TurKit: Tools for Iterative Tasks on Mechanical Turk

Greg Little, Lydia B. Chilton, Rob Miller, Max Goldman
MIT CSAIL
32 Vassar St
Cambridge, MA 02139 USA
{glittle,hmslydia,rcm,maxg}@mit.edu

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. This paper explores an alternative *iterative* paradigm, in which workers build on or evaluate each other's work. We 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. We present a variety of iterative experiments using TurKit, including image description, copy editing, handwriting recognition, and sorting.

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

General terms: Algorithms, Design, Economics, Experimentation

Keywords: Human computation, Mechanical Turk, toolkit

INTRODUCTION

Human effort is a vital part of many systems. Computers are still less effective than humans at many tasks, such as natural language processing and image recognition, and humans are clearly more suited to subjective tasks involving opinions and tastes. Human effort can be difficult to exploit, but systems like Wikipedia, Yahoo Answers and MTurk are making it possible to outsource human computation tasks over the internet.

Wikipedia, Yahoo Answers and the ESP Game [2] are systems where users volunteer their human computation because they value helping others, participating in a community, or playing a game. But these systems aren't designed for *arbitrary* human computation tasks. Whereas Yahoo Answers is a place where you can easily get romantic advice, it's harder to get people to correct the spelling in a document or look up 100 restaurant addresses on the web.

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. MTurk has already been leveraged by industry and academia for image labeling, verifying addresses, and tagging documents. MTurk will often get work done faster and can get work done that people seem not to want to do for free.

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. To deal with malicious, lazy, or erring turkers, requesters generally post multiple instances of each task.

This paper considers a different model for employing turkers: **iterative work**, 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. Iteration is a fundamental computing concept, which applies equally well to human computation. It is also a fundamental concept in design and engineering that leads to higher-quality results. Other human computation systems also use iteration, notably Wikipedia, which is built by humans iterating on each other's work. As far as we are aware, however, this paper presents the first examples in the literature of iterative tasks on a for-pay human computation system like MTurk.

The second contribution of this paper is TurKit, a toolkit for programming iterative processes on MTurk. The TurKit API contains functions that help write iterative MTurk tasks. The main challenge is making the entire system fault tolerant, so that bugs and system crashes do not lead to wasted money or time on MTurk. TurKit overcomes this challenge while maintaining a straightforward procedural programming model. In this model, HIT generation is a simple function call that posts the HIT only once and automatically stores (*memoizes*) its result in a database. As a result, a TurKit program is *idempotent*, able to be run repeatedly without reposting previously completed work.

In this paper we present several examples of iterative tasks done on MTurk using TurKit. The first set of examples concern *iterative text improvement*: describing an image, turning an outline into prose, editing for specific stylistic changes, brainstorming, and handwriting recognition. The second set concern *sorting*, using turkers as comparison functions in a sort algorithm. Finally, we present the TurKit API and programming paradigm, and discuss some lessons about using MTurk as a programming platform.

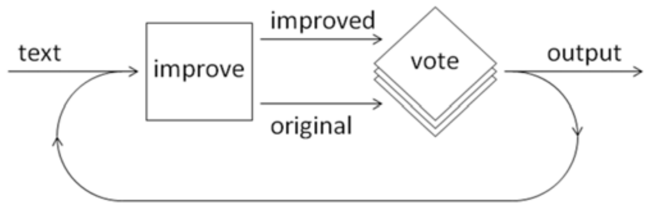
RELATED WORK

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. Bryant [7] makes observations about how people begin contributing to Wikipedia, and what tools expert contributors use to manage and coordinate their work.

MTurk provides a platform for performing Human Intelligence Tasks (HITs) where humans are motivated by money. This platform has been adopted for a variety of uses, both in industry and academia. Kittur [8] discusses how to run user studies on MTurk, while Sorokin [10] uses MTurk to label images. Thus far, the typical usage pattern for MTurk involves generating all the HITs that need to be completed, posting them to MTurk, and later downloading all the results. Several websites focus on managing HITs that fit this template (e.g. HIT-builder[1]). It is currently rare, however, to automatically generate new HITs based on the results of previous HITs. Sorokin proposes creating voting HITs in response to labeling HITs, but does not report any experiments using this technique.

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 [9]. In our experiment, we start with a seed of text and ask MTurk workers (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. Here is a schematic overview of the iterative text improvement process:



The boxes labeled “improve” and “vote” represent Human Intelligence Tasks (HITs) on MTurk. The improve-HIT simply asks a turker to improve a given body of text according to some instructions. Turkers are generally given one hour from the time they accept an improve-HIT before they must submit the result, or release it so that another turker may attempt it.

Voting is meant to prevent turkers from making detrimental changes to the text. The vote-HIT asks a turker which of two bodies of text is better according to a given criteria. Voters are shown the original text and the improved text in random order, without any explicit indication as to which was the original. The differences between the two texts are highlighted in yellow, e.g.:

a) The quick fox leaped over the dogs.
b) The quick brown fox jumped over the lazy dogs.

[1] http://www.hit-builder.com/

The turker who wrote the text being voted on is not allowed to vote. No turker is allowed to vote twice between the same two texts. Voting HITs must be completed within one hour of accepting the HIT.

The system requests two votes for each improvement. If the votes disagree, then a third vote is requested. The winning text after each voting stage is fed back into an improvement HIT. This process is repeated until a predefined amount of money is spent. We spent $0.25-$1 total for each experiment described below, generally paying $0.02-$0.10 for each improve-HIT and $0.01 for each vote-HIT.

Image Description

Our 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. In our experiments, we pay people to write the descriptions and improve on previous descriptions.

Figure 1 shows an example result from one of these experiments. Turkers were offered $0.02 for the improve-HIT, which had the following instructions:

- Please improve the description for this image.
- People will vote whether to approve your changes.
- Use no more than 500 characters.

Note that the first improve-HIT had the instructions “Please describe this image.”, since we didn’t have a starting description for them. The character limit was enforced using JavaScript, and turkers were shown a count indicating their current character usage.

The vote-HIT had these instructions:

- Please select the better description for this image.
- Your vote must agree with the majority to be approved.
- Differences are highlighted in yellow.

If there was only one description, voters were asked whether the description was “good” or “not great”.

The results from the first four improve-HITs for 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 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 4. Figure 1 shows the final version after 8 iterations.

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 grammer 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 we obtained the images from a public domain website, we were 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.”

We 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 the hypothesis, we 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 and were selected by the author for being scenes with multiple identifiable objects in them. We also chose a budget to spend, which was either \$0.25 or \$0.50, chosen randomly for each run. For each run, we 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, we 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, we 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%).

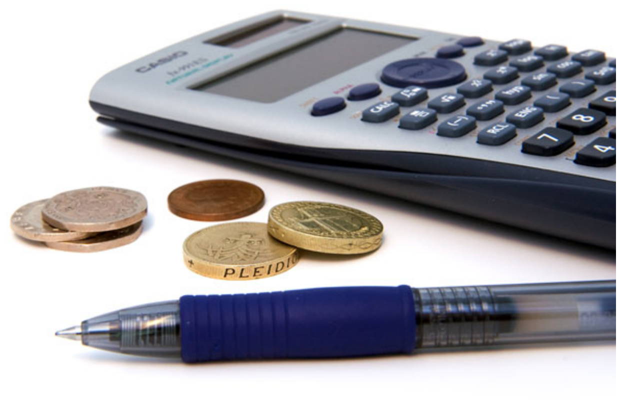


Figure 1: MTurk workers describe this image after eight iterations as:

“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."

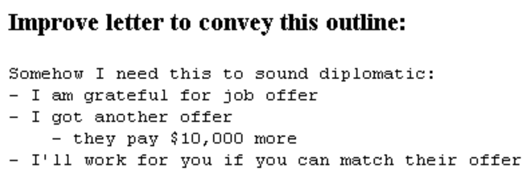
The rewards in these experiments are small (though typical of MTurk tasks), and it’s not clear how the results might be affected by larger rewards. It’s also worth comparing the iterative method against another crowdsourcing strategy: paying multiple turkers to write *independent* descriptions, and then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

Writing Tasks

Because the image description experiments indicated that turkers are reliable at writing tasks, our next three experiments relied more heavily on writing ability. The first task was to turn an outline of a document into text, the second was to change a passage from past to present tense and its first person pronouns from singular to plural, and the third was to convert a passage from passive to active voice and edit for style.

Outline to Prose

The first task involved writing a letter negotiating a job offer. Turkers were shown the following instructions:



Improve letter to convey this outline:

```
Somehow I need this to sound diplomatic:
- I am grateful for job offer
- I got another offer
    - they pay $10,000 more
- I'll work for you if you can match their offer
```

Voters were shown the same outline with the prompt: "Which letter is closer to conveying this outline:".

Turkers were paid $0.02 for the improve-HIT as before, but we did not hinge their approval on the voting process. Likewise, we did not require that voters be in the majority for approval, but we did require that they have a 90% approval rating on other MTurk tasks before voting. Selected outcomes of the improve-HIT are shown below:

version 1:

> I want to thank you for your gracious job offer. However, I have had another job offer willing to pay me $10,000 more for the same job. But, I am willing to forgoe that job offer because working with your company would be a great opportunity, that is, if you would like to match their offer.

version 3:

> I would like to thank you very much for your offer, however, I have received a better package from another firm. I still would like to work for company and would like to speak with you to see if there is anyway we could make that occur.

version 9:

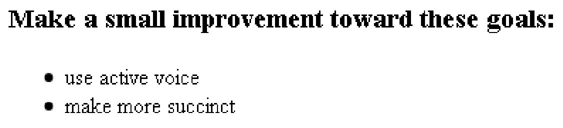
> Thank you very much for your offer. At this time, I have a competing offer on the table at a higher salary. If the remuneration packages were equal I would much prefer to work for yourselves, and I am open to discussing the possibility of your matching this offer. If this is possible, please contact me at your earliest convenience.

The initial improve-HIT made a first pass at turning the entire outline into prose. Subsequent improve-HITs mostly refine the language.

It's interesting that turkers voted to reject both the first and second version (not shown), which mentioned "$10,000" explicitly. It is also interesting to observe how the description of the other firm's offer progresses from "$10,000 more" to "better package" to "far more competitive offer" to "more competitive offer" to simply "competing offer". We didn't ask for reasons for particular edit could be that the turkers were not attending to the outline, but it's possible that turkers thought mentioning the dollar amount was not professional.

Active Voice

In many cases, stylistic changes can be made to a document by an editor without external knowledge about the content. For this task we focused on encouraging a succinct writing style and converting passive voice to active voice – for instance, using the active "We made mistakes" instead of the passive "Mistakes were made". We presented workers with the following instructions:



Make a small improvement toward these goals:

- use active voice
- make more succinct

The instructions were similar for voters, except the prompt was: "Select the paragraph which steps toward these goals:"

Selected iterations are shown below, starting with version 0, which was the original passage.

version 0:

> The earthquake was positively and utterly devastating to San Francisco. The worst things were that homes were destroyed, fortunes were lost, people were generally destitute. And by all modern standards the economy went into shock as the cost of simple home items such as bread and salt and nails tripled over night. Rebuilding the city took at least a decade. Repairs to buildings, roads and infrastructure such as plumbing were urgent and many. After all was said and done, the lesson in all this was to rebuild the city with earthquakes in mind.

version 5:

> The earthquake utterly devastated San Francisco. It destroyed homes and left people destitute. By modern standards, it forced the economy into shock, as the cost of everyday items tripled overnight. It took at least a decade to rebuild the city to withstand future earthquakes, most importantly buildings, roads and infrastructure.

version 8:

> The earthquake devastated San Francisco. It destroyed homes and left people destitute. It forced the economy into shock, tripling the cost of everyday items overnight. It took a decade to rebuild the city to withstand future earthquakes, especially buildings, roads and infrastructure.

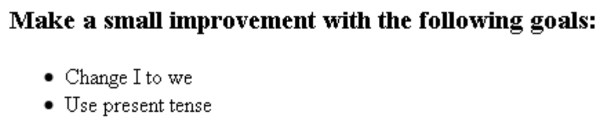
By the fifth version, the three instances of passive voice ("homes were destroyed", "fortunes were lost", "repairs were urgent") were replaced with active voice or eliminated. Further iterations removed many superfluous words and phrases.

We ran several experiments involving making a passage more succinct. In some of these trials, turkers would alter the meaning of the passage, a common risk of copy editing. Voters did not know which passage was the original, and so they could not know for sure if the meaning had changed. They could only vote on which adhered more to the criteria. In future trials, we will try keeping the original passage present in both the improve- and vote-HITs.

Changing Grammatical Tense

Occasionally a passage is written in past tense, and needs to be changed to present tense to match the style of another document. This is another stylistic change that can largely be made without external knowledge. In this experiment, we ask turkers to change a passage from past tense to present tense, and use plural rather than singular pronouns.

We presented workers with the following instructions:


Make a small improvement with the following goals:

- Change I to we
- Use present tense

The instructions were similar for voters, except the prompt was: “Select the paragraph stepping toward the goals:” Here are selected iterations of the passage, starting with version 0, which was the original passage.

version 0:

> I took my bag and proceeded to climb the mountain. I thought to myself that it would rain later. I was worried, but I tried not to let it discourage me. I figured that if it did rain, I could always build a shelter of some sort using rope. Of course, after climbing for a few hours, I discovered that I had forgotten any rope. I'm not sure how I was able to make this mistake. Fortunately, it didn't rain.

version 1:

> We take our bags and proceed to climb the mountain. We think to ourselves that it will rain later. We are worried, but we try not to let it discourage us. We figure that if it did rain, ee could always build a shelter of some sort using rope. Of course, after climbing for a few hours, we discover that we have forgotten any rope. We're not sure how we were able to make this mistake. Fortunately, it doesn't rain.

version 3:

> … We figure that if it did rain, we can always build a shelter of some sort using rope. …

version 4:

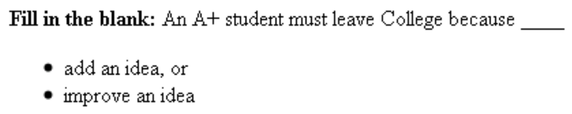
> … We figure that if it does rain, we can always build a shelter of some sort using rope. …

Although we asked workers to make small improvements, observe that the first turker made practically all of the necessary changes. The benefits of iteration may be seen in two subsequent iterations which correct a typo and change two remaining past-tense verbs into present tense.

It is worth noting that versions 2, 5 and 6 were all rejected for the same mistake: the turker converted all “we”’s back into “I”’s. We speculate that because most of the work was done in version 1, subsequent workers were confused about what to do. Fortunately the voters were not confused.

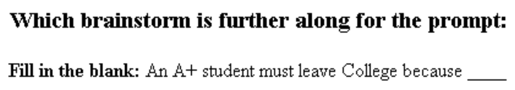
Brainstorming

Brainstorming is a technique for generating ideas. We ran several experiments using the iterative text improvement technique for brainstorming. We chose a topic that required no expert knowledge. In a task titled “Brainstorm Plot Ideas”, we provided the instruction:


Fill in the blank: An A+ student must leave College because ____

- add an idea, or
- improve an idea

The voting task had the instructions:


Which brainstorm is further along for the prompt:

Fill in the blank: An A+ student must leave College because ____

Here is version 6 of the brainstorm:

version 6:

> He has to go into a witness protection program.
>
> He is being investigated for photographing nude girls.
>
> He has been accepted for NASA training.
>
> His parents died in a car accident, and he had to take over the family business.
>
> She was bitten by, and turned into, a vampire.
>
> The new dean is the mother of the boy whose heart was broken (and not nicely) by the student.

The formatting is interesting—turkers followed the convention selected by the second turker of leaving a blank line between each idea. In this and two other brainstorming trials, turkers were reluctant to modify existing ideas. Most people added a single idea at the end. A few turkers expanded ideas already present, usually by adding text to the end of them. For instance, a subsequent version of the brainstorm above takes the idea “He has to go into a witness protection program” and adds the text “and finds out that everyone in his new neighborhood also left college for the same reason.”

Handwriting Recognition

Most OCR software focuses on recognizing printed fonts. The reCAPTCHA project applies human computation to correct errors in OCR [4]. Software for handwriting recognition typically requires access to the sequence of pen strokes that form characters, as supplied by a device like a digital tablet. Recognizing handwriting is difficult for computers. It can even be difficult for humans. Many students receive feedback on papers that they cannot decipher. A common solution to this problem is to show the bit of text to multiple people.

We wrote a passage with purposefully bad handwriting and in cursive, see Figure 2. Turkers were shown this image, and offered $0.05 to follow these instructions:

Make progress toward deciphering this handwriting.

- put words you are unsure about in (parenthesis)

Voters also saw the image, and had these instructions:

Which passage is closer to deciphering this handwriting?

- NOTE: unclear words may be marked with (parenthesis)

All iterations of a single run 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 2:

~~You (?) (?) (saved) (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 …

The final version is shown in its entirety in Figure 2. As noted in the figure, only four words were deciphered incorrectly, though some are still in parentheses. Workers did make good use of the parentheses, and it is interesting to see how the words in them change between iterations.

Performance

Figure 3 shows time and cost statistics for the text improvement tasks presented in this paper (except for the image description tasks, for which this data was not collected). Latency is the time that elapsed between the program posting a HIT and receiving a response. Latency values are actually overestimates because TurKit only polled every 5 minutes in these experiments. One improvement HIT generally required 15-30 minutes to complete, while voting (the total latency for all vote HITs needed by the iteration) typically took 7-15 minutes. Our HIT tasks also measured *turker time*, the time actually spent by turkers doing the task, using JavaScript embedded in the task page. For improvement HITs, the time on task was average less than 5 minutes per HIT, while each vote HIT took half a minute or less. Cost is the amount spent per iteration, divided between the improve HIT (which is fixed) and all the voting HITs (which is an average cost over all iterations, because many iterations required only 2 votes to choose the best out of 3). All our experiments cost less than $1 each.

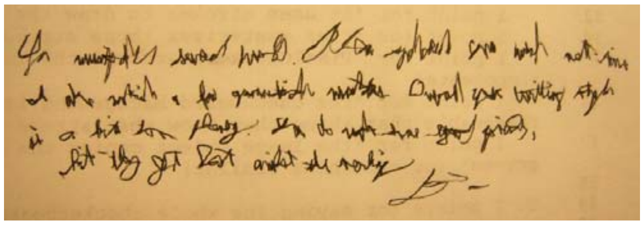

[illegible]

Figure 2: MTurk workers build on each other's work interpreting this handwriting. Turkers were instructed to put unknown words in parentheses. The result after six iterations is:

"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)"

According to our ground truth, the highlighted words should be "flowery", "get", "verbiage" and "B-" respectively.

		latency (min) per iteration		turker time (min) per HIT		cost per iteration		
	iterations	improve	all votes	imrpove	single vote	improve	all votes	total cost
outline to prose	10	23.98	59.64	4.15	0.43	$0.05	$0.027	$0.38
active voice	13	37.47	7.77	5.22	0.23	$0.05	$0.027	$0.39
grammatical tense	7	9.45	16.38	1.56	0.35	$0.02	$0.022	$0.18
handwriting	9	21.20	14.57	3.30	0.38	$0.05	$0.023	$0.46
brainstorming	24	13.34	7.81	1.37	0.32	$0.02	$0.024	$0.88

Figure 3: Average time and cost for text improvement experiments.

SORTING

In order to explore the extent to which our toolkit is capable of general purpose computation using MTurk, we tried sorting, using turkers to implement the comparison function. Sorting is not only a fundamental computational process, but also inherently iterative. An efficient comparison-based sort cannot know in advance all the comparisons it will need to make.

Algorithm

We model MTurk as a multiprocessor environment with many human-processors. We assume zero cost for communication between processors, since that is handled by a computer, much faster than any human-processor.

Given enough processors (and the right constant-time operations), sorting can be accomplished in $O(\log n)$ wall-clock time [6]. We use such an algorithm for our experiments. The algorithm is based on quicksort. The algorithm first chooses a pivot. Then it compares all of the elements to the pivot in parallel. This yields two lists, and the algorithm recursively applies the same technique to each list. If the pivots are well chosen, then the algorithm can only apply this recursion log n times before it is down to lists of just one element, hence the $O(\log n)$ running time as measured on the clock. Note that the algorithm actually requires $O(n \log n)$ person-hours of work on MTurk, since that many comparisons are required by the sort. This algorithm is encapsulated in the *sort* function of TurKit.

Experiments

We ran several experiments exploring the potential uses of sorting using this algorithm.

We ran three tasks requiring subjective sorting:

- Sort 20 personal travel pictures based on how good they were to show others
- Sort 10 t-shirt designs based on personal taste
- Sort 6 images of coats based on how well they would suit a particular person (a photo of the person was included)

Interestingly, the t-shirt design was also voted on by the population that would be wearing them, serving as somewhat of a comparison. The eventual wearers' second choice was the turkers' first choice. Notably, the eventual wearers' first choice was an inside joke which turkers ranked near the bottom which verified that the joke was not mainstream.

Discussion

This algorithm's running time is optimal if each comparison is performed by a separate turker, but it may have drawbacks if a single turker performs many comparisons. Consider that the initial set of comparisons are all against a single pivot, so a MTurk worker doing these comparisons will see the same item repeated over and over. This may be tedious, and the worker's impression of the repeated item may change over time, which is bad for comparisons involving subjective criteria. We plan to explore these implications more in future work.

This algorithm may not be optimal given the properties of the items we are attempting to sort. However, the toolkit itself seems capable of handling more complex algorithms, which we shall explore in future work.

TOOLKIT

An overview of TurKit and related systems is shown in Figure 4. A programmer writes a set of JavaScript files which are executed by TurKit. TurKit stores information about the running program in the JavaScript database, so that it can restart if the system crashes.

TurKit also creates Human Computation Tasks (HITs) on MTurk. When turkers view these HITs, they see an iFrame pointing to a web page. The programmer must create the web pages used for this purpose. These web pages may access the database before being displayed to turkers.

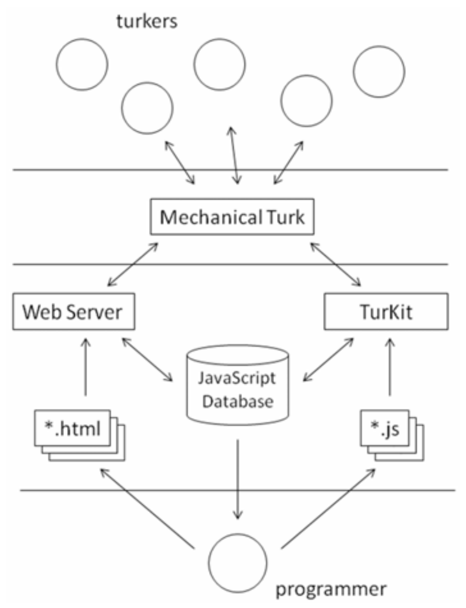


Figure 4: Architectural overview of TurKit and related systems. Arrows indicate the flow of information. The programmer controls the system by writing two sets of source code: HTML files for the web server, and JavaScript files executed by TurKit. Resulting output is retrieved via a JavaScript database.

When turkers complete tasks, it is possible for the web server to store the results directly in the database, or pass the results back to MTurk. In the latter case, the program running in TurKit can retrieve the results from MTurk and store them in the database.

The programmer retrieves results directly from the JavaScript database.

TurKit

TurKit is a Java program that executes JavaScript files and provides them with an API. The API includes functionality for communicating with MTurk, and the JavaScript database.

One core feature of TurKit is a trace API for storing information about the trace of a program's execution. This trace can then be used if a program crashes in order to put the program back into its previous state, without re-executing all the code. In particular, it does not re-execute code with side effects, like creating HITs.

TurKit also includes a utility API for executing common higher level tasks, like voting and sorting.

MTurk API

MTurk provides an API for a number of languages, including Java. TurKit provides a JavaScript wrapper for this API, and converts common data types into native JavaScript objects. This includes converting some XML data fields into JavaScript objects, for easier access.

The JavaScript wrapper adds additional fault tolerance to the Java API. This is necessary because MTurk requests will fail if they are made too quickly in succession. The JavaScript wrapper will retry a request up to ten times, sleeping for increasing intervals between each call.

All JavaScript wrappers for MTurk functions with side effects, like *createHit*, *approveAnswer* and *deleteHit*, wrap these calls inside a call to *once*. The *once* function ensures that the side effect only happens once, even if the program is re-executed. This is discussed more in the trace API section below.

Trace API

The trace API is implemented on top of the JavaScript database API. 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 three 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 when the entire program is restarted.

crash: The *crash* function throws a "crash" exception. The purpose of *crash* is to simulate crashing the program. It is used when the program cannot continue without more information, presumably from MTurk. The program will be re-executed by TurKit after five minutes, 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 "crash" exception. It returns true if the function executes normally and false if it crashes.

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.

resetTrace: The *resetTrace* function clears the trace history for the current program. It accepts a number as an optional parameter. If this number is supplied, then *resetTrace* still clears the trace history, but 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.

Utility API

TurKit provides several utility functions to cover common higher level MTurk tasks.

waitForHit: The *waitForHit* function accepts a HIT ID and returns a HIT JavaScript object complete with an array of answer objects representing the results returned from Mechancial Turk. If the HIT is not yet complete, this function throws the "crash" exception.

vote: The *vote* function manages a HIT where turkers vote between two or more options. Consider the simple case of a best-of-three vote between two options. In this case, the system only needs two votes, if they are both the same. It only needs to request a third vote if the first two votes disagree. The vote function handles this by extending a HIT to request more votes until enough votes agree with each other.

The vote function takes three arguments: a HIT ID for a vote-HIT; a function that extracts the turker's choice from the HIT's answer object; and the number of votes necessary for a choice to win and terminate the voting. The vote function returns an object representing the results of the vote, including the fields: *bestOption*, *totalVoteCount*, and *voteCounts*, which is itself an object storing the count for each voting option.

sort: The *sort* function takes two parameters: an array, and a comparator function. The comparator function is expected to accept two arguments a and b, and return -1, 0, or 1 depending on whether $a < b$, $a = b$, or $a > b$, respectively. Typically the comparator will use the vote function to compare the items on MTurk.

Note that JavaScript's own *Array.sort* function has the same parameters, and achieves the same result. However, the TurKit sort function uses a parallel sorting algorithm, and uses the *attempt* function internally to simulate parallel threads of comparisons on MTurk.

Example

Figure 5 shows an iterative text improvement task in TurKit. This code relies on two web pages running on a local web server, namely *improve.html*, and *verify.html*. When the program creates a HIT on line 8, it includes a URL to *improve.html*. The *text* value is encoded as a URL parameter appended to this URL. The *improve.html* page will display this value to turkers dynamically using client-side JavaScript. The value *newText* on line 13 comes from a POST parameter supplied as part of the form submitted by *improve.html* to MTurk. These values are encoded as XML by MTurk, but the TurKit API converts them to JavaScript for easy access.

Similarly, the URL supplied to *createHit* on line 16 encodes both *text* and *newText* as parameters. The form submitted by *verify.html* includes *voteForNewText* as a POST parameter, which is retrieved on line 22.

The first time this program is executed, it will create a HIT on line 8, and return the HIT Id. Then it will wait for this HIT on line 11, which will trigger a "crash" exception. When the program is re-executed, the *createHit* function on line 8 will return the memoized HIT Id from the previous run of the program. At this time, line 11 might return a value if turkers have completed the HIT already. When the program finally gets to the second iteration of the while loop, it will call *createHit* a second time. This will result in the creation of a new HIT Id, which will be memoized for future runs of the program. Each run of the program will get further and further until the *money* value reaches zero.

```
 1 var text = ""
 2 var money = 0.5
 3 var improveCost = 0.02
 4 var voteCost = 0.01
 5
 6 while (money > 0) {
 7     // improve text
 8     var hitId = createHit(...
 9         "improve.html"
10         ...)
11     var hit = waitForHit(hitId)
12     var answer = hit.answer[0]
13     var newText = answer.newText
14
15     // verify improvement
16     var voteHitId = createHit(...
17         "verify.html"
18         ...)
19     var voteResults = vote(
20         voteHitId,
21         function (answer) {
22             return answer.voteForNewText
23         },
24         2)
25     money -= voteResults.totalVoteCount *
26         voteCost
27
28     if (voteResults.bestOption == "yes") {
29         text = newText
30         acceptAnswer(answer)
31         money -= improveCost
32     } else {
33         rejectAnswer(answer)
34     }
35 }
```

Figure 5: Example TurKit program for iterative text improvement.

JavaScript Database

The JavaScript database is a replacement for a standard SQL relational database. The JavaScript database persists a JavaScript environment on disk. Functions and non-JavaScript objects are not persisted.

Queries into this database are simply strings of JavaScript code that are executed in the environment of the database, potentially having side effects on it. Query results are formatted as strings of JSON.

Since TurKit programs are written in JavaScript, and handle large amounts of data stored in JavaScript objects, it is convenient to be able to transfer these objects to and from the database without any conversion, other than to and from JSON. We omit details of our own implementation, since other such databases exist, including the storage mechanism in AppJet[2].

To make it easier to inspect the contents of the JavaScript database, we provide a simple UI. This UI is conceptually similar to phpMyAdmin for MySQL. The UI consists of an expandable tree view implemented in a web page. The tree view limits the number of results shown when a node is expanded, with options for showing more items. Having such an interface has proved useful in the process of debugging and maintaining TurKit programs, even while they are running.

DISCUSSION

Over the course of our experimentation and system building, we have learned a number of lessons.

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, we needed to save the state manually. The simplest way to do this involved persisting a small set of variables, including a state variable. We then needed to unwrap our program into a state machine with a giant switch-statement on the state variable.

After we had the trace API, we discovered a second benefit. We could make many changes to the program while still being able to re-execute it, and this turned out to be useful. We could make changes that would push more information into the database, that we had forgotten to store before, or instrument the code to gather new statistical data even on parts of the program that have 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, we decremented a money variable inside a call to once. Unfortunately, the money 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.

The attempt function also requires some multi-threaded thinking when implementing algorithms like a parallel sort. We have also discovered cases where we need to store extra state information that can't be represented with once and attempt. In a sense, these functions store a call stack, and what we need is space in this stack for local variables.

[2] AppJet: Instant Web Programming. http://appjet.com/

Development Cycle

Writing a new TurKit program often involves writing the accompanying web pages, and it is important to test the web pages in the MTurk sandbox before using them live. This testing is necessary to ensure that the web pages interact properly with MTurk. Most importantly, they need to POST the correct information to MTurk when a worker submits their response.

When possible, we have found it useful to create generic web pages that can adapt to different purposes with URL parameters. All the iterative text improvement tasks use the same web pages, where most of the content shown to users is encoded as part of the URL. These web pages contain code responsible for interacting with MTurk which we only needed to test once.

Once confident that a set of web pages work, it is good to test a new TurKit program by only allowing it to execute a little further each time. This can be achieved by calling crash right after each important action, like creating a HIT. The crash calls are removed after the programmer discovers that the program is behaving correctly up to that point.

After a TurKit program seems to be working, the challenge switches to debugging the user interfaces presented to turkers in order to guide turkers to provide the appropriate responses. The primary delay in this test cycle comes from waiting for turkers to complete HITs. Voting HITs may be completed in 10 minutes. Improvement HITs can take up to an hour, depending on the task involved, and the amount of money offered. A useful direction for future work would involve getting hard statistics on these times, but such research may be premature while the number of turkers is still growing.

Turkers

Our initial experiments assumed that MTurk workers would be trying to game the system. The whole idea of voting was to ensure that workers didn't get away with adding garbage to an image description, without our having to approve every improvement manually. To discourage turkers from cheating in the voting task, we required that their vote match the majority in order to get paid.

However, our experience is that turkers are generally not trying to game the system, but it is still a good idea to use voting. This is because of the high variance of responses to improvement tasks. The typical method of dealing with variance is to have multiple people perform the same task, and pick the result that is most common. The results of improvement tasks are likely to all be different, but voting helps simulate the effect by having multiple people agree that the work was an improvement.

Expertise

In our experiments, we have come to believe that many turkers are reliable writers. However, not all turkers are experts at everything, and there is currently no good way to route tasks to turkers according to required expertise. Interestingly, some turkers do know how to program, and we are currently exploring the use of this talent for iteratively creating programs. However, these experiments have run into more problems than the tasks involving writing.

CONCLUSION AND FUTURE WORK

We 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, we plan to explore more complicated algorithms using TurKit, such as a parallel sort algorithm that is more robust to human comparison functions that may be noisy or only partially ordered. 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

We would like to thank everyone who contributed suggestions and ideas to this work, including Thomas W. Malone, Robert Laubacher, and member 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 and Laura Dabbish. Labeling Images with a Computer Game. ACM Conference on Human Factors in Computing Systems, CHI 2004. Pages 319-326.
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. 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.
5. AppJet: Instant Web Programming. http://appjet.com/
6. Heidelberger, P., Norton, A., and Robinson, J. T. 1990. Parallel Quicksort Using Fetch-And-Add. IEEE Trans. Comput. 39, 1 (Jan. 1990), 133-138.
7. Susan L. Bryant, et al. Becoming Wikipedian: transformation of participation in a collaborative online encyclopedia. GROUP 2005.
8. Kittur, A., Chi, E. H., and Suh, B. 2008. Crowdsourcing user studies with MTurk. CHI 2008.
9. 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
10. Sorokin, A. and D. Forsyth, "Utility data annotation with Amazon MTurk," Computer Vision and Pattern Recognition Workshops, Jan 2008.