Games, walks and grammars: Problems I've worked on
Vervoort, M.R.

Citation for published version (APA):

General rights
It is not permitted to download or to forward/distribute the text or part of it without the consent of the author(s) and/or copyright holder(s), other than for strictly personal, individual use, unless the work is under an open content license (like Creative Commons).

Disclaimer/Complaints regulations
If you believe that digital publication of certain material infringes any of your rights or (privacy) interests, please let the Library know, stating your reasons. In case of a legitimate complaint, the Library will make the material inaccessible and/or remove it from the website. Please Ask the Library: http://uba.uva.nl/en/contact, or a letter to: Library of the University of Amsterdam, Secretariat, Singel 425, 1012 WP Amsterdam, The Netherlands. You will be contacted as soon as possible.
Appendix A

The inner workings of EMILE

At this moment of writing, the EMILE program consists of about 5500 lines of C++ code. However, most of that is code for data type representation, user interface, utility functions, various optimizations, etcetera. The algorithms themselves are fairly simple. Each of the sections of this chapter focuses on a different algorithm used in EMILE. For each algorithm, a synopsis is given, as well as explicit pseudo-code, and a summary of the constants controlling the algorithm that can be set by the user.

A.1 Gathering context/expression pairs.

Synopsis: EMILE maintains a matrix $M$ of the context/expression pairs it has encountered. This routine updates this matrix, given text from some input $I$.

Algorithm

```
sub learn_sentences(I)
  while (there is input to be read) do
    read the input $I$ up to the next end-of-sentence marker:
    set $s$ := the sentence read, converted to a sequence of words:
    if (length(s) ≤ max_sentence_length) then
      for each triple $(c_l, e, c_r)$ with $c_l^{-} e^{-} c_r = s$ do
        insert $(c_l^{-}(.)^{-} c_r, e)$ into $M$:
      end for
    end if
  end while
end sub
```

User definable settings
Appendix A. The inner workings of EMILE

- **max_sentence_length**: sentences longer than this are ignored.

- **end_of_sentence_markers**: a set of characters that EMILE interprets as marking the end of a sentence.

- **allow_multi_line_sentences**: a boolean variable. If this variable is set to false, the end of an input line is considered to indicate the end of the current sentence. If this variable is set to true, a sentence can span multiple lines. In either case, the **end_of_sentence_markers** also indicate the end of the sentence, as does an empty line (i.e. two end-of-lines separated by nothing but whitespace).

- **ignore_abbreviation_periods**: if the period symbol '.' is used as an end-of-sentence-marker, and this boolean variable is set to true, periods following a single letter are not considered to indicate the end of a sentence.

- **regular_expression_as_marker**: if this boolean variable is set to true, the content of the setting **end_of_sentence_markers** is interpreted as a regular expression, using the syntax of Extended Regular Expressions as defined in the *regex*(5) Unix man page. The standard C++ escape sequences are recognized, i.e. '\n' for newline, '\t' for tab etcetera. The settings of **allow_multi_line_sentences** and **ignore_abbreviation_periods** are ignored.

Although this is a more expressive mechanism for indicating end-of-sentence markers, it is not more readable: for instance, the default settings of the normal mode correspond to the regular expression

```
[!.;?]/\n[\r\n\t]*\n[\r\n\t]*\r/\n[^a-zA-Z]\n[^\r\n\t]\n
```

**Notes** When using regular expressions to mark the end of sentences, newlines have to be explicitly included in the regular expression in order to be taken into account. Note that on non-Unix systems, the '\n' symbol refers to the end-of-line marker customary on that system. The carriage return symbol '\r' can be used if there is a need to explicitly take into account line endings of non-Unix files when working on a Unix system.

An end-of-sentence marker is considered to be part of the sentence it is ending.

A sentence is converted into a sequence of words before it is searched for context/expression pairs. A word is a nonempty sequence of alphanumeric characters, or a single non-alphanumeric, non-whitespace symbol. Whitespace characters (spaces, tabs, newlines and carriage returns) function as word separators where necessary and are otherwise ignored.

For reasons of efficiency, contexts and expressions are not directly used as elements of the matrix. Instead, the actual contexts and expressions are stored in a table, and references to the entries in the table are used in the matrix.
There is a compilation option to put Emilé in ‘Morpheme Analysis’ mode. In this mode, each word (using whitespace as a delimiter) is treated as a separate element of \( S \), and is split into single characters for analysis. The settings related to end-of-sentence marking are ignored.

### A.2 Extracting the grammatical types from the matrix

**Synopsis** A type \( T = (T_C, T_E) \) is considered to have sufficient *support* if it satisfies the following three conditions:

1. \( \#(M \cap (T_C \times T_E)) \geq \#(T_C \times T_E) \cdot \text{total-support} \%
2. \( \forall c \in T_C : \#(M \cap ([c] \times T_E)) \geq \#T_E \cdot \text{context-support} \%
3. \( \forall c \in T_E : \#(M \cap (T_C \times [c])) \geq \#T_C \cdot \text{expression-support} \%

The program maintains a set \( G \) of grammatical types with sufficient support, and with contexts and expressions of length at most \( \text{max.primary.context.length} \) or \( \text{max.primary.expression.length} \), respectively. All these types are of maximal size (under the constraint of having sufficient support), and all are at or above a certain minimum size (as indicated by the settings \( \text{min.contexts.per.type} \) and \( \text{min.expressions.per.type} \)).

An element \( (c, \epsilon) \in M \) is considered *covered* by a type \( T \) if \( (c, \epsilon) \in T_C \times T_E \). This routine updates and enlarges \( G \) so that every element \( (c, \epsilon) \in M \) (that can be covered by a type of minimum size) is covered by a type in \( G \).

**Algorithm**

```plaintext
sub expand_grammar(G)
    for each \( T \in G \) do
        call enlarge_grammatical_type(T):
        if \( (\#T_C < \text{min.contexts.per.type}) \)
            or \( (\#T_E < \text{min.expressions.per.type}) \) then
            remove \( T \) from \( G \);
        end if
    end for
    for each \( (c, \epsilon) \in M \) do
        if \( \neg \exists T \in G : (c, \epsilon) \in T_C \times T_E \) then
            set \( T := ([c], \{\epsilon\}) \):
            call enlarge_grammatical_type(T):
            if \( (\#T_C \geq \text{min.contexts.per.type}) \)
                or \( (\#T_E \geq \text{min.expressions.per.type}) \) then
```
Appendix A. The inner workings of EMILE

\[ \begin{align*}
\text{insert } T \text{ into } G: \\
\text{end if} \\
\text{end if} \\
\text{end for} \\
\text{end sub}
\end{align*} \]

\[ \text{sub enlarge_grammatical_type}(T) \]
\[ \text{repeat} \]
\[ \text{set } X := \{ c' \text{ a context | } \{ T_C \cup \{ c' \}, T_E \} \text{ has sufficient support,} \\
\text{length}(c) \leq \text{max}_\text{primary}_\text{context}_\text{length} \} \cup \{ e' \text{ an expression | } \{ T_C, T_E \cup \{ e' \} \} \text{ has sufficient support,} \\
\text{length}(e) \leq \text{max}_\text{primary}_\text{expression}_\text{length} \}; \]
\[ \text{if } (X \neq \emptyset) \text{ then} \]
\[ \text{nondeterministically select } x \text{ from } X: \]
\[ \text{if } (x \text{ is a context}) \text{ then} \]
\[ \text{insert } x \text{ into } T_C: \]
\[ \text{else} \]
\[ \text{insert } x \text{ into } T_E: \]
\[ \text{end if} \]
\[ \text{end if} \]
\[ \text{until } (X = \emptyset); \]
\[ \text{end sub} \]

User definable settings

- \text{max}_\text{primary}_\text{context}_\text{length}, \text{max}_\text{primary}_\text{expression}_\text{length}: in order to ensure that the set of types found will converge if sufficiently many sentences are read, the search space can be limited to primary contexts and expressions of bounded size.

- \text{total}_\text{support%}, \text{context}_\text{support%}, \text{expression}_\text{support%}: these variables control the support required from the matrix for each type. The lower these values, the larger the size of the types found. Note that lowering one value while keeping the other values high will not have much effect.

- \text{min}_\text{contexts}_\text{per}_\text{type}, \text{min}_\text{expressions}_\text{per}_\text{type}: Types with fewer contexts or expressions than indicated by these settings are discarded.

Notes
The selection of \( x \) from \( X \) is not nondeterministic, but based on the amount of support that would be added to the grammatical type. For purposes of constructing a grammar, types of extremely small size usually are not very interesting. Types with less than \text{min}_\text{contexts}_\text{per}_\text{type} contexts or \text{min}_\text{expressions}_\text{per}_\text{type} expressions are discarded. This may cause some elements \((c, e) \in M\) to be uncoverable.
The type '[]' is always set to the type of whole sentences, with '(()())' as a secondary context and all encountered sentences as secondary expressions.

**Complexity** The `enlarge_grammatical_type` subroutine maintains a significant amount of auxiliary data to avoid having to repeat calculations to collect the set X. Initialization of the auxiliary data has an execution time of order \(O(\#\{(c, e) \in M \mid c \in T_C \lor e \in T_E\})\), while each iteration of the `repeat..until` loop has an average-case execution time of order \(O(\#(T_C \cup T_E))\) and a worst-case execution time of order \(O(\#\{(c, e) \in M \mid c \in T_C \lor e \in T_E\})\).

### A.3 Eliminating superfluous types

**Synopsis** It is possible that the contribution of some type to the coverage of \(G\) is made (nearly) superfluous by types found later, i.e. most or all of the context/expression pairs that are covered by that type, are also covered by other types of \(G\). This routine eliminates such types.

**Algorithm**

```plaintext
sub eliminate_superfluous_types(G)
    set cover(*) = 0;
    for each \(T \in G\) do
        for each \((c, e) \in M \cap (T_C \times T_E)\) do
            increment cover(c,e);
        end for
    end for
    for each \(T \in G\) do
        if \(\#\{(c, e) \in M \cap (T_C \times T_E) \mid \text{cover}(c,e) = 1\} < \text{type.usefulness.required}\) then
            remove \(T\) from \(G\);
            for each \((c, e) \in M \cap (T_C \times T_E)\) do
                decrement cover(c,e);
            end for
        end if
    end for
end sub
```

**User definable settings**

- **type.usefulness.required**: this variable determines how useful a type has to be (in terms of contributions to the coverage of \(G\)) in order to not be discarded. Setting this to 0 will prevent types from being discarded.
setting this to a high value will eliminate all but a few types of large size.
The default value is 1, which eliminates only types which do not contribute anything.

Notes The types are checked in order of increasing absolute total support. This
means that if the matrix can be covered by either a lot of small types or a single
big one, probability favors the latter result.

A.4 Identifying characteristic and secondary contexts and expressions

Synopsis  This routine finds, for each type $T$,

- the characteristic expressions of $T$, defined as those expressions which only
  appear with contexts of no type or of type $T$.

- the characteristic contexts of $T$, defined analogously.

- the characteristic expressions and contexts of $T$, defined as the characteristic expressions and contexts of $T$ if there are any, otherwise defaulting to
  the primary expressions and contexts of $T$.

- the secondary expressions of $T$, defined as its primary expressions and those
  expressions $e$ satisfying

\[
\#\{(c,e) \mid c \in T_E^h, (c,e) \in M\} \geq \#T_E^h \cdot \text{sec.context.support%} \quad (A.1)
\]

- the secondary contexts of $T$, defined as its primary contexts and those
  contexts $c$ satisfying

\[
\#\{(c,e) \mid e \in T_E^h, (c,e) \in M\} \geq \#T_E^h \cdot \text{sec.context.support%} \quad (A.2)
\]

Algorithm

sub identify_characteristic_and_secondary_aspects(G)
for each $T \in G$ do
    set $T_C^h := \emptyset$
    for each context $c \in T_E$ do
        if $\forall c : [(c,e) \in M \Rightarrow (c \in T_E \lor \exists \exists e \in G : e \in L_E)]$ then
            insert $e$ into $T_C^h$:
        end if
    end for
    set $T_E^h := \emptyset$
end for
A.4. Identifying characteristic and secondary contexts and expressions

for each expression \( e \in T_E \) do  
   if \( \forall c : [(c, e) \in M \rightarrow (u \in T_C \lor \exists U \in G : c \in U_C)] \) then  
      insert \( e \) into \( T_E^{ch} \);  
   end if  
end for  
if \( T_E^{ch} \neq \emptyset \) then  
   set \( T_C^* := T_E^{ch} \);  
else  
   set \( T_C^* := T_C ; \)  
end if  
if \( T_E^{ch} \neq \emptyset \) then  
   set \( T_E^* := T_E^{ch} \);  
else  
   set \( T_E^* := T_E \);  
end if  
set \( T_E^{se} := T_C \);  
for each context \( c \) do  
   if \( \#\{e \in T_E^* | (c, e) \in M \} \geq \#T_C^* \cdot \text{sec.context.support}\% \) then  
      insert \( e \) into \( T_C^* \);  
   end if  
end for  
set \( T_E^{se} := T_E \);  
for each expression \( e \) do  
   if \( \#\{e \in T_C^* | (c, e) \in M \} \geq \#T_C^* \cdot \text{sec.expression.support}\% \) then  
      insert \( e \) into \( T_E^{se} \);  
   end if  
end for  
end sub

User definable settings

- \text{sec.context.support}\%, \text{sec.expression.support}\%: lower values for these settings will increase the number of secondary contexts or expressions found. Note that the effects of these settings are independent (unlike with the settings for primary contexts and expressions).

- \text{scsp_for_no_characteristics}. \text{sesp_for_no_characteristics}\%: if a type has no characteristic expressions, the characteristic\* expressions default to the primary expressions. If required support percentages are low, this can result in a single expression being taken as indicative of several types. To prevent this, EMILE provides the setting \text{scsp_for_no_characteristics}. to be used instead of \text{sec.context.support}\% when a type has no characteristic expressions. Similar for \text{sesp_for_no_characteristics}%.
Appendix A. The inner workings of EMILE

A.5 Deriving grammatical rules

Synopsis  Emile uses the grammatical types it finds to infer grammatical derivation rules. A rule \( r : [T] \Rightarrow s_0[T_1]s_1[T_2] \ldots s_k \) is considered to be supported, if \( k = 0 \) and \( s_0 \in T_E^s \), or if \( k \geq 1 \) and for some \( i \leq k \).

\[ \#\{ e \in T_E^* | r \text{ with } [T] \text{ replaced by } e \text{ has support} \} \geq \#T_E^*:\text{rule.support} \% \quad (A.3) \]

An instantiation of a rule \( r : [T] \Rightarrow s_0[T_1]s_1 \ldots s_k \), \( k \geq 0 \), is an expression \( e \in T_E^* \) which can be obtained by replacing the type references \( [T_1], \ldots, [T_k] \) in \( r \) by expressions \( e_1' \in (T_1)^* \), \ldots, \( e_k' \in (T_k)^* \). A rule \( r \) for some type \( [T] \) is considered to be covered by other rules for \( [T] \) if all of its instantiations are also instantiations of one or more of the other rules.

EMILE tries to find a set of supported rules which contains no rules covered by other rules, is capable of generating the original sample, and cannot easily be reduced in size. To do this, the program maintains a set of used types \( V_{\text{used}} \), which initially contains only the whole-sentence type \([0]\). Whenever a type is added to \( V_{\text{used}} \), the program gathers all supported rules for all types of \( V_{\text{used}} \) which only use types from \( V_{\text{used}} \) (note that for this purpose, rules for a type are considered to be using that type). Then rules which are covered by other rules are eliminated, until a set of rules \( R \) is obtained in which every rule has at least one instantiation which is not shared with any other rule. The program adds types to \( V_{\text{used}} \) as long as this will not result in a large increase in the size of the resulting rule-set.

Algorithm

\[
\text{sub derive.rules}(G, R) \\
\text{for each } T \in G \text{ do} \\
\quad \text{set } R_T^{\text{sup}} := \{ [T] \Rightarrow e \mid e \in T_E^* \}; \\
\quad \text{set } R_T^{\text{sup}} := R_T^{\text{sup}} \cup \{ r \mid r \text{ is supported by } R_T^{\text{sup}}, r \text{ uses } [T] \text{ and only } [T] \}; \\
\quad \text{set } R_T := R_T^{\text{sup}}; \\
\quad \text{for each } r \in R_T \text{ do} \\
\quad \quad \text{if } (\exists r' \in R_T : r' \neq r \land r' \text{ covers } r) \text{ then} \\
\quad \quad \quad \text{remove } r \text{ from } R_T; \\
\quad \text{end if} \\
\text{end for} \\
\text{end for} \\
\text{set } V_{\text{used}} := \emptyset; \\
\text{set } R := \emptyset; \\
\text{set } R^{\text{sup}} := \emptyset; \\
\text{set } T_{\text{add}} := [0]; \\
\text{repeat} \\
\quad \text{insert } T_{\text{add}} \text{ in } V_{\text{used}}; \\
\quad \text{set } R^{\text{sup}} := R^{\text{sup}} \cup R_{T_{\text{add}}^{\text{sup}}}; \\
\text{end repeat} \\
\text{return } R^{\text{sup}} \} \\
\]
set $R := R_T$;

for each $T \in G$ do

if ($T \notin V_{\text{used}}$) then

set $R^\text{add}_T := \left\{ r \mid \begin{array}{ll}
    r \text{ is supported by } R^\text{sup}_T \cup R^\text{add}_T, \\
    r \text{ uses both } [T] \text{ and } [T_{\text{add}}]. \\
    r \text{ uses no types outside } V_{\text{used}} \cup \{T\}.
\end{array} \right\}$;

set $R^\text{sup}_T := R^\text{sup}_T \cup R^\text{add}_T$;

set $R_T := R \cup (R_T \cap R^\text{sup}_T) \cup R^\text{add}_T$;

for each $r \in R_T$ do

if ($\exists r' \in R_T : r' \neq r \wedge r'$ covers $r$) then

remove $r$ from $R_T$;

end if

end for

end if

end for

if ($\exists T : \#R_T < \#R + \text{ruleset.increase.disallowed}$) then

select $T_{\text{add}}$ from $G$ such that $T_{\text{add}} \notin V_{\text{used}}$ and $\#R_T$ is minimal

end if

until ($\forall T : \#R_T < \#R + \text{ruleset.increase.disallowed}$):

for each $r \in R$ do

if ($R \setminus \{r\}$ covers $r$) then

remove $r$ from $R$;

end if

end for

end sub

User definable settings

- $\text{rule.support}$: this variable controls the support required for a rule before it is considered for inclusion in the grammar. The lower this value, the more rules will be found.

- $\text{rsp.for.no.characteristics}$: if a type has no characteristic expressions, the characteristic expressions default to the primary expressions. If required support percentages are low, this can result in a single expression being taken as indicative of several types. To prevent this, EMILIE provides the setting $\text{rsp.for.no.characteristics}$ to be used instead of $\text{rule.support}$ when a type has no characteristic expressions.

- $\text{ruleset.increase.disallowed}$: this variable determines how useful a type has to be (in terms of the resulting reduction in the number of rules) in order to be used. Setting this to 0 requires types to reduce the number of rules, setting this to 1 will include types as long as they don't actually
increase the size of the rule-set. Higher values will allow more types to be included, at the expense of increasing the size of the rule-set.

Notes  The sets $R_I$ are actually stored as changes to be applied to $R$. For reasons of efficiency this routine only checks whether rules are covered by single other rules, everywhere except at the very end. This decreases computation time, and also allows for some other optimizations. Covered rules are eliminated in order of increasing complexity. This means that the final result will not contain any rule of which an abstraction exists that uses only types in $V_{used}$ and is supported. Rules of the form $[T] \Rightarrow [U]$ are only allowed if $\#T^e \geq \#U^e$, to prevent loops. For types $T \not\in V_{used}$, the rules in $R_T$ for $[T]$ are retained for reference purposes.

A.6 Short-circuiting superfluous types

Synopsis  If a type $T \in G$ has only a single rule in $R$, or if there is only one reference to $[T]$, then we can decrease the size of the rule-set, and remove $T$ from the set of used types, by substituting the rules of $[T]$ for all references to $[T]$.

Algorithm

```plaintext
sub short-circuit_types(G, R)
    for each $[T] \in V_{used}$ do
        if ($R$ contains only one rule for $[T]$) then
            let $r \in R$ be the unique rule for $[T]$;
            for each rule $r' \in R$ referring to $[T]$ do
                remove $r'$ from $R$;
                substitute $r$ for $[T]$ in $r'$;
                insert $r'$ into $R$;
            end for
            remove $[T]$ from $V_{used}$;
        end if
        if ($R$ contains only one reference to $[T]$) then
            let $r' \in R$ be the unique rule referring to $[T]$;
            remove $r'$ from $R$:
            for each rule $r \in R$ for $[T]$ do
                substitute $r$ for $[T]$ in $r'$;
                insert $r'$ into $R$;
            end for
            remove $[T]$ from $V_{used}$;
        end if
    end for
```
A.7 Parsing a sentence

Synopsis  Given a set of rules, we will sometimes want to see whether a given sentence is parsable with those rules. Furthermore, we will want to see what types unknown words must be assigned in order to make the sentence parsable. EMILE can check for parsability while allowing up to a user-settable number of words to be assigned arbitrary types, using a recursive algorithm.

Algorithm

function parse_phrase(R, s, T)
    if ((T ⇒ s) ∈ R) then
        return 0;
    else if (length(s) = 1) then
        return 1;
    else
        set n := ∞;
        for each rule ([T] ⇒ s₀[s₁]...s_k) ∈ R do
            for each sequence (s',...s') with s₀^s₁...s_k = s do
                set n = min(n, Σ_{i=1}^k parse_phrase(R, s', T_i));
            end for
        end for
        return n;
    end if
end function

function parse_sentence(R, s)
    if (parse_phrase(R, s, [0]) ≤ parser.tolerance) then
        return true;
    else
        return false;
    end if
end function

User definable settings

- **parser.tolerance**: This setting is the maximum number of words to which Emile will assign or reassign a type in order to make parsing of a sentence possible.
Notes  Types are assigned ‘on the fly’ only to single words, not to larger expressions. Emile keeps track of the parser-tolerances used in the search: if a particular search cannot possibly result in a better parsing than the best one found up to now, the searching doesn’t take place.