tDP: An Optimal-Latency Budget Allocation Strategy for Crowdsourced MAXIMUM Operations

Vasilis Verroios
Stanford University
verroios@stanford.edu

Peter Lofgren
Stanford University
plofgren@stanford.edu

Hector Garcia-Molina
Stanford University
hector@cs.stanford.edu

ABSTRACT

Latency is a critical factor when using a crowdsourcing platform to solve a problem like entity resolution or sorting. In practice, most frameworks attempt to reduce latency by heuristically splitting a budget of questions into rounds, so that after each round the answers are analyzed and new questions are selected. We focus on one of the most extensively studied crowdsourcing operations, the MAX operation (finding the best element in a collection under human criteria), and we study the problem of budget allocation into rounds for this operation. We provide a polynomial-time dynamic-programming budget allocation algorithm that minimizes the latency when questions form tournaments in each round. Furthermore, we study the general case where questions can be asked in any arbitrary way in each round. Our theoretical results for the general case indicate that our approach is also optimal under certain worst and average-case scenarios. We compare our approach to alternatives on Amazon Mechanical Turk, where many of our theory assumptions do not necessarily hold. We find that our approach is also optimal in practice and achieves a notable improvement over alternatives in most cases.

1. INTRODUCTION

Crowdsourced operators like sort [15] or filtering [17] have been well studied over the last years by the database community. The two main dimensions explored are *cost* and *accuracy*. Therefore, in most cases, a budget of maximum questions to ask is given and the objective is to maximize accuracy. In other cases, a threshold for accuracy is given and the objective is to reach this threshold spending the least number of questions possible.

In this paper, we focus on a third dimension: *latency*. While humans can answer questions that machine algorithms cannot yet handle, they usually need a substantial amount of time to answer each question. For many crowdsourcing applications, latency is a critical factor: for example, consider monitoring the passengers in an airport against a database with images of known terrorists or choosing the next move in time-controlled chess.

The most typical approach of using a crowdsourcing platform, is to send questions in successive rounds. That is, a crowdsourced operator sends some questions in the first round, receives the answers on these questions, uses the answers to select the questions for the second round, and so on, until the budget of questions is spent or until the attained accuracy is above the given threshold. Given this mode of operation, an interesting cost–latency tradeoff appears; or an accuracy–latency tradeoff, depending on the problem setting.

For example, consider trying to find the best political-campaign response to an opponent's attack one day before the elections, over a collection of 1000 responses, using pairwise comparisons. In one extreme, we first pick two random responses, ask a question between them, get back the answer, and find the best out of the two. (Let us assume for now that we need a single comparison to get the best out of two responses, i.e., workers are not malicious and are always right.) Next, we compare the winner of the first comparison with a third random response from the collection. We continue like this until only one response is left; the one that didn't lose any comparisons. This strategy needs $1000 - 1$ questions to reach to the best response. In the other extreme, we ask all $\binom{1000}{2}$ questions in one round, and we reach to the best response when the answers we get back indicate that a single response has won over all the others. Clearly, asking one question at a time will save a huge number of questions. However, if we can afford the cost of $\binom{1000}{2}$ questions and if there is a large number of workers interested in our task, asking all questions in one round may prove beneficial in terms of latency. For instance, if there are more than $\binom{1000}{2}$ platform workers interested in the task, and each worker needs exactly one minute to answer a question, the first strategy would require $1000-1$ minutes (i.e., around 17 hours), while the second one would require only 1 minute; assuming the time to process the answers is negligible and no other delays take place. On the other hand, if there was just a single worker interested in our task, the first strategy would, again, require $1000-1$ minutes, while the second one could go on for $\frac{1000*999}{2}$ minutes; assuming that for almost one year this dedicated worker is answering our questions on this collection, day and night.

The same cost–latency tradeoff also appears in many other crowdsourcing operations, e.g., Entity Resolution [27] or Graph Search [16]. When we started our work on latency and budget allocation, we faced a choice: either focus on

SIGMOD'15, May 31–June 4, 2015, Melbourne, Victoria, Australia.
Copyright © 2015 ACM 978-1-4503-2758-9/15/05 ...$15.00.
http://dx.doi.org/10.1145/2723372.2749440.

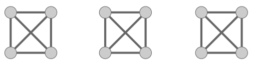


Figure 1: Example of three 4-element tournaments.

a specific crowdsourcing operator or study a general framework. We chose to focus on one specific operator because knowledge of the task at hand can impact the way questions are best allocated (more on this later). Thus, we felt it was important to fully understand latency and quantify potential improvements in a specific scenario before trying to generalize. The operator we focus on is finding the MAX element of a collection using pairwise comparisons. This problem is relatively simple, yet there are many opportunities for allocating questions wisely and for reducing latency.

In its most common setting, the MAX operator receives as input a vector of integers indicating how many questions should be posted in each round. The selection of the specific questions in each round, is done by a *question selection* algorithm that takes into account the answers to all the questions issued in rounds previous to the current one. At some point (before or after the end of the last round), the MAX operator decides that the answers retrieved, identify the MAX element in the collection.

Our focus in this paper is the computation of a good input vector for the MAX operator. That is, given an overall budget of questions that we can afford, we are interested in finding an allocation of this budget into rounds such that the overall latency until reaching to the MAX is minimized. For instance, in the debate-response example, when we were asking all $\binom{1000}{2}$ questions in one round, we were implicitly allocating our whole budget into a single round. Another possible allocation would be, for example, to split the budget of $\binom{1000}{2}$ questions into 10 batches, by allocating the same number of questions into each round. We call the algorithms that compute such allocations, *budget allocation* algorithms.

In addition to the overall budget of questions, we assume that *budget allocation* algorithms are also given as input a latency function $\mathcal{L}(q)$. This function gives an estimate of the time required to get back the answers when q pairwise comparisons are asked in one round. (Such a function can be estimated by the crowdsourcing platform based on statistics about the workers in the platform, their availability in different times during the day, and the type of the task. In Section 6.1, we run experiments to estimate $\mathcal{L}(q)$ for a specific task on Amazon Mechanical Turk.)

Our study focuses mostly on the latency and cost dimensions, assuming that humans do not make errors, or in other words, we need a single comparison to get the best out of two elements. There are several effective approaches for dealing with human errors [10, 12, 13, 14, 17, 22] and, thus, we choose to leave out this aspect of the problem, treating it as an orthogonal issue. Nevertheless, we also study an accuracy aspect of the problem: even when humans are always correct, some approaches may still not achieve a *singleton termination*, i.e., they may end up with more than one elements as candidates for the MAX after all questions are answered, as we will discuss later in the paper.

Most related to our work is the work of [23] that also studies the MAX operation in multiple rounds. However, the main focus of [23] is the accuracy and cost dimensions, with a simplified definition of latency: the total number of rounds. Moreover, there is no optimality guarantee for the hill-climb based heuristics proposed in [23]. Other papers (like [2, 10]), focus on finding the MAX after all (possibly erroneous) answers are retrieved, or focus on selecting which questions to ask in the next round using all previous answers as evidence.

One of the main contributions of this paper is tDP: a dynamic programming *budget allocation* algorithm that computes the optimal allocation of questions into rounds in polynomial time. This algorithm assumes the use of a very specific *question selection* algorithm that forms *tournaments*, in each round. For example, if the allocation for a round is 18 questions and there are 12 candidate elements, the tournament-formation algorithm forms three *tournaments* (cliques), each involving four elements; Figure 1 depicts the three *tournaments* using nodes to represent each element and edges to represent the questions between elements.

At first glance, the *tournament*-like formation of questions may seem a very restrictive way of asking questions in each round. Nevertheless, in Section 4, we present a number of interesting theoretical results showing that our dynamic programming algorithm, together with the tournament-formation question selection, is optimal under certain worst and average case scenarios, compared to any other combination of *budget allocation* and *question selection* algorithms. We also run experiments in Amazon Mechanical Turk, to test the effectiveness of our approach when some of the theory assumptions do not hold or when we have access only to a rough estimate of the latency function $\mathcal{L}(q)$. We compare our approach with several other approaches and we conclude that in all cases the dynamic programming algorithm gives the best outcome and in most cases outperforms the other approaches by a huge margin.

To summarize, our contributions are as follows:

- We formally define the problem of the optimal budget allocation for the MAX operator, in Section 2.
- We propose a dynamic-programming *budget allocation* algorithm, in Section 3.
- We analyze our approach, consisting of the *budget allocation* and *tournament question selection* algorithms, and we prove strong optimality guarantees in the average and worst-case scenarios, in Section 4.
- We present 4 alternative *budget allocation* heuristics that have a low computational cost, in Section 5.
- We experimentally compare our approach with the alternatives on Amazon Mechanical Turk, in Section 6.

2. PROBLEM DEFINITION

In this section, we give a formal definition to the problem of finding the MAX element of a collection, while minimizing the latency. In our problem definition, we assume that questions are selected by a specific *question selection* algorithm that forms *tournaments* between elements in each round. In Section 4, we discuss a generalized version of the problem, where questions can be selected by any *question selection* algorithm in each round, and we prove that the approach we propose is also optimal in that case.

2.1 Preliminaries

Our input is a set of elements, C, with $|C| = c_0$ elements, and a budget threshold, b, expressing the maximum number of questions we can ask, over all rounds. We are interested in finding the MAX element of C, by asking pairwise comparisons in each round. There is a true unknown permutation

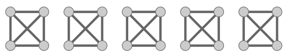


Figure 2: Tournament graph $G_T(20, 5)$.

for the elements of C, that gives the order of the elements; we assume a strict order without equalities between elements. The first element in this order is the MAX element.

We assume that for 2 elements a and b, a single comparison is sufficient for resolving their true relation, based on the unknown true permutation of C. In practice, to handle this error-free assumption a *reliable worker layer* (RWL) can sit between our algorithms and the crowdsourcing platform. Such an RWL can harness techniques proposed in recent studies [10, 12, 13, 14, 17, 22], to handle human errors (or subjectiveness). The input to RWL, in each round, is a set of questions and the output is a conflict-free set of correct answers; with one answer per question. For example, if we ask questions $\{(a, b), (b, c), (c, a)\}$, RWL may issue multiple comparison tasks for each question on the platform and, based on the worker responses, RWL will infer 3 answers such that there is no cycle between elements a, b, and c (e.g., by applying the algorithms proposed in [10]). When we get an answer from RWL indicating that an element a is greater than an element b, we say that *a won* over b.

In each round the questions form tournaments (in Section 4 we relax this assumption) with each tournament giving a single element as a *winner* for the next round; the *winner* is the element that is greater than all other elements in the tournament. We denote this tournament structure by the graph $G_T(c_{i-1}, c_i)$:

DEFINITION 1. ***Tournament Graph,*** **$G_T(c_{i-1}, c_i)$:** *A graph consisting of c_{i-1} nodes that form c_i cliques(tournaments). (As our notation suggests, the c_i winners of $G(c_{i-1}, c_i)$ will be the input for graph $G(c_i, c_{i+1})$.)*

For example, consider $G_T(20, 5)$, depicted in Figure 2. The graph nodes refer to elements and an edge between two nodes represents a comparison between the two respective elements. $G_T(20, 5)$ consists of 20 nodes forming five 4-cliques. After receiving the answers on the 6 questions in each clique-tournament, there will be a single element in the tournament that will have won in all 3 comparisons it was involved in. Note that since we ask all questions between elements in a tournament, and all answers provided by RWL are correct and conflict-free (based on the single underlying permutation for C), there will be exactly one element that wins all the comparisons it is involved in the tournament.

In the example of Figure 2, the 5 winners from the tournaments advance to the next round, where they are, once more, organized in tournaments, e.g., a single tournament of 5 elements, or two tournaments, one with 3 elements and one with 2 elements. Furthermore, we assume a random assignment of the advancing elements to the tournaments of the next round (see Section 5 for the full description of the Tournament-formation algorithm).

Note that the tournaments formed in a round may not be all of the same size. For instance, if instead of 20 we had 24 elements and we still wanted to organize them in 5 tournaments, we would form 4 tournaments of 5 elements and one tournament of 4 elements, as depicted in Figure 3. In the general case, when we have c_{i-1} elements and we want to form c_i tournaments, $c_{i-1} \bmod c_i$ tournaments will involve $\lceil \frac{c_{i-1}}{c_i} \rceil$ elements, and the rest of the tournaments will involve $\lfloor \frac{c_{i-1}}{c_i} \rfloor$ elements.

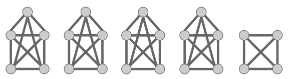


Figure 3: Tournament graph $G_T(24, 5)$.

The number of questions that a tournament graph "spends" in one round is given by function $Q(c_{i-1}, c_i)$:

DEFINITION 2. ***Questions function*** **$Q(c_{i-1}, c_i)$:** *Function giving the number of edges-questions in $G_T(c_{i-1}, c_i)$.*

For example, $G_T(20, 5)$ in Figure 2, uses $\binom{20/5}{2} * 5 = 30$ questions. In general, when c_{i-1} is a multiple of c_i, $Q(c_{i-1}, c_i)$ is given by:

$$Q(c_{i-1}, c_i) = \binom{c_{i-1}/c_i}{2} * c_i \tag{1}$$

However, c_{i-1} will not always be a multiple of c_i. For instance, consider $G_T(24, 5)$ in Figure 3. The first 4 tournaments (from the left), involve 5 elements, while the last one involves just 4. The number of questions in this case is $\binom{\lceil 24/5 \rceil}{2} * (24 \bmod 5) + \binom{\lfloor 24/5 \rfloor}{2} * (5 - 24 \bmod 5) = \binom{5}{2} * 4 + \binom{4}{2} * 1 = 46$. Overall, $Q(c_{i-1}, c_i)$ is given by:

$$\begin{aligned} &\binom{\lceil c_{i-1}/c_i \rceil}{2} \quad * \quad (c_{i-1} \bmod c_i) + \\ &\binom{\lfloor c_{i-1}/c_i \rfloor}{2} \quad * \quad [c_i - (c_{i-1} \bmod c_i)] \end{aligned} \tag{2}$$

Note that when c_{i-1} is a multiple of c_i, i.e, $c_{i-1} \bmod c_i = 0$, equation (2) is equivalent to equation (1).

The latency for getting all answers back in each round is a function of the number of questions we ask. We denote this function by $\mathcal{L}(q)$:

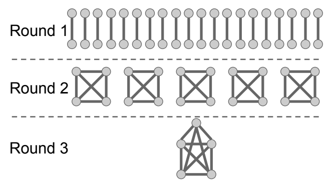
DEFINITION 3. ***Latency function,*** **$\mathcal{L}(q)$:** *Function that expresses the amount of time we have to wait for getting back the answers, when we ask q questions in a single round.*

For example, consider function $\mathcal{L}(q) = 60 + q$ seconds. When we form $G_T(24, 5)$ in one round, we have to wait for $\mathcal{L}(Q(24, 5)) = \mathcal{L}(46) = 106$ seconds to get back the 46 answers. Here, the constant term of 60 seconds in $\mathcal{L}(q)$, expresses an overhead of initiating a new round. The latency function can be based on statistics for a specific crowdsourcing platform and a specific type of task and it models the delays of the RWL. In addition, we assume that latency is an increasing function of q. Note that this assumption does not affect the optimality of our solution to Problem 1, defined in the next section. We only use this assumption to prove optimality for the, more general, Problem 2, in Section 4.1.

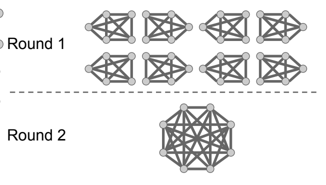
2.2 MinLatency Problem

Our objective is to decide on a sequence of tournament graphs, $G_T(c_0, c_1)$, $G_T(c_1, c_2)$, $\ldots$, $G_T(c_{r-1}, c_r)$, such that the aggregate number of edges-questions is less than or equal to the budget b and the overall latency is minimized. Note that we do not impose any constraint on the number of rounds r that the sequence must extend to.

For instance, consider a budget of $b = 108$ questions for finding the MAX of $c_0 = 40$ elements and the sequence



(a) Sequence $(40, 20, 5, 1)$



(b) Sequence $(40, 8, 1)$

Figure 4: Two tournament graph sequences.

$(c_i)_{i=0}^r = (40, 20, 5, 1)$, depicted in Figure 4(a) by $G_T(40, 20)$ in the first round, $G_T(20, 5)$ in the second round, and $G_T(5, 1)$ in the last round. The aggregate number of edges is $20 + 30 + 10 = 60$, that is, less than b. If we consider a latency function $\mathcal{L}(q) = 100 + q$ seconds, the overall latency to reach to the MAX is $\mathcal{L}(20) + \mathcal{L}(30) + \mathcal{L}(10) = 120 + 130 + 110 = 360$ seconds. Another sequence $(c'_i)_{i=0}^r = (40, 8, 1)$, depicted in Figure 4(b), achieves a lower overall latency without violating the budget constraint. In fact, $(c'_i)_{i=0}^r = (40, 8, 1)$ requires $80 + 28 = 108$ questions and gives an overall latency of $\mathcal{L}(80) + \mathcal{L}(28) = 180 + 128 = 308$ seconds.

The general formulation is given by the *MinLatency* optimization problem below:

PROBLEM 1 **(*MinLatency*)** **.** Given a set of c_0 initial elements, a budget constraint of b questions, and a latency function $\mathcal{L}(q)$, find the sequence $(c_i)_{i=0}^r$ solving:

$$\min_{(c_i)_{i=0}^r} \quad \sum_{i=1}^{r} \mathcal{L}(Q(c_{i-1}, c_i)) \tag{3}$$

$$\text{s.t.} \quad \sum_{i=1}^{r} Q(c_{i-1}, c_i) \leq b \tag{4}$$

$$c_r = 1, \forall i \ c_i \in \mathbb{N} \tag{5}$$

Equation (3) expresses the objective function over all sequences $\{c_i\}$, equation (4) expresses the budget constraint, and equation (5) expresses the obvious constraints, i.e., the last number in the sequence must be 1 (the MAX element), and all numbers in the sequence must be positive integers. (Note that a sequence $(c_i)_{i=0}^{r_1}$ consuming fewer overall questions than a sequence $(c'_i)_{i=0}^{r_2}$, does not necessarily give a lower overall latency. For example, if $r_1 > r_2$, c_i may give a much higher overall latency than c'_i, if the cost of initiating a new round in $\mathcal{L}(q)$ is high.)

Note that the optimization problem in Equations (3) to (5) will always have a solution, as long as the budget is greater than or equal to the number of initial elements (minus one). This property is formally stated in the following theorem.

THEOREM 1. *The* MinLatency *problem in Equations (3) to (5) has a solution if and only if* $b \geq c_0 - 1$.

PROOF.
We consider the following two cases:

- $b < c_0 - 1$: Let us assume that the problem of (3) to (5) has a solution. Each element needs to lose at least one comparison to stop being a candidate for the MAX. Since only one element must remain at the end of all rounds, we need to spend at least one comparison for all the other $c_0 - 1$ elements. Hence, we need a budget of at least $c_0 - 1$ questions and the problem of (3) to (5) cannot have a solution since constraint (4) cannot be satisfied in this case.
- $b \geq c_0 - 1$: Consider the sequence $(c_i)_{i=0}^{c_0 - 1}$ such that $c_{i+1} = c_i - 1$, i.e., in each round we ask a single question between two arbitrarily chosen elements and the winner advances to the next round. This sequence involves $c_0 - 1$ questions and ends with $c_r = 1$. Therefore, this sequence satisfies constraints (4) and (5) and constitutes a solution to the optimization problem.

□

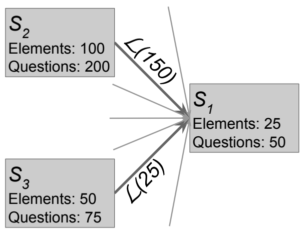


Figure 5: Insight for dynamic programming.

In the rest of the paper, we assume that we are always given a sufficient budget, i.e., $b \geq c_0 - 1$.

It is worth pointing out here that, in Section 4.1, we extend Problem 1 to the more general Problem 2, where questions do not necessarily form tournaments. Interestingly, the algorithm we describe in the next section gives the optimal solution to both problems, as we prove in Section 4.

3. DYNAMIC PROGRAMMING ALGORITHM

The *MinLatency* problem can be solved using dynamic programming. The insight for applying dynamic programming is based on the following observation: the sequence of tournament graphs that minimizes the latency when we are left with c_i elements and q remaining questions, does not depend on the sequence of graphs we followed in the previous rounds until reaching that state. That is, the lowest-latency sequence for reaching from the c_i elements and the q remaining questions, to the MAX element, is the same irrespectively of the sequence we used for reaching from the initial, c_0, elements and b questions to the remaining c_i elements and q questions.

In Figure 5 we use a box to represent a state where, after a number of rounds, we are left with c_i elements (not having lost any comparisons) and q questions (that we haven't used from our initial budget). The arrows express the cost of the transition from one state to another in a single round, i.e., the latency for that round. For instance, in Figure 5, we have a target state S_1 with 25 remaining elements and 50 remaining questions. We can reach S_1 in one round, from many other possible states. For example, we can reach S_1 from S_2 by asking 150 questions, i.e., we form $G_T(100, 25)$ and we reach from 100 elements to 25 elements by spending $Q(100, 25) = 6 * 25 = 150$ questions. A second state from which S_1 can be reached, is S_3. In order to reach S_1 from S_3, we need to form $G_T(50, 25)$, i.e., 25 tournaments with 2 elements each. The observation for dynamic programming is that the lowest-latency sequence from S_1 to the MAX element, does not depend on how we reach to S_1, i.e., through S_2, S_3 or any other possible state with a transition to S_1.

The above observation is formalized by the recursive function $OL(q, c_i)$:

DEFINITION 4. ***Optimal solution's latency,*** $OL(q, c_i)$**:** *The overall latency for the optimal solution to the* MinLatency *problem, given by equation (3), with input c_i elements and q remaining questions.*

For example, if we assume that the sequence $(c_i')_{i=0}^{r} = \{40, 8, 1\}$ (depicted in Figure 4(b)) is the optimal solution to the *MinLatency* problem with input 40 elements and 108 remaining questions, then

$$OL(108, 40) = \mathcal{L}(Q(40, 8)) + \mathcal{L}(Q(8, 1)) = \mathcal{L}(80) + \mathcal{L}(28)$$

The following recursive equation expresses our dynamic programming insight for $OL(q, c_i)$:

$$OL(q,c_i) = \min_{c_{i+1} \in [1,c_i)} \{\mathcal{L}(Q(c_i,c_{i+1})) + OL(q - Q(c_i,c_{i+1}),c_{i+1})\} \quad (6)$$

$$OL(q,1) = 0 \quad (7)$$

Equation (6) refers to all possibilities for the next round. That is, in the next round a tournament graph $G_T(c_i, c_{i+1})$ will be formed, for $c_{i+1} \in [1, c_i)$. For example, if $c_i = 40$ and $c_{i+1} = 20$, the overall latency until finding the MAX will be: *a)* $\mathcal{L}(Q(40, 20))$ for the next round plus *b)* the latency of the optimal solution for an input of 20 elements and a budget of $q - Q(40, 20)$ questions, i.e., $OL(q - Q(40, 20), 20)$. The optimal solution's latency for an input of c_i elements and a budget of q questions (i.e, $OL(q, c_i)$) is the one minimizing the overall latency over all possible next rounds (i.e., $\forall c_{i+1} \in [1, c_i)$).

Equation (7) states that once we reach at a single element (i.e., the MAX), we do not have to "pay" for any additional latency no matter how many questions are left unasked.

Algorithm 1 is the top-down dynamic programming algorithm for solving the *MinLatency* problem. The recursive function, $OL(q, c_i)$ in lines 13-32, implements Equations (6) and (7). In lines 19-21, we examine if the remaining budget after a transition from c_i to c_{i+1} elements, is enough to reach from c_{i+1} to the MAX , based on Theorem 1. In line 1, we allocate a memoization array to avoid re-computing the value of $OL(q, c_i)$, for the same input.

The time complexity of the algorithm is $O(c_0^2 * b)$, while the space complexity is $O(c_0 * b)$. Taking into account that $b \in [c_0 - 1, \binom{c_0}{2}]$, the time complexity ranges from $O(c_0^3)$ to $O(c_0^4)$ and the space complexity ranges from $O(c_0^2)$ to $O(c_0^3)$.

Algorithm 1 DP tournaments (tDP)

```
Input: b: Initial budget of questions
Input: c_0: Initial number of elements
Input: $\mathcal{L}(q)$: Platform's latency function
Output: $(c_i)_{i=0}^r$: optimal solution's sequence
1: $statesReached$ = allocate $b \times c_0$ array
2: $statesNext$ = allocate $b \times c_0$ array
3: run $OL(b, c_0)$
4: append $c_0$ to $(c_i)_{i=0}^r$
5: $c_i = c_0$
6: while $c_i > 1$ do
7:     $c_{i+1} = statesNext[b][c_i]$
8:     append $c_{i+1}$ to $(c_i)_{i=0}^r$
9:     $b = b - Q(c_i, c_{i+1})$; $c_i = c_{i+1}$
10: end while
11: return $(c_i)_{i=0}^r$
12:
13: Function $OL(q, c_i)$   #implements Equations (6) and (7)
14: if $c_i$ is 1 then
15:     return 0
16: end if
17: $l = \infty$
18: for $c_{i+1}$ in $[1, c_i]$ with step 1 do
19:     if $q - Q(c_i, c_{i+1}) < c_{i+1} - 1$ then
20:         continue to line 18
21:     end if
22:     if $statesReached[q - Q(c_i, c_{i+1})][c_{i+1}]$ is NULL then
23:         $l' = OL(q - Q(c_i, c_{i+1}), c_{i+1})$
24:     else
25:         $l' = statesReached[q - Q(c_i, c_{i+1})][c_{i+1}]$
26:     end if
27:     if $\mathcal{L}(Q(c_i, c_{i+1})) + l' < l$ then
28:         $l = \mathcal{L}(Q(c_i, c_{i+1})) + l'$; $next = c_{i+1}$
29:     end if
30: end for
31: $statesReached[q][c_i] = l$; $statesNext[q][c_i] = next$
32: return $l$
```

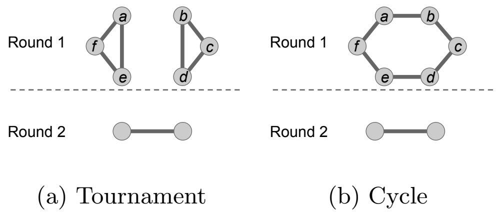


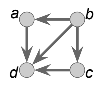
Figure 6: Allocation $(6, 1)$ under two different *question selection* algorithms.

4. THEORETICAL RESULTS

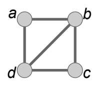
The *MinLatency* optimization problem given by equations (3) to (5), assumes questions are formed in tournaments in each round. Here, we compare the tournament approach with a more general approach: questions are still asked in rounds, but they can be selected in each round by any *question selection* algorithm. That is, in each round questions do not have to form a tournament graph, but can instead form any graph. The elements that win in all of the comparisons they are involved in, advance to the next round; just like in the tournament graphs where the winners of the tournaments are the only elements not having lost any comparison.

An example of the general approach is given in Figures 6(a) and 6(b). In this example, the number of initial elements, c_0, is 6, while the budget, b, is 7. Consider the allocation $(6, 1)$ for the budget of 7 questions, i.e., 6 questions in the first round, and 1 question in the second round. Figure 6(a) shows the question selection by the tournament *question selection* algorithm we have discussed so far. That is, $G_T(6, 2)$ is formed in the first round and $G_T(2, 1)$ in the second round. A different *question selection* algorithm could have selected questions as shown by Figure 6(b), i.e., form a cycle-graph in the first round and then ask a single question between two elements that advance to the second round; if more than one elements advance.

While in the tournament approach we know exactly how many elements advance from one round to the next, this is not the case with other *question selection* algorithms. For instance, in Figure 6(b), 1 to 3 elements may advance from the first round to the second round. (An example where 3 elements advance is when the answers we receive in the first round are $\{(a > b), (a > f), (c > b), (c > d), (e > d), (e > f)\}$, while an example where just 1 element advances is when the answers are $\{(a > f), (a > b), (b > c), (c > d), (d > e), (e > f)\}$.) In case just 1 element advances (the MAX), we will have a reduced latency compared to the tournament approach, since there will be no reason to run round 2. However, if 3 elements advance, then the single question will not be enough to find the MAX in the second round: 2 elements will remain after the second round and 1 of them will have to be selected as the MAX. We call the event of being left with exactly one element after (or even before) all questions are asked, *singleton termination*. In our experimental evaluation, we study the probability of *singleton termination* for different *question selection* algorithms.



(a) DAG



(b) Undirected

Figure 7: Example of DAG answer-represention.

The scenario where 3 elements advance in the second round, in Figure 6(b), is the worst case scenario for this example. (Note that we cannot have more than 3 elements not losing any comparisons in the first round.) In the worst case analysis we perform in this section, we assume that the worst case scenario happens in each round. In Appendix A, we also perform an average case analysis, where we focus on the expected number of elements that advance from one round to the next.

Our theoretical results consist of:

(1) A proof of optimality over multiple rounds in the worst-case scenario, for tDP combined with the tournament *question selection* algorithm. That is, tDP combined with the tournament-formation algorithm is guaranteed to give a latency lower than the latency achieved by any other combination of a *budget allocation* and a *question selection* algorithm, in the worst-case scenario (details in Section 4.1).

(2) A proof of optimality over a single round in the average-case scenario, for the tournament *question selection* algorithm. That is, the tournament-formation algorithm minimizes the expected number of elements that advance from a single round to the next one, under certain assumptions. In Appendix A, we discuss the details, including the assumptions and how tDP is related to those assumptions. Note that tDP combined with the tournament-formation algorithm is not necessarily optimal for the average-case scenario. Nevertheless, in our experimental evaluation where we examine what happens in the average case, tDP with the tournament-formation algorithm is substantially better than any other alternative we tried.

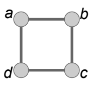
In the discussion of our theoretical results, we use a Directed Acyclic Graph (DAG) representation for the answers we get back from humans. A directed edge from node b to node a, expresses the answer $(a > b)$ to the comparison question asked between the two nodes. For example, in Figure 7(a), we ask the questions $\{(a, b), (b, c), (c, d), (d, a), (b, d)\}$ and we get back the answers $\{(a > b), (c > b), (d > c), (d > a), (d > b)\}$. Note that the answers we get back cannot form a cycle in the directed graph, since our answers are always correct and are based on the true unknown permutation of the initial elements that defines a strict element order. In practice, RWL resolves any formed cycles using methods like the ones proposed in [10].

4.1 Worst Case Analysis

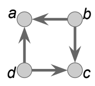
Before discussing our main result in Theorem 4, we give some definitions and results that will simplify the discussion for the main result.

DEFINITION 5. ***Remaining Candidates*** (RC) ***set of a DAG*** **:** *The set of nodes that do not have any outgoing edges in the DAG.*

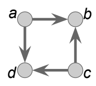
The set RC in our context, gives the elements that advance from one round to the next one, and remain as candidates for the MAX. For example, in Figure 7(a), $RC = \{d\}$ because d is the only element that has not lost any comparison; and since there are no other candidates remaining, d is the MAX element.



(a) Undirected



(b) DAG 1



(c) DAG 2

Figure 8: Example of *maxRC* sets of a graph.

DEFINITION 6. ***Maximum Remaining Candidates*** $(maxRC)$ ***set of an undirected graph*** G **:** *The largest RC set, for any DAG that can be formed by giving directions to the edges of G.*

For example, consider the undirected graph G in Figure 8(a). One $maxRC$ for G is given by the DAG of Figure 8(b) and consists of $\{a, c\}$. Note that, in this case, the $maxRC$ is not unique: the DAG of Figure 8(c) also gives an RC of 2 elements, consisting of $\{b, d\}$. On the contrary, the $maxRC$ of the graph in round 1 in Figure 9(a), is unique and consists of $\{a, c, e, g, i, k\}$.

The $maxRC$ of a graph G, expresses the worst case scenario when we ask the questions on G. That is, $maxRC$ gives the maximum possible number of remaining candidates for the MAX element, after asking the questions on G.

DEFINITION 7. ***Maximum Independent*** $(maxIND)$ ***set of an undirected graph*** G **:** *An independent set of a graph* $G = (V, E)$ *is a subset of* V *such that there is no edge in* E *connecting two nodes of this subset. The maxIND set is the independent set of G with the most nodes.*

Consider the undirected graph G, for the DAG of Figure 7(a), depicted in Figure 7(b). Set $\{d\}$ is an independent set of G. However, the $maxIND$ of G, is set $\{a, c\}$.

Next, we prove a theorem that will enable us to use a graph theory result from [4]. This graph theory result will be our main building block for Theorem 4.

THEOREM 2. *For any undirected graph G, a set of nodes is a maxIND set if and only if it is a maxRC set.*

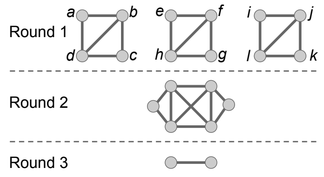
PROOF. The proof of Theorem 2 will be based on the Lemmas 1 and 2:

LEMMA 1. *For any undirected graph G, the maxRC set is an independent set of G.*

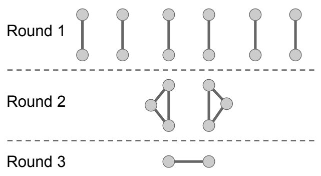
PROOF. We prove the contrapositive. Suppose that the $maxRC$ set, R, of an undirected graph G, is not an independent set of G. Then, there must be an edge e between two nodes a and b of R. Consider the DAG of G, for which R is an RC set. We give a direction to edge e, such that no cycle is formed in this DAG. By giving a direction to e, one of a or b acquires an outgoing edge and is no longer a remaining candidate. Hence, R cannot be an RC set.

LEMMA 2. *For any undirected graph G, the maxIND set is an RC set for a DAG of G.*

PROOF. We prove the contrapositive. Suppose that the $maxIND$ set is not an RC set. We can construct a DAG for G using the following process: *a)* pick a permutation p for the nodes of G, such that the top-$|maxIND|$ elements in the permutation are the nodes of the $maxIND$ set (order others arbitrarily) and *b)* for each edge e, give e a direction based on p, i.e., from the first endpoint, a, to the second, b, if b is greater than a in p and from b to a otherwise. Based on this process, no one of the nodes in the $maxIND$ set will have an outgoing edge: there are no edges between nodes in the $maxIND$ set (by the independent set definition), and for an edge between a node $a \in maxIND$ and a node $b \notin maxIND$, the direction can only be from b to a, because in p the top-



(a) non-tournament graphs



(b) tournament graphs

Figure 9: Examples of graph sequences for the *Generalized Worst MinLatency* problem.

$|maxIND|$ elements are the nodes of *maxIND*. Hence, we found a DAG for which the *maxIND* set is an *RC* set.

Based on Lemmas 1 and 2, for any undirected graph G, a *maxRC* set is an independent set and a *maxIND* set is also an *RC* set. Therefore, a set of nodes is a *maxIND* set if and only if it is a *maxRC* set.

□

Our optimality guarantees for the worst case scenario are based on a known graph theory result found in [4], summarized by the following theorem.

THEOREM 3. *For any positive integers c_{i-1} and c_i, any undirected graph $G = (V, E)$ that has $|V| = c_{i-1}$ and a $|maxIND| = c_i$, must have $|E| \geq Q(c_{i-1}, c_i)$, i.e., at least as many edges as the tournament graph $G_T(c_{i-1}, c_i)$.*

Theorem 3 suggests that we can reduce the number of questions we ask in each round, by replacing an arbitrary graph with a tournament graph. Figure 9(b), gives an example of replacing the graph sequence of Figure 9(a) with tournament graphs: $G_T(12, 6)$ in the first round, $G_T(6, 2)$ in the second round, and $G_T(2, 1)$ in the last round. (Note that, in Figure 9(a), the *maxRC* of the graph in Round 1, i.e., set $\{a, c, e, g, i, k\}$, has a size of 6, while the *maxRC* of the graph in Round 2 has a size of 2.) The number of questions in the first round drops from 15 to 6 and in the second round from 10 to 6. Thus, the new graph sequence allows us to find the MAX element with fewer questions. We formally state this implication of Theorem 3 in Lemma 3.

Now, let us define a generalized version of the *MinLatency* problem, where in each round questions can form any arbitrary graph. Using Theorems 2 and 3, we will prove that the Algorithm 1 optimally solves the general problem.

Our domain in the general problem consists of any sequence of graphs $(G_i = (V_i, E_i))_{i=0}^{r}$, such that $|V_{i+1}| = |maxRC(G_i)|$, i.e., the number of vertices for the graph in round $i + 1$ must be equal to the size of the *maxRC* set of the graph in the previous round. In other words, after each round, we examine the worst possible scenario, i.e., being left with the maximum number of candidates possible; given by the *maxRC* of the graph in that round. For example, in Figure 9(a), in the first round the graph of questions has a *maxRC* set of 6 elements, $\{a, c, e, g, i, k\}$, in the second round the graph of the 6 nodes has a *maxRC* set of 2 elements, which is the number of nodes in the graph of the third round.

As opposed to the *MinLatency* problem, where the graph in each round must be a tournament graph, here the domain is broader by allowing any graph of questions in each round. Still, the budget constraint and the objective function remain the same:

PROBLEM 2 (***Generalized Worst MinLatency***). Given a set of c_0 initial elements, a budget constraint of b questions, and a latency function $\mathcal{L}(q)$, find the sequence of graphs $(G_i = (V_i, E_i))_{i=0}^{r}$ solving:

$$\min_{(G_i=(V_i,E_i))_{i=0}^{r}} \quad \sum_{i=0}^{r} \mathcal{L}(|E_i|) \tag{8}$$

$$\text{s.t.} \quad \sum_{i=0}^{r} |E_i| \leq b \tag{9}$$

$$|V_0| = c_0, |maxRC(G_r)| = 1, \quad \forall i \; |V_{i+1}| = |maxRC(G_i)| \tag{10}$$

Note that in equation (10) the constraint $|maxRC(G_r)| = 1$ indicates that the *maxRC* set of the last round graph must have a size of 1, i.e., even in the worst case we will be left with a single element (the MAX element).

We will prove that Algorithm 1 optimally solves the *Generalized Worst MinLatency* problem using the following lemma:

LEMMA 3. *Consider a graph sequence $\{(G_i = (V_i, E_i))_{i=0}^{r}$ satisfying equations (9) and (10). We can replace each graph $G_i = (V_i, E_i)$ in the sequence with the tournament graph $G_T(|V_i|, maxRC(G_i))$ and reduce the overall cost given by equation (8).*

PROOF. Based on Theorem 3, $G_i = (V_i, E_i)$ cannot have fewer edges than $G_T(|V_i|, |maxIND(G_i)|)$. Based on Theorem 2, the *maxIND* and *maxRC* sets are equivalent, therefore, $G_T(|V_i|, |maxIND(G_i)|) \equiv G_T(|V_i|, |maxRC(G_i)|)$. Thus, the number of questions we ask in each round will be reduced or remain the same. Taking into account the fact that $\mathcal{L}(q)$ is an increasing function, we conclude that the overall cost given by equation (8) will be reduced (or remain the same).

□

Taking into account Lemma 3 and the fact that Algorithm 1 optimally solves the *MinLatency* problem we reach to the following theorem:

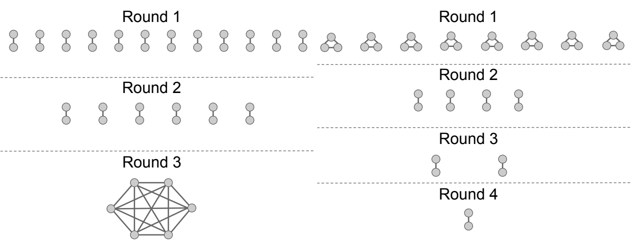
THEOREM 4. *Algorithm 1 optimally solves the* Generalized Worst MinLatency *problem.*

PROOF. Based on Lemma 3, any graph sequence satisfying the constraints (equations (9) and (10)) can have a lower cost if we replace the graphs with tournament graphs. Therefore, the optimal solution to the *Generalized Worst MinLatency* problem consists of a sequence of tournament graphs. Since Algorithm 1 finds the optimal sequence of tournament graphs using dynamic programming, it is also optimal for the *Generalized Worst MinLatency* problem.

□

5. ALTERNATIVE APPROACHES

In this section, we present some simple alternative *budget allocation* and *question selection* algorithms that we will later compare against the combination of tDP with the tournament formation algorithm. In particular, we discuss four heuristics for budget allocation and two *question selection* strategies. The two *question selection* strategies are the tournament-formation algorithm we discussed so far, and the combination of two algorithms proposed in [10]. Besides those two strategies, we also ran experiments (not reported in this paper) with a number of other *question selection* algorithms, proposed in [10], as we briefly discuss in the end of Section 5.2. The two strategies we present here are the ones that worked best in terms of latency and *singleton termination* probability.



(a) HE (b) HF

Figure 10: Examples of HE and HF allocations.

5.1 Budget Allocation Algorithms

We present two simple heuristics along with their uniform versions. In the description of the heuristics (Figures 10(a) and 10(b)) we assume the tournament-formation algorithm is applied in each round, however, the heuristics can also be combined with any other *question selection* algorithm.

Heavy End (HE): The HE heuristic allocates the budget "conservatively" in the first rounds, until it detects that the MAX can be found in a single round; this round becomes the last round and the remaining budget gets allocated there. "Conservative" here means that no question can be "redundant" in the rounds before the last one. Consider the simple example of Figure 10(a) and assume a budget, b, of 51 questions for finding the MAX out of the $c_0 = 24$ initial elements. In the first and second round, HE allocates the budget so that exactly one question is allocated per element. This "conservative" allocation halves the number of candidates in each round; from 24 to 12 and from 12 to 6. In the third round, the remaining budget is enough to form a single tournament with all the remaining candidates. Hence, HE computes allocation $(12, 6, 33)$ in this example; note that in the last round HE allocates all of the remaining budget and not just 15 questions that would be enough for $G_T(6, 1)$.

Heavy Front (HF): The HF heuristic applies the same process with HE, in reverse order. We use the same example to illustrate, depicted in Figure 10(b). HF starts from the last round and assumes that in previous rounds there was exactly one question per element, i.e., the allocation halves the number of candidates in each of the last rounds. At some point, the remaining budget is enough to reach from the initial number of elements to the current number of elements: this point in our example takes place in Round 2 where the remaining budget is enough to reach from the 24 initial elements to the 8 elements in Round 2 through $G_T(24, 8)$. Thus, HF generates allocation $(44, 4, 2, 1)$; note that in the first round we do not just allocate 24 questions, but all of the remaining budget.

uniform Heavy End (uHE): uHE first applies HE to compute the number of rounds that HE would involve. As opposed to HE, uHE will then uniformly distribute the budget into this number of rounds. In the example used in this section, uHE first applies HE to compute the $(12, 6, 33)$ allocation and then divides the budget of 51 questions into three rounds, which results to the allocation: $(17, 17, 17)$.

uniform Heavy Front (uHF): Like uHE, uHF first applies HF to compute the number of rounds and then performs a uniform budget distribution. In the example here, uHF divides the budget of 51 questions into four rounds, which results to the allocation: $(13, 13, 13, 12)$.

It is important to clarify at this point that uHE and uHF combined with the tournament-formation *question selection* algorithm, are two adaptations of the multiprocessor MAX algorithm described in [21] to our setting. As we discuss in detail in the related work section, in settings studied for multiprocessor systems there is no budget constraint. Algorithms uHE and uHF use the same number of questions (which corresponds to the number of processors in a multiprocessor system) in each round, without violating the constraint on the total number of questions that can be asked.

5.2 Question Selection Algorithms

Each *question selection* algorithm uses as input in a given round j:

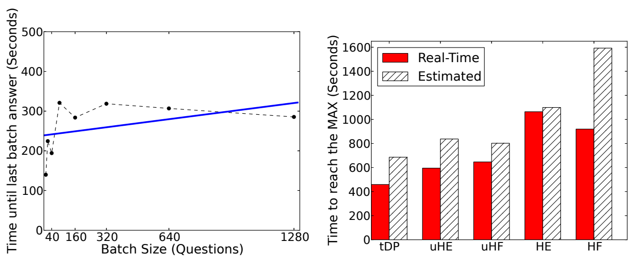
1. b_j: the available budget for round j; as given by the allocation generated by the budget allocation algorithm.
2. $\mathcal{C}_j$: the set of elements that have not lost any comparison in previous rounds, i.e., rounds 0 to $j - 1$.

Tournament-formation: The tournament-formation algorithm is the mechanism we discussed so far for selecting questions in each round. Here we give the general description of the algorithm. In each round the algorithm finds the lowest possible integer c_{j+1}, such that $Q(|\mathcal{C}_j|, c_{j+1}) \leq b_j$. That is, the algorithm tries to form the least number of tournaments possible, such that the questions required to form the tournaments is less than the budget available for this round. (Note that the fewer tournaments formed, the more questions required.) In case $b_j - Q(|\mathcal{C}_j|, c_{j+1}) > 0$, the remaining questions are randomly selected between elements of different tournaments. As stated earlier, the assignment of elements to tournaments is random, hence, the score of each element is not taken into account by the tournament-formation algorithm.

CT25: CT25 combines two *question selection* algorithms: *a)* SPREAD that randomly selects pairs of elements, as long as each element is involved in the same number of questions, and *b)* COMPLETE that spends part of the budget b_j on a single tournament between the "strongest" candidates and the rest of the budget to ask questions between the rest of the elements and the elements of the tournament; so that each element is involved in at least one question. (We describe the two algorithms in detail, in our technical report [25].)

Since COMPLETE needs to distinguish between "strong" and "weak" candidates, we need to rank each element in $\mathcal{C}_j$ based on all the answers from previous rounds. One option is to use the probability of an element being the MAX, to define the element's rank. However, as we prove in Appendix B.1, computing the probability of being the MAX is a #P-hard problem. Hence, we use instead a PageRank-like function that we discuss in detail in Appendix B.2.

CT25 applies SPREAD in the first 25% of all rounds and COMPLETE in the last 75% of all rounds. For example, if an allocation of 4 rounds is used, SPREAD will select the questions in the first round and COMPLETE will select the questions in the last 3 rounds. In experiments that we do not present here, we also tried CT50 and CT75, i.e., applying SPREAD in the first 50% and 75% of the allocation, respectively. In the same experiments, we tried a second *question selection* strategy that combines SPREAD with the GREEDY algorithm proposed in [10]. We give our main observations from those experiments in the experimental evaluation, in Section 6.8.



(a) Estimation of $\mathcal{L}(q)$ (b) Alternatives Comparison

Figure 11: Experiments in MTurk.

6. EXPERIMENTAL EVALUATION

Outline: In Section 6.1, we estimate $\mathcal{L}(q)$ for a specific comparison task and crowdsourcing platform. In Section 6.2, we use the estimated $\mathcal{L}(q)$ as an input to our algorithm and we issue the questions to the crowdsourcing platform to measure the real latency until finding the MAX: our objective is to verify that even when we are not fully aware of the real latency function and we just have a rough estimation for $\mathcal{L}(q)$, our approach shows significant gains compared to the alternatives. In Section 6.3, we compare different *question selection* algorithms and in Sections 6.4 to 6.6 we compare different *budget allocation* algorithms as we vary different parameters of the input. In Section 6.7, we study the running time of tDP.

Setup: We use a collection of 500 car photos. In each pairwise comparison, we present two car photos to a worker in Amazon Mechanical Turk (MTurk), and we ask the following question: "Which of the two cars is the most expensive?". The compensation for each answer is $0.01. All of our experiments were performed during weekdays, in the same 8-hour window every day. We simulate error-free workers by ignoring their answers and replacing them with correct answers, i.e., answers consistent with the true ordering of the 500 cars. As discussed in Section 2, handling human errors requires an RWL that harnesses techniques like question repetition and cycle resolution in the directed graph of answers: these techniques are thoroughly studied in related work [10, 12, 13, 14, 17, 22].

6.1 Estimation of $\mathcal{L}(q)$ in MTurk

In this section, we estimate the latency function, $\mathcal{L}(q)$, when posting car-comparison questions on MTurk. The latency function $\mathcal{L}(q)$ is required by our tDP algorithm to compute the budget allocation. In the next section, we apply tDP using this estimate of $\mathcal{L}(q)$, and we compare it to HE, HF, uHE, and uHF by posting real-time car-comparison questions on MTurk.

To estimate $\mathcal{L}(q)$, we publish batches of different sizes and for each batch size we measure the time from publishing the first question of the batch until receiving the last answer back. For each batch size we publish a batch 20 times and we compute the average over the 20 runs.

In Figure 11(a), the dashed line shows the time until the last answer is returned (y-axis) for the different batch sizes (x-axis). As the dashed line shows, the time needed to receive all answers back is considerably lower for batches up to 40 questions and then increases; for 80 questions or more. Note also the slight decrease in the time needed to receive the answers as the batch size increases from 320 to 1280 questions. The reason for this decrease is that more workers are attracted as the batch size increases in this range, and the increased parallelism compensates for the increased number of questions. Nevertheless, even in cases where the task is very popular to workers (as it is the case with the task used here), there is a point where the batch size becomes much greater than the number of workers interested in the task (we did not reach that point in Figure 11(a)). After that point, there will inevitably be a significant increase in the time to get additional answers back.

The solid line in Figure 11(a) is our estimation for $\mathcal{L}(q)$, fitted using least-squares linear regression. As we discussed earlier, our objective is not to accurately predict $\mathcal{L}(q)$, but to roughly estimate it. Therefore, we just use a linear function of the following form: $\mathcal{L}(q) = \delta + \alpha * q$ seconds. We estimate a δ of 239 and an α of 0.06.

6.2 Real-Time Experiments in MTurk

Using the linear-regression estimation for $\mathcal{L}(q)$, we apply tDP and we decide upon the allocation of $b = 4000$ questions to find the MAX in the collection of 500 cars. We also apply each of the other four allocation algorithms, HF, HE, uHF, uHE, to compute four additional allocations. For each allocation, we post questions in successive rounds on MTurk. The questions are selected by the tournament-formation algorithm. We analyze the answers after the end of each round, and we stop asking questions if just a single element (MAX) not having lost any comparison remains. We repeat the experiment five times for each allocation.

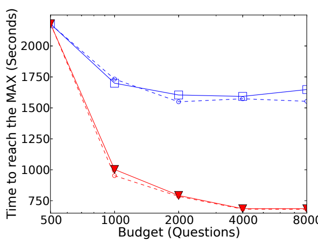
The y-axis of Figure 11(b) depicts the time from posting the first question of the first round in MTurk until reaching at the MAX element. The solid bars depict the time until reaching the MAX for each allocation. We also use stripped bars to plot the time until reaching the MAX based on the estimated $\mathcal{L}(q)$, i.e., for each round instead of actually posting the questions on MTurk, we compute the time it would take to get back all answers, if latency were exactly as predicted by $\mathcal{L}(q)$.

The stripped bars roughly approximate the solid ones. In spite of the coarseness of estimation, tDP is also more effective in the real-time experiments, giving a substantial 30% improvement over the second-best uHE. Furthermore, compared to HE and HF the improvement is more than 2x. Thus, even when the $\mathcal{L}(q)$ function is only a course approximation to the actual latency, it provides enough guidance for tDP to make a very good budget allocation.

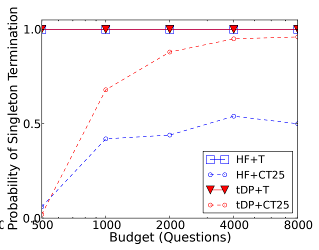
In the rest of the experiments, we use the estimated $\mathcal{L}(q)$ from Section 6.1, to compute the latency in each round. The reason is twofold. First, by using the same deterministic latency function for all approaches we avoid any variations on latency, induced by using a real-world crowdsourcing platform. (As our experiment in this section showed, the estimated latency follows the same trend with the real-time latency.) Second, from now on, in order to accurately measure the latency and *singleton termination* probability for each approach we increase the number of runs per experiment from 5 to 100: this number of runs induces a prohibitive cost for running the experiment on MTurk.

6.3 Comparison of the Question Selection Algorithms

We compare the two question selection strategies, tournament formation and CT25, when we vary the available budget from 500 questions to 8000 questions. In Figure 12(a), we plot the time until reaching the MAX (y-axis) for the different values of available budget (x-axis). Each curve refers

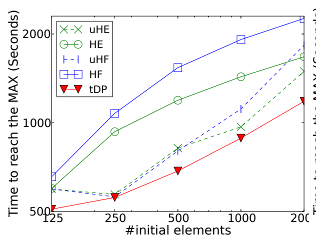


(a) Latency

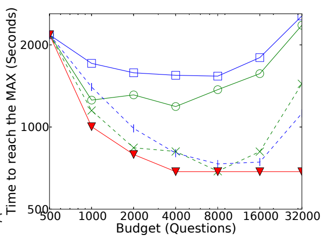


(b) *Singleton termination*

Figure 12: ***Question selection*** **algorithms.**



(a) $b = 4000$ questions



(b) $c_0 = 500$ elements

Figure 13: Latency for a fixed b or c_0.

to a combination of a *budget allocation* with a *question selection* algorithm. To avoid clutter, we plot only four curves here. Nevertheless, our findings (discussed next) also hold for the other combinations of *budget allocation* and *question selection* algorithms. As Figure 12(a) shows, there is a slight improvement in most cases when CT25 is used instead of the tournament algorithm, under both HF and tDP. However, this slight improvement comes with a significant cost as the *singleton termination* plot of Figure 12(b) points out.

In particular, in Figure 12(b) we plot the percentage of runs with a *singleton termination* for each approach (y-axis). The cost of the slight improvement in latency for the approaches under CT25, is the high probability of being left with more than a single element after all questions are asked, especially when the budget is low. In other words, in some runs the approaches under CT25 finish faster compared to the approaches under Tournament-formation, but in other runs the CT25 approaches are left with more than one element. Moreover, our tDP allocation under CT25 gives a much higher probability of *singleton termination* compared to HF + CT25, when the budget is more than 500 questions. On the other hand, the Tournament-formation is more likely to achieve a *singleton termination*: here, *singleton termination* is achieved in every run, for all budget values, under both HF and tDP.

In the experiments that follow we always combine tDP with the Tournament-formation algorithm and all the other *budget allocation* algorithms with CT25. Our goal is to explore whether our approach gives significant gains in latency compared to the alternatives, even if the alternatives have a low probability of *singleton termination.*

6.4 Different sizes for the collection of initial elements

The next direction we explore is to keep the budget, b, of 4000 questions fixed and use different collections of initial elements. Specifically, we use collections of 125, 250, 500, 1000, and 2000 elements. As in Figure 12(a), latency is measured using the estimated $\mathcal{L}(q)$ instead of actually publishing the questions on MTurk and waiting for the answers.

As Figure 13(a) points out, tDP gives a significant improvement over alternatives for certain sizes of the initial-element collection (x-axis). For example, for $c_0 = 2000$ elements, the increase in latency when using the second-best uHE instead of tDP is over 25%, while the increase in latency when using HF is over 90%. Note that for some number of initial elements, the uniform allocation algorithms (uHF and uHE) reach very close to the latency achieved by tDP. The reason is that, coincidentally, the allocations generated by uHF and uHE in those cases are “similar” to the one tDP generates. For instance, for 250 elements, uHF generates allocation $(1000, 1000, 1000, 1000)$, while tDP generates allocation $(884, 465)$. Since, in this example, uHF always reaches to the MAX element without having to initiate the third round, the latency for the tDP and uHF is almost the same. Overall, tDP always achieves the best latency and in cases uHF and uHE are “unlucky”, the improvement over the second-best is quite substantial.

6.5 Different sizes for the available budget

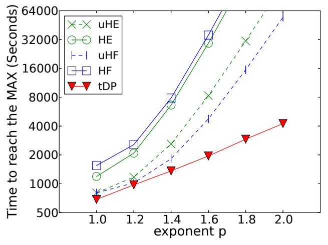
In Figure 13(b), we compare tDP with the other alternatives when we vary the available budget (x-axis) for finding the MAX in a collection of 500 initial elements. tDP constantly improves the overall latency as the budget is increased from 500 to 4000 questions, however, the improvement stops after 4000 questions and the latency remains the same for the increased budget. Surprisingly, this is not the case with all the other allocation algorithms: after a point, latency increases steeply and becomes two to four times the latency of tDP for 32000 questions.

The trend in Figure 13(b) can be explained by the fact that tDP, taking into account the $\mathcal{L}(q)$, chooses after a point (4000 questions in this case) not to spend any more of the available budget. In fact, tDP produces the same allocation, $(2250, 1225)$, for any budget available, after 4000 questions, i.e., tDP only uses 3475 questions from that point on. On the contrary, the other *budget allocation* algorithms, which do not take into account the latency function, always use the whole budget. As Figure 13(b) points out, using more questions can vastly increase latency instead of reducing it.

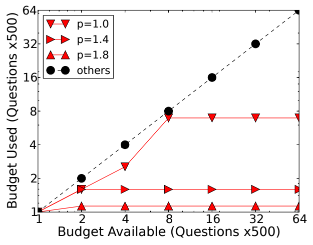
To illustrate this point further, consider the following example: finding the MAX in a collection of 500 elements using a budget of $b = \binom{500}{2} = 124750$ question. Forming the complete tournament $G_T(500, 1)$ in one round is feasible given the budget, however, workers may end up answering questions that are largely redundant and waste a lot of time, instead of focusing on a few remaining comparisons that reveal the MAX element. On the other hand, a strategy where a smaller number of questions is split into multiple rounds can be much more time-effective given the number of workers available and other parameters captured by the latency function $\mathcal{L}(q)$. Hence, our tDP approach doesn’t just provide a near-optimal allocation of the available budget, but it also finds if the available budget should be limited in order to improve the overall latency, taking into account the latency function $\mathcal{L}(q)$. We study further how tDP limits the budget used in the next section, where we run experiments using different latency functions $\mathcal{L}(q)$.

6.6 Experiments using other Latency Functions

In platforms like MTurk, workers find the tasks they are interested in through a browse/search interface. Once the number of questions in the batch becomes greater than the



(a) Latency



(b) Budget used

Figure 14: Comparison over different $\mathcal{L}(q)$ functions.

number of interested workers, the latency increases rapidly; although, as it appears in Figure 11(a), we did not reach that point with our car-images task. In addition, the entries in the browse/search display usually appear in chronological order, i.e., the latest the batch was posted, the higher it appears in the entry list. In such cases, as a large batch starts going down in the entry list over time, the number of interested workers drops; since workers often switch from one task to another even if there are still unanswered questions in the batch. Such factors affect the latency function $\mathcal{L}(q)$, which may become super-linear after a given batch size, q.

In this section, we study how the different approaches perform on a variety of latency functions. In particular, we try a more general $\mathcal{L}(q)$ (with respect to the linear one we used so far) of the form: $\mathcal{L}(q) = \delta + \alpha * q^p$ seconds. We keep a δ of 239 and an α of 0.06, and we vary p.

Figure 14(a) depicts the latency until reaching the MAX for each approach, for the different values of p (x-axis). The budget is 4000 questions for 500 initial elements. As p increases, the gap between tDP and the other approaches increases exponentially. For $p = 2.0$, tDP gives a 12x improvement over the second-best uHF. This improvement is a consequence of tDP limiting the budget used based on the $\mathcal{L}(q)$, as we discussed in Figure 13(b).

To illustrate further the budget-limiting behavior of tDP, we plot the budget used by tDP for different values of p, in Figure 14(b). The x-axis shows the budget available, while the y-axis shows how many of these questions are actually used by tDP. For example, if the available budget is 3000 questions and tDP generates allocation $(1000, 1000)$, then it uses 2000 questions out of the 3000 questions. Each curve refers to applying tDP under a specific p value, except for the "others" curve that refers to all the other allocation algorithms; the value of p does not matter for the other algorithms, since they do not take into account $\mathcal{L}(q)$.

The larger the value of p, the sooner tDP limits the budget used: for $p = 1.0$ the budget is limited at around 3500 questions, for $p = 1.4$ the budget is limited at around 800 questions, and for $p = 1.8$ the budget is limited at around 600 questions. Note also that even before the point where the curves become flat, tDP may decide to use less questions than the available budget. On the other hand, the other allocation algorithms use all of the available budget, until the available budget becomes more than the budget required to form a complete tournament with all of the 500 initial elements in one round, i.e., $\binom{500}{2} = 124750$ questions (not shown in Figure 14(b)). Effectively limiting the budget used, based on $\mathcal{L}(q)$, is the reason why tDP attains the substantial improvement over other alternatives, in Figure 14(a).

6.7 Cost of computing a tDP allocation

Algorithm tDP generates allocations that substantially improve the overall latency compared to heuristics. Nevertheless, there is a cost for computing a tDP allocation. In Figure 15, we measure the running time of tDP on a 1.8 GHz Intel Core i5 processor with 4 GB of RAM. Each curve refers to a different number of initial elements, the y-axis shows the running time in seconds, and the x-axis gives the budget of overall questions; as a multiple of the initial elements. For example, for $x = 8$ on the curve for $c_0 = 1000$ elements, the budget is $b = 8 * 1000 = 8000$ questions and the running time of tDP is around 4 seconds.

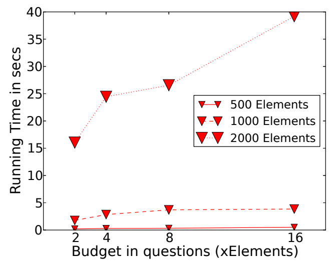


Figure 15: Running time of tDP.

Surprisingly, as Figure 15 shows, the running time for each curve slightly increases as we increase the budget. Since the time complexity of tDP is $O(c_0^2 * b)$, one would expect the running time to double when the budget b is doubled (x-axis). However, our top-down implementation of tDP effectively prunes the set of states (out of the overall $c_0 * b$ states) that we have to visit, as we double b. On the other hand, when we double the number of initial elements c_0 (from one curve to another), the running time does increase by a factor of 4; as one would expect based on the $O(c_0^2 * b)$ time complexity. In that case, tDP needs to examine more states (compared to when we double the budget), to find the optimal sequence to the MAX.

It is worth pointing out here that the running time of tDP is not critical, if we consider the time it takes for humans to answer all these questions. For instance, as we saw in Figure 13(a), it takes more than 1000 seconds to find the MAX for $c_0 = 2000$ elements and a budget, b, of 4000 questions. For such an input of $c_0 = 2000$ elements and $b = 4000$ questions, the running time of tDP is around 15 seconds as Figure 15 shows, which is two orders of magnitude lower than the 1000 seconds to find the MAX. In addition, one should take into account that the car-images task is one of the easiest tasks we could use and the average time to answer a question is around 3 seconds. For more difficult tasks where the number of interested workers decreases, the time to answer each question increases, and $\mathcal{L}(q)$ scales up, the computational cost of tDP starts to become even more negligible. In addition, in many cases, the same budget allocation is used multiple times by crowdsourcing workflows in different runs for the same type of task, with the same budget and number of initial elements. For the corner-cases where the running time of tDP cannot be considered trivial, there are various approaches proposed for implementing parallel dynamic programming algorithms (e.g. [20]) that can achieve a significant speedup.

6.8 Summary of our findings

We conclude the experimental section by summarizing our main findings from the experiments we presented here and other experiments that we did not include in this paper:
(1) Algorithm tDP always achieves the lowest latency. In addition, when combined with the tournament formation

algorithm it always achieves *singleton termination*. Combining tDP with other *question selection* strategies may, in some cases, give a slight improvement in latency, however, it may also cause a significant decrease in the probability of *singleton termination*. Hence, our experimental results are consistent with our theoretical optimality guarantees for our approach in the worst and average case scenarios.
(2) By batching multiple questions in one round, we seek to increase parallelism at the cost of redundancy. The tDP algorithm takes into account $\mathcal{L}(q)$ to estimate this cost of redundancy, and produces allocations that are not wasteful and may consume less than the available budget. Even when $\mathcal{L}(q)$ is just a rough estimation to the actual latency, the gains from effectively allocating and/or restricting the budget can be quite substantial.
(3) The uniform allocation algorithms, uHF and uHE, achieve lower latency compared to HF and HE, for any *question selection* strategy they are combined with.
(4) The uniform *budget allocation* algorithms achieve a higher probability of *singleton termination* compared to HF and HE, for any *question selection* strategy. The only exception is when the budget, b, is near the lower limit, i.e., the number of initial elements, c_0: in this case, HF and HE are usually more likely to achieve *singleton termination*.
(5) The Tournament-formation algorithm achieves a higher probability of *singleton termination* under any *budget allocation* algorithm, compared to the other *question selection* strategies, in almost all of the cases we studied. In some cases, the increased *singleton termination* probability comes with a very small cost in latency.
(6) While non-trivial, the running time of tDP is orders of magnitude lower than the time spent on the crowdsourcing platform due to the human factor.

7. RELATED WORK

The main focus in crowdsourcing research the last years has been on the accuracy and cost dimensions. Nevertheless, part of the study in a number of papers (e.g., [19, 16, 27, 24, 15, 23, 9]) is related to the latency dimension, as well. In paper [19] the problem of finding k elements satisfying a human-decidable property in a larger collection of elements is studied. Latency is defined by the number of rounds (in each round a batch of questions is issued) and the overall objective is to find the skyline of solutions on the latency-cost axes. Paper [16] also considers the latency dimension: questions for finding the target node in a graph (e.g., classifying an object given a hierarchy of classes) are issued in batches and latency is measured by the number of rounds. The number of rounds is also the metric defining latency in crowd entity resolution papers [27, 24].

MAX and sorting algorithms have been studied in the field of parallel computation (e.g., papers [5, 11, 21]): given a number of processors minimize the number of steps (in each step all processors run in parallel) until finding the MAX or sorting all elements. In such settings, all processors are synchronized and take the exact same time to process a comparison. Therefore, the number of steps becomes a very accurate proxy for latency. On the other hand, in a crowdsourcing platform, humans are not "synchronized" and the number of workers can vary in each round. Hence, latency depends on how many questions we ask, in each round. A second important difference is the budget constraint, which is necessary in crowdsourcing settings because of factors like the workers' monetary compensation, but is missing in the parallel computation studies. In particular, in multiprocessor systems, the number of processors is fixed and each processor runs one comparison in each step, without any restriction on the total number of comparisons processed.

Sorting [15], top-k [7], and MAX [10, 23] have also been studied extensively in the crowdsourcing literature. Finding the MAX while taking into account the graph of possibly erroneous human answers is the subject of paper [10]. This work also proposes question selection algorithms that we have used in our experimental evaluation. Closer to the scope of our paper is paper [23], where hill-climb techniques are proposed for finding the MAX in successive rounds. Once more, the metric for latency is the number of rounds. Nevertheless, in real-world crowdsourcing platforms, latency is not always a constant-time function (as implicitly assumed when the metric is the number of rounds), but greatly varies depending on factors like the task, the worker requirements, and the compensation [8]. Hence, our approach does not assume a specific latency function but is applicable for any function while providing strong optimality guarantees.

Another problem related to the one we study here, is the leader election problem in distributed systems [1, 3, 18, 26]. The main difference with our setting, is the lack of a central processing unit: in our case, there is a central unit that keeps track of all the comparisons taking place. On the contrary, in leader election algorithms, each node only knows a subset of the nodes that are still candidates for becoming a leader, based on exchanged messages. Without having a central unit to keep track of all "nodes" (elements) being eliminated, leader election algorithms focus on minimizing the number of messages (or rounds) until the leader is elected and every node is aware of that leader.

8. CONCLUSION

We studied the problem of allocating a budget of questions into rounds in crowdsourcing platforms. We focused on the MAX operator and we derived a polynomial dynamic-programming algorithm that computes the optimal allocation in terms of latency when questions form tournaments in each round. We also compared our approach to any other combination of budget allocation and question selection algorithms, and through our theoretical results we proved optimality in the worst-case scenario and we gave optimality guarantees for the average case scenario under certain assumptions. Our experiments show that our approach is always better than any other alternative we tried, offering important gains in latency, even when we have access only to a rough estimate of a platform's latency behavior on a specific task. Especially, in case the crowdsourcing platform is governed by non-linear (convex/concave) latency functions, the improvement over alternatives can be vast.

In this paper, we focused on the widely studied MAX operator to quantify the gains on latency when the operator is well-understood. The same cost-latency tradeoff arises in other crowdsourcing problems, but the question selection strategy must be adapted to those problems in order to maximize performance gains. Nevertheless, we believe that our dynamic programming approach can be adapted to other scenarios. Furthermore, we believe that having a latency function available, even if coarse, will make it possible to make good budget allocations.

9. ACKNOWLEDGEMENTS

We would like to thank Peter Shor for helping us derive our main results in the average case analysis. We would also like to thank Akash Das Sarma for the fruitful discussions we shared. This research was supported by the National Science Foundation under grant no. 1009916.

10. REFERENCES

[1] Y. Afek and E. Gafni. Time and message bounds for election in synchronous and asynchronous complete networks. In *PODC*, 1985.

[2] M. Ajtai, V. Feldman, A. Hassidim, and J. Nelson. Sorting and selection with imprecise comparisons. In *Automata, Languages and Programming (ICALP)*, 2009.

[3] B. Awerbuch. Optimal distributed algorithms for minimum weight spanning tree, counting, leader election, and related problems. In *STOC*, 1987.

[4] C. Berge. *Graphs and Hypergraphs.* North-Holland, Amsterdam, 1973.

[5] B. Bollobás and A. Thomason. Parallel sorting. *Discrete Applied Mathematics*, 1983.

[6] G. Brightwell and P. Winkler. Counting linear extensions is #p-complete. In *STOC*, 1991.

[7] S. B. Davidson, S. Khanna, T. Milo, and S. Roy. Using the crowd for top-k and group-by queries. In *ICDT*, 2013.

[8] S. Faradani, B. Hartmann, and P. G. Ipeirotis. What's the right price? pricing tasks for finishing on time. In *HCOMP*, 2011.

[9] R. Gomes, P. Welinder, A. Krause, and P. Perona. Crowdclustering. In *NIPS*, 2011.

[10] S. Guo, A. Parameswaran, and H. Garcia-Molina. So who won?: Dynamic max discovery with the crowd. In *SIGMOD*, 2012.

[11] R. Häggkvist and P. Hell. Sorting and merging in rounds. *SIAM J. Algebraic Discrete Methods*, 1982.

[12] P. G. Ipeirotis, F. Provost, and J. Wang. Quality management on amazon mechanical turk. In *HCOMP*, 2010.

[13] D. R. Karger, S. Oh, and D. Shah. Iterative learning for reliable crowdsourcing systems. In *NIPS*, 2011.

[14] X. Liu, M. Lu, B. C. Ooi, Y. Shen, S. Wu, and M. Zhang. Cdas: A crowdsourcing data analytics system. *PVLDB*, 2012.

[15] A. Marcus, E. Wu, D. Karger, S. Madden, and R. Miller. Human-powered sorts and joins. *PVLDB*, 2011.

[16] A. Parameswaran, A. D. Sarma, H. Garcia-Molina, N. Polyzotis, and J. Widom. Human-assisted graph search: It's okay to ask questions. *PVLDB*, 2011.

[17] A. G. Parameswaran, H. Garcia-Molina, H. Park, N. Polyzotis, A. Ramesh, and J. Widom. Crowdscreen: algorithms for filtering data with humans. In *SIGMOD*, 2012.

[18] D. Peleg. Time-optimal leader election in general networks. *J. Parallel and Distributed Computing*, 1990.

[19] A. D. Sarma, A. G. Parameswaran, H. Garcia-Molina, and A. Y. Halevy. Crowd-powered find algorithms. In *ICDE*, 2014.

[20] A. Stivala, P. J. Stuckey, M. G. de la Banda, M. Hermenegildo, and A. Wirth. Lock-free parallel dynamic programming. *J. Parallel and Distributed Computing*, 2010.

[21] L. G. Valiant. Parallelism in comparison problems. *SIAM J. Comput.*, 1975.

[22] P. Venetis and H. Garcia-Molina. Quality control for comparison microtasks. In *Int. Workshop on Crowdsourcing and Data Mining (CrowdKDD)*, 2012.

[23] P. Venetis, H. Garcia-Molina, K. Huang, and N. Polyzotis. Max algorithms in crowdsourcing environments. In *WWW*, 2012.

[24] V. Verroios and H. Garcia-Molina. Entity resolution with crowd errors. In *ICDE*, 2015.

[25] V. Verroios, P. Lofgren, and H. Garcia-Molina. tdp: An optimal-latency budget allocation strategy for crowdsourced maximum operations. *Technical report, Stanford University, available at* ***http://ilpubs.stanford.edu:8090/1129/***.

[26] J. Villadangos, A. Cordoba, F. Farina, and M. Prieto. Efficient leader election in complete networks. In *Parallel, Distributed and Network-Based Processing (PDP)*, 2005.

[27] J. Wang, G. Li, T. Kraska, M. J. Franklin, and J. Feng. Leveraging transitive relations for crowdsourced joins. In *SIGMOD*, 2013.

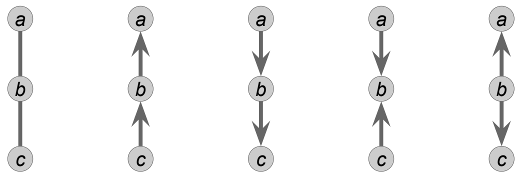
APPENDIX

A. AVERAGE CASE ANALYSIS

In the worst case analysis of Section 4.1, we examined the scenario where after asking a set of questions, the answers we get back leave us with the maximum number of candidate elements remaining. In this section, we examine what happens in the average case, i.e., we focus on the expected number of candidates that will remain, when we ask a set of questions. In addition, to simplify the analysis, we focus on a single round (as opposed to the worst-case analysis), i.e., we study when the expected number of candidates that will remain after a single round, is minimized. The analysis for multiple rounds is beyond the scope of this paper, however, in the end of this section we have a brief discussion related to this topic.

We start with the definition for the expected number of remaining candidates:

DEFINITION 8. ***Expected size of the Remaining Candidates set*** $(\mathbb{E}[\mathcal{R}])$ ***of an undirected graph*** G **:** *The expected size of the RC set, over all possible DAGs that can be formed by giving directions to the edges of* G. *(Note that the probability of each DAG may be different.)*



(a) undirected (b) DAG_1 (c) DAG_2 (d) DAG_3 (e) DAG_4

Figure 16: Example for the expected size of the RC **set of an undirected graph.**

For example, consider the undirected graph in Figure 16(a), involving three elements and two edges-questions. There are four possible DAGs that can be formed, depicted in Figures 16(b) to 16(e). Each of these four DAGs refer to a possible set of answers that we may get back when we ask questions $\{(a, b), (b, c)\}$. Consequently, each of the four DAGs has a probability of being formed: the probability of getting back the answers forming this DAG. Let us assume that each of the $3! = 6$ possible permutations of elements a, b, c have the same likelihood of being the true underlying permutation. Thus, the probability of DAG_1 in Figure 16(b) is $\frac{1}{6}$, since only permutation $\{a > b > c\}$ is consistent with answers $\{(a > b), (b > c)\}$. Likewise, the probability of DAG_2 is also $\frac{1}{6}$, since only permutation $\{c > b > a\}$ is consistent with answers $\{(a < b), (b < c)\}$, while the probability of DAG_3 and DAG_4 is $\frac{2}{6}$ (permutations $\{b > a > c\}$ and $\{b > c > a\}$ are consistent with the answers in DAG_3 and permutations $\{a > c > b\}$ and $\{c > a > b\}$ are consistent with the answers in DAG_4). Moreover, the RC set of DAG_1 is $\{a\}$, the RC set of DAG_2 is $\{c\}$, the RC set of DAG_3 is $\{b\}$, and the RC set of DAG_4 is $\{a, c\}$.

Let us denote by $\mathcal{R}$ the random variable for the size of the RC set. The expected size of the RC set, $\mathbb{E}[\mathcal{R}]$, is $\frac{1}{6} * 1 + \frac{1}{6} * 1 + \frac{2}{6} * 1 + \frac{2}{6} * 2 \approx 1.33$. (Note that the size of the *maxRC* set is 2, based on the RC set of DAG_4.)

In the example of Figures 16(a) to 16(e), we assumed that all 6 possible permutations have the same likelihood. In the results that follow, we assume that for the elements that remain after a number of rounds, all permutations of those elements have the same likelihood of being the true unknown permutation; consistent with the true unknown permutation of the initial elements. We call this state a *uniform history*:

DEFINITION 9. ***uniform history*** **:** *When after a number of rounds, all permutations of the elements that have not lost any comparison yet have the same likelihood, we say that the* history *is* uniform.

To illustrate, consider a toy example of 3 initial elements, a, b, and c. Before the first round all 6 permutations have the same likelihood; no questions are asked yet and we don't assume any prior information. Hence, we have a *uniform history* before the first round. In the first round we ask question (a, b) and we get the answer $a > b$. This answer lets us know that the true underlying permutation is $\{c > a > b\}$, $\{a > c > b\}$, or $\{a > b > c\}$. Thus, permutation $\{c > a\}$ happens only in case of $\{c > a > b\}$ being the true permutation, while permutation $\{a > c\}$ happens in the other two cases. In other words, after round 1 the permutations of the two remaining candidates, a and c, do not have the same likelihood and, hence, the *history* is *non-uniform.*

Note that when questions form tournament graphs in each round and when all the tournament graphs in one round have the same size, a *uniform history* is preserved after any number of rounds. For example, in Figure 9(b), all tournaments in the first round have a size of 2 elements, and all tournaments in the second round have a size of 3 elements. Therefore, there is a *uniform history* after round 1 and after round 2. In the end of the section, where we comment on the multiple-round analysis, we also discuss some of the *non-uniform history* effects.

Next we prove our main theoretical result for the average case analysis: $\mathbb{E}[\mathcal{R}]$ is minimized when G is a tournament graph, under the *uniform history* assumption, as we discuss in Theorem 5.

LEMMA 4. *When the* history *is* uniform, $\mathbb{E}[\mathcal{R}] = \sum_{v \in V} \frac{1}{d_v+1}$, *where $G = (V, E)$ is the graph that the questions form and d_v is the degree of element v in this graph.*

PROOF. Let us denote by S_v the indicator random variable for element v winning all of the d_v comparisons it is involved in. In particular, variable S_v is 1 when v wins all of the comparisons and 0 when v loses one or more comparisons. The expected size of the RC set, $\mathbb{E}[\mathcal{R}]$, is given by the expectation over all S_v, i.e., $\mathbb{E}[\sum_{v \in V} S_v]$. Because of the linearity of expectation, $\mathbb{E}[\sum_{v \in V} S_v] = \sum_{v \in V} \mathbb{E}[S_v]$ (even if S_v variables are not independent). Since we assume a *uniform history*, the ratio of permutations where an element, v, is greater than all of the elements being compared with, to all the possible permutations, is $\frac{1}{d_v+1}$. For instance, if a candidate a is being compared to two other candidates b and c, this ratio will be $\frac{1}{3}$, i.e., in one third of all the possible permutations a is greater than both b and c; given the *uniform history.* Therefore, $\mathbb{E}[S_v] = \frac{1}{d_v+1}$ and $\mathbb{E}[\mathcal{R}] = \sum_{v \in V} \frac{1}{d_v+1}$. □

LEMMA 5. *Given a* uniform history, $\mathbb{E}[\mathcal{R}]$ *is minimized when* $\forall v \in V$, $d_v \in [\lfloor \frac{2|E|}{|V|} \rfloor, \lceil \frac{2|E|}{|V|} \rceil]$, *where $G = (V, E)$ is the graph that the questions form.*

PROOF. In other words, the statement of Lemma 5 is that $\mathbb{E}[\mathcal{R}]$ is minimized when G is a regular graph, i.e., when all elements have the same degree, i.e., $\frac{2|E|}{|V|}$. In case $\frac{2|E|}{|V|}$ is not an integer, not all of the elements can have the same degree. In that case, $\mathbb{E}[\mathcal{R}]$ is minimized when the difference of the maximum degree with the minimum degree in G, is not more than one, i.e., $\lceil \frac{2|E|}{|V|} \rceil - \lfloor \frac{2|E|}{|V|} \rfloor$.

To prove this statement we consider the case where questions form a graph $G = (V, E)$ with at least two elements, a and b, with $d_a < \lfloor \frac{2|E|}{|V|} \rfloor$ and $d_b > \lfloor \frac{2|E|}{|V|} \rfloor$, i.e., the difference between the degrees of a and b is at least two. Now, let us change an edge, between b and a third element c, into a question between a and c. After this change, d_a is increased by 1, d_b is decreased by 1, while d_c remains the same. Hence, $\mathbb{E}[\mathcal{R}]$, which is equal to $\sum_{v \in V} \frac{1}{d_v+1}$ based on Lemma 4, decreases if $\frac{1}{d_a+2} + \frac{1}{d_b} < \frac{1}{d_a+1} + \frac{1}{d_b+1}$. Indeed,

$$\begin{aligned} \frac{1}{d_a+2} + \frac{1}{d_b} < \frac{1}{d_a+1} + \frac{1}{d_b+1} &\Rightarrow \\ \frac{d_a+2+d_b}{(d_a+2)d_b} < \frac{d_a+1+d_b+1}{(d_a+1)(d_b+1)} &\Rightarrow \\ d_a d_b + d_a + d_b + 1 < d_a d_b + 2d_b &\Rightarrow \\ d_a + 1 < d_b & \end{aligned}$$

The last relation is true since we assume $d_a < \lfloor \frac{2|E|}{|V|} \rfloor < d_b$.

What we just proved is that the more equal the element degrees in G are, the lower $\mathbb{E}[\mathcal{R}]$ is. Therefore, $\mathbb{E}[\mathcal{R}]$ is minimized when $\forall v \in V$, $d_v \in [\lfloor \frac{2|E|}{|V|} \rfloor, \lceil \frac{2|E|}{|V|} \rceil]$. □

THEOREM 5. *When the* history *is* uniform, *the tournament graph minimizes* $\mathbb{E}[\mathcal{R}]$.

PROOF. Note that in any tournament graph $G = (V, E)$, the degree of each element is either $\lfloor \frac{2|E|}{|V|} \rfloor$ or $\lceil \frac{2|E|}{|V|} \rceil$ when $\frac{2|E|}{|V|}$ is not an integer, otherwise, the degree is the same for all elements, i.e., equal to $\frac{2|E|}{|V|}$. Therefore, based on Lemma 5, $\mathbb{E}[\mathcal{R}]$ is minimized.

□

In our average-case analysis, we focused on the expected number of remaining candidates in a single round. Over multiple rounds, the analysis becomes much more complex. Let us assume that in each round tournaments of the same size are formed; just like in the example of Figure. 9(b).

Taking into account that *a)* the *uniform history* is preserved after each round (as we noted earlier), *b)* tournament graphs minimize $\mathbb{E}[\mathcal{R}]$ under a *uniform history* (as Theorem 5 states), and *c)* tDP finds the optimal allocation (under tournament graph formation), one may think that tDP under tournament-formation is also optimal over multiple rounds in the average-case scenario. Nevertheless, this is not necessarily the case. The reason is that tDP under tournament-formation is "greedy" in a sense: it tries to eliminate as many candidates as possible in every round (based on Theorem 5). A different approach would be to spend the "first" rounds trying to build a *non-uniform history* that would be exploited in subsequent rounds to "massively" eliminate candidates and reach to the MAX element. In other words, such an "exploration-exploitation" approach would first try to find "strong" and "weak" candidates in the "first" rounds, and then try to reach to the MAX element, in a "few" rounds, by spending most of the questions on "strong" candidates and the rest on "weak" candidates. In the experimental evaluation, we compare such an approach with our tournament-graph approach.

B. RANKING THE ELEMENTS IN EACH ROUND

As we discussed in Section 5.2, the remaining candidates after each round are ranked using a scoring function; the computed scores can be then used by the *question selection* algorithm that picks the questions for the next round. The most straightforward option is to rank each element based on the likelihood of the element for being the MAX, however, computing the MAX likelihood is a #P-hard problem. In this section, we give the #P-hardness proof and we describe the PageRank-like function we used in our experiments, to rank each element.

B.1 Computing the likelihood of being the MAX

The #P-hardness proof for computing the likelihood of an element being the MAX is based on a reduction from the problem of counting the number of linear extensions of a partially ordered set (LE-Count) [6]. Note that #P-hard problems are at least as difficult as NP-hard problems.

Suppose we have n elements, and after a number of rounds we want the probability that element i is the MAX. We represent all previous answers with a DAG $G = (V, E)$, where an edge $(j, k) \in E$ means element k won over j (i.e., we use the same DAG representation we used in our theoretical results).

DEFINITION 10. ***Respecting a set of answers*:** *Given a permutation σ and a DAG $G = (V, E)$, we say σ* respects *E if $\forall(j, k) \in E, \sigma(j) < \sigma(k)$.*

Then we can formalize the problem of computing the probability of an element being the MAX as

$$\text{P-Max}(i, n, E \subset [n] \times [n]) = \Pr[\sigma(i) = n | \sigma \text{ respects } E]$$

where $[n] = \{1, \dots, n\}$ and random variable σ is a permutation chosen from a uniform random prior. Equivalently,

$$\text{P-Max}(i, n, E \subset [n]\times[n]) = \frac{\#\{\sigma : \sigma(i) = n \text{ and } \sigma \text{ respects } E\}}{\#\{\sigma : \sigma \text{ respects } E\}}.$$

Also define

$$\text{LE-Count}(n, E) = \#\{\text{permutations } \sigma : \sigma \text{ respects } E\}$$

We now show that if we could compute P-Max in polynomial time, then we could also compute LE-Count in polynomial time. Since LE-Count is #P-complete [6], this shows that P-Max is #P-hard.

THEOREM 6. *Computing* P-Max *is #P-hard.*

PROOF. Let $(n, E \subset [n]\times[n])$ be a given input to LE-Count, so $G = ([n], E)$ is a DAG. Let τ be any fixed permutation consistent with E. This can be found in polynomial time using the topological sort algorithm. If we could find the probability of τ conditioned on E, we could find the number of linear extensions σ of E, since

$$\#\{\sigma : \sigma \text{ respects } E\} = \frac{1}{\Pr[\sigma = \tau | \sigma \text{ respects } E]}. \qquad (11)$$

When we write probabilities in this section, the only random variable is the permutation σ and it has a uniform prior. Now we can re-write this probability as a product of conditional probabilities. To clarify notation, we first write it for the case $n = 3$:

$$\begin{aligned}&\Pr[\sigma{=}\tau|\sigma \text{ respects } E]{=}\\&\Pr[\sigma(\tau^{-1}(1)){=}1|\sigma(\tau^{-1}(2)){=}2,\sigma(\tau^{-1}(3)){=}3,\sigma \text{ respects } E]\\&\cdot\Pr[\sigma(\tau^{-1}(2)){=}2|\sigma(\tau^{-1}(3)){=}3,\sigma \text{ respects } E]\\&\cdot\Pr[\sigma(\tau^{-1}(3)){=}3|\sigma \text{ respects } E].\end{aligned}$$

This can be seen by simply applying the definition $\Pr[A|B] = \frac{\Pr[A,B]}{\Pr[B]}$ to each probability and canceling common terms.

For general n,

$$\begin{aligned}&\Pr[\sigma{=}\tau|\sigma \text{ respects } E]{=}\\&\prod_{j=1}^{n}\Pr[\sigma(\tau^{-1}(j)){=}j|\forall k{>}j\ (\sigma(\tau^{-1}(k)){=}k),\sigma \text{ respects } E]\end{aligned} \qquad (12)$$

Finally, we note that each probability on the right-hand side of (12) is actually an instance of P-Max on a different graph. Given j,

$$\begin{aligned}&\Pr[\sigma(\tau^{-1}(j)){=}j|\forall k{>}j\ (\sigma(\tau^{-1}(k)){=}k),\sigma \text{ respects } E]{=}\\&\qquad\Pr[\sigma'(\tau^{-1}(j)){=}j|\sigma' \text{ respects } E']\end{aligned}$$

where σ' is a random permutation on $n' = j$ elements and E' is the set of answers in E only involving elements l such that $\tau(l) \le j$. This equality follows from the fact that there is a one-to-one correspondence between permutations σ on n elements respecting E and permutations σ' on n' elements respecting E'. Thus, given an oracle for P-Max, we can compute each of the n terms on the right side of (12). Then applying equation (11), we have computed the given instance of LE-Count. Since LE-Count is #P-complete, we conclude that P-Max is #P-hard.

□

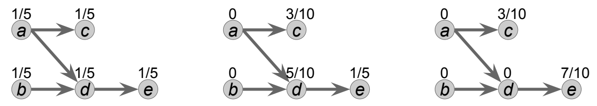
B.2 Scoring Algorithm

From a random-walks perspective, the element scores that our algorithm computes, express the following probability: consider a walker that chooses a starting point uniformly at random and at each element choses one of the outgoing edges uniformly at random. When she reaches an element with no outgoing edges, she is trapped there. The score of each element is the probability that the walker will be trapped at that specific element.

Algorithm 2 Scoring Function

```
Input: $C$: Collection of initial elements
Input: $G$: Directed graph of answers in previous rounds (0 to $j-1$)
Output: $\mathcal{C}_j$: Scores of elements that have not lost any comparisons
 1: $\forall e \in C$: $e[energy] = \frac{1}{|C|}$
 2: while there are elements with non-zero energy and at least one
    outgoing edge in $G$ do
 3:     $e$ = element with the least number of won comparisons (im-
        plicitly or explicitly) that has a non-zero energy.
 4:     $O$ = endpoints of the outgoing edges of $e$.
 5:     $\forall e' \in O$: $e'[energy] = e'[energy] + \frac{e[energy]}{|O|}$
 6:     if $|O| > 0$ then
 7:         $e[energy] = 0$
 8:     end if
 9: end while
10: $\mathcal{C}_j$ = set of elements with non-zero energy, along with their
    energies.
```

To compute the random-walk probabilities, our scoring algorithm transfers the *energy* of the elements that lost at least one comparison in previous rounds, to those that have not lost any comparisons yet. The full description is given by Algorithm 2. Initially, all elements have the same *energy*, $\frac{1}{c_0}$ (line 1). We start from the elements that have not won any comparisons in previous rounds (line 3) and we "pass" their *energy* to the elements that have won a comparison over them (if any). For instance, if an element a has not won any comparison but instead has lost to two elements b and c, a will transfer half of its *energy* to b and half of its *energy* to c. The *energy* of b and c then becomes $\frac{1}{c_0} + \frac{1}{2c_0}$, i.e., we add the received *energy* to the pre-existing *energy* of an element (line 5). (The energy of a becomes 0 after the transfer.) In the next steps (line 2), we transfer the *energy* of the elements that won exactly one comparison, to the elements that won a comparison over them (if any) and so on. At the end of the loop (lines 2 to 9), all the initial *energy* (which sums up to one) will accumulate to the elements that have not lost any comparisons.



(a) Step 0 (b) Step 3 (c) Step 4

Figure 17: Example run of the Scoring Function.

An example run of the algorithm is given in Figures 17(a) to 17(c). Initially, all elements have an *energy* of $\frac{1}{5}$, as depicted in Figure 17(a). In steps 1 and 2, a and b, which have not won any comparisons, transfer their energy through their outgoing edges. The energy of the elements after the first step becomes, 0 for a and b, $\frac{1}{5}+\frac{1}{10} = \frac{3}{10}$ for c, $\frac{1}{5}+\frac{1}{5}+\frac{1}{10} = \frac{5}{10}$ for d, and remains the same ($\frac{1}{5}$) for e (Figure 17(b)). In step 3, only element c has won exactly one comparison, but it does not have any outgoing edges, so all energies remain unchanged. (Note that element e has won over three elements; implicitly or explicitly.) In step 4, element d, which has won two comparisons, transfers its energy to e, which now becomes $\frac{1}{5} + \frac{5}{10} = \frac{7}{10}$ (Figure 17(c)). The loop ends in step 4 since there are no more elements with outgoing edges that have a non-zero energy.