p0892808
A "Go With The Winners" Approach to Finding Frequent Patterns

[Extended Abstract]

ABSTRACT

In their seminal work on Go With the Winners algorithms, D. Aldous and U. Vazirani [4] proved a sufficient condition for the number of particles needed for reaching the bottom of a tree with high probability via a GWW random walk; namely that the number should be proportional to a measure of the tree's imbalance times a polynomial $p(d)$, where d is the tree depth. However, to use this result in practice would require knowledge of the entire search tree which is infeasible for most problems. In this paper we improve slightly on this situation by deriving a recurrence relation that provides an upper-bound for a tree's imbalance in terms of the imbalance between tree levels that are close to one another, provided that these latter imbalances can be measured with sufficient accuracy.

We then turn our attention to the problem of finding both frequent and infrequent patterns in a database, and prove new NP and #P-completeness results for both problems. Furthermore, one of the most widely used algorithms for finding frequent patterns in memory-resident databases is a randomized algorithm first proposed by Gunopulos et al. [16]. We show that such an algorithm is precisely one for which the GWW paradigm was designed to improve on. Experimental results using the Splice-junction Gene Sequences Database [5] are also provided and lend empirical evidence of the benefits of using GWW.

1. INTRODUCTION

Many optimization problems can be alternatively cast as search problems through a particular state space. If one is exceptionally fortunate, the state space may possess convenient mathematical properties, and allow for an efficient search over the most likely areas of the state space that contain an optimal solution. However, tractable spaces are rarely encountered in either theoretical investigations or industrial optimization problems. For this reason, various search heuristics have been developed for searching through state spaces of potentially unknown structure (see [10, 21, 22, 23] for a well-rounded overview of the field). Moreover, recently there has been increasing interest in adaptive search algorithms that attempt to learn and estimate the structure of a state space for the purposes of both adjusting the needed resources and locating the appropriate areas of the state space for which the search should be conducted. Some algorithms of interest include the multi-start GRASP algorithms [13], and the algorithms of Boese et al. [6] and Boyan and Moore [9], the latter of which attempts to statistically learn more advantageous search trajectories based on past trajectories.

Aldous and Vazirani [4] were among the first to gain theoretical insight into the benefits of adaptive multi-start search algorithms over their nonadaptive counterparts. They assumed the state-space to be a tree, optimal solutions as the deepest leaves of the tree, and a search trajectory as a particle that randomly moves down the tree, starting at the root. Thus a non-adaptive multi-start search would consist of using a number of independent particle trajectories, and is referred to as *Algorithm 0*. To model adaptive multi-start search, they used a "Go With the Winners" (GWW) approach, in which a number of particles randomly move down the tree, and coordinate in such a way that any particle p_1 which reaches a leaf is immediately placed on a randomly selected internal node that is both at the same level as p_1 and currently hosts some other particle p_2; hence the name "Go With the Winners." They next proved a remarkable result, which states that with constant probability, a GWW state-space search will find an optimal (deepest) leaf provided at least $\kappa \cdot poly(d)$ particles are used, where $poly(d)$ is a polynomial in the depth d of the tree, and κ is a constant which measures the tree's imbalance. At the same time, trees with small κ values were provided for which Algorithm 0 required an exponential number of restarts before reaching the optimal leaf with high probability. Dimitriou and Impagliazzo [12] then showed how GWW could be applied to more general state spaces by decomposing them into trees, and showed how a variation of GWW performed well on trees with an appropriate expansion property between successive levels. This was in turn successfully applied to the problem of finding graph bisections [11]. Recently however, Peinado [19] has shown that for constant ϵ, with high probability a superpolynomial number of particles are needed for a GWW algorithm to find a clique of size $(1+\epsilon)\log n$ in a random graph with n vertices.

In this paper, our first contribution to the study of GWW algorithms involves the issue of overcoming a practical limi-

SAC '05 New Mexico USA

tation inherent in Aldous and Vazirani's seminal result. Indeed, to use their result in practice would require knowledge of the imbalance of the entire search tree which is infeasible for most problems. We improve on this situation by deriving a recurrence relation that provides an upper-bound for a tree's κ (imbalance) value in terms of the imbalance between tree levels that are close to one another, provided that these latter imbalances can be measured with sufficient accuracy.

We then turn our attention to the problem of finding both frequent and infrequent patterns in a database, and prove new NP and #P-completeness results for both problems. Finding frequent patterns represents a central problem in the field of Data Mining. (See [2, 1, 20, 7, 16, 17] for a history of the problem.) Furthermore, one of the most widely used algorithms for finding frequent patterns in memory-resident databases, is a randomized algorithm first proposed by Gunopulos et al. [16]. We show that this algorithm is precisely a version of Algorithm 0 with respect to a search tree whose leaves are frequent database patterns of maximum support. Hence this is precisely the type of algorithm for which the GWW paradigm was intended to improve on. Experimental results using the Splice-junction Gene Sequences Database [5] are also provided, and lend evidence of a GWW improvement over Algorithm 0.

The remainder of the paper is organized as follows. In Section 2, we present the original GWW algorithm along with our new recurrence relation. In Section 3, we apply the new method to the problem of searching a database for a maximal pattern and present several novel complexity results. Also, we show that our method improves upon that of Gunopulos et al. by outlining how to construct a database for which their method takes superpolynomial time and the adaptive GWW does not. In Section 4, we state some experimental results and mention some open problems and possible areas of future research.

2. DESCRIPTION OF THE ALGORITHM

In this section, we will describe the GWW algorithm originally presented by D. Aldous and U. Vazirani [4]. We then advance the theory by deriving a recurrence relation that provides an upper bound for a tree's imbalance in terms of the imbalance found between levels of the tree that are close together.

2.1 GWW

In this section, we will present the original GWW algorithm from [4] including the notation and definitions therein. The goal of GWW is the following: given a rooted tree of maximum depth d, locate a node at depth d.

We assume that for every vertex v in the tree, there is a probability distribution P_v associated with its children.

Definition 1. Algorithm 0: Given a particle that starts at the root, repeatedly choose a child randomly according to the distribution P_v until the particle arrives at a leaf. Repeat this procedure using independent particles a total of B times.

GWW: Repeat the following procedure, starting at stage 0 with B particles at the root: At stage i, each of the B particles is at some vertex at depth i. If all the particles are at leaves, then stop. Otherwise, some particles $j_1, j_2, \ldots, j_m$ are at non-leaves. Spread the remaining $B - m$ particles evenly among the positions of the m particles. If $B - m$ is not a multiple of m, pick a random subset of the m positions each of which gets assigned an extra particle. Then let each of the B particles move from its current vertex v to a vertex chosen according to the distribution P_v.

We also use some additional definitions that were first introduced by Aldous and Vazirani [4].

Definition 2. Let $p(v)$ be the probability that an Algorithm 0 particle makes it to vertex v. Let $p(v|w)$ be the probability that an Algorithm 0 particle started at the vertex w makes it down to the vertex v. (Then $p(v) = p(v|root)$.) If V_i represents the set of vertices on level i, define $a(i) = \sum_{v_i \in V_i} p(v_i)$, the probability that an Algorithm 0 particle makes it down to level i. Define $a(i|w) = \sum_{v_i \in V_i} p(v_i|w)$, the probability that an Algorithm 0 particle started at the vertex w makes it down to level i.

Finally, we define the parameters that govern the GWW process defined above.

Definition 3. Define

$$\beta = \min_{0 \le i < d} \frac{a(i+1)}{a(i)}$$

and

$$\kappa = \max_{0 \le i < j \le d} \kappa_{i,j}$$

where

$$\kappa_{i,j} = \frac{a(i)}{a^2(j)} \sum_{v_i \in V_i} p(v_i) a^2(j|v_i)$$

An alternative definition for $\kappa_{i,j}$ is as follows. Let W_i be the position[1] of an Algorithm 0 particle at depth i, conditioned on getting to depth i. Then $\kappa_{i,j} = \frac{Ea^2(j|W_i)}{(Ea(j|W_i))^2}$.

The important result of [4] is as follows.

Theorem 1. [4] The chance that GWW does not find a node at level d in the tree is $O(\frac{\kappa d^4}{B}(1 + \frac{1}{\beta d}))$ where all parameters are as in Definitions 1, 2, and 3. If $\beta = \Omega(\frac{1}{d})$ and $B = O(\kappa d^4)$, then the GWW algorithm succeeds with constant probability in time $O(\kappa d^6)$.

2.2 The Recursive Formula

In Section 2.1, we reviewed some of the theoretical concepts behind GWW. In this section, we will state and prove a recursive formula for the various $\kappa_{i,j}$ values. As a corollary of the recursive formula, we can algebraically prove a fact that one might intuitively suspect is true: in some sense, with high probability, the GWW algorithm requires not more than a polynomial number of particles more than Algorithm 0 requires iterations to achieve the same results.

The search for a recursive formula for the $\kappa_{i,j}$ values was undertaken for two primary reasons. Firstly, the obvious relations that one might hope to be true, perhaps $\kappa_{i,j} \le \kappa_{i,k}\kappa_{k,j}$ or $\kappa_{i,j} \ge \kappa_{i,k}\kappa_{k,j}$ can both be violated. Figure 1 illustrates counterexamples to both conjectures. The relation

[1] Throughout this paper, we will use the uppercase W_i to denote a random variable representing the position of a level i particle; whereas a lowercase w_i will stand for a particular position (or an instantiation of the random variable W_i).

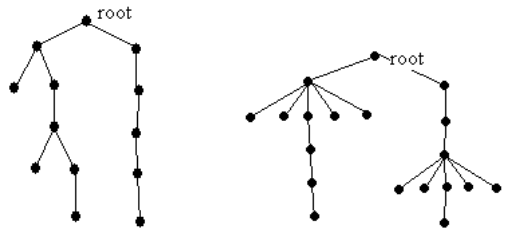

Figure 1: Two counterexamples. Assume that it is equally likely for a particle to choose any child of a vertex when it is time to move. Refer to the root as Level 0. Note that for the left tree, $\frac{34}{25} = \kappa_{1,5} > \kappa_{1,3}\kappa_{3,5} = \frac{10}{9} \times \frac{27}{25} = \frac{6}{5}$. For the right tree, $1 = \kappa_{1,5} < \kappa_{1,3}\kappa_{3,5} = \frac{13}{9} \times \frac{9}{5} = \frac{13}{5}$.

we get, Theorem 2, is not completely intuitive (to the authors, at least), making it somewhat interesting as a mathematical artifact. Another interesting fact about Theorem 2 is that it happens to be tight in the sense that equality is achieved if $k = i$. The second reason for searching for a possible recursive formula for the $\kappa_{i,j}$ was the hope that a method for upper or lower bounding the number of particles necessary to reach lower levels in the tree might be gleaned from the $\kappa_{i,j}$ values already discovered. In theory, GWW seems to be a powerful tool for finding the deepest node in a rooted tree, assuming κ is relatively small. However, in practice, GWW functions blindly. Either B particles are necessary to get to the deepest leaf or they aren't; unless you are aware of the values of κ and d in advance there is no way of knowing whether your search is truly at an end or whether more particles are necessary[2]. Our relation is sometimes able to improve the situation in cases where $\kappa_{i,j}$ estimates are known in advance for small $|i-j|$.

Throughout this section, we will make use of the notation and definitions in Section 2.1. Additionally, for the rest of the paper, we will use the following notation: for any two nodes v and w in the tree, $v \geq w$ means that v is a descendent of w in the tree. If we use a subscript on a node, it will refer to its level in the tree (e.g. w_i will refer to a vertex on level i).

Lemma 1. If $0 \leq i \leq k \leq j \leq d$,

$$Ea(j|W_i) = Ea(k|W_i)Ea(j|W_k)$$

Proof. Note that $Ea(j|W_i) = \frac{a(j)}{a(i)}$. □

Theorem 2. For any given tree,

$$\kappa_{i,j} \leq \min_{i \leq k \leq j} \frac{1}{Ea(k|W_i)} \kappa_{k,j}$$

[2]In Section 4, we get around this problem by setting a time limit on a single run of GWW and successively doubling the number of particles we use in each GWW trial until the time limit expires. In practice, however, it would be nice to know whether it would be worth it to extend the time limit "just a little while longer" if it means better results from the computation.

Proof. From the definition of $a(j|w_i)$, for any $0 \leq i \leq k \leq j \leq d$ and for any w_i,

$$a(j|w_i) = \sum_{w_k \geq w_i} p(w_k|w_i)a(j|w_k)$$

Squaring both sides and applying Jensen's Inequality, we get

$$a^2(j|w_i) \leq \sum_{w_k \geq w_i} p(w_k|w_i)a^2(j|w_k)$$

Taking expected values of both sides treating the position of w_i as the random variable,

$$Ea^2(j|W_i) = \sum_{w_i} \frac{p(w_i)}{a(i)} a^2(j|w_i)$$

$$\leq \sum_{w_i} \frac{p(w_i)}{a(i)} \sum_{w_k \geq w_i} p(w_k|w_i)a^2(j|w_k)$$

$$= \frac{a(k)}{a(i)} \sum_{w_i} \sum_{w_k \geq w_i} \frac{p(w_i)p(w_k|w_i)}{a(k)} a^2(j|w_k)$$

$$= \frac{a(k)}{a(i)} \sum_{w_k} \frac{p(w_k)}{a(k)} a^2(j|w_k)$$

$$= Ea(k|W_i)Ea^2(j|W_k)$$

Dividing both sides of this inequality by the square of the equation in Lemma 1, we get $\kappa_{i,j} \leq \frac{1}{Ea(k|W_i)}\kappa_{k,j}$. Because k was arbitrary, we get $\kappa_{i,j} \leq \min_{i \leq k \leq j} \frac{1}{Ea(k|W_i)}\kappa_{k,j}$. □

Corollary 1.

$$\kappa_{i,j} \leq \frac{a(i)}{a(j)} \text{ and } \kappa \leq \frac{1}{a(d)}$$

Note that by Theorem 1, the number of particles required by the GWW algorithm is only $O(\kappa d^4)$. The number of iterations necessary to get the same results from Algorithm 0 is proportional to $\frac{1}{a(d)}$, implying that with high probability, GWW cannot be outperformed by Algorithm 0 by more than an asymptotic polynomial factor in d.

Proof. Letting $k = j$ in the minimum on the right hand side of Theorem 2, we get $\kappa_{i,j} \leq \frac{1}{Ea(j|W_i)}\kappa_{j,j} = \frac{1}{Ea(j|W_i)} = \frac{a(i)}{a(j)}$. From this equation, we can take the maximum over i and j from both sides to get $\kappa = \max_{0 \leq i < j \leq d} \kappa_{i,j} \leq \max_{0 \leq i < j \leq d} \frac{a(i)}{a(j)} = \frac{1}{a(d)}$. □

3. DATABASE SEARCH

In [16], Gunopulos et al. restated the problem of finding maximal frequent patterns in databases as a search for the deepest node in a tree. In this section, we review this basic concept, prove several completeness results, and improve upon their algorithm.

3.1 The Tree Structure

We consider our database D to be made up of a series of records, all of which contain an equal number of characteristics. Each characteristic will be assigned a letter from a finite alphabet Σ. We define a database D of n characteristics to be an $m \times n$ matrix with entries from Σ. The

rows $r_1, \ldots, r_m$ of D represent the database records. It is appropriate to think of r_i as a vector whose ith component r_{ij} is the jth entry (jth characteristic) of the ith row (ith record) of D.

Let $\mathcal{R}_n$ denote the set of all n-dimensional vectors which may be defined over $\Sigma \cup \{*\}$ where $* \notin \Sigma$. We refer to such a vector in $\mathcal{R}_n$ as a *pattern*. Define the partial order $(\mathcal{R}_n, \leq)$, where $r_1 \leq r_2$ iff, for every $1 \leq j \leq n$, either $r_{1j} = r_{2j}$ or $r_{1j} = *$. In that case, we say that r_2 *contains* r_1. Intuitively, the $*$ symbol reflects a "wildcard" character. For example, $(1, *, 0) \leq (1, 0, 0)$. It is also instructive to think of the ordering in terms of quantity of information in the pattern: there is more information in the record $(1, 0, 0)$ than $(1, *, 0)$ and the two records are clearly comparable.

Given an $m \times n$ database D and positive integer $t \leq m$, we say that an n-dimensional vector p is a *t-frequent pattern* of D iff at least t rows of D contain p. (Throughout this paper, we will drop the t- from t-frequent when the parameter t is unambiguous.) Moreover, if there is no frequent pattern q that properly contains p, then p is said to be a *maximal* frequent pattern. Finally, the *support* of a frequent pattern p, $supp(p)$, will denote the number of non-starred components of p.

Given a threshold value t, we form a poset structure from the database D as follows. The vector $(*, *, \ldots, *)$ is at the root. Clearly, this is always a frequent pattern, and it has support 0. For all vectors x and y, y is a parent of x iff $y \leq x$ and $supp(y) = supp(x) - 1$. This structure is equivalent to the Hasse diagram of the poset induced by the partial order $(\mathcal{R}_n, \leq)$. Notice that the leaves of this poset correspond precisely to the maximal frequent patterns of D. (For an illustration of the poset structure for a given database, see Figure 3.)

To form a tree structure from this poset, for any given pattern p, we identify not only the characters and positions that we have chosen but also the order in which they have been chosen. In other words, the poset element $(1, *, 0, 1)$ will be split into 6 tree elements:

$$(1_1, *, 0_2, 1_3), (1_1, *, 0_3, 1_2), (1_2, *, 0_1, 1_3),$$

$$(1_2, *, 0_3, 1_2), (1_3, *, 0_1, 1_2), (1_3, *, 0_2, 1_1)$$

The first of these $(1_1, *, 0_2, 1_3)$ indicates that the first component was chosen before the third component which was chosen before the fourth component. For all "tree" patterns x and y, y is a parent of x iff y is a parent of x in the poset structure and all non-$*$ components in y are chosen in the same order as those in x. Again, the leaves of this poset correspond to the maximal frequent patterns of D (plus some additional information). The number of nodes in the tree is vastly more than the number of nodes in the corresponding poset, but the "balance" properties are preserved.

Thus, we arrive at the decision problem we consider in this section: Given an $m \times n$ database D with characteristics chosen from a finite alphabet Σ, a positive threshold value $t \leq m$, and a positive integer K, does there exist a t-frequent pattern in D with support greater than or equal to K? Or, equivalently, does there exists a node of depth K in the tree corresponding to the database D with threshold value t?

3.2 Computational Complexity

In this section we turn our attention to the problem of finding both frequent and infrequent patterns in a database, and prove new NP and #P-completeness results for both problems.

Definition 4. Given two patterns, x and y, we define their *intersection* $x \cap y$ to be the vector that contains (a) a $*$ in any position where x and y contain different characteristics and (b) the characteristic of x (or y) in any position where x and y contain the same characteristic. This definition generalizes trivially to include intersection over a set of patterns $\bigcap_{x \in S} x$.

Definition 5. [DBS] Database Search: Given an $m \times n$ database D with characteristics chosen from a finite alphabet Σ, a nonnegative threshold value $t \leq m$, and a nonnegative integer $K \leq n$, does there exist a t-frequent pattern in D with support greater than or equal to K? Or, equivalently, does there exist a set of database records S such that $|S| \geq t$ and $\bigcap_{x \in S} x$ has support greater than or equal to K?

In [15], Gunopulos et al. prove several assertions similar to the ones made below. The only difference between our assertion and theirs is the difference in the definition of a "frequent" pattern. They consider a frequent pattern in a 0-1 database to be only those frequent patterns with support consisting of all 1's. Our definition is slightly more general in that we also allow 0's to appear in a frequent pattern[3]. We have altered their proof to take into account our expanded definition.

Theorem 3. [DBS] is NP-complete.

Proof. [DBS] is trivially in NP. We can assume without loss of generality that $\Sigma = \{0, 1\}$. We reduce from the balanced complete bipartite subgraph problem, which is known to be NP-complete, [GT24] in [14]. The balanced complete bipartite subgraph problem is as follows: Given a bipartite graph $G = (V, E)$ and a positive integer $K \leq |V|$, are there two disjoint subsets $V_1, V_2 \subseteq V$ such that $|V_1| = |V_2| = K$ and such that $u \in V_1, v \in V_2$ implies that $\{u, v\} \in E$?

Let $G = (V, E)$ be a bipartite graph and let K be a positive integer such that $K \leq |V|$. Consider each row of the adjacency matrix of G as a record in a database D. Modify D by adding $|V| + 1$ more rows to this database that consist of all 1's (i.e. $(1, 1, \ldots, 1)$). Call this new database D'.

We claim that there exists a $|V| + K + 1$-frequent pattern in the database D' with support greater than or equal to K if and only if there exists a complete bipartite graph of the required shape in G. To see this, note that if p is a frequent pattern for D', then no component of p is zero. Hence the support of p has all ones and implies the existence of K rows of D with the same K columns containing the value one. □

In addition Gunopulos et al. [16] have shown, using their definitions, that counting the number of maximal t-frequent patterns is #P-complete. We can make the corresponding claim about our expanded definition using the same database expansion technique as in Theorem 3.

Definition 6. [#DBS] Database Count: Given an $m \times n$ database D with characteristics chosen from a finite alphabet Σ, a nonnegative threshold value $t \leq m$, and a nonnegative integer $K \leq n$, how many t-frequent patterns in

[3]To put it a different way, Gunopulos et al. assume that it is only important when a given characteristic is frequently "on"; whereas our definition extends this notion to take into account the fact that it may also be important when a given characteristic is frequently "off" as well.

D with support greater than or equal to K are there? Or, equivalently, how many distinct sets S of database records are there such that $|S| \geq t$ and $\bigcap_{x \in S} x$ has support greater than or equal to K?

Theorem 4. [#DBS] is #P-complete.

In addition, the database search problem as formulated above in Definition 5 can be easily transformed into other natural problems other than the somewhat theoretical 0/1 sentence extraction as mentioned in [15]. For example, given a list of places and associated visiting times for various people, it is trivial to form this into a database[4] where maximum frequent patterns correspond to "popular places" at specific times. To phrase this example another way, assume that a robot moves on a bounded grid one unit either north, south, east, or west per time step. Given a large database of paths that the robot has previously taken, determine his likely positions at various times. (Both of these formulations seem to be militarily useful as both aim to predict future movements of unknown and somewhat unpredictable entities.)

Also, in passing, we note that in a similar manner one can define what it means to be a t-infrequent pattern of D. Generally speaking, it is a difficult problem to find many records in a database with many characteristics in common; hence, [DBS] is a difficult problem. Similarly, it is also difficult to find a small collection of records with very little in common. Boros et al. [7, 8] have previously defined this concept and presented some results about computing such patterns. Gunopulos et al. [16] have also defined a t-infrequent pattern problem. We prefer to phrase the concept in terms of the intersection operator.

Definition 7. A t-infrequent pattern in a database D is a pattern that can be obtained by intersecting at most t records of the database.

Definition 8. [DBS2] Database Count #2: Given an $m \times n$ database D with characteristics chosen from a finite alphabet Σ, a nonnegative threshold value $t \leq m$, and a nonnegative integer $K \leq n$, does there exist a t-infrequent pattern in D with support less than or equal to K?

Theorem 5. [DBS2] is NP-complete.

Proof. [DBS2] is trivially in NP. We can assume without loss of generality that $\Sigma = \{0, 1\}$. We reduce from the dominating set problem, which is known to be NP-complete, [GT2] in [14]. The dominating set problem is as follows: Given a graph $G = (V, E)$ and a positive integer $K \leq |V|$, is there a dominating set of size K or less for G, i.e., a subset $V' \subseteq V$ with $|V'| \leq K$ such that for all $u \in V - V'$ there is a $v \in V'$ for which $\{u, v\} \in E$?

Let G' be the graph that comes from adding a free vertex with no edges to the graph G. In other words, let $G' = (V \cup \{w\}, E)$ where $w \notin V$. Let the adjacency matrix for G' be M. Alter M so that all diagonal elements are 1 (i.e. $\forall i, M_{ii} = 1$). Form the database D from M by letting each row of M be a vector in D.

We now claim that there exists a dominating set of size K or less for G if and only if there exists a $(K+1)$-infrequent pattern in D with support equal to 0. To show this, assume that there exists a dominating set in G of size K or less. By the definition of dominating set, if we intersect each of the rows corresponding to the vertices in the dominating set of G together with the row corresponding to the special vertex w, the intersection must be $(*, *, \ldots, *)$. Conversely, assume that there exists a set of $K+1$ rows in D that intersect to give $(*, *, \ldots, *)$. Note that the row corresponding to w must be included in this intersection; otherwise, the column corresponding to w will contain a zero. For any given characteristic, there must be some row that contains a one in that characteristic; otherwise, upon intersection with w, we get zero instead of $*$. □

Note that the correspondence between dominating sets and infrequent sets we set up in the proof of Theorem 5 is a bijection. Every distinct dominating set of size K or less in G corresponds precisely to a single set of $K+1$ rows (or less) in D with intersection having support zero. Because the counting version of the dominating set problem is #P-complete, we trivially obtain the following.

Definition 9. [#DBS2] Database Count #2: Given an $m \times n$ database D with characteristics chosen from a finite alphabet Σ, a nonnegative threshold value $t \leq m$, and a nonnegative integer $K \leq n$, how many t-infrequent patterns are there in D with support less than or equal to K?

Theorem 6. [#DBS2] is #P-complete.

So we have the interesting state of affairs that though it is trivial to solve the decision and counting problems corresponding to finding frequent patterns with small support and infrequent patterns with large support, both of the "dual" problems are thought to be extremely difficult.

It is also interesting to note that Boros et al. [7] have shown that the collection of t-infrequent sets with support less than or equal to K can be enumerated in quasi-polynomial time (i.e. in time $O(n^{f(n)})$ where $f(n)$ is bounded by a polylogarithm in n, if n is the input size).

3.3 Description of the Large Frequent Pattern-Finding Algorithm

We will start this section by defining the Algorithm 0 distribution of database search particles. The search structure that we use (outlined above in Section 3.1) was first defined by Gunopulos et al. [16], and has the desirable property that every leaf is a maximal frequent pattern. To determine a particle trajectory down the tree, the following randomized procedure is used. We assume that we are given an $m \times n$ database with characteristics chosen from a finite alphabet Σ, a positive threshold value $t \leq m$, and a positive integer K.

- Initialize t-frequent pattern p_0 to be the pattern consisting of all $*$'s .
- Stage i, $i \geq 0$: Assume that p_i has been defined. Compute the set $\mathcal{P}$ of all patterns that are t-frequent, contain p_i, and have support exactly one greater than p_i. If $\mathcal{P} = \emptyset$, then stop: p_i is a maximal element. Otherwise, let p_{i+1} be a pattern that is selected uniformly at random[5] from $\mathcal{P}$. Proceed to Stage $i+1$.

[4] For a given day, simply let the ith component of the associated database vector be the character representing the place where the person was at time i.

[5] In practice, of course, one wouldn't compute this entire set

The above process induces a probability distribution over the set of maximal frequent patterns, which we henceforth refer to as its Algorithm 0 distribution.

We now define the GWW distribution for the same problem.

- Initialize a multiset consisting of B copies of the t-frequent pattern p_0 to be the pattern consisting of all $*$'s. Call this S_0.
- Stage i, $i \geq 0$: Assume that the multiset S_i has been defined. Compute the set $\mathcal{P}$ of all patterns that are t-frequent, contain at least one p_i from S_i, and have support exactly one greater. If $\mathcal{P} = \emptyset$, then stop: all patterns in the multiset S_i are at maximal elements. If $\mathcal{P} \neq \emptyset$, we do the following. For each $p \in S_i$, if there does not exist an element in $\mathcal{P}$ that properly contains p, replace p with an element $p' \in S_i$ selected uniformly at random that is properly contained within some element of $\mathcal{P}$. Select B patterns to include in the multiset S_{i+1} in the following way. For each element $p \in S_i$, select an element uniformly at random from $\mathcal{P}$ that properly contains p. Proceed to Stage $i+1$.

Of course, an interesting question to ask is: what should the value of B be? This and other issues will be discussed in Section 4.

In practice, the Algorithm 0 distribution is usually sufficient for an acceptable search of the space. Either the search tree is fairly small (indicating a large threshold value t) with few leaves or large and somewhat balanced (indicating a small threshold value for t in which most patterns are considered frequent). In the former case, only a small number of particles from Algorithm 0 will be necessary to sample from the leaves of the tree because Algorithm 0 hits a leaf at every iteration. In the latter case, almost every iteration of Algorithm 0 is expected to hit a leaf near the bottom of the tree because so few of the tree paths are short. If one is given an intermediate value of t, however, and a sufficiently complex database, the search tree could get skewed enough in some cases to warrant using GWW.

4. SIMULATION/CONSLUSIONS

Curiously, though we have shown that the GWW algorithm cannot be outperformed by the corresponding Algorithm 0 introduced by Gunopulos et al. [16] by more than a small margin, we are unable to produce a database, neither through explicit construction nor via a theoretical existence proof, for which GWW is provably better than Algorithm 0. Though it seems intuitive that there is such a database given the large number of trees with small κ values, we are forced to leave open the question of whether or not such a database exists. Oddly enough, this is the reverse of the usual situation. In many instances, an algorithm is presented that performs best on special types of databases that rarely appear in practice. In this case, our algorithm seems to perform well in practice, but quantifying *how well* in comparison with other commonly-used algorithms seems difficult.

In their experiments, Gunopulos et al. [16] used Algorithm 0 to generate maximal frequent patterns. However, GWW is oftentimes better than Algorithm 0 at finding the deepest leaf of a tree; and with respect to finding frequent patterns, this corresponds to being better at finding the maximal frequent patterns with largest support. In this section, we give a real-world example of such a database.

In some cases, an asymptotic result can yield an algorithm that functions perfectly well in theory and yet does not perform well in practice. A case to consider is the famous AKS sorting network [3], a theoretical network that is able to sort n numbers in the sorting network model in depth $O(\log n)$. The constant hidden by the asymptotic notation in [3] is easily more than the number of particles in the known universe. For this reason, we wrote a program in C/C++ to simulate the random[6] action of particle walks using both Algorithm 0 and GWW. After a lengthy Internet search, we chose a database from the UCI Machine Learning Repository [5] to be our test case. Specifically, we used the "Splice-junction Gene Sequences Database" donated by Geoffrey Towell, Noordewier, and Shavlik. This database consists of a series of 3190 rows of sixty characters (along with some other information) corresponding to the commonly-known base pairs from molecular biology: A, G, T, and C. We considered each row to be a vector in the database and each character within a given row, a separate characteristic of the vector. Thus, we tested the GWW adaptation on a database of 3190 vectors with 60 characteristics. This is a relatively small database compared to some of those regularly used in industry, but just large enough to illustrate the importance of our GWW adaptation.

The first issue we address is the question mentioned in Section 3.3. In practice, how does one determine the optimal value of B, the number of "particles" to use in the GWW algorithm? We suggest the method of successive doubling. Start out the GWW algorithm with $B = 1$. Once the GWW algorithm completes, note the lowest depth node, double B, and start again. We declare the algorithm finished when one run of the GWW algorithm exceeds a certain preset time limit, in our case 20 minutes. If one is interested in determining how long it will take for the successive doubling to "discover" lower nodes in the tree[7], one can use the recursive relation derived in Theorem 2 of Section 2.2. One can simply declare the lowest depths of the tree to be "undiscovered" in the sense that the algorithm probably does not have a good distribution of particles down at those levels. An upper bounded guess for the low depth κ_{jk} values ($j, k \gg 1$) then yields an upper estimate for the κ_{ij} ($j \gg 1$). By Theorem 1, the maximum of these κ_{ij} values gives us κ. We did not use these estimates in our implementation because we had already decided on a preset time limit per experiment.

In our experimental trials, we define one *elementary operation* to be the movement of a single particle from root to leaf. Asymptotically, the "running time" is therefore domi-

$\mathcal{P}$ in order to sample randomly from it. One would simply consider the $|\Sigma|^{n-supp(p_i)}$ set of possible choices for p_{i+1}, choose one at random, and if it happens to be frequent, call it the choice. If not, we mark that possibility as not frequent and try again until we either find an acceptable choice or determine that none exists. (Further implementation details to be discussed in Section 4.)

[6]The random number generator used was initially proposed in [18].

[7]By "discovering" the lower depth of the tree, we mean get a good distribution of particles at the lowest depths. Our successive doubling technique will probably encounter low nodes before the GWW actually guarantees a good distribution of particles at those nodes.

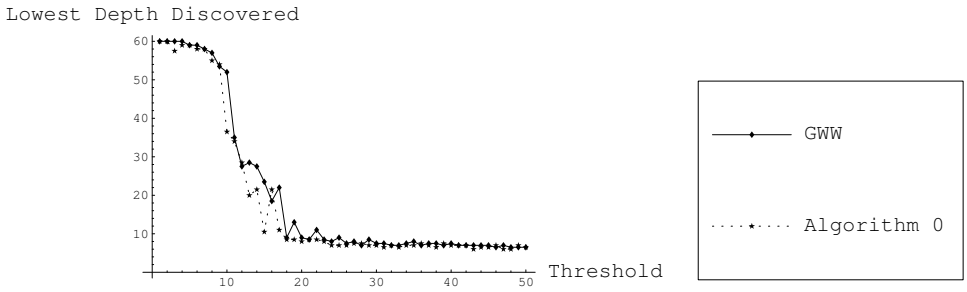

Figure 2: Experimental Results: Two trials of each averaged together.

nated by the last run of GWW.

We tested the doubling method of GWW twice using all thresholds from 1 to 50. For each twenty minute trial, we saved the depth of the lowest node encountered and the number of particles total (summed over all of the doublings) that it took GWW to encounter that depth. We then allowed the Algorithm 0 method to run using the same number of trials per threshold (i.e. with the same number of elementary operations) as the corresponding GWW trial. The results of the trials were averaged together; our results are displayed in Figure 2. Note that GWW and Algorithm 0 seem to perform comparably except in a fairly small portion of the graph between the thresholds 10 and 20. In this range, GWW significantly outperforms Algorithm 0. It is in this area that we conjecture that the tree becomes complex enough so that (i) the tree is not large enough so that it is extremely easy for Algorithm 0 to find a deep leaf node and yet (ii) the tree is not small enough so that a small number of trials with Algorithm 0 suffice to discover the entire tree. (These are the same issues discussed in Section 3.3.) For the time being, this will have to remain a conjecture however. A full enumeration of the tree at low thresholds even for a small simple database seems to be intractable. As evidence, we present a snapshot of the full poset structure of the same database at threshold 450 in Figure 3.

5. REFERENCES

[1] R. Agrawal, H. Mannila, R. Srikant, H. Toivonen, and A. Inkeri Verkamo. Fast discovery of association rules. In U. Fayyad etal, editor, *Advances in Knowledge Discovery and Data Mining*, pages 307–328, Menlo Park, CA, 1996.

[2] Rakesh Agrawal, Tomasz Imielinski, and Arun N. Swami. Mining association rules between sets of items in large databases. In Peter Buneman and Sushil Jajodia, editors, *Proceedings of the 1993 ACM SIGMOD International Conference on Management of Data*, pages 207–216, Washington, D.C., 26–28 1993.

[3] Mikl'os Ajtai, J'anos Koml'os, and Endre Szemer'edi. An O(n log n) sorting network. In *Proceedings of the Fifteenth Annual ACM Symposium on Theory of Computing*, pages 1–9, Boston, Massachusetts, April 1983.

[4] David Aldous and Umesh V. Vazirani. "Go With the Winners" Algorithms. In *IEEE Symposium on Foundations of Computer Science*, pages 492–501, 1994.

[5] C.L. Blake and C.J. Merz. UCI repository of machine learning databases, 1998.

[6] K. D. Boese, A. B. Kahng, and S. Muddu. A new adaptive multi-start technique for combinatorial global optimizations. *Operations Research Letters*, 16(2):101–113, 1994.

[7] Endre Boros, Vladimir Gurvich, Leonid Khachiyan, and Kazuhisa Makino. An inequality limiting the number of maximal frequent sets, dimacs technical report 2000-37, rutgers university. Technical report, 2000.

[8] Endre Boros, Vladimir Gurvich, Leonid Khachiyan, and Kazuhisa Makino. On the complexity of generating maximal frequent and minimal infrequent sets. In *Symposium on Theoretical Aspects of Computer Science*, pages 133–141, 2002.

[9] J. Boyan and A. Moore. Using prediction to improve combinatorial optimization search. In *Sixth International Workshop on Artificial Intelligence and Statistics*, 1997.

[10] C.Blum and A.Roli. Metaheuristics in Combinatorial Optimization: Overview and Conceptual Comparison Technical report TR/IRIDIA/2001-13, IRIDIA, Universit Libre de Bruxelles, Belgium. Technical report, 2001.

[11] Dimitriou and Impagliazzo. Go with the winners for graph bisection. In *SODA: ACM-SIAM Symposium on Discrete Algorithms (A Conference on Theoretical and Experimental Analysis of Discrete Algorithms)*, 1998.

[12] Tassos Dimitriou and Russell Impagliazzo. Towards an analysis of local optimization algorithms. In *Proceedings of Symposium on Theory of Computing*, pages 304–313, 1996.

[13] T. A. Feo and M. G. C. Resende. Greedy randomized adaptive search procedures. *Journal of Global Optimization*, 6:109–133, 1995.

[14] M. Garey and D. Johnson. *Computers and Intractability*. Freeman and Co., New York, 1979.

[15] Dimitrios Gunopulos, Roni Khardon, Heikki Mannila,

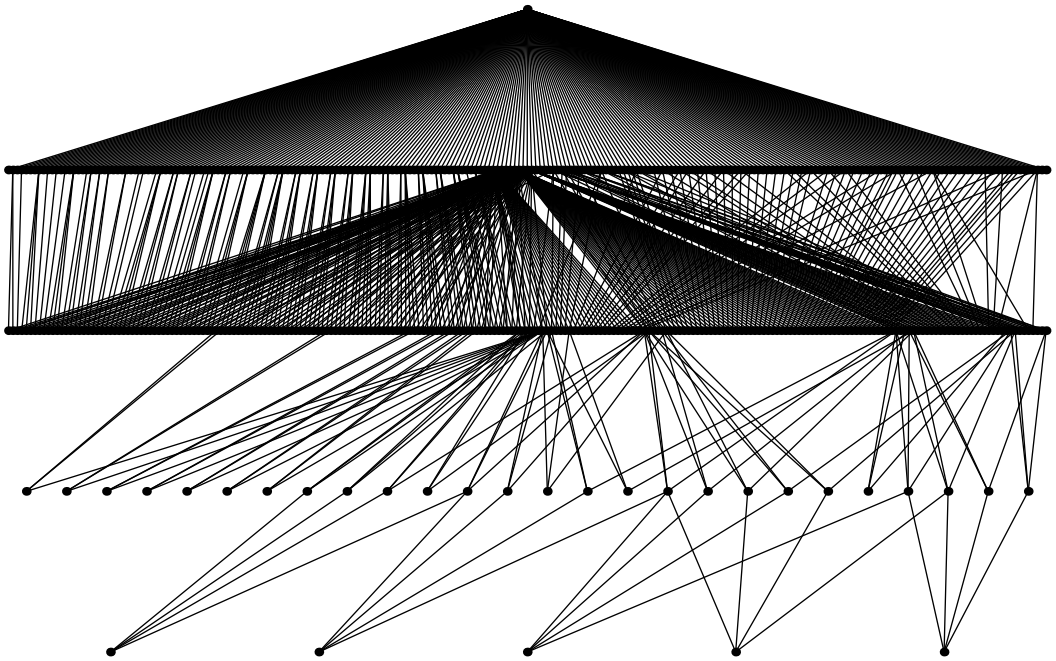

Figure 3: The poset structure of the molecular biology database at threshold level 450

Sanjeev Saluja, Hannu Toivonen, and Ram Sewak Sharma. Discovering all most specific sentences. *ACM Transactions on Database Systems*, 28(2):140–174, 2003.

[16] Dimitrios Gunopulos, Heikki Mannila, and Sanjeev Saluja. Discovering all most specific sentences by randomized algorithms. In *ICDT*, pages 215–229, 1997.

[17] Heikki Mannila, Hannu Toivonen, and A. Inkeri Verkamo. Efficient algorithms for discovering association rules. In Usama M. Fayyad and Ramasamy Uthurusamy, editors, *AAAI Workshop on Knowledge Discovery in Databases (KDD-94)*, pages 181–192, Seattle, Washington, 1994. AAAI Press.

[18] M. Matsumoto and T. Nishimura. Mersenne twister: A 623-dimensionally equidistributed uniform pseudorandom number generator. *ACM Trans. on Modeling and Computer Simulation*, 8(1):3–30, 1998.

[19] M. Peinado. Go with the winners algorithms for cliques in random graphs. In *ISAAC 2001*, pages 525–536, 2001.

[20] Jr. R. J. Bayardo. Efficiently mining long patterns from databases. In *Proceedings of the ACM SIGMOD*, pages 85–93, Seattle, Washington, 1998.

[21] V. J. Rayward-Smith, I. H. Osman, C. R. Reeves, and G. D. Smith. *Modern Heuristic Search Methods.* John Wiley and Sons, New York, 1996.

[22] M. Yagiura and T. Ibaraki. On metaheuristic algorithms for combinatorial optimization problems. *Transactions of the Institute of Electronics, Information and Communication Engineers*, J83-D-1(1):3–25.

[23] W. Zhang. *State Space Search: Algorithms, Complexity, Extensions, and Applications.* Springer-Verlag, 1999.