Asynchronous Active Recommendation Systems

(Extended Abstract)

Baruch Awerbuch[1,⋆], Aviv Nisgav[2], and Boaz Patt-Shamir[3,⋆⋆]

[1] Dept. of Computer Science, Johns Hopkins University, Baltimore, MD 21218, USA
baruch@cs.jhu.edu

[2] School of Electrical Engineering, Tel Aviv University, Tel Aviv 69978, Israel
avivns@eng.tau.ac.il

[3] School of Electrical Engineering, Tel Aviv University, Tel Aviv 69978, Israel
boaz@eng.tau.ac.il

Abstract. We consider the following abstraction of recommendation systems. There are players and objects, and each player has an arbitrary binary preference grade ("likes" or "dislikes") for each object. The preferences are unknown at start. A player can find his grade for an object by "probing" it, but each probe incurs cost. The goal of a recommendation algorithm is to find the preferences of the players while minimizing cost. To save on cost, players post the results of their probes on a public "billboard" (writing and reading from the billboard is free). In asynchronous systems, an adversary controls the order in which players probe. Active algorithms get to tell players which objects to probe when they are scheduled. In this paper we present the first low-overhead algorithms that can provably reconstruct the preferences of players under asynchronous scheduling. "Low overhead" means that the probing cost is only a polylogarithmic factor over the best possible cost; and by "provably" we mean that the algorithm works with high probability (over internal coin tosses) for all inputs, assuming that each player gets some minimal number of probing opportunities. We present algorithms in this model for exact and approximate preference reconstruction.

1 Introduction

Recommendation systems are an important ingredient of modern life, where people must make decisions with partial information [7]. Everyday examples include buying books, going to a movie, choosing an on-line store etc. Computer-related examples include, among others, choosing peers in a potentially hostile peer-to-peer environment, or choosing a route in an unreliable network. The basic idea underlying such systems is that users can use the experience reported by others so as to improve their prediction of their own opinions. However, users may differ in their opinions either because they have different "tastes," or because their objectives may be different (e.g., in a peer-to-peer network some users may wish to destroy the system). Obviously, only users

⋆ Partially supported by NSF grants CCF 0515080, ANIR-0240551, and CCR-0311795, and CNS-0617883.

⋆⋆ This research was supported in part by the Israel Science Foundation (grant 664/05) and by Israel Ministry of Science and Technology.

© Springer-Verlag Berlin Heidelberg 2007

whose preferences are similar to those of many others can enjoy the advantages of recommendation systems.

Most recommendation systems in use rely on various heuristics, trying to match a user with others of similar taste [6] or an item with similar items [8]. Recently, recommendation systems were looked at from the algorithmic viewpoint, using the following framework [5,4]. There are n users (also called "players"), and m products (a.k.a. "objects"), and each user has an unknown grade for each product. Each user can find his grade for each product by means of "probing" that product, but each probe incurs a unit cost (probing represents buying a book, or renting a movie etc.). The system provides a public "billboard" on which users post the results of their probes for the benefit of others.

Existing work. There are a few variants in the literature regarding the power of the algorithm and its goal. In this paper, we assume that the role of the algorithm is to tell the player which object to probe whenever the player gets a chance to probe. The goal is to correctly output all (or most) of the player's preferences, even though a player may probe only a negligible fraction of the object space. This problem has been studied before, and some previous solutions exist, each with its own drawback:

- *Committee.* Some algorithms (e.g., in [5,4]) require some players to probe all objects. This solution is problematic in practice both because it's unfair, and because it is vulnerable to malicious users, who may obstruct the selection of the committee, or may gain control over some committee members.
- *Separability assumptions.* Some algorithms (e.g., in [5]) work only if the preferences of the players admit a "low rank approximation," which translates to a severe restriction on the solvable inputs. These restrictions do not appear to apply in many cases.
- *Synchrony.* In [2,1], the algorithm is synchronous in the sense that time proceeds in global rounds. Each player probes once in each round; results from previous rounds are used to determine which objects to probe in the next round. Synchrony is hard to implement in practice, even if we assume that players are willing to follow the protocol: e.g., some players may wish to go much faster than others.

The first two disadvantages were removed recently [2,1].

Our contribution. In this paper, we take the next step by showing how to overcome the latter difficulty: we present an asynchronous algorithm to reconstruct the full preference vectors of players with similar tastes, regardless of the schedule in which they take probing steps. The total probing cost to a set of players with the same taste is larger than the best possible by only a *polylogarithmic* factor, provided that each player in the set makes some minimal number of probes, or that the number of similar players is at least a polylog fraction. We use a randomized scheduling methodology, which may be interesting in its own right.

More precisely, we assume that there is an arbitrary schedule that specifies a sequence of players so that at each step, the next player according to the schedule may take a probing action. The algorithm can control only which object does that player probe, based on the results posted on the billboard so far and random coin tosses. (We assume that the schedule is *oblivious*, i.e., the schedule is fixed ahead of time and may not depend on the outcome of coin tosses.) In this model the number of players that share a given taste is not very important: what counts is the total *volume* of probes done by players of a given taste. Therefore, our primary complexity measure is the total *work* by a given player set, i.e., the volume of probes performed by that set of users in a given schedule.

Clearly, the minimal amount of work that has to be done to find m grades is m probes, because each object must be probed at least once. In this paper we show that after the execution of $\tilde{O}(m)$ probes by the members of a single taste group, the correct vector appears as the output of one of the members, and thereafter it will propagate to all other members of the taste group in $\tilde{O}(n)$ additional probes (once they are given sufficiently many probing opportunities). We note that our algorithms does not terminate: rather, the output value is continuously updated, but once it has reached the correct output the output remains correct.

Our main result for exact type reconstruction is as follows ($\tilde{O}(\cdot)$ and $\tilde{\Omega}(\cdot)$ ignore polylogarithmic factors).

Theorem 1. *Fix a schedule where $\tilde{\Omega}(1)$ fraction of the first $\tilde{O}(m)$ probes are by players with identical preferences. Then with high probability, after $\tilde{O}(m)$ work by these players, their output stabilizes on their true preferences.*

A similar result holds for the approximate case.

Theorem 2. *Fix a schedule where $\tilde{\Omega}(1)$ fraction of the first $\tilde{O}(m)$ probes are by players with preferences at distance $D = \tilde{O}(1)$ from each other for some given D. Then with high probability, after $\tilde{O}(m)$ work by these players, their output is with distance $O(D)$ from their true preferences.*

Our algorithms consider only players performing some minimal work, which is unavoidable. To see that, consider executions where there are k tastes, and each taste group gets a $1/k$ fraction of the total number of probes. Symmetry considerations can be used to prove that $\Omega(k)$ probes are needed for a user just to figure out to which group he belongs (see [3] for a formal argument).

Organization. The remainder of this paper is organized as follows. We formally define the model and some notation in Section 2. In Section 3 we describe and analyze the algorithm for exact preference reconstruction. In Section 4 we present our algorithm for approximate reconstruction of the preference vectors. The analysis of the approximate algorithm is omitted from this extended abstract.

2 Preliminaries

Model. The input consists of n *players* and m *objects*. Each player p has an unknown binary *grade* for each object, and these grades are represented by a vector $v(p) \in \{0,1\}^m$ sometimes called the *preference vector*. An execution proceeds according to a *schedule*, which is an arbitrary infinite sequence of player identifiers. In a single step of an execution, the player selected by the schedule may *probe* a single object, i.e., learn its grade for that object. The identity of the object probed by the player is under the control of the local algorithm run by the player. We assume that the results of all previous probes by all players are available to everyone, and in particular they may be used by players to determine which object to probe next. The algorithm maintains, at each player p, an output vector $g(p)$, which is an estimate of the preference vector $v(p)$. The output vector changes over time. The goal of the algorithms in this work is to use similarity between players in order to minimize the total number of probes they perform. We consider two cases: in the *exact* reconstruction case, the goal is that $g(p)$ will stabilize on $v(p)$ exactly. In the *approximate* case, we are satisfied if from some point on, $g(p)$ differs from $v(p)$ in no more than D grades, for some given parameter D. In the former case, a player can rely only on other players with exactly the same preference vector, while in the latter case, a player may also "collaborate" with players whose preference vectors are close to his own.

Comments. First, for simplicity of presentation, we shall assume in the remainder of this paper that $m = n$. The extension to arbitrary number of objects is straightforward. Second, we note that it may be the case that some players do not report their true results. For our purposes, such players are not considered to be similar to honest players with the same preference vector.

Notation. Given a grade vector v and a set of coordinates (i.e., objects) O, let $v|_O$ denote the projection of v on O, i.e., the vector resulting from v by picking only the coordinates in O. For any two vectors u, v of the same length, we denote by $\mathrm{dist}(v, u)$ the Hamming distance between u and v. For a set O of coordinates, we define $\mathrm{dist}_O(v, u) = \mathrm{dist}(v|_O, u|_O)$, i.e., the number of coordinates in O on which the two vectors differ.

Unless otherwise stated, all logarithms in this paper are to base 2. $\tilde{O}(\cdot)$ and $\tilde{\Omega}(\cdot)$ ignore polylogarithmic factors.

2.1 Building Block: Algorithm SELECT

One of the basic building blocks we use extensively in our algorithms is the operation of selection, defined as follows. The input to a player p consists of a set V of grade vectors for a set of objects O, and a distance bound D. It is assumed that $\min\{\mathrm{dist}|_O(v(p), v) : v \in V\} \le D$. The goal of the algorithm is to find the vector from V which is closest to $v(p)$ on O.

```
SELECT(V, D)
(1) Repeat
      (1a) Let $X(V)$ be set of coordinates on which some two vectors in $V$ differ.
      (1b) Execute Probe on the first coordinate in $X$ that has not been probed yet.
      (1c) Remove from $V$ any vector with more than $D$ disagreements with $v(p)$.
    Until all coordinates in $X(V)$ are probed or $X(V)$ is empty.
(2) Let $Y$ be the set of coordinates probed by $p$ throughout the algorithm. Find the
    set of vectors $U \subseteq V$ closest to $v(p)$ on $Y$, i.e., $U = \{v \in V : \forall u \in V :$
    $\mathrm{dist}_Y(u, v(p)) \ge \mathrm{dist}_Y(v, v(p))\}$ .
(3) Return a randomly selected vector from $U$.
```

The main property of SELECT is summarized in the following lemma.

Lemma 1 ([1]). *If* $\mathrm{dist}(v, v(p)) \le D$ *for some* $v \in V$, *then the output of* SELECT(V, D) *by player* p *is the closest vector to* $v(p)$ *in* V. *The total number of probes executed in* SELECT(V, D) *is less than* $(D + 1)|V|$.

To aid readability, we write SELECT_EXACT(V) for SELECT$(V, 0)$.

2.2 Randomized Multiplexing

In our algorithms, we use a simple methodology for running multiple sequential tasks in parallel by a single player. In this extended abstract we only give an informal overview of the general framework.

The setup is as follows. Without getting into the details of the local computational model, let us assume that there is a well-defined notion of a sequential program, that consists of a sequence of atomic steps. Given the notion of a sequential program, we define the concept of a *task* recursively, as either an infinite-length sequential program, or a tuple $\texttt{rmux}(p_1 : \mathcal{T}_1, \ldots, p_n : \mathcal{T}_n)$, where $n \geq 1$, and for all $1 \leq i \leq n$, $\mathcal{T}_i$ is a task and p_i is a positive real number called the *relative allocation* of $\mathcal{T}_i$. It is required that $\sum_{i=1}^{n} p_i = 1$.

Graphically, a task can be visualized as a rooted tree, where edges are labeled by real numbers between zero and one, and leaves are labeled by sequential programs. For example, consider the task

$$\mathcal{T}_0 = \texttt{rmux}(\tfrac{1}{3}:\mathcal{T}_1, \tfrac{1}{2}:\mathcal{T}_2, \tfrac{1}{6}:\mathcal{T}_3) ,$$

where $\mathcal{T}_1 = \texttt{rmux}(\frac{1}{2} : \mathcal{T}_{11}, \frac{1}{2} : \mathcal{T}_{12})$, and each of the tasks $\mathcal{T}_2, \mathcal{T}_3, \mathcal{T}_{11}$ and $\mathcal{T}_{12}$ is a sequential program. This task is illustrated in Figure 1.

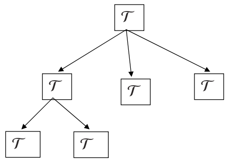

Fig. 1. Example of a multiplexing tree

The semantics of executing a task is like that of a multitasking operating system: each sequential program has its own state, called context. The sequential programs are executed in parallel, one step at a time. A time slot is allocated to a program at leaf ℓ with probability which is the product of the labels on the path leading from the root to ℓ. (In the figure, for example, the allocations of $\mathcal{T}_{11}, \mathcal{T}_{12}$ and $\mathcal{T}_3$ is $\frac{1}{6}$ each.). In each system step, a random leaf is chosen according to its probability, and a single instruction is executed in the context of that leaf's program; as a result, the context is updated (and possibly some global side effects take place). The contexts of all other leaf programs remain unchanged.

Using standard large deviations bounds, we have the following result for a set of players executing the same task asynchronously in parallel.

Theorem 3. *Consider an asynchronous schedule of T steps, and suppose all players execute the same task $\mathcal{T}$. Let $\mathcal{T}_0$ be a sequential program whose probability in $\mathcal{T}$ is p. Then for any $\delta > 0$: if $p \cdot T \geq 3 \log 2/\delta$, then with probability at least $1 - \delta$, the total number of steps of program $\mathcal{T}_0$ done by all players together is $p \cdot T \cdot (1 + o(1))$ as $T \to \infty$.*

Intuitively, Theorem 3 says that if the expected number of steps that $\mathcal{T}$ should get is at least logarithmic, then with high probability the absolute deviation from the expected value is smaller than any positive constant factor.

The `rmux` construct is useful for stabilizing sequential programs, namely programs whose output stops changing after sufficient work was done. Even without explicit indication of stabilization, when programs with acyclic dependencies are run in parallel using `rmux`, their output will stabilize in a bottom-up fashion.

$$\begin{pmatrix} v_1^1 & \cdots & v_{n/2}^2 & & & \\ \vdots & \ddots & \vdots & & ? & \\ v_1^{n/2} & \cdots & v_{n/2}^{n/2} & & & \\ & & & v_{n/2+1}^{n/2+1} & \cdots & v_n^{n/2+1} \\ & ? & & \vdots & \ddots & \vdots \\ & & & v_{n/2+1}^{n} & \cdots & v_n^{n} \end{pmatrix}$$

Fig. 2. Matrix representation of the synchronous algorithm. Rows represent players and columns represent objects. After returning from the recursion, the entries in two quadrants are unknown.

3 Exact Preference Reconstruction

In this section we develop an algorithm for exact types. More formally, let P be any set of players with the same preference vector v_P. Each player p maintains an output vector $g(p)$. The goal of the algorithm is to minimize the total number of probes by players in P until their output vector stabilizes on v_P precisely. We start with some intuition and outline the general structure of the algorithm. In Section 3.1 we specify the algorithm, and in Section 3.2 we analyze it.

The synchronous algorithm. Our starting point is the synchronous algorithm $\mathcal{D}$ from [2]. To gain some intuition, we briefly review the way the synchronous algorithm works. First, it is assumed that a lower bound α on the relative frequency of players with exactly the same taste is given. The algorithm proceeds as follows. Given a set of players and a set of objects, the players and objects are split into two subsets each. Each half of the players recursively determines the values of half of the objects, and then the results are merged. Figure 2 gives a matrix representation of the situation after returning from the recursive call. Merging (i.e., filling in the missing entries) is done by applying SELECT_EXACT to the preference vectors that are sufficiently popular in the other half, where "sufficiently popular" means that the preference vector is supported by, say an $\alpha/2$ fraction of the players. This guarantees correctness, because with high probability, at least an $\alpha/2$ fraction of any large enough random player set are players of the given type (whose global frequency is at least α).

Asynchronous algorithm: basic structure and main ideas. Algorithm $\mathcal{D}$ does not work in the asynchronous case, because it is an adversarial schedule that controls which player gets to probe and when. Hence the number of players of a specific type that execute the algorithm at a given recursion level may be arbitrary, and the crucial popularity threshold becomes meaningless. If we try all possible vectors, the cost to a player increases to the trivial $\Theta(n)$.

Our approach to solve this difficulty can be intuitively described as based on the following ideas. Consider a specific type P. Clearly, the amount of work a single player $p \in P$ needs to do (on average) is inversely proportional to the amount of help he gets from other players of P. In our case, let us first assume that the *density* (i.e., fraction) of probes done by players of P in a given prefix of the execution is some known value

α (provided by an oracle to be implemented later). Note that even when we are given the density of the probes by players of P, we cannot readily apply the synchronous algorithm, because, for example, it may be the case that all these probes are allocated by the schedule to the same player, which, according to $\mathcal{D}$, is supposed to probe only a few objects. We cope with this problem using randomization as follows.

First, as a matter of convenience, we view the recursive partitioning of the object set as a complete binary tree, with the root corresponds to all objects, each of its two children to half of the objects etc. Each leaf corresponds to about $4/\alpha$ objects. In Algorithm $\mathcal{D}$, the players are arranged in a parallel tree, and the execution proceeds from the leafs toward the root level by level, where in each node each player executes SELECT_EXACT.

Here, we use spatial and temporal randomization to overcome the asynchronous schedule. By spatial randomness we mean that when a player gets a chance to probe, he effectively assumes the identity of a random player. This way the work is more-or-less evenly divided over the objects. By temporal randomness we mean that instead of going over the leaf-root path in an order, in our algorithm, after the player have chosen a random identity (and thus a leaf), he chooses a random node along the path from that leaf to the root. This ensures that all nodes will get their "right" amount of work—but not necessarily in the right order. To prove correctness, we show that node outputs stabilize to their correct values inductively, starting from the leaves and ending at the root. We note that this randomization increases the required number of probes by an $O(\log n)$ factor.

Two more ideas are used in the final algorithm. First, we eliminate the assumption that the density α of the probes by players of P is known by running multiple instances of the algorithm in parallel, using the `rmux` construct. Version i works under the assumption that $\alpha \geq 2^{-i}$. The player chooses among the various instances using SELECT_EXACT. Second, consider the case where some players wake up after most many players have already found the correct vector. To avoid duplicating the work, each player continuously looks for a good complete recommendation, by trying all possible output vectors generated by other players. We show that once the correct output appears as $g(p)$ for some player, it spreads quickly to all working players.

Finally, let us address the issue of the number of probes by a player. We note that the number of probes by a player in the synchronous algorithm is $O(\frac{\log n}{\alpha})$. In the asynchronous algorithm, only players doing $\Omega(\frac{\log^3 n}{\alpha})$ probes are useful for the algorithm. Another way to guarantee that sufficient work is useful is to require that the total work done by player of the given type is at least $\Omega(n \log^3 n/\alpha)$.

We present the algorithm in a bottom up fashion: first, we describe an algorithm that assume that the density of the probes by players of P is known, and then give the top-level algorithm.

3.1 The Algorithm

Algorithm for a given α. The objects are recursively divided into a tree structure as described above. Given a node v, $\mathrm{obj}(j)$ denotes the set of objects associated with v. If v is not a leaf, then it has two children denoted $c_1(v)$ and $c_2(v)$. Each node v has a list $G(v)$ of possible grade vectors, posted by the players.

The players work in elementary batches called *jobs*, where each job consists of $4/\alpha$ probes. Jobs are executed within the context of a single tree node. The goal of a job at node v is to append another vector to $G(v)$. The *job algorithm* at a node v is as follows.

```
JOB(v)          // 0 < α ≤ 1 is a given parameter
(1) If $v$ is a leaf, probe all objects in $\mathrm{obj}(v)$ and post the results in $G(v)$.
(2) If $v$ is an internal node:
    (2a) Read the list $G(c_1(v))$ and let $B_1$ be the set of the $2/\alpha$ most popular vectors in
         it (break ties arbitrarily). Similarly, construct a set $B_2$ of the $2/\alpha$ most popular
         vectors in $G(c_2(v))$.
    (2b) $g \leftarrow$ concatenation of SELECT_EXACT($B_1$) and SELECT_EXACT($B_2$); append
         $g$ to $G(v)$.
    (2c) If $v$ is the root, $g(p) \leftarrow$ SELECT_EXACT($\{g(p), g\}$).
```

Note that since the schedule is asynchronous, SELECT_EXACT in Step 2b is not done atomically (Step 2c consists of a single probe). Asynchrony has no effect on the output if v is a leaf (Step 1), because the set of objects probed in this case is always $\mathrm{obj}(v)$. But if v is an internal node, the situation is different: while the probing of Step 2b is carried out, the contents of the lists $G(c_1(v))$ and $G(c_2(v))$ may change. In our algorithm these lists are read once, at the beginning of the job (Step 2a) resulting in lists B_1 and B_2 whose contents is then frozen throughout the remainder of the execution of the job.

The algorithm for a given α is simply "execute jobs at random:"

```
Algorithm BASIC($\alpha$)
 Repeat forever: pick a random node $v$ and execute JOB($v$).
```

Note that Algorithm BASIC(α) is a non-terminating sequential program. Its output is the vector $g(p)$ (written by a root job), which changes over time.

Algorithm for Unknown α. We now explain how to execute the algorithm without knowledge of α. Let P be a set of players with the same preference vector v_P. As we show later, BASIC(α) guarantees that for some player $p_0 \in P$, we will eventually have $g(p_0) = v_P$, provided that α is the relative density of probes by players of P. Thus, we need to solve two problems: how to choose the right value of α, and how to disseminate v_P to the other players of P, once v_P is discovered. Our approach is to solve both problems using the `rmux` construct. Choosing the right value of α is done by trying all $\log n$ powers of $1/2$ in parallel as possible values of α. To disseminate v_P, each player p repeatedly compares his own output $g(p)$ (which is common to all his instances of PULL and BASIC) with the output of a randomly chosen player. Formally, we define the following simple task:

```
Algorithm PULL($V, D$)
 Repeat forever:
    pick a random vector $v \in V$ and execute $g(p) \leftarrow$ SELECT($\{g(p), v\}$, $D$).
```

The main algorithm can now be specified as follows. Let $\alpha_{\min}$ be the smallest value of α for which the algorithm is designed, and define $I = \log(1/\alpha_{\min})$ (note that $I = O(\log n)$ if the schedule length is polynomial in n). We run I tasks in parallel, and the

PULL task. We note that the set of vectors sent to PULL changes over time: if $g(p')$ is chosen in PULL, its *current* value is copied over and sent to SELECT_EXACT.

```
Main Algorithm for Exact Reconstruction

  rmux(
    1/2:  rmux ($\frac{1}{I}$:BASIC($\frac{1}{2}$), $\frac{1}{I}$:BASIC($\frac{1}{4}$), ..., $\frac{1}{I}$:BASIC($\frac{1}{2^I}$)),
    1/2:  PULL($\{g(p') \mid p'$ is a player$\}$, 0)  // the set sent to PULL is dynamic
  )
```

3.2 Analysis

We now analyze the main algorithm. Fix a specific set of players P with identical preference vector v_P. Below, we first do some straightforward accounting, and then analyze the instance of the basic algorithm that runs with the "correct" α value. We show that after $\tilde{O}(n)$ work in that instance, at least one player in P holds v_P in its output vector. We show that after this point, only $\tilde{O}(m)$ more work in total is needed until that output reaches all other players in P.

We start with some necessary notation. Fix an arbitrary schedule $\mathcal{S}$. Define $T_P = 32In\log^2 n = \tilde{O}(n)$, and let $\mathcal{S}_0$ be the shortest prefix of $\mathcal{S}$ that contains T_P probes by players in P. We denote $T_0 = |\mathcal{S}_0|$. Let $\alpha_0 = T_P/T_0$. We assume that $\alpha_0 \geq 2^{-I}$. Finally, define $T_1 = T_0/2I$. As immediate corollaries of Theorem 3, we have the following.

Lemma 2. *With high probability, each instance of the basic algorithm gets $T_1(1+o(1))$ probes in $\mathcal{S}_0$, of which an $\alpha_0(1+o(1))$ fraction are executed by players in P.*

Lemma 3. *Consider an instance $A =$ BASIC(α) run by the main algorithm. The number of probes executed at a given node of A in $\mathcal{S}_0$ is, with high probability, $\frac{2T_1}{\alpha n}(1+o(1))$, of which an $\alpha_0(1+o(1))$ fraction are done by players of P.*

Define $\alpha_1 = \frac{4nI\log n}{T_P}\alpha_0 = \frac{\alpha_0}{8\log n}$, and let $i_0 = \lceil \log 1/\alpha_1 \rceil$. Henceforth, we focus on the specific task executing BASIC(2^{-i_0}). Let us call this task A_{i_0}. We will use the following concept.

- A leaf ℓ is said to be *done at time t* if at least $\log n$ jobs were completed by players of P in ℓ in the time interval $(0, t]$.
- An internal node v at height $h > 0$ is said to be *done at time t* if there exists some time $t' < t$, such that both children of v are done by time t', and at least $\log n$ jobs of v were fully executed by players of P in the time interval $(t', t]$.

Note that *done* is a stable predicate: once a node is done, it is considered done throughout the remainder of the prefix of $\mathcal{S}$ containing T_0 probes. The significance of the notion is made apparent in the following key lemma.

Lemma 4. *Consider the execution of A_{i_0}. Suppose a node v is done at time t_0. Then at all times $t_0 \leq t \leq T_0$, at least $\alpha_1/2$ fraction of the vectors in $G(v)$ are the correct grade vectors of players in P for* $\mathrm{obj}(v)$.

Proof: By induction on the height of v. For the base case, suppose that v is a leaf. By Lemma 3, the total number of probes at v at any given time is no more than $\frac{2T_1}{\alpha_1 n}(1+o(1))$, and since each job contains $\frac{4}{\alpha_1}$ probes, the total number of jobs in a leaf v (and hence the size of $G(v)$) is at most $\frac{T_1}{2n}(1+o(1))$. Therefore, once the total number of jobs by players of P in v exceeds $\frac{\alpha_1 T_1}{2n} = \frac{4nI\log n}{T_0} \cdot \frac{T_0}{2I} \cdot \frac{1}{2n} = \log n$, the number of vectors in $G(v)$ that are correct for P is at least an $\alpha_1/2$ fraction, as required.

For the induction step, assume that the lemma holds for height $h-1$ and consider a node v at height h. Let u_1 and u_2 be the children of v. By definition and the induction hypothesis, we have that starting at the time t' when both u_1 and u_2 were done, all jobs at v had the correct vectors of them among their B_1 and B_2 lists (Step 2a of the job algorithm). By the correctness of SELECT_EXACT, each of these jobs will write the correct output in $G(v)$. Next, by Lemma 3, we have that the total number of probes in v is again at most $\frac{2T_1}{\alpha_1 n}(1+o(1))$, which means that $|G(v)| \le \frac{T_1}{2n}(1+o(1))$, because each job at v consists of $4/\alpha_1$ probes. As in the base case, once at least $\frac{\alpha_1 T_1}{2n} = \log n$ jobs are completed at node v after its children are done, the correct vector for the players of P will be in the most popular vectors of v. ∎

The proof of Lemma 4 hints at the main argument of the theorem: we need to show that the nodes in the computation tree become gradually done. The remaining difficulty lies in the asynchrony: Lemma 3 talks about the total number of probes by players in P throughout the execution, while Lemma 4 talks about jobs, and at specific times. However, there is a logarithmic factor between α_0 used in Lemma 3 and α_1 used in Lemma 4; as we show next, this additional freedom, together with a guarantee on the minimum work done by each player, suffice to prove the result.

Lemma 5. *If each player in P executes at least $\frac{256I\log^2 n}{\alpha_0}$ probes, then with high probability, each player in P executes at least $2\log n$ jobs in instance A_{i_0}.*

Proof: The size of each jobs is $\frac{4}{\alpha_1} = \frac{32\log n}{\alpha_0}$. The expected number of probes by each player in P is at least $\frac{128\log^2 n}{\alpha_0}$ in instance A_{i_0}. By Chernoff inequality the probability each player probes less than half of the expectation is at most $\exp(-\Omega(\frac{\log^2 n}{\alpha_0}))$. Since there are at most n players each of them probes, with high probability, at least $\frac{64\log^2 n}{\alpha_0}$ times, i.e., $2\log n$ jobs. ∎

Lemma 6. *If the number of probes in A_{i_0} by players in P is $\alpha_0 T_1 \ge 16n\log^2 n$, then at time T_0 the root node is done w.h.p.*

Proof: Consider only work done by players of P during the execution of A_{i_0}. Define time intervals inductively as follows: t_0 is the start of the schedule. Suppose that t_h is defined, for $0 \le h < H$, where $H = \log\frac{\alpha n}{4}$ is the tree height. Define t_{h+1} to be the first time in which we have $\frac{\alpha_1 n\log n}{2^h}$ completed jobs that were executed at nodes at height h and started after t_h. Let us call these jobs "effective jobs."

To prove the lemma, it suffices to show that we can define these time points up to t_{H+1}, and that $t_{H+1} \le T_0$: this implies that at time $t_{H+1} \le T_0$, the root node is done. First, note that a job that starts in the interval $(t_h, t_{h+1}]$ may finish its execution outside that interval. However, for any given $h \ge 0$, a player may start at most one job in $[t_h, t_{h+1}]$ that doesn't finish in that interval. Since by Lemma 5, each player executes at least $2 \log n$ jobs, and since $H \le \log n$, at least half of the jobs are fully executed within one time interval.

We now prove that $t_{H+1} \le T_0$. Consider jobs executed within one time interval. In the first interval, half of the jobs are in leaves, so w.h.p., t_1 occurs before $2\alpha_1 n \log n(1+o(1))$ such jobs are executed. Since the number of effective jobs is halved from one time interval to the next, and since the number of nodes in height h is half the number of nodes in height $h-1$, in each interval there are at most $2\alpha_1 n \log n(1+o(1))$ jobs, and t_{H+1} is before $2\alpha_1 H n \log n(1+o(1)) \le 2\alpha_1 n \log^2 n$ jobs. Since by time T_0 $16n \log^2 n$ probes are executed by players in P, the number of jobs executed within single interval is at least $16n \log^2 n \frac{\alpha_1}{4} \cdot \frac{1}{2} \ge 2\alpha_1 n \log^2 n$, and it follows that $t_{H+1} \le T_0$.

For any h, the $\frac{\alpha_1 n \log n}{2^h}$ effective jobs are distributed over $\frac{\alpha_1 n}{2^{h+2}}$ nodes. By Chernoff bound each node is associated with at least $\log n$ effective jobs with probability at least $1 - \frac{1}{n^{9/8}}$. As there are $\frac{\alpha_1 n}{2} = O(n/\log n)$ nodes then by time t_{H+1} all nodes in the computational tree are done with high probability. ∎

By Lemma 2, the schedule for A_{i_0} consists of $\frac{16n \log^2 n}{\alpha_0}$ probes, of which an α_0 fraction are by players in P. Therefore, by Lemmas 6 and 4, by time T_0, the root node of A_{i_0} is done, and at least one player in P has $g(p) = v_p$. Next, we show that the PULL task allows players in P learn this vector in "epidemic" style. The key is that once a player $p \in P$ tests v_P in PULL, that player will eventually assign $g(p) \leftarrow v_P$, and furthermore, p will never change his $g(p)$ value ever again, because it will never find it to be inconsistent with his preferences. As more players assign v_P to their output vector, the probability of a player to choose it in PULL increases.

For the next lemma, call a probe by player $p \in P$ *non-stabilized* if $g(p) \ne v_P$ at the time of the probe.

Lemma 7. *Suppose that the root node of A_{i_0} is done at some point. Then after $O(n \log n)$ additional non-stabilized probes, we have $g(p) = v_P$ for all $p \in P$.*

Proof Sketch: Once A_{i_0} is done, at least one player $p_0 \in P$ executed JOB(root) when c_1(root) and c_2(root) are already done, and after that job is done we have $g(p_0) = v_P$. Let σ_l, for $l > 0$, be a random variable whose value is the number of non-stabilized probes done starting at the time that the lth player in P assigns v_P as its output, and ending at the time that the $(l+1)$st such player assigned v_P as its output.

Consider σ_l: With probability $1/2$, the algorithm chooses a random player and examines its opinion. As there are l players holding v_P as their output, the probability of each probe to produce the right vector is $\frac{l}{2n}$, and hence the expected value of σ_l is $2n/l$. It follows that after expected $\sum_{l=1}^{|P|-1} \frac{2n}{l} < 2n \ln |P|$ non-stabilized probes, all $|P|$ players will have v_P as their output. Using standard arguments, it can also be shown that $O(n \log n)$ such probes are also sufficient w.h.p. ∎

We can now conclude with the following theorem, which combines the previous results to show the full picture: if enough probes are done by players with the same preference vector, their output stabilizes on their true preference vector.

Theorem 4. *Let $\mathcal{S}$ be a schedule such that at least α fraction of the probes are by players with exactly the same preference vector v_P running the algorithm for exact reconstruction. Then with high probability, after total work of $(1+\frac{8}{\alpha})33nI\log^2 n$, the output of all these players has stabilized on the correct value, where $I \geq \log(1/\alpha)$.*

Proof: Let P be the players with the same preference vector. Assume there are at least $(1+\frac{8}{\alpha})33nI\log^2 n$ probes by players in P in $\mathcal{S}$.

Consider the prefix T of the schedule in which $\frac{32+264/\alpha}{33+264/\alpha}\alpha|\mathcal{S}| > (32+\frac{264}{\alpha})nI\log^2 n$ probes are by players in P. The fraction of probes by players in P in this prefix is $\alpha' > \frac{32}{33}\alpha$. The number of players in P that don't make at least $\frac{256I\log^2 n}{\alpha'} < \frac{264I\log^2 n}{\alpha}$ probes is at most $|P| \leq n$. Therefore, at least $32nI\log^2 n$ probes are made by players who probe at least $\frac{256\log^3 n}{\alpha'}$ times each. By Lemmas 6 and 4, the root node at A_{i_0} is done and at least one player in P has its preference vector as its opinion at the end of the prefix T. By Lemma 7, after at most $n\log n$ additional non-stabilized probes, all players in P learn their preference vector. ∎

Theorem 1 is an immediate corollary of Theorem 4.

4 Approximate Preference Reconstruction

In Section 3 we presented an algorithm for reconstructing the preference vectors without any error (w.h.p.). One drawback of that algorithm was that collaboration took place only among players with identical preferences. In many cases, however, the number of players that share the exact same preferences may be small. In this section we extend on the results of Section 3 and present an asynchronous algorithm that allows players to use recommendations of any player whose preference vector differs from their own in no more than D objects, for some given parameter D. The output of the algorithm, at each player, may contain $O(D)$ errors. The total work done by the players of the similar preferences in our algorithm is $\tilde{O}\left(nD^{5/2}\right)$. The analysis of the algorithm is omitted from this extended abstract. For simplicity, we assume $D = O(\log n)$ here.

4.1 Algorithm

The asynchronous algorithm is based on the synchronous algorithm SMALL presented in [1]. The algorithm works as follows. Let P be set of players and $D \geq 0$ be such that $\mathrm{dist}(v(p), v(p')) \leq D$ for any $p, p' \in P$. As in the exact reconstruction case, we first assume that the algorithm is given the density parameter α. Conceptually, the algorithm consists of three phases. In the first phase, the object set O is partitioned into $s = O(D^{3/2})$ random parts denoted $\{O_j\}_{j=1}^{s}$, and Algorithm BASIC(α) is run by all players on each O_j. Algorithm BASIC is guaranteed to succeed only if there are sufficiently many probes by players whose preferences on the objects of O_j are identical.

Fortunately, it can be shown that with probability at least $\frac{1}{2}$, a random partition of O will have, in each part, "many" players in P fully agreeing. Therefore, if K independent random partitions of O are used, then one of them will succeed in all parts with probability at least $1 - 2^{-K}$.

Typically, a player in P shares his exact preferences with sufficiently many other players in P and have correct output for $\text{BASIC}(\alpha)$ in some of the O_j parts, but in other parts, his result of $\text{BASIC}(\alpha)$ is unpredictable. To remedy this problem, in the second phase of the Algorithm, players adopt as their output, for each object part O_j, the closest of the most popular output vectors of Algorithm $\text{BASIC}(\alpha)$. In the full paper we show that concatenating these s partial vectors by a player in P results in a preference vector that contains no more than $5D$ errors.

Due to asynchrony, it may be the case that only a single player in P arrives at the correct vector in the second phase. The third phase of the algorithm disseminates this vector to other players using the PULL mechanism. This phase may introduce D more errors, so that the final output may contain up to $6D$ errors.

Asynchrony also implies that sequential execution of the phases cannot be guaranteed. We solve this problem by running all sequential programs in parallel, using the `rmux` construct. This ensures that at the price of polylogarithmic blowup in the number of probes, once the output of one phase has stabilized, it can be used as input to the next phase.

Finally, the assumption of a given parameter α is lifted by running a logarithmic number of possible α values in parallel. See the complete algorithm below.

Let $s = 100D^{3/2}$. The algorithm uses $K = O(\log n)$ random partitions of O: for $1 \le k \le K$, the kth partition is $O = O_1^k \cup O_2^k \cup \ldots \cup O_s^k$. The top level algorithm below uses the task $\text{PASTE}_{i,k}$, which pastes together all s components of the output corresponding to $\alpha = 2^{-i}$ in the kth partition.

```
Algorithm APPROX($D$)

rmux(
    1/3: rmux($\frac{1}{K \cdot s \cdot I}$ : BASIC($2^{-i}$) on $O_j^k$, for randomly chosen $1 \le i \le I$,
              $1 \le j \le s,\ 1 \le k \le K$) ,
    1/3: rmux($\frac{1}{K \cdot I}$ : PASTE$_{i,k}$ for randomly chosen $1 \le i \le I,\ 1 \le k \le K$) ,
    1/3: rmux(PULL($\{g(p') \mid p'$ is a player$\}$ , $6D$)
    )
```

Let $A_j^{i,k}$ be the k execution of the basic algorithm for $\alpha = 2^{-i}$ on the object set O_j^k, let the output of each such execution at given time be the 2^{i+1} most popular vectors in the root. Each player p maintains an output $g(p)$ which changes over time. Note this $g(p)$ isn't updated by the task JOB(root) as in the algorithm for exact reconstruction.

Operator PASTE **:** Algorithm PASTE, specified below, gets as input a partition index k, an α value 2^{-i} (repersented by i), and continuously updates the player's output vector for this case.

```
PASTE$_{i,k}$
Repeat forever:
(1) For each $j \in \{1, \ldots, s\}$:
    Let $V$ be the $2^{i+1}$ most popular vectors in $G(\text{root})$ of $A_j^{i,k}$
    $u_j^{i,k} \leftarrow$ SELECT$(V, D)$
(2) Let $u$ be the concatenation of $u_j^{i,k}$ over all $j$.
    Execute PULL$(\{u, g(p)\}, 5D)$.
```

As the output of $A_j^{i,k}$ might change over time, it is read once at the start of the procedure and frozen throughout the execution. Note that each execution of this operator takes $O(sD2^{i+1})$ probes.

The performance of the algorithm is summarized in the theorem below (proof omitted). Theorem 2 is an immediate corollary of Theorem 5 below.

Theorem 5. *Let P be a set of players such that $\text{dist}(p, p') \leq D$ for any $p, p' \in P$. Let $\mathcal{S}$ be a schedule with a prefix of length $T = \Omega(\frac{nD^{5/2} I \log^3 n}{\alpha^2})$, of which at least fraction α of the probes are by players of P executing Algorithm* APPROX. *By the end of the prefix, with high probability, there exists a player $p_0 \in P$ with $\text{dist}(g(p_0), v(p_0)) \leq 5D$. In $O(Dn \log n)$ additional work by players in P with $\text{dist}(g(p), v(p)) > 6D$, all these players will have, with high probability, $\text{dist}(g(p), v(p)) \leq 6D$.*

References

1. Alon, N., Awerbuch, B., Azar, Y., Patt-Shamir, B.: Tell me who I am: an interactive recommendation system. In: Proc. 18th Ann. ACM Symp. on Parallelism in Algorithms and Architectures, pp. 1–10. ACM Press, New York (2006)
2. Awerbuch, B., Azar, Y., Lotker, Z., Patt-Shamir, B., Tuttle, M.: Collaborate with strangers to find own preferences. In: SPAA 2005. Proc. 17th ACM Symp. on Parallelism in Algorithms and Architectures, pp. 263–269. ACM Press, New York (2005)
3. Awerbuch, B., Patt-Shamir, B., Peleg, D., Tuttle, M.: Adaptive collaboration in synchronous p2p systems. In: ICDCS 2005. Proc. 25th International Conf. on Distributed Computing Systems, pp. 71–80 (2005)
4. Awerbuch, B., Patt-Shamir, B., Peleg, D., Tuttle, M.: Improved recommendation systems. In: Proc. 16th Ann. ACM-SIAM Symp. on Discrete Algorithms, pp. 1174–1183. ACM Press, New York (2005)
5. Drineas, P., Kerenidis, I., Raghavan, P.: Competitive recommendation systems. In: STOC 2002. Proc. 34th ACM Symp. on Theory of Computing, pp. 82–90. ACM Press, New York (2002)
6. Resnick, P., Iacovou, N., Suchak, M., Bergstrom, P., Riedl, J.: Grouplens: an open architecture for collaborative filtering of netnews. In: CSCW 1994. Proc. 1994 ACM Conf. on Computer Supported Cooperative Work, pp. 175–186. ACM Press, New York (1994)
7. Resnick, P., Varian, H.R.: Recommender systems. Commun. ACM 40(3), 56–58 (1997)
8. Sarwar, B., Karypis, G., Konstan, J., Reidl, J.: Item-based collaborative filtering recommendation algorithms. In: Proc. 10th International Conf. on World Wide Web (WWW), pp. 285–295 (2001)