Human Processing

(Position Paper)

Paul Heymann
Department of Computer Science
Stanford University
Stanford, CA, 94305, USA
heymann@stanford.edu

Hector Garcia-Molina
Department of Computer Science
Stanford University
Stanford, CA, 94305, USA
hector@cs.stanford.edu

ABSTRACT

In the future, humans will work for machines. This paper proposes an environment focused on modularity and reuse that makes human programming accessible to regular programmers. We compare the environment to the two most common environments available today.

Categories and Subject Descriptors

H.5.m [**Information interfaces and presentation (e.g., HCI)**]: [Miscellaneous]

Keywords

human processing, human programming, human computation, models

1. INTRODUCTION

Crowdsourcing uses large numbers of humans, available through the Internet, to perform useful tasks, either for pay or for free. Crowdsourcing is especially useful for tasks that are best performed by humans (as opposed to computers), e.g., tasks that involve image analysis, subjective evaluation, or natural language skills. We focus on *microtasks*, tasks that are usually less than five minutes in duration. (The overall task may be lengthy, but individual pieces are short.)

Developing a crowdsourcing application involves a lot of work, e.g., developing a web interface for the human workers to receive their assignments and return their results, computer code to divide the overall application into individual tasks to be done by humans, computer code to collect results, and so on. This paper describes a novel programming environment that automates many of the programming steps that must be performed in crowdsourcing applications. We start by giving a simple crowdsourcing example (Section 2). We show how a programmer would attack this example using two existing programming environments, which we call *Basic Buyer* (Section 3) and *Game Maker* (Section 4). Then, we show how the programmer would attack the same example using our novel proposed environment, *Human Processing* (Section 5). Finally, we contrast all three environments and describe remaining challenges in the area (Section 6).

2. MOTIVATING EXAMPLE

"Priam," the editor of a photography magazine, wants to rank photos submitted to the magazine's photo contest. For each environment below, we explain how Priam might go about accomplishing this task.

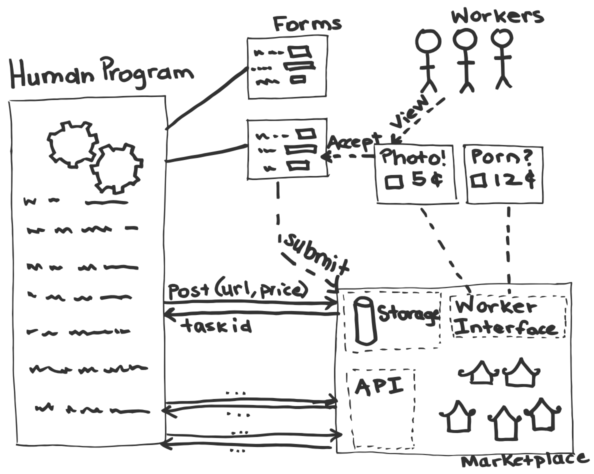

Figure 1: Basic Buyer human programming environment. A human program generates forms. These forms are advertised through a marketplace. Workers look at posts advertising the forms, and then complete the forms for compensation.

3. BASIC BUYER

The premise of Basic Buyer (BB) is that workers do short microtasks for pay, based on listings on a website (a *marketplace*). The BB environment is modeled on usage of Amazon's Mechanical Turk [1],[1] though a similar environment could be used with Gambit Tasks [2] or LiveWork [3]. However, because the programmer in BB targets a marketplace directly, and interaction patterns with marketplaces vary, switching marketplaces requires rewriting previous code.

The BB environment (Figure 1) works as follows:

1. The programmer (Priam) writes a normal program.
2. That program can, in the course of execution, create HTML forms at one or more URLs. This can happen in any of the usual ways that people currently generate web forms using web application frameworks.
3. The program can also interact with a marketplace, a website where *workers* (users on the Internet visiting the marketplace) look for tasks to complete. The pro-

[1]In particular, correspondences for the operations mentioned in this section are `post` → `CreateHIT`, `assignments,get` → `GetAssignmentsForHIT`, `approve` → `ApproveAssignment`, `reject` → `RejectAssignment`. We ignore bonuses and qualifications for ease of exposition.

gram can make one of five remote procedure calls to a monetary marketplace:

`post(url, price)` → `taskid` Tell the marketplace to display a link to `url` with the information that, if completed, the worker will be paid `price`. The URL `url` should correspond to a form which posts to the marketplace. The returned identifier `taskid` gives a handle for further interaction with the marketplace related to this posted task.

`assignments(taskid)` → `assignids` Return a list of identifiers `assignids` for looking up individual completions of the form associated with `taskid`.

`get(assignid)` → `dict` Get a dictionary that corresponds to a worker submitting the form associated with the `url` associated with the `taskid` associated with the given `assignid`. The dictionary contains which worker completed the task (a `workerid`) and the results of the form, as key-value pairs.

`approve(assignid)` Request that the marketplace pay the worker associated with `assignid` the `price` associated with the `taskid` that that `assignid` corresponds to.

`reject(assignid)` Request that the marketplace not pay the worker associated with `assignid` the `price` associated with the `taskid` that that `assignid` corresponds to.

The program posts one or more URLs, waits for assignments, gets the results, and then approves or rejects the work.

Priam determines that workers are best at ranking five photos at a time, so a web page is designed to display five photos and provide five entry fields for the ranks one through five. A computer program now needs to be written to read the photos from a database and generate multiple posts corresponding to groups of five photos. The program needs a strategy to do its work: for instance, it may employ a type of merge-sort strategy: divide the photos into disjoint sets of five, and rank each set. Then the sorted sets (runs) can be merged by repeatedly calling on workers.

In addition to the sorting logic itself, there is a lot of other "administrative" work that needs to be done. Of course, assignments need to be approved (paying workers for their work), but more importantly Priam needs to determine pricing, if and when the work being submitted is good, which workers are good, and so on. For example, one worker (a "spammer") might simply fill in junk in order to get paid. This spammer would need to be caught and their work ignored. Priam also may not pay enough initially, or may need to change his price over time depending on market conditions.

4. GAME MAKER

The Game Maker (GM) environment is modeled on the "Games with a Purpose" (GWAP) literature [5]. Many games have been developed, though our model is based most closely on the ESP Game (a photo captioning game). A key point in the GM environment is that to date, programmers have not shared interfaces or source code for popular games. For example, even though the ESP Game serves many players each day, it is not possible for Priam to get the (actual) ESP Game to label his own images. This means that the programmer usually has to develop and promote a new game, even if previous examples exist! (Even if the most popular GWAPs did have open interfaces, it is likely that switching between GWAPs would require rewriting code.)

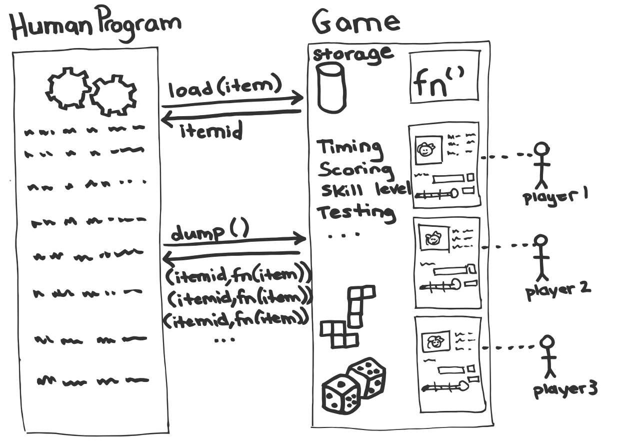

Figure 2: Game Maker human programming environment. The programmer writes a human program and a game. The game implements features to make it fun and difficult to cheat. The human program loads and dumps data from the game.

The GM environment (Figure 2) works as follows:

1. The programmer (Priam) writes two programs: the main program and a "game with a purpose."
2. The game is designed to take input items and compute some function `fn` of each input item by coercing players to compute the function during game play. For example, the ESP Game takes photos as input items and produces text labels as outputs [5].
3. The interaction between the main program and the game is simple:

   `load(item)` → `itemid` Add a new item for humans to compute the game's function on. Return an identifier for the item.

   `dump()` → `((itemid,res),...)` Get a list of all results that have been computed. Each returned tuple includes an itemid and the result of computing the game's function on the original item.
4. While the function `fn` computed is usually quite simple (e.g., "give some labels for this image"), the game itself is usually quite complex. This is for two reasons: the game must be fun, and the game must be difficult to cheat. Making the game fun can be time consuming, requiring features such as timed game play, multiple players, leaderboards, and quality graphic design. Making the game difficult to cheat can be equally time consuming, requiring features such as randomization, gold standards, statistical analysis, and game design according to particular templates (e.g., "output-agreement," "input-agreement" [5]).
5. The game may be a Flash game or any other format, the fact that it is used for human computation does not impact the technical details of how we program it.

Priam determines that the magazine's readers might be willing to play a game to determine the best photo. As with the Basic Buyer case, Priam needs to write a program to

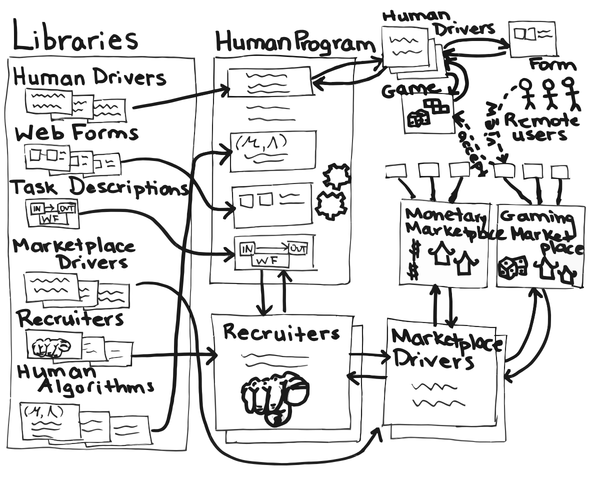

Figure 3: Human Processing programming environment. HP is a generalization of BB and GM. It provides abstractions so that algorithms can be written, tasks can be defined, and marketplaces can be swapped out. It provides separation of concerns so that the programmer can focus on the current need, while the environment designer focuses on recruiting workers and designing tasks.

handle the sorting logic. The program could then use the `load` and `dump` operations to get data in and out of the game. However, he now also needs to write a game where it is fun to sort groups of five photos, and then promote the game online. Lastly, he needs to make sure that players cannot cheat, either to make a particular contestant's photo do well, or for the player to do better by inputting bad data.

5. HUMAN PROCESSING

The Human Processing (HP) environment builds upon the BB and GM environments through abstraction. The HP environment (Figure 3) works as follows:

1. The programmer (Priam) writes a normal program. The programmer may also write one or more implementations of (see below) human drivers, human tasks, marketplace drivers, or recruiters. However, the point of the HP enviroment is to maximize code reuse, so ideally, existing implementations should cover the programmers' common use cases.
2. A *human driver* is a program which manages an associated web form or other user interface (so that the main program and other components do not have to talk directly to the user interface). It is so named because it manages the interaction with humans, much like a device driver manages a physical device on a computer. A human driver supports four operations:

   `open()` → `driverid` Make the associated user interface available to remote users. By *remote users* we mean workers in the BB model, players in the GM model, or other people capable of completing tasks. Returns an identifier for the driver.

   `send(driverid, msg)` Send message `msg` to the driver to change its behavior. In Priam's case, if he was using a driver for a game like the ESP Game he would use a send operation to load input photos.

   `get(driverid)` → `(d,e) or 0` Get a (result) data object d from the interface, with execution context e about how that data object was acquired. If no new data is available, return nothing. `get` is how results are returned from the driver. Both d and e are dictionaries of key-value pairs. For example, Priam's photo comparison interface returns d as
   `{ranks: (1,4,2,3,5), taskid: TID183}`
   (ranks are the output, and tasks are defined below) and e as
   `{workerid: WID824}`
   (a worker who completed the task).

   `close(driverid)` Make the associated user interface unavailable to remote users.

   A human program `open`s a driver and then `send`s setup messages. Human drivers for web forms may only receive one setup message, though those for games may be sent many messages to load inputs. Execution context comes from user interaction, for example, how long did the task take and which worker completed it? Such information can help with quality control in the main program. Finally, the human driver is closed. Note that by itself, a human driver can make its associated user interface available to remote users. However, it does not handle the problem of finding remote users to interact with the user interface.
3. The programmer reuses or defines structures called *human task descriptions*. A human task description consists of an input schema, an output schema, a human driver, a web form, and possibly other metadata. A human task description can be instantiated into one or more *human task instances*. These instances contain information as key-value pairs such as when the task started, a price if any, and so on. For example, a task description for Priam's case might look like ...
   ```
   {input:  (photo1, photo2, photo3, ...),
   output:  (int, int, int, int, int),
   webform:  compare.html,
   driver:  comparer.py}
   ```
   ... while a task instance might look like ...
   ```
   {start:  20090429,
   price:  $0.07,
   taskid:  TID272}.
   ```
4. A *marketplace driver* provides an interface to a marketplace. *Marketplaces* are a general term for both monetary marketplaces like Amazon's Mechanical Turk [1] (websites where workers are paid in money) and gaming marketplaces like GWAP (websites where users choose among many games and are paid in points or enjoyment). The environment may have many drivers for different marketplaces, and these drivers may have different interfaces depending on what the marketplaces themselves support.
5. The programmer avoids programming to any particular marketplace driver if at all possible. Instead, the programmer targets a *recruiter*, which is a program that serves as an interface to one or more marketplace drivers.[2] Recruiters support at least one operation:

   `recruit(taskid)` Ensure that the task instance

[2]In practice, some services like Dolores Labs' CrowdFlower may also be viewed as a form of recruiter.

`taskid` is completed by workers. The recruiter uses the task instance to find out how the user interface associated with the task is accessed. For example, if it is a web form, the task instance includes the URL of the web form. Then, the recruiter interacts with one or more marketplace drivers. In the case of the marketplace from the BB environment, one strategy might be to gradually increase the price until workers complete the web form. The recruiter also interacts with the human driver associated with the task instance to determine when no more workers are needed.

6. The environment includes a library of *human algorithms* to encourage code reuse. A human algorithm is a parameterized program which can handle many possible needs. (For example, it might include algorithms for sorting, clustering, and iterative improvement [4].) Often, it will be parameterized by human task descriptions, but other parameters might be used as well. For example, a pair-wise sort algorithm might take a human task description consisting of a human driver and web form to compare two items. The human task description would determine if the items compared were photos, videos, or something else.

The Human Processing environment is the novel environment we propose and have partially implemented in a project at Stanford called HPROC.

In the HP environment, Priam's workload is much reduced. A pairwise sorting algorithm HUMAN-PAIR-SORT is already included in the library. Priam may define a human driver and web form for comparing two photos, though these might already be available. Then, Priam defines a human task consisting of comparing two photos using the human driver, web form, and appropriate schemas. Lastly, Priam runs HUMAN-PAIR-SORT with his human task and a pre-defined recruiter. An example pre-defined recruiter is one that increases prices one cent each hour using Amazon's Mechanical Turk, though more complex recruiters exist.

6. DISCUSSION

HP extends BB and GM in compelling ways:

- *Cost.* BB excels for small numbers of tasks where programmer time is valuable. GM excels for large numbers of tasks where cheaper work is valuable. HP excels at both by providing payment optimizing recruiters and the opportunity to degrade to either BB or GM.
- *Ease.* BB quickly becomes complicated as the programmer gets bogged down in trivia like pricing. GM requires heavy attention to game play and cheaters. HP allows the programmer to focus on the tasks to be completed, rather than infrastructure.
- *Reuse.* There are no mechanisms in BB for reusing algorithms, forms, or administrative functionality. Current GM implementations do not share interfaces, and games tend to be specialized to specific use cases. By contrast, abstractions in HP allow for a library of infrastructure. Algorithms can target recruiter interfaces, recruiters can target market drivers, and so on.
- *Independence.* Programs in BB tend to be focused on a particular marketplace. Programs in GM tend to be tied to a particular web site's gaming user base. By contrast, programs written to HP have an independence due to marketplace drivers. (Likewise, algorithms, human drivers, and forms may have a similar independence.) Switching marketplaces or other infrastructure can require substantial rewriting in BB or GM, but does not in HP.
- *Algorithms.* General algorithms can be written to target a higher level interface in HP, but it is not clear how general algorithms can be reused in BB or GM.
- *Separation of Concerns.* Researchers or infrastructure writers can focus on improving recruiters, algorithms, and human drivers in HP, independent of a main program's code.

The more environments that implement HP, the easier it will be to leverage disparate work in algorithms, recruiters, and human drivers.

There are three main challenges in the future for HP.

1. *Verification, Quality Control.* GM focuses a great deal on verification, but BB and HP do not. How should we identify bad output? How do we identify high and low quality workers? We would like to see a generic, modular way to handle verification and quality control in an environment like HP.
2. *Recruiters.* We would like to see arbitrarily advanced recruiters. For example, not only would we like to see recruiters that price tasks on monetary marketplaces, but we would also like to see recruiters that can choose amongst alternative, equivalent task plans based on price and quality.
3. *Algorithms.* Algorithms targeted for the HP environment need to be developed for various purposes. For example, sorting with people is not the same as sorting with a computer! The HP environment provides a natural way to benchmark algorithms, based on cost, time, input, and output with a given recruiter.

Handling algorithmic design, verification, and quality control will further increase the ease of using HP. Advanced recruiters can improve the results produced by the main program (both in terms of cost and quality). We believe that HP is a strong foundation for future work in human computation, allowing for much greater reuse and modularization of common functionality.

7. REFERENCES

[1] `http://www.mturk.com/`.

[2] `http://getgambit.com/`.

[3] `http://www.livework.com/`.

[4] G. Little, L. Chilton, M. Goldman, and R. Miller. TurKit: Tools for Iterative Tasks on Mechanical Turk. In *HCOMP '09: SIGKDD Workshop on Human Computation.*

[5] L. von Ahn and L. Dabbish. Designing Games with a Purpose. *Commun. ACM*, 51(8):58–67, 2008.