Customizing Information Capture and Access

DANIELA RUS
Dartmouth College
and
DEVIKA SUBRAMANIAN
Rice University

This article presents a customizable architecture for software agents that capture and access information in large, heterogeneous, distributed electronic repositories. The key idea is to exploit underlying *structure* at various levels of granularity to build high-level indices with task-specific interpretations. Information agents construct such indices and are configured as a network of reusable modules called structure detectors and segmenters. We illustrate our architecture with the design and implementation of smart information filters in two contexts: retrieving stock market data from Internet newsgroups and retrieving technical reports from Internet FTP sites.

Categories and Subject Descriptors: H.3.3 [**Information Storage and Retrieval**]: Information Search and Retrieval—*selection process*; H.3.4 [**Information Storage and Retrieval**]: Systems and Software—*current awareness systems*; *information networks*

General Terms: Design, Documentation, Experimentation

Additional Key Words and Phrases: Information gathering, software agents, table recognition

1. INTRODUCTION

The proliferation of information in electronic form and the development of high-speed networking make the problem of locating and retrieving infor-

D. Rus has been supported by the Advanced Research Projects Agency of the U.S. Defense Department under ONR contract N00014-92-J-1989, by ONR contract N00014-92-J-39, and by NSF contract IRI-9006137. D. Subramanian has been supported by NSF contract IRI-8902721. This work was also supported in part by the Advanced Research Projects Agency under grant no. MDA972-92-J-1029 with the Corporation for National Research Initiatives (CNRI). Its content does not necessarily reflect the position or the policy of the U.S. Government or CNRI, and no official endorsement should be inferred.
Authors' addresses: D. Rus, Department of Computer Science, Dartmouth College, Hanover, NH 03755; email: rus@cs.dartmouth.edu; D. Subramanian, Department of Computer Science, Rice University, Houston, TX 77005; email: devika@cs.rice.edu.

mation in vast electronic environments one of the grand challenges of computer science. Examples of electronic corpora include repositories of newspapers and technical reports, data from high-energy physics experiments, weather satellite data, and audio and video recordings. Examples of tasks that query and manipulate these electronic collections include content-based retrieval of technical reports, access of documents via citations, summaries of stock prices from archives, and retrieval of temporal weather patterns from a weather database. The goal of *information capture and access* (ICA) is to organize and filter electronic corpora guided by user-specified tasks. We organize data by acquiring partial models that associate task-level content with information, which in turn facilitates location and retrieval. Our research goal in this article is to develop methods for solving the ICA problem and to provide a computational paradigm for *customizing* this process in heterogeneous, distributed repositories.

A diverse collection of tools has been developed for information capture and access. Information capture tools like Gopher and the World Wide Web (WWW) provide hierarchical and networked organization of data, but they require substantial manual effort to build and maintain. Information access over Gopher and WWW is via manually hard-wired hyperlinks and keyword search. In addition, existing automated search engines take little advantage either of nontextual cues in electronic data or of other underlying structures. Consider the task of finding papers on cognitive theories of practice substantiated by human data. A word-based query involving the keywords cognitive, theory, practice, and data produces hundreds of papers, only a small fraction of which are actually relevant. The problem is that in a database of psychology papers these keywords occur frequently in varying contexts. Thus, the precision value for the search is low. A better automatic filter can be constructed by exploiting knowledge about conventions of data representation. For example, the fact that experimental data are generally presented in tables and graphs in technical papers can be used to substantially reduce the number of potentially relevant matches to the query. Cues like tables and graphs complement the textual content of documents. In addition, they generalize to other types of media (like video). Information retrieval tools need both textual and "structural" cues for query formulation, as well as for information capture and access.

We propose an approach to ICA that relies on *structural* cues to construct data indices at the appropriate level of granularity. We call this approach *structure-based information capture and access*.[1] The term structure refers to any pattern that is at a level of abstraction higher than the basic unit of data representation (e.g., characters and pixels). Tables, figures, lists, paragraphs, and sections are standardized layout-based abstractions for documents. Theorems, lemmas, examples, and counterexamples are content-based abstractions. These structures have evolved as conventions for organizing documents and can be naturally exploited as filters to select

[1]Conventional word-based systems like Smart [Salton and McGill 1983] are also structure-based ICA systems. However, they exploit only one type of structure in the data.

relevant data. In general, high-level structures are not immediately available, and computation is needed to reveal them.

We advance the theory that such structure is a natural basis for modularizing information search computations. In this article, we provide a framework for synthesizing *customized* information processing engines by assembling prefabricated modules that reveal and detect structure in data.

These prefabricated structure-detecting and revealing modules can be implemented as static programs. For distributed corpora, we claim that it is more advantageous to organize them as *transportable software agents*. Transportable agents can travel from machine to machine, processing information locally, thus avoiding costly data transfers over congested networks. Agents, in addition to being transportable, can learn and can operate autonomously.

We draw inspiration from robotics [Brooks 1986; 1990; Donald et al. 1993; 1995] to design information agents. The basic modules for physical robots are sensors and effectors. Our information agents are autonomous sensori-computational "circuits" comprised of a network of *virtual sensors* to detect, extract, and interpret structure in data and *virtual effectors* for transportation. This article describes our agent architecture and the sensory modules required for processing information. These modules fit on a transportable platform called *Agent-Tcl*[2] [Gray 1995; 1996; Rus et al. 1997].

Two types of sensori-computational modules are necessary for ICA: (1) structure detectors, which efficiently decide whether a block of data has a specified property (for example, "is this block of text a table?"), and (2) segmenters, which partition the data into blocks that are appropriate for the detectors. The modules are efficient, reliable in the presence of uncertainties in data interpretation and extraction, and fault tolerant. The modules are capable of tuning their performance parameters to the environment and task at hand.

We illustrate our approach to ICA with an example developed in full detail in the rest of the article. Consider the task of finding precision-recall measures for a specific collection (such as the CACM) [Cohen 1993] from a scanned archive of technical reports. Using knowledge that precision-recall numbers are contained in articles on information retrieval and are frequently displayed in tables, we first filter the information retrieval papers from the electronic archive.[3] We then extract tables from these papers with an efficient table detector and then interpret the extracted tables to find the precision-recall measures. Figure 1 shows a *zoomed-out* view of a paper on information retrieval that highlights the presence of tables. The requested measure is then obtained by *zooming-in* the tables, to examine their contents. In our approach, we select tables from the pages of the

[2]This is a transportable version of Tcl/Tk [Gray 1995] that allows programs to suspend execution and move to a different machine running an Agent-Tcl server. The current capabilities include process migration, message passing, and communication.

[3]We use the information retrieval system Smart [Allen and Salton 1993] to cluster on this topic.

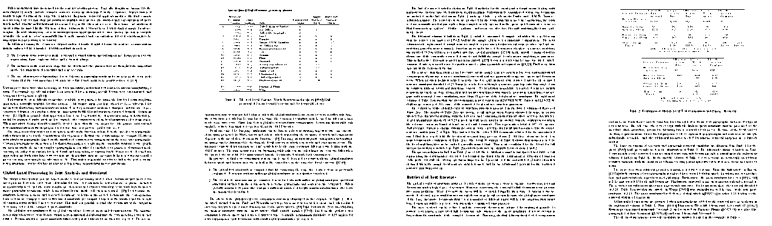

Fig. 1. A zoomed-out view of an article on information retrieval.

article using simple geometric computations. We then analyze the tables at the level of words to obtain the desired information.

We will refer to sensori-computational modules, such as the table detector above, as smart filters. We will show how smart filters can be composed to form customizable search engines. Throughout this article, we will refer to the result of the composition interchangeably as information agents and as search engines [Rus and Subramanian 1993].

1.1 Related Work

The proposal by Kahn and Cerf [1988] for organizing architectures for retrieving information from electronic repositories provides the context for the problem addressed in this article. We are inspired by research in several distinct areas: information retrieval and filtering, automated document structuring, agents, information visualization, and mobile robotics.

Information Retrieval and Filtering. We can interpret the classical work [Salton and Buckley 1990; Salton and McGill 1983] in information retrieval (IR) as follows: the data environment is text; the unit of structure is typically a word; the detectors constructed in the IR literature are pattern matchers over words; the segmenters are word indexes over documents. We propose augmenting the set of recognizable structures to include higher-level entities: tables, graphs, pictures, time histories of patterns, etc. This permits us to handle heterogeneous forms of information (numbers, weather maps, simulation data).

Work in information filtering (IF) can be viewed as a complement to the work on information retrieval, where the focus of research is in characterizing the information in an environment that is relevant to a user. Techniques developed in the IF community for unobtrusive user modeling in multimedia environments, and calculation of the information that can be ignored based on query specifications, are important in the design of information agents.

Automated Document Structuring. The goals of the document-structuring community are to identify the key constituents of a document image (sections, paragraphs, pictures, etc.) from its layout and to represent the logical relationship between these constituents. This is a first step toward analyzing the content of documents.

Document structuring is usually done in two phases. In the first phase, the location of the blocks on the page is determined. In the second phase, the blocks are classified, and the logical layout of the document is calculated. Previous work on block segmentation of documents include Jain and Bhattacharjee [1992], Nagy et al. [1992], and Wang and Srihari [1989]. The methods are developed in the context of very specific applications; segmenting pages of technical journals [Nagy et al. 1992; Wang and Srihari 1989] and locating address blocks in letters [Jain and Bhattacharjee 1992]. The methods (the run-length smoothing algorithm [Wong et al. 1982] and the recursive XY cuts algorithm [Nagy et al. 1992]) use area-based techniques; that is, every pixel in the document is examined at least once in the process of generating the segmentation. We draw on this work to define parametric segmenters for electronic document retrieval.

Previous work on classifying and logically relating blocks includes Tsujimoto and Asada [1992], Rus and Summers [1995], Fujisawa et al. [1992], Nagy et al. [1992], Wong et al. [1982], Mizuno et al. [1991], and Wang and Srihari [1989]. Methods for discriminating between text blocks and pictures in bitmapped images are presented in Wang and Srihari [1989], Fujisawa et al. [1992], Tsujimoto and Asada [1992], and Nagy et al. [1992]. A language for representing the hierarchical structure of documents is given in Fujisawa et al. [1992]. Tsujimoto and Asada [1992] and Mizuno et al. [1991] extract the logical structure by a bottom-up approach that starts with the finest units of structure and then computes aggregates.

Agents. There has been a recent flurry of activity in the area of designing intelligent, task-directed agents that live and act within realistic software environments. These are called knowbots by Kahn and Cerf [1988], transportable agents by Rus et al. [1997], softbots by Etzioni and Weld [1994], sodabots by Kautz et al. [1994], software agents by Genesereth and Ketchpel [1994], personal assistants by Maes [1994] and Mitchell et al. [1994], and information agents by Rus and Subramanian [1993]. A knowledge-level specification of knowbots is provided by Kahn and Cerf [1988, chap. 4], who outline the required capabilities without committing to a specific implementation. The modular information agent architecture, organized around the idea of structure that is presented here, can be treated as a specific implementation proposal for this specification. In common with Kahn and Cerf, our initial applications are in the retrieval of documents for which a user may only be able to specify an imprecise description. Our agent architecture can also be used to set up Personal Library Systems [Kahn and Cerf 1988] "that selectively view, organize, and update contents" of an electronic library for individual use.

We are interested in the same class of tasks as Etzioni and Weld [1994], Kautz et al. [1994], Maes [1994], and Mitchell et al. [1994]. Etzioni and Weld synthesize agents that are Unix shell scripts by using classical AI planning techniques. The focus of Mitchell et al.'s and Maes' work is in the interaction between users and agents. They propose using statistical and machine-learning methods for building user models to control the agent

actions. Genesereth and Ketchpel [1994] propose a declarative agent communication language.

Information Visualization. The GUI community has developed a range of visual methods for interacting with large information sets such as CACM [Cohen 1993]. The work most relevant to our project is that of the information visualization group at Xerox Parc [Robertson et al. 1993]. They have developed an integrated architecture that allows a user to browse through a rich information space and visualize the results of retrievals in interesting three-dimensional views (e.g., walkthroughs in 3D rooms). The major use of structure in their work is to summarize and present information to a user.

Mobile Robotics. The analogy between mobile robots in unstructured physical environments and information agents in rich multimedia data environments is not just metaphorical. We have observed [Brooks 1986; 1990] that the lessons learned in designing task-directed mobile robots can be imported to the problem of information capture and access. We were influenced in defining topology-based segmenters and structure detectors by the work of Donald [1995] and Donald et al. [1993; 1995], who consider the problem of determining the information requirements to perform robot tasks using the concept of information invariants and perceptual equivalence classes.

2. ORGANIZING PRINCIPLES FOR INFORMATION-GATHERING SEARCH ENGINES

For any specific information access query, no matter how complicated, we can write a special-purpose program on top of existing tools to search for the answer. However, there is such great variety in the information landscape that it is impractical to provide a special search program for each possible query. This is because building each one from scratch takes a considerable amount of time and expertise. We are not arguing against the use of search engines specialized for specific query classes. We propose speeding up the process of building specialized search engines for classes of queries by recycling the computations they perform. This requires the recognition and reuse of significant computational modules; in particular, we need (1) a basis for the modularization and (2) schemes for combining the modules into complete search engines.

The basis for the modularization is structure at multiple levels of granularity in the electronic repository. By structure we mean any regularity or pattern present in the data. Word counts as used in traditional information retrieval are examples of statistical structure, and tables or graphs are examples of geometric structures. Sometimes structure is not apparent in raw data. In such cases, modules that segment and transform data to reveal underlying structure can lead to effective search algorithms.

We show that it is possible to synthesize special-purpose search algorithms by combining simple modules for detecting and filtering structure.

Each module can be viewed as an operation in a calculus of search. The set of all modules together with the combination operators defines a high-level language for specifying search engines.

The desirable properties of segmenting and structure-detecting modules are

—*Reliability*: Detectors and segmenters that rely on perfect data interpretation perform poorly. Consider for instance the design of a structure detector for tables. The general organization of a table is in rows and columns, but the actual table content and layout differ with each example. Tables with partially misaligned or missing records should be detected with a specified degree of accuracy and confidence. In this article, we design structure detectors and segmenters to be robust in the presence of uncertainty in data representation and to have bounds of accuracy for data interpretation.

—*Efficiency*: The objective of information gathering is to find good answers in response to a user's query, as fast as possible. The data environment is so large that neither structure detectors nor segmenters can afford to look at every data item, so it is important to design these units efficiently. Our measures of efficiency are tuned to specific applications. We prove bounds on efficiency (typically time and space bounds) as a function of performance accuracy. For instance, we prove that our segmenter has complexity linear in the perimeter of the regions defined by a white space border of width d in a document (rather than in the area of these regions).

—*Error detection and recovery*: Modules should be designed to recover from errors in the segmentation and interpretation of data. Additionally, since our data are in a distributed, wide-area network, it is important that they detect and recover from network failures. This is typically accomplished by having each module be self-correcting—a lesson about organization of complex systems [Brooks 1986; 1990] that was discovered in the context of mobile robotics and insect intelligences. Our designs incorporate task-specific error detection and recovery schemes.

A complex search is organized as a network of on-line computations. Each node in the network corresponds to a module; modules interact by sharing data. Our information agents (1) are transportable across machines, (2) are customizable to a user specification, and (3) have autonomy in decision making.

In the remainder of this section we discuss details of designing modules (segmenters and detectors) and organizing them as information search engines.

2.1 Segmenters

Segmenters partition data at the appropriate grain size for structure detectors. The table detector expects information to be broken up into

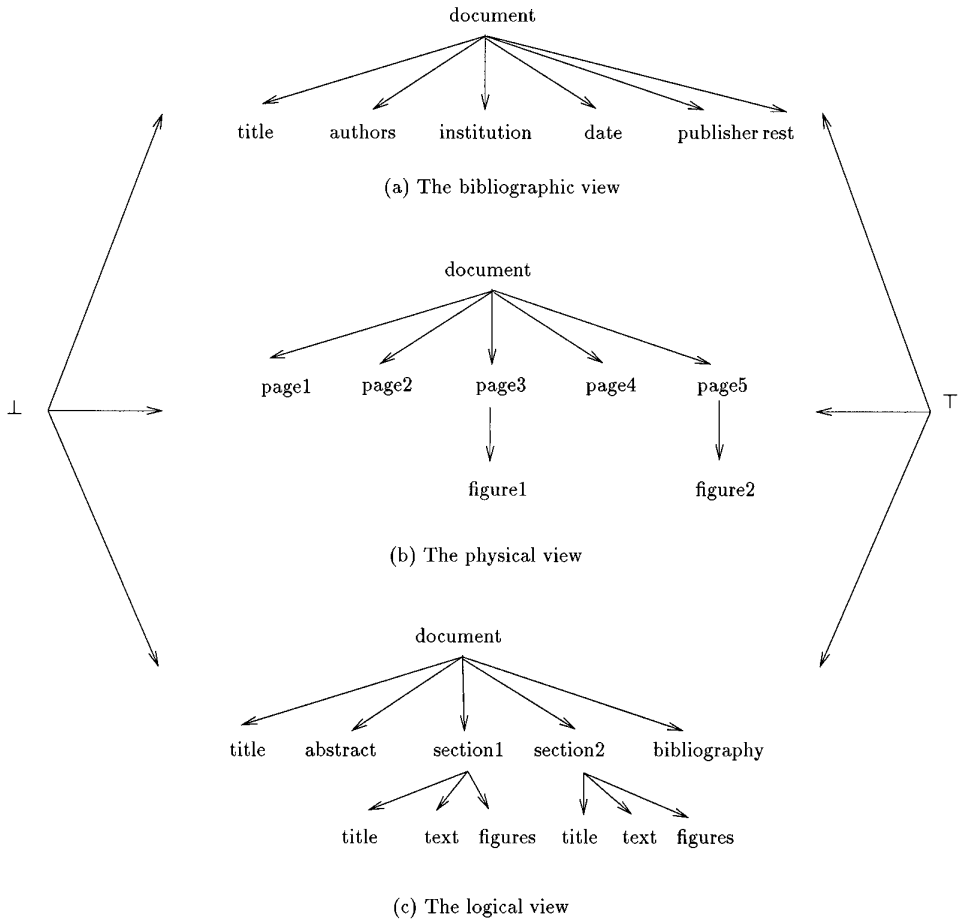

Fig. 2. The lattice of syntactic topologies of an electronic document. The bibliographic view contains the information in a bibliographic record. The physical view captures the page layout of the document. The logical view captures the relation between the pieces of the document. There are also semantic topologies, with various conceptual views.

pieces at the paragraph level; this is generated by a paragraph segmenter described later in this section.

We model granularity shifts in the descriptions of the data using concepts from topology [Munkres 1975]. A topology over a set S is a set of subsets of S that are collectively exhaustive—the union of the subsets yields S. A topology is closed under the operations of union and finite intersection. The coarsest description of a data collection W, where the only distinction made is between W and the rest of the world, is called a trivial topology and consists of two subsets of W—W itself and the null set $\emptyset$.

Definition 2.1.1. A segmenter n is a function that takes a topology of the world W and produces another topology of W, n: $\tau_1(W) \rightarrow \tau_2(W)$.

Each segmenter generates a view of the document (see Figure 2). A

special class of segmenters produce finer partitions of a given view of a document. Let τ_1 be a partition of the document in Figure 1 into a title, abstract, sections, and a bibliography. Let τ_2 partition the same article into a title, abstract, section 1, . . . , section n, and a bibliography as shown in Figure 2(c). Since τ_2 distinguishes sections unlike τ_1, τ_2 is a refinement of τ_1.

Definition 2.1.2 (*Refinement*). Let τ_1 and τ_2 be two topologies over a set S. If $t_1 \in \tau_1 \quad t_2 \in \tau_2 \; t_1 \subset t_2$ then τ_2 is a refinement of τ_1.

A topology τ_2 that satisfies Definition 2.1.2 makes finer distinctions in the data than the topology τ_1. There can be many distinct refinements of a topology.

A refinement segmenter computes a specific topological refinement of the data to aid the detection of structure at a certain grain size. A segmenter that extracts individual sections in an article can provide them to a structure detector that filters sections with certain properties (e.g., sections containing definitions). The set $\mathcal{T}$ of all topologies of a data collection W is computed from a set $\mathcal{N}$ of segmenters as a finite (functional) composition of elements of $\mathcal{N}$. Since the refinement relation between topologies is a partial order, $\mathcal{T}$ is a lattice. The top element of the lattice $\mathcal{T}$ is the trivial topology of W. The bottom element $\perp$ is a topology consisting of singleton subsets containing the elements of W. This topology cannot be refined any further. The join operation $\vee$ on the lattice takes topologies $\tau_1 \in \mathcal{T}$ and $\tau_2 \in \mathcal{T}$ and produces a topology $\tau \in \mathcal{T}$ such that τ_1 and τ_2 are refinements of τ. The meet operation $\wedge$ takes topologies $\tau_1 \in \mathcal{T}$ and $\tau_2 \in \mathcal{T}$ and generates $\tau \in \mathcal{T}$ such that τ is a refinement of both τ_1 and τ_2. The lattice $\mathcal{T}$ of topologies of the data environment reveals structures at different levels of detail.

The abstract specification of a segmenter allows us to characterize its behavior independent of its implementation. Modeling the world as a lattice of topologies generated by segmenters provides computational support for the idea of generating successive refinements that zoom-in to the required information.

We distinguish between logical and physical segmenters. A segmenter that extracts sections of an article for table detection generates a logical segmentation of the data environment. In contrast, agents like Netfind [Schwartz and Tsirigotis 1991] employ segmenters that partition the nodes in the Internet into relevant and irrelevant sets.

We illustrate the idea of segmenters with a general algorithm that can partition two-dimensional documents with arbitrary layout. A special case of the algorithm presented here has been implemented and tested in the context of the Cornell technical report collection [Rus and Summers 1995]. The segmenter in Rus and Summers [1995] automatically synthesizes a logical view of a document by analyzing the geometry of the white spaces in the left and right margins.

2.1.1 *An Example: Segmenting Documents.* Given a pixel array of a document, the segmenter's goal is to partition the document into regions

that capture its layout. This problem has two parts: determining where the regions are and classifying them according to their layout structure. Examples of regions are titles, text blocks, pictures, tables, captions, etc. In what follows, we present an algorithm for finding specific regions of layout structure.

Definition 2.1.1.1. Let B be a polygonal partition of the document. A border of width d is the set $Border\ (B, d) = B \oplus S_d^1 - B$. Every element of $Border\ (B, d)$ is a white space.[4]

A document is treated as a pixel array. Intuitively, a border of white space of width d exists around a polygonal region B in the document if we can roll a coin of diameter d around the region. We define an inclusion relation between polygonal partitions S_1, S_2 in a document; in particular we say that S_1 is included in S_2, denoted $S_1 \subset S_2$, if every pixel in S_1 is also a pixel of S_2.

Definition 2.1.1.2. A layout region B relative to a border of width d in a document D is a document partition for which $B \cup Border\ (B, d) \subset D$, and there is no region $B' \subset B$ with this property.

This definition identifies borders around both convex and nonconvex document partitions. It cannot, however, identify rings of white space contained entirely within a region. Note that the geometric definition of a border is parameterized on d, the width of the border. It allows us to construct a hierarchical model of the document layout without relying on a predefined set of document features like sections, captions, etc. The levels of the hierarchy are defined by different values of d. That is, we can partition a document into regions at different grain sizes according to the border of white space that can be drawn around them. For example, if the task is to partition the front page of *The New York Times* into blocks, values of $d > 0.2$ inches extract the entire title, while values of $d < 0.1$ inches separate each character of the title into its own block. If we are given a set of characteristic d values for a document, we can segment that document into regions for each d value. The region topology generated with d_1 is a refinement of the region topology generated with d_2 if $d_1 < d_2$. Thus a collection of d values defines a totally ordered set of region topologies for a document. The coarsest element in the set consists of one region: the entire document. The ith element of the set contains regions relative to a border width of d_i. The $(i + 1)$th element is constructed recursively by refining the regions found in the ith partition with a (smaller) d value of d_{i+1}. Each topology is computed by rolling coins of successively smaller sizes through the maze of white spaces in a document.

The block segmentation algorithm in Figure 3 finds regions by detecting borders of width d. It traces the perimeters of identified regions. Its computational complexity is $O(p)$, where p is the number of pixels that do

[4] $A \oplus B = \{a + b \mid a \in A, b \in B\}$ is the Minkowski sum of sets A and B. S_d^1 is a circle of diameter d.

Notation:
$C = \{\langle v, b\rangle\}$ denotes the critical vertex list
where v denotes an (x, y) location in the pixel array
and b is a sweep direction
(1 for horizontal and -1 for vertical).
R denotes the list of vertices for the new regions.
O denotes the list of vertices for the input region.
d denotes the border width.
p is the pixel array for region O.
Input: d, O, p.

Initialization: $C = (\langle o, 1\rangle)$, for some $o \in O$.
$R = O$.
While $C \neq \emptyset$ **do**
begin
$\langle v, b\rangle = pop(C)$;
sweep p from v in direction b looking for
runs of white space r longer than d bounded by vertices v_1 and v_2.
For each v_i that has not been visited along b,
$C = C \cup \{\langle v_i, -b\rangle\}, R = R \cup \{v_i\}$.
end

Fig. 3. The perimeter tracer for the block segmentation algorithm for a given border width d. The regions are determined by connecting vertices found by the algorithm.

not occur in any of the identified regions. For each region, the algorithm examines a number of pixels linear in the perimeter of the region, rather than its area. For dense documents, like lead pages of a newspaper, this is a significant reduction in complexity.

PROPOSITION 2.1.1.3. *The perimeter-tracing algorithm identifies polygonal partitions that are unions of axis-parallel rectangular regions with borders of width d as specified in Definition 2.1.1.2. For a pixel array of size* $m \times n$, *it identifies regions by examining no more than* $O(p)$ *pixels, where* p *is the number of pixels that occur outside identified regions.*

The restriction to unions of axis-parallel rectangles is due to the sweeping strategy employed by the algorithm. The size d of the border width has to be chosen carefully for the algorithm to perform well, i.e., for the number p of pixels examined to be significantly smaller than mn. With a border width equal to the intercharacter spacing, the algorithm examines every pixel, similar to existing area-based methods, and generates the finest region topology of the document.

How reliably does the algorithm identify meaningful blocks in a scanned document? The accuracy of the block partitions is a function of the d values made available to the block segmenter. Significant d values, denoting the width of white spaces between logical units like paragraphs and sections, extract partitions of the document at the paragraph and section level respectively. These spacings can be provided by a user or estimated by the system using random sampling of regions of the document. The correctness of our block segmentation scheme relies on regularities in the environment of documents—in particular, the fact that documents are typeset with some

standard conventions. Most documents have a rectangular layout produced with a finite set of fonts. Each font size has a characteristic spacing between lines and characters. Our algorithm relies on the following generic typesetting rules. A superset of these conventions are found in Nagy et al. [1992].

(1) Printed lines are roughly horizontal.
(2) The base lines of characters are aligned.
(3) Word spaces are larger than character spaces.
(4) Paragraphs are separated by wider spaces than lines within a paragraph, or by indentation.
(5) Illustrations are confined to rectangular frames.

Suppose the interparagraph spacing is d_p, and we supply the algorithm with a d value of $d_p \pm \epsilon_p$, where $0 \leq \epsilon_p \leq d_p$. How likely are we to extract paragraphs in the scanned document using the block segmenter? The answer to this question is determined entirely by the data and not the algorithm. If d values in the interval $[d_p - \epsilon_p, d_p + \epsilon_p]$ are not associated with other "natural" logical units of the document, our algorithm will correctly produce a partition of the document at the paragraph level. The algorithm is robust in environments where significant d values are spaced more than ϵ_p apart. A more formal statement of this intuitive analysis of robustness requires the formulation of layout models of scanned documents. A formal robustness analysis is presented in the next subsection for a structure detector that operates on paragraph-level regions in an ASCII document.

The algorithm, as presented in Figure 3, assumes that there is no noise in the data and that we can reliably detect borders. In particular, on any sweep line, horizontal or vertical, we assume that we can find clean runs of white pixels of length larger than d. In reality, runs of white pixels are polluted by black pixels that occur at random locations. For instance, letters like f, g, j, p, q, and dots on i's protrude into the white space around a region. We associate a tolerance parameter ϵ_d, with every d value. ϵ_d represents the number of black pixels that can be ignored in the detection of a run of white pixels of length d. The tolerance ϵ_d should be chosen directly proportional to the value of d. Metaphorically speaking, we treat each coin of size d as a bulldozer that can push ϵ_d or fewer black pixels out of the way.[5] In our implementation of the block segmenter, we experimentally determined these values for documents drawn from the scanned technical report archive at Cornell.

2.2 Structure Detectors

Structure detectors are programs that can decide whether a block of data has a specified property P. An example is the property of being a table or a graph for a block of text.

[5]We thank Jim Davis for this idea.

```
100a Introduction to Computer Programming 4    Lec1 TR    9:05 Ives 120
        Wagner                                 Lec2 TR   11:15 Ives 120
100b Introduction to Computer Programming 4    Lec1 TR    9:05 Olin 255
        Van Loan                               Lec2 TR   11:15 Ives 110
101  The Computer Age                     3         TR    1:25 Upson B17
        Segre
102  Intro Microcomputer Applications     3    Lec1 TR   10:10 RR 105
        Hillman (Ag. Engr.)                    Lec2 TR   12:20 RR 105
108  A Taste of UNIX and C                1(4wk)    MWF   3:35 Olin 255
        Glade                             su
```

Fig. 4. A schedule of the introductory computer science courses.

Definition 2.2.1. A structure detector is a computable function $s : \tau(W) \rightarrow 2^{\tau(w)}$ defined on a topology $\tau(W)$ to a *discrete* subset $t(W)$ of that topology, such that $t(W)$ has the property P.

A structure detector s for a property P is *complete* if it finds all the subsets of $\tau(W)$ that satisfy P. A structure detector s for property P is *robust* if whenever it recognizes $t(W)$, it recognizes all its ϵ-perturbations.

2.2.1 *An Example: Detecting Tables.* *Webster's Seventh Dictionary* defines a table as a "*systematic arrangement of data usually in rows and columns for ready reference.*" Implicit in this definition is a *layout* component and a *lexical* component: the data are organized in columns of similar information. Consider the structure in Figure 4: the records are two lines long; the columns in the second line of a record do not align with the columns in the first; some columns extend into adjacent ones; and there are lexical irregularities in its records. In spite of these imperfections, the layout and lexical structures are clear, and we identify the structure as a table. Our goal is to create a structure detector that checks for column and content structure while tolerating irregularities to within specified error bounds.

The measure for the column layout of a block of text is given in terms of a data structure called the *white space density graph* and is denoted by WDG. Let B be a block of text of n rows and m columns and $w : \{c \mid c$ is a character$\} \rightarrow \{0, 1\}$ with w (*space*) $= 1$ and $c \neq space$, $w(c) = 0$.

Definition 2.2.1.1 (*Vertical Structure*). The *white space density graph* of B is the polygonal line WDG : $[0, m] \rightarrow [0, 1]$ defined by the points $\mathrm{WDG}(i) = 1/n \sum_{j=0}^{n} w(B_{i,j})$, $0 \leq i \leq m$.

Figure 6 shows the WDG associated with the table in Figure 4.

Definition 2.2.1.2 (*Deviations in Vertical Structure*). Given an error tolerance ϵ_v, a block of text has *column structure* if it occurs between two successive local maxima above $1 - \epsilon_v$ in the WDG.

Each local maximum is a candidate column separator. A candidate column is a real table column only when it has corresponding horizontal lexical structure. We are far from being able to identify row structure based on semantic content, but semantic uniformity in rows is highly correlated with

lexical uniformity. We exploit this correlation in the design of a table detector that is robust in the presence of layout imperfections.

The process of discerning lexical structure is facilitated by the presence of nonalphabetic characters. For example, it is easy to recognize that the entries of the sixth column in Figure 4 represent similar information, since they have a very regular and distinct lexical pattern. In distinguishing lexical structure, we identify the following equivalence classes of characters: alphabetic, numeric, and special (each special character is in its own class). Let $c_0, c_1, \ldots c_n$ denote the columns of a table. We use regular expressions for generalizing the contents of a column. In Figure 4, all items in the sixth column (the meeting times) can be described by the following conjunctive regular expression $NN : NN$, where N is a symbol denoting a number and where the colon (:) is special character. The *lexical description* of a column c_i is a nontrivial regular expression r_i that describes the smallest possible language that includes all elements of c_i. The regular expression $r_1 + \ldots + r_n$ is a *trivial* generalization of a given set of the elements $r_1, \ldots, r_n$; otherwise it is *nontrivial*.

Definition 2.2.1.3 (*Horizontal Structure*). Consider the columns $c_1 \ldots c_n$ of a block of text satisfying Definition 2.2.1.2, and consider the lexical descriptions $r_1 \ldots r_n$ of these columns. This text also has *row structure* if and only if the language described by $r_1, r_2 \ldots r_n$ is nonempty.

Now consider Figure 4, which has small irregularities in the lexical structure of the columns. To express this more rigorously, let M be a metric for string comparison (we use the Levenshtein metric [Sankoff and Kruskal 1983]). Given $\epsilon > 0$, two strings a and b are ϵ-similar if $M(a, b) \leq \epsilon_h$. We use ϵ_h-typings, defined below, of the regular expressions that correspond to the entries of a column in order to control the imperfections we allow in horizontal structure.

Definition 2.2.1.4 (*Deviations in Horizontal Structure*). Given $\epsilon_h > 0$ and a set of strings, an ϵ_h-*typing* is a division of the set into disjoint subsets such that any two strings in the same subset are ϵ_h-similar.

Lexical typing for a table is done in two parts. Each candidate column is analyzed to determine a regular expression for its type. The alphabet of types is generated by ϵ_h-typing the column elements. The lexical type of the table is obtained by computing the minimum regular expression over the column types. The data in the list associated with each column are typed by grouping all the elements of the column, that are at least ϵ_h-similar (for a given ϵ_h), into an equivalence class. This step in the algorithm allows for the occurrence of multiline records in a table and for ϵ_h tolerance in the record units. A minimal ϵ_h-typing partitions the elements of the column in the coarsest possible way.

Figure 5 describes the table detection algorithm. An example application of this algorithm to the table in Figure 4 follows. The first step in the algorithm is to create the WDG associated with the table, by calculating the percentage of blank spaces in each column of the block. Figure 6 shows

Input: A text block B, vertical tolerance ϵ_v, horizontal tolerance ϵ_h
Output: *true* if B is a table and *false* otherwise.

1. Form the WDG of the text block.
2. Find column separators of height $1 - \epsilon_v$ in the WDG. If none exist, the text is not a table, so quit; otherwise continue.
3. Perform an ϵ_h-typing on the regular expression representations of the entries of each column.
4. Find the lexical structure these typings imply, if any. If such a structure is found, the text is a table; otherwise, it is not a table.

Fig. 5. The table detection algorithm.

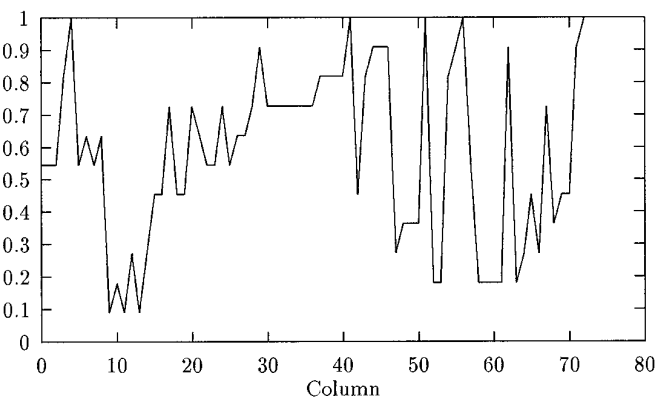

Fig. 6. A white space density graph for Figure 4.

the WDG associated with the text in Figure 4. The second step is to look for column separators, i.e., peaks in the graph of height at least $1 - \epsilon_v$. This graph has six high peaks that are associated with the rivers of white space flowing between the seven columns in Figure 4. In the third step, each candidate column is analyzed for lexical structure. If the column description patterns can be combined into a regular expression across the entire table, the block of text is a table; otherwise it is not.

We now analyze the robustness of the algorithm. Computing peaks in the WDG is quite easy. What is not obvious is how to determine a reasonable threshold value ϵ_h that robustly and efficiently filters tables from basic text. We measure efficiency as the cost of the actual computation and the probability that base text is passed through unnecessary lexical analysis. One approach is to require the user to specify the value of ϵ_h using his or her knowledge about the data environment. Another approach is to have the algorithm statistically learn the value of ϵ_h by analyzing the WDG of tables identified by the user. A third solution does not rely on user assistance, but rather makes use of a probabilistic analysis of WDGs of basic text. The question we ask is "for a high peak value in the WDG, what

is the probability that it corresponds to a true table column rather than a random distribution of spaces in basic text?" From this analysis, we extract a tolerance parameter that can be used as an absolute lower bound on ϵ_h for detecting tables with irregularities in layout.

The analysis makes the following assumptions:

—The average word length that occurs in text is known. For the English language, Kucera and Francis [1967] have determined that the average word length of distinct words is 8.1 characters, but of word occurrences in written text, it is 4.7 characters. For simplicity, we assume that in basic text the average word length is 4 characters.

—The blank spaces in base text are distributed independently. This is due to the fact that the lengths of words and of the spacing between them are variable, and their occurrences in a line of text are random. We have tested the independence of the distribution of white space by extensive experiments with *Splus* [Statistical Sciences 1991]. This implies that the blank spaces of a line have a binomial distribution.[6]

Let B be a block of text of n rows and m columns. Let p be the probability that a character c in a row is blank, and let $q = 1 - p$ be the probability that the character is nonblank.[7] Let WDG be the white space density graph for B. Denote by WDG(k) the value for the kth column of B. Application of Chebyshev's theorem[8] yields the following:

COROLLARY 2.2.1.5. *If the absolute value of a peak in the* WDG *is greater than* $np + h\sqrt{npq}$, *the probability that the peak is an occurrence of basic text is* $1/h^2$.

In other words, by setting the peak threshold to $np + h\sqrt{npq}$, we ensure that with probability $1/h^2$ the presence of any value above the threshold is a candidate column separator in a table. The user can specify required confidence in identifying columns ($1/h^2$), and we can calculate the peak threshold, since n, p, q, and h are all known.

A block of text is a table if it has both vertical and horizontal structure. We now consider the complexity of lexical component analysis.

PROPOSITION 2.2.1.6. *Finding the minimum* ϵ_h*-typing is NP-complete.*

PROOF. Reduction to partitions into cliques. □

Even though finding the minimum typing is a hard problem, a useful ϵ_h-typing can be found efficiently. An element is placed in a partition only if it is $\epsilon_h/2$-similar to the original element of that partition; when an

[6]An interesting problem is to determine when the binomial distribution approaches a normal distribution; at present, we have no solution to this problem.

[7]An average word length of four characters yields $p = 0.2$ and $q = 0.8$.

[8]For any distribution with standard deviation σ, at least a fraction $1 - (1/h^2)$ of the measurements differ from the mean by amounts at most $h\sigma$.

$$\begin{bmatrix} t_1 & t_2 & t_4 & t_6 & t_8 & t_9 & t_{10} \\ t_0 & t_3 & t_0 & t_6 & t_8 & t_9 & t_{10} \\ t_1 & t_2 & t_4 & t_6 & t_8 & t_9 & t_{10} \\ t_0 & t_3 & t_0 & t_6 & t_8 & t_9 & t_{10} \\ t_1 & t_2 & t_4 & t_0 & t_8 & t_9 & t_{10} \\ t_0 & t_3 & t_0 & t_0 & t_0 & t_0 & t_0 \\ t_1 & t_2 & t_4 & t_6 & t_8 & t_9 & t_{10} \\ t_0 & t_2 & t_0 & t_6 & t_8 & t_9 & t_{10} \\ t_1 & t_2 & t_4 & t_6 & t_8 & t_9 & t_{10} \\ t_0 & t_3 & t_5 & t_0 & t_0 & t_0 & t_0 \end{bmatrix}$$

Fig. 7. Type matrix for the course table in Figure 4.

element is encountered for which no such partition exists, it becomes the original element of a new partition.

The types of the entries of each column are assembled into the *type matrix*. An $m \times n$ type matrix is constructed for a block of text of m lines and n columns. If $t_{ij} = t_{i'j'}$ the data in row i and column j and the data in row i' and column j' are ϵ-similar. The type matrix for the table in Figure 4 is given in Figure 7. A GCD algorithm [Blum and Kozen 1978] can be used to determine the type, if any, of the overall matrix and thus to decide whether the matrix represents a table. We provide for error tolerance in the typing of each column by supplying an error parameter ϵ_r. This parameter specifies the amount of "noise" in the pattern that defines the type of a column. For example, if $\epsilon_r = 0.2$, the type of the fifth column in Figure 7 is taken to be t_8; the two entries that are labeled as t_0 are treated as noise.

We have implemented this table detector that is robust with respect to layout imperfections and used it to build search engines for retrieval tasks whose answers are found in tabular form.

2.2.2 *Experiments with the Table Detector.* We have tested the performance of the table detector on several thousand articles culled from a number of Usenet news groups[9] over a period of a few weeks. Each article was partitioned into paragraphs and then filtered through the table detector. The results were compared with human labeling (as tables and nontables) of the same data. From a subset of the data consisting of messages in comp.biz.hardware, 711 segments were analyzed. The table detector correctly identified 147 out of the 151 tables labeled by a human. The parameter setting was $\epsilon_v = 10\%$, $\epsilon_h = 30\%$, and $\epsilon_r = 50\%$. The results of our experiments yielded 2.65% false negatives and 2.95% false positives. For perfect tables (i.e., tables with perfect vertical alignment and with uniform lexical structure) the recognition accuracy is 100%.

[9]These groups include clari.tw.stocks, biz.comp.hardware, biz.jobs.offers, comp.benchmarks, alt.internet.news, and bionet.journals.contents.

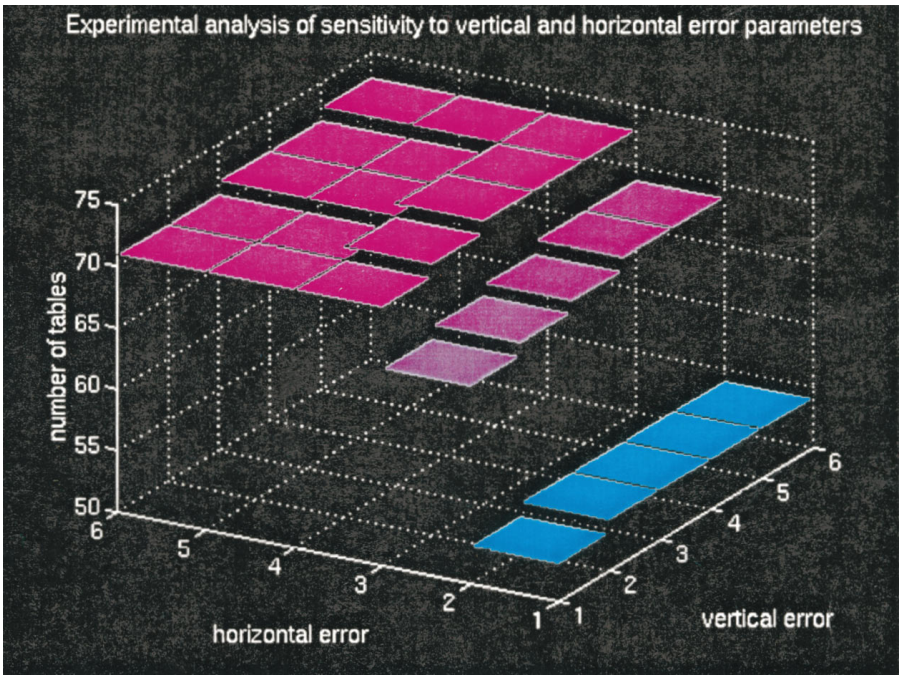

Fig. 8. The dependency of the performance of the table detector on ϵ_v and ϵ_h. The performance numbers are doubly encoded color and height: bright colors and high values along the x-axis denote good performance.

False negatives occur for imperfect tables when there is noise in the vertical alignment greater than the specified error limit, or when records (rows) are nonuniform. Also, false negatives occur when tables are embedded so that the text around them bias the vertical and/or the horizontal structure of the block. Tables with multiline headers where the ratio of the size of the header to that of the table exceeds the horizontal error tolerance are also misidentified. False positives occur when there is accidental alignment of blanks in a paragraph of text. This happens more frequently for short paragraphs (less than five lines).

We studied the sensitivity of the table detector on a data set consisting of 74 tables of very different formats. We counted the number of tables detected by varying the vertical noise ϵ_v from 5% to 25% in increments of 5% and by varying the horizontal noise ϵ_h from 20% to 40% in increments of 5%. The results are plotted in Figure 8. The algorithm is much more sensitive to values of ϵ_v than of ϵ_h. The worst performance over the parameter range we studied was for $\epsilon_v = 5\%$ and $\epsilon_h = 20\%$: 53 out of 74 tables were recognized. The best performance was for $\epsilon_v \geq 15\%$: 73 out of 74 tables were recognized. The one table that was not detected had very irregular lexical content. Figure 9 shows the table that was not recognized, and Figure 10 shows a complicated table that was recognized.

```
Tape Backups:
  Conner/Maynard Int. 250Mb, QIC-80,1 FREE DC2120 tape          -> $170
  Colorado Internal 250Mb, QIC-80                               -> $180
  DC-2120 120/250Mb Tapes                                       -> $22
```

Fig. 9. An example of a figure with irregular lexical content that is not recognized by the table detector within the parameter range we studied. The first column consists of the entire text to the left of the "→". The lexical analysis failed on the first column due to the presence of many mixed special characters. If the text consisting of "250Mb, QIC-80" had lined up, the table would have been recognized. This mismatch problem is exacerbated by the fact that the table is short.

```
 256k cache for all 486 boards                                  Add $25
  486DX2-66, 64k cache,   ""    ""  (Clock Doubler) (Intel CPU) -> $580
Local Bus Upgrades:
  VESA local bus ISA boards (2 VESA/ISA slots, 6 ISA slots)     Add $10
  486 Motherboard w/  NO CPU, 256k Cache                        -> $105
  486 VESA Local Bus Motherboard w/ NO CPU, 2 VLB, 256k Cache   -> $120
  486 VLB w/NO CPU, 256k Cache, 2 VLB, ZIF, Pentium upgrdbl     -> $150
  486 VLB w/NO CPU, 256k Cache, 2 VLB, ZIF,US dsgnd,Pntm upgrdbl-> $160
Misc:
  486 CPU Heat Sink Fan (Clips on to any 486 CPU)               -> $15
```

Fig. 10. An example of a table with irregular lexical content and misalignment that was recognized by the table detector. This table has 20% nonwhite space in the first and last column separators, and so a parameter setting with $\epsilon_v \geq 20\%$ is successful.

2.3 Assembly of Information Agents

Users assemble search engines from task specifications. In this section, we discuss how simple agents are constructed from available detectors and segmenters and how complex agents can be built from simple ones. To synthesize an information agent for a given task, the user

(1) identifies a set of structures at different levels of detail that are relevant to the task and chooses (or creates) detectors that can recognize these structures,
(2) chooses segmenters that partition data at the right granularity for each detector,
(3) composes the agent from these segmenter/detector pairs, and
(4) interprets the computed data.

For example, to find precision-recall measures for the CACM collection [Cohen 1993] from a given article, the designer uses knowledge that the answer to the query is found in tabular form. Since tables need to be identified, the relevant structure detector is the one introduced in Section 2.2.1. The segmenter that partitions the environment for processing by the table recognizer is the block segmenter in Section 2.1.1 with a border width parameter that filters paragraphs in the environment. Once tables are identified, they need to be further refined into rows by the block segmenter

with the appropriate border width parameter. We compose the two segmenter/detector pairs (table level and row level) in series to extract rows of identified tables in the environment. Finally, the contents of each row have to be interpreted to obtain the precision-recall measures. This last step requires knowledge about the form of precision-recall measures.

2.3.1 *Formalizing Agents.* We represent an agent in the language of *circuits* [Balcázar et al. 1988]. Asynchronous circuits are a useful engineering metaphor for assembling agents from detectors and segmenters. We draw upon the rich formal history of circuit theory to provide principled specifications of the computations that agents perform.

Definition 2.3.1.1. An asynchronous agent circuit $A = (V, E)$ is a directed acyclic graph representation of a computation. The elements of V denote structure-detecting or segmenting operations, and the edges in E denote data paths between the nodes. Each path in the graph is an alternating sequence of segmenters and structure detectors.

We provide circuit descriptions for the modules introduced in the previous section. A structure detector is both a filter over data (e.g., a table detector picks out blocks that are tables) and a boolean function (e.g., a table detector checks whether a block is a table). The definition of a structure detector (Definition 2.2.1) specifies it as a predicate which can be viewed extensionally as the subset of a set or intensionally as a boolean function over a set. We rely upon the correspondence between these two representations of a predicate to define structure detectors as circuits.

Each structure detector (see Figure 11(a)) is composed of a combinational circuit that computes the predicate (denoted by the circle) and a latch (denoted by the square) that filters the data. The input to the structure detector is a stream of data. Both the latch and the combinational circuit receive the data at the same time. The combinational circuit checks whether the block of data has some property and uses this value as a toggle to release the data as output.

Complex detectors can be built from simple ones by using boolean operations of $\wedge$, $\vee$, and $\neg$ on the predicate components of the individual modules. Figures 11(b) and (c) show the composition operations.

A segmenter can also be described as a data-driven, asynchronous circuit (see Figure 11(d)). A segmenter is composed of a combinational circuit that computes a predicate, a data shredder, and a latch. The computed predicate determines where to place the "hyperplanes" for shredding the data. The shredder physically partitions the data into units; this partition is converted into a data stream by the latch. Unlike structure detectors, we do not consider direct compositions of segmenters. This is because we view segmenters as preprocessors of data for detectors, and we prefer to compose segmenter/detector pairs instead of composing segmenters. Thus, for example, if we wanted to segment a file into individual lines with two segmenter modules S1 and S2 where S1 segments a file into paragraphs, and S2

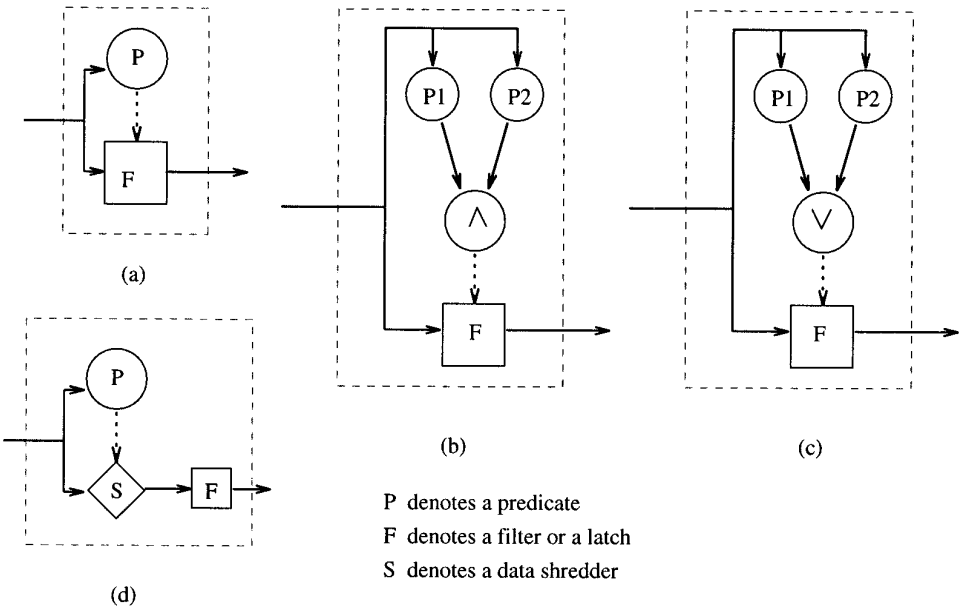

Fig. 11. Circuits for structure detectors and segmenters.

segments paragraphs into lines, we construct the path S1-id-S2 in the agent graph, where id is the identity structure detector.

Structure-detecting and segmenting circuits are assembled into agent circuits by synthesizing a data flow graph. A detector and a segmenter can be connected in series in any order to form a simple agent. For this composition to make sense, we need the output topology of the first circuit in the series to match the input topology of the second circuit. For example, a table detector should be connected to a segmenter that partitions a document into blocks at the paragraph level. The constraints on matching topologies are type theoretic and define minimal conditions for correct composition. We call the topology-matching constraints *calibration* constraints. Calibration constraints ensure that data at the right granularity pass between the components.

Definition 2.3.1.2 (*Simple Agent*). A simple agent is constructed from a segmenter $S : \tau_1(W) \rightarrow \tau_2(W)$ or a detector $D : \tau(W) \rightarrow 2^{\tau(W)}$ connected in series, denoted $S \cdot D$, such that the calibration constraint $S(\tau_1(W)) = \tau(W)$ holds. The series composition $D \cdot S$ can be constructed provided $\tau_1(W) = D(\tau(W))$.

This simple agent takes a partition $\tau_1(W)$ of the data and generates a subset of a refinement of $\tau_1(W)$ that satisfies s. We define two composition schemes for agents and identify calibration constraints on each scheme.

Definition 2.3.1.3 (*Serial Composition*). An agent $a_1 : \tau_{in1}(W) \rightarrow \tau_{out1}(W)$ can be serially composed with an agent $a_2 : \tau_{in2}(W) \rightarrow \tau_{out2}(W)$ in that

order yielding a new agent $a:\tau_{in1}(W) \rightarrow \tau_{out2}(W)$ constructed from the functional composition of a_1 and a_2, provided the calibration constraint $\tau_{out1}(W) = \tau_{in2}(W)$ holds. We also require that the composition respect the alternation constraint between segmenters and detectors. We denote the composition $a_1 \cdot a_2$.

Definition 2.3.1.4 (*Parallel Composition*). An agent $a_1:\tau_{in1}(W) \rightarrow \tau_{out1}(W)$ can be composed in parallel with an agent $a_2:\tau_{in2}(W) \rightarrow \tau_{out2}(W)$, yielding an agent $a:\tau_{in1}(W) \rightarrow \tau_{out1}(W) \times \tau_{out2}(W)$, provided the calibration constraint $\tau_{in1}(W) = \tau_{in2}(W)$ holds. We denote the composition $a_1 \parallel a_2$.

If a_1 and a_2 are simple agents, then the above operation constitutes sharing of a segmenter. Parallel composition of two simple agents allows for different recognizers to operate on the same data partition. For instance, a table detector and a detector for recognizing graphs in text both employ paragraph-level partitions.

2.3.2 *Relating Task and Environment Structure.* The design of appropriate structure detectors and segmenters for a task relies on underlying assumptions about conventions for representing information in the data environment. Consider the task of designing an agent to help protein crystal growers access the latest information about the primary amino-acid sequence, solubility, and molecular weight of the protein they wish to crystallize. Such knowledge is available in databases (their number is growing rapidly) accessible by tools like Gopher and Mosaic. There are a number of programs available at various Internet sites to compute properties of the protein, e.g., the isoelectric point from knowledge of the primary amino-acid sequence. At this time, human crystal growers manually compile information available from these databases and perform conversions to get the data in a uniform framework to run protein simulation programs to get all the computed properties. This is a tedious task that can be automated with the construction of a search engine with knowledge about the forms in which this information occurs and with methods for recognizing and extracting it.

What structure detectors and segmenters are needed for this task? How do we systematically relate the computations they perform to the specific information that needs to be extracted? The designer uses knowledge that some biology databases store information in a relational format and in which others store them as an association list of property-value pairs. The segmenters for this engine partition the data environment into the known databases and within each database recognize and partition the data into a relational table or an association list (a-list). The parametric a-list segmenter has knowledge about association lists and how to recognize ϵ-perturbations of a-lists. So rigid formatting constraints on data are not needed for the proper recognition. By layering the segmenter/detector pairs appropriately, the designer ensures that the right task-specific components are extracted.

2.3.3 *Data Interpretation.* By data interpretation, we mean attaching meaning in task-specific terms to the computations being performed by each component. Interpretation need not, and in fact generally is not, done by the search engine. The designer ensures that the search engine maintains an invariant—the mapping between the results of computations (recall these are topologies) and the "meaning" of the extracted data. In the Stock Filter, discussed in the next section, the designer establishes a mapping between the structure extracted by the table detector and a "stock table"—the latter is a task-specific category, the former a geometric object. The mapping constitutes the interpretation of items in the rows as companies and stock prices and constitutes the items in the columns as high, low, and closing values. The designer incorporates checks in each agent to ensure the integrity of this mapping between results of computations performed by the agent and their task-specific interpretations. There are two implementation choices for data interpretation. It can be procedurally encoded by the designer in structure detectors. Alternatively, interpretation constraints can be declaratively encoded and interpreted at run-time for the "parsing" or "filtering" of data. The tradeoffs between these two extremes in implementation choices can be made using established methods for analyzing interpreted versus compiled code. While declarative encodings usually permit more flexible interpretation of data, compiled schemes are generally more efficient. Further examples of data interpretation are discussed in the context of the Stock Filter and the Bib Filter, described in the following sections.

3. EXAMPLE 1: COMPILING REPORTS FROM TABULAR DATA

Consider the task of compiling a stock report for AT&T for a given period of time using an electronically archived collection of *The New York Times*. For each day in the given time interval, we can structure the task as follows:

(1) We partition the paper into sections using the segmenter in Section 2.1 with the border parameter that characterizes sections in *The New York Times*.
(2) We filter the business section from the paper (this is where the stock data are most likely to occur) by using a structure detector for section titles.
(3) We partition the business section into blocks at the paragraph level, using the segmenter in Section 2.1 with the border parameter that characterizes paragraphs.
(4) Since stock data in *The New York Times* are represented in tabular form, or in graphical form, we select the tables and graphs using the table detector and the graph detector.
(5) We zoom into each table and graph to extract the specific information on AT&T.

We have implemented a smart filter that performs this operation on data coming through newsgroups (see Figure 12). The first-level segmenter is a

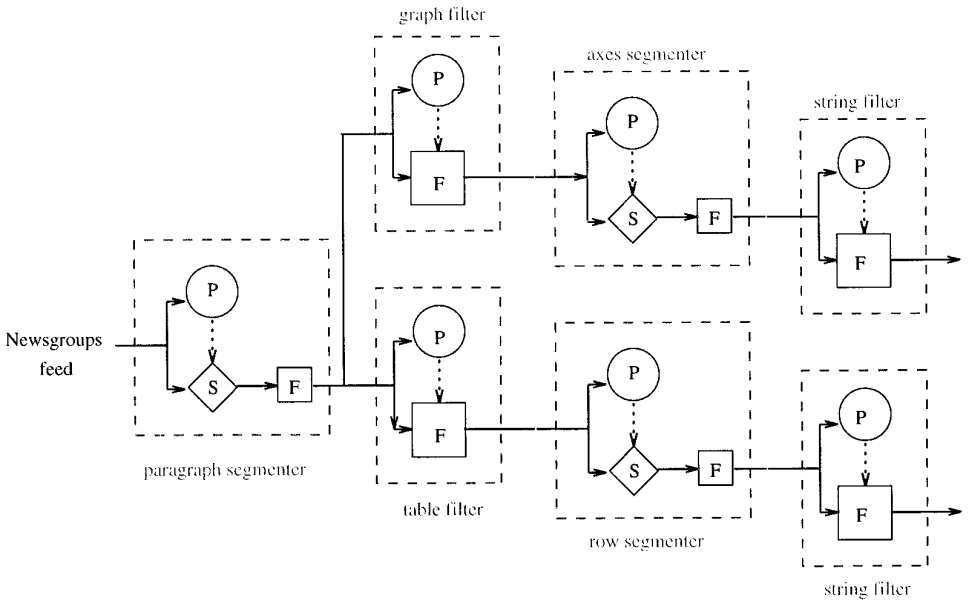

Fig. 12. The circuit representation of a filter for compiling stock reports.

block segmenter that takes postings, one at a time, and produces its paragraph-level partition. Two structure detectors follow it. One filters out the tables, and the other filters out the graphs. The subset of paragraph blocks that are recognized as tables is provided to a row segmenter, in order to separate the records. The rows are individually passed to a string filter to identify the AT&T records.

The data interpretation phase is complex. The filter makes no assumptions about the format of the stock data other than that it is a table. The fields for the high, low, and closing value have to be identified and extracted from the record retrieved by the string comparator. If there is a header, the interpreter scans it looking for the keyword "close" and determines the location of the closing value in the record. If there is no header, the interpreter uses other information about the format of the table. It uses the fact that stock market tables usually contain three columns corresponding to high, low, and closing values and that the closing value is between high and low. This information, which can be encoded as rules, can be used to identify the location of the closing column.

We have implemented the Stock Filter for the data domain of Internet newsgroups. Typical messages consist of a combination of prose and tables that vary in format. This search engine extracts tables from a given list of business newsgroups (an example of an extracted table is shown in Figure 13) which are then searched for records labeled "ATT." An extracted record is interpreted by a procedure attached to the table detector. The procedure contains rules that define the High, Low, and Closing columns on a table of stock data. Figure 14 shows the X-Window user interface for this filter, and sample results for running this filter are given in Figure 15.

Issue	Notes	Dividend	P-E Ratio	Volume (100s)	High /Ask	Low /Bid	Close /Current	Change	Age
3Com	0			8200	34.00	33.13	33.25	-0.50	
AST	0			3905	15.00	14.00	14.00	-0.75	
ATT	N	1.32	20	13493	58.75	57.50	58.50	1.25	
Adaptec	0			9713	26.38	25.50	25.75	-0.63	
AdobeS	0	0.32		4622	45.00	43.00	43.38	-0.38	
AdvMicr	N		9	6039	23.38	22.75	23.25	0.13	
Aldus	0			1095	19.00	18.25	18.75	-0.13	
Alias	0			284	9.75	9.25	9.38	-0.13	

Fig. 13. A data segment of stock quotations extracted by the table detector from the newsgroup clari.tw.stocks (March 18, 1993).

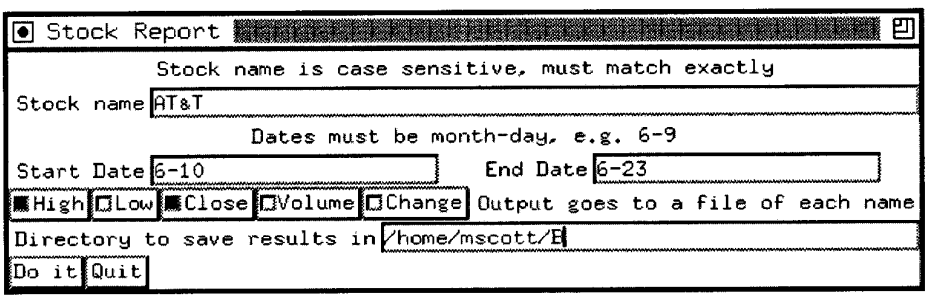

Fig. 14. The user interface of the stock filter.

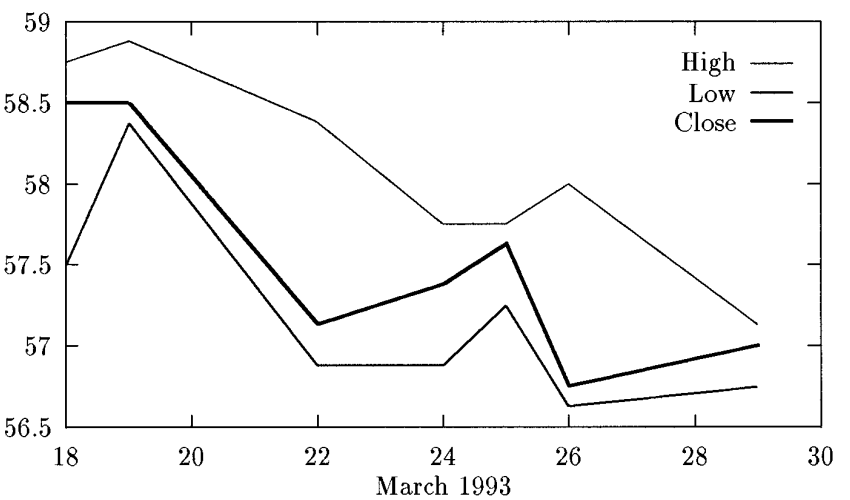

Fig. 15. Compilation of a stock report for AT&T over time, using the stock filter in the environment of newsgroups.

We note that the Stock Filter can be used to retrieve any other type of information that is present in tabular form. In particular, we have instantiated the design in Figure 12 for the task of retrieving precision-recall measures for specific collections from our database of scanned technical

Evaluation Parameters	3.Restricted Sentence Match	4.Output With Text Sections	5.Full Text Section or Paragraphs
Retrieved Documents all queries	1106	1204	1345
Recall after 5 docs	0.4583	0.4834	0.5043
Recall after 15 docs	0.6667	0.7141	0.7631
Precision after 5 docs	0.7689	0.7867	0.8067
Precision after 15 docs	0.5022	0.5274	0.5519
11-point Average Precision	0.6920	0.7520 +9.7%	0.8198 +18.5%

Fig. 16. An extracted table from an information retrieval article.

reports on information retrieval. For the paper shown in Figure 1, a block segmenter detects paragraph blocks, and the table detector extracts the table in Figure 16. The three-line record of the table is further processed by string comparators to extract the actual measures. The same search engine can be used for any other retrieval task whose answer exists in tabular form, using appropriate data interpretation procedures.

4. EXAMPLE 2: AUTOMATIC INFORMATION GATHERING WITH CUSTOMIZED INDICES

With increasing amounts of heterogeneous distributed data, there is a need for tools that can autonomously obtain information with minimal human intervention. The complexity of the problem is in finding the location of a document in a huge and unorganized world. It is impractical to look at every server in the world for every single query. Our idea is that once we have invested the effort to collect information relevant to a specific query, we remember the list of sites found and use it to perform a selective search when asked another query from the same class.

For example, suppose you are interested in books written in French; you would need to know the list of bookstores (physical and electronic) that carry French books. A roll call of bookstores would cluster the relevant sources. When you need a book in French, you would begin by checking with the stores in the computed cluster. Subsequently, when bookstores carrying French books open or close, you would want a way of adding or deleting them from the cluster. Our goal is to provide a facility for creating and maintaining clusters of information sources customized for specific query classes.

We have designed and implemented a search engine that actively finds and orders information sources containing technical reports about a given

topic.[10] The engine is customizable by each user. For each topic of interest, a user invokes the engine to compute a cluster of relevant sites. Each node in the cluster consists of a location and a tree of paths to directories within that site. The first time around, every site is examined to construct the initial list. In our previous example, every bookstore has to be polled to find out if it carries French books. Once constructed, the cluster is used to efficiently compute specific queries about the given topic. For example, given a cluster with sites that contain papers on the query class "robotics," a specific query like the most recent version of the paper by Rodney Brooks on Cog can be answered by restricting the search to the given sites. Since the information landscape changes, the cluster is not a static entity. Updates to the cluster involve sites as well as paths within a site. We consider two methods for keeping the cluster current. The *lazy* method updates the cluster only when the answer to the specific query is not found in the current list. The *eager* method automatically adds new relevant servers as they become available. Both methods work incrementally.

We call the cluster of sites and paths at sites a customized index for a given query class. The customized index comprises part of the internal state of such an agent. It drives the physical navigation of the agent through the Internet. The index prunes away a large fraction of irrelevant sites. The cost for the pruning is paid by the first query in that class and is amortized over all subsequent queries. Of course, this amortization is not valid if data changes so frequently as to make the index out of date as soon as it is created.

Bib Agent is built on top of the Alex [Cate 1992] file system which provides users transparent access to files located in all the FTP sites over the world. Bib Agent can thus use Unix commands like cd and ls to navigate to the directories and subdirectories accessible by anonymous FTP. A structure detector collects the contents of each directory (using the ls Unix utility and a filter) and selects a subset of directories for the next set of navigators. These directories are processed in parallel. The Bib Agent has knowledge about the structure of anonymous FTP directories—for instance, it consults directories named pub, papers, or users but not bin. This task-specific knowledge is used to navigate autonomously as far as possible. At each node, the Bib Agent estimates the cost of its exploration by examining the number and size of its files and subdirectories. If the cost is above a threshold, the agent provides the user with the opportunity to help prune and prioritize its options.[11] If the user responds, the agent remembers the user preferences and uses them in the future.

The directory tree at each site is traversed in a breadth-first manner. For each encountered file, Bib Agent establishes a type by examining the name and selects appropriate searching and displaying routines for that type

[10]In the current implementation, queries are specified by listing keywords. Our selective search approach can be coupled with more sophisticated methods for query specifications. For instance, we can use the table detector for specifications involving tables.
[11]In our present implementation, a query window (see Figure 17) is displayed.

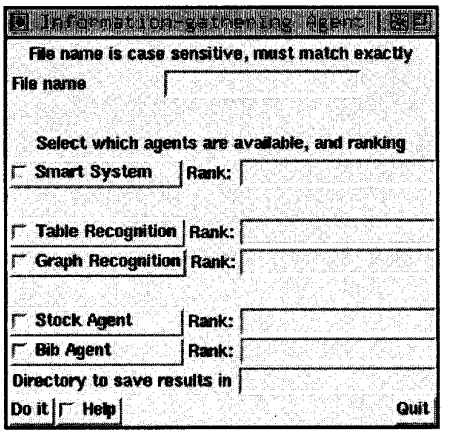

Fig. 17. The query window that serves as interface for our agents.

(e.g., ghostview for PostScript files and MPEG for video files, etc.). A sample output from our implementation is shown in Figure 18.

Bib Agent is a learning agent that is transportable (through FTP). It incrementally constructs a road map of the Internet indexed by query classes. The road map consists of cached paths to information. These cached paths allow it to get to these locations more easily in the future. Bib Agent also customizes the cached paths from user input as it searches. Unsuccessful paths are pruned quickly by this approach, and a user can customize Bib Agent with his or her own preferences by showing it some examples.

The complexity of this agent arises from the considerable knowledge about the Unix file organization embedded in each structure detector in the tree. We illustrate this using an example from our existing implementation. When the agent for this task reaches the directory /vol/alex/edu/mit/ai it has a choice of seven large subdirectories to search. Our detector uses the README file to aid in the selection of a subset of subdirectories for further investigation. Bib Agent knows all the error messages of the Alex system and provides customized routines to handle them. In some cases, Bib Agent asks the user to help resolve an error message from Alex. For instance, Bib Agent knows about connection timeouts and will delay reconnecting to the site by a certain amount. If repeated timeouts occur, Bib Agent informs the user and asks for the next course of action.

4.1 Experimental Data

We have tested the performance of our Bib Agent by simulating a distributed collection. The test data consists of the Cornell computer science technical report library for the years 1968 through 1994. The technical reports are stored in various formats: text (abstract only), PostScript, DVI, and HTML. We treat the reports from each year as belonging to a separate site. Within each site reports are stored in a complex directory-subdirectory structure. Our agent automatically navigates through this world.

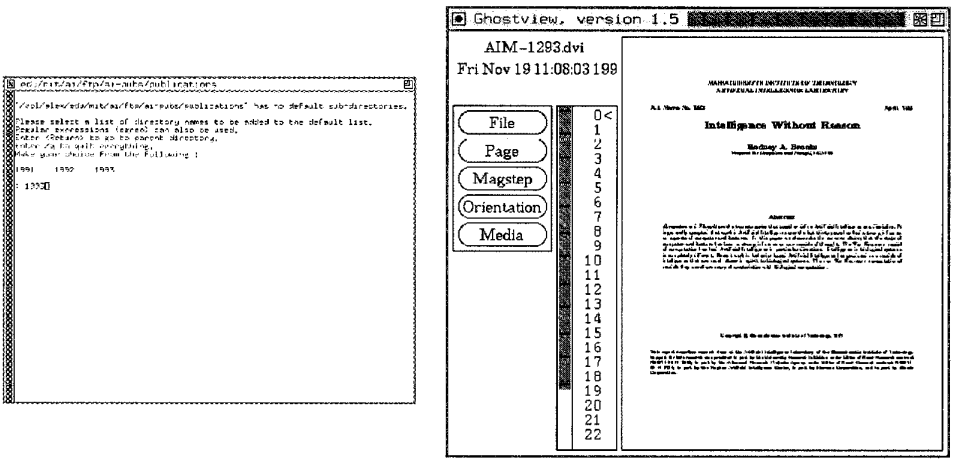

Fig. 18. The user interface of Bib Agent. The left image shows a query window that was displayed in response to a query for ai papers on *biological* models and systems. The agent automatically navigates to the *mit/ai/ftp/ai-pubs/publications* directory but does not know how to select between the *1991*, *1992*, and *1993* subdirectories. The user selects 1991, and the agent returns the only paper that mentions biological models in the abstract. The first page of this document is shown on the right.

For each site, the Bib agent performs a breadth-first search of the directory-subdirectory structure, examining all files that it encounters along the way using the search filter (e.g., keywords) provided by the user. The path list computed for each site consists of all the paths to directories that contain files on the given topic.

Data from executing the Bib Agent are shown in Figures 19 and 20. In Figure 19 we show the cluster of papers that talk about theories (the filled bullets), experiments (the bold bullets), and both theories and experiments (the shaded bullets). These clusters contain a small fraction of all the existing technical reports. Thus, a specific query, for example "theory and experiments in robotics at Cornell," quickly retrieves the paper that corresponds to one of the shaded bullets. Similarly, the ISIS cluster can be used for efficient retrieval as well as compiling statistics on the history of the ISIS group.

The present implementation of Bib Agent uses keyword query specifications. We can reconfigure the Bib Agent into one that processes specifications involving tables by replacing the text (keyword) filters in our implementation by table filters. The ease of construction of new variants of the Bib Agent is made possible by our modular construction kit of detectors and segmenters. We are currently working on an implementation of Bib Agent in Agent-Tcl with a World Wide Web interface. This will be provided to external users as a service of the Dienst Technical Report server [Davis and Lagoze 1995].

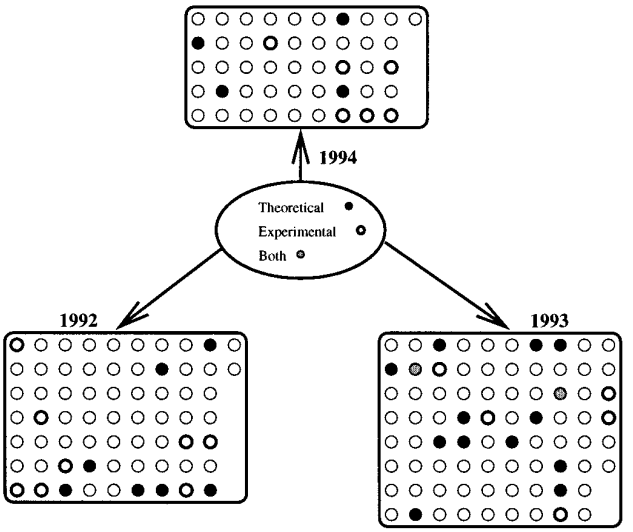

Fig. 19. The results from running the Bib Agent to cluster the Cornell technical reports on the query classes Theory and Experiments. The world consists of three distributed nodes, each containing technical reports from a different year.

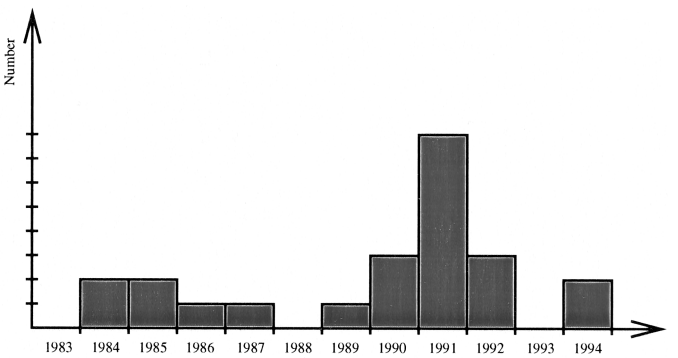

Fig. 20. The results of running the Bib Agent to collect papers on the ISIS project. The world consists of 12 distributed nodes, each containing technical reports from a different year.

5. DISCUSSION

Our goal is to develop and prototype a methodology for customizable information and access tools for large, distributed, heterogeneous information domains. This is challenging for many reasons: (1) there is a mismatch between the granularity of data representation and the task specification, (2) there is little agreement about how to specify general retrieval tasks, and (3) the amount of data is too large for exhaustively searching it. Our hypothesis is that the structure that exists in electronic data at various levels of detail embodies implicit representational conventions and that smart filters composed of structure detectors and segmenters that use this structure are a suitable computational paradigm for organizing the search for information.

The key technical challenge is association of content with information. By this we mean finding structural cues or substitutes for semantic information at different levels of detail. To do this, we need a formal framework for analyzing what information is necessary for performing a task. Such a framework, based on the notion of *information invariants*, has been discussed in the robotics context by Donald [1995] and Donald et al. [1993] and in the theoretical literature by Blum and Kozen [1978].

Our long-term goal is to computationally characterize methods such as statistics over character sequences [Pearce and Nicholas 1993; Salton and McGill 1983], statistics over word occurrence, layout and geometry, and other notions of structure with respect to information content. There are many important questions that arise in the context of structure-based information retrieval.

—For a given class of ICA tasks and data repositories, what are the appropriate structural cues?

—What information is encoded by a given structure?

—What class of partial models can be constructed with a given segmenter?

—Given an agent what class of ICA tasks is it good for?

—Can we define a computational hierarchy of ICA tasks? That is, is there an order relation on the information necessary to solve various classes of tasks?

We advocate modular construction of smart filters from libraries of structure detectors and segmenters. These detectors and segmenters come with performance guarantees. For instance, the table detector can detect tabular information to within user-specified tolerance in row and column misalignments. Our scheme is suited for fast prototyping of customized information agents.

Our design philosophy is orthogonal to the philosophy behind large, general-purpose systems such as the World Wide Web and Gopher. When is this design methodology more appropriate than others? To answer this question, we need a theoretical basis for characterizing ICA tasks and for measuring the effectiveness of alternate architectures on task classes. The notion of structure introduced in this article is a first step toward characterizing tasks in an implementation-independent manner. We constructed a smart filter for the class of retrieval tasks whose answers can be found in tabular form. We used this filter to build a search engine for compiling stock reports and for finding precision-recall measures. Our current implementations of this engine have been tested on Internet newsgroups, Internet FTP sites, and on the CS-TR project.[12] We would like to provide our tools for smart-filter construction to a much larger body of users and gather feedback.

[12]The CS-TR project is a nationwide effort to create an electronic library of computer science reports.

Several criticisms can be levied against this task-directed information agent approach to ICA tasks.

(1) *Task specification: at what level of detail do we need to specify tasks?* To specify a task we need to describe the desired information at a detailed enough level that an agent can locate and retrieve it from the environment. An analogy helps in clarifying issues in task specification. Suppose you want a visiting alien to match all the socks in your laundry. You will probably tell him or her to "match the socks and put them away in my sock drawer." Now what does he or she have to know to do this? First, he or she needs to know that socks come in pairs and that matching them means finding all pairs, and next, where the laundry is in your apartment and where socks are likely to be found (in the dryer, washer, under the bed, under the sofa cushions, etc.). These constitute task-specific knowledge that needs to be known to accomplish the task. Some of the knowledge is essential to perform the task correctly, and others aid in efficient accomplishment of the task. If your servant does not know what matching socks means, he or she will be unable to perform the task at all. If preferred locations of socks in your apartment are not known, it will probably take longer to do the task. It is useful to think of information agents as electronic organizers; these agents give us sanitized views of information worlds. They accomplish important, routine tasks that we need done, but do not have the patience to do.

(2) *Matching structures to tasks: how can we identify structures relevant to a task?* To design information agents we need to identify structures in the environment that can lead us to the required information for solving the task. These structures exist because humans use and design conventions for representing information. For instance, typesetting rules embodied in TeX have standardized the visual appearance of technical papers. Structure can thus be used as a cue for task-level (semantic) content. Can one match structures to tasks in a general way, or is it going to reduce to 1001 special cases? Since this matching relies fundamentally on conventions for representing information, this question translates to one of characterizing these conventions. Such a characterization is outside the scope of this article. Our hope is that as we construct more agents in our testbed environment of scanned technical reports, we will identify patterns in matching structures to tasks.

(3) *Scale up: this approach may be good for simple ICA tasks, but does it work for more complex tasks?* In our framework, task complexity is characterized by how well understood the mapping between task and environmental structures is. We can view structure detectors as bandpass filters which extract information within specific frequency ranges. A library of structure detectors pulls out different bands of interest. Our library contains parametric structure detectors and corresponding segmenters for commonly occurring patterns in the data environment.

We have demonstrated their use in the construction of agents for two useful task classes. Problems caused by heterogeneity and noise in data are solved by our approach. However ill-specified tasks still present a challenge.[13] Our focus in this article has been to expand the class of tasks that can be solved to beyond those specified by keywords. Our approach supports the scaling of task classes to those specifiable and solvable by geometric or visual cues.

(4) *Solution complexity: when word-based indices work impressively well, why do we need high-level structures to extract information?* We do not suggest the supplanting of existing word-based methods by our structure-based techniques; rather, we envision their synergistic combination to provide rapid filtering in large text environments. For example, the ability to recognize the logical and layout structure of documents provides automatic support for the SGML-like markup of documents that text retrievers like Smart require. for nontext environments, structures based on geometry and other domains are necessary for content-directed retrieval of information.

We recognize that issues relating to how users specify structures associated with tasks, and the development of a broad query language, are important to make the agent approach user friendly and its use widespread. We hope that this article will provide inspiration for such investigations.

In the end, how well our approach performs is an empirical question. Whether users of the information superhighway prefer to build their own "hot rods" with a structure detector and segmenter kit, or take "public transportation" that serves all uniformly,[14] will ultimately be judged by history.

ACKNOWLEDGMENTS

We thank John Hopcroft for proposing the problem of information capture and access and for his guidance and support. Special thanks to James Allan and K. Sivaramakrishnan for their careful reading of an earlier draft. Thanks go to Jim Davis, Bruce Donald, Dean Krafft, T. V. Raman, Jonathan Rees, Matthew Scott, Kristen Summers, and Rich Zippel for enlightening discussions. We are also very grateful to the anonymous reviewers for carefully reading earlier drafts and for making suggestions that helped us improve the article considerably.

[13]Consider, for instance, John Hopcroft's test for a conceptual retrieval system "how does the Brown Computer Science Department's research budget compare with that of Cornell's?" Where is this information to be found? What are suitable indicators for this metric (other than the research budget itself)? This query highlights the fact that there is a tremendous amount of information about the environment that the designer needs to know upfront in order to solve information retrieval tasks.

[14]At the lowest common denominator over usages.

REFERENCES

ALLAN, J. AND SALTON, G. 1993. The identification of text relations using automatic hypertext linking. In the *Workshop on Intelligent Hypertext, The ACM Conference on Information Knowledge Management.* ACM, New York.

BALCÁZAR, J. L., DÍAZ, J., AND GABARRÓ, J. 1988. *Structural Complexity.* EATCS Monograph on Theoretical Computer Science, vol. 1. Springer-Verlag, Berlin.

BELKIN, N. AND CROFT, W. 1992. Information filtering and information retrieval: Two sides of the same coin. *Commun. ACM 35,* 12 (Dec.), 29–38.

BLUM, M. AND KOZEN, D. 1978. On the power of the compass (or, why mazes are easier to search than graphs). In *Proceedings of the Symposium on the Foundations of Computer Science.* IEEE, New York, 132–142.

BROOKS, R. 1986. A robust layered control system for a mobile robot. *IEEE J. Robot. Automat. RA-2* (Apr.).

BROOKS, R. 1990. Elephants don't play chess. In *Design of Autonomous Agents,* P. Maes, Ed. MIT/Elsevier, Cambridge, Mass.

CANNY, J. AND GOLDBERG, K. 1993. A "RISC" paradigm for industrial robotics. In *Proceedings of the International Conference on Robotics and Automation.* IEEE, New York.

CATE, V. 1992. Alex: A global file system. In *Proceedings of the Usenix Conference on File Systems.* USENIX Assoc., Berkeley, Calif.

COHEN, J., Ed. 1993. *Commun. ACM 36,* 4 (Apr.).

CREAN, P., RUSSELL, C., AND DELLON, M. V. 1991. Overview and programming guide to the Mind image management systems. Tech. Rep. X9000627, Xerox, Inc., Palo Alto, Calif.

DAVIS, J. AND LAGOZE, C. 1995. Dienst—An architecture for distributed document libraries. *Commun. ACM 38*, 4 (Apr.), 47.

DONALD, B. 1995. Information invariants in robotics. *Artif. Intell. 72,* 217–304.

DONALD, B., JENNINGS, J., AND RUS, D. 1993. Information invariants for cooperating autonomous mobile robots. In *Proceedings of the International Symposium on Robotics Research.* Carnegie-Mellon Univ., Pittsburgh, Pa.

DONALD, B., JENNINGS, J., AND RUS, D. 1995. Minimalism + distribution = supermodularity. *J. Exper. Theoret. Artif. Intell.* To be published.

ETZIONI, O. AND WELD, D. 1994. A softbot-based interface to the Internet. *Commun. ACM 37,* 7 (July), 72–76.

FUJISAWA, H., NAKANO, Y., AND KURINO, K. 1992. Segmentation methods for character recognition: From segmentation to document structure analysis. *Proc. IEEE 80,* 7.

GENESERETH, M. AND KETCHPEL, S. 1994. Software agents. *Commun. ACM 37,* 7 (July), 48–53.

GRAY, R. 1995. Transportable agents. Tech. Rep. PCS-TR95-261, Dept. of Computer Science, Dartmouth College, Hanover, N.H.

GRAY, R. 1996. Agent Tcl: A flexible and secure mobile agent system. In *Proceedings of the 4th Annual Tcl/Tk Workshop.* ACM, New York.

HEARST, M. AND PLAUNT, C. 1993. Subtopic structuring for full-length document access. In *Proceedings of the 16th Annual International ACM SIGIR Conference on Research and Development in Information Retrieval.* ACM, New York, 59–68.

HOPCROFT, J. AND ULLMAN, J. 1979. *Introduction to Automata Theory, Languages, and Computation.* Addison-Wesley, Reading, Mass.

HUTTENLOCHER, D., KLANDERMAN, G., AND RUCKLIDGE, W. 1993. Comparing images using the Hausdorff distance. *IEEE Trans. Patt. Anal. Machine Intell. 15,* 9, 850–863.

HUTTENLOCHER, D., NOH, J., AND RUCKLIDGE, W. 1992. Tracking non-rigid objects in complex scenes. Tech. Rep. TR92-1320, Cornell Univ., Ithaca, N.Y.

JAIN, A. AND BHATTCHARJEE, S. 1992. Address block location on envelopes using Gabor filters. *Patt. Recog. 25,* 12.

KAHLE, B. 1991. Overview of wide area information servers. WAIS Online Doc. Online 15 (Sept. 5), 56–60.

KAHN, R. AND CERF, V. 1988. The world of knowbots. Report to the Corporation for National Research Initiative, Arlington, Va.

KAUTZ, H., SELMAN, B., AND COEN, M. 1994. Bottom-up design of software agents. *Commun. ACM 37,* 7 (July), 143–145.

KUCERA, H. AND FRANCIS, W. 1967. *Computational Analysis of Present Day American English.* Brown University Press, Providence, R.I.

LESK, M. 1991. The CORE electronic library. In *Proceedings of SIGIR.* ACM, New York.

MAES, P. 1994. Agents that reduce work and information overload. *Commun. ACM 37,* 7 (July), 31–40.

MITCHELL, T., CARUANA, R., FREITAG, D., MCDERMOTT, J., AND ZABOWSKI, D. 1994. Experience with a learning personal assistant. *Commun. ACM 37,* 7 (July), 81–91.

MIZUNO, M., TSUJI, Y., TANAKA, T., TANAKA, H., ISASHITA, M., AND TEMMA, T. 1991. Document recognition system with layout structure generator. *NEC Res. Devel. 32,* 3.

MUNKRES, J. 1975. *Topology: A First Course.* Prentice-Hall, Englewood Cliffs, N.J.

NAGY, G., SETH, S., AND VISHWANATHAN, M. 1992. A prototype document image analysis system for technical journals. *Computer 25,* 7.

PEARCE, C. AND NICHOLAS, C. 1993. Generating a dynamic hypertext environment with n-gram analysis. In *Proceedings of the ACM Conference on Information Knowledge Management.* ACM, New York, 148–153.

ROBERTSON, S. 1981. The methodology of information retrieval experiment. In *Information Retrieval Experiment,* K. Sparck Jones, Ed. Butterworths, Durban, S. Africa, 9–31.

ROBERTSON, G., CARD, S., AND MACKINLAY, J. 1993. Information visualization using 3D interactive animation. *Commun. ACM 36,* 4 (Apr.), 57–70.

RUS, D. AND SUBRAMANIAN, D. 1993. Multi-media RISSC informatics: Retrieving information with simple structural components. In *Proceedings of the ACM Conference on Information and Knowledge Management.* ACM, New York.

RUS, D. AND SUMMERS, K. 1995. Using whitespace for automated document structuring. In *Advances in Digital Libraries,* N. Adam, B. Bhargava, and Y. Yesha, Eds. Lecture Notes in Computer Science, vol. 916. Springer-Verlag, New York.

RUS, D., GRAY, R., AND KOTZ, D. 1997. Transportable information agents. In *Proceedings of the 1st International Conference on Autonomous Agents.* ACM, New York. To be published.

SALTON, G. 1989. *Automatic Text Processing: The Transformation, Analysis, and Retrieval of Information by Computer.* Addison-Wesley, Reading, Mass.

SALTON, G. AND BUCKLEY, C. 1990. Improving retrieval performance by relevance feedback. *J. Am. Soc. Inf. Sci. 41,* 4, 288–297.

SALTON, G. AND MCGILL, M. 1983. *Introduction to Modern Information Retrieval.* McGraw-Hill, New York.

SANKOFF, D. AND KRUSKAL, J. 1983. *Time Warps, String Edits, and Macromolecules: The Theory of Practice of Sequence Comparison.* Addison-Wesley, Reading, Mass.

SCHWARTZ, M. AND TSIRIGOTIS, P. 1991. Experience with a semantically cognizant Internet white pages directory tool. *J. Internetworking Res. Exper.* (Mar.).

SCHWARTZ, M., EMTAGE, A., KAHLE, B., AND NEUMAN, B. 1992. A comparison of Internet discovery approaches. *Comput. Syst. 5,* 4.

STATISTICAL SCIENCES. 1991. *Splus Reference Manual.* Statistical Sciences, Inc., Seattle, Wash.

TSUJIMOTO, S. AND ASADA, H. 1992. Major components of a complete text reading system. *Proc. IEEE 80,* 7.

WANG, D. AND SRIHARI, S. 1989. Classification of newspaper image blocks using texture analysis. *Comput. Vis. Graph. Image Process. 47.*

WONG, K., CASEY, R., AND WAHL, F. 1982. Document analysis system. *IBM J. Res. Devel. 26,* 6.

Received December 1993; revised December 1994 and August 1995; accepted February 1996