Smart Plans in Knowledge Bases

John Lennon

October 16, 2017

1 Definitions

1.1 Linear Path

Definition 1 (Linear Path)**.** *A path is said to be linear if it can be written with n relations $r_1...r_n$ and has the form $P(X_1, ..., X_n) = r_1(X_1, X_2)...r_n(X_n, X_{n+1})$.*

Notation 1. *To simply, we will write $P = r_1...r_n$.*

Definition 2 (Linear function)**.** *A query or a function is said to be linear if it is a linear path. In what follows, we will always suppose that X_1 is the input of the function or the query, the other varibles being the outputs.*

1.2 Smart Plans

Definition 3 (Smart Plan)**.** *A plan P is said to be smart if, for a given query Q and for all knowledge bases K, either P gives no result on K or an answer of Q in K is among the results of calling P for K. (We consider that if Q has no answer at all on K, then a path is always smart on K).*

In what follows, we suppose that we have access to all intermediate outputs of a linear path.

Property 1. *A linear smart plan P for a linear query $Q = r_1...r_n$ is of the form $P = l_1r_1...l_nr_nl_{n+1}$ where $l_1, ..., l_n, l_{n+1}$ are linear paths.*

1.3 Fully Categorized Knowledge Base

Definition 4 (Fully Categorized Knowledge Base)**.** *A knowledge base K is said to be fully categorized when for all N_1, N_2 nodes in K, we have:*

$$Relations(N_1) \cap Relations(N_2) = \emptyset \text{ or } Relations(N_1) = Relations(N_2).$$

This property means that, as soon as two nodes have at least one common relation, they must have identical relations.

Definition 5 (Fully Categorized Knowledge Base Under a Set of Functions)**.** *A knowledge base K is said to be fully categorized under a set F of functions when for all N_1, N_2 nodes in K, we have:*

$$Relations(N_1) \cap Relations(N_2) \subset \neg S \;\; or$$
$$Relations(N_1) \cap S = Relations(N_2) \cap S.$$

where S is the set of all relations used in all functions in F.

It means that a knowledge base is fully categorized if we ignore all relations which are not used by functions in F.

1.4 Way-Back Lists

Definition 6 (Way-Back List 1)**.** *A way-back list is a list recognized by the following algorithm.*

Algorithm 1: Way-Back Algorithm

```
Data: a list l of relations
Result: whether l is a way-back list or not
σ = empty stack;
for c in l do
    if c == head(σ)^-1 then
        pop(σ);
    end
    else
        push(σ, c);
    end
end
if σ is empty then
    return l is a way-back list.
end
else
    return l is not a way-back list.
end
```

Definition 7 (Way-Back List 2)**.** *The way-back lists can be defined by the following context-free grammar:*

- *Terminals: all possible relations r in a knowledge base and their opposite r^{-1}.*
- *Non-Terminals: C*
- *Start Symbol: S*
- *Relations:*
  - $S \to C$
  - *For all r in the relations,* $C \to CrCr^{-1}C$
  - $C \to \epsilon$

Definition 8 (Way-Back List 3)**.** *The way-back lists can be defined by the following indexed grammar:*

- *Terminals: all possible relations r in a knowledge base and their opposite r^{-1}.*
- *Non-Terminals: C*
- *Start Symbol: S*

- *Relations:*
  - $S \to C[]$
  - $C[] \to \epsilon$
  - *For all r in the relations:*
    - $C[\sigma] \to C[\sigma r^{-1}]r$
    - $C[\sigma] \to C[\sigma]rC[r^{-1}]$
    - $C[\sigma] \to rC[r^{-1}\sigma]$
    - $C[\sigma] \to C[r^{-1}]rC[\sigma]$
    - $C[r\sigma] \to rC[\sigma]$

Theorem 1. *The definitions 6, 7 and 8 are equivalent.*

Proof. Let's call L_1, L_2 a,d L_3 the languages generated by definitions 6, 7 and 8.
Let l be a list recognized by definition 6. To find the associated rules that generate l using definition 8, we say that when an element r is pushed on the stack, we use the rule $C[\sigma] \to rC[r^{-1}\sigma]$ and when an element is pulled from the stack, we use the rule $C[r\sigma] \to rC[\sigma]$. As at the end, the stack is empty, it also means that the stacks in the grammar are also empty (everything went out thanks to the last rule) and we can finish by doing $C[] \to \epsilon$. So, we have $L_1 \subseteq L_3$.
In the same way, we can transform the grammar rules into push/pop operations:

1. $C[\sigma] \to C[\sigma r^{-1}]r$: r^{-1} was pushed and now we pop it, leaving the other operations on the left.
2. $C[\sigma] \to C[\sigma]rC[r^{-1}]$: we push r and in the future we will pop it, leaving the other operations on the left.
3. $C[\sigma] \to rC[r^{-1}\sigma]$: we push r and in the future we will pop it, leaving the other operations on the right.
4. $C[\sigma] \to C[r^{-1}]rC[\sigma]$: r^{-1} was pushed and now we pop it, leaving the other operations on the right.
5. $C[r\sigma] \to rC[\sigma]$ allows the individual past and future pops and pushes.

For example, if we consider the sequence:
$C[] \to^1 C[a^{-1}]a$
$\to^2 C[a^{-1}]bC[b^{-1}]a$
$\to^5 a^{-1}bC[b^{-1}]a$
$\to^3 a^{-1}bcC[c^{-1}b^{-1}]a$
$\to^4 a^{-1}bcC[d^{-1}]dC[c^{-1}b^{-1}]a$
$\to^{3*5} a^{-1}bcd^{-1}dc^{-1}b^{-1}a$
where $\to^i$ is the call to the i^{th} rule in the grammar. The sequence can be transformed into:
$push_{TODO}(a^{-1}), pop(a)$
$push_{TODO}(a^{-1}), push(b), pop_{TODO}(b^{-1}), pop(a)$
$push(a^{-1}), push(b), pop_{TODO}(b^{-1}), pop(a)$
$push(a^{-1}), push(b), push(c), pop_{TODO}(c^{-1}), pop_{TODO}(b^{-1}), pop(a)$
$push(a^{-1}), push(b), push(c), push_{TODO}(d^{-1}), pop(d), pop_{TODO}(c^{-1}),$
$pop_{TODO}(b^{-1}), pop(a)$
$push(a^{-1}), push(b), push(c), push(d^{-1}), pop(d), pop(c^{-1}), pop(b^{-1}), pop(a)$
At the end, the stack is empty because for all elements, we plan to pop it. It cannot get stuck (impossible to pop) by construction. So, we have $L_3 \subseteq L_1$ and

then $L_1 = L_3$.
The same way, we can prove $L_1 = L_2$ by transforming the rules into push/pop operation on the stack. $\square$

Property 2. *We have the following properties.*

1. *The way-back lists are closed under concatenation: if l_0 and l_1 are way-back lists, then $l_0 l_1$ is also a way-back list.*
2. *All list of relation l can be written $l = l_0 l_1 ... l_n$ where l_i are alternatively way-back and not way back lists.*

Theorem 2. *Let l be a way-back list with parameters $X_1, ..., X_n$. For all fully categorized knowledge bases K, if l admits results, then $X_n = X_1$ is one of them.*

Lemma 1. *If l is a way-back list, then its length is $2 * n$ with $n >= 0$.*

Proof of the theorem. Let K be a fully categorized knowledge base. By induction we have:
If $|l| = 0$, nothing to say.
If $|l| = 2$, there exists a relation r such that $l = rr^{-1}$ and then either r is not a relation for X_0 and in this case l admits no result or $X_0 = X_2$ is one of the solution.
If $|l| = 2*n$ and we suppose the theorem true for all lists l' with a length strictly less than $2*n$. We remove the last element of l and call it c. From the grammar definition of way-back lists, we can say that l can be written $l = l_0 c^{-1} l_1 c$ where l_0 and l_1 are way-back lists. Using the theorem, we have, if X_i is the output of l_0: $X_0 = X_i$ (from l_0) and $X_{i+1} = X_{2*n-1}$ (from l_1). So, the problem can be reduced to $l = c^{-1}c$ where we have a set of inputs with X_0 among them. If c is a relation of X_0, then we have $X_0 = X_{2*n}$ (see the case $|l| = 2$). If c is not a relation of X_0 then, as K is a fully categorized knowledge base, c is not a relation for any of the inputs of c (as all inputs share the relation before c) so l admits no result. $\square$

Theorem 3. *A plan $P = lQ$ (for a query Q composed of one relation) is smart on fully categorized knowledge bases if and only if l is a way-back list.*

Proof. Let's suppose we have a way-back list l. Let K be a fully categorized knowledge base where Q has an answer on a node X. From theorem 2, we know that if we apply l on X, either we have no result or X is among results. If we have no result, we are done. If we do have results, if we apply Q to these results, it be also applied to X and so the result of calling Q on X is among results. We conclude that we have a smart plan.

Let's suppose now we have a list l which is not a way-back list. From property 2 about way-back lists, we can decompose l into sublists. We write $l = l_0 l_1 ... l_n$ where $n > 0$ (as l is not a way-back list). w.l.o.g. we can suppose that all l_i where i are even are way-back lists whereas all l_i where i are odd are not way-back lists. One can build a knowledge base where, starting from a node X, all way-back lists come back only to the starting node and non way-back lists only go to new nodes. In such a knowledge base, $P = lQ$ is not a smart plan. $\square$

Corrollary 1. *A plan $P = l_1 r_1 ... l_n r_n$ (for a query $Q = r_1 ... r_n$) is smart on fully categorized knowledge bases if and only if l_1, ..., l_n are way-back lists.*

2 Smart Plans with Functions

2.1 Indexed Grammars

The indexed grammars were first introduced in [**?**]. They generate the class of indexed languages which contains all context-free languages and is contained within the class of context-free languages. A nice feature of this class of languages is that it conserves closure properties and decidability results. In addition to the set of terminals and non terminals from the context-free grammars, the indexed grammars introduce the set of *index symbols*.

In this work, we adopt the notations introduced by Hopcroft and Ullman in []. According to [], an indexed grammar is defined a 5-tuple $G = (N, T, F, P, S)$ where

- N is a set of variables or nonterminal symbols,
- T is a set ("alphabet") of terminal symbols,
- F is a set of index symbols (indices),
- $S \in N$ is the start symbol, and
- P is a finite set of productions.

In productions as well as in derivations of indexed grammars, a stack consisting of a string of index symbols is attached to every nonterminal symbol $A \in N$, denoted by $A[\sigma]$ ($\sigma \in F*$). For an index stack $\sigma \in F*$ and a string $\alpha \in (N \cup T)*$ of nonterminal and terminal symbols, $\alpha[\sigma]$ denotes the result of attaching (coping) $[\sigma]$ to every nonterminal in α. Each production in P takes one of the following forms:

$$\begin{aligned} A[\sigma] &\to \alpha[\sigma] \\ A[\sigma] &\to B[f\sigma] \\ A[f\sigma] &\to \alpha[\sigma] \end{aligned}$$

2.2 Generating the Index Grammar

Given a query $q(a, x)$, a set of path functions F, we explain how to translate them into index grammar rules. In what follows, all functions are supposed to be linear, with only one input.

The first thing to notice here is that, when a function $f = r_1...r_n$ is given, as it is possible to access all outputs, we can consider that we also have the subfunctions $f_1 = r_1$, $f_2 = r_1 r_2$, ..., $f_n = r_1...r_n$.

Why is it essential to have access to these subfunctions? Let's imagine we have a query q but the only function given is $f = q\,a$. The algorithm we are going to present or a simple top-down approach might consider that too much information was generated and one needs to compensate the a by an other function. With the subfunction $f' = q$, we do not have this problem.

So, from the set of path functions F, an other set of path functions can be generated which contains all the subfunction of functions in F. We call this new set F'.

We are going to use the functions from F' to explore all possible combinations of functions $f_1...f_k$, with f_1, ..., f_k in F' such that $f_1...f_k = lq$ where l is a way-back list. This way, we are going to obtain all possible smart plans for the query q.

Here, indexed grammars need to be introduced: one needs to remember which relations need to be compensate in order to create a correct way-back list. For example, if we call a function $f = a\,b\,c$, the relations a^{-1}, b^{-1} and c^{-1} will have to exist in the final result (except if a, b or c are part of the query). So, either functions created before f were called to generate them or functions will have to generate them in the future. The relations which have to be created in the future are pushed on a stack, which is the stack used in indexed grammars.

Then, each function can be consumed in several ways. At a given moment, only a part of a function may be required. So, the rest of the function will have to be compensate in the future. In what follows, we will present the indexed grammar rules which represent these partial consumptions.

2.3 Rules

Definition 9 (Middle Rules). *For a middle rule, the middle of a function is consumed and so, both the begin and the end are required to be completed in the future. Let $f = r_1...r_n$ be a linear function. We call middle rules extracted from f the production rules (for an indexed grammar):*
For all $1 \leq i \leq j \leq n+1$, we extract the rule:
$C[r_i...r_j\sigma] \rightarrow C[r_{i-1}^{-1}...r_1^{-1}]r_1...r_nC[r_n^{-1}...r_{j+1}^{-1}\sigma]$
where σ represents the stack and r_k is an empty relation if k is not in $[1;n]$.

2.4 Example

Let's suppose we have the function $f_1 = c\,c\,b$. Then, the generated rules will be:

- $C[c\,c\,b\,\sigma] = c\,c\,b\,C[\sigma]$
- $C[c\,c\,\sigma] = c\,c\,b\,C[b^{-1}\,\sigma]$
- $C[c\ \sigma] = c\,c\,b\,C[b^{-1}\,c^{-1}\,\sigma]$
- $C[\sigma] = c\,c\,b\,C[b^{-1}\,c^{-1}\,c^{-1}\,\sigma]$
- $C[\sigma] = C[b^{-1}\,c^{-1}\,c^{-1}]\,c\,c\,b\,C[\sigma]$
- $C[b\,\sigma] = C[c^{-1}\,c^{-1}]\,c\,c\,b\,C[\sigma]$
- $C[b\,c\,\sigma] = C[c^{-1}]\,c\,c\,b\,C[\sigma]$
- $C[c\,\sigma] = C[c^{-1}]\,c\,c\,b\,C[b^{-1}\,\sigma]$

Let's suppose we have the function $f_2 = c^{-1}$. Then, the generated rules will be:

- $C[c^{-1}\sigma] = c^{-1}\,C[\sigma]$
- $C[\sigma] = c^{-1}\,C[c\,\sigma]$
- $C[\sigma] = C[c]\,c^{-1}\,C[\sigma]$

2.5 Initialization Rule

To initialize our indexed grammar, we need to add the following rules:

- $S \to C[q]$
- $C[] \to \epsilon$

where S is the starting non-terminal and q is the query. Then, for all functions, the middle rules described in definition 9 must be generated (the non-terminal C is the same for all functions).

2.6 Example

Let's keep our previous functions $f_1 = c\,c\,b$ and $f_2 = c^{-1}$ and let's suppose we have the query $q = b$. A word can be found in the following way:

$$\begin{aligned} C[b] &\to^{f_1} C[c^{-1}\,c^{-1}]\,c\,c\,b\,C[] \\ &\to^{f_2} c^{-1}\,C[c^{-1}]\,c\,c\,b\,C[] \\ &\to^{f_2} c^{-1}\,c^{-1}\,C[]\,c\,c\,b\,C[] \\ &\to^{end} c^{-1}\,c^{-1}\,c\,c\,b \end{aligned} \tag{1}$$

So, a solution would be to call f_2 twice and then to call f_1.

2.7 Reduced Form Rules

Although the rules are easier to understand with definitions **??** and **??**, it is more practical for algorithmic purposes to transform the rules into a reduce form.

2.7.1 Reduced Left Rules

Let $i \in [1; n]$. The associated rule is $C[r_1 ... r_i \sigma] \to r_1 ... r_n C[r_n^{-1} ... r_{i+1}^{-1} \sigma]$. Let's transform it into reduced rules:

- $C[r_1\sigma] \to B_1[\sigma]$
- $B_1[\sigma] \to A_1[\sigma]C_2[\sigma]$
- $A_1[\sigma] \to r_1$
- $C_2[r_2\sigma] \to B_2[\sigma]$
- $B_2[\sigma] \to A_2[\sigma]C_3[\sigma]$
- $A_2[\sigma] \to r_2$
- ...
- $C_i[r_i\sigma] \to B_i[\sigma]$
- $B_i[\sigma] \to A_i[\sigma]C^{i+1}[\sigma]$
- $A_i[\sigma] \to r_i$
- $C_{i+1}[\sigma] \to A_{i+1}[\sigma]C_{i+2}[\sigma]$
- $A_{i+1}[\sigma] \to r_{i+1}$
- ...
- $C_n[\sigma] \to A_n[\sigma]C^{back}[\sigma]$
- $A_n[\sigma] \to r_n$
- $C^{back}[\sigma] \to C_{i+1}^{back}[\sigma]T[\sigma]$

- $C_{i+1}^{back}[\sigma] \rightarrow C_{i+2}^{back}[r_{i+2}^{-1}\sigma]$
- ...
- $C_{n}^{back}[\sigma] \rightarrow C_{n+1}^{back}[r_{n}^{-1}\sigma]$
- $C_{n+1}^{back}[\sigma] \rightarrow C[\sigma]T[\sigma]$
- $T[\sigma] \rightarrow \epsilon$

Here, we have a total of $4 * n - i + 1$ rules.

2.7.2 Reduced Right Rules

Let $i \in [2; n]$. The associated rule is $C[r_i ... r_n\sigma] \rightarrow C[r_{i-1}^{-1} ... r_1^{-1}]r_1 ... r_n C[\sigma]$. Let's transform it into reduced rules:

- $C[r_i\sigma] \rightarrow B[\sigma]$
- $B[\sigma] \rightarrow A_1[\sigma]D[\sigma]$
- For all relations r, $A_1[r\sigma] \rightarrow A_1[\sigma]$ (the stack is emptied).
- $A[] \rightarrow A_0^{back}[\sigma]$
- $A_0^{back}[\sigma] \rightarrow A_1^{back}[r_1^{-1}\sigma]$
- ...
- $A_{i-2}^{back}[\sigma] \rightarrow C[r_{i-1}^{-1}\sigma]$
- $D[\sigma] \rightarrow E_1[\sigma]C_{i+1}[\sigma]$
- $C_{i+1}[r_{i+1}\sigma] \rightarrow C_{i+2}[\sigma]$
- ...
- $C_n[r_n\sigma] \rightarrow C[\sigma]$
- $E_1[\sigma] \rightarrow F_1[\sigma]E_2[\sigma]$
- ...
- $E_n[\sigma] \rightarrow F_n[\sigma]E_{n+1}[\sigma]$
- $E_{n+1}[\sigma] \rightarrow \epsilon$
- $F_1[\sigma] \rightarrow r_1$
- $F_n[\sigma] \rightarrow r_n$

Here, we have a total of (Number relations $+ 3 * n + 4$) rules.

2.7.3 Reduced Middle Rules

Let $1 \leq i \leq j \leq n$, the associated rule is: $C[r_i ... r_j\sigma] \rightarrow C[r_{i-1}^{-1} ... r_1^{-1}]r_1 ... r_n C[r_n^{-1} ... r_{j+1}^{-1}\sigma]$. Let's transform it into reduced rules (it will combine methods we saw for the right rules and the left rules).

- $C[r_i\sigma] \rightarrow B[\sigma]$
- $B[\sigma] \rightarrow A_1[\sigma]D[\sigma]$
- For all relations r, $A_1[r\sigma] \rightarrow A_1[\sigma]$ (the stack is emptied).
- $A[] \rightarrow A_0^{back}[\sigma]$
- $A_0^{back}[\sigma] \rightarrow A_1^{back}[r_1^{-1}\sigma]$
- ...
- $A_{i-2}^{back}[\sigma] \rightarrow C[r_{i-1}^{-1}\sigma]$
- $D[\sigma] \rightarrow E_1[\sigma]C_{i+1}[\sigma]$
- $C_{i+1}[r_{i+1}\sigma] \rightarrow C_{i+2}[\sigma]$
- ...
- $C_j[r_j\sigma] \rightarrow C^{back}[\sigma]$
- $E_1[\sigma] \rightarrow F_1[\sigma]E_2[\sigma]$
- ...

- $E_n[\sigma] \to F_n[\sigma]E_{n+1}[\sigma]$
- $E_{n+1}[\sigma] \to \epsilon$
- $F_1[\sigma] \to r_1$
- $F_n[\sigma] \to r_n$
- $C^{back}[\sigma] \to C^{back}_{j+1}[\sigma]T[\sigma]$
- $C^{back}_{j+1}[\sigma] \to C^{back}_{j+2}[r^{-1}_{j+1}\sigma]$
- ...
- $C^{back}_{n}[\sigma] \to C^{back}_{n+1}[r^{-1}_{n}\sigma]$
- $C^{back}_{n+1}[\sigma] \to C[\sigma]T[\sigma]$
- $T[\sigma] \to \epsilon$

Here, we have a total of $(\text{Number relations} + 4 * n + 7 - j)$ rules.

2.7.4 Total Number of Rules

Let $f_1...f_k$ be k linear functions with respectively $n_1, ..., n_k$ relations. Let R be the set of all relations (it can be reduced to the set of relations used by the f_is as only them can be pushed on the stack). For $i \in [0; k-1]$,

- f_i has n_i left rules, so a total after reducing of $\sum_{j=1}^{n} [4 * n_i - j + 1] = \mathcal{O}(n_i^2)$
- f_i has $n_i - 1$ right rules so a total after reducing of $\mathcal{O}(n_i * |R| - |R| + n_i^2)$
- f_i has n_i^2 middle rules so a total after reducing of $\mathcal{O}(n_i^2 * |R| + n_i^3)$

At the end, if we sum for all functions, and knowing that left and right rules are included in middle rules, and we have $n = n_1 = ... = n_k$, we have $\mathcal{O}(k * (n^3 + n^2 * |R|))$ relations. If $|R| = \mathcal{O}(n)$ then we have $\mathcal{O}(k * n^3)$ relations.

2.8 Problem

Given a set of linear functions F, is it possible to know whether there exists a smart plan which only uses functions in F.
(give further explanations, examples...)

Hypothesis 1. *In what follows, we assume that if we can call a linear function $f = r_1...r_n$ then we can also call the functions $f_i = r_1...r_i$ for $i \in [1; n]$.*

2.9 Are Middle Rules Required When We Have Intermediate Function

Let's show a counter-example where middle rules are required. We consider we have the functions:

- $f_1 = c \quad ba$
- $f_2 = d^{-1} \quad b^{-1} \quad e^{-1}$
- $f_3 = e \quad c^{-1}$
- $f_4 = d$

So, we also have the subfunctions:

- $f_2' = d \quad b^{-1}$
- $f_1' = c \quad b$

- Others which are not useful here

Now, let's suppose we want to get a. Without middle rules, we would get:

$$
\begin{aligned}
C[a] &\rightarrow^{f_1} C[b^{-1}]\, c^{-1} \\
&\rightarrow^{f_2'} C[d]\, d^{-1}\, b^{-1}\, C[c^{-1}]\, c\, b\, a \\
&\rightarrow^{f_4} d\, d^{-1}\, b^{-1}\, C[c^{-1}]\, c\, b\, a \\
&\rightarrow^{f_3} d\, d^{-1}\, b^{-1}\, C[e^{-1}]\, e\, c^{-1}\, c\, b\, a \\
&\rightarrow^{f_2} d\, d^{-1}\, b^{-1}\, C[b\, d]\, d^{-1}\, b^{-1}\, e^{-1}\, e\, c^{-1}\, c\, b\, a \\
&\rightarrow^{f_2} d\, d^{-1}\, b^{-1}\, C[c^{-1}]\, c\, b\, C[d]\, d^{-1}\, b^{-1}\, e^{-1}\, e\, c^{-1}\, c\, b\, a
\end{aligned} \tag{2}
$$

We see that a loop appear as we need $C[c^{-1}]$ to compute $C[c^{-1}]$ so middle functions are required.

2.10 Algorithm

Algorithm 2: Algorithm Smart Plan From a Set of Functions

Data: A set of functions F and a linear query q
Result: whether there exists a smart plan composed only of calls to functions in F

1 Replace all functions f in F by a their subfunctions as describe in hypothesis 1;
2 Create an indexed grammar as follows:
- $S \rightarrow C[q]$
- $C[] \rightarrow \epsilon$
- For all functions $f \in F$, derive the middle rules in their reduced form as describes in 2.7.3 (the C is common to all rules).

return *Whether $L(G) = \emptyset$ or not using the algorithm described in [1]*

Proof. Correctness: Let's show that only smart plans are generated. To do so, we can transform left and right rules into way-back list rules as shown in definition 8. We will prove the case of left rules.
Let n be an integer, $i \in [1, n]$ and let's consider the left rule $C[r_1...r_i\ sigma] \rightarrow r_1...r_n C[r_n^{-1}...r_{i+1}^{-1}\sigma]$. The rule can be transformed into:

- $C[r_1\sigma] \rightarrow r_1 C_2[\sigma]$
- ...
- $C_i[r_i\sigma] \rightarrow r_i C_{i+1}[\sigma]$
- $C_{i+1}[\sigma] \rightarrow r_{i+1} C_{i+1}[r_{i+1}^{-1}\sigma]$
- ...
- $C_n[\sigma] \rightarrow r_n C[r_n^{-1}\sigma]$

Let n be an integer, $i \in [1, n]$ and let's consider the right rule $C[r_i...r_n\ sigma] \rightarrow C[r_{i-1}^{-1}...r_1^{-1}]r_1...r_n C[\sigma]$. The rule can be transformed into:

- $C[\sigma] \rightarrow C_1[r_{i-1}^{-1}]r_{i-1}C_1'[\sigma]$
- $C_1[\sigma] \rightarrow C_2[\sigma r_{i-2}^{-1}]r_{i-2}$
- ...

- $C_{i-1}[\sigma] \to C[\sigma r_1^{-1}]r_1$
- $C_1[r_i\sigma] \to r_i C_2'[\sigma]$
- ...
- $C_{n-i+1}[\sigma] \to r_n C[\sigma]$

We can prove the same result for middle rules by using the same ideas used for left and right rules.
As middle rules can be written as rules which are rules extracted from definition 8, we conclude that we will generate plans $P = l$ where l is a way-back list and if q is composed of no relation.

Lemma 2. *$C[r_1...r_n]$ will generate lists of the form $l_0 r_1 l_1 ... r_n l_n$ where l_0, ..., l_n are way-back lists.*

Proof. Let's say we modify the starting symbol to $S \to r_1^{-1}...r_n^{-1}C[r_1...r_n]$. This way, we have a way-back list rule. So, we will generate way-back lists beginning by $r_1^{-1}...r_n^{-1}$, it means that it generates words of the form $r_1^{-1}...r_n^{-1}l_0 r_1 l_1 ... r_n l_n$ and so $C[r_1...r_n]$ generates lists of the form $l_0 r_1 l_1 ... r_n l_n$. □

So by calling $C[q_1...q_n]$, we have results of the form $P = l_0 q_1 l_1 ... q_n l_n$ and by using hypothesis 1, we have access to all the required q_i. As shown in theorem 3, if we have a fully categorized knowledge base, P is a smart plan.

Completeness: The grammar can be seen as an exhaustive search: at each production rule call, we try to call all functions. These functions can either consume relations which were required and give additional ones or asked in the future for given relations to be able to be called. Notice that function can be called even if nothing is required with $C[\sigma] \to r_1...r_n C[r_n^{-1}...r_1^{-1}\sigma]$ and $C[\sigma] \to C[r_n^{-1}...r_1^{-1}]r_1...r_n C[\sigma]$ in order to be exhaustive. □

2.11 Reducing Constraints

As we saw before, a fully categorized knowledge base is required to be able to apply the results. However, when using functions calls, not all constraints are needed as some relations are never used. So, a fully categorized knowledge base under the given set of functions can be used.
One can also notice that relations cannot be called in any order. This order is determined by the set of functions. So, it is possible to extract axioms from a set of functions which need to be true in the knowledge base.
Let $F = f_1...f_n$ a set of linear functions. Let $f \in F$ be a linear function with $f = a_1...a_n$, $n > 1$. We can deduce the following axioms:

- For all $i \in [1, n-1]$, $a_i \Leftrightarrow a_{i+1}$ is true for all nodes or is false for all nodes (i.e. $a_i \Leftrightarrow \neg a_{i+1}$ is true for all nodes).
- Let $f_1 = a_1...a_n$ and $f_2 = b_1...b_m$ be two functions from F ($n \geq 1$ and $m \geq 1$). We have: $a_n \Leftrightarrow b_1$ is true for all nodes or is false for all nodes (i.e $a_n \Leftrightarrow \neg b_1$ is true for all nodes).

These axioms can be reduced even more, without having to assume the transition between functions.

Hypothesis 2. *Given a set of function F and a knowledge base K, we suppose we have for all $f \in F$, $f = a_1...a_n$, $n > 1$:*
For all $i \in [1, n-1]$, $a_i^{-1} \Leftrightarrow a_{i+1}$ is true for all nodes in K or is false for all nodes in K (i.e. $a_i^{-1} \Leftrightarrow \neg a_{i+1}$ is true for all nodes in K).

Theorem 4. *With hypothesis 2, the algorithm 2 gives whether there exists a smart plan or not (but the grammar used in algorithm 2 might generate non-smart plans).*

Proof.

Definition 10 (Function Loop)**.** *Let $\mathcal{F}$ be a set of linear functions and $l = f_1...f_n$ ($f_1, ..., f_n$ are functions in $\mathcal{F}$) be a list of relations built with $\mathcal{F}$. We call a function loop a sequence of function from l, $f_i...f_j$ ($i < j$) such that $f_i...f_j$ is a way-back list.*

Lemma 3. *If the grammar used in algorithm 2 generates a non-empty answer, there exists at least one solution without a function loop.*

Proof. Let lq be the shortest (in term of number of relations) non-empty word of the grammar in the algorithm 2 for a query q (of length 1 to simplify). We saw that l is a way-back list. l can be written $l = f_1...f_n a_1...a_k$ ($a_1, ..., a_k$ are the first relations of the last function $f_{n+1} = a_1...a_k q$). l does not strictly contain function loops, meaning that there exists no $i, j \in [1, n], i < j$ such that $f_i...f_j$ is a way-back list. Otherwise, the shortest word would be $l = f_1...f_{i-1} f_{j+1}...f_n$. □

Lemma 4. *Let K be a knowledge base, $\mathcal{F}$ a set of linear functions and $l = f_1...f_n$ ($f_1, ..., f_n$ are functions in $\mathcal{F}$) be a way-back list built with $\mathcal{F}$ without a function loop different from l. Let r be a relation from l such that $l = l_0 r l_1 r^{-1} l_2$. Let $\mathcal{N}_1$ be the set of nodes obtained by calling l_0 on K and $\mathcal{N}_1^r$ the set of nodes in $\mathcal{N}_1$ which have a r relation. Let $\mathcal{N}_2$ be the set of nodes optained after calling $l_0 r l_1 r^{-1}$. Then, under hypothesis 2, we have $\mathcal{N}_1^r \subseteq \mathcal{N}_2$ or $\mathcal{N}_2 = \emptyset$.*

Proof. Let's prove the lemma by induction on the number n of relations between r and r^{-1}.
For $n = 0$, we have rr^{-1} and the result is obvious.

Let $n > 0$. We consider that the lemma is true for all $k < n$. We write $l = l_0 r r_1 l_1 r_1^{-1} r_2 l_2 r_2^{-1} ... r_k l_k r_k^{-1} r^{-1} l_{k+1}$ ($l_0, ..., l_k$ are way-back lists by construction) where we have n relations between r and r^{-1}. We call $\mathcal{N}^i$ the set of nodes obtained before calling r_i and $\mathcal{N}^{k+1}$ the set of nodes obtained before calling r^{-1}.

Let's prove that $\mathcal{N}^{1,r_1} \subseteq \mathcal{N}^{k+1,r^{-1}}$ ($\mathcal{N}^{1,r_1}$ is the subset of $\mathcal{N}^1$ with the relation r_1 and $\mathcal{N}^{k+1,r^{-1}}$ the subset of $\mathcal{N}^{k+1}$ with the relation r^{-1}).

If a function stops on r, no function (used in l) can stop on r_1^{-1} nor on any r_i^{-1} (otherwise there would be a loop as $r_1 l_1 r_1^{-1}$ and all the $r_1 l_1 r_1^{-1} ... r_i l_i r_i^{-1}$ are way-back lists). With the induction hypothesis, we have $\mathcal{N}^{i,r_{i+1}} \subseteq \mathcal{N}^{i+1}$ for all $i > 1$. As we do not stop as described before, we have $\mathcal{N}^i = \mathcal{N}^{i,r_{i+1}}$. So, we have $\mathcal{N}^{1,r_1} \subseteq \mathcal{N}^{k+1,r^{-1}}$.

Now we suppose we do not stop on r but a function stops on r_j^{-1} with $1 < j < k$. For the same reasons than before, no function can stop on any r_i^{-1} $(i \neq j)$. It means that, for $i \neq j$, we have at least one function which contains $r_i^{-1}r_{i+1}$ and $r_k^{-1}r^{-1}$. So, from hypothesis 2, we have for $i \neq j$, $r_i \Leftrightarrow r_{i+1}$, $r_k \Leftrightarrow r^{-1}$ and $r^{-1} \Leftrightarrow r_1$ (no stop on r_1). If one of the axioms is that $a_i^{-1} \Leftrightarrow \neg a_{i+1}$ is true for all nodes in K, then we have no result at all, i.e. $\mathcal{N}_2 = \emptyset$. We consider it is not the case. By transitivity, we also have $r_j \Leftrightarrow r_{j+1}$ and we have the transition $r_j^{-1}r_{j+1}$ for all nodes. Using the same arguments than before, $\mathcal{N}^{1,r_1} \subseteq \mathcal{N}^{k+1,r^{-1}}$.

Then, we can conclude that $\mathcal{N}_1^r \subseteq \mathcal{N}_2$. □

Corrollary 2. *Let K be a knowledge base, $\mathcal{F}$ a set of linear functions and $l = f_1...f_n$ ($f_1, ..., f_n$ are functions in $\mathcal{F}$) be a way-back list built with $\mathcal{F}$ without a function loop different from l. Then, either l gives no result on K or the input is also among the outputs of l.*

We know that if there are words in the grammar in algorithm 2, then there are words without function loop in them and we proved with corrollary 2 that a word without a loop is a smart plan. So there is at least one smart plan amoung the words. In addition, as all smart plans are way-back lists, all of them are in the grammar. So, the algorithm 2 still gives us whether there exist smart plans or not.

□

3 Reducing the Grammar

3.1 Remove Forced Loops

As we proved in theorem 4, one could remove all the loops and still obtain the same result after algorithm 2. Then, would it be possible to remove rules from the grammar that add nothing but function loops? By doing so, we hope to reduce the computation time of algorithm 4 and of the exploration to find a plan. However, we might lose the completeness of the grammar but it does not change the result of the algorithm 2.

Theorem 5. *By removing the rules:*

- $C[\sigma] \rightarrow r_1...r_nC[r_n^{-1}...r_1^{-1}\sigma]$
- $C[\sigma] \rightarrow C[r_n^{-1}...r_1^{-1}]r_1...r_nC[\sigma]$

only plans with function loops are removed.

Proof. Let consider the rule $C[\sigma] \rightarrow r_1...r_nC[r_n^{-1}...r_1^{-1}\sigma]$. This rule means that a function is called without using anything on the stack. Using this rule may generate a loop between r_1 and b, where b was the symbol at the top of the stack when the rule was called. Let suppose it is not the case. It means that a function will be called such that ab (where a is a relation) is a part of that

function. Let's call this function $f = x_1...x_k a b y_1..y_l$.
Instead of calling $C[\sigma] \to r_1...r_n C[r_n^{-1}...r_1^{-1}\sigma]$, we call the rule associated with f: $C[by_1...y_k]$. Using lemma 2 and completeness, we know we get all possible words of the for $r_1...r_n l_0 r_1 l_1...r_n l_n b y_1 l_1^2...y_k l_k^2$ which are makable and in particular, what should have appeared with the rule we have just erased. If we still obtain a rule of the form $C[\sigma] \to r_1...r_n C[r_n^{-1}...r_1^{-1}\sigma]$, we redo the same operation and it will end as f is finite.
Then, all rules of the form $C[\sigma] \to r_1...r_n C[r_n^{-1}...r_1^{-1}\sigma]$ which do not create loops can be erased by deleting first the outer ones so the procedure ends.
For the rule $C[\sigma] \to C[r_n^{-1}...r_1^{-1}]r_1...r_n C[\sigma]$, only results with function loops are created as a function ends on r_n, $C[r_n^{-1}...r_1^{-1}]$ generate lists of the form $r_n^{-1}l_n...r_1^{-1}l_1$ (l_1 ... l_n are way-back lists) and $r_n^{-1}l_n...r_1^{-1}l_1 r_1...r_n$ is a way-back list.
We conclude only plans with function loops are removed. □

3.2 Introduce End Symbol

By introducing an end symbole, that we will call *end*, it is possible to reduce the number of rules used in the reduced form for right rules.

3.2.1 Initialization Rules

As we introduce an end symbol, we have to modify the initialization rules. The new ones are, for a query $q = q_1...q_n$:

- $S[\sigma] \to C_1^{init}[end\,\sigma]$
- $C_1^{init}[\sigma] \to C_2^{init}[q_1\,\sigma]$
- ...
- $C_n^{init}[\sigma] \to C[q_n\,\sigma]$
- $C[end \quad \sigma] \to \epsilon$

Here, we simply add the end symbol before everything else to know where to stop and the say that when an end symbol is met, then it is the end.

3.2.2 Reduced Right Rules

For the right rules, we replace the first rules presented in 2.7.2:

- $C[c_i\sigma] \to B[\sigma]$
- $B[\sigma] \to A_{-1}^{back}[\sigma]D[\sigma]$
- $A_{-1}^{back}[\sigma] \to A_0^{back}[end \quad \sigma]$
- $A_0^{back}[\sigma] \to A_1^{back}[r_1^{-1}\sigma]$
- ...

Here, by adding the end symbol on the stack, we obtain the same result as emptying the stack.

3.2.3 Number of Rules

Now, we always have $\mathcal{O}(k * n^3)$ rules.

4 Emptyness Algorithm

4.1 Introduction

The algorithm for testing the emptyness of an indexed grammar presented in [1] has a disadvantage: a set of rules is generated before actually applying the algorithm. The size of this set is exponential. So, at the end the algorithm has an exponential complexity even in the best case. For computational purposes, it is great to lower this bound, even if the upper bound is still exponential.

We propose here to generate the rules on the fly so we do not generate useless rules. To apply the algorithm, as in [1], we need rules in their reduced form. More precisely, they should be either a production, a consumption, a duplication or an end rule.

Definition 11 (Production Rule)**.** *We call "production rules" rules of the form:*

$A[\sigma] \rightarrow B[f\sigma]$

where A and B are non-terminals, f is a production symbol (here production symbols are the terminals) and σ is the stack.

Definition 12 (Consumption Rule)**.** *We call "consumption rules" rules of the form:*

$A[f\sigma] \rightarrow B[\sigma]$

where A and B are non-terminals, f is a production symbol (here production symbols are the terminals) and σ is the stack.

In what follows, we call $Cons(f)$ the set of all the consumption rules which use f as a production symbol.

Definition 13 (Duplication Rule)**.** *We call "duplication rules" rules of the form:*

$A[\sigma] \rightarrow B[\sigma]C[\sigma]$

where A, B and C are non-terminals and σ is the stack.

Definition 14 (End Rule)**.** *We call "end rules" rules of the form:*

$A[\sigma] \rightarrow a$

where A is a non-terminals, a is a terminal and σ is the stack.

4.2 Initialization

Instead of writting all the rules presented in [1], we keep track of all marked sets. We initialize the algorithm as follows:

1. $marked \leftarrow dictionary()$, $marked$ gives for all non-terminal the sets which are marked.
2. For all non-terminals A, $marked[A] = List()$
3. For all non-terminals A, $marked[A].append(set(A))$
4. For all end rules $A[\sigma] \rightarrow a$, $marked[A].append(set())$

The idea behind marked the symbols is presented in [1]: N is a set from $marked[A]$ at the end of the algorithm if and only if there exists some ω in N^* such that ω can be derived from A with our indexed grammar. So, the grammar is not empty if and only if the empty set is marked for S, i.e. an end symbol can be reached from S.

The algorithm will loop on the rules until no more new sets are marked. During the loop, we process duplication and production rules differently.

4.3 Duplication Rule Processing

For the duplication rule $A[\sigma] \to B[\sigma]C[\sigma]$, we mark for A all the $N_B \cup N_C$ where N_B is marked for B and N_C is marked for C.

4.4 Production Rule Processing

For the production rule $A[\sigma] \to B[f\sigma]$:

1. If there exists a rule of the form $B[f\sigma] \to C[\sigma]$ (where C is a non-terminal) in $Cons(f)$ then:
   (a) For A, mark all the N_B where N_B is marked for B.
   (b) If the empty set is marked for B, for all rules $D[f\sigma] \to E[\sigma]$, mark for A all the N_E where N_E is a set marked for E.
2. For all marked sets for B $N_B = \{C_1, C_2, ..., C_r\}$, for all combinations of rules from $Cons(f)$ $C_1[f\sigma] \to D_1[\sigma]$, ..., $C_r[f\sigma] \to D_r[\sigma]$ (we need exactly only rule for each C_i), mark for A $N = \cup_{i=1}^{r} N_{D_i}$ for all N_{D_i} marked for D_i $(1 \leq i \leq r)$

4.5 Final Algorithm

Algorithm 3: Algorithm Emptyness Indexed Grammar

```
Data: a set of rules in reduced form (4.1)
Result: whether the grammar is empty or not
Initialize the algorithm (4.2);
while new sets are marked do
    for each rule do
        if the rule is a duplication rule then
            do the processing for duplication rule (4.3);
        end
        else if the rule is a production rule then
            do the processing for production rule (4.4);
        end
    end
end
return the grammar is not empty if and only if the empty set is marked
  for S
```

4.6 Optimization Over Rules Order

In this algorithm, the order of the rules may be important. In particular, with the reduced forms presented in 2.7.3, the algorithm is faster if the rules are processed in the opposite order they are presented in this paper.

In particular, if we build a graph of dependency of non-terminal symbols in the grammar, i.e. a graph in which the directed edges $A \to B$ mean that the modifications on A depends on modifications on B, then it is better to start with rules which depend on nothing (here end rules). Then recusively we choose non-terminals for which the non-terminals it depends on had been processed (if possible).

On our examples, it was proven to be appoximately 10 times faster to reverse the order of the reduced rules.

5 Introduce Equivalence Rules

CAREFUL : IT DOES NOT WORK (counter example : f = cc, query is c q and we have $c \Leftrightarrow q$)

Often, in knowledge bases, different paths can have the same or the meaning. For example, the child relation has the same meaning that the opposite of the parent relation. However, some functions may consider only one path and ignore all others which are equivalent. So, the rules must be introduced in the grammar.

Definition 15 (Equivalence Rule)**.** *Let $r_1, ..., r_N$ be N binary relations, $X_1, ..., X_{N+1}$ be $N + 1$ variables, $l_1, ..., l_K$ be K binary relations and $Y_1, ..., Y_{K+1}$ be $K + 1$ variables. A equivalence rule is a rule of the form:*
$r_1(X_1, X_2) \wedge ... \wedge r_N(X_N, X_{N+1}) \Leftrightarrow l_1(Y_1, Y_2) \wedge ... \wedge l_K(Y_K, Y_{K+1})$
where $X_1 = Y_1$ and $X_{N+1} = Y_{K+1}$.

Notation 2. *We write $r_1...r_N \Leftrightarrow l_1...l_K$ instead of $r_1(X_1, X_2) \wedge ... \wedge r_N(X_N, X_{N+1}) \Leftrightarrow l_1(Y_1, Y_2) \wedge ... \wedge l_K(Y_K, Y_{K+1})$ where $X_1 = Y_1$ and $X_{N+1} = Y_{K+1}$.*

Hypothesis 3. *If we have the rule of $r_1...r_N \Leftrightarrow l_1...l_K$, then the existence of the path $r_1...r_N$ is equivalent to the existence of the path $l_1...l_K$.*

Property 3. *Let's suppose we have a rule $r_1...r_N \Leftrightarrow l_1...l_K$. We introduce the four rules:*

- $C[r_1...r_N\sigma] \to C[l_1...l_K\sigma]$
- $C[l_1...l_K\sigma] \to C[r_1...r_N\sigma]$
- $C[r_N^{-1}...r_1^{-1}\sigma] \to C[l_K^{-1}...l_1^{-1}\sigma]$
- $C[l_K^{-1}...l_1^{-1}\sigma] \to C[r_N^{-1}...r_1^{-1}\sigma]$

in the grammar from the algorithm 2. Then, under hypothesis 3 and 2, the grammar produces words of the form lq where q is the query and where if l is called on a knowledge base, its input is also among outputs.

Proof. TODO □

6 Updating the query

Once the algorithm 3 returns a result, a set of symbols will be marked. It means that, if the marked symbols are remembered and the algorithm runs again, then the output will be immediate. Now, let's consider 2. Is it possible to update the query by keeping as much marked symbols as possible?

What we need to do is to replace the rules using the C^{init}s by the new rules derived from the new query. Then, the marked symbols for the C^{init}s and S must be reset. Then, no other modification is required.

7 Multiple Inputs Functions

7.1 Tree-Like Function

7.2 Linear Multiple Inputs Functions

References

[1] Alfred V. Aho. Indexed grammars : An extension of context-free grammars. *J. ACM*, 15(4):647–671, October 1968.