MIT Open Access Articles

TurKit: Human Computation Algorithms on Mechanical Turk

The MIT Faculty has made this article openly available. ***Please share*** how this access benefits you. Your story matters.

Citation	Greg Little, Lydia B. Chilton, Max Goldman, and Robert C. Miller. 2010. TurKit: human computation algorithms on mechanical turk. In Proceedings of the 23nd annual ACM symposium on User interface software and technology (UIST '10). ACM, New York, NY, USA, 57-66. Copyright 2010 ACM
As Published	http://dx.doi.org/10.1145/1866029.1866040
Publisher	Association for Computing Machinery
Version	Final published version
Citable link	http://hdl.handle.net/1721.1/60950
Terms of Use	Attribution-Noncommercial-Share Alike 3.0
Detailed Terms	http://creativecommons.org/licenses/by-nc-sa/3.0/

Massachusetts Institute of Technology

DSpace@MIT

TurKit: Human Computation Algorithms on Mechanical Turk

Greg Little[1], *Lydia B. Chilton*[2], *Max Goldman*[1], *Robert C. Miller*[1]

[1]MIT CSAIL
{glittle, maxg, rcm}@mit.edu

[2]University of Washington
hmslydia@cs.washington.edu

ABSTRACT

Mechanical Turk provides an on-demand source of human computation. This provides a tremendous opportunity to explore algorithms which incorporate human computation as a function call. However, various systems challenges make this difficult in practice, and most uses of Mechanical Turk post large numbers of independent tasks. TurKit is a toolkit for prototyping and exploring truly algorithmic human computation, while maintaining a straight-forward imperative programming style. We present the crash-and-rerun programming model that makes TurKit possible, along with a variety of applications for human computation algorithms. We also present a couple case studies of TurKit used for real experiments outside our lab.

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

General terms: Algorithms, Design, Experimentation

Keywords: Human computation, Mechanical Turk, toolkit

INTRODUCTION

Amazon's Mechanical Turk (MTurk) is a 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 used by industry and academia for labeling images, categorizing products, and tagging documents.

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. We want to explore tasks that build on each other. Figure 1 shows a simple example of an algorithm that generates a list of suggestions for what to see in New York, and then sorts them. A more sophisticated example might have turkers iteratively improve a passage of text, and vote on each other's work. In general, this paper considers *human computation algorithms*, where an algorithm coordinates the contributions of humans toward some goal.

```
ideas = []
for (var i = 0; i < 5; i++) {
    idea = mturk.prompt(
        "What's fun to see in New York City?
         Ideas so far: " + ideas.join(", "))
    ideas.push(idea)
}

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

Figure 1: Naturally, a programmer wants to write an algorithm to help them visit New York City. TurKit lets them use Mechanical Turk as a function call to generate ideas and compare them.

Human computation and Mechanical Turk are already being explored and studied in the HCI community [7] [8] [9]. We want to extend this study to explore *algorithms* involving humans, which is an HCI issue in itself. It requires knowing the right interface to present to each turker, as well as the right information for the algorithm to pass from one turker to the next.

Unfortunately, implementing algorithms on MTurk is not easy. HITs cost money to create, and may take hours to complete. Algorithms involving many HITs may run for days. These factors present a significant systems building challenge to programmers. Programmers must worry about issues like: what if the machine running the program crashes? What if the program throws an exception after a bunch of HITs have already been completed? These challenges are prohibitive enough to prevent easy prototyping and exploration of human computation algorithms.

This paper introduces the crash-and-rerun programming model to overcome these systems challenges. In this model, a program can be executed many times, without repeating costly work.

TurKit is a toolkit for writing human computation algorithms using the crash-and-rerun model. TurKit allows the programmer to think about algorithmic tasks as simple straight-line imperative programs, where calls to MTurk appear as ordinary function calls.

This paper makes the following contributions:

- *Crash-and-Rerun Programming*: A novel programming model suited to long running processes where local computation is cheap, and remote work is costly.
- *TurKit Script*: An API for writing algorithmic MTurk tasks using crash-and-rerun programming.
- *TurKit Online*: A public web GUI for running and managing TurKit scripts.
- *Example Applications*: Examples of algorithmic tasks explored in our lab, as well as algorithmic tasks explored by people outside our lab using TurKit.
- *Performance Evaluation*: An evaluation of TurKit's performance drawn from a corpus of 20 scripts posting almost 30,000 tasks over the past year.

CRASH-AND-RERUN PROGRAMMING

Consider a standard quicksort algorithm which outsources comparisons to Mechanical Turk (see Figure 2). This is a scenario where a local algorithm is making calls to an external system. Local computation is cheap, but the external calls cost money, and must wait for humans to complete work. The algorithm may need to run for a long time waiting on these results.

The challenge in this scenario is managing state over a long running process. This state can be kept in the heap, but this is dangerous in case the machine reboots or the program encounters an error. The error may be easy to fix, but all the state up to that point is lost.

State can be managed in a database, but this complicates the programming model, since we need to think about how to record and restore state. This can be particularly cumbersome for recursive algorithms like quicksort, which would require storing some representation of the call stack in the database.

The insight of crash-and-rerun programming is that if our program crashes, it is cheap to rerun the entire program up to the place it crashed, since local computation is cheap. This is true as long as rerunning does not re-perform all of the costly external operations from the previous run.

The latter problem is solved by recording information in a database every time a costly operation is executed. Costly operations are marked in a new primitive called *once*, meaning they should only be executed once over all reruns of a program. Subsequent runs of the program check the database before performing operations marked with *once* to see if they have already been executed.

Note that this model requires the program to be deterministic, since we are essentially storing complicated state in the logic of the program itself, rather than storing it explicitly in the database. Hence, *once* is important in the following conditions:

- *Non-determinism*. Since all calls to *once* need to happen in the same order every time the program is executed, it is important that execution be deterministic. Wrapping non-deterministic calls in *once* ensures that their outcomes are the same in all subsequent runs of the program.
- *High cost*. The whole point of crash-and-rerun programming is to avoid incurring more cost than necessary. If a function is expensive (in terms of time or money), then it is important to wrap it in *once* so that the program only pays that cost the first time the program encounters the function call. Typical tasks posted to Mechanical Turk cost 1 to 10 cents, and take between 30 seconds and an hour to complete.
- *Side-effects*. If functions have side-effects, then it may be important to wrap them in *once* if invoking the

```
quicksort(A)
    if A.length > 0
        pivot ← A.remove(A.randomIndex())
        left ← new array
        right ← new array
        for x in A
            if compare(x, pivot)
                left.add(x)
            else
                right.add(x)
        quicksort(left)
        quicksort(right)
        A.set(left + pivot + right)

compare(a, b)
    hitId ← createHIT(...a...b...)
    result ← getHITResult(hitId)
    return (result says a < b)
```

Figure 2: Standard quicksort algorithm that outsources comparisons to Mechanical Turk.

```
quicksort(A)
    if A.length > 0
        pivot ← A.remove(once A.randomIndex())
        left ← new array
        right ← new array
        for x in A
            if compare(x, pivot)
                left.add(x)
            else
                right.add(x)
        quicksort(left)
        quicksort(right)
        A.set(left + pivot + right)

compare(a, b)
    hitId ← once createHIT(...a...b...)
    result ← once getHITResult(hitId)
    return (result says a < b)
```

Figure 3: Standard quicksort augmented with the *once* primitive, to remember costly and non-deterministic operations for subsequent runs.

side-effect multiple times will cause problems. For instance, accepting results from a HIT multiple times causes an error from Mechanical Turk.

We can add `once` to our quicksort algorithm by surrounding the non-deterministic random pivot selection, as well as the expensive MTurk calls (see Figure 3). These modifications maintain the imperative style of the algorithm.

If the program crashes at any point, then subsequent runs will encounter all calls to `once` in the same order as before. Any calls which succeeded on a previous run of the program will have a result stored in the database, which will be returned immediately, rather than re-performing the costly or non-deterministic operation inside `once`.

Since crashing is so inexpensive in this model, we can crash instead of blocking. For instance, we implement `getHITResult` by crashing the program if the results are not ready, rather than blocking until the results are ready. This works because `once` only stores results if the operation succeeds.

If the user needs to change an algorithm so that it is incompatible with a recorded sequence of `once` calls, then they can clear this record in the database, and start afresh. `Once` also detects when the database is out of sync with the program by recording information about each operation, and ensuring that the same operation is performed on subsequent runs. In such cases, the program crashes, and the user is notified that the database and program no longer agree.

The benefits of the crash-and-rerun model include:

- *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 been 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 these results.
- *Easy to Implement*: Crash-and-rerun programming is easy to implement, and 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 `once`.
- *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, and print it to a file for analysis in an external program, like Excel.

TURKIT SCRIPT

TurKit Script is built on top of JavaScript. Users have full access to JavaScript, in addition to a set of APIs designed around crash-and-rerun programming and Mechanical Turk. JavaScript was chosen because it is a common scripting language, popularized primarily within webpages, but general purpose enough for many prototyping applications.

Crash-and-Rerun

TurKit supports crash-and-rerun programming in JavaScript by providing the `once` function described in the previous section. `Once` accepts another function as an argument. It calls this function, and if it succeeds (i.e. it returns without crashing), then it records the return value in the database, and returns the result back to the caller. When `once` is called on a subsequent run of the script, it checks the database to see whether a return value has already been stored. If so, it skips calling the argument function, but rather simply returns the stored value. For example:

```
var r = once(function () {
    return Math.random()
})
```

The first time the script runs, the function is evaluated, generating a new random number. This number is stored as the result for this call to `once`. The next time the script runs, `Math.random` is not called, and the random number generated on the previous call is returned instead.

TurKit also provides a convenient way to crash a script. The `crash` function throws a "crash" exception. Crash is most commonly called when external data is not ready, e.g., tasks on MTurk are not complete.

TurKit automatically reruns the script after an adjustable time interval. Rerunning the script effectively polls Mechanical Turk every so often to see if any tasks have completed. In addition, the online version of TurKit receives notifications from MTurk when tasks complete, and reruns any scripts waiting on these tasks.

Parallelism

Although TurKit is single-threaded, and the programmer does not need to worry about real concurrency in the sense of multiple paths of execution running at the same time, it does provide a mechanism for simulating simple parallelism. This is done using `fork`, which creates a new branch in the recorded execution trace. If `crash` is called inside this branch, the script resumes execution of the former branch. Note that `fork` can be called within a `fork` to create a tree of branches that the script will follow.

`Fork` is useful in cases where a user wants to run several processes in parallel. They may want to run them in parallel for efficiency reasons, so they can post multiple HITs on Mechanical Turk at the same time, and the script can make

progress on whichever path gets a result first. For example, consider the following code:

```
a = createHITAndWait()        // HIT A
b = createHITAndWait(...a...) // HIT B

c = createHITAndWait()        // HIT C
d = createHITAndWait(...c...) // HIT D
```

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

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 the first time the script runs, HITs A and C will be created, and each subsequent time it runs, it will check on both HITs to see if they are 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 E that required results from both HIT B and D:

```
fork(... b = ...)
fork(... d = ...)
join()
E = createHITAndWait(...b...d...) // HIT E
```

Using Mechanical Turk

The simplest way to use Mechanical Turk in TurKit is with the *prompt* function. This function shows a string of text to a turker, and returns their response:

```
print(mturk.prompt("Where is UIST 2010?"))
```

Prompt takes an optional argument specifying a number of responses to be returned as an array, so we can ask 100 people for their favorite color like this:

```
mturk.prompt("What is your favorite color?", 100)
```

In addition to these high level functions, TurKit provides wrappers around Amazon's MTurk REST API. These wrappers build on the crash-and-rerun library to make these calls safe, e.g., the *createHIT* function calls *once* internally so that it only creates one HIT over all runs of a program. These wrappers use the same naming conventions as MTurk, and handle the job of converting XML responses from Amazon into suitable JavaScript objects. TurKit also provides a *waitForHIT* function which crashes unless the results are ready. It is called *wait* because from the programmer's perspective, it waits for the results to be ready before returning.

Voting

The crash-and-rerun programming model allows us to encapsulate human computation algorithms into functions, which can be used as building blocks for more sophisticated algorithms.

One common building block is voting. We saw voting early on in Figure 1, but did not explain how it worked. Consider a simple voting function, where we want a best 3-out-of-5 vote. This is possible using a single HIT with 5 assignments (Amazon will ensure that each assignment is completed by a different turker). However, if we want to be even more cost efficient, we could ask for just 3 votes, and only ask for additional votes if the first 3 are not the same. This implies a simple algorithm:

```
function vote(message, options) {
    // create comparison HIT
    var h = mturk.createHITAndWait({
        ...message...options...
        assignments : 3})

    // get enough votes
    while (...votes for best option < 3...) {
        mturk.extendHIT(...add assignment...)
        h = mturk.waitForHIT(h)
    }

    // cleanup and return
    mturk.deleteHIT(h)
    return ...best option...
}
```

TurKit's version of this function takes an optional third parameter to indicate the number of votes required for a single option. One could also imagine extending this function to support different voting schemes.

Sorting

Another building block is sorting. A first attempt at sorting is simple using the crash-and-rerun model. We just take JavaScript's sort function and pass in our own comparator. Recall from Figure 1:

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

One problem with this approach is that all of the comparisons are performed serially, and there is no good way to get around this using JavaScript's *sort* function because it requires knowing the results of each comparison before making additional comparisons. However, in TurKit we can implement a parallel quicksort, as shown in Figure 4. This implementation is fairly straightforward, and shows where TurKit's parallel programming model succeeds. Limits of this approach are discussed more in the discussion section.

Creating Interfaces for Turkers

The high level functions described so far use Mechanical Turk's custom language for creating interfaces for turkers. However, more complicated UIs involving JavaScript or

```
quicksort(a) {
    if (a.length == 0) return
    var pivot = a.remove(once(function () {
        return Math.floor(a.length * Math.random())
    }))
    var left = []
    var right = []
    for (var i = 0; i < a.length; i++) {
        fork(function () {
            if (vote("Which is best?",
                   [a[i], pivot]) == a[i]) {
                right.push(a[i])
            } else {
                left.push(a[i])
            }
        })
    }
    join()
    fork(function () {
        quicksort(left)
    })
    fork(function () {
        quicksort(right)
    })
    join()
    a.set(left.concat([pivot]).concat(right))
}
```

Figure 4: A parallel quicksort in TurKit using *fork* and *join*.

CSS require custom webpages, which Mechanical Turk will display to turkers in an iframe.

TurKit provides methods for generating webpages and hosting them on TurKit's server. Users may create webpages from raw HTML, or use templates provided by TurKit to generate webpages with common features.

One basic template feature is to disable all form elements when a HIT is being previewed. MTurk provides a preview mode so that turkers can view HITs before deciding to work on them, but turkers may accidently fill out the form in preview mode if they are not prevented from doing so.

TurKit also provides a mechanism for blocking specific turkers from doing specific HITs. This is useful when an algorithm wants to prevent turkers who generated content from voting on that content. This feature is implemented at the webpage level (in JavaScript) as a temporary fix until Amazon adds this functionality to their core API.

Implementation

TurKit is written in Java, using Rhino[1] to interpret JavaScript code, and E4X[2] to handle XML results from MTurk. State is persisted between runs of a TurKit script by serializing a designated global variable as JSON. This variable is called *db*.

The crash-and-rerun module makes use of *db* to store results between runs of the script. The basic idea is to record a trace of once calls in an array. As the script runs, we maintain a pointer to the next location in this array.

When *once* is called, it checks the information stored at the next location in the trace. If there is a return value there, it returns this immediately. Otherwise, it calls the function passed as a parameter to *once*. If the function succeeds, then it writes information about this call into the trace. After the call to *once* completes, the pointer moves to the next location in the trace.

Implementing *fork* requires managing a stack of instruction pointers. *Fork* also consumes an element in the array of *once* calls, except instead of storing a return value there, it stores another array of *once* calls.

The *crash* function is implemented by throwing a "crash" exception. This exception is caught internally by the *fork* function, so that it can pop the forked branch off the stack of instruction pointers, and return. If *crash* is ever called, even if it is caught by a *fork*, then TurKit will schedule a rerun of the script after some time interval.

ONLINE WEB INTERFACE

Figure 5 shows the TurKit web-based user interface, an online IDE for writing TurKit scripts, running them, and automatically rerunning them. The interface also has facilities for managing projects, editing files, viewing output, and managing the execution trace.

The run controls allow the user to run the project, and start and stop automatic rerunning of the script. This is necessary in the crash-and-rerun programming model 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 Mechancial Turk until the results are ready.

There are also controls for switching between sandbox and normal mode on Mechanical Turk, as well as clearing the database. Together, these tools allow users to debug their scripts before letting them run unattended. Sandbox mode does not cost money, and is used for testing HITs. Users typically run a script in sandbox mode and complete the HITs themselves in the MTurk sandbox.

After the script appears to be working in the sandbox, the programmer may reset the database. Resetting the database clears the execution trace, as well as deletes any outstanding HITs or webpages created by the script. The user may now run the script in normal mode, and it 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 recorded execution trace.

The *execution trace* panel shows a tree view representing the recorded actions in previous runs of the script. 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, *createHIT* has a link to the Mechan-

[1] http://www.mozilla.org/rhino/
[2] http://en.wikipedia.org/wiki/ECMAScript_for_XML

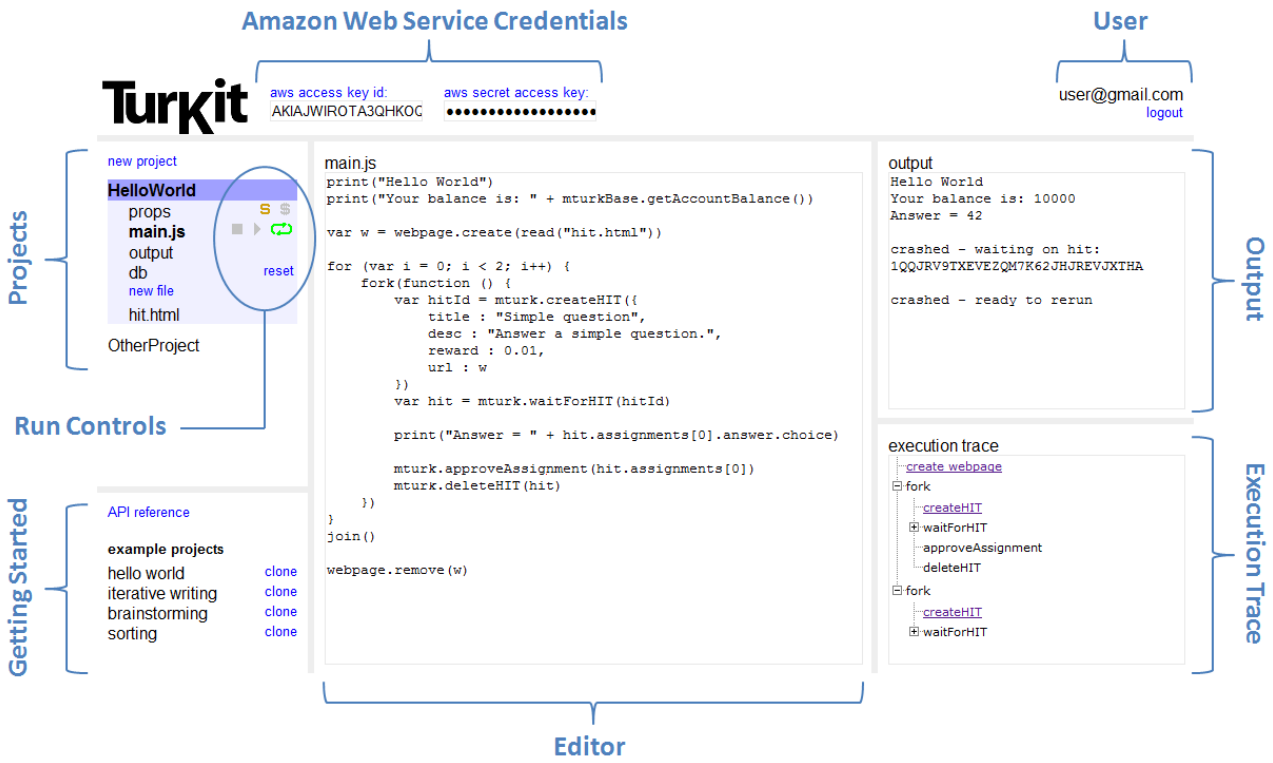


Figure 5: This is the TurKit web user interface, an online IDE for writing TurKit scripts, running them, and automatically rerunning them. Projects appear on the left, an editor appears in the center, and output appears to the right. There is also an *execution trace* pane showing the history of recorded actions. The run controls area has options for switching between sandbox and normal mode on Mechanical Turk, running the script, letting the script rerun automatically, and resetting the script. The lower-right contains a link to the TurKit API reference, as well as example projects which can be cloned as a starting point for writing scripts.

ical Turk webpage for the HIT, and the `webpage.create` function has a link to the public webpage that was created.

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.

Implementation

The web-based GUI runs on Google App Engine[3] (GAE). This choice was made because it is a free scalable server, and because it provides an easy way for users to log in using their existing Google account.

The web-app is built on top of TurKit, with extra security enhancements. In particular, Rhino generally allows JavaScript code to access Java directly. In order to protect users from damaging the server, or accessing each other's data, we only allow access to a secure set of Java classes.

EXAMPLE APPLICATIONS

This section describes applications we have explored using TurKit, as well as use cases outside our group.

[3] http://code.google.com/appengine/

Iterative Writing

TurKit has been used to run many experiments which involve asking one turker to write a paragraph with some goal. The process then shows the paragraph to another person, and asks them to improve it. The process also has people vote between iterations, so that we eliminate contributions which don't actually improve the paragraph. This process is run for some number of iterations. Figure 6 shows template code for a simple version of this algorithm. We have run many scripts like this to describe images (see Figure 7). These scripts are slightly more complicated because we need to generate a UI displaying an image.

From our iterative paragraph writing experiments, we have observed that most improvements involve making the paragraph longer (note that we limit the size to 500 characters). Also, people tend to keep the style and formatting introduced by earlier turkers in an iterative sequence.

Blurry Text Recognition

As another example of an iterative task using a similar structure, but achieving a different goal, consider the task of doing hard OCR. This is similar to reCAPTCHA [2], except it may work when the text is so unreadable that context and seeing other people's guesses may be necessary to decipher the passage. Figure 8 shows an example transcription of an artificially blurred passage.

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

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

Figure 6: Template for a simple iterative text improvement algorithm.

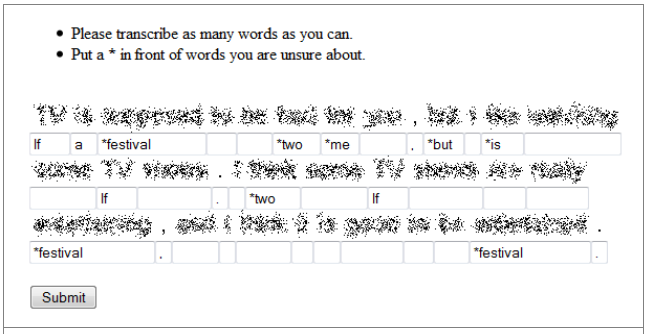
Iteration 1: Lightening strike in a blue sky near a tree and a building.
Iteration 2: The image depicts a strike of fork lightening, striking a blue sky over a silhoutted building and trees. (4/5 votes)
Iteration 3: The image depicts a strike of fork lightning, against a blue sky with a few white clouds over a silhouetted building and trees. (5/5 votes)
Iteration 4: ~~The image depicts a strike of fork lightning, against a blue sky- wonderful capture of the nature.~~ (1/5 votes)
Iteration 5: 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)
Iteration 6: 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. (4/5 votes)

Figure 7: Iterative text improvement of an image.

We can see the guesses evolve over several iterations, and the final result is almost perfect. We have had good success getting turkers to translate difficult passages, though there is room for improvement. For instance, if one turker early in the process makes poor guesses, these guesses can lead subsequent turkers astray.

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" [4]. The thesis presents a



Iteration 4: TV is* *festival ____ was *two *me ____ , *but ____ *is ____ ____ TV ____ . I *two ____ tv ____ ____ ____ *festival , ____ I ____ ____ is* ____ it ____ *festival .

Iteration 6: TV is supposed to be bad for you , but I ____ watching some TV *shows . I think some TV shows are *really *advertising , and I ____ ____ is good for the ____

Iteration 12: TV is supposed to be bad for you , but I am watching some TV shows . I think some TV shows are really entertaining , and I think it is good to be entertained .

Figure 8: Blurry text recognition. Errors are shown in red. The error in iteration 12 should be "like" instead of "am", according to ground truth.

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 a particular decision problem:

Consider a sequence of N numbers, each chosen randomly between -10 and 10. The goal of the participants is to guess whether the sum of the N numbers is positive or negative. Each person is provided two options, "negative" or "positive", and sometimes a third option "I don't know". Person x in the sequence is given three pieces of information:

- the fact that they are person x in the sequence
- the x^{th} number of the N numbers
- the decision of the $(x - 1)^{th}$ person, if $x > 1$

TurKit was used to simulate this setup using real humans on Mechanical Turk, and run 50 trials of this problem for two conditions: with and without the option "I don't know". The first condition replicated the findings of prior results which used classroom studies, and the second condition 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 used an early version of TurKit, and had difficulty discovering the parallelization features in that version.

Psychophysics Experimentation

Phillip Isola, a PhD student in Brain and Cognitive Science, is using TurKit to explore psychophysics. He is 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 [3].

His current experiments involve using TurKit to run genetic algorithms where humans perform both the mutation and selection steps. For instance, he has 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.

He has also applied genetic algorithms to sorting. In one experiment, he shows users a list of animals, and asks them to reposition one of the animals in the list. Other users select the best ordering from several candidates. Users are not told how they should sort the animals. In one instance, the result is an alphabetical sorting.

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 also used an early version of TurKit, and had difficulty discovering the parallelization features. This issue is discussed more in the Discussion section below.

PERFORMANCE EVALUATION

This paper claims that the 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 past 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 9 and Figure 10 give a sense for how long tasks take to complete once they are posted on MTurk. Figure 9 shows the round-trip time for the first assignment to complete after posting HITs with various payoffs. Part of this time is spent waiting for turkers to accept each task, and the rest is spent waiting for turkers to perform the work. Our higher paying tasks are typically more difficult, so we expect them to take longer to perform. However, if we subtract this time, the chart still increases, meaning it takes longer for turkers to *start* the higher paying tasks. One explanation is that turkers sort by reward, and 10-cent tasks are not on the first page of results. Another explanation is that turkers are looking for quick and easy tasks.

Figure 10 gives a better picture of the round-trip time-to-completion for the 1-cent tasks. The average is 4 minutes, where 82% take between 30 seconds and 5 minutes. About 0.1% complete within 10 seconds. The fastest is 7 seconds.

Figure 11 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, since

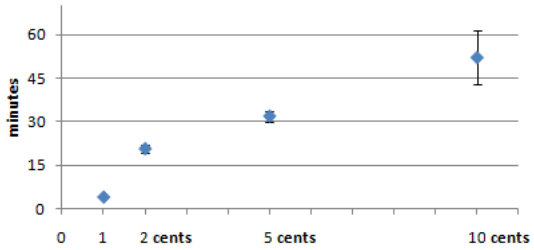


Figure 9: Average time until the first assignment is completed after posting a HIT with 1, 2, 5, or 10 cents reward. Error bars show standard error.

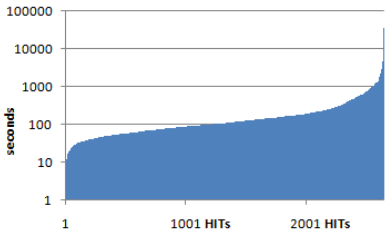


Figure 10: Time until the first assignment is completed for 2648 HITs with 1 cent reward. Five completed within 10 seconds.

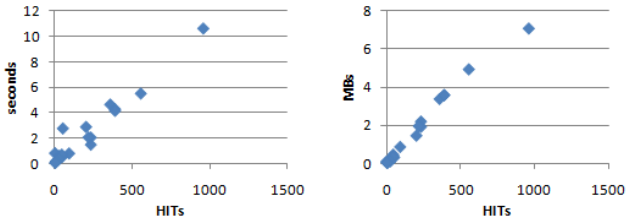


Figure 11: Time and space requirements for 20 TurKit scripts, given the number of HITs created by each script.

this measure is more correlated to space and time requirements than “calls to once” or “assignments created”. Note that for every HIT created, there is an average of 6 calls to once, 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.

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 programming 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 only be faster than 70% of hit-completion times. For such a slow script, it may be worth investigating options beyond the crash-and-rerun model.

DISCUSSION

We have iterated on TurKit for over a year, and received feedback from a number of users (four in our group, and

two outside our group, noted above). This section discusses what we've learned, including some limitations of TurKit, and areas for future work.

Usability

The TurKit crash-and-rerun programming model makes it easy to write simple scripts, but users have uncovered a number of usability issues. 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` is dangerous, and must be wrapped in `once`. This led us to override `Math.random` with a wrapper that uses a random seed the first time the script executes, and uses the same seed on subsequent runs (until the database is reset).

Users were also often unclear about which aspects of a TurKit script were stored in the execution trace, and which parts could be modified or re-ordered. This was due primarily to the fact that many functions in TurKit call `once` internally (such as `createHIT` and `waitForHIT`). We mitigated this problem by adding a view of the execution trace to the GUI, making clear which aspects of the script were recorded. This also allows users to delete records from the execution trace for fine-grained control of their script. Doing this before required advanced knowledge of how the trace was stored in the database.

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.

Scalability

The crash-and-rerun model favors usability over efficiency, but does so at an inherent cost in scalability. Whereas a conventional program could create HITs and wait for them in an infinite loop, a crash-and-rerun program cannot. The crash-and-rerun program 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.

One way to overcome this barrier is to use continuations and coroutines. 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. Continuations could be saved after all important calls (like createHIT), and a try-catch block around the entire script would catch any exceptions and store all the continuations in a database. The main drawback of this approach is that a serialized continuation includes the code of the script, so it cannot be reused if the script changes. This means that users could not incrementally modify their code between runs of a program, or use retroactive print-line debugging.

Parallel Programming Model

Parallel programming in the crash-and-rerun model is not completely general. For instance, we proposed a parallel version of quicksort that performs the partition in parallel, and then sorts each sublist in parallel. However, it joins between partitioning the elements, and sorting the sublists. In theory, this is not necessary. Once we have a few elements for a given sublist, we should be able to start sorting it right away (provided that we chose a pivot from among the elements that we have so far). Doing so is possible in TurKit by storing extra state information in the database, but is infeasible using `once`, `fork` and `join` alone.

Experimental Replication

The crash-and-rerun programming model offers a couple of interesting benefits for experimental replication using Mechanical Turk. 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 Mechanical Turk. 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. We post most of our experiments on the Deneme[4] blog, along with the TurKit code and database needed to rerun them.

RELATED WORK

Programming Model

Crash-and-rerun programming is related to early work on reversible execution [11], as well as more recent work on the Java Whyline which can answer causality questions about a program after it has already executed [10]. 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 must be written using an event-driven state-machine model.

Some innovative web application frameworks allow for an imperative model, including Struts Flow[5] and stateful django[6]. These and similar systems serialize continuations between requests in order to preserve state, which means they do not share many of the important benefits of crash-and-rerun programming, including incremental programming

[4] http://bit.ly/deneme-blog
[5] http://struts.apache.org/struts-sandbox/struts-flow/index.html
[6] http://code.google.com/p/django-stateful/

and retroactive debugging. 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 TurKit scripts may involve hundreds of people over several days.

Human Computation

Human computation systems generally involve many workers making small contributions toward a goal. Quinn and Bederson give a good overview of distributed human computation systems [16]. Individual systems have also been studied and explored in academic literature, including Games with a Purpose [1], Wikipedia [9] [15], and Mechanical Turk [7] [8] [12] [13] [14].

Human Computation Algorithms

Many human computation systems are embarrassingly parallel, where tasks to not depend on each other. Human computation algorithms involve more complicated orchestration of human effort, where workers build on each other's work. Kosorukoff uses humans in genetic algorithms [11]. Wikipedia itself may be viewed as a human computation algorithm. Each article involves many humans adding, improving and moderating content.

TurKit is a toolkit for exploring human computation algorithms. Human genetic algorithms, and processes within Wikipedia can be encoded as TurKit scripts and tested on Mechanical Turk. The applications and algorithms presented in this paper are merely first attempts at exploring this space. Already Dai, Mausam and Weld propose decision-theoretic improvements to algorithms proposed in this paper [5], which we could encode and test empirically using TurKit.

CONCLUSION

TurKit is a toolkit for exploring human computation algorithms on Mechanical Turk. We introduce the crash-and-rerun programming model for writing fault-tolerant scripts. Using this model, TurKit allows users to write algorithms in a straight-forward imperative programming style, abstracting Mechanical Turk as a function call. We present a variety of applications for TurKit, including real-world use cases from outside our lab.

The online version of TurKit is available now, as well as the source code: turkit-online.appspot.com. In addition to enhancing the TurKit UI and API, we are actively using TurKit to continue exploring the field of human computation algorithms as future work.

ACKNOWLEDGMENTS

We would like to thank everyone who contributed to this work, including Mark Ackerman, Michael Bernstein, Jeffrey P. Bigham, Thomas W. Malone, Robert Laubacher, Manal Dia, Phillip Isola, everyone who has tried TurKit, and members of the UID group. This work was supported in part by Xerox, 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. Cornsweet, T.N. The Staircase-Method in Psychophysics. The American Journal of Psychology, Vol. 75, No. 3 (Sep., 1962), pp. 485-491
4. Manal A. Dia. "On Decision Making in Tandem Networks". M.Eng. Thesis at MIT. 2009.
5. Dai, P., Mausam, Weld, D.S. Decision-Theoretic Control of Crowd-Sourced Workflows. AAAI 2010.
6. Feldman, S. I. and Brown, C. B. 1988. IGOR: a system for program debugging via reversible execution. In Proceedings of the 1988 ACM SIGPLAN and SIGOPS Workshop on Parallel and Distributed Debugging.
7. Heer, J., Bostock, M. Crowdsourcing Graphical Perception: Using Mechanical Turk to Assess Visualization Design. CHI 2010.
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. Ko, A. J. and Myers, B. A. Finding causes of program output with the Java Whyline. CHI 2009.
11. Kosorukoff A. Human based genetic algorithm. IlliGAL report no. 2001004. 2001, University of Illinois, Urbana-Champaign.
12. Mason, W. and Watts, D. J. Financial incentives and the "performance of crowds". KDD-HCOMP 2009.
13. Snow, R., O'Connor, B., Jurafsky, D., and Ng, A. Y. Cheap and fast---but is it good?: evaluating non-expert annotations for natural language tasks. EMNLP 2008.
14. Sorokin, A. and D. Forsyth, "Utility data annotation with Amazon MTurk," Computer Vision and Pattern Recognition Workshops, Jan 2008.
15. Susan L. Bryant, et al. Becoming Wikipedian: transformation of participation in a collaborative online encyclopedia. GROUP 2005.
16. Quinn, A. J., Bederson, B. B. A Taxonomy of Distributed Human Computation. Technical Report HCIL-2009-23 (University of Maryland, College Park, 2009).