UNIVERSITY OF AMSTERDAM

UvA-DARE (Digital Academic Repository)

Games, walks and grammars: Problems I've worked on

Vervoort, M.R.

Publication date
2000

Link to publication

Citation for published version (APA):
Vervoort, M. R. (2000). *Games, walks and grammars: Problems I've worked on*. ILLC.

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: https://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.

UvA-DARE is a service provided by the library of the University of Amsterdam (https://dare.uva.nl)

Download date:20 Jan 2022

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

- `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[␣\r\n\t]*\r|^\\.|[^a-zA-Z]\\.|[^␣\r\n\t].\\.
  ```

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 Emile 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. $\#(M \cap (T_C \times T_E)) \geq \#(T_C \times T_E) \cdot$ `total_support%`
2. $\forall c \in T_C : \#(M \cap (\{c\} \times T_E)) \geq \#T_E \cdot$ `context_support%`
3. $\forall e \in T_E : \#(M \cap (T_C \times \{e\})) \geq \#T_C \cdot$ `expression_support%`

The program maintains a set G of grammatical types with sufficient support, and with contexts and expressions of length at most `max_primary_context_length`, or `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 `min_contexts_per_type` and `min_expressions_per_type`).
An element $(c, e) \in M$ is considered *covered* by a type T if $(c, e) \in T_C \times T_E$. This routine updates and enlarges G so that every element $(c, e) \in M$ (that can be covered by a type of minimum size) is covered by a type in G.

Algorithm

```
sub expand_grammar(G)
   for each T ∈ G do
      call enlarge_grammatical_type(T):
      if ((#T_C < min_contexts_per_type)
            or (#T_E < min_expressions_per_type)) then
         remove T from G:
      end if
   end for
   for each (c, e) ∈ M do
      if (¬∃T ∈ G : (c, e) ∈ T_C × T_E) then
         set T := ({c}, {e}):
         call enlarge_grammatical_type(T):
         if ((#T_C ≥ min_contexts_per_type
               or (#T_E ≥ min_expressions_per_type)) then
```

```
            insert T into G;
         end if
      end if
   end for
end sub

sub enlarge_grammatical_type(T)
   repeat
      set X := {c' a context | (T_C ∪ {c'}, T_E) has sufficient support,
                               length(c) ≤ max_primary_context_length},
               ∪{e' an expression | (T_C, T_E ∪ {e'}) has sufficient support,
                               length(e) ≤ max_primary_expression_length};
      if (X ≠ ∅) then
         nondeterministically select x from X;
         if (x is a context) then
            insert x into T_C;
         else
            insert x into T_E;
            end if
         end if
      until (X = ∅);
   end sub
```

User definable settings

- `max_primary_context_length`, `max_primary_expression_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.
- `total_support%`, `context_support%`, `expression_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.
- `min_contexts_per_type`, `min_expressions_per_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 `min_contexts_per_type` contexts or `min_expressions_per_type` expressions are discarded. This may cause some elements $(c, e) \in M$ to be uncoverable.

The type '0' 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 $O(\#\{(c,e) \in M \mid c \in T_C \vee e \in T_E\})$, 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 $O(\#\{(c,e) \in M \mid c \in T_C \vee e \in T_E\})$.

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

```
sub eliminate_superfluous_types(G)
   set cover(*) = 0;
   for each T ∈ G do
      for each (c,e) ∈ M ∩ (T_C × T_E) do
         increment cover(c,e);
      end for
   end for
   for each T ∈ G do
      if (#{(c,e) ∈ M^(T_C × T_E) | cover(c,e) = 1}
            < type_usefulness_required) then
         remove T from G;
         for each (c,e) ∈ M ∩ (T_C × 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

$$\#\{(c,e) \mid c \in T_C^*, (c,e) \in M\} \geq \#T_C^* \cdot \texttt{sec\_context\_support\%} \quad \text{(A.1)}$$

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

$$\#\{(c,e) \mid e \in T_E^*, (c,e) \in M\} \geq \#T_E^* \cdot \texttt{sec\_context\_support\%} \quad \text{(A.2)}$$

Algorithm

```
sub identify_characteristic_and_secondary_aspects(G)
   for each T ∈ G do
      set T_C^ch := ∅
      for each context c ∈ T_C do
         if ∀e : [(c,e) ∈ M → (e ∈ T_E ∨ ¬∃U ∈ G : e ∈ U_E)] then
            insert c into T_C^ch;
         end if
      end for
      set T_E^ch := ∅
```

```
        for each expression e ∈ T_E do
            if ∀c : [(c, e) ∈ M → (u ∈ T_C ∨ ¬∃U ∈ G : c∈U_C)] then
                insert e into T_E^ch;
            end if
        end for
        if (T_C^ch ≠ ∅) then
            set T_C^* := T_C^ch;
        else
            set T_C^* := T_C;
        end if
        if (T_E^ch ≠ ∅) then
            set T_E^* := T_E^ch;
        else
            set T_E^* := T_E;
        end if
        set T_C^se := T_C;
        for each context c do
            if (#{e ∈ T_E^* | (c, e) ∈ M} ≥ #T_E^* · sec_context_support%) then
                insert c into T_C^se;
            end if
        end for
        set T_E^se := T_E;
        for each expression e do
            if (#{c ∈ T_C^* | (c, e) ∈ M} ≥ #T_C^* · sec_expression_support%) then
                insert e into T_E^se;
            end if
        end for
    end for
end sub
```

User definable settings

- `sec_context_support%`, `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).

- `scsp_for_no_characteristics`, `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 `scsp_for_no_characteristics`, to be used instead of `sec_context_support%` when a type has no characteristic expressions. Similar for `sesp_for_no_characteristics%`.

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 $s_0 \in T_E^{se}$, or if $k \geq 1$ and for some $i \leq k$,

$$\#\{e \in T_E^* \mid r \text{ with } [T_i] \text{ replaced by } e \text{ has support}\} \geq \#T_E^* \cdot \texttt{rule\_support\%} \quad \text{(A.3)}$$

An *instantiation* of a rule $r : [T] \Rightarrow s_0[T_1]s_1 \ldots s_k$, $k \geq 0$, is an expression $e \in T_E^{se}$ which can be obtained by replacing the type references $[T_1], \ldots, [T_k]$ in r by expressions $e'_1 \in (T_1)_e^{se}, \ldots, e'_k \in (T_k)_e^{se}$. 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_{used}, which initially contains only the whole-sentence type [0]. Whenever a type is added to V_{used}, the program gathers all supported rules for all types of V_{used} which only use types from V_{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_{used} as long as this will not result in a large increase in the size of the resulting rule-set.

Algorithm

```
sub derive_rules(G, R)
    for each T ∈ G do
        set R_T^sup := {[T] ⇒ e | e ∈ T_E^se};
        set R_T^sup := R_T^sup ∪ {r | r is supported by R_T^sup, r uses [T] and only [T]};
        set R_T := R_T^sup;
        for each r ∈ R_T do
            if (∃r' ∈ R_T : r' ≠ r ∧ r' covers r) then
                remove r from R_T;
            end if
        end for
    end for
    set V_used := ∅;
    set R := ∅;
    set R^sup := ∅;
    set T_add := [0];
    repeat
        insert T_add in V_used;
        set R^sup := R^sup ∪ R_T_add^sup;
```

```
        set R := R_T;
        for each T ∈ G do
            if (T ∉ V_used) then
                set R_T^add := { r | r is supported by R^sup ∪ R_T^sup.
                                     r uses both [T] and [T_add].
                                     r uses no types outside V_used ∪ {T} };
                set R_T^sup := R_T^sup ∪ R_T^add;
                set R_T := R ∪ (R_T ∩ R_T^sup) ∪ R_T^add;
                for each r ∈ R_T do
                    if (∃r' ∈ R_T : r' ≠ r ∧ r' covers r) then
                        remove r from R_T;
                    end if
                end for
            end if
        end for
        if (∃T : #R_T < #R + ruleset_increase_disallowed) then
            select T_add from G such that T_add ∉ V_used and #R_T is minimal
        end if
    until (¬∃T : #R_T < #R + ruleset_increase_disallowed);
    for each r ∈ R do
        if (R−{r} covers r) then
            remove r from R;
        end if
    end for
end sub
```

User definable settings

- `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.

- `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, EMILE provides the setting `rsp_for_no_characteristics`, to be used instead of `rule_support%` when a type has no characteristic expressions.

- `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_T 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 $\#T_E^{se} > \#U_E^{se}$, to prevent loops.
For types $T \notin V_{used}$, 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

```
sub short-circuit_types(G, R)
   for each [T] ∈ V_used do
      if (R contains only one rule for [T]) then
         let r ∈ R be the unique rule for [T];
         for each rule r' ∈ 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' ∈ R be the unique rule referring to [T];
         remove r' from R;
         for each rule r ∈ 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
```

```
end sub
```

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] ⇒ s_0[T_0]s_1 ... s_k) ∈ R do
            for each sequence (s'_1, ..., s'_k) with s_0^s'_1^s_1^ ... ^s'_k^s_k = s do
                set n = min(n, Σ_{i=1}^{k} parse_phrase(R, s'_i, 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.