# Crowd-Logic: Implementing and Optimizing Human Computation Algorithms using Logic Programming

*Hao Lü and James Fogarty*
Computer Science & Engineering
DUB Group, University of Washington
Seattle, WA 98195
{hlv, jfogarty}@cs.washington.edu

## ABSTRACT
Human computation is a powerful approach to addressing problems that remain beyond the reach of traditional computing and has been demonstrated in a variety of applications. However, implementing human computation programs remains challenging. We present Crowd-Logic, a general-purpose tool for implementing human computation algorithms, which allows application developers to declaratively specify the high-level control flow of algorithms using logic programming and implement human computation units using imperative programming (e.g., Java). The logical representation allows Crowd-Logic to maximally reuse the results from prior human computation, optimize the control flow, and reduce the general cost of human computation. We validate our optimization techniques on two sample applications: sorting a list of reviews using human judgment and shortening text using human rewrites.

**Keywords:** Crowdsourcing, logic programming, planning, machine learning

## INTRODUCTION
Human computation is a powerful approach to addressing problems that remain hard for computers but easy for humans to solve. Markets for human computation, such as Amazon Mechanical Turk (MTurk) [2], allow developers to programmatically post tasks that can be completed by human workers in return for payment. Programs can thus integrate human computation into their algorithms. Recent systems have used human computation to answer visual questions from blind people posted from their mobile phones [4], shorten and proofread texts, and run scripts in natural languages [3].

However, implementing human computation programs introduces new challenges. First, human feedback is slow; it takes time to both find a worker and for that worker to complete a task. Second, calls to human workers can introduce monetary cost, which can quickly accumulate over time. Third, human feedback is unreliable, which means systems must be architected such that some human workers verify or improve the work of others [3]. Developers therefore have to manage tradeoffs between speed, money, and reliability in designing their algorithms. For example, additional verification steps can improve reliability but increase monetary costs and reduce speed.

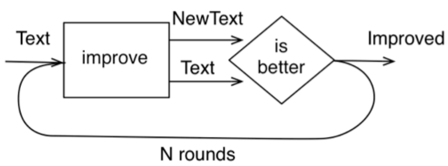
These challenges also introduce new opportunities for traditional computation. Since human computation can become the new computation bottleneck, developers may prefer to spend more computational resources on the human computation if this reduces delays, saves money, or improves reliability. For example, one such opportunity is to reuse results from prior human computations.

Existing general-purpose human computation tools have focused on APIs for easily accessing human computation markets [2,9] and managing long running programs [9]. TurKit [9], for example, enables a program to efficiently pause and continue in order to improve the stability of the program over time. To our knowledge, no prior work has focused on tool that supports managing the cost of traditional computation against the cost of human computation.

We present Crowd-Logic, a general-purpose tool for implementing human computation algorithms. In Crowd-Logic, the high-level structure of an algorithm is specified using logic programming and human computations are encapsulated into logical primitives, which are implemented via imperative programming (e.g., Java). The logical representation of algorithms allows Crowd-Logic to maximally reuse the results from prior human computation. It also introduces opportunities for traditional computation to optimize algorithms for reducing the general cost of human computation.

In this paper, we first show how developers can implement a human computation algorithm in Crowd-Logic. We then show how Crowd-Logic can maximally reuse the results of prior human computation. Next, we discuss the techniques for optimizing algorithms to reduce the general cost of human computation. We then validate our optimization techniques in two sample applications: sorting a list of restaurant reviews using human judgment and shortening text using human rewrites. Finally, we discuss the current limitations and opportunities for future work.

(a)




(b)

```
Rule 1 | one_round(Image, Text, NewText) :-
       |  improve(Image, Text, NewText),
       |  is_better(Image, NewText, Text).
Rule 2 | one_round(Image, Text, Text).
Rule 3 | iterate(0, Image, Text, Text).
Rule 4 | iterate(N, Image, Text, NewText) :-
       |  N > 0,
       |  one_round(Image, Text, T),
       |  iterate(N - 1, Image, T, NewText).
```

(c)

```
class MyLibrary {
    @HumanPrimitive{cost=1}
    public void improve(
        BindingTable bt,
        CTerm image,
        CTerm text,
        @OUT CTerm newText) {
      // Ask a person to improve the
      // given description.
    }

    @HumanPrimitive{cost=3}
    public boolean is_better(
        BindingTable bt,
        CTerm image,
        CTerm newText,
        CTerm text) {
      // Ask people to judge if the
      // the new text is better than
      // the original text.
    }
```

Figure 1: Generate plain text descriptions for image. (a) The common approach is to use the iterative text improvement workflow. (b) The logical representation of the iterative text improvement workflow. (c) The two Java functions that implement the corresponding human computation units in the logic program.

Our main contributions are:

- Crowd-Logic, a general-purpose tool for implementing human computation algorithms that maximally enables reuse of prior human computation, and
- A set of techniques for optimizing algorithms to reduce the general cost of human computation.

## IMPLEMENTING AN ALGORITHM WITH CROWD-LOGIC

In this section, we present an overview of Crowd-Logic via an example of using Crowd-Logic to implement a common human computation use case, *DescribeImage*: generating plain text descriptions for images. More specifically, in this image description problem, the task is to generate a good plain text description for a given image by asking human workers to describe the image.

Previous research has looked at this problem [9,12]. A common approach is to use an iterative improvement workflow as shown in Figure 1a. The workflow iterates over improving a description ("improve") and verifying the improvement ("is better"). In each round of the iteration, first, a human worker is asked to improve an old description for the image by writing a new description. Then, a few human workers are asked to vote on if the new description is indeed better than the old one. The better version will then be improved on in the next round. The iteration terminates after a predetermined number of rounds.

The workflow in Figure 1a can be broken down into two parts: the control flow and two basic human computation units (e.g., "improve" and "is better"). Correspondingly, the Crowd-Logic implementation of the above workflow has two parts: a logic programming part for specifying the control flow (shown in Figure 1b), and a Java part that implements the basic human computation units (shown in Figure 1c).

The logic programming part contains four logic rules. Rule 1 specifies the control flow of a single round of the iteration ("`one_round`"), which consists of two sequential steps: "`improve`" and "`is_better`". Rule 2 specifies that take the original description should be used if a better description is not found in Rule 1. Rule 3 and 4 specify recursively the iteration with N rounds of "`one_round`".

The Java part contains two functions that implement the two human computation units of the same names in the logic program. Figure 1c shows the skeleton of their implementation. Each human computation unit is implemented in its corresponding method of the library class. Crowd-Logic will load the library and scan for methods that implement human computation units.

The annotation "`@HumanPrimitive`" is used on Java methods to indicate that a particular method is an implementation of a human computation unit. The name of the method is same as its logical counterpart. Some metadata can be specified in the annotation. For example, the code in Figure 1c specifies the cost of "`improve`" is 1 and the cost of "`is_better`" is 3. The definition of the cost is up to the developers.

The annotation "`@Out`" applies on the parameters of the Java methods to indicate that a parameter is intended for storing an output value from human workers (an output parameter). For example, the "`newText`" parameter of "`improve`" is to store the rewrite of the old description. In contrast, the method "`is_better`" does not have such output parameters. The result of its human computation is communicated back via its return value.

The above demonstrates how Crowd-Logic combines logic programming and imperative programming. An imperative implementation could have a similar structure to Figure 1, however, when Crowd-Logic runs, one advantage is that it can maximally reuse the prior results of "`improve`" and "`is_better`", for example, if there are `K` rounds of improvement in the past, Crowd-Logic will start from the `K+1` round.

### Logic Programming

In the rest of this subsection, we introduce the basic notions in logic programming that will be used in this paper. More details can be found in [15].

Logic programming has two basic constructs.

***Terms*** are the single recursive data structure in logic programming. A *term* can be:

- *Number*: strings in number form, i.e., `0`, `3.14`.
- *Variable*: strings that start with an upper-case letter or underscore, i.e., `Image`, `Text`, `N`. At runtime, a variable can be assigned a value by *binding* it to another term.
- *Atom*: general strings with no other meanings, i.e., `true`, `'a.png'`.
- *Compound Term*: which are composed of an *atom* (name) and a list of *terms* (arguments), i.e., `improve(Image, Text, NewText)`. A *compound term* can mean:
  - *Predicate,* can be true of false depending on the values of its argument.
  - *Primitive,* a *predicate* that is defined outside the logic programs and usually in an imperative program. They provide abstractions that are difficult to express in logic programs. In Crowd-Logic, we use human primitive when it provides an abstraction to some human computation.
- *List*: a special *compound term* to represent a list. The following two are equivalent in representing a list consists of 1, 2 and 3: `[1,2,3]`, `[1|[2, 3]]`.

***Rules*** take the following form:

$$A :- B_1, B_2, \ldots , B_n.$$

This states that `A` (the head of the rule) is true if all the `Bi`'s (the body of the rule) are true. `A` and `Bi`'s are all logical *predicates*. A rule with an empty body is called a *fact*, which states that the head predicate is always true. A rule with no head *predicate* is called a *query*, which initiates the computation.

In logic programming, the computation is done interpretively. To solve a query, the interpreter will search for a proof (a sequence of logical deductions from the *facts*) for its body and the *bindings* for all its *variables*.

To get a text description for an image using *DescribeImage*, we can input a query to Crowd-Logic. For example, the following query will generate a description for the image 'a.png' using 10 iterations:

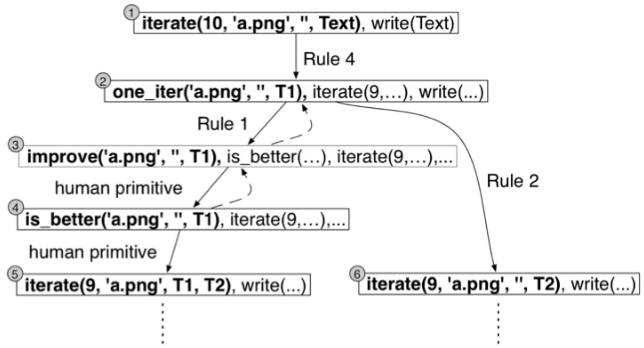



Figure 2: A proof search tree: an illustration of the goal-reduction interpretation using rules from *DescribeImage*. (This example does not consider reusing prior human computation.)

```
:- iterate(10, 'a.png', '', Text), write(Text).
```

(The predicate "`write`" is a built-in *primitive* that outputs the value of "`Text`" to the console.)

There are many existing search procedures for logic programs. Crowd-Logic adopts the goal-reduction search procedure. Each predicate is interpreted as a *goal*. Rules are interpreted as ways to reduce a goal (head of the rule) into a sequence of sub-goals (body of the rule). The facts and primitives are terminals and are reduced to an empty sequence. The procedure of searching for a proof is interpreted as searching for a sequence of reductions from the initial sequence of goals to an empty one. The search space is a tree structure with each state corresponding to a goal sequence. The first goal of a state is always the active goal, which is to be reduced next. We will refer to this search tree as a *proof search tree*. The search strategy is backtracking, e.g., traversing the tree in depth-first order.

A sample proof search tree is shown in Figure 2. The initial query is turned into state 1 with two goals. Rule 4 can reduce the active (first) goal in state 1 into two sub-goals and result in state 2. Both Rule 1 and Rule 2 can reduce the first goal of state 2 and transition to state 3 and state 6 respectively. Since Rule 1 is specified before Rule 2 in the logic program, it has a higher priority and the search process will try Rule 1 first. The search process will then evaluate the human primitives in states 3 and 4, as they are the active goals. If the result of "`is_better`" is false, the search process will backtrack to state 2 and try Rule 2. This search process will eventually reach a state "`iterate(0, ...), write(...)`" in which the first goal is a fact and second goal is a primitive.

While logic programming languages are simple yet as expressive as imperative languages, most developers are unfamiliar with logic programming. In addition, the API support for logic programming languages to access existing human computation markets is lacking. These make it a high threshold to implement a human computation algorithm entirely in logic programs. However, logic programming has the benefit of enabling effective reuse and optimization in human computation. Crowd-Logic lowers such threshold by allowing developers to only specify the high-level algorithms in logic programs.

## REUSE HUMAN COMPUTATION

Since human computation can be slow, unreliable and expensive, the opportunity to reuse results from prior human computation becomes valuable. Crowd-Logic assumes that the results from human primitives are reusable unless otherwise specified. When the interpreter searches for a solution, it always tries to reuse the prior results of the human primitives when possible.

Consider the task of sorting a list of restaurant reviews based on their sentiments. One solution is to implement the quicksort algorithm and ask human to compare the sentiments of two reviews. During the sorting, many comparisons between two reviews may have already been done before. They can result from situations when new reviews have been added and the old reviews have already been sorted. They can result from the previous runs that have terminated unexpectedly. They can also result from other tasks that share the same human comparison unit. These comparisons can reuse the prior comparison results rather than asking human to compare them again.

The above is the simplest case of reusing prior human computation results. It can be realized by storing all prior comparisons in a local database and every time a human comparison is required, the interpreter reuses the prior result if the same comparison is recorded in the database.

Consider a more complex example. In *DescribeImage*, the interpreter can encounter the two sequential primitives as shown in Figure 3a when searching for a solution. When it is at the first primitive, e.g., "`improve`", assuming that the same human computation (improving an empty description of 'a.png') has been done a few times before, the interpreter has to make a decision on whether or not to reuse the prior results and if so, which result to be used. However, the interpreter does not have enough information to make such decision. It is until when it sees the next primitive, e.g., "`is_better`", does the interpreter know that it can reuse a result that is better than the empty description. If no such reuse is possible, the interpreter should then invoke the "`improve`" primitive to get a new human rewrite. Things will get more complicated when multiple rounds of improvements are concerned as shown in Figure 3b.

To enable reuse in the general case, Crowd-Logic introduces a constraint solver and sees all human primitives as constraints. For example, the primitive "`improve('a.png', T1, T2)`" in Figure 3b is interpreted as a constraint that constrains the values of the two variables `T1` and `T2` to have both occurred in one of the prior runs of the primitive. A constraint can be satisfied if such values exist.

Constraints can have dependencies between each other via variables. For example, the constraint "`improve('a.png', T1, T2)`" depends on the other constraint "`improve('a.png', '', T1)`", because its

**(a)**

```
improve('a.png','',T), is_better('a.png',T,'')
```

**(b)**

```
improve('a.png','',T1), is_better('a.png',T1,''),
improve('a.png',T1,T2), is_better('a.png',T2,T1)
```

**(c)**

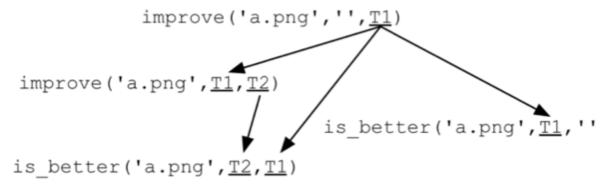



Figure 3 Constraints and their dependency graph

argument `T1` is an output from the latter. A dependency graph can be obtained by representing constraints as nodes and dependencies as edges. Figure 3c shows the dependency graph of the constraints in Figure 3b.

The interpreter stores a list of constraints and maintains a dependency graph for them. When it encounters a constraint, the interpreter adds the constraint to its list of constraints. If the constraint solver determines that the resulting constraints are no longer satisfied, the interpreter then backtracks. A set of constraints is satisfied when there exists an assignment for each variable such that each constraint is satisfied. When a constraint is backtracked, the interpreter decides whether or not to invoke the human computation depending on whether it depends on other constraints (from the same rule in the logic program).

When a constrained variable is to be used in a non-constrained way, such as invoking human computation, or print its content, the constraint solver will find all its possible assignments and the interpreter will backtrack on all its assignments.

We have described how constraints and the constraint solver are used in the interpreter. In the rest of this section, we examine the implementation of our constraint solver. Since checking if a set of constraints can be satisfied is equivalent to checking if there are possible assignments for all variables, we only describe the algorithm used for finding all possible assignments of a variable.

Crowd-Logic stores all the prior results from human primitives in a local relational database. Each human primitive is managed in its own database table. For example, the primitive "`improve`" is managed in the table "`improve`". Each entry of a database table records a prior computation indexed by their ids.

To find all possible assignments of a constrained variable, the constraint solver first finds all the constraints that contain the variable. Then, it computes a connected component in the dependency graph that contains this set of constraints. It then translates the constraints into the joins of corresponding database tables. The possible assignments of the variable are the values from the column of the joint table that corresponds to the output of the variable. For example, the finding of all possible assignments of variable `T` in constraints shown in Figure 3a can be translated into the following SQL query:

```
SELECT improve.col_3
 FROM improve, is_better
 WHERE improve.col_3 = is_better.col_2
  AND improve.col_1 = 'a.png'
  AND improve.col_2 = ''
  AND is_better.col_1 = 'a.png'
  AND is_better.col_3 = ''
```

The use of constraint solver essentially allows the decision on what value to be reused to be much delayed, so that the computation can get much more efficient and the reuse of prior human computation can be maximized.

## OPTIMIZATION AND DECLARATIVE APPROACH

Human computation introduces new potential for tradeoffs between the costs of traditional computation and the costs of human tasks. A developer may prefer to spend large amounts of traditional computational resources if this reduces delay or cost, or improves quality associated with human tasks. For example, TurKontrol [5] uses a planning algorithm to control the quality of a task. In Crowd-Logic, such new tradeoffs allow our interpreter to use more traditional computational resources to optimize the execution of the logic programs.

Logic programs have a declarative, logical interpretation, which specifies the computation without describing the actual control flow. It gives the interpreter some freedom in deciding how to execute a program. The interpreter can optimize an algorithm by choosing the best control flow based on factors such as delay and monetary cost.

In the next section, we examine how Crowd-Logic optimizes for the general cost of human computation.

## OPTIMIZING FOR COST

As a simple example, the following logic program provides two rules to determine if `X` is in one of the two given groups (assuming "`in_group`" is a human primitive):

```
in_group2(X, A, B) :- in_group(X, A).
in_group2(X, A, B) :- in_group(X, B).
```

In its declarative interpretation, the interpreter has two choices of control flows: either test the first rule first or test the second rule first.

Consider the situation where "`in_group(X, B)`" (or "`in_group(X, A))`" is previously computed, the interpreter can reduce the cost of human computation by choosing to test the second (or first) rule first as it will directly reuse the prior result and potentially eliminate the need to test the other rule.

In a more difficult situation where no reuse is available, e.g., neither "`in_group(X, A)`" nor "`in_group(X, B)`" is ever computed, if however the interpreter can predict which group `X` is more likely to be in, it can reduce the average cost of human computation by choosing to test the more likely group first. Such prediction can be enabled through a statistical machine learning system trained on the previous results of computing "`in_group`".

The above two situations illustrate how Crowd-Logic optimizes control flows to reduce the cost of human computation. To realize such strategy, Crowd-Logic implements a set of techniques, including introducing two new nondeterministic predicates and a best-first search strategy with heuristics from a lookahead algorithm.

### Nondeterministic Predicates

Though a logic program could have a declarative interpretation with only partial control flow, the language specification usually forces a procedural interpretation with a deterministic control flow. In the "`in_group2`" example, a common procedural interpretation assumes the first rule will always be tested first. Such assumption is indeed important to many logic programs, for example, in Figure 1, the order of Rule 1 and Rule 2 is rigorous and the semantics will change if they are interchanged.

In order to enable a partial control flow, Crowd-Logic introduces two new primitives, "`nondeterministic`" and "`choose`". Developers can use them to specify where they want the nondeterministic semantics.

The primitive "`nondeterministic`" takes the name of a predicate as argument. It informs the interpreter that when in the search for a proof for the given predicate, the order of the rules to be tried can be arbitrary. For example, to make the aforementioned "`in_group2`" example work in Crowd-Logic, the developer can add the following query:

```
:- nondeterministic(in_group2).
```

The primitive `choose` will choose an item from a list in any nondeterministic order. For example, "`in_group2`" can be rewritten in a single rule with "`choose`":

```
in_group(X, [A, B]) :-
        choose([A, B], G), in_group(X, G).
```

It is a handy shortcut for specifying nondeterministic behavior with lists. In theory, "`choose`" can be specified by "`nondeterministic`". However, implementing it as a primitive is much more efficient.

With the introduction of the above two predicates, the interpreter is allowed to make decision upon nondeterministic predicates about which action to take in its search for a proof. Essentially, we would like to find the decisions that minimize the cost of human computation. Such minimization is possible with small search space, as in the "in_group2" example. However, it is not always possible when the search space is large.

Crowd-Logic takes a best-first search strategy when solving a query: when there are multiple actions available for a nondeterministic predicate, it computes a heuristic for each action and try the best action first.

The challenge is to find good heuristics. Many good heuristics come from the domain knowledge. However, being a general programming language, the domain knowledge of the program is not readily available.

In the next section, we describe a lookahead algorithm, which is able to generate a partial probabilistic lookahead tree and compute some useful heuristics.

### A Lookahead Algorithm

The idea of our lookahead algorithm is to look ahead in the proof search tree so that the interpreter can estimate the cost of going down different paths. The more states it can look ahead, the better such estimation could be. The sub tree that the lookahead process has traversed is called the lookahead tree.

We use the quicksort algorithm as an example to introduce a variety of techniques that our lookahead algorithm employs to deal with human primitives and state explosion. Our quicksort implementation is shown in Figure 4. The predicate "le" is a human primitive that compares two items using human judgment. We use "choose(Xs, X)" to pick a pivot, which allows the interpreter to choose a better pivot to reduce the cost of human computation: when the interpreter encounters a state of choosing a pivot, it looks ahead in the proof search tree to compute heuristics for each pivot candidate. It then picks the best pivot and continues.

When the lookahead process tries to traverse the proof search tree from a state of choosing a pivot, it will quickly find many human primitives as illustrated in Figure 5a. The interpreter does not know whether a never-encountered human comparison, e.g., "le(H,X)", will return true or false before executing it. Stopping the lookahead process at these human comparisons will lead to bad heuristics because the lookahead tree is too restricted.

To address this problem, we introduce probabilistic edges. When a never-encountered human comparison is found in the lookahead process, we keep expanding the lookahead tree as if the comparison would return true and assign the probability of this comparison being true to the expanded edge. The lookahead tree in Figure 5a can be expanded into Figure 5b. Without any knowledge about the comparisons, we assign the probability of the edge to be 0.5. The probabilities allow us to estimate the average cost.

```
R1  qsort(Xs, Ys) :- qsort(Xs, Ys, []).
R2  qsort([], Ys, Ys).
R3  qsort(Xs, Ys, Zs) :-
       choose(Xs, X), remove(Xs, X, Ws),
       partition(X, Ws, As, Bs),
       qsort(Bs, Cs, Zs),
       qsort(As, Ys, [X|Cs]).
R4  partition(X, [], [], []).
R5  partition(X, [H|Xs], [H|As], Bs) :-
       le(H, X), partition(X, Xs, As, Bs).
R6  partition(X, [H|Xs], As, [H|Bs]) :-
       partition(X, Xs, As, Bs).
R7  remove([X|Xs], X, Xs).
R8  remove([Y|Xs], X, [Y|Ys]) :-
       X\=Y, remove(Xs, X, Ys).
```

Figure 4: Quicksort (using human judgment for comparison) implemented in Crowd-Logic.

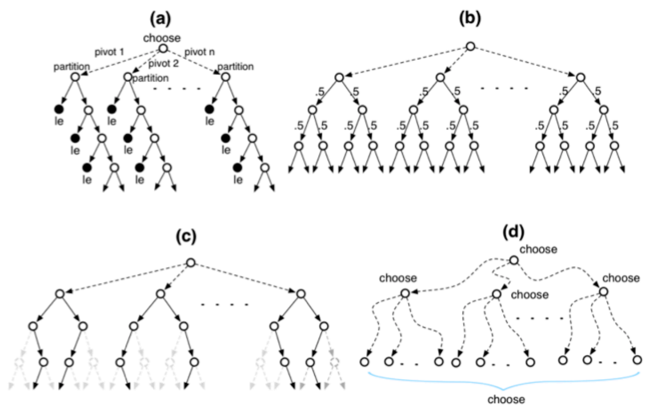



Figure 5: Lookahead trees: (a) Non-probabilistic. (b) Probabilistic. (c) Probabilistic with sampling. (d) States can still explode on hard problems.

```
@Classifier(key="le")
public double lessThan(
        BindingTable bt,
        CTerm a, CTerm b) {

    // Call the statistical machine learning
    // system to classify if a is less than b.
}
```

Figure 6: Annotation "@Classifier" to offer probability estimation for human comparisons that enables more informed sampling in the lookahead.

Though the probabilistic approach enables more states to be explored in the lookahead process, the ability to look down all the paths in the proof tree could generate a tree that grows exponentially. It soon becomes intractable to explore much deeper as illustrated in Figure 5b.

To address this problem, we limit the number of active paths that can be explored in the lookahead process under each choice state, e.g., pivot. If at any time the number of active paths exceeds this limit, we prune the least likely paths based on their probabilities. Figure 5c shows a limit of 2 active paths.

***Using Better Probabilities***

Since the sampling method is based on the probabilities of the paths, it will be more effective if we have a better prediction than 0.5 for the human primitives. The better prediction could also produce better heuristics.

One way to obtain a better prediction is to use a statistical machine learning system. For example, we can compute a set of features for a pair of two items and train a statistical classifier on the prior comparison results. We can then use the probability from the classification result.

Crowd-Logic allows developers to provide such probability via an annotated Java method, as shown in Figure 6. The developer implements a method that takes the same parameters as the actual implementation of the human primitive to be predicted. The method returns a probability. In order to inform the interpreter about the availability such prediction, the method needs to be annotated using "`@Classifier`" with the corresponding human primitive as "`key`". When the library is loaded, the interpreter will scan for such methods and use them to predict human primitives in the lookahead process.

***Bounded or Further Pruning***

However, even with the perfect prediction and sampling, the lookahead process still cannot explore all the paths.

In sorting, with perfect prediction, all the states for choosing pivots will still remain active in the lookahead process as illustrated in Figure 5d. There is still an exponential number of paths can be explored.

There two options under such situation, both are available in Crowd-Logic. One option is to stop the lookahead after certain depth of "`choose`" states. For example, in sorting, we can only look ahead the next three possible pivots.

The other option is to further prune down the size of tree by only keeping the best paths. The goodness of a path can be measured by their cost efficiency. Cost efficiency is the cost in human computation versus the progress it has contributed. The more progress has made with less human computation cost, the better the path is. In sorting, we can use the number of pivots have been picked as progress. However, in general, the progress information requires domain knowledge.

Crowd-Logic allows developers to provide the progress information in their logic programs. Developers can inform the interpreter about the new progress by using primitive "`progress`". The primitive "`progress`" takes a number as argument. The interpreter will track the total progress in each state. The meaning of the progress is up to developers. In our quicksort example, we can use the number of partitions have been made as progress by adding "`progress(1)`" after each "`partition`".

***Human Primitives with Output Parameters***

We now look at human primitives with output variables. Consider the primitive "`improve`" in *DescribeImage* example. In the lookahead process, there is little can be said about the new text that the human worker will produce before actually we actually run it. The lookahead bind these unknown human output variables process to placeholders. The placeholder saves the information about where its value comes from.

When the logic program requires computation on the placeholder, such as the length of a string and the value of an integer, the lookahead process will stop unless the developers can provide a Java method to compute it. We will see a concrete example in our validation section.

**Computing Heuristics from the Lookahead Tree**

After generating a probabilistic lookahead tree, the next step is to compute heuristics from it. Based on the heuristics, the interpreter will decide which path to take to solve the query.

The heuristic $f$ can be the average cost or average cost efficiency (cost/progress), which can be computed recursively. Let c(i) be the cost of the selected goal at i, and c'(i) be the cost of the sub-tree under state i. When $i$ is deterministic, we have:

$$c'(i) = c(i) + \sum_{pa(j)=i} \left( c'(j) \times p(i,j) \times \prod_{pa(k)=i,k<j} p_{fail}(k) \right)$$

$p(i,j)$ is the probability of the transition from $i$ to $j$, which can be less than $1$ when the selected goal at $i$ is a human primitive. $p_{fail}$ is the probability that a proof cannot be found under that state. When $i$ is deterministic, its children have a linear order. $k$ is a left sibling of $j$. $p_{fail}$ can be computed recursively by:

$$p_{fail}(i) = \prod_{pa(j)=i} p_{fail}(j) \times p(i,j)$$

When $i$ is nondeterministic, $c'(i)$ and $p_{fail}(i)$ can take the value of its best child:

$$c'(i) = c'(j) \wedge p_{fail}(i) = p_{fail}(j), j = \underset{pa(k)=i}{\operatorname{argmax}}(f(k))$$

The function f is the heuristic function. The average progress can be computed similarly as the average cost.

## IMPLEMENTATION

Crowd-Logic is implemented in Java. It implements a subset of standard Prolog. It uses Java DB (Derby) to manage its local databases. All the human primitives from the examples are also implemented in Java. In the evaluation, the human tasks are implemented on MTurk using its Java SDK to manage HITs and Amazon S3 to store question HTML.

## VALIDATION

In this section, we validate Crowd-Logic and its optimization techniques by implementing two human computation algorithms, sorting a list of reviews using human judgment and shortening text using human rewrites.

### Quicksort

The first algorithm is sorting a list of reviews by their sentiments using the quicksort algorithm. We want to validate that our optimization is able to optimize the control flow to reduce the cost of human computation when there are opportunities for reuse and better prediction for human primitives.

We have already examined the quicksort implementation in Crowd-Logic in the previous sections (and in Figure 4). For comparison, a similar imperative implementation can be found in [9]. The implementation of the human primitive "`le`" can be as simple as asking three people if one review is more positive than the other. A classifier can be learned from a set of prior comparison results. However, what is more interesting about this quicksort algorithm is how much our optimization algorithm can reduce the cost of human computation, i.e., the number of human comparisons.

More specifically, we examine how the quicksort algorithm performs with our optimization under different conditions: the combinations of different amounts of re-usable data, e.g., 0%, 20%, 40%, 60% and 80% of the total comparisons, and classifiers with different accuracies, e.g., 0.5 (a random guess), 0.8 and 1.0 (an oracle), and a baseline condition where the quicksort algorithm always chooses the first element as the pivot. Reuse is enabled for all conditions.

We conduct an experiment on sorting a list of 20 reviews. The experiment contains a total of 100 runs. In each run, the list is first randomized and then sorted by the quicksort algorithm across all conditions. The percentage of total amount reduction in human comparisons from the baseline is computed for each condition and then averaged over the 100 runs.

For the purpose of our experiment, we simulate the human comparison by assuming a total order of the reviews. We also simulate the predictions from the classifier. For a classifier of accuracy 0.8, we first compute a distorted order of the reviews from their total order so that only 80% of the comparisons between two reviews remain the same. The probability that one review is better than the other is 0.8 if it is consistent with the distorted order and 0.2 otherwise.

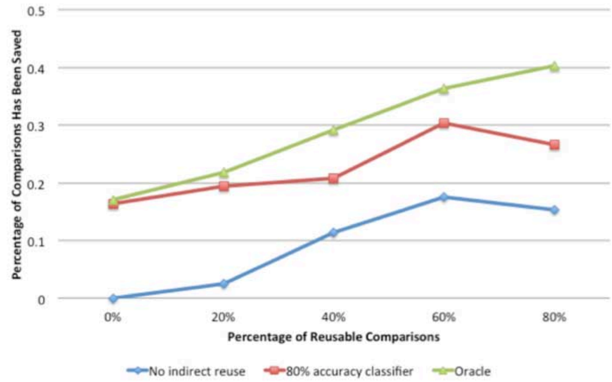



Figure 7 The percentage of comparisons saved under different conditions compared with the baseline: always using first element as pivot.

The result of the experiment is shown in Figure 7. The more prior results that are available or the more accurate the classifier is, the more reduction can be gained from the optimization. The accuracy of 0.8 is quite common between machine learning systems, yet it is almost as helpful as the oracle. With a common classifier and some prior results, the optimization algorithm can produce 20% to 30% reductions. When no prior results can be reused, a common classifier can still give us about 15% reductions, and when no classifier is available, a random guess can still give us about 15% reductions. These results validates that our optimization algorithm is effective.

### Shortening Text

We want to validate that our optimization is able to

```
R0  shorten_ranges(Paragraph, ShortenLength, Ranges, []) :- ShortenLength =< 0.

R1  shorten_paragraph(Paragraph, ShortenLength, Patches) :-
      find(Paragraph, Ranges),
      shorten_ranges(Paragraph, ShortenLength, Ranges, Patches).
R2  shorten_ranges(Paragraph, ShortenLength, [], []).
R3  shorten_ranges(Paragraph, ShortenLength, Ranges, [patch(From, Length, Text)|Patches]) :-
      choose(Ranges, range(From, Length)),
      remove(Ranges, range(From, Length), NewRanges),
      shorten_verify(5, Paragraph, From, Length, Text),
      TextLength is length(Text),
      progress(Length - TextLength),
      shorten_ranges(Paragraph, ShortenLength - Length + TextLength, NewRanges, Patches).
R4  shorten_verify(0, Paragraph, From, Length, substring(Paragraph, From, Length)).
R5  shorten_verify(Trial, Paragraph, From, Length, Text) :-
      Trial > 0, fix(Paragraph, From, Length, Text), verify(Paragraph, From, Length, Text),
R6  shorten_verify(Trial, Paragraph, From, Length, Text) :-
      Trial > 0, shorten_verify(Trial-1, Paragraph, From, Length, Text).
```

Figure 8: The logical representation of *ShortenText* in Crowd-Logic

```
@Estimator(key="shorten")
public Term estimate(
        BindingTable bt, CTerm arg) {
    // If arg is 'length(placeholder)', where
    // the placehoder is for the output text
    // from fix,
    // return the original length * 4 / 5.
}
```

Figure 9 An annotated Java method that could provide additional information about how to compute with a placeholder.

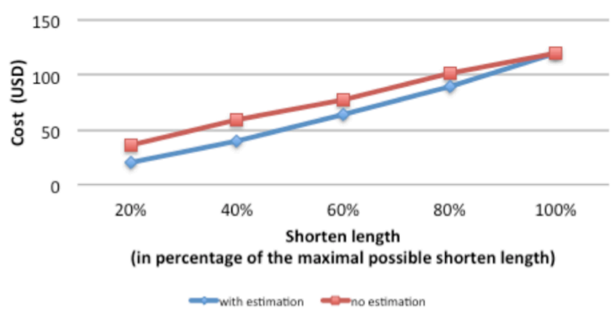



Figure 10 Cost when only a certain percentage of the maximal shorten length is required

optimize the control flow to reduce the cost of human computation when the estimation of human answers is available.

Soylent [3] demonstrates that human computation can be used to shorten text paragraphs without changing their meanings by using a proper workflow (Find-Fix-Verify). We implemented *ShortenText* to accomplish the same task with the same workflow as in Soylent. The logic programming part of *ShortenText* is shown in Figure 8. There are three human primitives: "`find`", "`fix`", and "`verify`". The human computation is realized via MTurk.

In the task of shortening a text, progress can be represented by how much has been shortened. Thus, in *ShortenText*, when a rewrite is verified, the change in length is reported as progress through primitive "`progress`".

In Soylent, although the interface provides a slider to let the user control how much to be shortened, the algorithm for shortening is unaware of the how much to be shortened. It always tries to find all possible rewrites even when the user only wants to shorten text by a couple of words. While the original algorithm can be modified to take into account how much to be shortened, we show that Crowd-Logic allows this to be done easily: by adding **R0**. The interpreter will terminate once the paragraph is short enough.

The interpreter is also able to prioritize which range to shorten first with the estimation about how much a range can be shortened. The estimation can be provided via an annotated Java method as shown in Figure 9. We provide a simple estimation that the length of the new text is 80% of the original text length.

To validate the optimization, we run *ShortenText* on 50 paragraphs from TOEFL writing test. For our purpose (to validate the optimization), we use simulation for "`find`", "`fix`" and "`verify`", rather than asking human to perform the tasks, as the purpose our experiment is not to validate if text can be shortened using human computation which has been validated in Soylent [3].

To simulate "`find`" we randomly generate 4 to 8 text ranges with varying length from 10 to 50. To simulate "fix", we generate a string with a random length between 50% and 90% of the original length. To simulate "verify", we assume that probability of a good answer to be 70%. We assume "fix" costing 0.05 USD and "verify" costing 0.12 USD (three votes, each 0.04 USD).

We only measure the cost of "`fix`" and "`verify`" with and without estimation. ("`find`" is the always same under both conditions) and vary the lengths to be shortened: 20%, 40%, 60%, 80% and 100% of the max possible shortening length. The result is shown in Figure 10. It validates that the optimization is able to better optimize the cost of human computation with the help of estimation.

## RELATED WORK

Multiple tools have been proposed for making programming human computation easier. MTurk provides a basic API [2] for managing human tasks on its crowdsourcing platform. TurKit [9] provides a thin Java/JavaScript API wrapper around MTurk API. It also introduces a crash-and-rerun programming model to avoid running redundant human tasks when re-running a program. However, since it is implemented by restoring the program state, unlike Crowd-Logic, it does not actually reuse redundant human tasks. The same human tasks will be run again if the program or the data input changes.

CrowdForge [8] is a general purpose framework for crowdsourcing complex tasks by splitting and recombining complex human computation. Jabberwocky [8] provides a similar framework, ManReduce. In addition, it implements a human and machine resource management system and a high-level procedural programming language for programming in ManReduce. Crowd-Logic allows a human computation program to be built on any crowdsourcing platform including Jabberwocky, and is not designed towards any particular framework.

The declarative approach to human computation has received much interest in the database research. CrowdDB [1] is a database system that processes queries using human computation. Both the queries and data models are specified in the declarative language SQL. Deco [7] is a similar system that uses a simple extension to SQL to pose queries. Qurk [14] is another SQL query system enables human-based processing. All these systems enable reuse of prior human computation as in Crowd-Logic. However, unlike the SQL-like query languages, the logic programming language in Crowd-Logic allows developers to implement general human computation algorithms.

Much work has explored optimization techniques for human computation tasks. TurKontrol [10,11] is a planner that uses a decision-theoretical model to learn and optimize the iterative improvement workflow for better utility. Marcus A., Wu E. and et al. [5,6] studied the various algorithms for sorting and joining in Qurk. CrowdScreen [10] studied the optimization problem of filtering data based on a set of human-verifiable properties. In contrast, Crowd-Logic is agnostic to the tasks to be optimized. The input to our optimization process is the actual code that specifies the algorithm.

## DISCUSSION

### The Boundary of Logic Programming

In our hybrid framework, the separation of what is implemented in logic programming and what is implemented in imperative programming is not strictly defined. It is up to the developers to decide where to set the boundary.

In the *DescribeImage* example, the implementation of the primitive "`is_better`" computes the majority from multiple human votes. It hides the actual voting process from the iterative improvement workflow. An alternative is to specify the voting process in logic rules and implement a single vote as a human primitive in imperative language.

There are two obvious extremes here. On one side, a developer can implement the entire human computation algorithm using imperative programming as a single human primitive. Then it loses all the benefits from Crowd-Logic, e.g., reusing human computation and optimization cost. On the other side, a developer can put as much as they can into the logic programs and only implements basic API to the crowdsourcing platforms in imperative programs. This would cause a much larger state space in proof search tree and degrade the performance of the optimization process. Both extremes are not good practices.

### Parallelism

Crowd-Logic did not look at enabling and optimizing for parallelism in workflows. However, logic programs offer intrinsic parallelism and parallel logic programming is one of the heavily explored topics in logic programming [13]. An easy approach is to find all the independent goals in the search state and then search them concurrently. More sophisticated approach is to use lookahead to optimize for the speed rather cost, and find a much larger set of human computations to run in parallel. These suggest an avenue of future work for Crowd-Logic.

### Reuse and Pure Logic Programming

The reuse of human computation is essential to the declarative approach. Consider the implementation of "an input agreement": an answer is agreed when two people have both verified. One plausible specification can be the following rule (where "`verify`" is a human primitive that asks a person to verify a given answer):

```
agree2(A) :- verify(A), verify(A).
```

In Crowd-Logic, the above rule will not behave as expected because the second verify will always directly reuse the first "`verify`". This is consistent with its logical interpretation: $P \wedge P = P$. We can introduce side effect to human primitives as in impure logic programming languages, however, but then `agree2` cannot be reused. A better option here is to implement `agree2` as a human primitive. However, if the developer has to reuse the individual "`verify`", it still can be done:

```
agree2(A) :-
  verify(A), verify(B), id(A) \= id(B), A=B.
```

"`id`" is the id of an answer and both the inequality of the ids of A and B and the equality of A and B are interpreted as constraints in our constraint solver.

### Indirect Reuse to Replace Human Computation

Crowd-Logic only uses machine prediction as guidance to optimize the control flow. All results from the human primitives are still obtained from the human labor. This is less ideal when a highly accurate machine learning system is available. One easy approach is to monitor the quality of machine results with human results, and replace human when the quality is on par. More complicated approach is to mix human computation with machine results to achieve higher quality, lower monetary cost, and faster. This also can be one of our future directions.

## CONCLUSION

We presented Crowd-Logic, a tool for implementing and optimizing human computation algorithms using logic programming. It enables maximally reusing prior human computation results by using constraint solving. We also explored adding nondeterministic choices to the logic programming and optimizing control flows on these choices to reduce the general human cost in human computation algorithms.

## REFERENCES

1. Ahmad, S., Battle, A., Malkani, Z., and Kamvar, S. The jabberwocky programming environment for structured social computing. *Proc. UIST 2011*, 53-64.

2. Amazon Mechanical Turk Documentation. http://aws.amazon.com/documentation/mturk/.

3. Bernstein, M.S., Little, G., Miller, R.C., et al. Soylent : A Word Processor with a Crowd Inside. *Proc. UIST 2010*, 313-322.

4. Bigham, J.P., Jayant, C., Ji, H., et al. VizWiz: nearly real-time answers to visual questions. *Proc. UIST 2010*, 333-342.

5. Dai, P., Mausam, and Weld, D.S. Decision-Theoretic Control of Crowd-Sourced Workflows. *AAAI 2010*, 1168-1174.

6. Dai, P., Mausam, and Weld, D.S. Artificial Intelligence for Artificial Artificial Intelligence. *AAAI 2011*, 1153-1159.

7. Franklin, M.J., Kossmann, D., Kraska, T., Ramesh, S., and Xin, R. CrowdDB: answering queries with crowdsourcing. *Proc. SIGMOD 2011*, 61-72.

8. Kittur, A., Smus, B., Khamkar, S., and Kraut, R.E. Crowdforge: Crowdsourcing complex work. *Proc. UIST 2011*, 43–52.

9. Little, G., Chilton, L.B., Goldman, M., and Miller, R.C. TurKit : Human Computation Algorithms on Mechanical Turk. *Proc. UIST 2010*, 57-66.

10. Marcus, A., Wu, E., Karger, D., Madden, S., and Miller, R. Human-powered sorts and joins. *Proceedings of the VLDB Endowment 5*, 1 (2011), 13-24.

11. Marcus, A., Wu, E., Karger, D.R., Madden, S., and Miller, R.C. Demonstration of Qurk: A Query Processor for Human Operators. *Proc. SIGMOD 2011*, 1315-1318.

12. Noronha, J., Hysen, E., Zhang, H., and Gajos, K.Z. PlateMate : Crowdsourcing Nutrition Analysis from Food Photographs. *Proc. UIST 2011*.

13. Parameswaran, A., Garcia-molina, H., Park, H., and Widom, J. CrowdScreen : Algorithms for Filtering Data with Humans. *Proc. SIGMOD 2012*.

14. Parameswaran, A., Park, H., Garcia-Molina, H., Polyzotis, N., and Widom, J. *Deco: Declarative Crowdsourcing*. Stanford InfoLab, 2011.

15. Sterling, L. and Shapiro, E. *The Art of Prolog: Advanced Programming Techniques*. The MIT Press, 1986.