Generation of Referring Expression Using Prefix Tree Structure

Sibabrata Paladhi
Department of Computer Sc. & Engg.
Jadavpur University, India
sibabrata_paladhi@yahoo.com

Sivaji Bandyopadhyay
Department of Computer Sc. & Engg.
Jadavpur University, India
sivaji_cse_ju@yahoo.com

Abstract

This paper presents a Prefix Tree (Trie) based model for Generation of Referring Expression (GRE). The existing algorithms in GRE lie in two extremities. Incremental algorithm is simple and speedy but less expressive in nature whereas others are complex and exhaustive but more expressive in nature. Our prefix tree based model not only incorporates all relevant features of GRE (like describing set, generating Boolean and context sensitive description etc.) but also try to attain simplicity and speed properties of Incremental algorithm. Thus this model provides a simple and linguistically rich approach to GRE.

1 Introduction

Generation of referring expression (GRE) is an important task in the field of Natural Language Generation (NLG) systems (Reiter and Dale, 1995). The task of any GRE algorithm is to find a combination of properties that allow the audience to identify an object (target object) from a set of objects (domain or environment). The properties should satisfy the target object and dissatisfy all other objects in the domain. We sometimes call it distinguishing description because it helps us to distinguish the target from potential distractors, called contrast set. When we generate any natural language text in a particular domain, it has been observed that the text is centered on certain objects for that domain. When we give introductory description of any object, we generally give full length description (e.g. “The large black hairy dog”). But the later references to that object tend to be shorter and only support referential communication goal of distinguishing the target from other objects. For example the expression “The black dog” suffices if the other dogs in the environment are all non black. Grice, an eminent philosopher of language, has stressed on brevity of referential communication to avoid conversational implicature. Dale (1992) developed Full Brevity algorithm based on this observation. It always generates shortest possible referring description to identify an object. But Reiter and Dale (1995) later proved that Full Brevity requirement is an NP-Hard task, thus computationally intractable and offered an alternative polynomial time Incremental Algorithm. This algorithm adds properties in a predetermined order, based on the observation that human speakers and audiences prefer certain kinds of properties when describing an object in a domain (Krahmer et al. 2003). The Incremental Algorithm is accepted as state of the art algorithm in NLG domain. Later many refinements (like Boolean description and set representation (Deemter 2002), context sensitivity (Krahmer et al 2002) etc) have been incorporated into this algorithm. Several approaches have also been made to propose an alternative algorithmic framework to this problem like graph-based (Krahmer et al. 2003), conceptual graph based (Croitoru and Deemter 2007) etc that also handle the above refinements. In this paper we propose a new Prefix Tree (Trie) based framework for modeling GRE problems. Trie is an ordered tree data structure which allows the organization of prefixes in such a way that the branching at each level is guided by the parts of prefixes. There are several advantages of this approach: 1) Trie data structure has already been extensively used in many domains where search is the key operation. 2) The structure is scalable and various optimized algorithms are there for time, space optimizations.

In this paper it is shown how scenes can be represented using a Trie (section 2) and how description generation can be formalized as a search problem (section 3). In section 4 the algorithm is explained using an example scene. In section 5, the basic algorithm is extended to take care of different scenarios. The algorithm is analyzed for time complexity in section 6 and conclusion is drawn in section 7.

2 Modeling GRE Using Trie Structure

In this section, it is shown how a scene can be represented using a trie data structure. The scheme is based on Incremental algorithm (Reiter and Dale 1995) and incorporates the attractive properties (e.g. speed, simplicity etc) of that algorithm. Later it is extended to take care of different refinements (like relational, boolean description etc) that could not be handled by Incremental algorithm. Reiter and Dale (1995) pointed out the notion of '**PreferredAttributes**' (e.g. Type, Size, Color etc) which is a sequence of attributes of an object that human speakers generally use to identify that object from the contrast set. We assume that the initial description of an entity is following this sequence (e.g. "The large black dog") then the later references will be some subset of initial description (like "The dog" or "The large dog") which is defined as the prefix of the initial description. So, we have to search for a prefix of the initial full length description so that it is adequate to distinguish the target object. Following the Incremental version we will add properties one by one from the '**PreferredAttributes'** list. In our model, the root consists of all entities in the domain and has empty description. Then at each level, branching is made based on different values of corresponding preferred attribute. The outgoing edge is labeled with that value. For example, at the first level, branching is made based on different values of 'Type' attribute like 'Dog', 'Cat', 'Poodle' etc. A node in Trie will contain only those objects which have the property(s) expressed by the edges, constituting the path from root to that node. After construction of the Trie structure for a given domain in this way, referring expression generation problem for an object **r** is reduced to search the tree for a node which consists of **r** and no other object. Description for **r** can be found from the search path itself as we have said earlier. Now we will introduce some notations that we will use to describe the actual algorithm. Let **D** be the Domain, **r** be the target object and **P** be the '**PreferredAttributes'** List. $[\![N_i]\!] = \{d \mid d \in D$ and d is stored at node $N_i\}$ where N_i is an i-th level node. Obviously $[\![N_o]\!] = D$ since N_o is root node. $E(N_i, N^k_{i+1})$ is an edge between parent node N_i and N^k_{i+1}, k-th child of that node (considering an enumeration among children nodes). Since every edges in Trie are labeled, thus $\{E\} \subseteq \{N\} \times L \times \{N\}$, where $\{E\}$ and $\{N\}$ are set of all edges and nodes respectively in the tree and L is the set of attribute values. Let $Val(E(N_i, N^k_{i+1}))$ denotes the label or value of the edge and $[\![Val(E(N_i, N^k_{i+1}))]\!] = \{d \mid d \in D$ and d is satisfied by the edge value$\}$ i.e. the set contains those objects who have this property. We define $[\![N^k_{i+1}]\!] = [\![N_i]\!] \cap [\![Val(E(N_i, N^k_{i+1}))]\!]$ where N_i and N^k_{i+1} are parent and child node respectively. Similarly $[\![N^k_i]\!] = [\![N_{i-1}]\!] \cap [\![Val(E(N_{i-1}, N^k_i))]\!]$. Ultimately, we can say that $\forall i\ [\![N_i]\!] = [\![N_o]\!] \cap [\![Val(E(N_o,N_1))]\!] \cap \ldots\ldots \cap [\![Val(E(N_{i-1},N_i))]\!]$. Since our construction is basically a tree, each node is reachable from root and there exists a unique path from root to that node. So, for each node in the tree we will get some description. We will formulate referring expression construction as search in the constructed tree for the node $_{\mathbf{min(k)}}\{\mathbf{N_k}\}$ such that $[\![N_k]\!] = \{\mathbf{r}\}$. If N_k is leaf node then description of **r** will be same with the full description but if it is an intermediate node then description is some proper prefix of initial description. But the point is that, in both cases the later reference is prefix of initial one (as both "ab" and "abc" are prefixes of "abc").

3 Basic Algorithm

Based on above discussions, algorithms are developed for construction of Trie from the domain and generation of reference description for any object in that domain. The Trie construction algorithm **ConstructTrie(D,P,T)** is shown in figure 1, Referring expression generation algorithm **MakeRefExpr(r,p,T,L) is** shown in figure 2, where T is a node pointer and p is pointer to parent of that node. Our algorithm **MakeRefExpr** returns set of attribute-values L to identify **r** in the domain. As discussed earlier, it is basically a node searching algorithm. In course of searching, if it is found that an intermediate node **N** doesn't have **r** i.e. $\mathbf{r} \notin [\![\mathbf{N}]\!]$ then our search will not move forward through the subtree rooted at **N**. Our search will proceed through next level iff $\mathbf{r} \in [\![\mathbf{N}]\!]$. For a node $\mathbf{N_k}$, if we get $[\![\mathbf{N_k}]\!] = \{\mathbf{r}\}$ then we have succeeded and our algorithm will return **L**, set of descriptions for that node. If there is no distinguishing description exists for **r**, then $\varnothing$ (null) will be returned**.** We would like to point out that our algorithm will find out only one description that exists at the minimum level of the tree. Moreover, a description is added to **L** only if it is distinguishing i.e. the connecting edge must remove some contrasting object(s). Thus, the child node should contain less number of objects than that of parent node. In this case, cardinality of parent $\mathbf{N_i}$ (Card($\mathbf{N_i}$)) will be greater than that of child (Card($\mathbf{N_{i+1}}$)). This condition is included in our algorithm and if (Card (P$\rightarrow$N)) > Card (T$\rightarrow$N) holds then only the value is added

```
ConstructTrie(D, P, T) {
  If ($D = \emptyset \vee P = \emptyset$)
  Then Stop
  Else
    Create a node N at T
    Set $[\![ N ]\!] = D$
    Extract front attribute $A_i$ from list P
    $P' = P - \{A_i\}$
    For each value $V_j$ of attribute $A_i$ do
       Create Edge $E_j$ with label $V_j$ as $T \rightarrow Next_j$
       $D_j' = D \cap [\![ Val(E_j) ]\!]$
       ConstructTrie($D_j'$, $P'$, $T \rightarrow Next_j$)
    End For
  End If
}
```

Figure 1. Prefix Tree Generation Algorithm

```
MakeRefExpr(r, P, T, L) {
   If ($r \notin [\![ T \rightarrow N ]\!]$)
       Then $L \leftarrow \emptyset$
       Return L
   Else If ($\{r\} = [\![ T \rightarrow N ]\!]$)
       $L = L \cup Val(P \rightarrow E_j)$
       Return L
   Else If (isLeaf (T) $\wedge \{r\} \subset [\![ N ]\!]$)
       Then $L \leftarrow \emptyset$
       Return L
   Else {
      If (Card($P \rightarrow N$) > Card ($T \rightarrow N$))
         Then $L = L \cup Val(P \rightarrow E_j)$
      P = T
      For each outgoing edge $T \rightarrow Next_j$ ($E_j$) do
        $L'$ = MakeRefExpr(r, P, $T \rightarrow Child_j$, L)
          If ($L' \neq \emptyset$)
          Then Return $L'$
   } }
```

Figure 2. Expression Generation Algorithm

P->N and T->N respectively represents parent and child node. After finding a distinguishing description for **r,** search will neither move further down the tree nor explore the remaining branches of the current node. Search will explore the next branch only if the search in current branch returned NULL description i.e. when $L' = \emptyset$ in the algorithm. If we reach a leaf node and that contains **r** along with other objects then it is not possible to distinguish **r**'. In that case, the algorithm returns NULL indicating that no description exists at all. It has been later shown that some distinguishing description may still exist and the algorithm will be modified to find that. It should be mentioned that once the prefix tree is constructed offline, it can be used repetitively to find description for any object in the domain throughout the text generation phase. Our **MakeRefExpr()** algorithm is very simple and it doesn't employ any set theoretic operation, which is a non trivial task, to find current contrast set at every steps of algorithm. In existing algorithms, computing referential description for every object require computing similar things (like finding current contrast set, ruled out objects) again and again. And it has to be repeated every time the object is referred. It is not possible to generate description once, store it and use it later because of the fact that domain may also change in course of time (Krahmer, 2002). That's why every time we want to refer to 'r', such rigorous set operations need to be computed. But in our prefix tree structure, once the tree is constructed, it is very easy to find description for that object using simple tree search function. It is also very easy to add/delete objects to/from domain. We have to follow just the initial properties of that object to find the proper branching at each level, followed by addition /deletion of that object to /from relevant nodes, which is essentially a search operation. The disadvantage of our algorithm is that space complexity is high but it can be tackled using bit Vector representation of individual nodes of the prefix tree. Besides, several methods are there for compressing Trie structure. But these optimization techniques are beyond the scope of our current discussion.

4 Formalizing A Scene using Prefix Tree

Consider an example scene in figure 3, from [Krahmer 2002]. In this scene, there is a finite domain of entities D. Let D = {d1, d2, d3, d4}, P = {Type, Size, Color} and values are Type = {dog, cat}; Size = {small, large}; Color = {black, white}. A scene is usually represented as a database (or knowledge base) listing the properties of each element in D. Thus:

d_1 : 〈 Type : dog 〉, 〈 Size : small 〉, 〈 Color: white 〉
d_2 : 〈 Type : dog 〉, 〈 Size : large 〉, 〈 Color: white 〉
d_3 : 〈 Type : dog 〉, 〈 Size : large 〉, 〈 Color: black 〉
d_4: 〈 Type : cat 〉, 〈 Size: small 〉, 〈 Color: white 〉

Now it will be shown how our **MakeRefExpr()** algorithm will find a description for a target object **r**. Let **r** = {d_1}. In the first phase, starting from root, edge labeled D is traversed. Since d_1 exists in the node and D discards some objects (d_4), D is distinguishing description and it is added to L. In the next phase the node connected by the edge labeled L does not contain d_1 so search will not proceed further. Rather the node connected by the edge labeled S contains d_1. Since, d_1 is the only object, then we are done and the referring expression is "The small dog". But for d_2, we have to search upto the leaf node which generates the description "The large white dog".

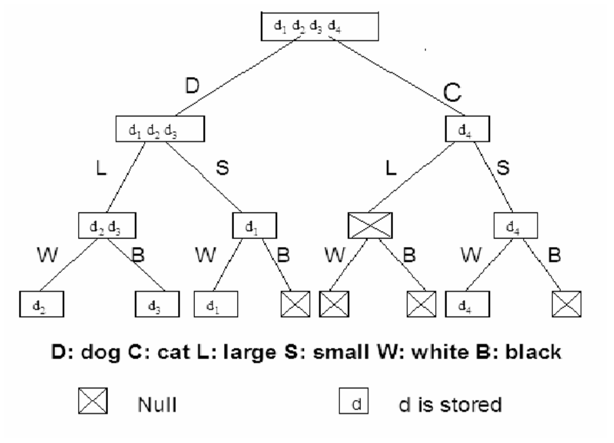

Figure 3. Scene Representation

5 Extension of Basic Algorithm

5.1 Specifying Overlapping Values

Deemter (2002) has shown incompleteness of Incremental algorithm in case of overlapping values. Due to vagueness of properties, sometimes it is hard to classify an object in a particular class. Consider the example scene D = {a,b,c,d} Color: {Red(a,b); Orange(a,c,d)} Size: {Large(a,b); Small(c,d)}. In this case **a** can not be properly classified by Color type. Incremental algorithm always select Red(a,b) at first phase, since it rules out maximum distractors and returns failure because it can't distinguish **a** from **b** at second phase. Deemter(2002) suggested inclusion of all overlapping values that are true of target while also removing some distractors. So, referring expression for **a** is "The red orange desk". But it fails to obey Gricean maxims of conversational implicature. We consider the failure as 'Early Decision' problem and defer the decision making in our model. We keep in our mind the fact that human beings seldom take instantaneous decision. Rather they consider all opportunities in parallel and take decision in the favor of the best one at later point of time. Since, our algorithm searches in parallel through all promising branches until some description is found; it mimics the capabilities of human mind to consider in parallel. Our algorithm will generate "The large orange desk" which will help audiences to better identify the desk. The execution sequence is shown in figure 4.

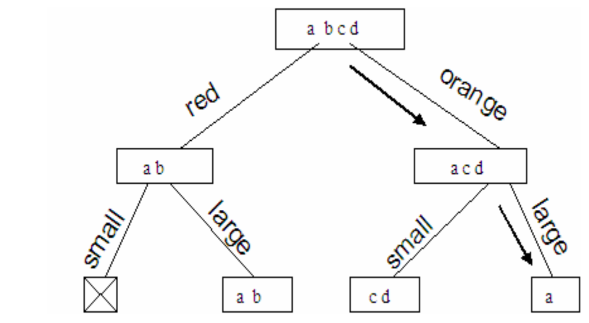

Figure 4. Dealing with overlapping values

5.2 Describing Set of Objects

Generation of referring description for a set of objects is very important in NLG. Deemter's (2002) suggestion can be easily incorporated into our framework. We will represent target **r** as set of objects. Now our algorithm will try to find a node in the tree which only consists of all objects in the set **r**. In this way, we can find a distinguishing description for any set, for which description exists. In figure 3, the description for the set {d_2,d_3} is "The large dogs". Thus, our basic algorithm is able to describe set of objects. In case of set like {d_2, d_3, d_4} where there is no separate node consisting all the object, we need to partition the set and find description for individual set. In our case the possible partitions are {d_2, d_3} and {d_4} for which separate nodes exist.

5.3 Boolean Descriptions

Deemter (2002) shown that Incremental algorithm is only intersectively complete. But he argues that other Boolean combination of properties can be used to generate description for an object. Consider the example from (Deemter, 2002). Let D = {a, b, c, d, e} Type: {Dog(a,b,c,d,e); Poodle(a,b)} Color: {Black(a,b,c); White(d,e)} and r = {c}. In this scenario Incremental algorithm is not able to individuate any of the animals. However a description for c exists, "The black dog that is not a poodle". Since {c} = [[Black]] ∩ [[¬ Poodle]]. Deemter (2002) has modified the Incremental algorithm by adding negative values for each attribute. Now we will show that our basic algorithm can be modified to take care of this situation. In our basic algorithm **ConstructTrie()**, we add branches at each level for negative values also. In this case our simple routine **MakeRefExpr() is** able to find boolean description while remaining as close as to Incremental algorithm. In figure 5, we show part of the trie structure, which is generated for the above scene. The dashed arrows show the alternative search paths for node containing {c}.

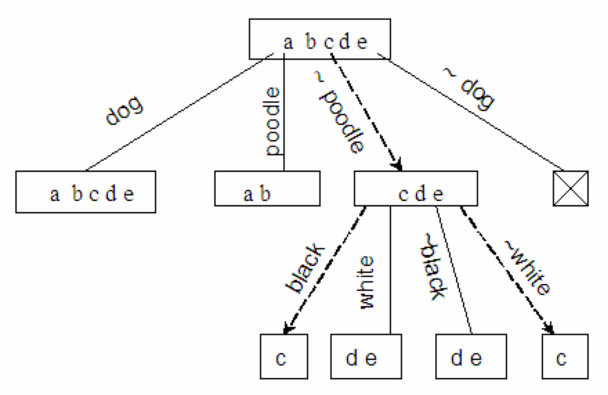

Figure 5. Trie structure (Partial) incorporating negation of properties

For referring objects using disjunction of properties we have do same thing as negations. We have to extend our prefix tree structure by adding extra edges at different levels for making implicit information explicit as described in [Krahmer 2002].

5.4 Incorporating Context Sensitivity

Krahmer and Theune [2002] have added the notion of context sensitivity into GRE. Earlier algorithms assumed that all objects in environment are equally salient. Krahmer and Theune refined the idea by assigning some degree of salience to each object. They proposed that during referring any object, the object needs to be distinguished only from those objects which are more salient (having higher salience weight). An object that has been mentioned recently, is linguistically more salient than other objects and can be described using fewer properties ("The dog" instead of "The large black hairy dog"). They introduced the concept of centering theory, hierarchical focus constraints in the field of NLG and devised a constant function mapping **sw: D** $\rightarrow \mathbb{N}$, where **sw** is salience weight function, **D** is domain and $\mathbb{N}$ is set of natural numbers. We can incorporate this idea into our model easily. In each node of the prefix tree we keep a field 'salience weight' (sw) for each of the object stored in that node in the form (d_i, sw_i). During describing an object if we find a node that is containing **r** where it is the most salient then we need not traverse higher depth of the tree. So, we have to modify **MakeRefExpr()** algorithm by adding more conditions. If the current node is **N** and both **1) r**$\in [\![N]\!]$ **and 2)** $\forall d \in [\![N]\!] (d \neq r \rightarrow sw(d) < sw(r))$ hold then **r** is the most salient and the edges constituting the path from root to **N** represents distinguishing description for **r**. In figure 6, **a** is most salient dog and referred to as "The dog" whereas **b** is referred to as "The small dog".

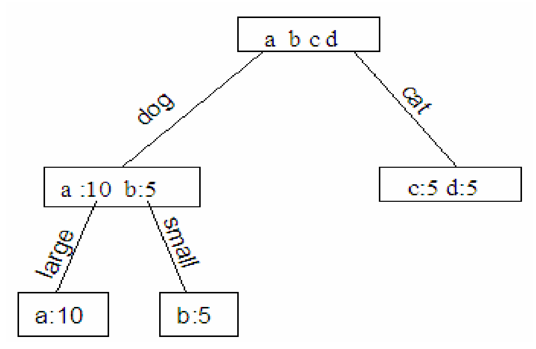

Figure 6: Trie structure (Partial) representing Context Sensitivity

5.5 Relational Descriptions

Relational descriptions are used to single out an object with reference to other one. For example "The cup on the table" is used to distinguish a cup from other cups which are not on the table. Dale and Haddock (1991) first offer the idea of relational description and extend Full Brevity algorithm to incorporate this idea. Later Krahmer et al. (2003) Graph based framework for generating relational description. We follow Krahmer (2002) and denote relations as Spatial: {In(a,b); Left_of(c,d)} etc. Then we treat 'Spatial' as another attribute and consider 'In', 'Left_of' as different values for that attribute. In this way, our basic algorithm itself is capable of handling relational descriptions. The only modification that we add that when a relation R is included, the **MakeRefExpr()** should be called again for the relatum. Thus, if Val(E(N_i, N^k_{i+1})) expresses a relation of **r** with **r′** then we have to call **MakeRefExpr (r′,p,T,L)** again to find description for **r′.**

5.6 Modeling Full Brevity

In this section, we will show that our prefix tree structure can be so modified that it can generate shortest possible description which is requirement of Full Brevity (Dale, 1992). Consider a scene where a domain is identified by set of n attributes $\{A_1, A_2 \ldots A_n\}$. We can generate n! number of different permutations of A_i's $\forall\, i \in [1,n]$. We consider each permutation as different **PreferredAttributes** list P_k and generate all possible prefix trees T_k for each P_k $\forall\, k \in [1,n!]$ for same domain D. Now, we connect roots of all trees with a common dummy root node with edges having empty description (ε). Now, if we search the branches of new combined tree in parallel, it's obvious that we can always find the target node at lowest possible level. Thus we can generate shortest length description using our algorithm.

6 Complexity of The Algorithm

Let the domain entities are identified by **a** number of attributes and each attribute has on the average **k** number of different values. So, our **ConstructTrie()** algorithm takes $O(k^a)$ time. Now we will consider different cases for analyzing the time complexity of our MakeRefExpr() algorithm.

1) In case of non overlapping properties, our search tree will be pruned at each level by a factor of k. Thus the time complexity will be $O(\log_k(k^a))$ = O(a) which is linear.

2) In case of overlapping properties, we have to search whole tree in worst case (although in average cases also there will be large pruning, as found from test cases) which will take $O(k^a)$ time.

3) In case of achieving full brevity requirement, both time and space complexity will be exponential as in the original algorithm by Dale (1992).

7 Conclusions

In this paper, we present a new Prefix tree (Trie) based approach for modeling GRE problems. We construct the trie in such a way that a node at a particular level consists of only those objects which are satisfied by values of the edges, constituting the path from root to that node. We formulate description generation as a search problem. So, when we reach the target node, the attribute values corresponding to the edges in the path automatically form the distinguishing description. Different scenarios of GRE problems like representation of set, boolean descriptions etc. is taken care of in this paper. We have shown that in simple non overlapping scenarios, our algorithm will find distinguishing description in linear time.

8 References

E. Krahmer and M. Theune. 2002. Efficient Context Sensitive Generation of Referring Expressions. *CSLI Publ, Stanford* : 223 – 264

E. Krahmer, S. van Erk and A. Verlag. 2003. Graph based Generation of Referring Expressions *Computational Linguistics*, 29(1): 53-72

H. Horacek. 2004. On Referring to Set of Objects Naturally. *Proceedings of Third INLG, Brokenhurst, U.K: 70-79*

M. Croitoru and van Deemter. 2007. A conceptual Graph Approach to the Generation of Referring Expressions. *Proceedings of IJCAI 2007* : 2456-2461

R. Dale and N. Haddock. 1991. Generating Referring Expressions containing Relations. *Proceedings of Fifth ACL- EACL conference*, 161-166

R. Dale. 1992. *Generating Referring Expressions: Building Descriptions in a Domain of Objects and Processes.* MIT Press

R. Dale and E. Reiter. 1995. Computational Interpretations of the Gricean Maxims in the generation of Referring Expressions. *Cognitive Science* (18): 233 – 263

van Deemter. 2002. Generating Referring Expressions: Boolean Extensions of Incremental Algorithm. *Computational Linguistics* 28(1): 37-52