Modeling Social Network Relationships via t-Cherry Junction Trees

Brian Proulx and Junshan Zhang
School of Electrical, Computer and Energy Engineering, Arizona State University, Tempe, AZ, 85287

***Abstract*—The massive scale of online social networks makes it very challenging to characterize the underlying structure therein. In this paper, we employ the t-cherry junction tree, a very recent advancement in probabilistic graphical models, to develop a compact representation and good approximation of an otherwise intractable model for users' relationships in a social network. There are a number of advantages in this approach: 1) the best approximation possible via junction trees belongs to the class of t-cherry junction trees; 2) constructing a t-cherry junction tree can be largely parallelized; and 3) inference can be performed using distributed computation. To improve the quality of approximation, we also devise an algorithm to build a higher order tree gracefully from an existing one, without constructing it from scratch. We apply this approach to Twitter data containing 100,000 nodes and study the problem of recommending connections to new users.**

I. Introduction

Over the past decade, online social networks have exploded in popularity. For instance, Facebook has grown to 1.11 billion users since its inception in 2004 [1]. People with smartphones are seemingly constantly connected to multiple online social networks, such as Twitter, Facebook, or Tumblr. Even the largest social networks continue to grow at a rapid pace. For example, in May of 2013, it was measured that approximately 135,000 new users signed up for Twitter every day [2].

The enormous scale of social networks highlights the need for a systematic quantification of social relationships of users that can be updated efficiently. Clearly, an exact characterization of the joint distribution of all possible social relationships in a social network is impossible in practice, and performing perfect inference would be computationally infeasible. To address these challenges, we propose a framework building on a recently developed tool in probabilistic graphical models to approximate this joint distribution: the t-cherry junction tree, first proposed in [3]. A t-cherry junction tree is a structure that has theoretical guarantees on accuracy of approximation, as well as allowing for efficient, exact inference once it is constructed. Other graphical models are often used, such as factor graphs [4], but these offer no guarantees on their approximation and may not convergence when performing inference [5]. *Notably, when using a junction tree, any dependence loops among random variables, which often occur in social relationships, can be easily handled by incorporating those random variables into a single cluster*. Further, the t-cherry junction tree is a data-driven approach, which does not require a specific model of user relationships. In a nutshell, this approach is tailored to offer a rigorous characterization of social ties, as opposed to approaches based on heuristics. We also demonstrate the flexibility of the t-cherry junction tree by developing methods to build a higher order t-cherry junction tree approximation without constructing it from scratch.

Given the massive scale, accurate mining and approximation of large social networks cannot be computed quickly on a single machine. The amount of data to be processed calls for parallel (cloud) computing. A vital strength of the junction tree structure is the ease in which building and utilizing this structure can be parallelized, which is another subject of this study.

To demonstrate the utility of this new approach for characterizing users' relationships, we investigate a specific application, namely recommending connections to new users of a social network. This is an important problem as online social networks remain committed to attracting and keeping new users. Once a new user has created an account, ensuring that this new user is actively connected to the online community is an important method to retain new users. A user with many connections in the social network receives more updates more frequently than one with only a handful of connections. Therefore, it is important to recommend potential connections in which a new user may be interested. As a new user begins to form online relationships, the probabilities of forming relationships to different users are likely to change due to the new information gained.

For this application, we seek to predict whether a user new to the online social network will form a relationship to others in the social network. For ease of reference, we call this problem the *new user recommendation problem*, which is related to the traditional "link prediction problem" [6]. In traditional link prediction, the set of users in the network remains fixed while links are added between them over a period of time. Thus there are no new users to consider, and the timescale of interest is typically long as links may form slowly over time. Rather than having a fixed node set, the problem we study is to model the links that a new user, not a node previously in the network, will add to users currently in the network. In general, new users add connections in a social network much faster than long-time users. Thus, in the timescale under consideration, connections between users already in the network remain relatively static, whereas the

This research was supported in part by the U.S. National Science Foundation under Grant CNS-1218484, and DoD MURI project No. FA9550-09-1-0643

978-1-4799-3360-0/14/$31.00 ©2014 IEEE

new user will potentially make many new connections.

We investigate this problem from a purely topological viewpoint: we assume that the connections between users are the only data available. We do not incorporate attribute data of the users, as other works have [4] [7]. Once a new user begins to make connections to other users in the network, the probability that this user will connect with different users in the network must be updated so that the most likely social connections are recommended.

We summarize below the main contributions in modeling social relationships using a t-cherry junction tree.

- We develop a framework based on the t-cherry junction tree to approximate the underlying social relationships in a large-scale social network. Building on the greedy algorithm in [8], we construct a t-cherry junction tree by parallelizing most of the computations. Further, we present an exact method to test if each possible entry is valid by formalizing the characteristics of a t-cherry junction tree.
- We devise a scheme involving two algorithms to construct a higher order t-cherry junction tree in order to improve the approximation. This process is orders of magnitude faster than computing a higher-order tree from scratch, and indeed simulations indicate that the resulting higher order approximation reduces the KL-divergence by 150% compared to the original one. The two main steps in this scheme are: I) the order update step, in which each cluster adds a random variable from one of its neighboring clusters, and hence changes a k-order *t-cherry junction tree* to a (k+1)-order *junction tree* (not necessarily a t-cherry junction tree); II) the t-cherry conversion process, which takes the resulting *junction tree* and refines each connection between clusters to reestablish the t-cherry property, while simultaneously greedily minimizing the Kullback-Leibler divergence between the approximation and the true distribution.
- This new framework is applied to the specific problem of recommending the most likely social connections to a new user by using a 100,000 user Twitter dataset. First, an existing tool, METIS [9], is used to partition the social graph into 1,560 disjoint subgraphs. For each of these subgraphs, a 4-order t-cherry junction tree is constructed, and then these individual trees are joined together using a slight variant of the t-cherry conversion process. The resulting KL-divergence between the t-cherry junction tree and the joint distribution is calculated explicitly, which is novel as the size of the joint distribution prevents a direct computation. Additionally, we leverage this structure to perform inference to update the probability of social relationships as new relationships form. Updating these probabilities is performed in a few minutes, despite involving 100,000 random variables.

The organization of the rest of the paper is as follows. We begin with an introduction to the t-cherry junction tree in Section II and present a method for constructing it in Section III. In Section IV, a process to gracefully improve the approximation given by a t-cherry junction tree is developed. The application of these tools to the new user recommendation problem is then presented in Section V. We conclude the paper in Section VI.

II. T-Cherry Junction Trees

Denote the underlying joint distribution, containing N random variables, as $p(X_1, \ldots, X_N)$. When N is large, computing this joint distribution directly is generally too costly. Instead, a junction tree is created to approximate this distribution by using only "marginal" distributions corresponding to a small subset of random variables. Every junction tree has an underlying Markov random field that corresponds to the dependence structure inherent in the junction tree.

Definition 2.1: A **junction tree** is a tree structure over a variable set $X_1, \ldots, X_N$ with the following properties:

1) Each node of the junction tree is a subset of random variables, denoted X_C, and is called a cluster. Associated with each cluster is the distribution $p(X_C)$. These clusters represent the maximal cliques of the associated Markov random field.
2) Every edge connecting two clusters in the junction tree contains a separator. A separator is a subset of random variables containing the intersection of the two clusters being linked: $X_S = X_{C_1} \cap X_{C_2}$. As with clusters, each separator has the distribution $p(X_S)$ associated with it.
3) If a random variable is contained in two different clusters, it is also contained in every cluster on the path between those two. This is called the running intersection property.
4) The union of all clusters is the entire set of random variables $X_1, \ldots, X_N$.

Additionally, the **treewidth** of a junction tree is defined as the number of variables within the largest cluster. A slightly less restrictive concept than the junction tree is the **clique tree**. The difference is that a clique tree does not necessarily satisfy the running intersection property.

In order to characterize the quality of the junction tree approximation, we denote the entropy of a random variable X as $H(X)$ and the entropy of a random vector $\boldsymbol{X}$ as $H(\boldsymbol{X})$. Also, define the information content of a random vector $\boldsymbol{X} = [X_1, \ldots, X_k]$ as

$$I(\boldsymbol{X}) = \sum_{\boldsymbol{X}} p(\boldsymbol{X}) \log \left(\frac{p(\boldsymbol{X})}{p(X_1) \ldots p(X_k)} \right). \quad (1)$$

Using these definitions, the quality of the approximation is given by the following result [8].

Lemma 2.1: The Kullback-Leibler divergence between a junction tree approximation and the actual distribution is:

$$KL(p_{JT}(\boldsymbol{X}), p(\boldsymbol{X})) = -H(\boldsymbol{X}) - \sum_{X_C \in \mathcal{C}} I(X_C) + \sum_{X_S \in \mathcal{S}} (v_S - 1) I(X_S) + \sum_{i=1}^{N} H(X_i), \quad (2)$$

where v_S is the number of clusters which contain all of the variables in separator S.

Noting that the first and last terms of the expression do not depend on the junction tree, the closest approximation for a fixed treewidth is formed by constructing the set of clusters and separators to maximize the weight of the junction tree, defined as

$$w \triangleq \sum_{X_C \in \mathcal{C}} I(X_C) - \sum_{X_S \in \mathcal{S}} (v_S - 1) I(X_S). \tag{3}$$

However, finding the tree that maximizes the weight is an NP-hard problem [8]. Despite this challenge, there is a subclass of junction trees that is guaranteed to contain the maximum weight junction tree: t-cherry junction trees.

The t-cherry junction tree was originally derived from the t-cherry tree [10] and the simplex m-multitree [11]. We define the t-cherry junction tree in a slightly different, though equivalent, manner than it was originally presented in [3].

Definition 2.2: The **k-order t-cherry junction tree** is a junction tree with the following properties:

1) Each cluster contains k random variables.
2) Every separator contains k-1 random variables.

Thus two connected clusters, denoted C_1 and C_2, each contain a single random variable not contained in the other cluster. That is, $|C_1 \setminus C_2| = 1$, and $|C_2 \setminus C_1| = 1$.

We emphasize that the class of k-order t-cherry junction trees is important because the junction tree with the largest weight and hence the closest approximation to the actual distribution, is a member of this class [8].

Lemma 2.2: In the class of all k-width junction trees, the k-order t-cherry junction tree having the greatest weight is the best approximation to the actual distribution.

III. Data-Driven Construction of a t-Cherry Junction Tree

As the class of t-cherry junction trees is guaranteed to contain the best possible approximation among all k-order junction trees, we next investigate how to build a t-cherry junction tree. To this end, we develop an algorithm and show that the running time of this algorithm can be drastically reduced via parallel computation.

As mentioned above, constructing an optimal k-order t-cherry junction tree is an NP-hard problem. Based on [12], we design a greedy algorithm to construct a k-order t-cherry junction tree. It is worth noting that there are two key issues we resolve here. First, the process of checking if a cluster-separator pair is a valid addition is now explicitly formulated, and efficiently implemented by two checks. Our second contribution is to parallelize the process of listing all possible cluster-separator pairs, which allows the tree to be constructed significantly faster.

The basic idea of this algorithm is as follows. It begins by listing all possible clusters-separator pairs, and then orders them by weight. It selects the heaviest cluster-separator pair and adds it to the junction tree if the resulting clique tree is a valid t-cherry junction tree. If that cluster-separator pair is not a valid addition, it is discarded and the next heaviest pair is evaluated. This algorithm is motivated by the Chow-Liu algorithm for constructing a second order approximation [13], and indeed this algorithm is exactly the Chow-Liu algorithm when k is set to two. The output of this algorithm is a t-cherry junction tree, represented by a set of clusters $\mathcal{C}$ and set of separators $\mathcal{S}$, and a listing of the parent for each cluster. As clusters are added to the t-cherry junction tree, these sets are updated.

As presented in **Algorithm 1**, the construction has two phases: the first phase is **table construction**, and the second is **cluster addition**. It begins with the table construction phase. First, a table T listing all of the $k\binom{N}{k}$ possible cluster-separator pairs is constructed. That is, a list is made of all of the $\binom{N}{k}$ subsets of size k as clusters, and for each cluster, all of the k possible choices for the separator are listed. Denote each cluster-separator pair entry, with the cluster labeled as C' and the separator as S', in table T as follows: set $T(i,1) = C' \setminus S'$ and $T(i,2) = S'$. The single variable $C' \setminus S'$ is called the **dominating vertex** of the cluster [11]. For each of the $k\binom{N}{k}$ entries in table T, calculate the weight of the entry as $w = I(X_{C'}) - I(X_{S'})$ and set $T(i,3) = w$. Next, this table is sorted by heaviest weight, and the sorted table is labeled $T^\star$.

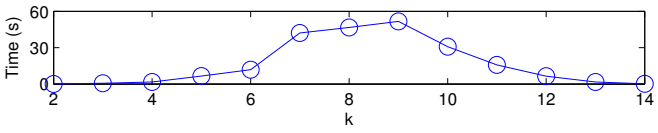
After the table $T^\star$ is constructed, the cluster addition phase takes place. Starting with an "empty" junction tree, determine whether the heaviest remaining entry can be added to the junction tree. Each cluster, aside from the first cluster entered, will have a parent associated with it. To facilitate checking if the cluster-separator addition is valid, maintain a binary vector of all nodes currently represented in the junction tree. This vector is denoted V, where $V(v) = 1$ represents that variable v is contained in at least one cluster in the junction tree.

Next, we examine the condition under which a cluster-separator pair can be added and maintain a valid t-cherry junction tree. The variables in S' must be contained within another cluster to have a valid parent. If the dominating vertex of C' is already in a cluster of the network, then this cluster-separator pair is not a valid addition because there are only two possible manners in which this occurs, neither of which results in a valid t-cherry junction tree. The first possibility is that the dominating vertex is also contained within the parent cluster. This results in the two clusters being identical, which is not valid as the separator would contain k variables. The other possibility is that the dominating vertex is contained within a cluster in the tree that is not the parent cluster. As the dominating vertex is not in S', the running intersection property would be invalidated.

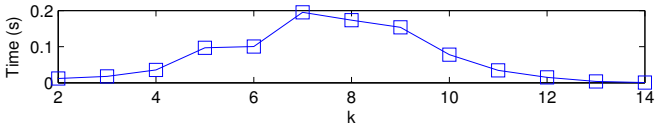
Though this approach is heuristic and cannot guarantee that the t-cherry junction tree created has the global maximum weight, it has the benefit of being largely parallelizable to decrease the overall running time. Other methods for constructing a junction tree have been proposed, but these methods operate by selecting links in the underlying Markov random field individually in a step-by-step manner [14] [15]. However, these methods cannot be parallelized and are applicable to only small datasets.

Algorithm 1 Greedy Algorithm for Constructing a k-Order t-Cherry Junction Tree

$\mathcal{C} = T^\star(1,1) \cup T^\star(1,2)$
$\mathcal{S} = \varnothing$
$V\left(T^\star(1,1) \cup T^\star(1,2)\right) = 1$
$i = 2$
while $\exists v$ such that $V(v) = 0$ **do**
  $d' = T^\star(i,1)$
  $S' = T^\star(i,2)$
  $C' = d' \cup S'$
  if $V(d') = 0$ **then**
    for $j = 1 : |\mathcal{C}|$ **do**
      $C = \mathcal{C}(j)$
      if $S' \subset C$ **then**
        parent(C') $= C$
        $\mathcal{C} = \mathcal{C} \cup C'$
        $\mathcal{S} = \mathcal{S} \cup S'$
        $V(d') = 1$
        break loop
      end if
    end for
  end if
  $i = i + 1$
end while



(a) Table construction



(b) Cluster addition

Fig. 1: Time of each phase of building a t-cherry junction tree

It can be shown that the overall time to construct a junction tree is dominated by the table construction phase on average, and therefore decreasing the time needed to construct the table will greatly decrease the time needed to build a t-cherry junction tree. To get a more concrete sense, we run the greedy algorithm using a Twitter data set [16], using 15 binary random variables and vary the width of the tree generated from 2 to 14. From this Twitter data set, we select the 500 nodes with the most connections and randomly select 15 nodes for each iteration. The results for each treewidth were averaged over 200 runs. The average times needed to generate the table for different treewidths and the average time to construct the junction tree are shown in Fig. 1. Clearly, the time to generate the table is several orders of magnitude larger than the time needed to construct the junction tree for all treewidths.

As the performance bottleneck of this approach is the table construction phase, we use parallel computing to decrease the runtime. This parallel computation can be exported via cloud computing [17] in order to drastically reduce the time needed to construct a t-cherry junction tree. Each entry of the table can be computed separately from every other entry and then these individual entries can be sorted by weight to form the complete table.

IV. A Closer Approximation: from k-Order to (k+1)-Order

While low order t-cherry junction trees can be constructed quickly, constructing high order ones from scratch would be formidable. In order to improve the approximation generated by a low order t-cherry junction tree, we propose an iterative method to update a k-order t-cherry junction tree to (k+1)-order gracefully, which refers to using an incremental process based on the previously constructed k-order tree. There are two steps to this process. The first step is the **order update** process, by which each cluster is expanded from k variables to k+1 variables. This step maintains the junction tree structure, but cannot guarantee that the t-cherry property is retained. Reestablishing the t-cherry property is the second step, namely via **t-cherry conversion**.

A. The Order Update Process

Clearly, the first step of updating from a given k-order t-cherry junction tree to a (k+1)-order junction tree is to add a "good" variable to each cluster. We develop an algorithm to add a variable to each cluster by including a variable contained in a neighboring cluster. The variable added to a cluster must be from one of its neighboring clusters in order to satisfy the running intersection property, or else the result would be a clique tree, not a junction tree.

The algorithm we design to perform this process is called the **Order Update Algorithm**, and it operates by implementing a series of **update steps**, each of which is the process of adding a variable to a cluster, until each cluster in the junction tree is of size k+1. Each update step consists of adding an **additional variable** to an **eligible cluster**, a cluster which has not yet added a variable. Each cluster can only add a single variable as the treewidth of the updated junction tree is k+1. Once an eligible cluster has added a variable, it becomes an **ineligible cluster**.

To determine which variable is the "best" variable for a cluster to add, each potential update step has a weight assigned to it. This weight represents the increase in the weight of the junction tree after the update step takes place. To calculate this weight, note that the update step process is equivalent to adding a new edge in the underlying Markov random field. This insight allows the weight to be calculated efficiently using a result from [14]. To formulate this result, and for ease of exposition, for the rest of this section we denote the **active cluster**, the cluster adding the variable, as C_A and the **donating cluster**, the neighboring cluster that contains the additional variable, C_D. The separator of these clusters is S and the additional variable is v. After the update step is

performed, we denote the resulting cluster as $C'_A \triangleq C_A \cup v$. The edge to be added in the Markov random field is between the additional variable v and the single node $d \triangleq C_A \setminus C_D$. Utilizing Corollary 4.1 in [14], the weight increase of the junction tree due to an update step is:

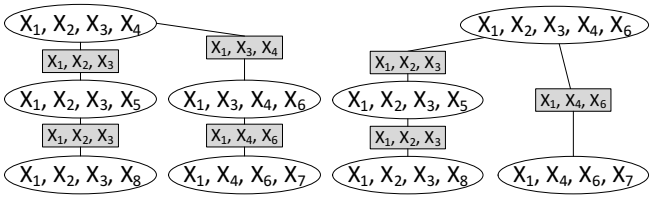
$$\Delta W(d, v) = H(S \cup d) + H(S \cup v) - H(S \cup d \cup v) - H(S). \quad (4)$$

There are three steps that need to be performed for each update step. The first is to simply include the additional variable in the active cluster, and mark the active cluster as ineligible. Now that the active cluster contains a new variable, each neighbor of the active cluster has a new potential update step as the current additional variable v could be added to each neighbor, which was not possible before. Note that the donating cluster does not have a new potential update step because it already contains the current additional variable. So the second step is to find all new potential update steps for the neighbors of the active cluster.

The third step is more involved than the previous two. As two connected clusters differ by only a single variable, some update steps will result in the donating cluster becoming a subset of the active cluster. When this occurs, the donating cluster must be removed from the junction tree, and its neighbors must be connected to the active cluster instead. After these connections are migrated to the active cluster, some potential update steps of each neighbor of the donating cluster must be changed to reflect that any potential update step that used C_D as the donating cluster must now use C'_A as the donating cluster. Also, each of these new neighbors of C'_A has a new potential update step in which the additional variable would be d.

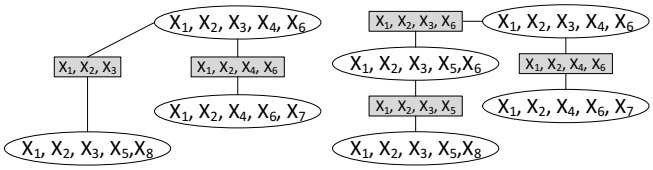
We now present the Order Update Algorithm as **Algorithm 2**. At the beginning, a list of all potential update steps is constructed, denoted T, where each entry is of the form $\{w, v, C_A, C_D\}$, which represents cluster C_A adding variable v from cluster C_D with the resulting weight increase of the junction tree being w. This list is sorted by weight. Next, update steps are executed in a greedy manner by performing the update step with the greatest w, but only if the active cluster is eligible. If not, this entry is simply discarded. After each update step, the three necessary steps previously outlined are carried out. The algorithm continues until all clusters are marked ineligible. To track the eligibility of clusters, there is a binary vector over the set of clusters denoted E, where $E(i) = 1$ represents that cluster C_i is eligible. The entire set of clusters in the junction tree is labeled $\mathcal{C}$ and the set of all neighbors of a cluster C_i is denoted $\mathcal{N}(C_i)$.

An example demonstrating a single update step to move from a 4-order to a 5-order junction tree is presented in Fig. 2 for a small junction tree. After one update step, the cluster $\{X_1, X_2, X_3, X_4\} \triangleq C_1$ adds variable X_6 from its neighboring cluster $\{X_1, X_3, X_4, X_6\} \triangleq C_2$ to become cluster $\{X_1, X_2, X_3, X_4, X_6\} \triangleq C'_1$. Because $C_2 \subset C'_1$, C_2 is removed and all connections from C_2 to any clusters aside



(a) Original (b) After update step

Fig. 2: An example of one update step



(a) After order update process (b) After t-cherry conversion

Fig. 3: An example of t-cherry conversion

from C'_1 are then connected to C'_1. Note that the separators from all neighbors of C_1 and C_2 remain unchanged after the update process. There are three updates that are made to the list of possible update steps. The first is that cluster $\{X_1, X_2, X_3, X_5\}$ can now add variable X_6 from cluster C'_1, so this entry is added to the list. The second addition is that cluster $\{X_1, X_4, X_6, X_7\}$ can now add variable X_2 from cluster C'_1. The third is that cluster $\{X_1, X_4, X_6, X_7\}$ can still add variable X_5, but the entry in T needs to be changed so that the cluster from which it adds this variable is set to C'_1, not C_2 as it was previously set. In total, there are two additions to the list of possibilities, and one change.

It is important to note that after each update step, the resulting clique tree is still a junction tree.

Theorem 4.1: Each update step preserves the junction tree structure, i.e., it does not reduce a junction tree to merely a clique tree.

The proof of this theorem is relegated to the appendix.

B. The t-Cherry Conversion Process

Though the previous algorithm returns a (k+1)-order junction tree, it does not necessarily return a (k+1)-order t-cherry junction tree. The reason for this is that the k-order t-cherry junction tree has separators that contain k-1 variables, so a (k+1)-order junction tree must have separators that contain k variables. However, the separators in the junction tree returned from the algorithm may have separators of size k-1. This occurs when two connected clusters, C_1 and C_2, have added variables from other clusters, i.e., C_1 added a variable from a cluster other than C_2, and vice-versa. This implies that the separator $S = C_1 \cap C_2$ is unchanged from its original k-1 nodes, as the intersection of the two clusters remains the same.

We next present an algorithm that converts the (k+1)-order junction tree returned from **Algorithm 2** into a (k+1)-order t-cherry junction tree. We note that this algorithm can be extended in a straightforward manner to convert any (k+1)-order junction tree, not only those returned by **Algorithm 2**,

Algorithm 2 Order Update Algorithm

1: $E(C_i) = 1 \; \forall \; C_i \in \mathcal{C}$
2: **while** $\exists i$ such that $E(i) = 1$ **do**
3: $C_A = T(1,3)$
4: **if** $E(C_A) = 1$ **then**
5: $v = T(1,2)$
6: $C_D = T(1,4)$
7: $d = C_A \setminus C_D$
8: $C_A = C_A \cup v$
9: $E(C_A) = 0$
10: **for all** $C_i \in (\mathcal{N}(C_A) \setminus C_D)$ and $E(C_i) = 1$ **do**
11: $v' = C_i \setminus C_A$
12: $w = \Delta W(v', v)$
13: Add entry $\{w, v', C_i, C_A\}$ to table T
14: **end for**
15: **if** $C_D \subset C_A$ **then**
16: **for all** $C_i \in (\mathcal{N}(C_D) \setminus C_A)$ **do**
17: Connect C_i to C_A
18: $v' = C_i \setminus C_A$
19: $w = \Delta W(d, v')$
20: Add entry $\{w, d, C_i, C_A\}$ to table T
21: **end for**
22: **for all** $T_i \in T$ **do**
23: **if** $T_i(4) = C_D$ **then**
24: $T_i(4) = C_A$
25: **end if**
26: **end for**
27: Remove cluster C_D from the junction tree
28: **end if**
29: **end if**
30: Delete the first entry in T
31: Sort T by w
32: **end while**

into a (k+1)-order t-cherry junction tree. The reason for the difference is that a general (k+1)-order junction tree may have clusters of any size between 1 and k, as the treewidth is the size of only the largest cluster in the entire junction tree, whereas in the (k+1)-order junction tree returned by **Algorithm 2**, all clusters are of size $k+1$.

An earlier algorithm [8], developed to convert a junction tree to a t-cherry junction tree, requires that a root node be selected, and the resulting tree is heavily dependent upon the choice of root node. Our approach does not require a root node to be selected, and it greedily maximizes the weight of the junction tree as it operates. Also, the underlying process, maintaining separators of size k, is clearly articulated in our algorithm.

The t-cherry conversion process is completed when each separator has been transformed to include k variables. Thus we design an algorithm to ensure that each separator contains k variables, and we call this the **t-Cherry Bud Conversion Algorithm**, presented as **Algorithm 3**. A **bud** of the t-cherry junction tree is a cluster-separator-cluster triplet, and thus the number of buds is the number of separators in the junction tree. For each separator in the junction tree, the bud can be replaced by a set of clusters and separators that have the t-cherry property and still remain a junction tree. This process transforms each bud in turn, until the entire junction tree has been converted to a (k+1)-order t-cherry junction tree. In other words, the t-cherry conversion process operates by transforming all buds into t-cherry compliant buds by transforming each using **Algorithm 3**.

Algorithm 3 operates over two clusters and their separator. Denote one cluster as C_1, the other C_2, and S as their separator. The underlying concept is to "pull forward" variables from C_1 to create more clusters in order to ensure that adjacent clusters are separated by k variables (as the treewidth is k+1) and that the running intersection property is maintained. In total, the two original clusters are replaced by $|C_2| - |S| + 1$ clusters, as each variable contained in $C_2 \setminus S$ must be a dominating vertex of its own cluster. Denote the set of dominating vertices not yet assigned to clusters as $\mathcal{D}$, and the set of dominating vertices already placed in clusters as $\mathcal{E}$. At the beginning of the algorithm, $\mathcal{D} = C_2 \setminus S$ and $\mathcal{E} = \varnothing$. A key component of this algorithm is the set of "free" variables; that is, the set of variables that can be pulled forward from the previous cluster to a new cluster to ensure the separator is of size k. We denote this set of free variable as $\mathcal{F}$. The first iteration is slightly different from the next $|C_2| - |S|$ iterations so it is outside of the main loop. The sets of the new clusters and separators returned are denoted $\mathcal{C}'$ and $\mathcal{S}'$ respectively. These two sets replace C_1, S, and C_2 in the junction tree. For each iteration, the choice of F and d are made to maximize the weight of the resulting cluster-separator pair, as seen in lines 4 and 11.

An example of this process is shown in Fig. 3. Fig. 3(a) shows a 5-order junction tree, after the order update process that began in Fig. 2 has completed all of its steps. Notice that the right branch already satisfies the t-cherry property, so it does not need to be converted. However, the left branch, the cluster-separator-cluster connection $\{X_1, X_2, X_3, X_4, X_6\}$ $\{X_1, X_2, X_3\}$ $\{X_1, X_2, X_3, X_5, X_8\}$, does not satisfy the t-cherry property as the separator has only 3 variables, not 4. The result after running **Algorithm 3**, shown in Fig. 3(b), has an additional cluster inserted between the two original clusters to make a t-cherry junction tree.

In order to convert all buds into t-cherry buds, **Algorithm 3** is run twice on each bud. For the second application of this algorithm, the choice of C_1 and C_2 is reversed. After both cluster and separator sets are calculated, the choice that results in the heaviest weight is used to replace the bud in the junction tree.

As with **Algorithm 2**, it is vital to show that this approach returns not only a junction tree structure, not merely a clique tree, but also specifically a t-cherry junction tree.

Theorem 4.2: The clique tree, constructed via performing **Algorithm 3** twice on each bud, is a t-cherry junction tree.

The proof of this result can be found in the appendix.

To demonstrate the improvement from updating a k-order

Algorithm 3 t-Cherry Bud Conversion Algorithm

Require: $C_1, C_2, S = C_1 \cap C_2$
$\mathcal{D} = C_2 \setminus C_1$
$\mathcal{C}'_1 = C_1$
$\mathcal{F} = \{F : F \subset C_1, |F| = k - |S|\}$
$F^\star, d^\star = \underset{F\in\mathcal{F}, d\in\mathcal{D}}{\arg\max}\ I(d \cup S \cup F) - I(S \cup F)$
$\mathcal{C}'_2 = d^\star \cup S \cup F^\star$
$\mathcal{S}'_1 = S \cup F^\star$
$\mathcal{D} = \mathcal{D} \setminus d^\star$
$\mathcal{E} = d^\star$
for $i = 2 : (|C_2| - |S|)$ **do**
  $\mathcal{F} = \{F : F \subset \left(\mathcal{S}'_{i-1} \setminus (\mathcal{E} \cup S)\right), |F| = k - |\mathcal{E}| - |S|\}$
  $F^\star, d^\star = \underset{F\in\mathcal{F}, d\in\mathcal{D}}{\arg\max}\ I(d \cup \mathcal{E} \cup S \cup F) - I(\mathcal{E} \cup S \cup F)$
  $\mathcal{C}'_{i+1} = d^\star \cup \mathcal{E} \cup F^\star \cup S$
  $\mathcal{S}'_i = \mathcal{E} \cup F^\star \cup S$
  $\mathcal{D} = \mathcal{D} \setminus d^\star$
  $\mathcal{E} = \mathcal{E} \cup d^\star$
end for
return $\mathcal{C}', \mathcal{S}'$

t-cherry junction tree to a (k+1)-order t-cherry junction tree, we apply this approach to each of the 1,560 individual t-cherry junction trees constructed in Section V. The average increase in the weight of the junction tree after the order update step is 19%, and the average weight increased another 110% on top of the order update weight after performing the t-cherry conversion process. Thus after both steps, on average the weight of the t-cherry junction tree increased 150%. Both the order update and t-cherry conversion processes combined required five seconds on average to run for each individual tree, a significantly shorter amount of time compared to building a 5-order t-cherry junction tree from scratch, which takes roughly one and a half hours. This approach is a very fast method to significantly improve the approximation of a t-cherry junction tree.

V. An Example Application: New User Recommendation in a Twitter Network

Next we apply the tools developed in the previous sections to the new user recommendation problem. A t-cherry junction tree is constructed to approximate the joint distribution of social connections in the social network, and then this junction tree is used to perform inference to update the probabilities of social links forming. Our experimental setup is to approximate the probability of following users in a set of 100,000 Twitter users using the dataset [16]. We first preprocess the data and extract the 100,000 most connected users. The random variables in this application are whether a new user will follow a certain user. The joint distribution $p(X_1, \ldots, X_N)$ is a collection of $N = 100,000$ binary random variables, where $X_i = 1$ represents the outcome that a new user to Twitter will follow user i. To construct a t-cherry junction tree, the "marginal" distributions containing k and $k-1$ variables must be computed. Rather than attempting to model human social behavior directly, we use a data driven approach instead. Each of these distributions is computed using the empirical data by counting the number of the 100,000 users that follow each user.

As constructing a t-cherry junction tree requires constructing a table with $k\binom{N}{k}$ entries, a small treewidth, $k = 4$, is chosen, as is typical for constructing junction trees. Due to the fact that directly constructing a 100,000 variable t-cherry junction tree is computationally infeasible, the social graph is partitioned using METIS [9], specifically the implementation [18], into 1,560 disjoint subgraphs, with each subgraph containing 64 or 65 users. We call the t-cherry junction tree constructed for each subgraph an **individual tree**. Each individual t-cherry junction tree can be computed in a reasonable amount of time, roughly five minutes, using **Algorithm 1**. It is necessary to partition the graph into *disjoint* subgraphs in order to ensure that when the individual t-cherry junction trees are joined together, the running intersection property is not violated. The METIS software partitions by attempting to minimize the number of edges cut to form the partitions. Retaining the maximum number of edges in the partitions is ideal as attempting to keep densely connected social communities together is important. In order to build these individual trees, 7 computers were used operating in parallel.

Next, we examine the process of connecting the individual t-cherry junction trees together in order to form one connected t-cherry junction tree. Two individual t-cherry junction trees can be joined together by connecting two clusters, one from each tree, using **Algorithm 3** to ensure that the resulting combination is still a t-cherry junction tree. It is important that only two clusters are joined together; otherwise loops will be present, and thus it will not be a junction tree. There are two sub-problems in this process. The first is that each individual tree can be joined with any other tree, and the second is that after choosing which two individual trees to join together, each cluster within each tree is a potential choice to use in connecting these two trees.

To attempt to maintain the original social structure of the full social graph, the number of edges cut in the partitioning process is the metric used to decide which individual trees to be joined. The process begins by computing the number of edges cut between all partitions, and then combining the two individual trees that have the maximum number of edges cut between them. This process continues in a greedy manner by selecting the individual tree that has the most edges cut to any of the already joined individual trees. This continues until all of the individual t-cherry junction trees have been joined together.

Having selected which individual t-cherry junction trees to join, the other sub-problem is to decide which two clusters should be connected to actually join the junction trees together. To join these individual junction trees together, the social structure of the original graph was considered. For each two individual trees being joined together, the number of edges cut

between the users in each cluster in both trees was counted. Then the two clusters which had the most edges cut between them were selected to be connected in order to join the two individual trees. After selecting these two clusters, they are linked together using **Algorithm 3**, which ensures that the t-cherry property is preserved. After all individual trees were joined together in this manner, the result was a single t-cherry junction tree containing 100,000 variables. This t-cherry junction tree captures the joint probability distribution of a new user following any of the existing 100,000 users.

The entire tree building process, from initial partitioning through the combination of the individual trees took slightly over 19 hours, the vast majority of which was spent constructing the individual junction trees. To calculate the KL-divergence between this approximation and the joint distribution, we calculate the overall weight of this combined junction tree as 926. The estimated joint entropy is 11.5 and the sum of the individual entropies is 12,596. Using Equation (2), the KL-divergence of this approximation is 13,511. While this represents a large difference between the approximation and the joint distribution, this KL-divergence is still calculable, which is novel, and this approximation uses only 99,997 distributions containing 4 variables, and 99,996 distributions containing 3 variables, resulting in a total of 2,399,920 data entries to store these distributions, to model a distribution containing 100,000 variables. Compared to storing the full distribution, which requires $2^{100,000}$ entries, this is a very small number of entries.

With the complete t-cherry junction tree constructed, the final piece of the solution to the new user recommendation problem can be developed. This last piece is the inference process. Once a new user has begun to select other users to follow, the values of the random variables representing these relationships are known, and the probability of the unknown relationships can then be updated.

Computing this new distribution, performing inference, for a junction tree can be evaluated in a computationally efficient manner by using message passing to update all clusters to their new distributions [19]. This process is quick and there are other works to parallelize this process [20], and the software suite GraphLab can be used as an off-the-shelf implementation [21]. For the full junction tree containing all 100,000 users, inference takes less than two minutes. We test our approach by calculating the area under the ROC curve (AUC) by testing 1,000 random users contained in the Twitter dataset, but not in the 100,000 users used to build the junction tree. The resulting AUC is 0.5519, which is impressive considering the massive scale of this social network and the low order of the t-cherry junction tree.

VI. Conclusion

In this paper, we developed a framework based on the t-cherry junction tree to characterize users' relationships in online social networks. To this end, we devised an algorithm to construct a k-order t-cherry junction tree where most of the computations are parallelized. In order to improve the approximation further, we proposed a scheme consisting of the order update and t-cherry conversion steps to construct a higher order t-cherry junction tree. The proposed scheme is significantly faster than building a higher order t-cherry junction tree from scratch and greatly decreases the KL-divergence between the approximation and the joint distribution compared to the original one. This new framework was applied to the new user recommendation problem by creating a probabilistic model of 100,000 user relationships in a Twitter dataset by building 1,560 individual t-cherry junction trees and connecting them together.

To the best of our knowledge, this work is the first using a t-cherry junction tree approach to study social networks. In general, the junction tree has many strengths that allow for a compact and rigorous characterization of the underlying structure of social networks.

References

[1] (2013, May) Facebook reports first quarter 2013 results. [Online]. Available: http://investor.fb.com/releasedetail.cfm?ReleaseID=761090

[2] (2013, May) Twitter statistics. [Online]. Available: http://www.statisticbrain.com/twitter-statistics/

[3] E. Kovacs and T. Szantai, "On the approximation of a discrete multivariate probability distribution using the new concept of t-cherry junction tree," vol. 633, pp. 39–56, 2010.

[4] H. Zhuang, J. Tang, W. Tang, T. Lou, A. Chin, and X. Wang, "Actively learning to infer social ties," *Data Mining and Knowledge Discovery*, vol. 25, no. 2, pp. 270–297, 2012.

[5] K. P. Murphy, Y. Weiss, and M. I. Jordan, "Loopy belief propagation for approximate inference: An empirical study," in *In Proceedings of Uncertainty in AI*, 1999, pp. 467–475.

[6] D. Liben-Nowell and J. Kleinberg, "The link prediction problem for social networks," in *Proceedings of the twelfth international conference on Information and knowledge management*, ser. CIKM '03, 2003, pp. 556–559.

[7] M. Rowe, M. Stankovic, and H. Alani, "Who will follow whom? exploiting semantics for link prediction in attention-information networks," in *Proceedings of the 11th international conference on The Semantic Web - Volume Part I*, ser. ISWC'12, 2012, pp. 476–491.

[8] T. Szantai and E. Kovacs, "Hypergraphs as a mean of discovering the dependence structure of a discrete multivariate probability distribution," *Annals of Operations Research*, vol. 193, no. 1, pp. 71–90, 2012.

[9] G. Karypis and V. Kumar, "A fast and high quality multilevel scheme for partitioning irregular graphs," *SIAM J. Sci. Comput.*, vol. 20, no. 1, pp. 359–392, Dec. 1998.

[10] J. Bukszar and A. Prekopa, "Probability bounds with cherry trees," *Mathematics of Operations Research*, vol. 26, no. 1, pp. pp. 174–192.

[11] J. Bukszar, "Upper bounds for the probability of a union by multitrees," *Advances in Applied Probability*, vol. 33, no. 2, pp. pp. 437–452.

[12] T. Szantai and E. Kovacs, "Discovering a junction tree behind a markov network by a greedy algorithm," *Optimization and Engineering*, vol. 14, pp. 503–518, 2013.

[13] C. Chow and C. Liu, "Approximating discrete probability distributions with dependence trees," *Information Theory, IEEE Transactions on*, no. 3, pp. 462–467, May.

[14] A. Deshpande, M. Garofalakis, and M. Jordan, "Efficient stepwise selection in decomposable models," in *In Proc. UAI*. Morgan Kaufmann Publishers, 2001, pp. 128–135.

[15] F. Malvestuto, "A backward selection procedure for approximating a discrete probability distribution by decomposable models," in *In Proc. Kybernetika*, 2012, pp. 825–844.

[16] R. Zafarani and H. Liu, "Social computing data repository at ASU," 2009. [Online]. Available: http://socialcomputing.asu.edu

[17] A. Iosup, S. Ostermann, M. Yigitbasi, R. Prodan, T. Fahringer, and D. H. J. Epema, "Performance analysis of cloud computing services for many-tasks scientific computing," *Parallel and Distributed Systems, IEEE Transactions on*, vol. 22, no. 6, pp. 931–945, 2011.

[18] (2013, July) Metis 5.1.0 software. [Online]. Available: http://glaros.dtc.umn.edu/gkhome/metis/metis/download

[19] F. Kschischang, B. Frey, and H.-A. Loeliger, "Factor graphs and the sum-product algorithm," *Information Theory, IEEE Transactions on*, vol. 47, no. 2, pp. 498–519, 2001.

[20] Y. Xia and V. Prasanna, "Distributed evidence propagation in junction trees on clusters," *Parallel and Distributed Systems, IEEE Transactions on*, vol. 23, no. 7, pp. 1169–1177, 2012.

[21] Y. Low, J. Gonzalez, A. Kyrola, D. Bickson, C. Guestrin, and J. M. Hellerstein, "Graphlab: A new parallel framework for machine learning," in *Conference on Uncertainty in Artificial Intelligence (UAI)*, Catalina Island, California, July 2010.

APPENDIX A
PROOF OF THEOREM 4.1

It suffices to check all conditions in Definition 2.1. Denote the active cluster as C_1 and the neighboring cluster containing the additional variable v as C_2, with the separator S. Additionally, denote the new cluster of size k+1 as C_1', and the set of all clusters connected to C_1, excluding C_2, as $\mathcal{N}_1$, where these neighboring clusters are labeled $N_{1_i} \in \mathcal{N}_1$, and likewise for nodes connected to C_2, excluding C_1, as $N_{2_i} \in \mathcal{N}_2$.

The first condition of the definition is trivially satisfied. To check the second condition, we show that the separator S is correctly constructed, and all other separators remain unchanged. If $C_2 \not\subset C_1'$, then $S' \triangleq C_1' \cap C_2 = S \cup v$. Next, we demonstrate that each N_{1_i} is connected to C_1' with the same separator it originally had, which we denote $S_{1_i} = C_1 \cap N_{1_i}$. First, no variables need to be removed from S_{1_i} as all variables in C_1 are in C_1'. Secondly, no variables need to be added to S_{1_i} due to the fact that the running intersection property of the original tree guarantees that $v \notin N_{1_i}$. Thus S_{1_i} remains unchanged for every i. If C_2 is removed from the junction tree, we show that each $S_{2_i} \triangleq C_2 \cap N_{2_i}$ remains the same despite connecting to C_1' instead of C_2. As $C_2 \subset C_1'$, no variables need to be removed from this separator. No variables need to be added as the running intersection property of the original tree ensures that C_1' and S_{2_i} do not share any variables that were not in C_2.

The third condition, the running intersection property, clearly still holds if $C_2 \not\subset C_1'$. If $C_2 \subset C_1'$, each N_{2_i} now connected to C_1' still satisfies the running intersection property because every variable in C_2 is contained in C_1'. So any path between clusters with a shared variable that originally traversed C_2 now traverses C_1' with that same variable. And clearly the fourth condition is satisfied as no variables were removed from the junction tree.

APPENDIX B
PROOF OF THEOREM 4.2

We begin with proving that this approach preserves the junction tree structure by checking the conditions of Definition 2.1. Denote the original clusters C_1 and C_2 and their separator as S. The resulting replacement clusters are labeled $C_{r_1}, \ldots, C_{r_{|C_2|-|S|+1}}$.

The first and second conditions of the definition are clearly satisfied. In order to prove the running intersection property, note that $C_{r_1} = C_1$, and likewise $C_{r_{|C_2|-|S|+1}} = C_2$. Any path originally terminating on C_1 or C_2 still has the running intersection property. Therefore, once it is proved that $C_{r_1}, \ldots, C_{r_{|C_2|-|S|+1}}$ satisfy the running intersection property internally, the overall running intersection property holds. This property is violated if and only if C_{r_i} contains a variable present in $C_{r_j} \setminus C_{r_{i-1}}$ $\forall j < i-1$. The dominating vertex of C_{r_i}, $d_i^\star$ is not contained in any previous vertex due to the fact that is was selected from $\mathcal{E}$. So only the remaining k nodes of C_{r_i} could possibly be contained within C_{r_j}. The original separator S is contained within every cluster, thus only the remaining $k-|S|$ variables of C_{r_i} could possibly be contained within $C_{r_j} \setminus C_{r_{i-1}}$. The dominating vertex of $C_{r_{i-1}}$, $d_{i-1}^\star$, is contained within $\mathcal{E}$ when C_{r_i} is constructed and is not contained in any cluster C_{r_j} $j < i-1$ by the construction of $\mathcal{E}$. Additionally, any previous dominating vertex is contained within $\mathcal{E}$ for the next iteration and is thus is all future clusters. Therefore only $k-|S|-|E|$ variables remain as candidates for variables contained in $(C_{r_i} \cup C_{r_j}) \setminus C_{r_{i-1}}$. These remaining variables are precisely the variables chosen from the "free variable" set $\mathcal{F}$. However, by the definition of $\mathcal{F}$, all possible $F \in \mathcal{F}$ are a subset of $C_{r_{i-1}}$. Thus there are no variables that could be present in $C_i \setminus C_{i_1}$ that are present in C_{r_j} $\forall j < i-1$. Therefore the running intersection property is satisfied. The fourth condition of the definition holds as no variables are removed.

As each call of **Algorithm 3** preserves the junction tree structure, the entire process also preserves the junction tree structure.

To prove that a t-cherry junction tree structure is returned, note that if each call of **Algorithm 3** returns a t-cherry junction tree, then the resulting junction tree from transforming each bud is a t-cherry junction tree. A (k+1)-order junction tree is a (k+1)-order t-cherry junction tree if and only if each separator $S = C_1 \cap C_2$ contains k nodes, i.e., C_1 and C_2 each contain only a single node that is not contained within the other cluster. This is clear to see as:

$$\begin{aligned} C_{r_{i+1}} \setminus C_{r_i} &= (d_{i+1}^\star \cup \mathcal{E}_{i+1} \cup F_{i+1}^\star \cup S) \setminus (d_i^\star \cup \mathcal{E}_i \cup F_i^\star \cup S) \\ &= d_{i+1}^\star \cup (\mathcal{E}_{i+1} \setminus (d_i^\star \cup \mathcal{E}_i)) \cup (F_{i+1}^\star \setminus F_i^\star) \quad (5) \\ &= d_{i+1}^\star. \quad (6) \end{aligned}$$

Similarly,

$$\begin{aligned} C_{r_i} \setminus C_{r_{i+1}} &= (d_i^\star \cup \mathcal{E}_i \cup F_i^\star \cup S) \setminus (d_{i+1}^\star \cup \mathcal{E}_{i+1} \cup F_{i+1}^\star \cup S) \\ &= ((d_i^\star \cup \mathcal{E}_i) \setminus \mathcal{E}_{i+1}) \cup (F_i^\star \setminus F_{i+1}^\star) \quad (7) \\ &= F_i^\star \setminus F_{i+1}^\star. \quad (8) \end{aligned}$$

Note that $|F_i^\star \setminus F_{i+1}^\star| = 1$ as

$$\begin{aligned} |F_i^\star \setminus F_{i+1}^\star| &= k - |\mathcal{E}_i| - |S| - (k - |\mathcal{E}_{i+1}| - |S|) \quad (9) \\ &= |\mathcal{E}_{i+1}| - |\mathcal{E}_i| = 1. \quad (10) \end{aligned}$$

Therefore each pair of connected clusters differs by only a single node in each.