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 where (Σ, T) is a connected graph with distinguished root $\sigma_\top$, with disjoint branches, and is upwards connected: if $\sigma_1 \in \textbf{ancestors}(\sigma_2)$, then $\sigma_1 T \sigma_2$.

A tree dependency relation *looks like* a tree but is not necessarily one: although there is a tree-like structure, all vertices are connected to their ancestors.

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

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 source and sink vertices, $\textbf{sources}(G)$ and $\textbf{sinks}(G)$ respectively. A synchronization graph G is recursively defined 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 $G = (V, E)$ and

$$(t, u), (u, s) \in E, \qquad \text{for all } t \in \textbf{sinks}(G_1), \qquad \text{for all } s \in \textbf{sources}(G_2),$$

while $V_1 \sqcup V_2 \subseteq V$ and $E_1 \sqcup E_2 \subseteq E$.

iii. (Parallel Union): If $G_1 = (V_1, E_1)$ and $G_2 = (V_2, E_2)$ are synchronization graphs, then $G = G_1 || G_2$ is also a synchronization graph where $G = (V_1 \sqcup V_2, E_1 \sqcup E_2)$.

Note that synchronization graphs induce a natural partial order: $u \preceq v$ if either (1) $u = v$ or (2) v is reachable from u. In formally then, $\textbf{sources}(G)$ and $\textbf{sinks}(G)$ are the *least* and *greatest* elements of G respectively.

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 $\textbf{next}()$ 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 $l \in L$ such that $\textbf{depth}(v) = \textbf{depth}(l)$.
2. It depends on more than one $M \subseteq L$. Furthermore, $\textbf{depth}(\tau(m)) < \textbf{depth}(\tau(v))$ for all $m \in M$. In this case v acts as a common "synchronization point" for M.

   Note that this can generalize case 1, but is separate purely for presentation.
3. v is *not* dependent any of L, and there exists a *most recent depenency* of v, written $u^\star = \bigvee\{u \in G : \tau(u) T \tau(v)\}$ is found and v appends to this.
4. v is *not* dependent any of L, and there does *not* exist a $u^\star$, then v is parallel unioned to G.

Parsing Linearized Stream 1: Algorithm

```
Initialize L → ∅
Initialize (V, E) ← (∅, ∅).
while v ← next() succeeds do
    M ← {l ∈ L : τ(l)Tτ(v)}
    V ← V ∪ {v}
                                        /* There is something for v to merge with */
    if |M| > 0 then
        V ← V ∪ {v}
        E ← E ∪ {(m, v) : m ∈ M}
        L ← (L \ M) ∪ {v}
    else
        u⋆ = ⋁{u ∈ V : τ(u)Tτ(v)}
                    /* There is a ''most recent dependency'' of v to merge with */
        if u⋆ exists then
            V ← V ∪ {v}
            E ← E ∪ {(u⋆, v)}
            L ← L ∪ {v}
                          /* v is to be parallel unioned with the existing graph */
        else
            V ← V ∪ {v}
            L ← L ∪ {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* $\mathbf{depth}(\tau(l)) = \mathbf{depth}(\tau(v))$ *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** □