worker quality data suggests some research questions. First, how do we aggregate quality data from different sorts of tasks into a single worker quality rating? This may be impossible, since workers may have different qualities at different sorts of tasks, in which case the research question involves coming up with a set of quality dimensions.

Second, how do we prevent cheating? For instance, on MTurk, workers can increase their approval rating by creating a bunch of tasks, doing those tasks themselves, and then approving all the results. This is a difficult research problem, and probably involves developing a quality rating for requesters as well as workers.

6.1.2 Task Interfaces

When I first began my research in human computation, I felt like the biggest programming challenge was finding a way to deal with the high latency of human computation tasks. TurKit does a fair job of dealing with this issue. The real technical bottleneck now for writing human computation algorithms is creating interfaces for all of the tasks, and wiring together these interfaces with the algorithm.

Graphical user interfaces are notoriously difficult to design in general, and human computation tasks are no exception. The field of human computation can draw insight from the field of human computer interaction, but introduces its own peculiar set of design constraints. In particular, human computation tasks fall in an odd place in the design space. Human computation tasks are similar to data entry interfaces, except

that data entry interfaces are typically used by workers who have been trained in how to use them. Human computation tasks, on the other hand, are often faced with novel workers, and so the interface itself needs to instruct workers in what they are doing, as well as provide an efficient method for inputting data.

Wiring interfaces together with MTurk and the user's backend system introduces its own set of complications. Some human computation systems on MTurk involve interfaces that send data directly to a database. In this case, the control algorithm must manage three separate entities involved with a task interface: the database, the webpage for the task, and MTurk. These entities must also talk to each other.

This is what makes `mt.promptMemo('what is your favorite color?')` so appealing. It draws a layer of abstraction around the entire interface system, including the webpage, possibly a database, and MTurk itself. The crucial idea here is to couple the code for wiring an interface to MTurk with the interface itself. This draws a clean separation between code that orchestrates a collection of tasks, and code that controls an individual task.

This abstraction may also make it easier for developers to share interface templates. One could even imagine packaging an interface template as a web service with a standard REST API similar to that of MTurk itself. Human computation may be uniquely suited to take advantage of web service abstractions, since both money and the web are involved in even the simplest human computation algorithms.

6.1.3 Basic Operations and Algorithms

Computer science has a set of basic operations and algorithms for solving common problems, like addition and sorting. Human computation would benefit from the same, so that people don't need to keep thinking at the level of creating new MTurk tasks.

Subjective Assessment

One simple task for which humans have a significant edge over computers is subjective assessment. A couple basic subjective operations include comparing two items, or rating a single item. Since these operations are so common and useful, it would be nice to have some standard interfaces that are optimized.

One issue we face is quality control. It is difficult to know if people are lying about subjective assessments. Even if they are not lying, it is hard to know how hard they are trying — if we are asking for a subjective opinion, how informed an opinion is it? If we are comparing the aesthetic quality of two images, did the user casually glance at the images, or really consider each image? The most obvious approach to try is seeing how correlated people are with each other. However, this discourages genuine disagreement. Another potential avenue for addressing subjective quality control is Bayesian Truth Serum [41], which has already been explored a bit in the context of human computation in [46], where it was linked with higher performance on tasks.

In addition to getting workers to provide genuine assessments, and perhaps as part of it, it would be good to know which interfaces make this easiest. When comparing two images, should they be presented side by side, or one above the other? How big should the images be? What instructions should there be? What interface should be provided for making the choice? If we are asking for a rating, the interface has even more options. Should we be using a Likert scale from 1 to 7, a rating from 1 to 10, or a star rating from 1 to 5?

Subjective Sorting

Sorting is one of the most basic algorithms in computer science, and subjective sorting may be one of the most basic human computation algorithms. We still do not know the best way to sort using humans. Should we make O(n log n) comparisons, or O(n) absolute ratings, and then sort using those ratings? Again, these are difficult questions to answer, since there is no ground truth. This is probably one of the most important open questions in human computation.

Brainstorming / List Generation

Brainstorming, or list building, seems like a basic operation for any kind of automated design or creative process, like writing an article. We might have a paragraph topic, and want to brainstorm supporting points. Or, drawing from [32], we might have a decision to make about a car, and we want to brainstorm dimensions along which cars might be compared (issues like safety and gas efficiency). Note that brainstorming builds on our previous operations, since subjective sorting may be a useful way of finding the best items from a brainstorming session.

Clustering / Feature Detection

Clustering and feature detection are viewed as fairly high-level algorithms from traditional computation point of view, but for human computation, I think they may be fundamental. It seems common to have a set of items, and want to know a set of dimensions that seem useful for grouping them. One way to do this is to show people all pairs of items, and ask how similar they are (a rating task). Then a standard clustering or dimensionality reduction algorithm can be applied. Of course, this suffers from the problem that the clusters or dimensions are somewhat inscrutable to humans — the algorithm does not tell us what the clusters or dimensions mean.

Another approach, building on our previous operations, is to have people brainstorm dimensions, picking the best ones, and then having people rate each item along each dimension. This approach may offer many advantages over traditional machine learning approaches, in particular, the dimensions would have human readable names and meanings.

This algorithm might be difficult to standardize, but the payoff seems high as it would make a good building block for more complex algorithms. For instance, if we wanted to write an article about a given topic, we might brainstorm issues related to that topic, and then run a human clustering algorithm to group the issues into sections.

6.1.4 Human Modeling

Computer science gains a lot of power by the fact that all computation can be broken down into a small set of operations, and these operations are easy to model in terms of time. This modeling may even take into account the physical hardware, e.g., a certain Intel processor might perform floating operations faster if it has access to a special floating point chip.

Good models of low level instructions provides two crucial benefits. First, it allows people to design complex algorithms without needing to run physical tests to affirm every design decision. Second, it allows compilers to optimize algorithms.

We would like to gain the same benefits in the realm of human computation. However, human tasks have many more independent variables that need to be modeled than traditional digital operations. These variables include properties of tasks, like difficulty and enjoyableness, as well as properties of humans doing tasks, like skill-level and motivation. These models must also account for change over time, as a human learns or becomes bored of various tasks. Modeling humans also requires modeling human computation markets, since different markets will have different distributions of humans, as well as different interfaces and protocols for connecting humans with tasks.

The dependent variables are also more complicated for human tasks. On a computer, we mostly care how long an operation takes to execute, and possibly what effect it has on a microprocessor's memory and cache. However, we generally assume the operation will execute perfectly. For human tasks, we care how long they take, as well as how well they are done. Also, quality is a multidimensional property in itself, and may be task dependent, adding to the complexity of the model.

Because the modeling problem is so large, one good research direction is discovering which aspects of the model are the most important. Another fruitful direction may be developing real-time empirical models, as tasks are being completed. Research along this latter direction is already underway in [29] and [16].

6.1.5 Low Overhead

Right now, human computation is something that researchers and big businesses play around with. There is a lot of overhead to experiment with the technology. I hope that TurKit will help, but it still requires programming expertise. I think we need to dramatically lower the barrier of entry to experimenting with outsourcing, so that almost anybody can play around with it. I think one good way to do this is to come up with a set of generic processes, like brainstorming given a topic, sorting a set of items according to a criteria, proofreading some text, etc.. and making web interfaces for these tasks that people can access with PayPal.

The reason to lower the barrier to entry for experimenting with this technology is to discover new and interesting ways it can be used. Researchers try to be creative, but I do not think they can compete with hundreds of thousands of people trying out different ideas. We see this phenomena in websites already. It has been my experience, and the experience of others, that practically any good idea for a website already exists. It is almost a law. If an idea for a website is good, that means it must exist — at least, something very close to it. I believe the reason for this is that building web sites has a very low barrier to entry these days, and that if we want to explore a new space like human computation, we should lower the barrier to entry to take advantage of the creative resources of the crowd.

6.1.6 Bottom-up versus Top-down

The goal of this thesis has been to build complex human computation algorithms capable of outperforming experts at various tasks. Thus far, we have been pursuing a bottom-up path toward this goal. This approach follows the tradition of computer science, where we start with a relatively simple set of operations, and build increasingly high-level abstractions over those operations. Eventually, these high-level abstractions allow programmers to write extremely complex algorithms to solve a large range of problems.

One alternative to the bottom-up approach is a top-down approach. The top-down approach would involve looking at how experts already solve the sorts of problems that we would like to write human computation algorithms for, and having these experts outsource small pieces of these problems. For instance, a programmer might outsource a well-defined function, and a writer might outsource a proofreading task. We would then seek to outsource more pieces, and begin to break the pieces down into smaller pieces. This is really just an extension of the way companies already break down knowledge work tasks, but using hyper-expertise task markets to push the scale even smaller.

The bottom-up approach faces some challenges that may be aided by the top-down approach.

Human Computation versus Digital Computation

Although it seems natural to pursue human computation along the same path as digital computation, human computation differs in a couple of important ways. First, the space of low-level operations for human computation is extremely large, and it may not be efficient to try and map out the entire space, as can be done with digital operations. The top-down approach may provide some forward-looking insight into which operations are the most useful to map.

Second, human computation does not lend itself as easily to abstraction as digital computation. There is an inherent tradeoff between the generality and efficiency of abstractions. Davenport describes this tradeoff in [17] as he attempts to speed up REDUCE, a general algebra system, by tailoring it to his specific needs, and hence making it less general. This is an example of where a computer scientist was pushing the speed limits of digital computation, and needed to make some compromises in the abstraction he was using. However, computer scientists have been benefited greatly by Moore's Law, which roughly predicts a doubling in processing speed every two years. On today's hardware, programmers often have the luxury of building extremely general abstractions without making any perceptible difference in execution speed.

However, human computation tasks are unlikely to benefit from an exponential speed increase, meaning that human computation programmers will need to grapple with the problem of finding useful abstractions that do not compromise cost and speed. As before, the top-down research approach may lend insight into which abstractions are worth developing.

Expertise

Human computation is usually thought of in terms of tasks that almost any human could do, however, there is no reason to restrict human computation systems in this way. In fact, the sophisticated algorithms we hope to create may require experts to perform some of the pieces. For instance, it may be difficult to have a human computation algorithm write a program if none of the humans involved know how to write code.

However, for socioeconomic reasons, it may be difficult to pursue a pure bottom-up approach to human computation, because it may be hard to find sufficient expertise within micro-task markets. The transition from current hiring practices to micro-tasks may be too large of a jump. The top-down approach, on the other hand, may provide a more graceful transition for workers, from full-time jobs to relatively large freelance jobs, to increasingly small freelance jobs, to hyperspecialized micro-tasks.

In either case, hyperspecialization presents its own set of research challenges. In particular, routing tasks to experts is a problem. Part of the reason it is a problem on MTurk is that the search interface has few affordances allowing experts to find tasks geared toward their expertise. There is a keyword search box, but nothing like the hierarchical view of various expertise types offered by sites like Elance[2] and oDesk[3]. Of course, one issue with sites like these with respect to human computation is that they do not yet have complete hiring APIs. Freelancer[4] does, but the bidding system still makes it a bit cumbersome. It would be nice to have a system abstracted enough that a computer can make a request requiring a certain expertise, and have

[2] http://www.elance.com
[3] http://www.odesk.com
[4] http://www.freelancer.com

a reasonable expectation that a person with the appropriate expertise will do the task for a reasonable price. This abstraction might involve a human computation algorithm that hires some humans to hire other humans. However, to get this all to work still seems to rely on people with some expertise in identifying good people to hire, and since the risk of failure is higher here, quality control is even more important.

6.2 Axes of Good and Evil

It is possible that human computation and micro-labor markets are a simple and natural extension of current hiring practices. However, it is also possible that human computation dramatically reshapes the labor market, in much the same way as the industrial revolution. In a sense, human computation, coupled with electronic micro-task markets, is a new and poorly understood technology with the potential to influence humanity on a global scale. It is with that potential in mind that this section discusses possible pros and cons of the technology to keep in mind as researchers and practitioners as we move forward.

6.2.1 Globalization versus Low Wages

Human computation is pushing in the direction of a global economy. Many workers on freelancing sites like Elance, oDesk and Freelancer work in developing countries all over the world. MTurk has also opened up to India, and may have workers from other countries as well, provided they have access to bank accounts in the United States.

From one point of view globalization is good, providing work to people in poor countries. From another point of view it is bad, forcing people in rich countries to compete in terms of wages with people in poorer countries, where the cost of living is lower. The hope, of course, is that although wages may fall in the short term for people in rich countries, the increase in production due to a higher efficiency labor distribution will lead to products and services that increase the standard of living for everyone.

There are also concerns that the wages offered on sites like MTurk are so low as to be exploitative, causing fears of so called "digital sweatshops" [63]. However, it is important to consider that many workers may accept lower wages because certain tasks are fun, interesting, or pass the time, similar to playing a Flash game on the internet. It is certainly the case that more enjoyable tasks can be completed on MTurk for lower wages [40]. I have even had workers submit results for a task offering no money at all, but posing a vague and apparently intriguing question "what is the answer?".

Another thing to note is that currently, freelance and micro-task markets are niche, and comparatively little work is routed through them. There are many more workers than work to be done. Under these conditions, it makes sense that wages would be low. Hence, it may be premature to worry about current wages in these markets, since they may change radically if we can convince more people to hire labor.

The real question to answer now is, how do we get people to trust freelance labor and micro-task markets? If wages are too low, they may not attract high quality workers, which in turn may perpetuate skepticism about the viability of using freelance labor. This is a classic chicken and egg problem, and one worthy of the attention of researchers in the field of human computation.

6.2.2 Learning versus Cheating

Outsourcing on a small-scale offers the potential to aid education. For instance, people may be able to hire just-in-time tutors to clarify concepts. Also, if we are able to break down tasks into bite-sized pieces, it may be possible for people to learn new skills by doing simple tasks in a domain, and gradually learning to do more complicated tasks — a form of online apprenticeship.

On the other hand, one common concern about small-scale outsourcing is that it enables cheating. It is already the case that outsourcing sites have been used by students to hire people to do their homework. This sort of cheating is similar to plagiarism, but may be more difficult to detect, since the results may in fact be novel.

This is a difficult problem to tackle, particularly with human computation, since we are trying to move toward increasingly small-scale tasks, which may be the same size and complexity of standard homework questions. This causes a problem for solutions which try to detect "homework problem" tasks. It also poses a problem for workers, who may want to avoid doing other people's homework, but have difficulty distinguishing homework tasks from legitimate tasks. I have experienced this myself doing a task on oDesk — I did a task which may have been part of a legitimate application, but may also have been a graphics related homework assignment.

However, when thinking about the risk of cheating, it is also important to remember that outsourcing may enable higher level problem solving abilities for students. If we hamper outsourcing to reduce cheating on current homework assignments, we also give up a rich area of new homework assignments that involve outsourcing. One could imagine teaching students to break down vague problems into clear specifications which can be outsourced, and teaching them to recognize good quality results. This is similar to teaching students how to use calculators effectively, which involves more than simply entering mathematical expressions. It also involves knowing what mathematical expression to use to solve a given problem, and knowing how to recognize mistakes, like a negative result that should be positive.

6.2.3 Personal Autonomy versus Dehumanization

One hope of the emergence of freelance work and human computation is that it will increase personal autonomy and freedom. The idea is that people can choose where to work, when to work, and what to work on. In many ways, they are their own boss. The system may also help break down barriers to entering certain fields. For instance, it is very hard to get hired as a full-time software engineer in America without a computer science degree. Employers are using the degree as a quality control filter. However, it is easier to get hired as a freelancer without a degree, since an employer only cares whether a freelancer knows how to do a specific task.

Of course, the idea of human computation also has a flavor of thinking about human workers as cogs in a machine. This has some risks. First, workers may not

like being viewed as cogs by their bosses. Second, bosses may be more likely to mistreat or exploit employees because they never really see them or get to know them.

On the other hand, workers may like the separation from their bosses. There is already a stereotypical tension between employees and bosses, where the cultural wisdom is that it is difficult to really be friends with your boss, since the relationship is asymmetrical.

By the same token, workers often value getting to know other workers at their same level. The stereotype involves workers talking at the water cooler. It may be important to facilitate this same ability online with virtual water coolers. In fact, although MTurk does not provide any facilities for workers to get to know each other, a separate website called Turker Nation[5] developed to meet this need, suggesting that worker value communication with their peers.

6.2.4 Transparency versus Hidden Agendas

In one sense, freelancing sites provide more transparency with regard to what people are working on. Anyone can browse the tasks posted on MTurk, Elance or oDesk. Some tasks are secretive, and reveal more information to the worker once they have been hired, but many tasks reveal the entire specification publicly. This makes sense for two reasons. First, it is easier for workers to self-select the tasks they are most suited for if they know the exact requirements before bidding. Second, small-scale tasks often do not reveal too much about the larger project that a task is a part of.

This gets to the evil side of this coin. Since it is not always clear what the larger agenda is of a task, there is the potential for workers to unwittingly perform tasks that aid an evil hidden agenda. Jonathan Zittrain gives the chilling example of kids playing a face detection game-with-a-purpose, where the purpose is to identify political protesters in a country that would like to imprison those protesters.

Although this is an important issue to consider, it is important to bear in mind that hidden agendas are not exclusive to the domain of human computation. Large

[5]http://turkers.proboards.com/index.cgi

companies sometimes have operations overseas that may harm the environment or exploit workers, and people working for the company or buying their products are not aware of the evil they are doing. The usual method for dealing with this sort of thing, once it is exposed, is to boycott the company's products. In fact, human computation markets may be used at a meta level to police themselves and discover hidden agendas. This may be aided by the fact that nefarious companies doing work through human computation markets probably leave a longer digitized "paper trail" in which to detect patterns. Still, it is probably worth keeping these issues in mind up front as we design the next generation of outsourcing and freelance intermediaries, so that they facilitate and encourage ethical business practices.

6.2.5 Hyper-specialization versus Deskilling

We mentioned in the introduction the possibility of an industrial revolution of knowledge work. The idea is that we may be able to breakdown knowledge work tasks into small well-defined pieces, similar to the way an assembly line breaks down the task of constructing a physical product into well-defined operations. Much of the desired efficiency gain would come from knowledge workers hyper-specializing in small well-defined tasks.

However, hyper-specialization may also lead to deskilling. For instance, a full-time programmer at a top software company today must be well versed in a number of languages and APIs, and know how to adapt to new programming domains. If we manage to breakdown the task of software engineering into well-defined pieces, it may be that software can be developed by people who only know how to perform a single task within the system.

Deskilling has a number of disadvantages. From a labor point of view, it devalues work, since workers with less skill are able to compete in the market. Another potential disadvantage is that deskilling reduces the number of people who have a holistic view of a problem domain, which may mean less innovation in the domain.

Advancement in human computation probably will lead to some deskilling, but as we saw in the industrial revolution, it may also lead to new skill-based jobs. For

instance, we will need people capable of breaking down problems into well-defined pieces that can be outsourced, and integrating the results.

Of course, human computation can be a bit meta. One could imagine a human computation algorithm capable of breaking down problems into well-defined pieces and integrating the results. At this point, the question becomes, who is at the top telling the system what to do? I hope the answer is: we are all at the top. Everyone uses outsourcing to accomplish their own creative goals by hiring people to do what they do not know how to do, and financing this by working on other people's tasks.

6.3 The Future

This section is pure speculation, mixed with a bit of hope. I am going to list some things that I think will happen in the future, not so future dwellers can look back and see whether I was right, but rather as a list of goals. Here is what I see:

6.3.1 Experts Integrating Human Computation

I see a style of programming where a programmer expresses a problem in terms of well-defined modules which are outsourced in parallel in real-time, and then integrated. I believe even hackers — programmers trying to write code as quickly as possible with little regard for future maintainability — will write code this way. Hackers are known for writing hairy masses of tightly coupled code. Why would they bother to write clearly specified modules? I believe parallelization is the key, and clearly specifying modules may be the only way to parallelize code with any hope of integrating the components.

I see a style of writing where an author works with a crowd of expert writers as well as unskilled writers. The author may sketch an outline of what they want to say, and ask a crowd to brainstorm supporting points for each section. Other workers may rank and cluster these ideas, and still other workers may turn the results into prose. All the while, the author can monitor the system, and make changes, like removing

a section if it is not going in the right direction, or supplying additional instructions. The author would be more of a director of the writing process.

I see a style of graphic design based around a human computation algorithm. Sites like 99designs run a large contest for a finished design. I see a multistage process where the first stage has people draw rough sketches, and other people rank and cluster these sketches. The best sketches are improved in the next stage with more refined sketches. The best of these are then done up in Illustrator, and the crowd ranks the results so the end user has just a few high-quality logos to chose from. As before, a top-level designer may guide the process at each stage, or refine the design specifications.

6.3.2 Meta-Experts

I see people who are experts at routing tasks to experts. They will be paid money to route a task to someone who can do it well. There may even be networks of expert routers, with algorithms that distribute payments appropriately. These algorithms may be similar to the method employed by MIT's Red Balloon Challenge Team[6].

I see people who are experts at taking vague ideas and forming concrete specifications given the available technologies. When a non-programmer has an idea for a new iPhone app, for instance, they are not always aware of what the iPhone is capable of, or exactly what they want. These experts would specialize in making someone's ideas concrete, so that the work could be outsourced, or the idea rejected as infeasible.

6.3.3 New Self-Expression

I see a style of creative expression that involve outsourcing parts of a project that the creative director is least adept at. Perhaps I have an idea for a three panel comic, but I can't draw. I can draw a rough sketch and outsource the final drawing to someone else, iterating with them until the result is satisfactory. Perhaps it will be possible

[6]http://balloon.media.mit.edu/mit/payoff/

for people with ideas for painting or songs, but no skills in painting or music, to see their ideas realized.

I see a form of creative barter, where people finance their own projects by helping other people with their projects. For instance, I can program in JavaScript, so I can help people with their web projects, and thereby finance my desire to create music.

6.3.4 Human Computation Algorithms

I see wholly automated human computation algorithms that take some high level instructions, and do not return until they have a finished result. An input might be "create a server for people to play Go online", or "research and write an article about the recent X scandal". This would be analogous to an assembly line that takes ore in one end, and delivers a car on the other end.

I see programs that are able to repair themselves. I have a cron job that runs every 10 minutes, checking a Rhapsody RSS feed to see what music I listened to, and uploading the results to Last.fm. However, what happens if Rhapsody changes their RSS feed, or Last.fm changes their API? The script will throw an exception which I may not notice for days. I see programs like this hiring workers to repair themselves, presenting workers with an online interface for editing and debugging code, as well as hiring other workers to verify the solution before integrating the changes.

Chapter 7

Conclusion

This thesis embarked along an ambitious path to create complex human computation algorithms capable of writing essays, designing software, and creating presentations. I had even hoped that this thesis would be written using such an algorithm — alas. Still, this thesis makes several contributions which I hope will help others advance the field of human computation.

First, we presented a toolkit for writing human computation algorithms on MTurk. The high latency of human computation presents a unique set of design constraints for human computation programming. The TurKit toolkit helps manage this high-latency using the crash-and-rerun programming model, which remembers costly results between runs of a program so that they do not need to be redone. In particular, the system remembers the results of HITs posted on MTurk so that they do not need to be re-posted. This allows programmers to maintain a relatively fast iterative design cycle as they prototype new algorithms, while still programming in a familiar imperative style.

We focused on trying to maintain a familiar programming style. We wanted to maintain the illusion of introducing a human procedure call to a conventional programming paradigm. However, our efforts come with some caveats. For instance, the crash-and-rerun programming model differs in some subtle ways from traditional programs, which are not always clear to new programmers in this model. Also, the crash-and-rerun model does not scale to continuously running processes, since it will

take longer and longer for the program to get back to where it was after rerunning. Finally, the parallel programming capabilities of the system are difficult to discover and use.

There are many decent alternatives to crash-and-rerun programming that fix the scalability and usability issues. However, the crash-and-rerun programming style introduces some compelling advantages difficult to find elsewhere. Most importantly, crash-and-rerun programs can be modified between reruns. This allows a programmer to write the first part of an algorithm, make sure it works, and then write the next part of the algorithm without needing to wait for the first part to execute on MTurk. In fact, it is possible to modify parts of an algorithm which have already executed, in order to collect additional information into variables for later use, or print debugging information.

The purpose of TurKit is to promote exploration in the field of human computation, and we site several case studies where people have used it toward that end. TurKit is open source, and available at http://turkit-online.appspot.com.

We used TurKit ourselves to implement and compare iterative and parallel human computation processes. In the iterative approach, each worker sees the results from previous workers. In the parallel process, workers work alone. Our experiments apply each algorithm to a variety of problem domains, including writing, brainstorming, and transcription. We use TurKit to run several instances of each process in each domain.

We discover that iteration increases the average quality of responses in the writing and brainstorming domains, but that the best results in the brainstorming and transcription domains may come from the parallel process. This seems related to the greater variance of responses generated in the parallel condition. We also see that providing guesses for words in the transcription domain can lead workers down the wrong path, like a gradient descent toward a local minimum.

More experiments like this will need to be run in order to really chart the space of low-level human computation algorithms, but this is a start. We need a foundation upon which to build larger algorithms, and hopefully these experiments contribute to that foundation.

Next, we examine a case study of building a larger human computation system, composed of several phases, each of which uses an iterative or parallel algorithm to achieve some goal. The system performs transcription on a set of hand-completed forms, with an effort not to reveal too much information from the forms to the human workers. We decompose the problem into three phases: the first phase identifies information regions on the form, the second phase provides labels for each region, and the final phase has humans transcribe what is written in each region.

We ran this system on a real-world set of forms, and discovered a number of pros and cons. For the most part, the system worked well, including the system for dynamically partitioning the region selection task between workers. No worker needed to draw all the regions, but we also did not need to divide up the form upfront. On the other hand, we discovered a number of risks related to showing transcription workers only a small window of a form, since sometimes the transcription would be aided by seeing other parts of the form.

After getting our hands dirty building tools, systems and running experiments, we come away with an impression of the state-of-the-art of human computation. We express our own view of where we are at, what problems we face, and where we think human computation is headed. I think the most pressing problem today is getting more people to use human computation — the more people use it, the more brains will be motivated to fix all the problems. The biggest hurdle to getting more people to help explore human computation is getting them to understand what can be done, making it easy for them to do it, and ensuring high quality results. Also, we should not just think about this from a bottom-up approach, getting people to use micro-tasks on MTurk. We should also consider exploring top-down approaches, looking at how modern knowledge workers go about their tasks, and finding possible pieces to outsource as part of their workflow.

One thought to leave with is this. Humanity has achieved many great things over the course of human existence, even though individual humans only live for 100 years. Hence, a 100 year window is enough for a person to learn the current state-of-the-art in humanity, at least in some narrow domain, and make a contribution in

that domain that allows the next person who comes along to get even further. We also see companies with achievements surpassing any individual worker, where each employee only works perhaps 30 years at the company. Today, many people move between companies every 5 years. How small can we make this window? If any one person can only contribute to a company for a month, can it achieve arbitrarily great things? If we design a human computation algorithm with clever protocols for focusing workers on different sub-problems within a system, and present them with just the most relevant information, can we make this window as small as a day, or an hour? If so — a big if — can we parallelize the algorithm? And if so, what can nearly 7 billion people achieve in a day using this algorithm?

Bibliography

[1] Business process execution language for web services (bpel), version 1.1. http://www.ibm.com/developerworks/library/specification/ws-bpel/, July 2002.

[2] Lada A. Adamic, Jun Zhang, Eytan Bakshy, and Mark S. Ackerman. Knowledge sharing and yahoo answers: everyone knows something. In *Proceeding of the 17th international conference on World Wide Web*, WWW '08, pages 665–674, New York, NY, USA, 2008. ACM.

[3] Luis Von Ahn, Manuel Blum, Nicholas J. Hopper, and John Langford. Captcha: using hard ai problems for security. In *Proceedings of the 22nd international conference on Theory and applications of cryptographic techniques*, EUROCRYPT'03, pages 294–311, Berlin, Heidelberg, 2003. Springer-Verlag.

[4] Joyce Berg, Robert Forsythe, Forrest Nelson, and Thomas Rietz. Results from a dozen years of election futures markets research, 2001.

[5] Michael S. Bernstein, Greg Little, Robert C. Miller, Björn Hartmann, Mark S. Ackerman, David R. Karger, David Crowell, and Katrina Panovich. Soylent: a word processor with a crowd inside. In *Proceedings of the 23nd annual ACM symposium on User interface software and technology*, UIST '10, pages 313–322, New York, NY, USA, 2010. ACM.

[6] Jiang Bian, Yandong Liu, Eugene Agichtein, and Hongyuan Zha. Finding the right facts in the crowd: factoid question answering over social media. In *Proceeding of the 17th international conference on World Wide Web*, WWW '08, pages 467–476, New York, NY, USA, 2008. ACM.

[7] Jeffrey P. Bigham, Chandrika Jayant, Hanjie Ji, Greg Little, Andrew Miller, Robert C. Miller, Robin Miller, Aubrey Tatarowicz, Brandyn White, Samual White, and Tom Yeh. Vizwiz: nearly real-time answers to visual questions. In *Proceedings of the 23nd annual ACM symposium on User interface software and technology*, UIST '10, pages 333–342, New York, NY, USA, 2010. ACM.

[8] Frederick P. Brooks. *The Mythical Man-Month: Essays on Software Engineering.* Addison-Wesley, 1975.

[9] Susan L. Bryant, Andrea Forte, and Amy Bruckman. Becoming wikipedian: transformation of participation in a collaborative online encyclopedia. In *Proceedings of the 2005 international ACM SIGGROUP conference on Supporting group work*, GROUP '05, pages 1–10, New York, NY, USA, 2005. ACM.

[10] Marcelo Cataldo, Patrick A. Wagstrom, James D. Herbsleb, and Kathleen M. Carley. Identification of coordination requirements: implications for the design of collaboration and awareness tools. In *Proceedings of the 2006 20th anniversary conference on Computer supported cooperative work*, CSCW '06, pages 353–362, New York, NY, USA, 2006. ACM.

[11] Meeyoung Cha, Haewoon Kwak, Pablo Rodriguez, Yong-Yeol Ahn, and Sue Moon. I tube, you tube, everybody tubes: analyzing the world's largest user generated content video system. In *Proceedings of the 7th ACM SIGCOMM conference on Internet measurement*, IMC '07, pages 1–14, New York, NY, USA, 2007. ACM.

[12] Lydia B. Chilton, John J. Horton, Robert C. Miller, and Shiri Azenkot. Task search in a human computation market. In *Proceedings of the ACM SIGKDD Workshop on Human Computation*, HCOMP '10, pages 1–9, New York, NY, USA, 2010. ACM.

[13] Lydia B. Chilton, Clayton T. Sims, Max Goldman, Greg Little, and Robert C. Miller. Seaweed: a web application for designing economic games. In *Proceedings of the ACM SIGKDD Workshop on Human Computation*, HCOMP '09, pages 34–35, New York, NY, USA, 2009. ACM.

[14] T.N. Cornsweet. The staircase-method in psychophysics. *The American Journal of Psychology*, 1962.

[15] Dan Cosley, Dan Frankowski, Loren Terveen, and John Riedl. Suggestbot: using intelligent task routing to help people find work in wikipedia. In *Proceedings of the 12th international conference on Intelligent user interfaces*, IUI '07, pages 32–41, New York, NY, USA, 2007. ACM.

[16] Peng Dai, Mausam, and Daniel S. Weld. Decision-theoretic control of crowd-sourced workflows. In *In the 24th AAAI Conference on Artificial Intelligence (AAAI10*, 2010.

[17] James H. Davenport. Fast reduce: the trade-off between efficiency and generality. *SIGSAM Bull.*, 16:8–11, February 1982.

[18] Richard Dawkins. *The Blind Watchmaker.* Norton & Company, Inc, 1986.

[19] Manal Dia. On decision making in tandem networks. Master's thesis, Massachusetts Institute of Technology, 2009.

[20] Steven Dow, Julie Fortuna, Dan Schwartz, Beth Altringer, Daniel Schwartz, and Scott Klemmer. Prototyping dynamics: sharing multiple designs improves exploration, group rapport, and results. In *Proceedings of the 2011 annual conference on Human factors in computing systems*, CHI '11, pages 2807–2816, New York, NY, USA, 2011. ACM.

[21] Steven P. Dow, Alana Glassco, Jonathan Kass, Melissa Schwarz, Daniel L. Schwartz, and Scott R. Klemmer. Parallel prototyping leads to better design results, more divergence, and increased self-efficacy. *ACM Trans. Comput.-Hum. Interact.*, 17:18:1–18:24, December 2010.

[22] Stuart I. Feldman and Channing B. Brown. Igor: a system for program debugging via reversible execution. *SIGPLAN Not.*, 24:112–123, November 1988.

[23] Uri Gneezy and Aldo Rustichini. Pay enough or don't pay at all. *Quarterly Journal of Economcis*, 115:791–810, 1998.

[24] D.A. Grier. Programming and planning. *Annals of the History of Computing, IEEE*, 33(1):86 –88, jan. 2011.

[25] David Alan Grier. *When Computers Were Human.* Princeton University Press, 2005.

[26] Carl Gutwin, Reagan Penner, and Kevin Schneider. Group awareness in distributed software development. In *Proceedings of the 2004 ACM conference on Computer supported cooperative work*, CSCW '04, pages 72–81, New York, NY, USA, 2004. ACM.

[27] Jeffrey Heer and Michael Bostock. Crowdsourcing graphical perception: using mechanical turk to assess visualization design. In *Proceedings of the 28th international conference on Human factors in computing systems*, CHI '10, pages 203–212, New York, NY, USA, 2010. ACM.

[28] Guido Hertel, Sven Niedner, and Stefanie Herrmann. Motivation of software developers in open source projects: an internet-based survey of contributors to the linux kernel. *Research Policy*, 32:1159–1177, 2003.

[29] Eric Huang, Haoqi Zhang, David C. Parkes, Krzysztof Z. Gajos, and Yiling Chen. Toward automatic task design: a progress report. In *Proceedings of the ACM SIGKDD Workshop on Human Computation*, HCOMP '10, pages 77–85, New York, NY, USA, 2010. ACM.

[30] Aniket Kittur, Ed H. Chi, and Bongwon Suh. Crowdsourcing user studies with mechanical turk. In *Proceeding of the twenty-sixth annual SIGCHI conference on Human factors in computing systems*, CHI '08, pages 453–456, New York, NY, USA, 2008. ACM.

[31] Aniket Kittur and Robert E. Kraut. Harnessing the wisdom of crowds in wikipedia: quality through coordination. In *Proceedings of the 2008 ACM conference on Computer supported cooperative work*, CSCW '08, pages 37–46, New York, NY, USA, 2008. ACM.

[32] Aniket Kittur, Boris Smus, and Robert E. Kraut. Crowdforge: Crowdsourcing complex work. Technical report, Carnegie Mellon University, 2011.

[33] Andrew J. Ko and Brad A. Myers. Finding causes of program output with the java whyline. In *Proceedings of the 27th international conference on Human factors in computing systems*, CHI '09, pages 1569–1578, New York, NY, USA, 2009. ACM.

[34] A. Kosorukoff. Human based genetic algorithm. In *Systems, Man, and Cybernetics, 2001 IEEE International Conference on*, volume 5, pages 3464 –3469 vol.5, 2001.

[35] Anand P. Kulkarni, Matthew Can, and Bjoern Hartmann. Turkomatic: automatic recursive task and workflow design for mechanical turk. In *Proceedings of the 2011 annual conference extended abstracts on Human factors in computing systems*, CHI EA '11, pages 2053–2058, New York, NY, USA, 2011. ACM.

[36] Karim R. Lakhani and Eric von Hippel. How open source software works: "free" user-to-user assistance. *Research Policy*, 32(6):923 – 943, 2003.

[37] Greg Little, Lydia B. Chilton, Max Goldman, and Robert C. Miller. Exploring iterative and parallel human computation processes. In *Proceedings of the ACM SIGKDD Workshop on Human Computation*, HCOMP '10, pages 68–76, New York, NY, USA, 2010. ACM.

[38] Greg Little, Lydia B. Chilton, Max Goldman, and Robert C. Miller. Turkit: human computation algorithms on mechanical turk. In *Proceedings of the 23nd annual ACM symposium on User interface software and technology*, UIST '10, pages 57–66, New York, NY, USA, 2010. ACM.

[39] Thomas W. Malone, Robert Laubacher, and Chrysanthos N. Dellarocas. Harnessing Crowds: Mapping the Genome of Collective Intelligence. *SSRN eLibrary*, 2009.

[40] Winter Mason and Duncan J. Watts. Financial incentives and the "performance of crowds". In *Proceedings of the ACM SIGKDD Workshop on Human Computation*, HCOMP '09, pages 77–85, New York, NY, USA, 2009. ACM.

[41] Dražen Prelec. A bayesian truth serum for subjective data. *Science*, 306(5695):462–466, 2004.

[42] Alexander J. Quinn and Benjamin B. Bederson. Human computation: a survey and taxonomy of a growing field. In *Proceedings of the 2011 annual conference*

on Human factors in computing systems, CHI '11, pages 1403–1412, New York, NY, USA, 2011. ACM.

[43] Al M. Rashid, Kimberly Ling, Regina D. Tassone, Paul Resnick, Robert Kraut, and John Riedl. Motivating participation by displaying the value of contribution. In *Proceedings of the SIGCHI conference on Human Factors in computing systems*, CHI '06, pages 955–958, New York, NY, USA, 2006. ACM.

[44] Eric S. Raymond. *The Cathedral and the Bazaar.* O'Reilly & Associates, Inc., Sebastopol, CA, USA, 1st edition, 1999.

[45] Emile Servan-Schreiber, Justin Wolfers, David M. Pennock, and Brian Galebach. Prediction Markets: Does Money Matter? *Electronic Markets*, 14(3):243+.

[46] Aaron D. Shaw, John J. Horton, and Daniel L. Chen. Designing incentives for inexpert human raters. In *Proceedings of the ACM 2011 conference on Computer supported cooperative work*, CSCW '11, pages 275–284, New York, NY, USA, 2011. ACM.

[47] Walter A. Shewhart. *Economic Control of Quality of Manufactured Product*, volume 509. D. Van Nostrand Company, Inc, 1931.

[48] M. Six Silberman, Joel Ross, Lilly Irani, and Bill Tomlinson. Sellers' problems in human computation markets. In *Proceedings of the ACM SIGKDD Workshop on Human Computation*, HCOMP '10, pages 18–21, New York, NY, USA, 2010. ACM.

[49] Rion Snow, Brendan O'Connor, Daniel Jurafsky, and Andrew Y. Ng. Cheap and fast—but is it good?: evaluating non-expert annotations for natural language tasks. In *Proceedings of the Conference on Empirical Methods in Natural Language Processing*, EMNLP '08, pages 254–263, Stroudsburg, PA, USA, 2008. Association for Computational Linguistics.

[50] A. Sorokin and D. Forsyth. Utility data annotation with amazon mechanical turk. In *Computer Vision and Pattern Recognition Workshops, 2008. CVPRW '08. IEEE Computer Society Conference on*, pages 1 –8, june 2008.

[51] James Surowiecki. *The Wisdom of Crowds: Why the Many Are Smarter Than the Few and How Collective Wisdom Shapes Business, Economies, Societies and Nations.* Doubleday, 2004.

[52] H. Takagi. Interactive evolutionary computation: fusion of the capabilities of ec optimization and human evaluation. *Proceedings of the IEEE*, 89(9):1275 –1296, sep 2001.

[53] Donald W. Taylor, Paul C. Berry, and Clifford H. Block. Does Group Participation When Using Brainstorming Facilitate or Inhibit Creative Thinking? *Administrative Science Quarterly*, 3(1):23–47, 1958.

[54] Alan M. Turing. Computing machinery and intelligence. *Mind*, 1950.

[55] Luis von Ahn. *Human Computation*. PhD thesis, Carnegie Mellon University, 2005.

[56] Luis von Ahn. Games with a purpose. *Computer*, 39(6):92 –94, june 2006.

[57] Luis von Ahn and Laura Dabbish. Labeling images with a computer game. In *Proceedings of the SIGCHI conference on Human factors in computing systems*, CHI '04, pages 319–326, New York, NY, USA, 2004. ACM.

[58] Luis von Ahn, Shiry Ginosar, Mihir Kedia, Ruoran Liu, and Manuel Blum. Improving accessibility of the web with a computer game. In *Proceedings of the SIGCHI conference on Human Factors in computing systems*, CHI '06, pages 79–82, New York, NY, USA, 2006. ACM.

[59] Luis von Ahn, Ruoran Liu, and Manuel Blum. Peekaboom: a game for locating objects in images. In *Proceedings of the SIGCHI conference on Human Factors in computing systems*, CHI '06, pages 55–64, New York, NY, USA, 2006. ACM.

[60] Luis von Ahn, Benjamin Maurer, Colin McMillen, David Abraham, and Manuel Blum. recaptcha: Human-based character recognition via web security measures. *Science*, 321(5895):1465–1468, 2008.

[61] Georg von Krogha, Sebastian Spaeth, and Karim R. Lakhani. Community, joining, and specialization in open source software innovation: A case study. *Ssrn Electronic Journal*, 2003.

[62] Justin Wolfers and Eric W. Zitzewitz. Prediction Markets. *SSRN eLibrary*, 2004.

[63] Jonathan Zittrain. Work the new digital sweatshops. *Newsweek*, 2009.