TurKit: Tools for Iterative Tasks on Mechanical Turk

Greg Little
MIT CSAIL
32 Vassar Street
Cambridge, Ma 02139, USA
glittle@gmail.com

ABSTRACT
Mechanical Turk (MTurk) is an increasingly popular web service for paying people small rewards to do human computation tasks. Current uses of MTurk typically post independent parallel tasks. I am exploring an alternative iterative paradigm, in which workers build on or evaluate each other's work. I describe TurKit, a new toolkit for deploying iterative tasks to MTurk, with a familiar imperative programming paradigm that effectively uses MTurk workers as subroutines, such as the comparison function of a sorting algorithm. The toolkit handles the latency of MTurk tasks (typically measured in minutes), supports parallel tasks, and provides fault tolerance to avoid wasting money and time. I also present a few examples of iterative tasks using TurKit, including image description, and handwriting recognition.

ACM Classification: H5.2 [Information interfaces and presentation]: User Interfaces. - Prototyping.

General terms: Algorithms, Design, Economics, Experimentation

Keywords: computation, Mechanical Turk, toolkit

INTRODUCTION
MTurk is an increasingly popular web service for paying people to do simple human computation tasks. Workers on the system (turkers) are typically paid a few cents for Human Intelligence Tasks (HITs) that can be done in under a minute. Currently, MTurk is largely used for *independent* tasks. Task requesters post a group of HITs that can be done in parallel, such as labeling 1000 images. I consider a different model for employing turkers: **iterative tasks**, in which a succession of turkers do tasks that build each other. For example, turkers can take turns improving a passage of text; verify each other's work by voting on it; and implement the comparison function of an iterative sorting algorithm.

Next I will touch on some related work, followed by a presentation of a couple examples of iterative tasks on Mechanical Turk. The paper will finish with an overview of TurKit—the toolkit used to create these tasks—followed by directions for future work.

UIST'09, October 4–7, 2009, Victoria, BC, Canada.

Figure 1: Mechanical Turk workers (turkers) asked to iteratively improve a description of this image.

RELATED WORK
One challenge in writing human computation algorithms is motivating humans to do work. One approach is Games With a Purpose [1], where humans perform useful computation as a byproduct of playing computer games. User-generated content websites such as Wikipedia use human computation to generate content, and this content along with social factors seem to motivate future contributions. MTurk provides a platform for performing Human Intelligence Tasks (HITs) where humans are motivated by money.

ITERATIVE TEXT IMPROVEMENT
The iterative text improvement experiments take inspiration from the way some Wikipedia articles grow from a simple sentence into a fully fledged article as multiple people make small contributions [4]. My experiments so far start with a seed of text and ask turkers to improve it according to some instructions. After each attempted improvement, additional turkers vote whether the change is indeed an improvement. The winning text is fed back into the system for further improvement, until a stopping condition is met.

I have explored a number of iterative text improvement tasks, including copy editing, brainstorming, and converting an outline into prose. For reasons of space, I present only two examples here: image description and handwriting recognition.

Image Description

My first experiments involved writing descriptions for images. These experiments were inspired by Phetch [3], a game where humans write and validate image descriptions in order to make images on the web more accessible to people who are blind.

Figure 1 shows an example image from one of these experiments. Turkers were offered $0.02 to improve the description of this image. Voters were paid $0.01 to decide whether to keep each improvement. Selected iterations from a run of this experiment are shown below:

version 1:

> A parial view of a pocket calculator together with some coins and a pen.

version 2:

> ~~A view of personal items a calculator, and some gold and copper coins, and a round tip pen, these are all pocket and wallet sized item used for business, writting, calculating prices or solving math problems and purchasing items.~~

version 3:

> A close-up photograph of the following items:
>
> A CASIO multi-function calculator
> A ball point pen, uncapped
> Various coins, apparently European, both copper and gold
>
> Seems to be a theme illustration for a brochure or document cover treating finance, probably personal finance.

version 4:

> …Various British coins; two of £1 value, three of 20p value and one of 1p value. …

version 8:

> A close-up photograph of the following items:
>
> A CASIO multi-function, solar powered scientific calculator.
>
> A blue ball point pen with a blue rubber grip and the tip extended.
>
> Six British coins; two of £1 value, three of 20p value and one of 1p value.
>
> Seems to be a theme illustration for a brochure or document cover treating finance - probably personal finance.

Version 2 is struck out to indicate that it was voted down, so the turker who wrote version 3 did so as an improvement to version 1. Version 3 stood as a template that was incrementally improved, as seen in version 8.

It was often the case that early iterations would heavily influence the structure and tone of the final description. Most turkers would add a sentence at the end, or fix grammar mistakes in the text. Turkers were reluctant to remove a sentence entirely, though they would add information as seen in version 4.

Turkers frequently speculated about the scene, and often introduced external knowledge. In Figure 1, for example, one turker speculated that the image is for a finance brochure, and another added the information that the coins were British. Since I obtained the images from a public domain website, I was generally unable to confirm their speculations, but in at least one case, a turker correctly identified an image as being an "Iraq-Iran war memorial in Bagdad."

Evaluation

I hypothesize that paying turkers to iteratively improve an image description, given a certain budget for the whole process, should yield better results than paying a single worker the entire budget. To test this hypothesis, I ran an image-description experiment eleven times. In each run, an image was chosen randomly from a set of ten images (all taken from www.publicdomainpictures.net). A budget was also chosen randomly to be either $0.25 or $0.50. For each run, I ran one iterative improvement process using the budgeted amount of money, spending $0.02 per improve-HIT and $0.01 per vote-HIT. As a control, I also posted a single improve-HIT using the same image and offering the entire $0.25 or $0.50 budget as its reward. Every turker was only allowed to complete one HIT in the entire study, to ensure that the results were not biased by a particularly zealous turker.

After running each pair of experiments, I created a HIT asking turkers to compare the descriptions generated by each method for the given image. Twenty fresh turkers voted for each pair.

The votes favored the iterative method over the single-worker method 9 of 11 times (82%).

The rewards in these experiments are small (though typical of MTurk tasks), and it's not clear how the results might be affected by larger rewards. It's also worth comparing the iterative method against another crowdsourcing strategy: paying multiple turkers to write independent descriptions, and then using a vote to choose the best. These experiments are future work.

Handwriting Recognition

OCR software focuses on recognizing printed fonts. The reCAPTCHA project applies human computation to correct errors in OCR [2]. Recognizing handwriting is difficult for computers. It can even be difficult for humans. Many students receive feedback on papers that they cannot decipher.

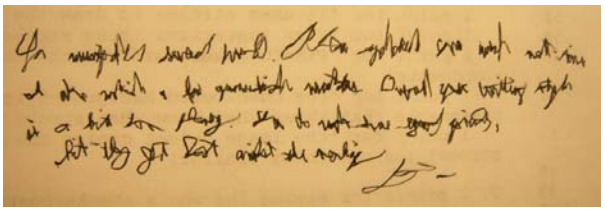

Figure 2: A sample of bad handwriting.

A common solution to this problem is to show the bit of handwriting to multiple people.

I wrote a passage with purposefully bad handwriting (Figure 2). Turkers were shown this image and offered $0.05 to make progress toward deciphering it. They were instructed to leave words they were unsure about in (parenthesis).

Selected iterations of this experiment are shown:

version 1:

> You (?) (?) (?) (work). (?) (?) (?) work (not) (time). I (?) (?) a few grammatical mistakes. Overall your writing style is a bit too (phoney). You do (?) have good (points), but they got lost amidst the (writing). (signature)

version 4:

> You (misspelled) (several) (words). (?) (?) (?) work next (time). I also notice a few grammatical mistakes. …

version 5:

> You (misspelled) (several) (words). (Plan?) (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)

version 6:

> 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)

The final version has only four mistakes shown in red. According to the ground truth, these words should be "flowery", "get", "verbiage" and "B-" respectively. Some other words are still left in parentheses. Workers made good use of parentheses, and it is interesting to see how the words in them change between iterations.

TOOLKIT

An overview of TurKit and related systems is shown in Figure 3. A programmer writes a set of JavaScript files which are executed by TurKit. TurKit stores information about the running program in the JavaScript database. TurKit also exposes an API to the JavaScript programs for communicating with MTurk and the JavaScript database.

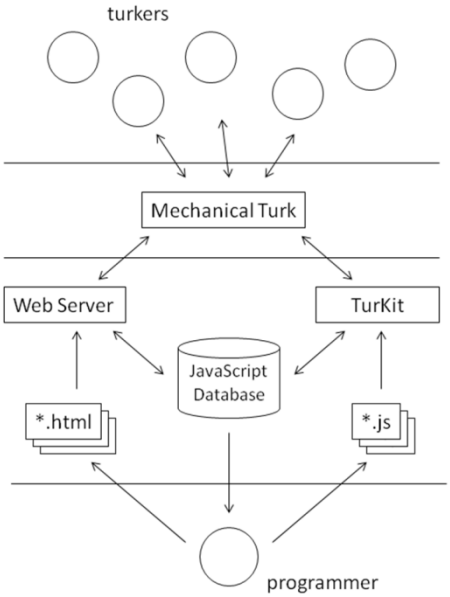

Figure 3: Architectural overview of TurKit and related systems. Arrows indicate the flow of information. The programmer controls the system by writing two sets of source code: HTML files for the web server, and JavaScript files executed by TurKit. Resulting output is retrieved via a JavaScript database.

One core feature of TurKit is a trace API for storing information about the trace of a program's execution, e.g., for memoizing the results of costly function calls. This trace can then be used if a program crashes in order to put the program back into its previous state.

Trace API

The trace API is implemented on top of the JavaScript database. It uses the database to store information about a program's trace of execution, so that when it is restarted, it can return to where it left off, without re-executing expensive code.

The API consists of the following functions:

once: The *once* function accepts a function as an argument, and guarantees that it will only be executed successfully once. If the function crashes, then it will be re-executed in a subsequent run of the program. If the function executes successfully, then a deep clone of the return value is memoized in the database, and returned to the caller. When the program is re-executed, the memoized result is returned without re-executing the function. The primary use of *once* is to guard expensive side effecting calls to MTurk so that they are only ever executed one time, even if the entire program is restarted.

stop: The *stop* function throws a "stop" exception. It is used when the program cannot continue without more information, presumably from MTurk. The program will be re-executed by TurKit after one minute, giving time for turkers to complete work on HITs. This polling interval is adjustable.

attempt: The *attempt* function takes a function as an argument. It executes this function inside a try/catch block which catches the "stop" exception. It returns true if the function executes normally and false if it stops prematurely.

The *attempt* function is conceptually similar to starting a new thread. The line of execution within the *attempt* function may be waiting on a different series of HITs than other lines of execution in the same program.

Note that *once* and *attempt* functions may be nested. However, it is important that all *once* and *attempt* functions at the same level of nesting are called in the same order when the program is re-executed. If this order cannot be guaranteed, then the programmer must supply a unique string identifier as a second parameter to all calls to *once* and *attempt* that may appear out of order within a nesting level.

get/setFrameValue: The *once* and *attempt* functions may be viewed as creating stack frames in the execution trace. Using this analogy, the *getFrameValue* and *setFrameValue* functions provide a way of storing and accessing values in the current frame.

resetTrace: The *resetTrace* function clears the trace history for the current program. It takes a version number as a parameter, and it remembers this number. If *resetTrace* is called with the same number when the program is re-executed, then it will do nothing. The programmer may increment this number to clear the trace history again.

Programming Paradigm

The idea of recording a trace of the program in order to re-execute it turned out to work really well for a couple of reasons. First, it allowed the program to be written in a straightforward procedural style. A previous version of the toolkit did not include the trace API. In order to write programs that were robust to system crashes, I needed to save the state manually. The simplest way to do this involved persisting a small set of variables, including a state variable. I then needed to unwrap the program into a state machine with a giant switch-statement on the state variable.

With the trace API, I discovered a second benefit. I could make many changes to the program while still being able to re-execute it, and this turned out to be useful. I could make changes that would print out status information that I had forgotten to print before, or instrument the code to gather new statistical data even on parts of the program that had already run.

The programming paradigm doesn't feel as complicated as multithreaded programming or parallel programming, but it is easy to make some mistakes. A common mistake is putting code inside a call to *once* that shouldn't be there. In one case, I decremented a *moneyRemaining* variable inside a call to *once*. Unfortunately, the *moneyRemaining* variable was re-initialized at the beginning of each run of the program. Fortunately, for safety, TurKit enforces a maximum limit on money spent per day (this is adjustable).

The *attempt* function also requires some multi-threaded thinking when implementing algorithms like a parallel sort. My implementation of parallel sort also prompted the necessity of the *getFrameValue* and *setFrameValue* functions; more discussion of these functions and their role in TurKit has been omitted for space.

CONCLUSION AND FUTURE WORK

I have described TurKit, a new toolkit for programming iterative tasks on MTurk using a familiar imperative programming model, and applied it to a variety of example tasks. For future work, I plan to explore more complicated algorithms using TurKit, such as a parallel iterative text improvement algorithm which uses multiple tracks of iterative improvement, which are then combined. Also valuable to users of TurKit would be a detailed study of MTurk's properties as a programming system - latency, error rate, turker expertise, etc.

ACKNOWLEDGMENTS

I would like to thank everyone who contributed suggestions and ideas to this work, including Thomas W. Malone, Robert Laubacher, and members of the UID group. 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 the MIT Center for Collective Intelligence. 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.

REFERENCES

1. Luis von Ahn. Games With A Purpose. IEEE Computer Magazine, June 2006. Pages 96-98.
2. Luis von Ahn, Ben Maurer, Colin McMillen, David Abraham and Manuel Blum. reCAPTCHA: Human-Based Character Recognition via Web Security Measures. Science, September 12, 2008. pp 1465-1468.
3. Luis von Ahn, Shiry Ginosar, Mihir Kedia and Manuel Blum. Improving Accessibility of the Web with a Computer Game. ACM Conference on Human Factors in Computing Systems, CHI Notes 2006. pp 79-82.
4. Kittur, A. and Kraut, R. E. 2008. Harnessing the wisdom of crowds in wikipedia: quality through coordination. CSCW '08. ACM, New York, NY, 37-46