Appsheet: Efficient use of web workers to support decision making

Alexander J. Quinn, Benjamin B. Bederson
University of Maryland
Department of Computer Science
Human-Computer Interaction Lab
College Park, MD 20742 USA
{aq, bederson}@cs.umd.edu

ABSTRACT
The wealth of information and social resources online has raised the bar for the quality of decisions that individuals and businesses can make. Human computation and social mediums have also increased the potential for finding relevant information or opinions and making them a part of a decision-making process. However, the strategies that individuals employ when confronted with too much information—satisficing, information foraging, etc.—are more difficult to apply with a large, distributed group. Appsheet is a new technology foundation that uses a spreadsheet model of a decision to guide distributed search parties in support of decision-making applications.

INTRODUCTION
The explosive growth in availability of information online has opened a bounty of resources for users making plans and strategic decisions. Users can turn to web pages and online databases for questions requiring objective information. Subjective questions can often be answered by a post on a social network or discussion forum, or even a direct email or chat message to a trusted colleague.

The challenge comes when trying to leverage these mediums for more complex problems that require more thorough analysis. While it may be easy to find an answer to a specific question, a solution to the overarching problem may be more elusive. Consider the following questions:

Where should I buy groceries this week to save money?
If grocery stores made all of their prices freely available as a downloadable price list or through an API, then customers could save money by comparing the total cost of the items on their shopping list at each store.

Where/when can I vacation to balance cost vs. preference?
If every airline made their best fares available in a form that could be imported into a spreadsheet, then travelers could assign a preference weight to each destination, departure date, and return date they were considering, and balance with the costs to find the best itinerary for each situation.

What graduate programs should I apply to?
If all relevant details about every program were readily available in one place, we could easily identify the ones that best fit our personal criteria.

Which of the 6,000 paper submissions should be accepted?
If reviewers had unlimited time, every paper could be read by every relevant reviewer, ensuring greater consistency.

Ideally we would like answers that account for the specific needs of each individual or situation and make use of all available information. In reality, time to gather the information is limited, so we compromise. We accept a suboptimal decision based on partial information or take the first option that meets some baseline standard. This behavior by individual decision-makers, termed *satisficing* by Simon [36], results from the inherent tradeoff between the cost of deciding and the benefit of a well-reasoned decision that utilizes as much information as possible.

The rapid development of online micro-task markets, such as Amazon Mechanical Turk, presents options that did not previously exist. Now, one can easily hire web workers to gather information, paying a small amount of money for a small amount of work. For example, Smartsheet is a commercial service that facilitates the process of hiring workers to search for data [17].

Options have also emerged for more subjective matters. Question-answer forums (i.e., Yahoo Answers [21]) and social network site features (i.e., Facebook Questions [15]) allow users to solicit answers to questions in cases where a public discussion might yield a better result. To receive an answer, the question should not be too complex, and it should be expressed concisely, so that it not require much effort to respond to [13]. For other problems demanding more privacy or control over who provides the answers, tasks can be posted to an internal messaging system.

A limitation of such distributed approaches is that they follow a static workflow. Requests are sent and then the requester waits until all results have been received. In contrast, individual information foraging tends to follow a dynamic workflow [32]. Users seek out the information that will have the greatest impact on the final result, constantly adjusting the strategy based on the information seen so far. Current crowdsourcing systems offer no such facility.

Appsheet is a research prototype we built to demonstrate a new method of coordinating workers to support decision making tasks. To reduce human effort wasted gathering unnecessary information, it leverages a model of the decision provided by the user in the form of a spreadsheet.

To start, the user (someone with a decision to make) creates an empty spreadsheet with formulas that will calculate the end result. In cells where information is needed, the user enters a special `ASK(…)` formula. The parameters to this formula communicate to the Appsheet server that some information is needed and the range of values that is expected. Appsheet analyzes the formulas and sends requests to the helpers specified by the user. Using a value of information analysis, the system eliminates any ASK formulas that will not affect the final result. Since humans are generally much slower and more expensive than CPUs, the system aggressively prioritizes the inputs to minimize unnecessary requests.

Appsheet is primarily intended for problems that are oriented toward gathering a lot of details and then selecting a subset of them, typically based on the relation to the rest. This includes the examples above and, more generally, any problem that can be modeled using operations such as minimum, maximum, median, top-*n*, *n*th-largest, pick-any, or if/else. Since expected bounds of the inputs are specified by the user a priori, such operations can often be precisely computed with only partial information.

Appsheet is not intended for needle-in-a-haystack search problems (i.e., find Mayor X's salary). Also excluded are problems where a large set of information is gathered and all of it is to be used (i.e., find salaries of all mayors to publish a complete list in a newspaper).

The key contribution of this paper is a technical foundation for coordinating and optimizing human data gathering processes in support of decision making.

NEED FOR RESEARCH

One of the primary ways Appsheet helps users make decisions is by coordinating many participants to gather the requisite information. This can be thought of as a form of collaborative search.

Collaborative search has been well-explored for small groups of searchers, both co-located and remote [3, 29, 30]. Typically, the collaborators are working toward a common goal that is important to both. Appsheet is different in that the initiative comes primarily from the person who initiates the process. The human helpers may or may not also be stakeholders in the problem. Also, Appsheet aims to support much larger, more distributed, and possibly anonymous groups working together.

Crowdsourcing search has demonstrated some limited commercial viability by ChaCha, a company that hires remote workers to interpret users' search terms and return high quality search results in real time [9, 23]. Smartsheet, mentioned earlier, is the most similar project to Appsheet. Paid web workers enter information to fill a spreadsheet-like interface [17]. However, Smartsheet is only for collecting monolithic lists of information. It does not allow for collecting a cell of data at a time, and there is no support for goal-directed control based on the formulas and information obtained. In contrast, Appsheet supports requesting data one cell at a time, and it uses the formulas in the model to dynamically decide which information to collect next.

Aardvark was a commercial service that, until 2011, allowed a person to enter a question into an instant messaging client and get a response from another user of the service who was expected to be knowledgeable about the topic of the question [23]. This was good for simple questions where one person might reasonably be expected to know the answer. It did not help with more complex tasks where information needed to be gathered from different sources. However, the mode of communicating with helpers was novel. We envision using something similar with Appsheet in the future to reduce the delay between when a request is sent and when it is answered. While interrupting helpers with requests in chat windows might be considered too intrusive in many situations, we suppose that among a workgroup that is already working actively on the problem, it might be acceptable and yield productivity gains sufficient to warrant the intrusions.

Freebase is a commercial project that has hired workers to enter information into a very large encyclopedic ontology [6]. Crowdsourcing has also been used to gather data to make traditional search engines better [22, 25]. These efforts are aimed at producing large databases for future use, as opposed to solving a single problem, as in the case of Appsheet. However, they are similar in that they coordinate web workers to gather information from the web and organize it in a manner that supports the application.

Several efforts have been made to develop easy programming models for directing web workers. TurKit is a programming system for using Mechanical Turk. It allows programmers to write programs in JavaScript as if all requests to workers were handled immediately, essentially factoring out the usual lag in waiting for results [24]. Whereas TurKit is more generally applicable (beyond decision-making), Appsheet uses far more aggressive methods for optimizing evaluation in order to make more efficient use of the human effort. CrowdLang is another programming system, which aims to better support the task of building systems that interact with crowd labor, often in conjunction with complementary machine resources [28]. Like Appsheet, both of these support the use of a programming language to describe a dynamic workflow that web workers will follow.

Note that spreadsheet formulas are considered a first-order functional programming language [1]. Functional programming is a paradigm in programming languages that emphasizes data flow through mathematical functions instead of variables and mutable data structures. As evidence of the versatility of spreadsheet formulas as a programming language, Casamir showed how many elementary functional programming exercises (i.e., towers of Hanoi, Fibonacci sequences, generation of permutations, etc.) can actually be accomplished using clever spreadsheet formulas [8].

Appsheet could have been built around a different programming language. We briefly considered using a variant of JavaScript, Python, or Scala. However, from the perspective of optimization, spreadsheets have the advantage that many semantic properties, such as the order of evaluating parameters and operands, are not well defined. For example, whereas in most languages, evaluation of the expression `X AND Y` always proceeds left-to-right, the creators of spreadsheets make no such guarantee. Appsheet actively exploits these ambiguities by actively choosing the order that will minimize human effort in the overall workflow.

A simpler "language" for specifying crowdsourced workflows is the so-called *human macro* found in Soylent [4]. To use this feature, a user of Microsoft Word describes in natural language some text manipulations to be done. Then, paid web workers read the instructions and perform the manipulations. Even simpler yet, VizWiz allows blind users to take a photo using a smartphone and then speak a question about the photo in natural language; answers are given by a paid web worker [5]. Like Appsheet, both of these projects use paid web workers to provide crucial functionality in an end-user application.

Whereas Appsheet is essentially applying a spreadsheet interface to working with crowdsourcing channels, Qurk and CrowdDB have applied a database interface. Both projects allow a user to write SQL-like queries to be answered by web workers, even including sort and join operations [16,26,27]. Although the objective and mechanisms are very different from Appsheet, these are similar in that they use a goal-directed process to generate requests to paid web workers.

In addition to the languages mentioned above, some projects have focused on supporting complex tasks using crowdsourcing, typically using some sort of divide and conquer strategy [19, 20]. Like Appsheet, these projects use a specification of the end goal provided by the user, and dynamically decide what tasks to issue to workers.

At the heart of Appsheet and most of the aforementioned projects is the basic idea of using software to direct humans to perform a process according to an explicitly defined algorithm. This is known as *human computation* [33]. Appsheet is a specific instance of human computation that is goal-driven, using a model.

Human computation is closely intertwined with artificial intelligence. In fact, human computation is sometimes called *artificial artificial intelligence* [2, 37]. Dai et al have used artificial intelligence methods to achieve specific quality levels from untrusted paid web workers, by using a decision-theoretic optimization model to balance result quality with the cost of obtaining the result [12]. That work assumes that all requests must be fulfilled. In contrast, Appsheet sidesteps the issue of quality and focuses instead on achieving a solution to a high level decision problem with as few requests as possible. The value of information analysis it uses is rooted in artificial intelligence methods for decision-making [35].

Within machine learning, active learning is a technique whereby computer estimates of task difficulty are used to actively solicit new training data from humans in order to improve the accuracy of a classifier [38]. On the surface, this might appear to be similar to Appsheet, since it involves makes requests using algorithms that aim to achieve a goal while minimizing the burden on the human helpers. The main difference is that active learning creates a predictive model (classifier), whereas Appsheet helps a particular user achieve a single goal in a prescribed way according to the formulas in that user's decision model.

Appsheet may be used as a means of group decision support [31], but that is not the primary focus. Whereas typical group decision support systems are geared toward situations where all participants are stakeholders to some degree, the premise of Appsheet is that one user, the person who provides the model, owns the process and is the primary stakeholder. The helpers might have an interest in the outcome, but that is not an assumption. Also, unlike typical group decision support systems, Appsheet does not explicitly support brainstorming or voting.

One of the advantages of Appsheet is that it allows one to benefit from many information sources, rather than just a single search engine or a single database. Similarly, the Liquid Query paradigm aims to support queries that span multiple domains that might otherwise be accessible only by multiple search services, using an SQL-like syntax [7]. The queries would be handled automatically, without help from web workers or other such human helpers.

A key characteristic of the problem domain of Appsheet in the context of search is that the tasks involve performing a series of many searches in a systematic, pre-prescribed way. Search Pad is a research prototype that was designed to support such *search missions* by automatically detecting such searches that appear to be inter-related and offering interface support for note-taking [14]. In the case of Appsheet, the role of note-taking is handled implicitly by the spreadsheet model that collects the results.

Sensemaking is a model that describes a broader class of complex search behavior in terms of the costs of searching and processing the information found [34]. It models the process of an individual developing an understanding of a topic, rather than pursuing a single, limited objective, as in the case of Appsheet. *Information scent* is a closely related concept which models the subjective value of a resource, as perceived by the information-seeker, along with the cost of accessing it [10]. Because it essentially balances the value of the information with the cost of acquisition, it is a closer analogy for what Appsheet does in a distributed way.

APPSHEET

With Appsheet, the user creates a spreadsheet to match the specifics of their particular task. Appsheet is implemented as an extension to the spreadsheet application. The use of Appsheet can best be explained by an example.

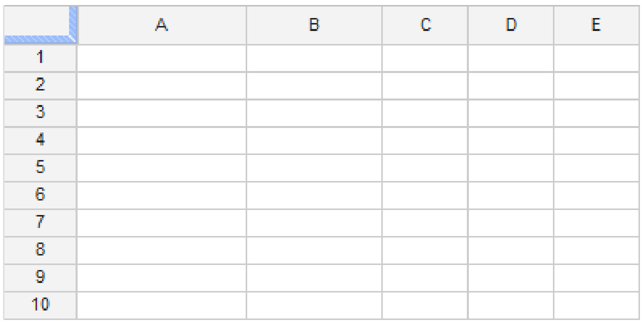
Andrew operates a catering business that needs to buy a substantial amount of food each week—typically about 50 items totaling about $1,000. Depending on the week, any of the five grocery stores nearby might have some of the items on sale. All of the grocery stores make their weekly flyers available online in PDF format, but there is no API or central database of current prices for groceries in a particular area. Therefore, Andrew will hire web workers to search through the flyers and look for sale prices on the items so he can get the best deal possible at a single store.

The process starts with a blank spreadsheet (Figure 1a). Next, he fills in the needed ingredients, the names of the stores, and formulas for the totals at the bottom (Figure 1b). For compactness, we will abbreviate the scenario to only 5 items and 3 stores.

Cell C7 calculates the sum of the prices at Store A.

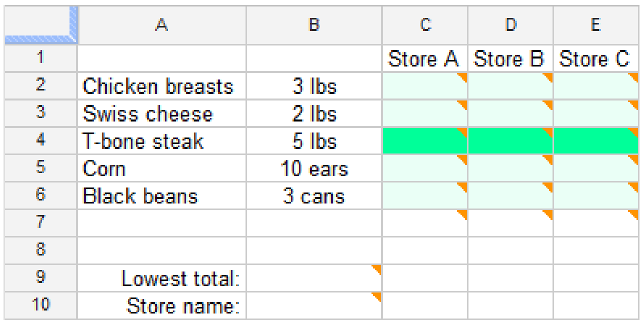
```
=SUM(C2:C6)
```

Cells D7 and E7 contain similar formulas for the total prices at Store B and Store C.



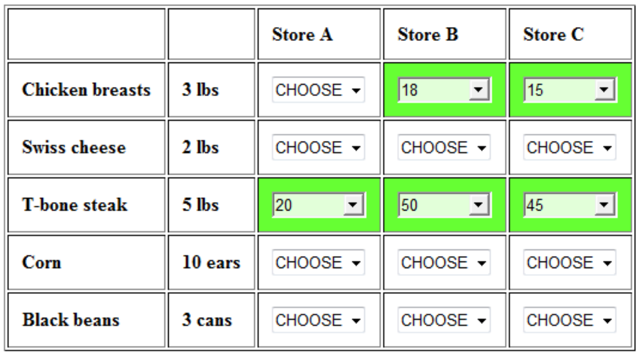
	A	B	C	D	E
1					
2					
3					
4					
5					
6					
7					
8					
9					
10					

(a) Start with a blank spreadsheet.



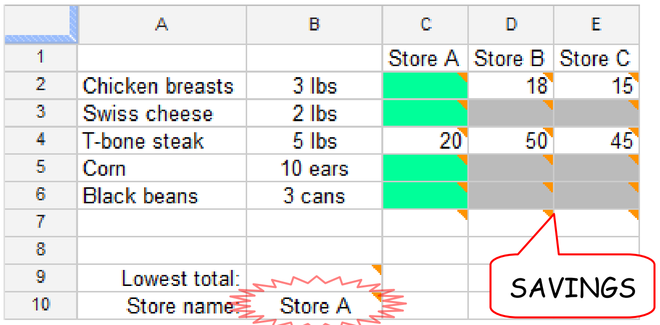
	A	B	C	D	E
1			Store A	Store B	Store C
2	Chicken breasts	3 lbs			
3	Swiss cheese	2 lbs			
4	T-bone steak	5 lbs			
5	Corn	10 ears			
6	Black beans	3 cans			
7					
8					
9	Lowest total:				
10	Store name:				

(b) Create a blank decision model with ASK formulas in cells C2:E6. Appsheet highlights priority cells in dark green.



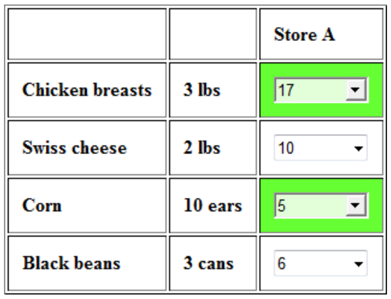
		Store A	Store B	Store C
Chicken breasts	3 lbs	CHOOSE	18	15
Swiss cheese	2 lbs	CHOOSE	CHOOSE	CHOOSE
T-bone steak	5 lbs	20	50	45
Corn	10 ears	CHOOSE	CHOOSE	CHOOSE
Black beans	3 cans	CHOOSE	CHOOSE	CHOOSE

(c) Helper fills in the five highest priority cells. This form is automatically generated from the spreadsheet. Note that the form generator uses the same priority ordering, but slightly different rules to decide which cells to color green.



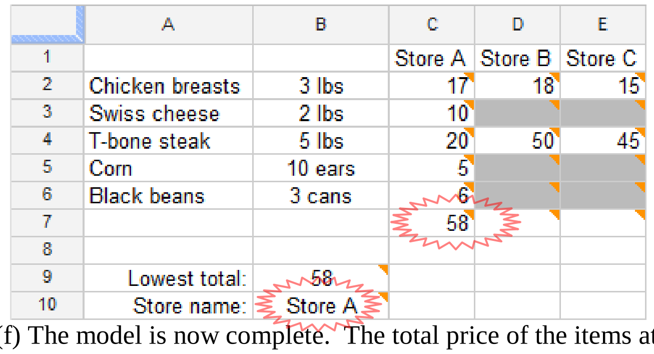
	A	B	C	D	E
1			Store A	Store B	Store C
2	Chicken breasts	3 lbs		18	15
3	Swiss cheese	2 lbs			
4	T-bone steak	5 lbs	20	50	45
5	Corn	10 ears			
6	Black beans	3 cans			
7					
8					
9	Lowest total:				
10	Store name:	Store A			

(d) Appsheet pushes the answers to the spreadsheet and updates the priorities. Store A is now guaranteed to have the lowest total price, so the store name is now displayed in B10. Remaining requests for Store B and Store C are no longer needed (savings).



		Store A
Chicken breasts	3 lbs	17
Swiss cheese	2 lbs	10
Corn	10 ears	5
Black beans	3 cans	6

(e) Next time a helper loads the form, requests that no longer needed or already fulfilled are omitted.



	A	B	C	D	E
1			Store A	Store B	Store C
2	Chicken breasts	3 lbs	17	18	15
3	Swiss cheese	2 lbs	10		
4	T-bone steak	5 lbs	20	50	45
5	Corn	10 ears	5		
6	Black beans	3 cans	6		
7			58		
8					
9	Lowest total:	58			
10	Store name:	Store A			

(f) The model is now complete. The total price of the items at the least expensive store, Store A, is shown in C6 and B9.

Figure 1. Using Appsheet begins with a blank spreadsheet and ends with a conclusive answer to the decision question.

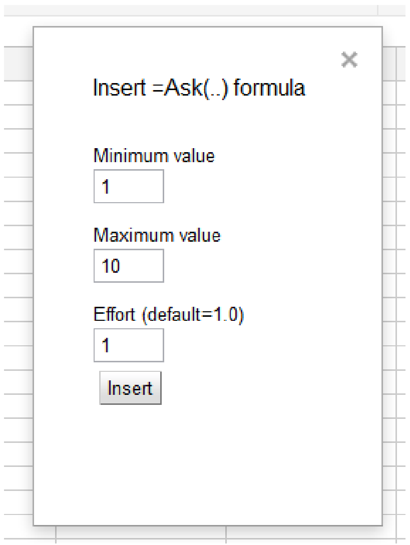


Figure 2. Formula builder for creating ASK formulas

Cell B9 calculates the minimum of the sums for each store.

```
=MIN(C7:E7)
```

Cell B10 selects the name of the store with the lowest total.

```
=INDEX(C1:E1, 1, MATCH(B9, C7:E7, 0))
```

`MATCH(…)` returns the position (1, 2, or 3), and `INDEX(…)` returns the contents of that position within the list of store names. Note that `MIN(…)`, `INDEX(…)` and `MATCH(…)` are all standard functions found in most spreadsheet applications; they are not specific to Appsheet.

In cells C2:E6 (columns C through E, rows 2 through 6), he will enter `ASK(…)` formulas (specific to Appsheet) to indicate requests for information. The syntax is as follows:

=ASK(*cost*, *minval*, *maxval*)

The *cost* parameter is the effort expected to fetch this piece of information. The units are arbitrary—relative to other requests in that model. It is used for prioritizing the cells relative to one another. The *minval* and *maxval* parameters specify the range of values that are allowed or expected.

In cell C2, Andrew fills in the following formula:

```
=ASK(1, 6, 18)
```

This means that the 3 pounds of chicken breasts are expected to cost between $6 and $18. For this example, we will take the cost of all requests to be the same (cost=1).

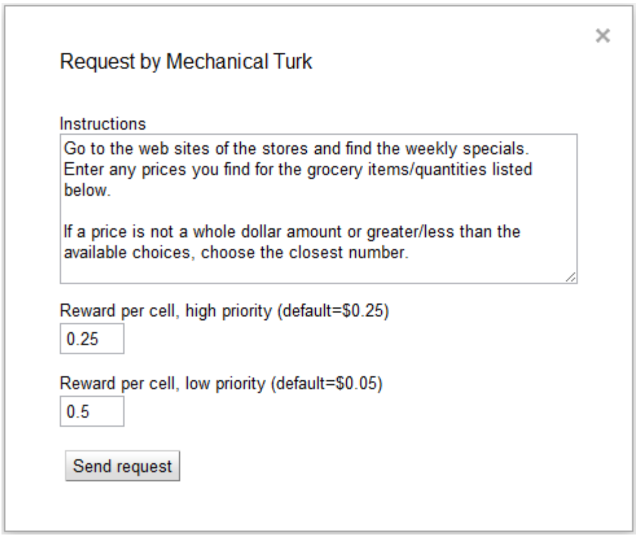
Andrew fills in similar formulas for the other items on the shopping list. Since the `ASK(…)` function is not a standard spreadsheet feature, the system provides a dialog box to help users build the formulas (Figure 2).

As Andrew creates the model, the system analyzes the formulas and categorizes each request as either *high priority* (dark green), *low* priority (light green), or *no* priority (gray). High priority cells are those which, if fulfilled next, would provide the greatest expected cost savings. In other words, on average, fulfilling these first would minimize the cost of determining which store has the best deal. No priority cells are those that will not affect the end result at all. For example, when comparing several alternatives, once the most important attributes are known, it is usually possible to ignore less important attributes for all but the primary contenders. Appsheet prioritizes the inputs automatically, with no prior knowledge of the user's model, using only the formulas entered by the user.

Appsheet uses the spreadsheet formulas to generate a web form for entering data (Figure 1c). Notice that the web form is based on the contents of the spreadsheet. Also notice that the cells highlighted in green are for the highest priced items. If one store had a good sale on these items and the others did not, then this might be enough information in itself to determine which store would have the lowest total overall, even without knowing the prices of the other items (Figure 3).

Using the Appsheet interface, Andrew can hire workers on Mechanical Turk to go to each store's web site, find the prices, and enter them in the generated form. As results are received, Appsheet updates the priorities and posts new tasks on Mechanical Turk, as needed. If workers do the highlighted cells first, then the work will be done with minimal overall human effort and hence lower monetary cost to Andrew for hiring the workers to help him decide. Therefore, he will probably offer much higher price for the high priority cells, but very little for the others .

In addition to using Mechanical Turk, Andrew can also choose to email a link to the form to colleagues who have previously agreed to help. This is also done directly from the spreadsheet application using the Appsheet interface. This might be appropriate if specific expertise were needed (i.e., specialty ingredients) or if there was a need for privacy (i.e., if he did not want other catering companies to know what he is cooking).



Instructions

Go to the web sites of the stores and find the weekly specials. Enter any prices you find for the grocery items/quantities listed below.

If a price is not a whole dollar amount or greater/less than the available choices, choose the closest number.

Reward per cell, high priority (default=$0.25): 0.25

Reward per cell, low priority (default=$0.05): 0.5

Figure 3. Dialog for starting requests by Mechanical Turk

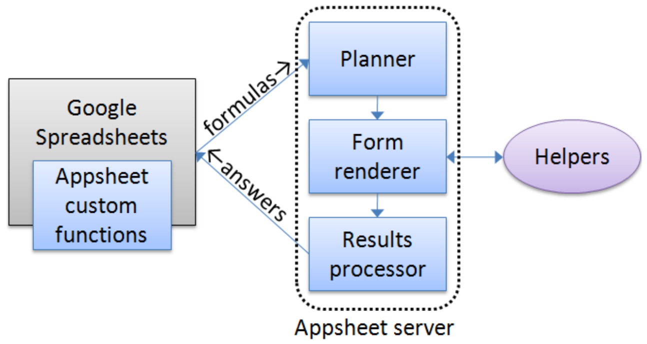


Figure 4. System architecture overview

While the requests are pending, Appsheet displays nothing (empty string) in cell B9, the name of the store with the lowest total price. As soon as enough data has been entered to calculate a conclusive answer, it displays the result of that formula. In essence, this happens when the worst case of the winner (based on the inputs not yet received) is better (lower price) than the best case of any of the losers.

A primary benefit of using Appsheet is that it does not need to fulfill all of the requests. Even though many cells were left blank, it was still able to calculate a result conclusively. This is similar to what individuals do when they have learned enough to know the outcome without seeing every detail of every alternative. For Andrew, this means that he may be able to get results faster and/or at a lower cost.

It should be emphasized that Appsheet does not depend on this particular decision model. It will work with whatever formulas and model structure the user enters. The details of this example and the others in this paper are incidental.

IMPLEMENTATION

The Appsheet implementation is comprised of a server application and some scripts made to work with Google Spreadsheets, a web-based spreadsheet. We chose to build on a web-based spreadsheet (as opposed to an add-on for Microsoft Excel, for example), because this is a research prototype; our primary purpose is to support experimentation and convey ideas to other researchers—not to deploy as a product. This works from any standard web browser; no other software or browser plugins are required.

The system architecture of Appsheet is shown in Figure 4. The Appsheet custom functions, including the `ASK(…)` function, are written in JavaScript using the Google Apps Script API. That module also includes the dialog boxes shown in Figure 2 and Figure 3.

Most of the core functionality resides in the Appsheet server, which is implemented in Python and runs on a separate web server. It includes the planner, form renderer, and the results processor.

The planner fetches the contents of every cell in the spreadsheet contents using the Google Data API. Each cell is parsed into an abstract syntax tree (AST) and connected to others wherever there is a cell reference. This structure can be used to calculate the current value of any cell.

Prioritization algorithms

Using the structure of ASTs, the planner can infer which cells the user is likely to care about (i.e., the `INDEX(…)` function in cell B10 of the example above). These are the *roots* of the resulting graph—cells which depend directly or indirectly on one or more requests. (Here we refer to `ASK(…)` functions as *requests*.)

The goal is to find an ordering of the requests that will minimize the expected cost of evaluating the model—finding a conclusive result for all root cells. We think of the *cost of an ordering* as the expected sum of the costs from all requests that would be fulfilled (not eliminated) if they were taken in that order. We define the *resulting expected model cost* of fulfilling a particular request as the expected cost of evaluating the model, if that request were fulfilled next.

One of the advantages of using spreadsheets for this work is that the order of evaluation for most operations is effectively arbitrary. Often, evaluating operands in one order can yield a higher probability of eliminating one of the others by short-circuit evaluation, or some other form of lazy evaluation. This is the basis of savings with Appsheet.

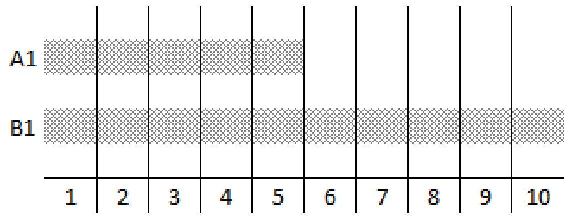
For purposes of illustration, consider this very trivial model.

A1: `=ASK(1,    1,    5)`
cost *minval* *maxval*

B1: `=ASK(1,    1,   10)`
cost *minval* *maxval*

C1: `=IF(A1 < B1,   "A",     "B")`
condition *value if true* *value if false*

Parsing this model yields the structure in Figure 5. The output ranges of cells A1 and B1 are as follows:



Note that in the current iteration of Appsheet, only integers, strings, and Boolean values are supported as data types, and

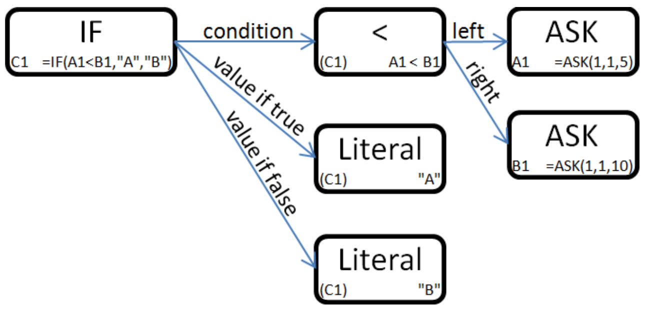


Figure 5. The trivial example is parsed and connected into a continuous abstract syntax tree (AST).

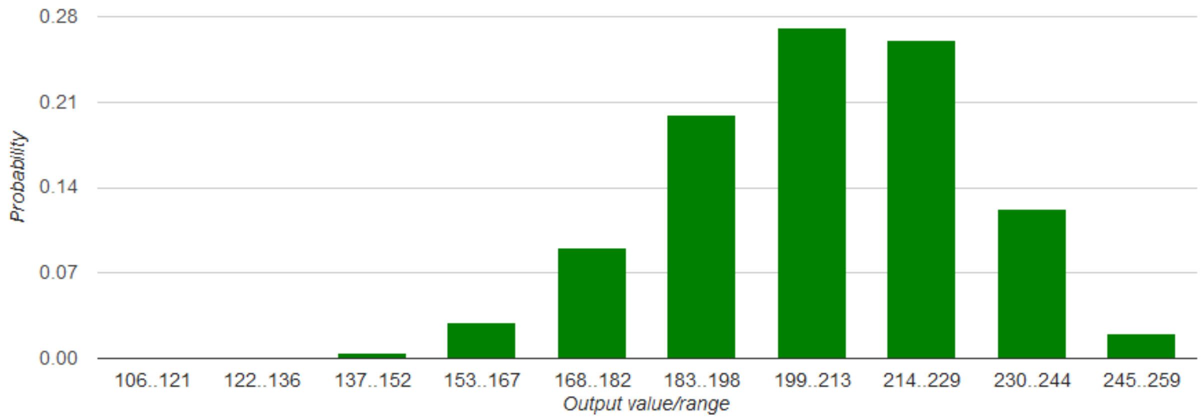


Figure 6. Output distribution of a =MAX(..) formula

each of the possible values of each input is taken to be equally probable (discrete uniform distribution).

When evaluating the comparison in the `IF(…)` function in cell C1 of the above model, we could evaluate A1 and B1 in either order: (A1, B1) or (B1, A1). Suppose we choose the latter and evaluate B1 first.

If the request in B1 receives any of the values 6, 7, 8, 9, or 10, then the comparison A1<B1 must be true, no matter which value A1 receives. Also, if B1 receives the value 1, then A1<B1 must be false. Therefore, we would only need A1 if B1 turns out to be 2, 3, 4, or 5. In other words, Pr(need A1 | have B1) = 0.4.

On the other hand, if we evaluate A1 first, no matter which value it receives, we will still need B1 in order to evaluate the comparison. Thus, Pr(need B1 | have A1) = 1.0.

Based on the above, we can say that the resulting expected model cost of fulfilling A1 is 2.0 because if A1 is fulfilled next, we will definitely need both B1 (cost=1) and A1 (cost=1) with probability 1.0. The resulting expected model cost of fulfilling B1 is 1.4 because we will need A1 with probability 0.4 and we will need B1 will probability 1.0.

More generally, for any model *M* we will calculate the expected resulting model cost of fulfilling each request that the root cells depend on. Sorting by that will give us our final ordering. The expected resulting model cost is calculated as:

$$\mathrm{E}(\mathrm{C}(M)|r_{next}\,\mathrm{next}) = \sum_{r\in reqs} \mathrm{C}(r)\mathrm{Pr}(M \text{ needs } r|r_{next}\,\mathrm{next})$$

… where C(*r*) is the cost specified in the `ASK(…)` function parameters, *reqs* is the set of all requests in the model, C(*M*) is the cost of evaluating the model, and Pr(*M* needs *r*) is the probability that evaluating every root in the model will require fulfilling request *r*. That can in turn be expressed as the probability that any root node will need request *r*. (Here we omit the condition for brevity.)

$$\mathrm{Pr}(M \text{ needs } r) = \mathrm{Pr}\left(\bigvee_{n\,\in roots} n \text{ needs } r\right)$$

The probability that a particular node *n* needs a request *r* is calculated recursively:

- If *n* is a request and *n* = *r*, then P(*n* needs *r*) = 1. (A request node needs itself.)
- If *n* is a request and *n* ≠ *r*, then P(*n* needs *r*) = 0. (A request node does not need any other nodes.)
- If *n* is any other node type, the probability is:

  $$\mathrm{Pr}(n \text{ needs } r) = \mathrm{Pr}\left(\bigvee_{op\in Ops_n} op \text{ needs } r \;\wedge\; n \text{ needs } op\right)$$

  … where *Ops*$_n$ is the set of operands (i.e., actual parameters to a function, etc.) taken by node *n*.

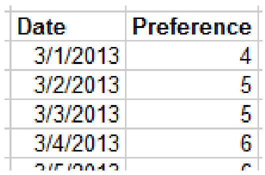
For simple arithmetic operations, P(*n* needs *op*)=1 for all of the operands, since you cannot calculate an arithmetic operation without all of its operands. For example, to calculate *a* + *b*, you must know the values of *a* and *b*, regardless of the bounds or distributions.

For operations where can potentially be optimized to eliminate requests—including the comparison operators, `MIN(…)`, `MAX(…)`, `IF(…)`, and many others—calculating the probability that the node needs each operand generally entails calculating the output distribution of the node (every possible output value for the node and the probability of each). Efficient algorithms for calculating both of these probabilities are specific to each operator or function, and beyond the scope of this paper.

Since it may be helpful for users to know the possible values for a cell, Appsheet includes an *Inspect* tool that displays a bar chart representation of the output distribution for any formula cell. An example from another model (not the example above) is shown in Figure 6.

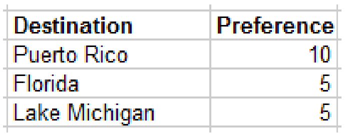
SPREADSHEET MODELING

In this section, we will illustrate how some other problems can be modeled in spreadsheets. In most cases, a fuller implementation would probably involve a more detailed model with greater attention to the nuances of the situation. Many of these will require further development of our prototype, but they are well within the capabilities of the method, and they illustrate the kinds of problems and



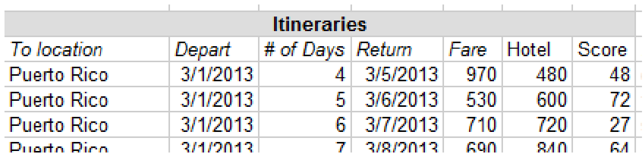
Date	Preference
3/1/2013	4
3/2/2013	5
3/3/2013	5
3/4/2013	6
3/5/2013	6

(a) Departure date and trip duration preferences



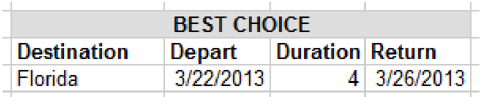
Destination	Preference
Puerto Rico	10
Florida	5
Lake Michigan	5

(b) Destination preferences



Itineraries						
To location	*Depart*	*# of Days*	*Return*	*Fare*	Hotel	Score
Puerto Rico	3/1/2013	4	3/5/2013	970	480	48
Puerto Rico	3/1/2013	5	3/6/2013	530	600	72
Puerto Rico	3/1/2013	6	3/7/2013	710	720	27
Puerto Rico	3/1/2013	7	3/8/2013	690	840	64

(c) Candidate itineraries, enumerated using formulas



BEST CHOICE			
Destination	**Depart**	**Duration**	**Return**
Florida	3/22/2013	4	3/26/2013

(d) Best choice, calculated by the spreadsheet formulas

Figure 8. A vacation itinerary is chosen using a spreadsheet with preference tables and formulas.

solutions that we are aiming to support. Also note that these use only standard spreadsheet functions and syntax.

Example: Vacation

Vicky wants to take a vacation sometime during March and she has three destinations in mind and a range of one month in which she might take the vacation. She is interested in taking a trip between 4 and 7 days. Cost is important. Any of the destinations would be acceptable depending on the cost, but she prefers Puerto Rico over the rest.

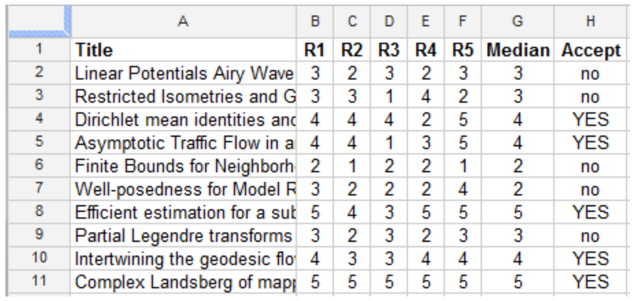
She makes a table of all of the travel dates that are compatible with her work schedule. She lists her preferences for each as a number from 1 to 10. She does the same for the candidate destinations. As for durations, she will take whichever is cheapest overall. She has already researched hotels and found the best deal in each of the three cities.

Vicky makes a table of every combination of destination, departure date, and duration (number of days). These combinations can be enumerated using spreadsheet formulas. Next to each, she has a formula to calculate the score for that option. Finally, at the top, she has a formula that selects the best option of all.

Since different travel web sites sometimes have different airfares, she asks workers to search for fares on several sites and find the best possible fare.

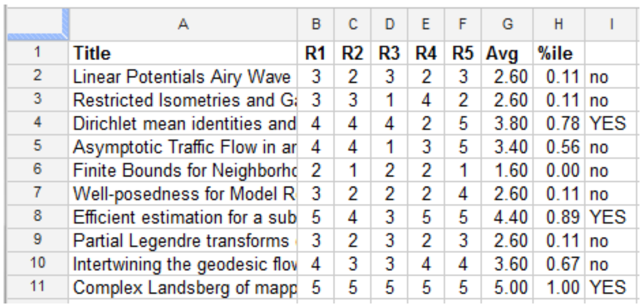
Example: Reviewing conference paper submission

Let us consider a very simplified review process. The XYZ Conference normally gives each paper three reviews to start with, and then requests more, if needed to form a strong consensus. They will accept any paper where the median of five reviews is at least 4.0 out of 5.0. If all five reviewers read each paper, then the results would be as shown in Figure 7a.



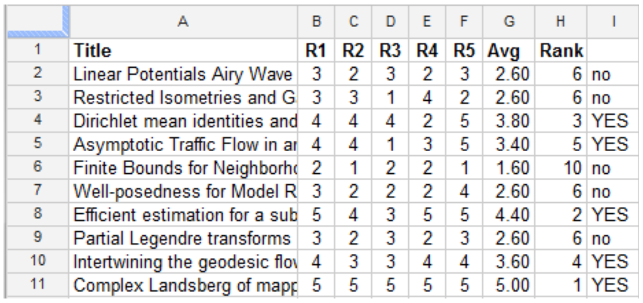
	A	B	C	D	E	F	G	H
1	**Title**	**R1**	**R2**	**R3**	**R4**	**R5**	**Median**	**Accept**
2	Linear Potentials Airy Wave	3	2	3	2	3	3	no
3	Restricted Isometries and G	3	3	1	4	2	3	no
4	Dirichlet mean identities anc	4	4	4	2	5	4	YES
5	Asymptotic Traffic Flow in a	4	4	1	3	5	4	YES
6	Finite Bounds for Neighborh	2	1	2	2	1	2	no
7	Well-posedness for Model R	3	2	2	2	4	2	no
8	Efficient estimation for a sul	5	4	3	5	5	5	YES
9	Partial Legendre transforms	3	2	3	2	3	3	no
10	Intertwining the geodesic flo	4	3	3	4	4	4	YES
11	Complex Landsberg of mapp	5	5	5	5	5	5	YES

(a) Accept if median of 5 scores is ≥4.0



	A	B	C	D	E	F	G	H	I
1	**Title**	**R1**	**R2**	**R3**	**R4**	**R5**	**Avg**	**%ile**	
2	Linear Potentials Airy Wave	3	2	3	2	3	2.60	0.11	no
3	Restricted Isometries and G	3	3	1	4	2	2.60	0.11	no
4	Dirichlet mean identities and	4	4	4	2	5	3.80	0.78	YES
5	Asymptotic Traffic Flow in ar	4	4	1	3	5	3.40	0.56	no
6	Finite Bounds for Neighborhc	2	1	2	2	1	1.60	0.00	no
7	Well-posedness for Model R	3	2	2	2	4	2.60	0.11	no
8	Efficient estimation for a sub	5	4	3	5	5	4.40	0.89	YES
9	Partial Legendre transforms	3	2	3	2	3	2.60	0.11	no
10	Intertwining the geodesic flov	4	3	3	4	4	3.60	0.67	no
11	Complex Landsberg of mapp	5	5	5	5	5	5.00	1.00	YES

(b) Accept top 25% by average score



	A	B	C	D	E	F	G	H	I
1	**Title**	**R1**	**R2**	**R3**	**R4**	**R5**	**Avg**	**Rank**	
2	Linear Potentials Airy Wave	3	2	3	2	3	2.60	6	no
3	Restricted Isometries and G	3	3	1	4	2	2.60	6	no
4	Dirichlet mean identities and	4	4	4	2	5	3.80	3	YES
5	Asymptotic Traffic Flow in ar	4	4	1	3	5	3.40	5	YES
6	Finite Bounds for Neighborhc	2	1	2	2	1	1.60	10	no
7	Well-posedness for Model R	3	2	2	2	4	2.60	6	no
8	Efficient estimation for a sub	5	4	3	5	5	4.40	2	YES
9	Partial Legendre transforms	3	2	3	2	3	2.60	6	no
10	Intertwining the geodesic flov	4	3	3	4	4	3.60	4	YES
11	Complex Landsberg of mapp	5	5	5	5	5	5.00	1	YES

(c) Accept top 5 papers by average score

Figure 7. Paper reviewing could be modeled in various ways, any of which could be optimized to save effort. The titles shown are computer-generated nonsense titles.

Note that in this case, since the papers under submission must be treated as confidential and the reviewers must be trusted experts in the subject area, web workers will not be used. The input form will be confined to a closed system, accessible only by the reviewers.

After three people have read the paper, if all of the reviews are less than 4.0, the median of five people could not possibly be 4.0 or greater. Thus, assigning the paper to any more reviewers would be a waste of effort. Appsheet would initially assign each paper to three reviewers. Based on those initial reviews, it would automatically request additional reviews, only if needed.

XYZ could have structured the process in many alternative ways, any of which could be automatically coordinated in the same way. For example, if the conference was targeting a 25% acceptance rate, it could accept every paper in the top 25 percentile (Figure 7b). For this method, cells H2 and I2 would have the following formulas:

```
H2:   =PERCENTRANK(G1:G11, G2)
I2:   =IF(H2>=0.75, "YES", "no")
```

If a specific number of accepted papers was desired, then the `RANK(…)` function could be used. Figure 7c illustrates a model for accepting the top 5 papers.

In this case, cells H2 and I2 would have the following formulas.

```
H2:   =RANK(G2, G1:G11)
I2:   =IF(H2<=5, "YES", "no")
```

Once there is enough information to determine which papers have rank ≤5, the process stops. It is not necessary to know the exact average score of any paper, as long as it is conclusively known which are accepted and which are not.

Example: Graduate school applications

Gary, an undergraduate student, has decided to submit applications to 6 graduate programs. The number is based on consultations with an undergraduate advisor, as well as Gary's ability to pay the application fees and test reporting fees. He wants to choose programs that have a lot of HCI faculty, preferably as part of an HCI lab in within a computer science department. However, an information studies program would also be acceptable. He also has preferences regarding proximity to his hometown, weather, local transportation system, political leaning of the state, and so on. Despite the many guides available online, none has all of this information. However, for a small amount of money, he can hire web workers (i.e., from Mechanical Turk) to gather the information for him. This could save time while he focuses on his current classes. It may also help him make a decision based on more complete information than he would have gathered on his own.

This model will be larger and more complex than the ones seen so far, but it will be similar to the last version of the paper reviewing model, using the `RANK(..)` function to choose the 6 institutions that best fit Gary's criteria.

FUTURE WORK

The prototype described here was created to demonstrate what we see as a big idea in crowdsourcing, and coordination of large, distributed teams online. What we have now does just enough to demonstrate the method. As with any research project, it is not intended as a marketable product. We intend to keep growing it. We have just begun to test with human helpers other than ourselves.

Since users of Appsheet will likely be soliciting similar information and even making very similar models for similar goals, we plan to enable users to share their models and the input data with one another, for mutual benefit.

Part of what makes the Appsheet idea unique is that it is not confined to any one source of human help. However, if future models make use of *both* paid web workers and trusted friends and colleagues, the system will need to support a way to define the relative costs of different helpers' time. For example, is it more "expensive" to have web workers do fifty tasks or ask co-workers to do ten tasks.

The system will also need to support a wider variety of request mediums. This will most likely include instant messaging and social networks, in addition to Mechanical Turk and email, which are currently supported.

Although our primary focus is on the problems and the request flow, we do plan to refine the user experience of creating models, starting the requests, and viewing progress. To support users who are not experts in spreadsheets, we plan to build simpler interfaces for common models, effectively hiding the spreadsheet details.

For users who do work directly with a spreadsheet, we plan to study how well they understand the state of the model and issues of uncertainty when the model contains incomplete or untrusted information. The system will also need better facilities for filtering out erroneous inputs and handling multiple responses for the same input request. We believe a good way to handle quality for anonymous paid web workers will be to give them a way to demonstrate where they found the information, most likely by a URL, and perhaps some description (i.e., XPath) of where in the page the data was found.

Currently the system does not work well with large models due to performance constraints. We are currently working to improve the performance of the prioritization algorithms.

Finally, it should be noted that nearly all attention so far has been focused on the primary user, the person who makes the model and initiates the activity. However, we believe more attention to the worker's interface will support better relations with workers, better quality, and a more viable long-term outlook for this capability.

ACKNOWLEDGEMENTS

We are grateful to Nick Chen, Lise Getoor, Chang Hu, Adam Marcus, Atif Memon, Kent Norman, Yaron Shlomi, Ben Shneiderman, Keith Walker, Tom Yeh, and many others for their kind advice and support as we have developed this project.

REFERENCES

1. Abraham, R., M. Burnett, and M. Erwig. Spreadsheet Programming. In *Wiley Encyclopedia of Computer Science and Engineering.* (2009), 2804-2810.
2. Amazon Mechanical Turk. www.mturk.com (Accessed Nov. 1, 2011)
3. Amershi, S. and Morris, M.R. CoSearch: a system for co-located collaborative web search. In *Proc. of CHI 2008*, ACM (2008), 1647-1656.
4. Bernstein, M.S., Little, G., Miller, R.C., et al. Soylent: a word processor with a crowd inside. In *Proc. of UIST 2010*, ACM (2010), 313-322.

5. Bigham, J.P., Jayant, C., Ji, H., et al. VizWiz: nearly real-time answers to visual questions. In *Proc. of UIST 2010*, ACM (2010), 333-342.
6. Bollacker, K., Evans, C., Paritosh, P., Sturge, T., and Taylor, J. Freebase: a collaboratively created graph database for structuring human knowledge. In *Proc. of SIGMOD 2008*, ACM (2008), 1247-1250.
7. Bozzon, A., Brambilla, M., Ceri, S., and Fraternali, P. Liquid query: multi-domain exploratory search on the web. *Proc. of WWW 2010*, ACM (2010), 161-170.
8. Casimir, R. J. Real Programmers Don't Use Spreadsheets. In *ACM SIGPLAN Notices* 27.6 (1992), 10-16.
9. ChaCha. www.chacha.com. (Accessed Nov. 1, 2011)
10. Chi, E.H., Pirolli, P., Chen, K., and Pitkow, J. Using information scent to model user information needs and actions and the Web. In *Proc. of CHI 2001*, ACM (2001), 490-497.
11. Calore, M. Get Rich (or at Least Paid) Quick. *Wired Magazine*. Sept. 28, 2006. Available at: www.wired.com/science/discoveries/news/2006/09/71850
12. Dai, P., Mausam, Weld, D. S. Decision-theoretic control of crowd-sourced workflows. In Proc. of *AAAI 2010* (2010).
13. Dearman, D. and Truong, K. N. 2010. Why users of Yahoo!: answers do not answer questions. In *Proc. of CHI 2010*. ACM, New York, NY, USA, 329-332.
14. Donato, D., Bonchi, F., Chi, T., and Maarek, Y. Do you want to take notes?: Identifying research missions in Yahoo! Search Pad. In *Proc. of WWW 2010*, ACM (2010), 321-330.
15. Facebook Questions. www.facebook.com/questions/ (Accessed Nov. 1, 2011)
16. Franklin, M., Kossmann, D., Kraska, T., Ramesh, S., and Xin, R. CrowdDB: Answering queries with crowdsourcing. *Proceedings of SIGMOD 2011*, ACM (2011).
17. Frei, B. Paid crowdsourcing: Current state & progress toward mainstream business use. Produced by Smartsheet.com, (2009).
18. Gilad-Bachrach, R., Bar-Hillel, A., and Ein-Dor, L. Efficient human computation: the distributed labeling problem. In *Proc. of the ACM SIGKDD Workshop on Human Computation*, ACM (2009), 70-76.
19. Kittur, A., Smus, B., and Kraut, R. CrowdForge: crowdsourcing complex work. In *Proc. of CHI 2011*, ACM (2011), 1801-1806.
20. Kulkarni, A.P., Can, M., and Hartmann, B. Turkomatic: automatic recursive task and workflow design for Mechanical Turk. In *Proc. of CHI 2011*, ACM (2011), 2053-2058.
21. Lada A. Adamic, Jun Zhang, Eytan Bakshy, and Mark S. Ackerman. 2008. Knowledge sharing and Yahoo answers: everyone knows something. In *Proc. of WWW 2008*, ACM (2008). 665-674.
22. Law, E., von Ahn, L., and Mitchell, T. Search war: a game for improving web search. In *Proc. of KDD 2009*, W*orkshop on Human Computation*, ACM (2009), 31-31.
23. Levene, Mark. An Introduction to Search Engines and Web Navigation. Wiley (2010). 191, 352.
24. Little, G., Chilton, L.B., Goldman, M., and Miller, R.C. Turkit: tools for iterative tasks on Mechanical Turk. In *Proc. of KDD 2009*, W*orkshop on Human Computation*, ACM (2009), 29-30.
25. Ma, H., Chandrasekar, R., Quirk, C., and Gupta, A. Page hunt: improving search engines using human computation games. In *Proc. of SIGIR 2009*, ACM (2009), 746-747.
26. Marcus, A., Wu, E., Karger, D.R., Madden, S.R., and Miller, R.C. Crowdsourced databases: Query processing with people. In *Proc. of CIDR* (2011).
27. Marcus, A., Wu, E., Karger, D.R., Madden, S.R., and Miller, R.C. Crowdsourced databases: Human-powered sorts and joins. In *Proc. of VLDB 2011*, (2011).
28. Minder, P. and Bernstein, A. CrowdLang-First Steps Towards Programmable Human Computers for General Computation. (2011).
29. Morris, M.R. and Horvitz, E. SearchTogether: an interface for collaborative web search. In *Proc. of UIST 2007*. ACM (2007), 3-12.
30. Morris, M.R., Lombardo, J., and Wigdor, D. WeSearch: supporting collaborative search and sensemaking on a tabletop display. In *Proc. of CSCW 2010*. ACM (2010), 401-410.
31. Nunamaker Jr, J.F., Briggs, R.O., Mittleman, D.D., Vogel, D.R., and Balthazard, P.A. Lessons from a dozen years of group support systems research: A discussion of lab and field findings. *Journal of Management Information Systems 13*, 3 (1996), 163-207.
32. Pirolli, P. and Card, S.K. Information foraging. *Psychological Review* 106(4). (1999), 643-675.
33. Quinn, A.J. and Bederson, B.B. Human computation: a survey and taxonomy of a growing field. In *Proc. of CHI 2010*, ACM (2011), 1403-1412.
34. Russell, D.M., Stefik, M.J., Pirolli, P., and Card, S.K. The cost structure of sensemaking. In *Proc. of the INTERACT 1993 and CHI 199*, ACM (1993), 269-276.
35. Russell, S. and Norvig, P. *Artificial Intelligence: A Modern Approach*. Prentice Hall (1995), 487-490.
36. Simon, H.A. Theories of bounded rationality. *Decision and organization 1*, (1972), 161-176.
37. Technology Quarterly: Artificial Artificial Intelligence. *The Economist*. June 10, 2006. Available at: http://www.economist.com/node/7001738
38. Tong, S. and Koller, D. Support vector machine active learning with applications to text classification. *The Journal of Machine Learning Research 2*, (2002), 45-66