gefördert vom
Bundesministerium
für Forschung
und Technologie

Verbmobil
Verbundvorhaben

Lexicon access on parallel machines

Markus Duda

HUB

Report Nr. 10
April 1994

April 1994

Markus Duda

Forschungsgruppe Computerlinguistik
Philosophische Fakultät II
Humboldt-Universität zu Berlin
Unter den Linden 6
10099 Berlin

Tel.: (030) 20192 - 555
e-mail: `Duda@compling.hu-berlin.de`

Gehört zum Antragsabschnitt: 5.7/5.8 Lexikonzugriff/–datenbank

Das diesem Bericht zugrundeliegende Forschungsvorhaben wurde mit Mitteln des Bundesministers für Forschung und Technologie unter dem Förderkennzeichen 01 IV 101 G gefördert. Die Verantwortung für den Inhalt dieser Arbeit liegt bei dem Autor.

Contents

1 Vorwort

Das Lexikon(-modul) ist in einem System zur Sprachverarbeitung eine zentrale Wissensquelle. Dies um so mehr, als unifikationsbasierte Grammatiken, wie z.B. die HPSG größtenteils lexikalisch orientiert sind. Die zentrale Bedeutung eines Lexikonmoduls erhält in einem Sprachverarbeitungssystem eine zusätzliche Perspektive dadurch, daß es nicht nur Wissensquelle für den Parser der Analyse, sondern auch für andere Verarbeitungsmodule wie etwa den Transfer oder die Generierung sein kann.

Will man ein eigenes Lexikonmodul einsetzen, so bedarf es einer äußerst effizienten Realisierung desselben, um tatsächlich aus der neuen Architektur einen Leistungsvorteil zu erhalten. Auf der anderen Seite erschweren komplexe Anforderungen an ein solches Modul die Umsetzung dieses Konzepts. Es fehlt ein eindeutiger Zugriffsschlüssel, etwa die orthografische Repräesntation, selbst eine Menge von Zugriffsschlüsseln für unterschiedliche Anfragetypen (abhängig von dem anfragenden Verarbeitungsmodul) ist apriori nicht vorauszusetzen. Unterspezifikation sowohl in der Anfrage als auch im Lexikoneintrag bzw. der Wunsch, in der Anfrage an das Lexikon enthaltene Merkmale neben einem Schlüssel zur Einschränkung des Suchraumes bzw. der Disambiguierung zu nutzen, können als Standard-Suchsituation betrachtet werden. Insbesondere die Verarbeitung gesprochener Sprache fordert ein effizient und mit hohem Durchsatz arbeitendes Lexikonmodul aufgrund der Menge produzierter Worthypothesen. Die in einem Worterkenner generierten Hypothesen stellt ein Wortnetz in ihrem zeitlichen Verhältnis zueinander dar. Gelingt es, ein solches Wortnetz zu komprimieren, Hypothesenmengen zusammenzufassen, so entsteht im Ergebnis ein reduziertes Wortnetz, welches jedoch mit stark unterspezifizierten Hypothesen besetzt ist. Dies führt zu unterspezifizierten Anfragen an das Lexikon.

Um einen solchen Anforderungskatalog an ein Lexikonmodul zu erfüllen, ist es notwendig, Algorithmen und Strukturen, wie sie im Lexikonmodul zur Darstellung linguistischer Information angewandt werden, an moderne Hardware-Architekturen anzupassen. Der vorliegende Beitrag stellt einen Ansatz vor, der auf der Ausnutzung der Parallelität beruht. Der Autor möchte anhand eines einfachen Beispiels wie dem Lexikonmodul belegen, daß die Ausnutzung paralleler Konzepte die Bewältigung der Komplexität linguistischer Probleme möglich macht.

2 Zusammenfassung

Lexikalische Einträge werden als gerichtete Graphen repräsentiert. Unter der Annahme, daß die für die Suche relevanten Teile dieser Graphen sich auf Bäume mit einer festen Maximaltiefe reduzieren lassen, wird ein Suchalgorithmus angegeben, der eine zu erwartende zeitliche Komplexität, linear zur Anzahl der lexikalischen Einträge, besitzt. Die Kodierung der lexikalischen Eintrage als **vollständige Bäume** erlaubt die theoretisch mögliche Berechnung der Suche mit einer maximalen Anzahl von Prozessoren im Paracomputermodell in einem Schritt.

Ein anderes Modell ergibt sich aus der Zerlegung des einen lexikalischen Eintrag repräsentierenden Baumes in die Menge seiner Pfade. Mit einer Numerierungsvorschrift für Pfade läßt sich nun eine totale Ordnung über alle Pfade aller lexikalischen Einträge definieren, was eine Suche in logarithmischer Zeit ermöglicht.

Auf der Basis der **Pfadzerlegung** und **-numerierung** wird eine Pipeline-Architektur entworfen, die die Suche im Lexikon mit maximalem Durchsatz auf eine gegebene Anzahl von Prozessoren mit dem Ziel optimaler Lastverteilung realisiert.

3 Abstract

To communicate with a computer in spoken language is an unattained challenge of Artificial Intelligence (AI) and Computational Linguistics. To solve such problems linguistic knowledge has to be combined with programming methods of AI and modern computer architectures. We will show how the complexity of linguistic processes can be handled by taking advantage of parallel architectures. In particular, speech systems where most lexicon queries are extremely underspecified suffer from the problem that the access to the lexicon module turns out to be a bottleneck.

We introduce the *search problem* over a given lexicon and compute its time complexity for two different encodings. With the help of a space consuming encoding we define a total order over a lexicon, and, having a total order, logarithmic time becomes valid for the complexity of sequential lexicon search. Next, we will speed up the search by parallelisation, making use of the *paracomputer*. Last, we describe a practical approach to the parallelisation of a lexicon module with the aim to maximize the throughput.

4 Introduction

The most common form of linguistic data representation is the **feature structure**. So, for linguistic applications in computer science the lexicon is an abstract data type over sets of feature structures with minimally one function for the search. The search function selects zero, one or more elements from the lexicon set which fit a search pattern. The example gives the lexicon description of the German word form *Weibern (dative plural form of female):*

$$\left[\begin{array}{ll}
\text{LEX} & : \textit{weib} \\
\text{ORTH} & : \text{"}\textit{Weibern}\text{"} \\
\text{SYNSEM} & : \left[\begin{array}{ll}
\text{LOCAL} & : \left[\begin{array}{ll}
\text{CAT} & : \left[\begin{array}{ll}
\text{HEAD} & : \left[\begin{array}{ll} \text{NFORM} & : \textit{nform} \\ \text{CASE} & : \textit{dat} \end{array}\right] \\
\text{LEX} & : +
\end{array}\right] \\
\text{CONTENT} & : \left[\begin{array}{ll}
\text{INDEX} & : \boxed{1} \left[\begin{array}{ll} \text{NUM} & : \textit{plu} \\ \text{PERS} & : 3 \\ \text{GEND} & : \textit{neut} \end{array}\right] \\
\text{RESTR} & : \boxed{2} \left[\begin{array}{ll} \text{REL} & : \textit{weib} \\ \text{ARG1} & : \boxed{1} \end{array}\right]
\end{array}\right] \\
\text{CONTEXT} & : \boxed{2}
\end{array}\right] \\
\text{NLOC} & : \textit{nonlocal}
\end{array}\right] \\
\text{QSTORE} & : \textit{list}
\end{array}\right]$$

A possible search pattern for this entry is

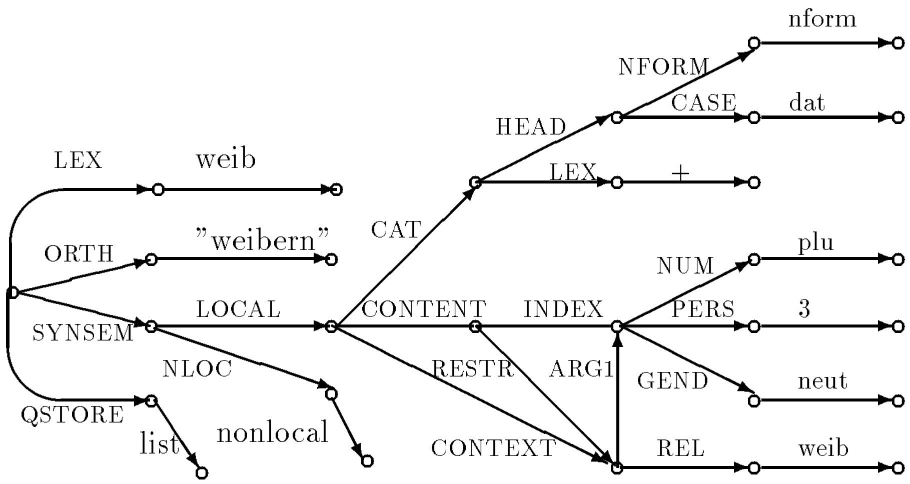
$$\left[\begin{array}{ll}
\text{SYNSEM|LOCAL} & : \left[\begin{array}{ll}
\text{CAT} & : \left[\begin{array}{ll} \text{HEAD|NFORM} & : \textit{nform} \end{array}\right] \\
\text{CONTENT} & : \left[\begin{array}{ll} \text{INDEX|NUM} & : \textit{plu} \\ \text{RESTR|REL} & : \textit{weib} \end{array}\right]
\end{array}\right]
\end{array}\right]$$

A feature structure describes some (linguistic) object by specifying values of various attributes. The value may itself be specified recursively by another feature structure or by an atomic value. We notate feature structures as *attribute value matrices (AVM)* [PS87].

Another way to interpret feature structures is to think of them as constraining the set of (linguistic) objects of the world (the lexicon) they describe. This view is appropriate for the search pattern.

The aim of a lexicon query is to complete the information given by the search feature structure with information from one or more lexical entries. The more constrained the search feature structure is, the less lexicon entries the query returns.

We also can represent feature structures as graphs:



5 The search problem

To search in the lexicon means to test each lexicon entry if it is *subsumed* by the feature structure that describes the search pattern. Since there is no unique node naming between the graph representations of the search pattern and the lexicon entry - they both have different domains - subsumption is not equal to subgraph testing.

Given two feature graphs, say S for a search pattern and L for a lexicon entry, the following procedure defines the *subsumption test:*

procedure **subsume**(S, L):
begin

(i) if S is marked with L, return **TRUE**[1] else mark S with L[2]

(ii) delete the roots of S and L, and isolate the rooted subgraphs $S_1, \ldots, S_p$ and $L_1, \ldots, L_q$.

[1]This rule eliminates cycles.
[2]If we think of numbered nodes, to mark S with L means to mark the root node of S with the number of the root node of L.

(iii) find a homomorphism h from the rooted subgraphs of S to the rooted subgraphs of L wrt. edge labelling. If there is no homomorphism, return **FALSE.**

(iv) performe **subsume**$(S_i, h(S_i))$ for all rooted subgraphs of S.

(v) if all subsume tests of the previous step were successful then return **TRUE**, otherwise return **FALSE**

end

Two properties of feature graphs make this procedure possible:

- every feature graph has a *root node* and
- if there are two edges k and s going from node r which share the same attribute as their label, then k and s are identical.

If the attribute labels are ordered, the homomorphism can be found in linear time. So, the **subsumption** procedure has $O(n)$ time complexity where n is the number of edges of an *acyclic* S. If S contains cycles, the needed time depends on L. A cycle in S with s nodes fits a cycle in L with l nodes after **sl** steps.

Subsumption can be seen as a special case of the subgraph isomorphism testing which is known to be NP-complete [Leu90].

6 Efficient encoding of feature structures

To make the search tractable it is necessary to restrict the graphs representing feature structures. Firstly, we want to focus the search on a few attributes which achieve greater discriminatory effect. Secondly, in most cases it turns out to be sufficient for the search if we think of feature structures as being trees with a fixed depth.

Next, we define appropriateness conditions for each attribute, so we know which edges can emanate from a certain node. As an example, if the labels

a, b, and c are appropriate for a given attribute l[3] then only edges labelled with a, b, or c can emanate nodes reached by l[4].

For the computation of complexity only binary trees of depth d are used. The lexicon has n_l entries.

6.1 Full trees

With the help of these appropriateness definitions, a space-consuming encoding of feature structures can be built as follows:

(i) construct a full tree recursively: take a root node r, for all appropriate labels *label* create edges $(r, i, label)$, take i as the new root node and proceed recursively;

(ii) introduce a boolean flag as the new edge label;

(iii) for any given feature structure copy this full tree and mark an attribute as existing in this feature structure using the boolean flag.

To give an example, for the set of labels $L = \{a,\ b,\ c\}$ and the appropriateness conditions

$$app(a) = b,\ c \qquad\qquad app(b) = a,\ c$$

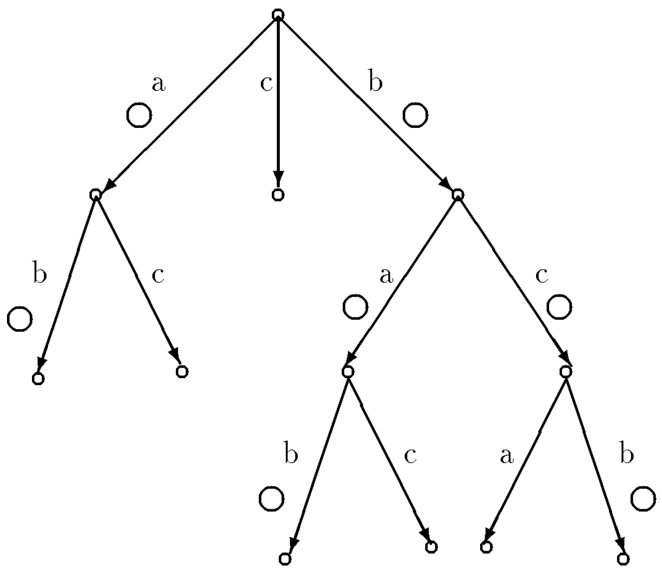
$$app(c) = a,\ b$$

the feature structure

$$\begin{bmatrix} a : b \\ b : \begin{bmatrix} a : b \\ c : b \end{bmatrix} \end{bmatrix}$$

is described by the following *full tree* where the circles mark the edges which are labelled with attributes existing in our feature structure:

[3] l is a label as well.
[4] edges labeled with l



The subsumption procedure over full trees works as follows:

procedure **subsume**(S, L):
begin

(i) delete the roots of the search pattern S and the lexicon entry L

(ii) compare the label array of both roots. If there is at least one flag which is set in the array of S and unset in the array of L, return **FALSE**

(iii) for all activated edges from S perform this procedure recursively

(iv) if all procedure calls from the previous step succeed, return **TRUE**, otherwise return **FALSE**.

end

This procedure solves the problem in $O((2^{d+1} - 2) * n_l)$ steps.

6.2 Path enumeration

Another interesting encoding is the separation of a tree representing a feature structure into all its paths. As an example, if there are 9 possible labels extended with a zero label[5], can be assigned each path a number of base 10:

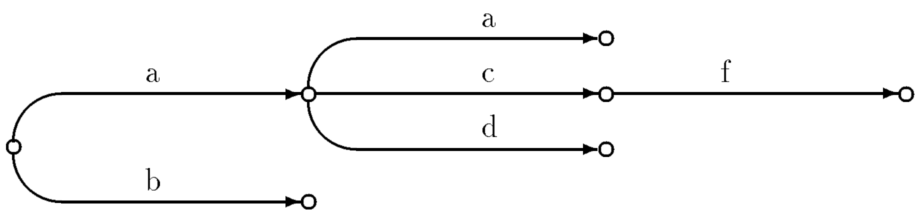
Let

(i) $L = \{a,\ b, \ldots,\ i\}$ be the set of Labels,

(ii) $h_0(l) = \begin{cases} [1..9] & : \quad l \in [a..i] \\ 0 & : \quad l = o \end{cases}$ and

(iii) $h_d()$ the enumeration function for paths of length d.

A possible graph over L is G



P = { aao, ado, acf, boo } is the path representation of G with

$$h_3(P) = \{110,\ 136,\ 140,\ 200\ \}$$

With the presented enumeration we get a total ordered set of all paths of all feature structures over the lexicon. Now, lexicon search can be defined in two steps:

(i) for each path of the search pattern:

- compute its path number,
- find the first element of the lexicon path set[6] which is greater than or equal to the number representing the search path, but less than the sum of the search path number and $base^x$, where x is the position of the last non-zero figure in the search path number,
- find the last element of the lexicon path set which is less than the addition of $base^x$ to the search path number.

[5]The zero label indicates that between two nodes there is no labeled edge defined. *Zero* is used as a filler.

[6]The lexicon path set is a set of numbers

The result of finding these two bounds is an interval, possibly empty, in which the search path subsumes all lexicon paths.

(ii) build the intersection over all the intervals.

Given a search path $p_s = a$, then for our example

(i) $h_3(p_s) = 100,\ x = 2$

(ii) $h_3(p_s) + 10^2 = 200$

(iii) the search interval is $[100,\ 200)$ and

(iv) the resulting subgraph $h_3(P_{p_s})\ is\ \{110,\ 136,\ 140\}$.

Since a full binary tree of depth d has 2^d different paths, the search of intervals for all paths is computed in maximal $2 * log_2(2^d * n_l) * 2^d$ comparisons. After 2^d subtractions the smallest interval is found and after $2^d * |Int_{min}|$ tests the intersection of all intervals results.

In the worst case, computing the intersection takes linear time over the lexicon size. Then the *full tree encoding* leads to better results. But in the average case, the resulting set of the search has cardinality, say, $1 \ldots 10$, and among the search paths there are some very discriminating ones.

With the assumption that in the average case

$$2^{d+1} \ll n_l\ and\ Int_{min} \ll n_l$$

logarithmic time results for the lexicon search with *path encoding*, since the paths are totally ordered.

The *path encoding* yields further advantages:

(i) The number of possible paths $|P|$ can be sized down for large lexicons to $|P| < n_l$.

(ii) The search is divided into two steps. This, if wanted, gives the chance to interrupt the search after creating the intervals and to stay within logarithmic time if the minimal interval Int_{min} is too large.

(iii) After finding one interval, a decision can be taken on whether to continue the search over the whole lexicon or to create a *new ordered path set* over the lexicon elements marked with the interval.

7 On parallel searching

7.1 Hardware requirements

In order to design a parallel architecture for the search and to look for maximum speedup we want to make use of the *paracomputer model with Concurrent Read and Exclusive Write (CREW)* [Sni89].

A paracomputer consists of many identical autonomous processors, each with its own local memory and its own program. In addition, the machine has a shared memory. Each processor can simultaneously in one step read any cell in shared memory and only one processor can exclusively write to a particular cell of shared memory.

The input is given in special input cells and the output in output cells.

7.2 Maximal speedup

To achieve maximal speedup we use the *full tree encoding* of feature structures. With this encoding, a subsumption test can be performed with maximally *O(1)* where we distribute the computation over $2^{d+1} - 2$ processors with d the fixed tree depth if we regard binary trees.

The local memory of each processor consists of:

- the label l of the edge, associated to the processor,
- the relative index of this edge in the full tree,
- the number of a unique output cell which is the same for all processors of one tree and which is initialized with **TRUE**,
- the program for read, comparison, and write

The parallel subsumption can be defined as:

procedure **parallel subsume**:
for i in $[1,\ 2^{k+1}-2]$ parallel do
begin

(i) read input cell i[7]

(ii) if i contains **TRUE** and l contains **FALSE**, try to write **FALSE** to the output cell. If write access is denied, do nothing[8].

end

The possible speedup for subsumption with the full tree encoding is linear.

For n_l lexical entries a maximal performance of *O(1)* is reachable for a parallel system with $n_l * (2^{k+1}-2)$ processors and concurrent read to shared memory. The output is given in n_l output cells.

7.3 Speedup with path enumeration

The search using path enumeration is divided into the search of 2^d intervals and the computation of the resulting intersection.

The speedup for computing the intersection is linear, while the speedup for search is logarithmic. Finding a boundary takes $O(log_{p+1}n_l)$ steps. This was shown by M. Snir[Sni89]. So, O(1) is reached with n_l processors.

Note that the output computed by search with path enumeration can be more compact in comparison to parallel search with full trees.

8 A practical approach

In real applications where different modules use the resource lexicon, the problem is to maximize throughput. Standard parallel architectures like

[7] i contains **TRUE** if this edge is specified in the input tree, otherwise **FALSE**

[8] In this case, another processor does the work

pipeline architecture are designed to maximize throughput.

In chapter 3.2 we introduced the path enumeration. This encoding gives not only an order but divides the search problem into independent subproblems. In data base systems large bases are kept tractable by defining indexes. To apply this method for lexicon search, we will build a set of index paths with the following properties:

(i) thinking of a path as a way through a rooted graph up to node n_d of depth d, n_d has to be a root of a subgraph s which for all lexical entries exists.

(ii) either s is atomic[9], or it is simple to define a total order over all possible s. For a given s there has to be exactly one interval $[s_0, s_1]$ that describes the possible candidates for subsumption.

(iii) the index path leads to a discriminating point w.r.t. subsumption for all lexical entries. n_d is called a decision sensitive point if s strongly selects possible solutions from the set of lexical entries.

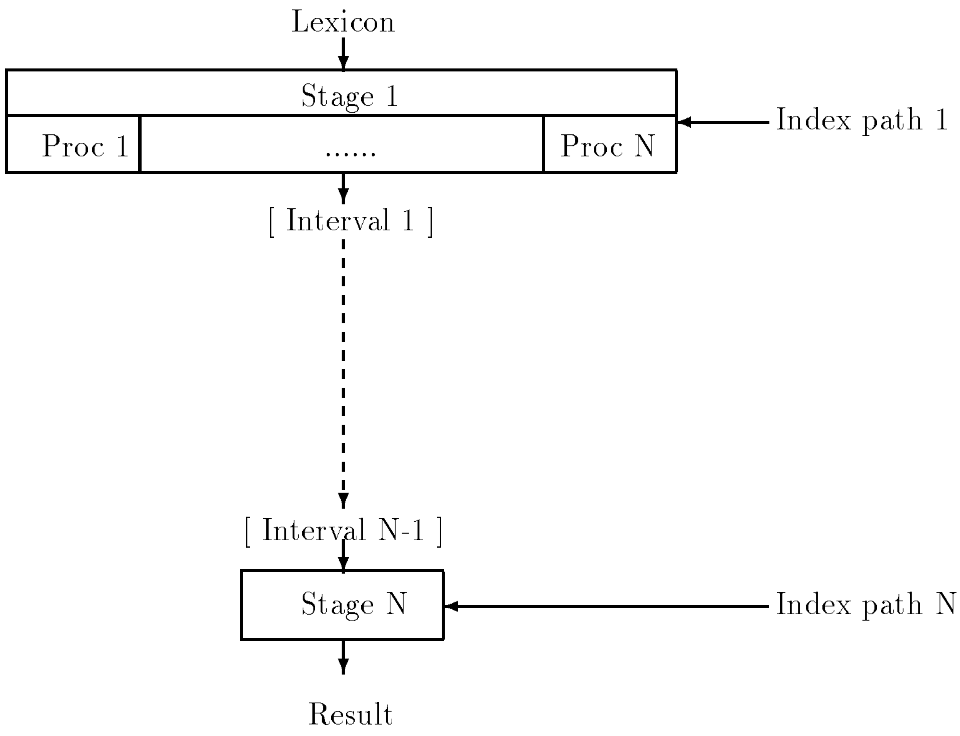
From the set of indexes we assign one index to each stage of a pipeline.

To answer a lexicon request, the search pattern is itself searched for index paths. This results in a dynamically linked pipeline. Each stage selects a subset from its input which consists of possible subsumption candidates w.r.t. to one index path. The following stage takes this subset as input. The output of the last stage represents the resulting set of lexicon entries which the search pattern subsumes. There are as many stages in the pipeline as index paths were found.

So, the search domain gradually decreases. After finishing the selection at the first stage, the next request can start with index search while the previous request is computed in the other stages. To meet the demand of equally distributed process loads, all elements of the pipeline are themselves parallelised. Due to the decreased computational demands at later stages, the first elements of the pipeline need a higher degree of parallelisation. A possible symmetric pipeline architecture with $\frac{n}{2}(n + 1)$ processors is:

---

[9] consists of only one edge



Linguistic and statistical information is needed for successful pipeline design. First we make use of linguistic knowledge to define a proper index system. To equally distribute process loads, statistical experiences and a possibly dynamic assignment from processors to pipeline stages are needed.

9 Conclusions

Lexical search is, with restrictions to the data model, simple to divide into subproblems. Under certain conditions, a space intensive encoding realizes the search problem in logarithmic time. Massive parallel systems can compute a lexicon search in one step. It is shown that parallel design of NLP algorithms can solve certain problems of time complexity in NLP applications.

References

[Kog81] P. Kogge. *The Architecture of Pipelined Computers.* Hemisphere Publishing Corporation, New York, 1981.

[Leu90] J. van Leuwen. *Handbook of Theoretical Computer Science - Algorithms and Complexity.* MIT-Press, Cambridge, 1990.

[PS87] C. Pollard and I. Sag. *Information-based syntax and semantics.* Number 13 in CSLI Lecture notes. CSLI, Stanford, 1987.

[Sni89] Marc Snir. On parallel searching. *SIAM Journal of Computation*, 14:688–708, 1989.

[Tan90] A. Tannenbaum. *Structured computer organization.* Prentice Hall, Englewood Cliffs, 1990.