Viewing Referring Expression Generation as Search

Bernd Bohnet[*] **and Robert Dale**[†]
[*]Institute for Intelligent Systems, University of Stuttgart, Germany
[†]Centre for Language Technology, Macquarie University, Sydney, Australia
Bohnet@iis.uni-stuttgart.de, Robert.Dale@mq.edu.au

Abstract

Almost all natural language generation (NLG) systems are faced with the problem of the generation of referring expressions (GRE): given a symbol corresponding to an intended referent, how do we work out the semantic content of a referring expression that uniquely identifies the entity in question? This is now one of the most widely explored problems in NLG: over the last 15 years, a number of algorithms have been proposed for addressing different aspects of this problem, but the different approaches taken make it very difficult to compare and contrast the algorithms provided in any meaningful way. In this paper, we show how viewing the problem of referring expression generation as a search problem allows us to recast existing algorithms in a way that makes their similarities and differences clear.

1 Introduction

A major component task in natural language generation (NLG) is the generation of referring expressions: given an entity that we want to refer to, how do we determine the content of a referring expression that uniquely identifies that intended referent? This is now a widely explored problem in the NLG literature; since at least [Dale, 1989], the standard conception of this task has been as follows:

1. We assume we have a knowledge base that characterises the entities in the domain in terms of a set of attributes and the values that the entities have for these attributes; so, for example, our knowledge base might represent the fact that entity e_1 has the value *cup* for the attribute *type*, and the value *red* for the attribute *colour*.
2. In a typical context where we want to refer to some e_i, which we call the *intended referent*, there will be other entities from which the intended referent must be distinguished; these are generally referred to as *distractors*. So, for example, we may want to distinguish a particular cup from all the other items present in the context of a dining table.
3. The goal of referring expression generation is therefore to find some collection of attributes and their values which distinguish the intended referent from all the potential distractors in the context.

Over the last 15 years, a wide variety of algorithms have been proposed to deal with specific aspects of this problem. For example, while earlier algorithms focussed on the use of attributes that correspond to simple one-place predicates (as might be realised, for example, by means of adjectives such as *red* or *large*), later work attempts to address the use of relational predicates (as might be realised by prepositions such as *in* and *next to*), and other work looks at the incorporation of boolean operators such as *not* and *and*. Consequently, we now have a considerable body of research in this area; however, it is difficult to establish just how these different algorithms relate to each other.

This paper represents a first step towards consolidating the results in this area, with the aim of developing a framework within which different algorithms can be compared and assessed. The structure of the paper is as follows. In Section 2, we provide a brief overview of work on the generation of referring expressions to date. In Section 3, we borrow a standard approach used in Artificial Intelligence (AI) to represent problems in an elegant and uniform way (see, for example, [Simon and Newell, 1963]; [Russell and Norvig, 2003]), sketching how GRE algorithms can be expressed in terms of *problem-solving by search*. In Section 4, we explore how the most well-known algorithms can be expressed in this framework. In Section 5, we discuss how this approach enables a more fruitful comparison of existing algorithms, and we point to ways of taking this work further.

2 A Brief Review of Work To Date

Although the task of referring expression generation is discussed informally in earlier work on NLG (in particular, see [Winograd, 1972; McDonald, 1980; Appelt, 1981], the first formally explicit algorithm was introduced in [Dale, 1989]. This algorithm, which we will refer to as the Full Brevity (FB) algorithm, is still frequently used as a basis for other GRE algorithms. The FB algorithm searches for the best solution amongst all possible referring expressions for an entity; the algorithm derives the smallest set of attributes for the referent in question, producing a referring expression that is both *adequate* (in the sense that it provides enough information to do what it needs to do) and *efficient* (in the sense that it does not provide any more information than it needs to).

This initial algorithm limited its application to one-place predicates. [Dale and Haddock, 1991] introduced a constraint-based procedure that could generate referring expressions involving relations (henceforth IR), using a greedy heuristic to guide the search.

As a response to the computational complexity of greedy algorithms, [Reiter and Dale, 1992; Dale and Reiter, 1995] introduced the psycholinguistically motivated Incremental Algorithm (IA). The most used and adapted algorithm, this is based on the observation that people often produce referring expressions which are informationally redundant; the algorithm uses a preference ordering over the attributes to be used in a referring expression, incorporating those attributes which rule out at least one potential distractor.

In recent years there have been a number of important extensions to the IA. The Context-Sensitive extension (CS; [Krahmer and Theune, 2002]) is able to generate referring expressions for the most salient entity in a context; the Boolean Expressions algorithm (BE; [van Deemter, 2002]) is able to derive expressions containing boolean operators, as in *the cup that does not have a handle*; and the Sets algorithm (SET; [van Deemter, 2002]) extends the basic approach to references to sets, as in *the red cups*. Some approaches reuse parts of other algorithms: the Branch and Bound (BaB; [Krahmer *et al.*, 2003]) algorithm uses the Full Brevity algorithm, but is able to generate referring expressions with both attributes and relational descriptions using a graph-based technique.

We have identified here what we believe to be the most cited strands of research in this area, but of course there are many other algorithms described in the literature: see, for example, [Horacek, 1997; Bateman, 1999; Stone, 2000]. Space limitations prevent a complete summary of this other work here, but our intention is to extend the analysis presented in this paper to as many of these other algorithms as possible.

All these algorithms focus on the generation of definite references. In principle, they would be embedded in a higher-level algorithm that includes cases for when the entity has not been previously mentioned (thus leading to an initial indefinite reference) or when the referent is in focus (thus leading to a pronominal reference). However, in practice, most work tends to focus exclusively on definite reference; see [Dale, 1989; Krahmer and Theune, 2002; Dale, 2003] for further discussion of the more general case.

3 GRE from the Perspective of Problem Solving

With so many algorithms to choose from, it would be useful to have a uniform framework in which to discuss and compare algorithms; unfortunately, this is rather difficult given the variety of different approaches that have been taken to the problem.

Within the wider context of AI, [Russell and Norvig, 2003] present an elegant definition of a general algorithm for problem solving by search. The search graph consists of *nodes* with the components *state* and *path-cost*; a problem is represented by an *initial-state*, an *expand-method* which identifies new states in the search space, a *queuing-method* which determines the order in which the states should be considered, and a *path-cost-function* which determines the cost of reaching a given state. In this framework, the search strategy is determined by the combination of *queuing-method* and *path-cost-function* used.

In the following, we use this framework to provide a characterisation of existing GRE algorithms in terms of problem solving by search. Given an intended referent, a set of properties true of the intended referent,[1] and the set of distractor entities from which we wish to distinguish the intended referent, we can conceptualise the search space as consisting of states that correspond to possible descriptions of the intended referent. Each state has three components: a description that is true of the intended referent, the set of distractor entities that the description also applies to besides the intended referent, and the set of properties of the intended referent that have not yet been considered in describing the referent.

1. The *initial-state* is of the form $\langle\{\}, C, P\rangle$, where C is the set of distractors in the initial context, and P is the set of all properties true of the intended referent.

2. The goal state is of the form $\langle\{\lambda x P_1, \lambda x P_2, \ldots\}, \{\}, P'\rangle$, where the first term contains a set of properties of the intended referent that, by virtue of the second term (the set of distractors) being an empty set, distinguish the intended referent; P' contains any properties of the intended referent not yet used in the description.

3. All other states in the search space are then intermediate states through which an algorithm will move as it adds new properties to the description.

4. The search strategy is carried out by the *expand-method* and the *queuing-method*, which together characterise the specific GRE algorithm (for example, FB, GH or IA) that is used.

5. The *path-cost-function* allows us to route the search as required; this can be used to take account of salience weights, or to embody some kind of heuristic search.

For any given algorithm, not all of the methods and functions need to be implemented; in particular, we will see that some algorithms do not require a *path-cost-function*.

[1]We will use the notion of a property to correspond to an attribute and its value; this will considerably simplify the notation we provide.

4 GRE Algorithms in Terms of Problem Solving

We adopt here an object oriented formalism,[2] since this allows the representation of dependencies between the algorithms by means of inheritance and overwriting.

To enable more fruitful comparison of the different GRE algorithms, we want to distinguish those aspects of the algorithms which are true of all algorithms, and those which are unique to each particular algorithm. In Section 4.1, we first describe the elements that are shared by all the algorithms; we then go on to describe the distinct aspects of each algorithm in turn.

4.1 Common Elements

Following from the previous section, the definitions of the *node* and *state* classes are as shown in Definition 1. This figure also shows the definitions for *initial-state* and *goal*, which remain constant across the algorithms.

Definition 1: The Node and State Classes

```
class Node {
  $s$ // State
  $path\text{-}cost$ // Cost of the path to this node
  $getState()$
    {return $s$} // returns the state of the node
}
class State {
  $L$ // Set of chosen properties and/or relations
  $C$ // Set of distractors
  $P$ // Set of available properties and/or relations
}
initialState() {return new State($\emptyset$,$C$,$P$)}
// the goal is the empty set of distractors
goal($s$) {
  if $s.C = \emptyset$ then return true
  else return false
}
```

Given these components, the main method *makeRefExp* is then as represented in Definition 2. This takes two arguments, which serve as the parameters that distinguish one algorithm from another: an *expand* method to create the successors of a given state, and a *queue* method, which defines how to insert nodes into the node queue. Depending on the order in which the nodes are inserted, different search strategies can be realized: for example, when the nodes are inserted at the front of the queue, the search strategy is depth-first; when the nodes are inserted at the end of the queue, the search strategy is breadth-first; when the nodes of the queue are sorted by the estimated distance to the goal, then the search type is best-first; and so on.

In addition, we may require a number of general-purpose methods which can be used by a number of different

[2]We follow the code conventions typically used in OO-languages, where the names of classes start with upper case characters, and the names of methods and variables start with lower case characters.

Definition 2: The Basic Algorithm Structure

```
makeRefExp() {
  // create a initial queue with a single node
  $nodeQueue$ ← [new Node(initialState())]
  while $nodeQueue \neq \emptyset$ do
    $node$ ← removeFront($nodeQueue$)
    if goal(node.getState()) then
      return node // success
    end
    $nodeQueue$ ← queue($nodeQueue$,expand($node$))
  end
  return nil // failure
}
```

algorithms. One such 'utility function' is the method *rulesOut*, which takes a property or relation p and a set of distractors, and returns the set of distractors which are ruled out by p.

With this machinery in place, we can now redefine the existing algorithms in terms of their core differences, which correspond essentially to different ways of expanding the search space.

4.2 The Full Brevity Algorithm

The distinctive property of the Full Brevity (FB) algorithm is that it computes all combinations of the available properties P with increasing length, so that it may find the shortest combination that succeeds in identifying the intended referent.

This behaviour is captured by the *expand* method shown in Definition 3. The method creates a set of successors by creating a node for each property p_i which has not so far been checked, provided that p_i rules out at least one distractor.

The FB algorithm uses a breadth-first search implementation of the queue, as shown in Definition 4. Consequently, any solution for which *goal* returns *true* will have a minimal number of properties, since the breadth-first search considers smaller combinations of properties first.

The FB algorithm uses the *expand* method, and *createNode* method which are shown in Definition 3 and it is invoked by a call of *makeRefExp* method which is shown in Definition 2.

4.3 The Incremental Algorithm

The distinctive property of the Incremental Algorithm is that it reduces the computational complexity of constructing a referring expression by considering properties to use in sequence from a predefined ordering of the available properties. The implementation of the *expand* method shown in Definition 5 provides this behaviour.

If the set of properties of the current state $s.P$ is not empty, then the first property p according to the given order O is chosen from the set of properties of the current state $s.P$, and a node is created with a new state by the method *createNode*. Note that the *createNode* method is the same as that used in the FB algorithm and shown in Definition 3.

Definition 3: The Full Brevity Algorithm

```
*expand*($node$) {
    $N \leftarrow \emptyset$
    $s \leftarrow node.getState()$
    **foreach** $p \in s.P$ **do**
        $N \leftarrow N \cup \{$ *createNode* $(node, p)\}$
    **end**
    **return** $N$
}
*createNode*($node$, $p$) {
    $s \leftarrow node.getState()$
    $out \leftarrow$ *rulesOut*($p$, $s.C$)
    **if** $out \neq \emptyset$ **then**
        **return new** Node($s.C - out$, $s.L \cup \{p\}$,
                            $s.P - \{p\}$)
    **else return new** Node($s.C$, $s.L$, $s.P - \{p\}$)
}
```

Definition 4: Breadth-first Queueing

```
*queue*($actNodes, newNodes$) {
    // append the nodes at the end
    **return** $actNodes \cup newNodes$
}
```

Unlike the *expand* method used in the FB algorithm, however, the set of nodes returned here contains only one node. The main method applies the *goal* predicate to this node; if this returns true, then the node containing the state with the list of properties for the referring expression is returned.

Definition 5: The Incremental Algorithm

```
$O$ // Predefined constant order of properties
*expand*($node$) {
    $N \leftarrow \emptyset$
    $s \leftarrow node.getState()$
    **if** $s.P \neq \emptyset$ **then**
        $p \leftarrow$ choose the first p in O, where $p \in s.P$
        $N \leftarrow N \cup$ { *createNode*($node, p$) }
    **end**
    **return** $N$
}
```

4.4 Extension of the IA to Sets

The algorithms considered so far have been concerned with constructing descriptions for individual referents; [van Deemter, 2002] introduced an algorithm which extends the IA to sets. The extension is shown in terms of our framework in Definition 6.

Note that, precisely because this algorithm is an extension of the IA algorithm, we reuse the *expand* method from that algorithm. Consequently, the extension requires only the rewriting of the *createNode* method, whereby an attribute p_i is only chosen when it does not rule out entities

Definition 6: The Set Algorithm

```
$R$ // Set of referents
*createNode*(node, $p$) {
    $out \leftarrow$ rulesOut($p, s.C$)
    **if** $(\neg\exists x \in R \& x \in out)\&(\exists x \in C \& x \in out)$ **then**
        **return new** Node($s.C - out$, $s.L \cup \{p\}$,$s.P - \{p\}$)
    **else return new** Node($s.C$, $s.L$, $s.P - \{p\}$)
}
```

from the set of intended referents R and when it rules out at least one entity from the set of distractors C. If a property does not fulfil that condition, then a node with the current state is returned and the process is continued, as in the IA, with the next property.

4.5 GRE Involving Relations

The algorithm for GRE Involving Relations (IR) introduced by [Dale and Haddock, 1991] is constraint-based. The search strategy used to fulfil the constraints is a combination of a greedy search, which chooses the relation that leads to the smallest set of distractors, and depth-first search to describe the entities, that is, the intended referent as well as entities which are referenced in the relations.

The strategy can be explained best by means of an AND/OR-tree, as shown in Figure 1. Here, the top node represents a state in which relational properties are to be considered as additions to the set of chosen properties. Each search step consists of two stages: in the first stage, we choose the relation p_i which rules out the largest number of distractors; in the second stage, each entity which is referenced by the chosen relation has to be described by repeating the process recursively. This is done in a depth-first manner, but if the related entity is not uniquely distinguished then the next p_j that the intended referent participates in is chosen, and so on. This process continues until all entities are uniquely described (*success*) or no further relations can be chosen (*failure*).

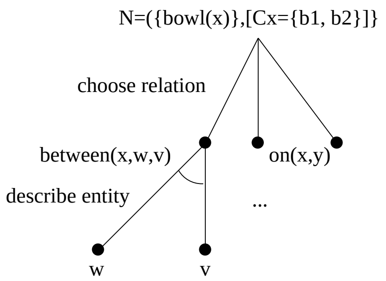

Figure 1: Expansion tree for the IR algorithm

The algorithm is represented in the problem solving paradigm as in Definition 7. Here, the *expand* method chooses a relation which rules out the largest number of distractors; it then calls the method *createNode*, which recursively calls *makeRefExp* for each new referent contained in the relation.

Definition 7: Involving Relations

```
$r$ // Referent
*expand*($node$) {
    $s \leftarrow node.getState()$
    $p_c \leftarrow nil$
    // Chosen relation is $p_c$, where $p$ rules out
    // the largest number of distractors
    **foreach** $p \in s.P$ **do**
        **if** $p_c = nil$ **or**
            $|rulesOut(p, s.C)| > |rulesOut(p_c, s.C)|$
        **then** $p_c \leftarrow p$
        **end**
    **end**
    $node_c \leftarrow$ *createNode*($node, p_c$)
    **if** ($node_c = nil$) **then return** $\emptyset$
    **else return** $\{node_c\}$
}
*createNode*($node, p$) {
    $s \leftarrow node.getState()$
    $C \leftarrow s.C -$ *rulesOut*($p, s.C$)
    $L \leftarrow s.L \cup \{p\}$
    // Extend Description
    **foreach** $r' \in \{r_p | r_p \in$ referents($p$) & $r_p \neq r\}$ **do**
        $node_{r'} \leftarrow$ *makeRefExp*($r'$)
        **if** $node_{r'} = nil$ **then return** $nil$ // failure
        $L \leftarrow L \cup node.getState().L$
    **end**
    **return new** Node($C, L, s.P - \{p\}$)
}
```

4.6 Context-Sensitive GRE

[Krahmer and Theune, 2002] also introduced a number of extensions to the IA: the use of salience weights in order to add a definite article to the description for the most salient entity; contrastive properties in order to add properties which impose a contrast between two entities; and a relational extension, similar in spirit but not in form to that in the IR algorithm described above.

Again, as for the SETS algorithm, the commonality with the IA algorithm surfaces as the reuse of the latter algorithm's *expand* method; only the *createNode* method needs to be rewritten, as in Definition 8. To model this variant in our framework, we introduce the following additional methods (cf. [Krahmer and Theune, 2002]):

- *contrastive* takes a referent r and a property p_i; it checks whether the property under consideration is contrastive.
- *mostSalient* takes a referent r, a set of properties, and a set of distractors; it checks whether every entity in the set of distractors has a lower salience weight than r.

5 Conclusions and Future Work

In the foregoing, we have shown how a number of the most frequently discussed algorithms for the generation of referring expressions can be represented within a common framework. The framework is, we believe, an intuitively appealing one: the process of constructing a referring expression

Definition 8: Context-Sensitive Algorithm

```
$r$ // Referent
*createNode*($node, p$) {
    $s \leftarrow node.getState()$
    $out \leftarrow$ *rulesOut*($p, s.C$)
    $C \leftarrow s.C - out$
    $L \leftarrow s.L$
    **if** ($out \neq \emptyset$ **or** *contrastive*($r, p$)) **then**
        $L \leftarrow L \cup \{p\}$
        **if** $v$ expresses a relation between $r$ and $r'$ **then**
            $node_{r'} \leftarrow$ *makeRefExp*($r'$)
            $L \leftarrow L \cup node_{r'}.getState().L$
        **end**
    **end**
    **if** *mostSalient*($r, L, C$) **then**
        $L \leftarrow L \cup$ { *defArt* }
        // The most salient rules out all distractors:
        $C \leftarrow \emptyset$
    **end**
    **return** new Node($C, L, s.P - \{p\}$)
}
```

is viewed as a search problem, where we effectively build a search space consisting of possible descriptions.

There are three significant advantages to this approach.

First, it allows us to determine what the algorithms have in common. This is particularly interesting in that it allows us to begin to assemble a collection of core functionalities that are usable in a variety of different approaches to GRE. This is apparent not only in terms of the general framework (where, for example, the notions of states and their initialisation, the definition of what it is to be a goal state, and the overall algorithmic pattern) are shared, but in terms of 'helper' routines (such as *rulesOut* and *mostSalient*) which can be modularised out of the essence of different algorithms and reused elsewhere.

Second, it makes it possible to see more clearly what the essential differences between the algorithms really are. In their original forms, these differences are obscured, due to the absence of a common vocabulary for expressing the algorithms; by representing the algorithms within a common framework, it becomes easier to see where the algorithms differ, and where the differences are simply due to differences in notation or presentation. By using the framework of problem solving as search, we have effectively decomposed the algorithms into a number of key elements: a search srategy, represented by the *queuing-method*, and an *expand-method*, which encompasses two aspects of each algorithm: the basic strategy adopted and the particular kinds of referring expressions covered. Furthermore, the *expand-method* decomposes into a general strategy for expansion (as found in, for example, the Full Brevity algorithm and the Incremental Algorithm), and a *createNode* method, which varies depending upon the kind of referring expression targetted.

Third, it allows us to see more clearly the logical space within which the algorithms reside, and to see ways of combining aspects of different algorithms. At its simplest, this is clearest with respect to the kind of search strategy used in the algorithms. Present formulations conflate the choice of search strategy with the other aspects of the algorithm (such as how subsequent nodes in the search space are computed); our approach separates out these different facets of the algorithms, and makes it much easier to see that the choice of search strategy is an independent decision. Consequently, for example, we can easily experiment with a variant of the IR algorithm that uses breadth-first search rather than depth-first search.

So far, we have used the framwework to express the most widely-known algorithms in the literature. Preliminary examination of the algorithms in [Krahmer *et al.*, 2003], [van Deemeter and Krahmer, forthcoming], and [Horacek, 2004] suggests that these will also be expressible within the framework described here. As we capture more algorithms in the framework, our intention is to tease out an inventory of basic constituent elements which can then be reassembled and integrated in different ways, so that we can derive a better understanding of the nature of the problem of referring expression generation.

Acknowledgements

This work was carried out while the first author was a visiting researcher at Macquarie University's Centre for Language Technology; the visit was supported by Macquarie University's Research Development Scheme. We are grateful to the members of the Centre for Language Technology who provided insightful comments on earlier presentations of this work, and to the anonymous IJCAI reviewers for their helpful comments.

References

[Appelt, 1981] D. E. Appelt. *Planning Natural Language Utterances to Satisfy Multiple Goals*. PhD thesis, Stanford University, 1981.

[Bateman, 1999] J. Bateman. Using Aggregation for Selecting Content when generating Referring Expressions. In *Proceedings of the 37th Annual Meeting of the ACL*. University of Marylad, 1999.

[Dale and Haddock, 1991] R. Dale and N. Haddock. Generating referring expressions involving relations. In *Proceedings of the 5th EACL*, pages 161–166, Berlin, Germany, 1991.

[Dale and Reiter, 1995] R. Dale and E. Reiter. Computational interpretations of the gricean maxims in the generation of referring expressions. *Cognitive Science*, 19(2):233–263, 1995.

[Dale, 1989] R. Dale. Cooking up referring expressions. In *Proceedings of the Twenty-Seventh Annual Meeting of the ACL*, pages 68–75, Vancouver, British Columbia, 1989.

[Dale, 2003] R Dale. One-anaphora and the case for discourse-driven referring expression generation. In *Proceedings of the Australasian Language Technology Workshop*. University of Melbourne, 2003.

[Horacek, 1997] H. Horacek. An Algorithm for Generating Referential Descriptions with Flexible Interfaces. In *Proceedings of the 35th Annual Meeting of the ACL*. University of Madrid, 1997.

[Horacek, 2004] H. Horacek. On Referring toSets of Objects Naturally. In H. Bunt and R. Muskens, editors, *Third International Natural Language Generation Conference*, pages 70 – 79. Springer-Verlag Heidelberg, 2004.

[Krahmer and Theune, 2002] E. Krahmer and M. Theune. Efficient context-sensitive generation of referring expressions. In *Information Sharing: Reference and Presupposition in NLG and Interpretation*, pages 223–264. CSLI, 2002.

[Krahmer *et al.*, 2003] E. Krahmer, S. van Erk, and A. Verleg. Graph-based generation of referring expressions. *Computational Linguistics*, 29(1):53–72, 2003.

[McDonald, 1980] D. D. McDonald. *Natural Language Generation as a Process of Decision-making Under Constraints*. PhD thesis, Massachusetts Institute of Technology, 1980.

[Reiter and Dale, 1992] E. Reiter and R. Dale. A fast algorithm for the generation of referring expressions. In *Proceedings of the 14th ACL*, pages 232–238, 1992.

[Russell and Norvig, 2003] S. Russell and P. Norvig. *Artificial Intelligence: A Modern Approach*. Prentice-Hall, Englewood Cliffs, NJ, 2nd edition, 2003.

[Simon and Newell, 1963] H. Simon and A. Newell. *GPS, a program that simulates thought*, pages 279–293. 1963.

[Stone, 2000] M. Stone. On identifying sets. In *Proceedings of the First International Natural Language Generation Conference*. Mitzpe Ramon, 2000.

[van Deemeter and Krahmer, forthcoming] K. van Deemeter and E. Krahmer. Graphs and booleans: On the generation of referring expressions. In H. Bunt and R. Muskens, editors, *Computing Meaning Vol. III*. Dordrecht: Kluwer, forthcoming.

[van Deemter, 2002] K. van Deemter. Generating referring expressions: Boolean extensions of the incremental algorithm. *Computational Linguistics*, 28(1):37–52, 2002.

[Winograd, 1972] T. Winograd. *Understanding Natural Language*. Academic Press, 1972.