

Efficient Decision Procedures for Context-Free Languages

Martin Fränzle

Informatics and Mathematical Modelling

The Technical University of Denmark

What you'll learn

1. Efficient algorithms for deciding

- whether a string belongs to a CFL described by a CNF grammar,
- whether a CFG defines the empty language.

Deciding language membership

The “CYK” algorithm

The membership problem

- **Problem:** Given a string $w \in T^*$ and a (CNF-style) CFG G , decide whether $w \in L(G)$.

The membership problem

- **Problem:** Given a string $w \in T^*$ and a (CNF-style) CFG G , decide whether $w \in L(G)$.
 - **Naive solution:**
 - As G is of CNF, the parse trees of G are binary trees, Hence, the parse tree for w — if existent — has exactly $2|w| - 1$ nodes labelled with variables.
- ⇒ Enumerate all parse trees of that size.

The membership problem

- **Problem:** Given a string $w \in T^*$ and a (CNF-style) CFG G , decide whether $w \in L(G)$.
 - **Naive solution:**
 - As G is of CNF, the parse trees of G are binary trees, Hence, the parse tree for w — if existent — has exactly $2|w| - 1$ nodes labelled with variables.
- ⇒ Enumerate all parse trees of that size.

Exponential in $|w|$!

Recursive split-and-match

Idea: For a CNF grammar $G = (V, T, P, S)$ and $X \in V$, membership $w \in L_X(G)$ can be recursively determined by

- if $|w| \geq 2$ then
 1. split w into $uv = w$ with $u \neq \varepsilon \neq v$
 2. find a production $X \rightarrow YZ$
 3. check $u \in L_Y(G)$ and $v \in L_Z(G)$.
- if $|w| = 1$ then
 - find a production $X \rightarrow w$.

Recursive split-and-match

Idea: For a CNF grammar $G = (V, T, P, S)$ and $X \in V$, membership $w \in L_X(G)$ can be recursively determined by

- if $|w| \geq 2$ then
 1. split w into $uv = w$ with $u \neq \varepsilon \neq v$ **nondeterministically**,
 2. find a production $X \rightarrow YZ$ **nondeterministically**,
 3. check $u \in L_Y(G)$ and $v \in L_Z(G)$.
- if $|w| = 1$ then
 - find a production $X \rightarrow w$.

Resolving the nondeterminism

1. Instead of nondeterministically selecting a variable and trying whether it matches, do always calculate the set of all variables that match:

For $w \in T^*$ and $i \leq j \in \{1, \dots, |w|\}$ we define

$$\text{Var}(w, i, j) = \{X \mid X \Rightarrow_G^* w_i, \dots, w_j\}$$

and calculate $\text{Var}(w, i, j)$ recursively

2. Check whether $S \in \text{Var}(w, 1, |w|)$.

Resolving the nondeterminism

1. Instead of nondeterministically selecting a variable and trying whether it matches, do always calculate the set of all variables that match:

For $w \in T^*$ and $i \leq j \in \{1, \dots, |w|\}$ we define

$$\text{Var}(w, i, j) = \{X \mid X \Rightarrow_G^* w_i, \dots, w_j\}$$

and calculate $\text{Var}(w, i, j)$ recursively by

$$\text{Var}(w, i, j) = \begin{cases} \{X \mid (X \rightarrow w_i) \in P\} & \text{if } i = j \\ \left\{ X \mid \begin{array}{l} (X \rightarrow YZ) \in P, \\ Y \in \text{Var}(w, i, m), \\ Z \in \text{Var}(w, m+1, j), \\ i \leq m < j \end{array} \right\} & \text{if } i < j \end{cases}$$

2. Check whether $S \in \text{Var}(w, 1, |w|)$.

Dynamic programming / Memoization

obs.: When deciding $w \in L(G)$ with above algorithm, various $\text{Var}(w, i, j)$ are computed more than once.

Dynamic programming / Memoization

Obs.: When deciding $w \in L(G)$ with above algorithm, various $\text{Var}(w, i, j)$ are computed more than once.

Idea: Whenever we calculate a certain instance $\text{Var}(w, i, j)$, we store it in a table for future reference. Should we ever need it again, we perform a table-lookup instead of a re-computation.

The technique of caching solutions rather than recomputing them is called memoization or dynamic programming.

Dynamic programming / Memoization

Obs.: When deciding $w \in L(G)$ with above algorithm, various $\text{Var}(w, i, j)$ are computed more than once.

Idea: Whenever we calculate a certain instance $\text{Var}(w, i, j)$, we store it in a table for future reference. Should we ever need it again, we perform a table-lookup instead of a re-computation.

- Use a table T indexed by $i, j \in \{1, \dots, |w|\}$,
- table elements are from $\mathcal{P}(V) \uplus \{\text{unknown}\}$, initialised w. unknown
- $\text{Var}(w, i, j) =$

```
if      T(i, j)  $\neq$  unknown
then  T(i, j)
else  let  x = ... /* recursive computation */;
        in  T(i, j) := x;
        x
```

The technique of caching solutions rather than recomputing them is called **memoization** or **dynamic programming**.

Analysis: cost in terms of $|w|$

1. Given $i \in \{1, \dots, |w|\}$, the price of calculating $\{X \mid (X \rightarrow w_i) \in P\}$ can be bounded by a constant which is independent from w .
2. If $\text{Var}(w, i, m)$ and $\text{Var}(w, m+1, j)$ are known for all $m \in \{i, \dots, j-1\}$, then the price of calculating

$$\left\{ X \mid \begin{array}{l} (X \rightarrow YZ) \in P, \\ Y \in \text{Var}(w, i, m), \\ Z \in \text{Var}(w, m+1, j), \\ i \leq m < j \end{array} \right\}$$

is proportional to $j - i$. As $j - i < |w|$, this price is $O(|w|)$.

3. There are
 - $|w|$ calculations of type (1.), i.e. of price $O(1)$ each,
 - $\frac{1}{2}|w|(|w| - 1)$ calculations of type (2.), i.e. of price $O(|w|)$ each.

$\Rightarrow$ the overall price is $O(|w| + \frac{1}{2}|w|(|w| - 1)|w|) = O(|w|^3)$.

This $O(|w|^3)$ algorithm is generally known as the “CYK algorithm” after its three independent developers

- *J. Cocke*, who circulated it privately,
- *T. Kasami*, who published it in an internal Air-Force memorandum,
- *D. H. Younger*, who published it in *Information and Control* 10:2 (1967)

Remark

- Memoization is a simple and often very efficient technique for enhancing the performance of programs!
- In some *functional* programming environments, memoization requires no coding effort from the programmer: it can be turned on by a keyword!
- It is one of the big advantages of (pure) functional programming that memoization is completely transparent — i.e., doesn't change the behaviour of the program (except for performance).
A “memoization” keyword in an imperative programming language would in general drastically alter the behaviour of the program: table-lookup yields no side-effects, while function calls may do.

Deciding language emptiness

The emptiness problem

- **Problem:** Given a CFG $G = (V, P, T, S)$, decide whether $L(G) = \emptyset$.

The emptiness problem

- **Problem:** Given a CFG $G = (V, P, T, S)$, decide whether $L(G) = \emptyset$.

- **Naive solution:** Compute

Base: $Gen_0 := T,$

Recursion: Iterate

$$Gen_{i+1} := \begin{cases} Gen_i \cup \{A\} & \text{if there is } (A \rightarrow \alpha) \in P \text{ with } A \notin Gen_i \\ & \text{and every symbol of } \alpha \text{ being in } Gen_i \\ Gen_i & \text{otherwise} \end{cases}$$

until $Gen_{i+1} = Gen_i$

and check whether $S \in Gen_i$.

The emptiness problem

- **Problem:** Given a CFG $G = (V, P, T, S)$, decide whether $L(G) = \emptyset$.

- **Naive solution:** Compute

Base: $Gen_0 := T$,

Recursion: Iterate

$$Gen_{i+1} := \begin{cases} Gen_i \cup \{A\} & \text{if there is } (A \rightarrow \alpha) \in P \text{ with } A \notin Gen_i \\ & \text{and every symbol of } \alpha \text{ being in } Gen_i \\ Gen_i & \text{otherwise} \end{cases}$$

until $Gen_{i+1} = Gen_i$

and check whether $S \in Gen_i$.

Algorithm is $O(|G|^2)$!

Analysis of naive solution

- Each iteration of

$$Gen_{i+1} := \begin{cases} Gen_i \cup \{A\} & \text{if there is } (A \rightarrow \alpha) \in P \text{ with } A \notin Gen_i \\ & \text{and every symbol of } \alpha \text{ being in } Gen_i \\ Gen_i & \text{otherwise} \end{cases}$$

requires a full sweep through all productions.

⇒ Each iteration is $O(\sum_{X \rightarrow \alpha \in P} (|\alpha| + 1)) = O(|G|)$.

- Need up to $\underbrace{|V|}_{O(|G|)}$ iterations until stabilization.

⇒ Algorithm is $O(|G|^2)$!

Better algorithm

1. Avoid visiting all productions when looking for implications of detecting a generating variable.

By building a table $T_1 : V \rightarrow \mathcal{P}(P)$ that assigns to each variable V the set of those productions $\in P$ that contain V in the body, we can visit exactly those productions that are affected by detecting V to be generating, namely $T_1(V)$.

Better algorithm

1. Avoid visiting all productions when looking for implications of detecting a generating variable.

By building a table $T_1 : V \rightarrow \mathcal{P}(P)$ that assigns to each variable V the set of those productions $\in P$ that contain V in the body, we can visit exactly those productions that are affected by detecting V to be generating, namely $T_1(V)$.

T_1 can be built on-the-fly while reading in the grammar G ; i.e. building it is $O(|G|)$.

Better algorithm (contd.)

2. Provide $O(1)$ check whether a certain production has body containing only known-to-be-generating symbols.

By building a table $T_2 : P \rightarrow \mathbb{N}$ that assigns to production the number of currently-not-known-to-be-generating variables in its body, we can provide an $O(1)$ test, namely $T_2(\text{production}) = 0$.

By furthermore building a table $T_3 : P \times V \rightarrow \mathbb{N}$ that assigns to each pair of production and variable the number of occurrences of that variable in the production's body, we can provide an $O(1)$ update rule for the previous table, namely
 $T'_2(\text{production}) = T_2(\text{production}) - T_3(\text{production}, V)$ when V is detected to be generating.

Better algorithm (contd.)

2. Provide $O(1)$ check whether a certain production has body containing only known-to-be-generating symbols.

By building a table $T_2 : P \rightarrow \mathbb{N}$ that assigns to production the number of currently-not-known-to-be-generating variables in its body, we can provide an $O(1)$ test, namely $T_2(\text{production}) = 0$.

By furthermore building a table $T_3 : P \times V \rightarrow \mathbb{N}$ that assigns to each pair of production and variable the number of occurrences of that variable in the production's body, we can provide an $O(1)$ update rule for the previous table, namely $T'_2(\text{production}) = T_2(\text{production}) - T_3(\text{production}, V)$ when V is detected to be generating.

T_2 and T_3 can be built on-the-fly while reading in the grammar G ; i.e. building them is $O(|G|)$.

Better algorithm (contd.)

3. Use a queue for memorizing the known-to-be-generating variables which have yet unexplored occurrences.
4. When popping a known-to-be-generating variable V from the queue of to-be-investigated variables, do the following:

```
    dequeue  $V$ ;  
    for each  $(X \rightarrow \alpha) \in T_1(V)$  do  
         $T'_2(X \rightarrow \alpha) := T_2(X \rightarrow \alpha) - T_3(X \rightarrow \alpha, V)$ ;  
        if  $T'_2(X \rightarrow \alpha) = 0$  /*  $X$  is generating */  
        then  $Gen' := Gen \cup \{X\}$ ;  
            enqueue  $X$   
        fi  
    od
```

Better algorithm (contd.)

3. Use a queue for memorizing the known-to-be-generating variables which have yet unexplored occurrences.
4. When popping a known-to-be-generating variable V from the queue of to-be-investigated variables, do the following:

```
    dequeue  $V$ ;  
    for each  $(X \rightarrow \alpha) \in T_1(V)$  do  
         $T'_2(X \rightarrow \alpha) := T_2(X \rightarrow \alpha) - T_3(X \rightarrow \alpha, V)$ ;  
        if  $T'_2(X \rightarrow \alpha) = 0$  /*  $X$  is generating */  
        then  $Gen' := Gen \cup \{X\}$ ;  
            enqueue  $X$   
        fi  
    od
```

Algorithm is $O(|G|)$!