AUTOMAN: A Platform for Integrating Human-Based and Digital Computation

Daniel W. Barowy Emery D. Berger Andrew McGregor

Department of Computer Science
University of Massachusetts, Amherst
Amherst, MA 01003
{dbarowy,emery,mcgregor}@cs.umass.edu

Abstract

Humans can perform many tasks with ease that remain difficult or impossible for computers. Crowdsourcing platforms like Amazon's Mechanical Turk make it possible to harness human-based computational power on an unprecedented scale. However, their utility as a general-purpose computational platform remains limited. The lack of complete automation makes it difficult to orchestrate complex or interrelated tasks. Scheduling human workers to reduce latency costs real money, and jobs must be monitored and rescheduled when workers fail to complete their tasks. Furthermore, it is often difficult to predict the length of time and payment that should be budgeted for a given task. Finally, the results of human-based computations are not necessarily reliable, both because human skills and accuracy vary widely, and because workers have a financial incentive to minimize their effort.

This paper introduces AUTOMAN, the first fully automatic *crowdprogramming* system. AUTOMAN integrates human-based computations into a standard programming language as ordinary function calls, which can be intermixed freely with traditional functions. This abstraction allows AUTOMAN programmers to focus on their programming logic. An AUTOMAN program specifies a confidence level for the overall computation and a budget. The AUTOMAN runtime system then transparently manages all details necessary for scheduling, pricing, and quality control. AUTOMAN automatically schedules human tasks for each computation until it achieves the desired confidence level; monitors, reprices, and restarts human tasks as necessary; and maximizes parallelism across human workers while staying under budget.

1. Introduction

Humans can perform many tasks with ease that remain difficult or impossible for computers. For example, humans are far better than computers at performing tasks like vision, motion planning, and natural language understanding. Most researchers expect these "AI-complete" tasks to remain beyond the reach of digital computers for the foreseeable future [16].

Recent systems have streamlined the process of hiring humans to perform computational tasks. The most prominent example is Amazon's Mechanical Turk, a general-purpose *crowdsourcing* system that acts as an intermediary between labor requesters and workers [1, 9]. Employers now use Mechanical Turk to perform jobs like image classification and accurate audio transcription.

However, harnessing of human-based computation at scale faces the following challenges:

- **Difficult to scale up complexity.** Current crowdsourcing platforms make it possible to create many standalone tasks, but lack support for interrelated or iterative tasks.
- **Hard to determine pay and time for tasks.** Employers must decide in advance the time alloted to a task and the payment for successful completion. It is both difficult and important to choose these correctly: workers will not accept jobs whose deadline is too short or where the pay is too low.
- **Scheduling complexities.** Employers must manage the tradeoff between latency (humans are relatively slow) and cost (more workers means more money). Because workers may fail to complete their tasks in the alloted time, jobs need to be tracked and reposted as necessary.
- **Low quality responses.** Finally, human-based computations always need to be checked: worker skills and accuracy vary widely, and they have a financial incentive to minimize their effort. Manual checking does not scale, and voting is inadequate, since workers may agree by random chance.

Contributions

This paper introduces AUTOMAN, a programming system that integrates human-based and digital computation. AUTOMAN addresses the challenges of harnessing human-based computation at scale:

Transparent integration of human and digital computation. AUTOMAN incorporates human-based computation as function calls in a standard programming language. The AUTOMAN runtime system transparently manages scheduling, budgeting, and quality control.

Automatic scheduling and budgeting. The AUTOMAN runtime system schedules tasks to maximize parallelism across human workers while staying under budget. AUTOMAN tracks job progress, rescheduling failed tasks and repricing jobs as needed.

Automatic quality control. The AUTOMAN runtime system performs automatic quality control management. AUTOMAN automatically creates enough human tasks for each computation to achieve the confidence level specified by the programmer.

For example, given a desired confidence level of 95% and a function with five possible answers, AUTOMAN initially schedules at least three tasks (human workers). Because the chances of all

three agreeing due to random chance is under 5%, a unanimous response would be considered acceptable. If all three workers do not agree, AUTOMAN will schedule another three tasks, at which point 5 out of 6 must agree to achieve a 95% confidence level.

Outline

The remainder of this paper is organized as follows. Section 2 presents background information on crowdsourcing platforms. Section 3 provides an overview of AUTOMAN's operation with a sample program. Section 4 describes the algorithms AUTOMAN uses to perform scheduling, budgeting, and quality control, and Section 6 provides details of AUTOMAN's software architecture. Section 7 presents an overview of our experience with AUTOMAN. Finally, Section 8 discusses related work, Section 9 describes planned future work, and Section 10 concludes.

2. Background: Crowdsourcing Platforms

Since, crowdsourcing is a novel application domain for programming language research, we start by summarizing the necessary background on crowdsourcing platforms. Our discussion in this section focuses on Amazon's Mechanical Turk, but other existing crowdsourcing platforms are similar.

Mechanical Turk acts as an intermediary between employers (known as *requesters*) and employees (*workers*, or colloquially, *turkers*) for short-term assignments.

Human Intelligence Tasks (HITs). In Mechanical Turk parlance, individual tasks are known as *HITs*, which stands for *human intelligence tasks*. HITs include a short description, the amount the job pays, and other details. Most HITs on Mechanical Turk are for relatively simple tasks, such as "does this image match this product?" Compensation is also correspondingly small, since employers expect that work can be completed on a time scale spanning from seconds to minutes. Pay for HITs range from a single penny to several dollars.

Each HIT is represented as a question form, composed of any number of questions, and associated metadata such as a title, description, and search keywords. Questions can be one of two types: a free text question, where workers can provide a free-form textual response, or a multiple-choice question, where workers make one of more selections from a list of possible options. We refer to the former as *open-ended questions* and the latter as *closed-ended questions*; AUTOMAN currently only supports closed-ended questions.

Requesters can also use Mechanical Turk as a transaction manager, allowing a third-party website to control the interaction with the worker by embedding the site in a browser IFrame. This third option is typically reserved for interactive tasks.

Requesters: Posting HITs. Mechanical Turk allows HITs to be posted manually, but also exposes a Web service API that allows basic details of HITs to be managed programmatically [1], including posting HITs, collecting completed work, and paying workers. Using this API, it is straightforward to post similar tasks to Mechanical Turk *en masse*. HITs sharing similar qualities can be grouped into *HIT groups*.

A requester may also instruct Mechanical Turk to parallelize a particular HIT by indicating whether each HIT should be assigned to more than on worker. By increasing the number of assignments, Mechanical Turk allows additional workers to accept work for the same HIT, and the system ensures that the parallel workers are unique (i.e., that a single worker cannot complete the same HIT more than once).

Workers: Performing Work. Mechanical Turk workers can choose any of the available tasks on the system for which they are qualified (see below): as of this writing, there are approximately 275,000 HITs posted. When workers choose to perform a particular HIT, they accept an *assignment*, which grants them a time-limited reservation for that particular piece of work; that is, no other worker may accept it.

HIT Expiration. HITs have two timeout parameters: the amount of time that a particular HIT should remain in the listings, known as the *lifetime* of a HIT, and the amount of time that a worker has to complete an assignment once it is accepted, known as the *duration* of an assignment. If after accepting an assignment for a HIT, a worker exceeds the assignment's duration without submitting completed work, the reservation is cancelled, and the work is returned to the pool of available assignments. If a HIT reaches the end of its lifetime without its assignments having been completed, the HIT expires and is removed from the job board.

Requesters: Accepting or Rejecting Work. Once a worker completes and submits an assignment, the requester is notified. The requester then can accept or reject the completed assignment. Acceptance of an assignment indicates that the completed work is satisfactory, and the worker is then automatically paid for his or her efforts. Rejection withholds payment, and the requester, if so inclined, may provide a textual justification for the rejection. (AUTOMAN automatically manages acceptance and rejection; see Section 3.2.)

Worker Quality Control. A key challenge in automating work in Mechanical Turk is attracting and retaining good workers, or at least discouraging bad workers from participating. However, Mechanical Turk provides no way for requesters to seek out specific workers.

Instead, Mechanical Turk provides a *qualification* mechanism to limit which workers may perform a particular HIT. For example, a common qualification is that workers must have an overall assignment-acceptance rate of 90%. However, given the wide variation in tasks on Mechanical Turk, overall worker accuracy is of limited utility.

For example, the fact that a worker who is skilled in and favors audio transcription tasks may have a high accuracy rating, but there is no reason to believe that this worker can also perform Chinese-to-English language translation tasks. Worse, workers who cherry-pick easy tasks and thus have a high accuracy rating actually may be less qualified than a worker who routinely performs difficult work that is occasionally rejected.

3. AUTOMAN Overview

AUTOMAN is a domain-specific language embedded in Scala [14]. Figure 1 presents a complete AUTOMAN program. The goal of AUTOMAN is to abstract crowdsourcing so that the important parts of a program's logic can be understood clearly, and so that large-scale crowdsourcing applications can be built without a programmer having to worry about complicated human interaction logic.

3.1 Using AUTOMAN

Figure 1 presents an example AUTOMAN program. The program "computes" which of a set of cartoon characters does not belong in the group. Notice that the programmer does not specify details about the chosen crowdsourcing backend (Mechanical Turk) except for account credentials.

Crucially, all details of crowdsourcing are hidden from the AUTOMAN programmer. The AUTOMAN runtime, discussed below, manages budgeting for the cost and time limits of the computation, task validation, or even what it means to have confidence in a particular computational result.

***Initializing* AUTOMAN.** After importing the AUTOMAN and Mechanical Turk adapter libraries, the first thing an AUTOMAN programmer does is to declare a configuration for the desired

```
import edu.anon.cs.automan.core._
import edu.anon.cs.automan.MTurk._

object WhichOneNotBelongSimple {
  def main(args: Array[String]) {

    // AutoMan configuration for MTurk:
    val config = MTurkConfig { c =>
      c.access_key_id     = "XXXX" // account info
      c.secret_access_key = "XXXX"
    }

    // Set up AutoMan parameters.
    val a = Automan { automan =>
      automan.budget = 8.00         // dollars
      automan.config = config       // declared above
    }

    // Define a human function.
    val WhichOne = a.Task[String] { t =>
      t.confidence = 0.95           // the default
      t.title = "Which one of these doesn't belong?"
      t.description = t.title
      t.question = a.MultipleChoiceQuestion(
        question_text = t.title,
        selection_texts =
          Map('oscar -> "Oscar the Grouch",
              'kermit -> "Kermit",
              'spongebob -> "Spongebob Squarepants",
              'cookie -> "Cookie Monster",
              'count -> "The Count")
      )
    }

    // Call the human-based function.
    val fd = WhichOne()

    // Start execution and print result.
    a.run()
    println(fd.value)
  }
}
```

Figure 1. A complete AUTOMAN program. This program computes, by invoking humans, which cartoon character does not belong in a given set. The AUTOMAN programmer specifies only credentials for Mechanical Turk, an overall budget, and the question itself; the AUTOMAN runtime manages all other details of execution (scheduling, budgeting, and quality control).

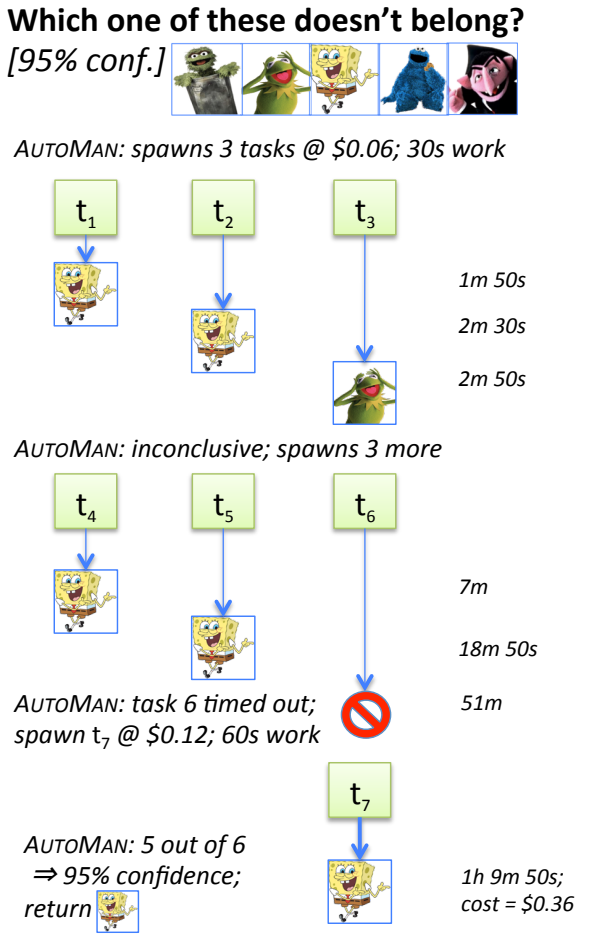

Figure 2. An actual execution of the example program on the left, with time advancing from top to bottom. AUTOMAN first spawns 3 tasks, which (for five choices) suffice to reach a 95% confidence level if all workers agree. Since the 3 workers do not agree, AUTOMAN doubles the number of tasks (5 of 6 must now agree). When task 6 times out, AUTOMAN spawns a new task and doubles both the budget and time allotted. Once task 7 completes, AUTOMAN returns the result.

crowdsourcing platform. This step provides implementations for AUTOMAN's platform-specific abstract classes. The programmer then instantiates and initializes an AUTOMAN runtime object, which is tied to the crowdsourcing system platform configuration.

Specifying* AUTOMAN *functions. Specifying a function in AUTOMAN differs considerably from defining a function in a conventional programming language. In AUTOMAN, functions are formulated as questions that the workers must answer.

Confidence level. An AUTOMAN programmer can optionally specify the degree of confidence they want to have in their computation, on a per-function basis. AUTOMAN's default confidence is 95% (0.95), but this can be overridden as needed.

Metadata and question text. Each question requires a title and description, used by the crowdsourcing platform's user interface. Here, these fields map to Mechanical Turk's fields of the same name. A question also includes a textual representation of the question, together with a map between symbolic constants and strings for possible answers.

Question variants. AUTOMAN supports two types of multiple-choice questions: questions where only one answer is correct ("radio-button" questions), or where any number of answers may be correct ("checkbox" questions). Section 4.3 describes how AUTOMAN's quality control algorithm handles these different types of questions.

Invoking a function. An AUTOMAN programmer can invoke a function as if it were any ordinary (digital) function. Here, the programmer calls the just-defined function `WhichOne` with no input. The function returns a FutureData object, which can be passed to other AbstractTasks in an AUTOMAN program before the human computation is complete.

***Running* AUTOMAN.** To invoke AUTOMAN, the programmer simply calls the `run()` function. The program submits its jobs, and eventually prints the result. AUTOMAN returns the selected item from the set of symbols defined in the question's definition.

3.2 AUTOMAN Execution

Figure 2 depicts an actual trace of the execution of the program from Figure 1, obtained by executing it with Amazon's Mechanical Turk. This example demonstrates that ensuring valid results even for simple programs can be complicated.

Starting Tasks. At startup, AUTOMAN examines the form of the question field defined for the task and determines that, in order to achieve a 95% confidence level for a question with five possible choices, at minimum, it needs three different workers to unanimously agree on the answer (see Section 4.3). AUTOMAN then spawns three tasks on the crowdsourcing backend, Mechanical Turk. To eliminate bias caused by the position of choices, AUTOMAN randomly shuffles the choices in each task.

AUTOMAN's default strategy is optimistic. For many tasks, human workers are likely to agree unanimously. Whenever this is true, AUTOMAN saves money by spending the least amount required to achieve the desired statistical confidence.

However, AUTOMAN also allows users to choose a more aggressive strategy that trades increased cost for reduced latency; see Section 4.2.

Quality Control. At 1 minute, 50 seconds, worker 1 accepts the task and submits "Spongebob Squarepants" as the answer. At 2 minutes, 30 seconds, worker 2 accepts the task and submits the same answer. However, at 2 minutes, 50 seconds, worker 3 accepts the task and submits "Kermit". In this case, AUTOMAN's optimism did not pay off, since worker 3's answer does not agree with worker 1 and 2. Because this result is inconclusive, AUTOMAN doubles the number of tasks.

Doubling is a simple but effective strategy because at worst, AUTOMAN will spawn no more than twice the amount of tasks actually needed for a valid computation. At this point, AUTOMAN recomputes the minimal number of agreements, which turns out to be five out of six. The Appendix presents a full derivation of the closed-form formulas that AUTOMAN uses to compute these values.

Note that doubling may, by providing added incentive to workers, speed the overall computation.

Initial Time Estimates. When defining a task, the programmer can indicate how much time they expect the worker to spend on working on a task. On Mechanical Turk, this number serves as an indication to the worker of the difficulty of the task. In AUTOMAN, this figure is set to 30 seconds by default.

A second time parameter, the "lifetime" of a task, indicates how long the crowdsourcing backend should keep the task around without any response from workers. In AUTOMAN, the task lifetime is set to 100 times the timeout.

At 51 minutes into the computation, task 6 reaches its lifetime expiration. Since AUTOMAN does not yet have a statistically valid result, it reschedules the task, this time extending both the task timeout and task lifetime by a factor of two.

Rebudgeting: Time and Pay. AUTOMAN does not require the programmer to specify exactly how much each worker should be paid. AUTOMAN currently uses the timeout parameter and calculates the base cost for the task using the US Federal minimum wage. For 30 seconds' worth of work at the current minimum wage, the result is $0.06 US dollars.

AUTOMAN automatically manages worker compensation, using the same doubling strategy to find the wage that the worker marketplace is willing to accept (see Section 4.1). In the absence of an automatic mechanism, programmers would be required to determine the fair wage of the task marketplace manually. Given the subjectivity of a "fair wage", knowing the appropriate wage *a priori* is complex at best.

At the same time that AUTOMAN extends the time allowed to work on a task, it also increases the pay. AUTOMAN's first 6 tasks were spawned with a reward of $0.06; after the timeout, the compensation for the seventh task was set to $0.12.

Automatic Task Acceptance and Rejection. One hour and 9 minutes into the computation, a worker submits a sixth answer, "Spongebob Squarepants". AUTOMAN again examines whether the answers agree and it finds that 5 out of the 6 answers agree. AUTOMAN can now reject the null hypothesis, i.e., that 5 workers agreed by choosing the same answer randomly, with a 95% confidence. The runtime system then returns the answer to the program, and the user's regular program resumes.

AUTOMAN then informs the crowdsourcing backend to pay the five workers whose answers agreed. Four workers are paid $0.06, and one is paid $0.12. The one worker whose answer did not agree was not paid. For Mechanical Turk, which supports rejection notifications, we inform the worker that their work was not accepted, and why not.

The fact that AUTOMAN does not pay for incorrect work reduces its cost, especially as the number of workers increases. For example, for a question with 5 choices, only 7 of 12 workers need agree to achieve a 95% confidence level. As the number of workers increases, the proportion required for agreement drops further, making rejecting incorrect work even more desirable.

```
1  // default parameters
2  confidence  = 0.95
3  MAX_WORKERS = 30
4  DEF_ASSN_TIMEOUT = 30 seconds
5  DEF_HIT_TIMEOUT  = DEF_ASSN_TIMEOUT * user_constant
6
7  // initial # workers, reward, and timeout
8  n = unanimous_n(confidence, num_choices)
9  min_agree = n
10 reward = minimum_wage_for(DEF_ASSN_TIMEOUT)
11 timeout = DEF_HIT_TIMEOUT
12
13 // adjust for time_value (cost/latency tradeoff)
14 remaining_n = n * max(1,time_value/minimum_wage)
15
16 // scheduler loop
17 while (remaining_n > 0 && n < MAX_WORKERS) {
18   // create new tasks as needed
19   spawn_task (n, reward, timeout, DEF_ASSN_TIMEOUT)
20   wait until (timedout || got_all_answers)
21   if (timedout) {
22     // double pay and time
23     reward *= 2
24     timeout *= 2
25     remaining_n -= answers.size
26   } else {
27     if (agree(answers) < min_agree) {
28       // insufficient responses;
29       // double tasks and compute new threshold
30       remaining_n *= 2
31       min_agree = min_thresh(remaining_n, confidence)
32     }
33   }
34 }
35 if (n >= MAX_WORKERS) {
36   FailedComputationException;
37 } else {
38   // Success.
39   return argmax(answers)
40 }
```

Figure 3. Pseudo-code for AUTOMAN's scheduling loop, which handles posting and re-posting jobs, budgeting, and quality control; the Appendix includes a derivation of the formulas for the quality control thresholds.

4. AUTOMAN Scheduler

Figure 3 presents pseudo-code for AUTOMAN's main scheduler loop, which comprises the algorithms that the AUTOMAN runtime uses to manage scheduling, budgeting price and time, and quality control.

4.1 Budgeting: Time and Pay

AUTOMAN uses a doubling strategy to manage both the time allotted to individual tasks, and their pay for successful completion. AUTOMAN's overriding goal is to find workers willing to perform the task. When not enough workers have accepted tasks, it may be either because the pay is too low or the time is too short. AUTOMAN punts on this distinction and simply doubles both the time allotment and the pay.

This simple strategy both works well in practice and is straightforward to analyze. Suppose the ideal amount of pay required to entice workers to perform a task is t. In the worst case, AUTOMAN will double an amount that was just slightly below t (e.g., $t - \epsilon$), and thus pay no more than twice as much as optimal.

The doubling strategy raises the prospect of a worker "gaming" the computation into paying a large sum of money for an otherwise simple task. However, even if all workers adopted the same strategy, market dynamics would prevent the amount of money from spiraling upward. As soon the task reaches an acceptable wage for some proportion of the worker marketplace, workers will accept the task. Forcing AUTOMAN to continue doubling to a very high wage would require collusion between workers on a scale that we believe is infeasible.

4.2 Trading Off Latency and Money

AUTOMAN's default strategy for spawning tasks is optimistic: it creates the smallest number of tasks required to reach the desired confidence level in the hope that the results will be unanimous.

However, AUTOMAN also allows programmers to specify a *time-value* for the computation. This time-value denotes the value of the programmer's time, rather than the wage paid to workers. AUTOMAN multiplies the number of tasks initially spawned by the ratio of the time-value to the minimum wage, and thus creates a far larger number of initial jobs. If enough jobs complete to achieve statistical confidence, AUTOMAN cancels any outstanding jobs.

This strategy runs the risk of paying substantially more for a computation, but can yield dramatic reductions in latency. We reran the example program given in Figure 1 with a time-value set to \$50, which is 7× larger than the current U.S. minimum wage. In two separate runs, the computation completed in 68 and 168 *seconds*; we also ran the first computation with the default time-value (minimum wage), and those computations took between 1 and 3 *hours* to complete.

4.3 Quality Control

AUTOMAN's quality control algorithm is based on collecting enough consensus for a given question to rule out the possibility (with a desired level of confidence) that the results are due to random chance. Section 5 justifies this approach.

Initially, AUTOMAN spawns just enough workers to meet the desired confidence level if they all agree. Figure 4 provides a graphical depiction of the initial confidence level function. Computing this value is straightforward: if k is the number of choices, and n is the number of tasks, the confidence level reached is $1 - k(1/k)^n$. AUTOMAN simply finds the lowest value of n where the desired confidence level is reached.

Variants of Multiple Choice Questions. For ordinary multiple choice questions where only one choice is possible ("radio-button" questions), k is exactly the number of possible answers. For multiple choice questions with c choices and any or all may be chosen ("checkbox" questions), k is much larger: $k = 2^c$.

For these questions, k is so high that a very small number of workers is required to reject the null hypothesis (random choice). However, it is reasonably likely that two lazy workers will simply select no answers, and AUTOMAN will erroneously accept that answer is correct.

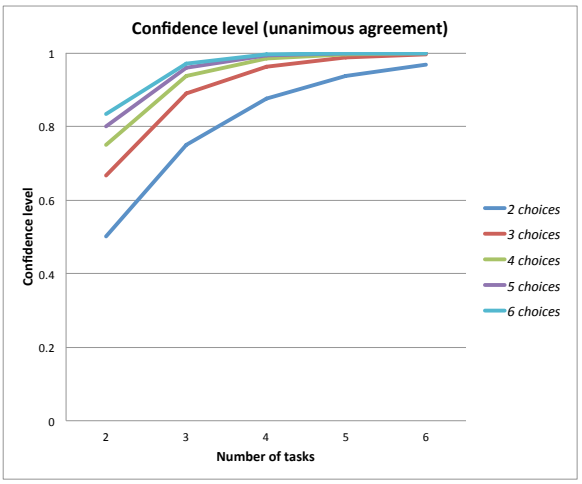

Figure 4. A graph of the number of tasks required to achieve a desired confidence level for questions with a given number of choices. As the number of choices increases, the number of tasks (workers) required quickly drops.

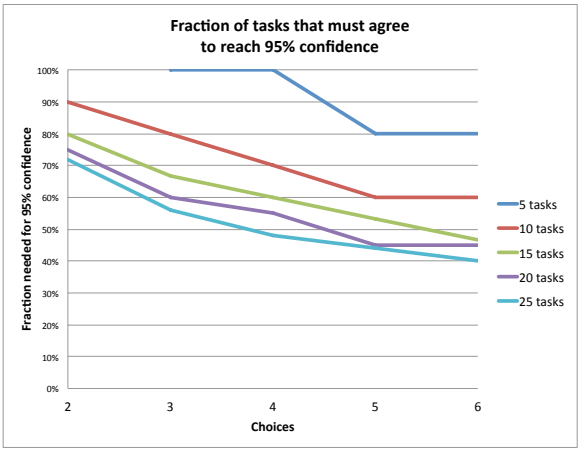

Figure 5. A graph of the *fraction* of tasks that must agree in order to achieve a 95% confidence level, for questions with a given number of choices. The fraction needed to agree drops as the number of choices increases, and as the number of tasks (workers) increases.

To compensate for this possibility, AUTOMAN treats any-number multiple choice questions specially. The AUTOMAN programmer must specify not only the question text, but also an *inverted* question. For instance, if the question is "Select which of these are true", the inverted question should read, "Select which of these are NOT true." AUTOMAN then ensures that half of the HITs are spawned with the positive question, and half with the inverted question. This strategy makes it less likely that lazy workers will inadvertently agree.

Formally, the algorithm depends on a two functions, t and ℓ, and two parameters α (one minus the confidence level) and β. The t and ℓ functions determine when it is safe to reject the null hypothesis, and when one should abandon a computation because there is no apparent consensus.

The algorithm proceeds as follows:

1. $n = \min\{m : t(m, \alpha) \neq \infty\}$

2. Ask n workers to vote on the answer of a question given k options.
3. If there is an option that has more than $t(n, \alpha)$ votes, return the most frequent option.
4. If $n < \ell(p^*, \beta)$, double n and repeat from step 2.

The $t(n, \alpha)$ and $\ell(p^*, \beta)$ in the algorithm are chosen so that:

1. If the workers are voting randomly (each answer is chosen with equal probability), then the probability that an answer meets the threshold of $t(n, \alpha)$ when n votes is cast is at most α.
2. If there is a "popular" option such that the probability a worker chooses it is $p > p^*$ (and other options are equally likely), then the above process terminates with an answer.

In the current implementation, AUTOMAN uses the t formula but adopts a fixed threshold that determines when a computation has failed. This approach is driven by the desire to keep the number of parameters required by an AUTOMAN programmer low; in any event, the threshold chosen is extremely conservative and dominates ℓ for most reasonable values. The Appendix includes derivations of closed-form formulas for t and ℓ.

Figure 5 uses the value of t to compute the normalized *fraction* of tasks that need to agree in order to reach $\alpha = 0.05$ (a confidence level of 95%). As the number of tasks and the number of choices increase, the fraction of the number of tasks needed for agreement decreases. For example, with 25 tasks and a question with 4 choices, only 48% (12 of 25) need agree in order to achieve 95% confidence. In other words, as the number of workers increases (with doubling), AUTOMAN needs an ever smaller fraction of those workers to agree, speeding convergence to a correct answer.

5. Quality Control Discussion

For AUTOMAN's quality control algorithm to work, two assumptions must hold:

- Workers are independent.
- Random choice is the worst-case behavior for workers; that is, they will not deliberately pick the wrong answer.

5.1 Worker Independence

Workers may break the assumption of independence in three ways: (1) a single worker may masquerade as multiple workers; (2) a worker may perform multiple tasks; and (3) workers may collude when working on a task.

Scenario 1: Sybil Attack. The first scenario, where one real user creates multiple electronic identities, is known in the security literature as a "Sybil attack" [5]. The practicality of a Sybil attack depends directly on how easy it is to generate multiple identities.

Carrying out a Sybil attack on our default backend, Mechanical Turk, would be extremely burdensome. Since Mechanical Turk provides a payment mechanism for workers, Amazon requires that workers provide uniquely identifying financial information, typically a credit card or bank account. These credentials are difficult, although not impossible, to forge.

Scenario 2: One Worker, Multiple Tasks. AUTOMAN avoids the second case (one worker performing multiple tasks) via a workaround of Mechanical Turk's existing mechanisms. Mechanical Turk provides a mechanism to ensure worker uniqueness for a given HIT (*i.e.*, a HIT with multiple assignments), but it lacks the functionality to ensure worker uniqueness across multiple HITs. For example, when AUTOMAN decides to respawn a task, it must be certain that workers who participated in previous instantiations of that task are excluded from future instantiations.

Our workaround for this shortcoming is to use Mechanical Turk's "qualification" feature in an inverse sense. Once a worker completes a HIT that is a part of a larger computation, AUTOMAN grants that worker special qualification (effectively, a "dequalification") that *precludes* them from participating in future tasks of the same kind. Our system ensures that workers are not able to request reauthorization.

Scenario 3: Worker Collusion. While it would be possible to attempt to lower the risk of worker collusion by ensuring that they are geographically separate (e.g., by filtering workers using IP geolocation), AUTOMAN currently does not take any particular action to prevent worker collusion. We view preventing this scenario as essentially impossible. Nothing prevents workers from colluding via external channels (e-mail, phone, word-of-mouth) to thwart the assumption of independence. We instead assume that, by spawning large numbers of tasks, AUTOMAN makes it difficult for any single group to monopolize them.

5.2 Random Worst Case

AUTOMAN's quality control function is based on excluding the possibility of random choices by workers; that is, workers who minimize their effort or make errors. It is possible that workers could instead act maliciously and deliberately choose incorrect answers.

We argue that participants in crowdsourcing systems have both short-term and long-term economic incentives to *not* deliberately choose incorrect answers, and that random choice is in fact the worst-case scenario.

First, a correct response to a given task yields an immediate monetary reward. If a worker has any information about what the correct answer is, it is against their own short-term economic self-interest to deliberately avoid it. In fact, as long as there is a substantial bias towards the correct answer, AUTOMAN's algorithm will eventually accept it.

Second, while a participant might out of malice choose to forego the immediate economic reward, there are long-term implications of deliberately choosing incorrect answers. Crowdsourcing systems like Mechanical Turk maintain an overall ratio of accepted answers to total answers submitted, and many requesters place high qualification bars on these ratios (typically around 90%). Incorrect answers thus have a lasting negative impact on workers, who, as mentioned earlier, cannot easily discard their identity and adopt a new one.

Anecdotally, we have found that Mechanical Turk workers are quite concerned when AUTOMAN rejects their answers, even when they know what the correct answer is and thus why their answer has been rejected. Several workers sent pleading e-mails justifying their answer and requesting approval, or apologizing for having misunderstood the question (for jobs whose reward was 6 cents).

6. System Architecture and Implementation

In order to cleanly separate the concerns of delivering reliable data to the end-user, interfacing with an arbitrary crowdsourcing system, and specifying validation strategies in a crowdsourcing system-agnostic manner, AUTOMAN is implemented in tiers.

6.1 Domain-specific language

The programmer's interface to AUTOMAN is a set of function calls, implemented as an embedded domain-specific language for the Scala programming language. The choice of Scala as a host language was motivated primarily by the desire to have access to a rich set of language features while maintaining compatibility with existing code. Scala is fully interoperable with existing Java code; the crowdsourcing system compatibility layer heavily utilizes this feature to communicate with Amazon's Mechanical Turk system. Scala also provides access to powerful functional language features

that simplify the task of implementing a complicated system. These function calls act as syntactic sugar, strengthening the illusion that crowdsourcing tasks really are just a kind of function call with an extra error tolerance parameter.

When using the AUTOMAN DSL, programmers first create an `AutoMan` instance, specifying an `AutomanConfig` object, which indicates which crowdsourcing system should be used (*e.g.*, Mechanical Turk) and how it should be configured (*e.g.*, user credentials, total budget, etc.). Next, the `Task` function is declared, and programmers provide the desired statistical `confidence` level, a `QuestionForm` object, and any other crowdsourcing-specific task parameters as required. When programmers call their `Task` function with some input data, the input is automatically boxed into a `FutureData` object, and an output `FutureData` object is returned.

From this point on, AUTOMAN handles communication with the crowdsourcing backend, task scheduling, quality control, and returning a result back to the programmer under budget and in a timely manner. The outputted `FutureData` object is available to use immediately, and may be passed as input to other `Task` function calls. If a `FutureData` object is accessed by native (non-AUTOMAN) functions, they will block if the runtime has not yet completed computation and propagated values into those objects.

Since our aim was to make task specification as simple as possible, and to automate as many functions as possible, our Mechanical Turk compatibility layer provides sane defaults for many of the parameters. Additionally, we delegate control of task timeouts and rewards to AUTOMAN, which will automatically adjust them to incentivize workers. Maximizing automation allows for concise task specification for the common cases. When our defaults are not appropriate for a particular program, the programmer may override them.

6.2 Abstract tasks and concrete tasks

The main purpose of the DSL is to help the programmer construct `AbstractTask` objects, which represent the system's promise to return a unit of data to the end-user. In reality, many actual tasks, which we call `ConcreteTask`s, may be created in the process of computing a single `AbstractTask`, however, this fact is hidden from the programmer.

AUTOMAN handles scheduling of all real tasks in the target crowdsourcing system. After programmers have defined their task in terms of an `AbstractTask` using the `Task` DSL keyword, they can then call the task as if it were a function. In other words, they provide input as arguments to the function, and receive output as a return value from the function, which can be fed as input to other tasks as desired.

In the background, AUTOMAN constructs a graph representing an abstract execution plan, identifying data dependencies by automatically boxing input and unboxing output parameters in `FutureData` objects which are embedded into the graph. The abstract execution plan allows AUTOMAN to inspect the planed work and to exploit all possible data parallelism present in the program. As soon as a task is called with some input data, AUTOMAN immediately begins scheduling jobs, only waiting if a given data dependency is not yet resolved.

6.3 Validation strategies

The manner in which jobs are scheduled and errors handled depends on the chosen `ValidationStrategy`. By default, we provide the `ReplicateAndAggregate` strategy, which performs the form of statistical error handling we outlined in earlier sections. However, in the event that more sophisticated error handling is required, the programmer may either extend or completely replace our base error-handling strategy by implementing the `ValidationStrategy` interface.

6.4 Third-party implementors

Implementors who wish to adapt the AUTOMAN runtime for additional crowdsourcing systems need only implement the following interfaces: `AutomanConfig`, `AbstractTask`, `ConcreteTask`, and `QuestionForm`. Programs for one crowdsourcing backend thus can be ported to a new system by including that compatibility code and specifying the proprietary system's configuration details.

7. Experience Report

A conventional evaluation of AUTOMAN is complicated by several factors. First, AUTOMANis an entirely new programming paradigm. There are no comparable systems, and thus no suite of benchmarks to use to compare AUTOMAN to other approaches. We view AUTOMAN primarily as an enabling technology that will allow programmers to explore an entirely new class of applications.

Second, running on crowdsourcing platforms adds an unprecedented level of non-determinism. There can be a wide variance in runtimes due to factors such as the time of day or day of the week. Because the population of workers is constantly in flux, no experiments are repeatable.

Instead, we report on our experience using AUTOMAN. We have written a number of AUTOMAN programs, including the cartoon character classifier described in Section 3, and a "good vs. evil" classifier. The good vs. evil question is "which of these Star Wars characters are GOOD" (and the inverted question, "which...are EVIL"). The question provides obfuscated links to images of five different Star Wars characters: Princess Leia, Han Solo, Luke Skywalker, Darth Vader, and Darth Maul (the latter two are evil).

The cartoon character classifier is a single-choice multiple choice question, while the Star Wars classifier is an any-number multiple choice question. We have also experimented with the time-value option to trade money for reduced latency (Section 4.2). The only feature that these programs do not exercise is AUTOMAN's support of complex control structure; we plan to build more complex applications that further show off AUTOMAN's unique features.

We have found AUTOMAN to be surprisingly easy to use, and that in every case to date, every one of AUTOMAN's features ends up getting exercised, including automatic scheduling, budgeting, and quality control. In particular, the quality control mechanism is robust and tremendously effective.

Because AUTOMAN handles all the low-level details of managing crowdsourcing, we have found ourselves spending more time on issues like the need to unambiguously phrase questions, and coming up with an appropriate and attractive title for questions (to entice workers).

8. Related Work

Programming the Crowd. While there has been substantial *ad hoc* use of crowdsourcing platforms, especially Amazon's Mechanical Turk, there has been little effort to manage workers programatically. Amazon's Mechanical Turk exposes a low-level API allowing jobs to be submitted, tracked, and checked programatically.

TurKit is a scripting system designed to make it easier to manage Mechanical Turk tasks [13]. TurKit Script extends JavaScript with a templating feature for common Mechanical Turk tasks, and adds checkpointing to avoid re-submitting Mechanical Turk tasks if a script fails. CrowdForge is a web tool that wraps a MapReduce-like abstraction on Mechanical Turk tasks [4, 11]. Programmers decompose tasks into *partition* tasks, *map* tasks, and *reduce* tasks. CrowdForge automatically handles distributing tasks to multiple users and collecting the results. Unlike AUTOMAN, neither TurKit nor CrowdForge automatically manage scheduling, pricing, or quality control; in addition, TurkIt's embedding in JavaScript also limits its usefulness for compute-intensive tasks.

CrowdDB models crowdsourcing as an extension to relational databases, providing annotations to traditional SQL queries that trigger the SQL runtime to crowdsource database cleansing tasks [8]. The SQL runtime is crowdsourcing-aware, so that the SQL's query planner can minimize operations that would otherwise be very expensive otherwise. Unlike AUTOMAN, CrowdDB is not a general platform for computing, and relies on majority voting as its sole quality control mechanism.

Turkomatic aims to crowdsource an entire computation, including the "programming" of the task [12]. Tasks are provided to the system in plain English, and the Turkomatic runtime proceeds in two steps: a map step and a reduce step. In the map step, workers provide an execution plan, which is then carried out in the reduce step. Like AUTOMAN, Turkomatic can be used to construct arbitrarily complex computations. However, Turkomatic does not handle budgeting or quality control, and also cannot be integrated with a traditional programming language.

Quality Control. CrowdFlower is a closed-source, web service that targets commercial crowdsourcing platforms [15]. To enhance quality, CrowdFlower uses a "gold-seeding" approach to identify likely erroneous workers, sprinkling questions with known answers into the question pipeline. CrowdFlower incorporates methods to programmatically generate this data via "fuzzing" as the system processes real work, in an effort to ease the gold-generation burden on the requester. Recognizing new types of errors remains a manual process. Like other work in this area, this approach focuses on establishing trust in the quality of a particular worker [10]. Rather than trust that one can extrapolate quality of work on a new task from a worker's past performance, AUTOMAN addresses work quality directly.

Shepherd provides interactive feedback between task requesters and task workers in an effort to increase quality; the idea is to *train* workers to do a particular job well [6]. This approach requires ongoing interaction between requesters and workers, while AUTOMAN requires none.

Soylent introduces the *find-fix-verify* pattern of quality control for written documents. The idea is to crowdsource three distinct phases: finding errors, fixing errors, and verifying the fixes [2]. Soylent can handle open-ended questions, which AUTOMAN currently does not support. However, unlike AUTOMAN, Soylent's approach does not make any quantitative guarantees about the quality of the output.

9. Future Work

We plan to build on the existing AUTOMAN prototype in the following directions:

Broader question classes. The current AUTOMAN prototype supports multiple-choice questions where exactly one of the answers is correct, or when zero or more may be correct. We plan to extend AUTOMAN to support questions with free-form answers (that is, open-ended questions). The validation strategy will add an intermediate step that depends on workers to rank answers, and then perform quality control on the rankings.

Exposing a Java API. AUTOMAN currently is implemented as a domain-specific language in Scala. While Scala makes it extremely simple to use human-based computations as if they were ordinary functions, we want to extend the benefits of AUTOMAN to users who are more comfortable programming in Java. We plan to extend AUTOMAN with an API that can be invoked from Java. We are planning to expose an API that *virtualizes* the existing Amazon Mechanical Turk SDK: programmers using the SDK will actually interact with AUTOMAN, which will manage scheduling, budgeting, and quality control automatically.

Persistent/restartable jobs. Programming humans can lead to subtle "bugs": for example, the wording of a question may turn out to be ambiguous, or have unintended interpretations. We expect AUTOMAN programs, like their counterparts in conventional programming environments, to be refined over time. However, we would also like to save the results of previous computations whenever possible to avoid wasting money. We plan to provide a facility that allows previous computations to be persisted and restored, like TurkIt [13].

Visualization tools. While AUTOMAN hides the management details of human-based computation beneath an abstraction layer, it can be useful for debugging to peel back this layer to see how tasks are progressing. AUTOMAN currently provides a simple logging mechanism, but as the number of jobs becomes large, navigating logs quicky becomes onerous. Building on the fact that the AUTOMAN runtime system already acts as a server, we plan to extend it with a web service that will allow AUTOMAN programmers to view the jobs in the system. We are initially planning to include visualizations of the execution graph (including summaries) and allow searching for jobs matching certain criteria.

10. Conclusion

Humans can perform many tasks with ease that remain difficult or impossible for computers. This paper presents AUTOMAN, the first *crowdprogramming* system. Crowdprogramming integrates human-based and digital computation. By automatically managing quality control, scheduling, and budgeting, AUTOMAN allows programmers to easily harness human-based computation for their applications.

References

[1] Amazon. Mechanical turk. `http://www.mturk.com`, June 2011.

[2] M. S. Bernstein, G. Little, R. C. Miller, B. Hartmann, M. S. Ackerman, D. R. Karger, D. Crowell, and K. Panovich. Soylent: a word processor with a crowd inside. In K. Perlin, M. Czerwinski, and R. Miller, editors, *UIST*, pages 313–322. ACM, 2010. ISBN 978-1-4503-0271-5. URL `http://dblp.uni-trier.de/db/conf/uist/uist2010.html#BernsteinLMHAKCP10`.

[3] A. DasGupta. *Probability for Statistics and Machine Learning: Fundamentals and Advanced Topics*. Springer, 1st edition, 2011.

[4] J. Dean and S. Ghemawat. MapReduce: Simplified Data Processing on Large Clusters. In *OSDI*, pages 137–150, 2004.

[5] J. R. Douceur. The Sybil Attack. In *Revised Papers from the First International Workshop on Peer-to-Peer Systems*, IPTPS '01, pages 251–260, London, UK, 2002. Springer-Verlag. ISBN 3-540-44179-4. URL `http://dl.acm.org/citation.cfm?id=646334.687813`.

[6] S. Dow, A. Kulkarni, B. Bunge, T. Nguyen, S. Klemmer, and B. Hartmann. Shepherding the crowd: managing and providing feedback to crowd workers. In *Proceedings of the 2011 annual conference extended abstracts on Human factors in computing systems*, CHI EA '11, pages 1669–1674, New York, NY, USA, 2011. ACM. ISBN 978-1-4503-0268-5. doi: http://doi.acm.org/10.1145/1979742.1979826. URL `http://doi.acm.org/10.1145/1979742.1979826`.

[7] W. Feller. *An Introduction to Probability Theory and Applications*, volume 1. John Wiley & Sons Publishers, 3rd edition, 1968.

[8] M. J. Franklin, D. Kossmann, T. Kraska, S. Ramesh, and R. Xin. Crowddb: answering queries with crowdsourcing. In T. K. Sellis, R. J. Miller, A. Kementsietsidis, and Y. Velegrakis, editors, *SIGMOD Conference*, pages 61–72. ACM, 2011. ISBN 978-1-4503-0661-4.

[9] J. Howe. The rise of crowdsourcing. *Wired Magazine*, 14(6):176–178, 2006. ISSN 1059-1028.

[10] P. G. Ipeirotis, F. Provost, and J. Wang. Quality management on amazon mechanical turk. In *Proceedings of the ACM SIGKDD Workshop on Human Computation*, HCOMP '10, pages 64–67, New York, NY, USA, 2010. ACM. ISBN 978-1-4503-0222-7. doi: http://doi.acm.org/10.1145/1837885.1837906. URL `http://doi.acm.org/10.1145/1837885.1837906`.

[11] A. Kittur, B. Smus, and R. E. Kraut. CrowdForge: Crowdsourcing Complex Work. Technical Report CMU-HCII-11-100, Human-Computer Interaction Institute, School of Computer Science, Carnegie Mellon University, February 2011.

[12] A. P. Kulkarni, M. Can, and B. Hartmann. Turkomatic: automatic recursive task and workflow design for mechanical turk. In *Proceedings of the 2011 annual conference extended abstracts on Human factors in computing systems*, CHI EA '11, pages 2053–2058, New York, NY, USA, 2011. ACM. ISBN 978-1-4503-0268-5. doi: http://doi.acm.org/10.1145/1979742.1979865. URL `http://doi.acm.org/10.1145/1979742.1979865`.

[13] G. Little, L. B. Chilton, M. Goldman, and R. C. Miller. TurKit: Human Computation Algorithms on Mechanical Turk. In *UIST*, pages 57–66, 2010.

[14] M. Odersky and M. Zenger. Scalable Component Abstractions. In *Proceedings of the 20th annual ACM SIGPLAN conference on Object-oriented programming, systems, languages, and applications*, OOPSLA '05, pages 41–57, New York, NY, USA, 2005. ACM. ISBN 1-59593-031-0. doi: http://doi.acm.org/10.1145/1094811.1094815. URL `http://doi.acm.org/10.1145/1094811.1094815`.

[15] D. Oleson, V. Hester, A. Sorokin, G. Laughlin, J. Le, and L. Biewald. Programmatic Gold: Targeted and Scalable Quality Assurance in Crowdsourcing. In *HCOMP '11: Proceedings of the Third AAAI Human Computation Workshop*. Association for the Advancement of Artificial Intelligence, 2011.

[16] D. Shahaf and E. Amir. Towards a theory of ai completeness. In *Commonsense 2007: 8th International Symposium on Logical Formalizations of Commonsense Reasoning*. Association for the Advancement of Artificial Intelligence, 2007.

A. Quality Control Threshold Derivation

Given parameters $n \in \mathbb{N}_0$ and $0 \leq p_1, p_2, \ldots, p_k \leq 1$ with $\sum_{i=1}^{k} p_i = 1$, we say $Z = (Z_1, \ldots, Z_k)$ is a multinomial distribution with parameters $(n, p_1, p_2, \ldots, p_k)$ if for any $z_1, z_2, \ldots \in \mathbb{N}_0$ with $\sum_{i=1}^{k} z_i = n$

$$\Pr[\forall i:\ Z_i = z_i] = \frac{n!}{z_1! z_2! \ldots z_k!} p_1^{x_1} \ldots p_k^{x_k}$$

For example, if n voters are given k options and each voter (independently) picks option i with probability p_i then Z_i will correspond to the number of votes received by the ith option.

To compute probability of a multinomial distribution, we follow approach outlined by DasGupta [3].

Lemma A.1. *For $0 \leq a_i \leq b_i \leq n$ and $i \in [k]$,*

$$\Pr[\forall i:\ Z_i \in [a_i, b_i]] = n! \cdot \operatorname{coeff}_{\lambda,n}\left(\prod_{i=1}^{k}\left(\sum_{j=a_i}^{b_i} \frac{(p_i\lambda)^j}{j!}\right)\right),$$

where $\operatorname{coeff}_{\lambda,n}(f(\lambda))$ is the coefficient of λ^n in the polynomial f.

Proof. Let $N \in \operatorname{Poi}(\lambda)$ be a random variable distributed according to the Poisson distribution with parameter λ. Then let $Z^N = (Z_1^N, \ldots, Z_k^N)$ is a multinomial distribution with parameters $(N, p_1, p_2, \ldots, p_k)$. Note that $Z = Z^n$. It's known (e.g., [7, pp. 216]) that

$$\begin{aligned}\Pr\left[\forall i:\ Z_i^N \in [a_i, b_i]\right] &= \prod_{i=1}^{k} \Pr\left[Z_i^N \in [a_i, b_i]\right] \\ &= \left(\prod_{i=1}^{k}\left(\sum_{j=a_i}^{b_i} \frac{(p_i\lambda)^j e^{-p_i\lambda}}{j!}\right)\right) \\ &= \left(e^{-\lambda}\prod_{i=1}^{k}\left(\sum_{j=a_i}^{b_i} \frac{(p_i\lambda)^j}{j!}\right)\right).\end{aligned}$$

But

$$\begin{aligned}&\Pr\left[\forall i:\ Z_i^N \in [a_i, b_i]\right] \\ =\ & \sum_{m=0}^{\infty} \Pr[N = m] \cdot \Pr[\forall i:\ Z_i^m \in [a_i, b_i]] \\ =\ & \sum_{m=0}^{\infty} \frac{\lambda^m e^{-\lambda}}{m!} \cdot \Pr[\forall i:\ Z_i^m \in [a_i, b_i]]\end{aligned}$$

and hence

$$\sum_{m=0}^{\infty} \frac{\lambda^m}{m!} \cdot \Pr[\forall i:\ Z_i^m \in [a_i, b_i]] = \left(\prod_{i=1}^{k}\left(\sum_{j=a_i}^{b_i} \frac{(p_i\lambda)^j}{j!}\right)\right).$$

The result follows by equating the coefficients of λ^n. □

Let X and Y be multinomial distributions with parameters $(n, 1/k, \ldots, 1/k)$ and $(n, p, q, \ldots, q)$ where $q = (1-p)/(k-1)$ respectively. The next lemma follow from Lemma A.1.

Lemma A.2.

$$\begin{aligned}\Pr\left[\max_i X_i < t\right] &= \mathcal{E}_1(n, t) \\ \Pr\left[\max_{i \geq 2} Y_i < t \leq Y_1\right] &= \mathcal{E}_2(p, n, t)\end{aligned}$$

where

$$\mathcal{E}_1(n, t) = \frac{n!}{k^n} \cdot \operatorname{coeff}_{\lambda,n}\left((\sum_{j=0}^{t-1} \lambda^j/j!)^k\right)$$

and

$$\mathcal{E}_2(p, n, t) = n! \cdot \operatorname{coeff}_{\lambda,n}\left(\left(\sum_{j=0}^{t-1} \frac{(q\lambda)^j}{j!}\right)^{k-1} \cdot \left(\sum_{j=t}^{\infty} \frac{(p\lambda)^j}{j!}\right)\right).$$

Note that $\mathcal{E}_1(n, n) = 1 - 1/k^{n-1}$ and define

$$t(n, \alpha) := \begin{cases} \min\{t : \mathcal{E}_1(n, t) \leq \alpha\} & \text{if } \mathcal{E}_1(n, n) \leq \alpha \\ \infty & \text{if } \mathcal{E}_1(n, n) > \alpha \end{cases}. \quad (1)$$

This ensures that when n voters each randomly chose an option, the probability that an option passes the threshold $t(n, \alpha)$ is at most α. Next we define

$$\ell(p^*, \beta) := \min\{n : \mathcal{E}_2(p^*, n, t(n, \alpha)) \leq \beta\}. \quad (2)$$

This ensures that if the voters have a bias of at least p^* towards a certain popular option (and all other options are equally weighted), then when we ask $\ell(p^*)$ voters, the number of votes cast for the popular option passes the threshold (and all other options are below threshold) with probability at least β.