Synchronization Graphs

R-KFC-A

1 Introduction

In this we examine parsing streams of tagged vertices into a canonical *synchronization graph*.

2 Background

Definition 1 (Dependency Relation)**.** A dependency relation $D \subseteq \Sigma \times \Sigma$ is a *symmetric* and *reflexive* relation on Σ.

Likewise an independence relation of D can be defined as the relative complement $I = (\Sigma \times \Sigma) \setminus D$.

Dependency relations are a general way of talking about *equivalence* relations between two streams of data $S_1, S_2 \in \Sigma^\star$, where we say $S_1 \equiv_D S_2$ if S_2 can be reached from S_1 (and vice versa) by appling permutations based on the independence relation I.

Definition 2 (Tree Dependence Relation)**.** A tree dependence relation $T \subseteq \Sigma \times \Sigma$ is a dependence relation such that (Σ, T) induces a graph with vertices Σ and edges T. The skeleton of (Σ, T) forms a tree with distinguished root $\sigma_\top$. Furthermore every vertex is upwards connected: if $\sigma_1 \in \mathbf{ancestors}(\sigma_2)$, then $\sigma_1 T \sigma_2$.

Anton: I have trouble describing "skeleton". It's just suppoesd to mean something similar to the "tree-like" dependency relation I've talked about before on the Jamboad"

There are a few functions we can define on T: Define the predecessor function with respect to this rooting, with $\mathbf{pred}(\sigma_\top) = \sigma_\top$; recursively define the depth function $\mathbf{depth}(\sigma_\top) = 0$ and $\mathbf{depth}(\sigma) = 1 + \mathbf{depth}(\mathbf{pred}(\sigma))$.

We define a *synchronization graph*, which is intended to model data streams that (1) are equipped with a dependence relationship and (2) have "synchronizing" (also: visibly pushdown / parallel / end-marker'd) behavior.

Definition 3 (Synchronization Graph)**.** A synchronization graph G is a directed acyclic graph with a unique *source* (top) vertex $\vee G$ and a unique *sink* (bot) vertex $\wedge G$. Recursively define G as follows:

i. (Base Case): A single vertex is a syncrhonization graph.

ii. (Sequential Concatenation): If G_1 and G_2 are synchronization graphs, then $G = G_1 \cdot u \cdot G_2$ is also a synchronization graph for a new vertex u, where

$$(\wedge G_1, u), (u, \vee G_2) \in G, \qquad \vee G = \vee G_1, \qquad \wedge G = \wedge G_2$$

iii. (Parallel Union): If G_1 and G_2 are synchronization graphs, then $G = u[G_1||G_2]v$ is also a synchronization graph for new vertices u, v where

$$(u, \vee G_1), (u, \vee G_2), (\wedge G_1, v), (\wedge G_2, v) \in G, \qquad \vee G = u, \qquad \wedge G = v$$

Note that synchronization graphs induce a natural partial order: $u \preceq v$ if either (1) $u = v$ or (2) v is reachable from u.

3 A Parsing Problem

In this problem setting, we are given:

i. A tree dependency relation T and its alphabet Σ

ii. A sequence of vertices $v_1, v_2, \ldots$ each labeled with an element of Σ: that is, $\tau(v_i) \in \Sigma$ for all i. This sequence is a linearization of a stream satisfying T.

The task is to generate a *canonical* synchronization graph.

4 A Basic Algorithm

A basic algorithm we have is to iteratively grow G. Let **next**() return the next vertex in the stream; iteratively keeping track of a *frontier* of leaf vertices L, every new vertex v has one of three possibilities:

1. It depends on exactly one $l \in L$.
2. It depends on more than one $M \subseteq L$. Furthermore, $\mathbf{depth}(\tau(m)) < \mathbf{depth}(\tau(v))$ for all $m \in M$. In this case v acts as a common "synchronization point" for M.
3. v is independent with all of L, in which case the *most recent depenency* of v, written $u^\star = \bigvee\{u \in G : \tau(u) T \tau(v)\}$ is found and v appends to this.

Parsing Linearized Stream 1: Algorithm

Precondition: data stream S begins and ends with a vertex labeled $\sigma_\top$

Initialize $L \to \emptyset$

Initialize $(V, E) \leftarrow (\emptyset, \emptyset)$.

Pop the first $v \leftarrow \textbf{next}()$ and set $V \leftarrow \{v\}$.

while $v \leftarrow \textbf{next}()$ *succeeds* **do**

  Let $M = \{l \in L : \tau(l) T \tau(v)\}$ $V \leftarrow V \cup \{v\}$

  if $M = \{l\}$ **then**

    $E \leftarrow E \cup \{(l, v)\}$

    $L \leftarrow (L \setminus \{l\}) \cup \{v\}$

  else if $|M| > 1$ **then**

    $E \leftarrow E \cup \{(m, v) : m \in M\}$

    $L \leftarrow (L \setminus M) \cup \{v\}$

  else

    $u^\star = \bigvee\{u \in V : \tau(u) T \tau(v)\}$

    $E \leftarrow E \cup \{(u^\star, v)\}$

    $L \leftarrow L \cup \{v\}$

return $G = (V, E)$

Lemma 1. *Two streams that are T-equivalent have the same decomposition under Algorithm 1.*

Proof. **TODO** □

Lemma 2. *At each iteration of the algorithm, there is at most one $l \in L$ such that* $\textbf{depth}(\tau(l)) = \textbf{depth}(\tau(v))$ *and* $\tau(l) T \tau(v)$.

Proof. **TODO: something about induction** □

Lemma 3. *At the each iteration, (V, E) respects T.*

Proof. **TODO: Also something about induction** □