AλgoVista—A Tool for Classifying Algorithmic Problems and Combinatorial Structures

Christian S. Collberg,* Todd A. Proebsting†

Technical Report TR04-09

April 28, 2004

Abstract

AλgoVista is a web-based search engine that assists programmers to classify algorithmic problems and combinatorial structures. AλgoVista is not keyword based but rather requires users to provide — in a very simple query language — *input⇒output* samples that give a rough description of the behavior of their needed algorithm. AλgoVista also allows algorithm designers to advertise their results in a forum accessible to programmers and theoreticians alike.

AλgoVista's search mechanism is based on a novel application of *program checking*, a technique developed as an alternative to program verification and testing.

The current AλgoVista database consists of over 300 problem descriptions including problems on graphs, trees, matrices, vectors, sets, numbers, and geometric objects.

AλgoVista can be searched textually as well as visually. A user creates a visual query by simply drawing it on the canvas of a web browser. Visual queries are parsed into their textual counter-part by an algorithm that relies on user input to resolve ambiguities.

AλgoVista operates at `http://algovista.cs.arizona.edu`.

1 Introduction

AλgoVista is a web-based, interactive, searchable, and extensible database of problems, algorithms, and combinatorial structures designed to bring together applied and theoretical computer scientists. Programmers can query AλgoVista to look for relevant theoretical results, and theoretical computer scientists can extend AλgoVista with problem solutions.

Unlike most other search engines, AλgoVista is not keyword-based. Keyword-based searching fails exactly in those situations when we are in the most need for accurate search results, namely when we are searching in a new and unfamiliar domain. For example, a programmer looking for an algorithm that solves a particular problem on graphs will not get any help from a keyword-based search engine if she does not know what the problem is called. A Google keyword search for ⌜`graph algorithms`⌝, for example, returns more than 1.2 million hits that the user has to browse manually.

Instead, AλgoVista is searched using a *query-by-example* technique. The user provides one or more simple

input⇒output

examples that give a (usually fuzzy and incomplete) description of the problem she is looking for. AλgoVista will match this description against the problem specifications in its database and return links to relevant web resources.

This query-by-example technique turns out to be remarkably successful: when looking for information on particular graph algorithms, for example, AλgoVista often returns the requested results in a few seconds.[1]

*Department of Computer Science, University of Arizona, Tucson, AZ 85721. `collberg@cs.arizona.edu`

†Microsoft Research, One Microsoft Way, Redmond, WA. `toddpro@microsoft.com`

[1] Note that AλgoVista is a *database* of problem descriptions and that the *input⇒output* examples described here are used to query this database. Previous work has attempted to use *input⇒output* examples to *generate* problem solutions automatically, which is a much harder problem. We discuss this difference further in Section 10.

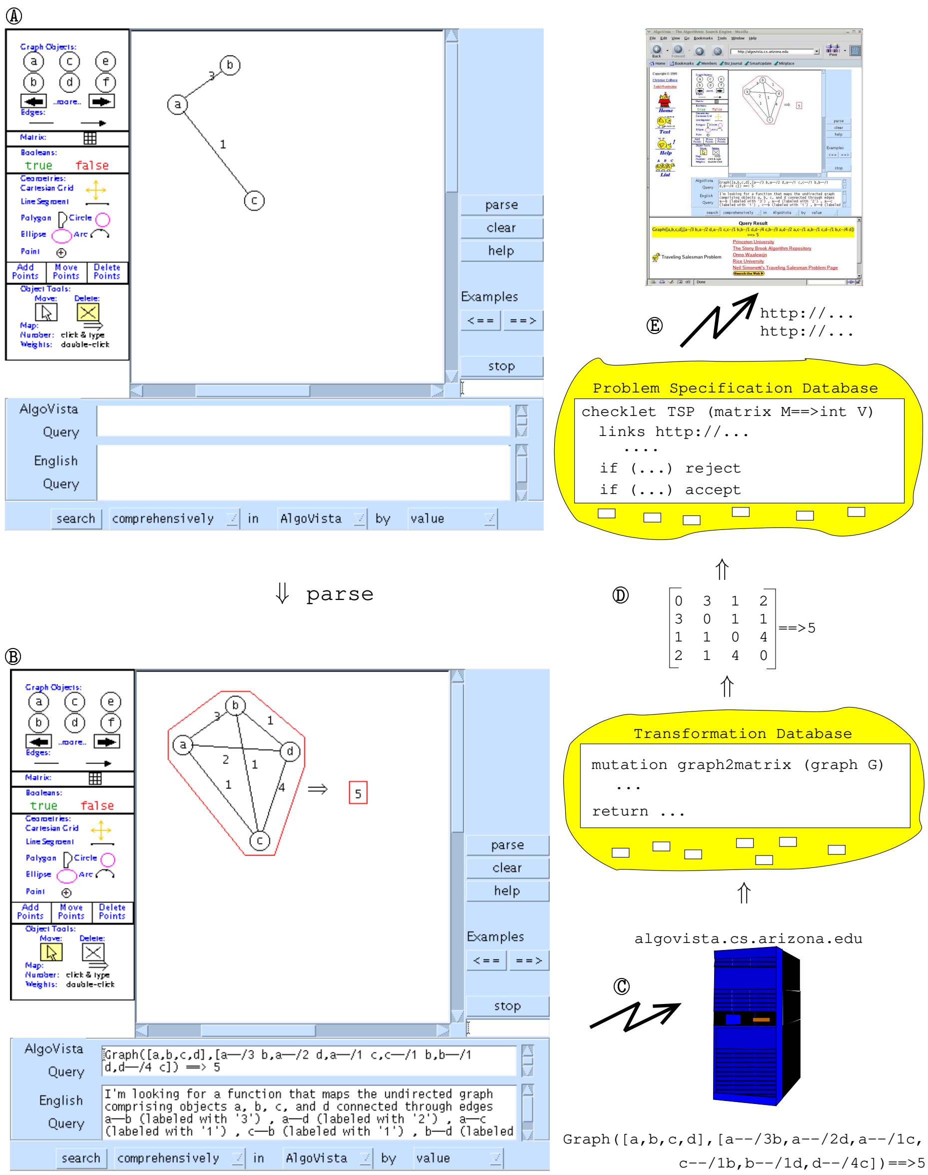


Figure 1: Overview of AλgoVista processing a query. The user enters the query by drawing it in an applet in a standard web browser. This graphical query is parsed into its textual counterpart and transfered to the AλgoVista server for processing. The query may go through a sequence of transformations before it is matched against the executable specifications in AλgoVista database. Finally, results in the form of a set of links to relevant web resources is returned to the user. Figure 2 shows an expanded view of the user's browser after results have been returned.

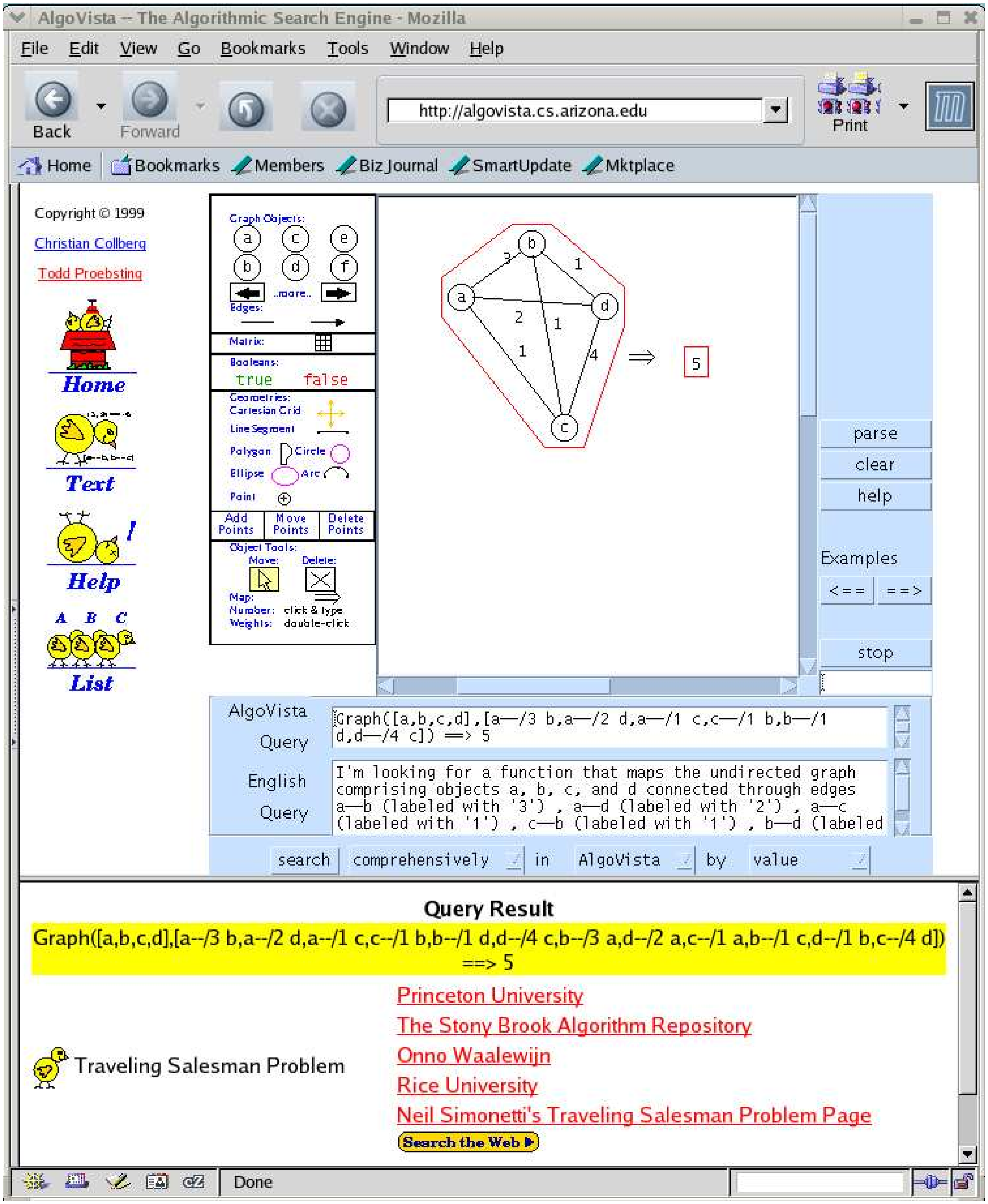


Figure 2: The AλgoVista graphical user interface running inside a standard web browser. The top right frame is an applet into which the user can enter a graphical query. The applet also displays the textual and English language queries corresponding to the graphical one. The bottom frame shows the result of the search.

1.1 Interacting with AλgoVista

To show how a user would typically interact with AλgoVista and how the search engine would field a query we will consider a simple example.

Consider a situation where a programmer is looking for an algorithm that "computes the shortest path through all the cities on a map." If the programmer is unfamiliar with graph theory terminology in general or the Traveling Salesman Problem in particular, she would find it difficult to locate any information about this problem on the Web. AλgoVista instead allows the user to search *without any knowledge of specialized terminology* — all that is required is that she can draw an instance of the problem she is looking to classify.

The AλgoVista search engine is accessed through a standard web browser, at `http://algovista.cs.arizona.edu`. AλgoVista can be searched using both a textual and a visual query language, but for the moment we will only consider visual queries. The visual user interface shown in Figure 2 has a main drawing window in which the user can enter her query. Query primitives include nodes and edges (to construct graphs, trees, and lists), matrices, vectors, booleans, numbers, and geometric objects. In our example, the user has formulated a query by drawing a weighted graph and indicating that the function she is looking for should map this graph to the number 5.

Figure 1 shows the sequence of events that takes place during the search:

1. At Ⓐ the user has started to enter her query. In this example an edge-labeled graph is being drawn by dragging nodes from the template canvas onto the main drawing canvas.

2. At Ⓑ the visual query is finished and the user has clicked the SEARCH button. The query asks which function maps the labeled graph on the left of the ==>-arrow to the number 5 on the right. The AλgoVista client analyzes the drawing and translates it into an equivalent textual query which is displayed in the query window.

3. Since there may be more than one way to interpret the drawing the client attempts to explain to the user how the query was parsed by a) drawing convex hulls around those elements of the drawing that the visual parser has decided belong together, and b) translating the textual query into English prose which is displayed in the English query window. If the user is unhappy with the resulting query she can press the PARSE button to cycle through all the possible visual parses.

   The convex hulls and the English prose query are not used as part of the search process, they merely aim to help the user understand the query they have just entered.

4. At Ⓒ the textual query is sent to the AλgoVista server for processing.

5. The AλgoVista server contains a large number of executable problem specifications (called *checklets*) against which queries should be matched. However, the user and the checklet may use different representations of the same query. For this reason, AλgoVista contains a set of *query transformations* which can make up for representational differences by transforming a query before it is being matched against the checklets in the database. At Ⓓ, the linked representation of the graph is translated into an adjacency-matrix representation.

6. At Ⓔ the mutated query is matched against the problem specifications in the database. In this case, the query matches the Traveling Salesman Problem.

7. Finally, links to information about this problem are returned and displayed in the user's browser. The result returned by the server consists of a classification of the problem, links to well-known web resources which describe the problem, links to implementations, and a *search-the-web*-button which invokes a standard keyword-based search engine (such as Google) with a list of keywords the server has determined to be relevant. Figure 2 shows an expanded view of the browser at the end of the query process.

It is important to note that the programmer could have entered *any* legal instance of the TSP problem. Furthermore, she could have expressed the input graph as an adjacency matrix, could have mapped the graph to a list of the nodes traversed, could have mapped the graph to the subgraph that makes up the tour, etc.

The main advantage of AλgoVista's use of *query-by-example* over standard keyword-based search is that it allows the user to express her query in whatever terms are most natural to her. A programmer who has never heard the term Traveling Salesman or who has no background in the fundamentals of graph theory can still search AλgoVista by drawing an example describing the function she is looking for.

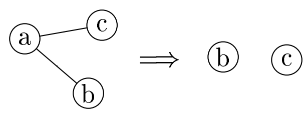
1.2 Applications of AλgoVista

Before we continue our description of AλgoVista's query languages and the algorithms used to search its problem specification database, we will briefly consider a few more examples in order to motivate the need for a specialized search engine for computer scientists. We will see how AλgoVista is particularly helpful when one is attempting to classify a problem outside ones own area of expertise and therefore has no knowledge of the relevant terminology.

Example 1: Suppose Bob is trying to write a program that identifies the locations for a new franchise service. Given a set of potential locations, he wants the program to compute the largest subset of those locations such that no two locations are close enough to compete with each other. It is trivial for him to compute which pairs of locations would compete, but he does not know how to compute the feasible subset. He starts by trying to come up with an example of how his program should work:

- If there are three locations a, b, c and a competes with b and c, then the best franchise locations are b and c.

If Bob is unable to come up with his own algorithm for this problem he might turn to one of the search-engines on the web. But, which keywords should he use? Or, Bob could consult one of the algorithm repositories on the web, such as `http://www.cs.sunysb.edu/~algorith`, which is organized hierarchically by category. But, in which category does this problem fall? Or, he could enter the example he has come up with into AλgoVista by drawing the query



The query expresses:

> "If the input to my program is two relationships, one between `a` and `b` and one between `a` and `c`, then the output is the collection `[b,c]`."

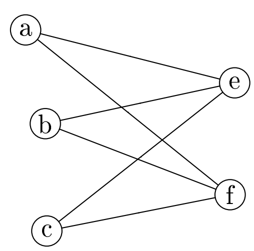
Another way of thinking about this query is that the input is a graph of three nodes `a`, `b`, and `c`, and edges `a-b` and `a-c`, but it is not necessary for Bob to know about graphs. AλgoVista returns to Bob a link directly to `http://www.cs.sunysb.edu/~algorith/files/independent-set.shtml` which contains a description of the Maximal Independent Set problem. From this site there are links to implementations of this problem.

Example 2: Suppose that Bob is writing a simple DNA sequence pattern matcher. He knows that given two sequences $\langle a,a,t,g,g,g,c,t\rangle$ and $\langle c,a,t,g,g\rangle$, the matcher should return the match $\langle a,t,g,g\rangle$, so he enters the textual query

```
([a,a,t,g,g,g,c,t],[c,a,t,g,g]) ==> [a,t,g,g]
```

into AλgoVista which (within seconds) returns the link `http://www.nist.gov/dads/HTML/longestcommn.html` to a description of the Longest Common Subsequence problem.

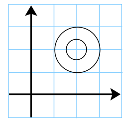
Example 3: AλgoVista is also able to classify combinatorial structures. Given the following query



AλgoVista might respond with:

> "This looks like a *bipartite graph*. `http://www.nist.gov/dads/HTML/bipartgraph.html` has more information about this structure."

Example 4: Suppose Bob is studying for a High School geometry test. He cannot remember what the term is for "two circles with a common center." So, he selects AλgoVista's *cartesian grid* and draws this simple figure:



In return he receives a link to `http://mathworld.wolfram.com/ConcentricCircles.html` where he can read more about Concentric Circles.

1.3 Organization

The remainder of this paper is organized as follows. In Section 2 we introduce *program checking*, the theoretical framework on which AλgoVista is built. In Section 3 we describe AλgoVista's textual query language. In Section 4 we show that in order for a user query to match the entries in the AλgoVista database they may need to be transformed by a series of *query transformations* into several different representations. In Section 5 we describe an efficient search algorithm. In Section 7 we describe the visual query language and how it is parsed. In Section 6 we describe how *checklets*, the problem specifications that make up the AλgoVista database, are constructed. In Section 8 we describe certain security issues. In Section 9 we evaluate the effectiveness of our keyword-less search strategy and the efficiency of our algorithms. In Section 10 we present some related work, in Section 11 we present future research directions, and in Section 12 we summarize our results.

2 The AλgoVista Checklet Database

The problem specifications in the AλgoVista database are executable and are called *checklets*. A checklet typically takes a user query *input*⇒*output* as input and either *accepts* or *rejects*. If the checklet accepts a query it also returns a description of the problem it checks for and links to relevant web resources. Figure 3 shows three simple checklets.

Figure 3 (a) shows a checklet for the sorting problem. It takes two integer arrays as argument, the unsorted *input* array and the sorted *output* array. The checklet accepts if the `output` array is sorted and contains the same elements as the `input` array.

Figure 3 (b), is a particularly interesting checklet for topological sorting. Any acyclic graph will typically have more than one topological order. It is therefore not possible for the checklet to simply run a topological sorting procedure on the input graph and compare the resulting list of nodes with the output list given in the query. Rather, the checklet must, as shown in Figure 3 (b), first check that every node in the input graph occurs in the output node list, and then check that if node f comes before node t in the output list then there is no path $t \leadsto f$ in the input graph.

AλgoVista checklets can also classify combinatorial structures. Figure 3 (c) shows a checklet that checks if the input graph is biconnected, by first executing a depth first search during which the articulations points of the graph are found. The checklet accepts if the graph has no articulation points.

Appendix A lists some of the more than three-hundred checklets currently in the database.

Checklets are implemented in Java. In Section 6 we will discuss in more detail how AλgoVista checklets are constructed.

2.1 Program Result Checking

AλgoVista can be seen as a novel application of *program (result) checking*, an idea first introduced by Manuel Blum [7]. The idea behind program checking is simply this. Suppose we are concerned about the correctness of a procedure P in a program we are writing. We intend for P to compute a function f, but we are not convinced it does so. We have three choices:

1. We can attempt to *prove* that $P \equiv f$ over the entire domain of P.

2. We can test that $P(x) = f(x)$, where x is drawn from a reasonable domain of test data.

3. We can include a *result checker* C_f^P with the program. For every actual input x given to P, the result checker checks that $P(x) = f(x)$.

For example, a result checker for a sorting routine would be a piece of code at the end of the routine that checks that the resulting array is indeed in sorted order and contains the same elements as the input array.

It is typical to require C_f^P and P to be independent of each other; i.e. they should be programmed using very different algorithms. We also want the checker to be *efficient*. To ensure that these conditions are met, it is generally expected that a result checker C_f^P should be asymptotically faster than the program P that it checks. That is, we expect that if P runs in time T then C_f^P should run in time $o(T)$.

Much work [28, 29, 8, 32, 9, 23] has gone into the study of efficient checkers for a variety of problem classes, and AλgoVista checklet writers can leverage this work. It is interesting to note that for many problems it is much more efficient to check the correctness of the solution than to compute the solution. For example, *sorting* takes $O(n \log n)$ time but checking the correctness of the sorted result can be done in almost linear time. The fact that checklets can be very efficient is part of the reason for AλgoVista's fast query times.

2.2 System Overview

Figure 4 gives an overview of the AλgoVista system architecture. The client is an applet running inside a standard Web browser. The user can enter either a visual query or a textual query that gets translated into a textual one. The resulting query is transfered to the server and processed by one of several search algorithms. In the most basic search algorithm the query is submitted to every checklet in the database and the response of every accepting checklet is returned:

```
function search (query)
   q ← parse(query)
   responses ← {}
   for every checklet c in the database do
      if c accepts the query with response r then
         responses ← responses ∪ {r}
   return responses
```

In Section 4 we will show a variant of this algorithm where a query may also undergo a set of *representation transformations* prior to being submitted to the checklets. These transformations try to compensate for the fact that user queries and checklets may use different data representations for the same problem. In Section 5 we explore more sophisticated algorithms that speed up search times significantly.

Figure 4 also shows that AλgoVista is a *communal* and *extensible* database. Arbitrary users can upload new checklets into the database, thereby extending it with new problem classifications and references to web-based resources. We will discuss this further in Section 8.

3 The Query Language

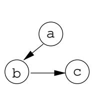
The AλgoVista query language was designed to be as simple as possible, while still allowing users to describe complex algorithmic problems. The language primitives include integers, floats, booleans, lists, tuples, matrices, atoms, links, and geometric objects (points, lines, circles, ellipses, arcs, and polygons). Links are (directed and undirected) edges between atoms that are used to build up linked structures such as graphs and trees. Table 1 shows the concrete syntax of the query language.

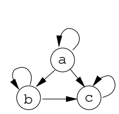
For example, ⌜`(1,true)`⌝ is a pair consisting of an integer and a boolean, ⌜`[2,3,5,7]`⌝ is a vector of integers, and ⌜`[2,3;5,7]`⌝ is a 2×2 matrix. Atoms, such as ⌜`x`⌝, are one-letter identifiers that are used to represent nodes of linked structures such as graphs and trees. They can carry optional node data of arbitrary type: ⌜`a/56`⌝, ⌜`a/[3,4]`⌝. Links between nodes can be directed (⌜`a->b`⌝) or undirected (⌜`a--b`⌝), and can carry optional edge data, as in ⌜`a->/56 b`⌝.

These simple primitives can be combined to produce complex queries. For example, the query

`[a->b,b->c]==>[a->a,a->b,a->c,b->b,b->c,c->c]`

asks which function maps

the directed graph 

to the directed graph 

<table>
<tr><th>primitive</th><th>syntax</th><th>examples</th></tr>
<tr><td>map</td><td>S_{in}==>S_{out}</td><td>(1,2)==>3</td></tr>
<tr><td>int</td><td>$(0..9)+$</td><td>123</td></tr>
<tr><td>float</td><td>$(0..9)*.(0..9)+$</td><td>123.456</td></tr>
<tr><td>boolean</td><td>true | false</td><td>true</td></tr>
<tr><td>pair</td><td>(S_1,S_2)</td><td>(1,2)</td></tr>
<tr><td>vector</td><td>$[S_1,S_2,\cdots]$</td><td>[1,2,3,2]</td></tr>
<tr><td>set</td><td>$\{S_1,S_2,\cdots\}$</td><td>{1,2,3}</td></tr>
<tr><td>matrix</td><td>$[S_{1,1},S_{1,2}\cdots;S_{2,1},S_{2,2}\cdots;\cdots]$</td><td>[1,2;3,4]</td></tr>
<tr><td>atom</td><td>$|$b$|$...$|$z$[/S]$</td><td>a, b/5</td></tr>
<tr><td>dedge</td><td>atom->$[/S]$ atom</td><td>a->b, c->/5 d</td></tr>
<tr><td>uedge</td><td>atom--$[/S]$ atom</td><td>a--b, c--/4 d</td></tr>
<tr><td>graph</td><td>graph([atom,...],[uedge,...])</td><td>graph([a,b],[a--b])</td></tr>
<tr><td>digraph</td><td>digraph([atom,...],[dedge,...])</td><td>digraph([a,b],[a->b,b->a])</td></tr>
<tr><td>dag</td><td>dag([atom,...],[dedge,...])</td><td>dag([a,b,c],[a->b,a->c,b->c])</td></tr>
<tr><td>tree</td><td>tree([atom,...],[dedge,...])</td><td>tree([a,b,c],[a->b,a->c])</td></tr>
<tr><td>list</td><td>list([atom,...],[dedge,...])</td><td>list([a,b,c],[a->b,b->c])</td></tr>
<tr><td><i>point</i></td><td><<i>float</i>,<i>float</i>></td><td><2.1,3.0></td></tr>
<tr><td>line</td><td>line(point,point)</td><td>line(<2.1,3.0>,<1.1,4.0>)</td></tr>
<tr><td>circle</td><td>circle(point,float)</td><td>circle(<2.1,3.0>,2.3)</td></tr>
<tr><td>ellipse</td><td>ellipse(point,float,float)</td><td>ellipse(<2.1,3.0>,2.3,3.4)</td></tr>
<tr><td>arc</td><td>arc(point,point,point)</td><td>arc(<2.1,3.0>,<1.1,3.3>,<2.2,1.0>)</td></tr>
<tr><td>polygon</td><td>polygon[line,line,...]</td><td>polygon[line(<2.1,3.0>,<1.1,3.3>),...]</td></tr>
<tr><td>grid</td><td>grid$[G_1,G_2,\cdots]$</td><td>grid[<2.1,3.0>,line(<2.1,3.0>,<1.1,4.0>)]</td></tr>
</table>

Table 1: The AλgoVista concrete query syntax. The S_α:s represent arbitrary query terms. The G_i:s represent arbitrary geometric objects. All G_i:s must be attached to a cartesian grid. Circles and ellipses take a center point argument as well as one or two radii, respectively. Shaded rows represent syntactic components which are internal to AλgoVista, and not lexically realized.

$\mathcal{T}[\![\textbf{INTEGER}]\!]$	$=$	$\mathbb{I}$
$\mathcal{T}[\![\textbf{FLOAT}]\!]$	$=$	$\mathbb{F}$
$\mathcal{T}[\![\textit{true}]\!]$	$=$	$\mathbb{B}$
$\mathcal{T}[\![\textit{false}]\!]$	$=$	$\mathbb{B}$
$\mathcal{T}[\![S_1 \texttt{ ==> } S_2]\!]$	$=$	$\texttt{Map}(\mathcal{T}[\![S_1]\!], \mathcal{T}[\![S_2]\!])$
$\mathcal{T}[\![\ \texttt{(}\ S_1\ \texttt{,}\ S_2\ \texttt{)}\ ]\!]$	$=$	$\texttt{Pair}(\mathcal{T}[\![S_1]\!], \mathcal{T}[\![S_2]\!])$
$\mathcal{T}[\![\ \texttt{[}\ S_1, S_2, \cdots\ \texttt{]}\ ]\!]$	$=$	$\texttt{if } \mathcal{T}[\![S_1]\!] = \mathcal{T}[\![S_2]\!] = \cdots \texttt{ then}$ $\texttt{Vector}(\mathcal{T}[\![S_1]\!]) \texttt{ else } \bot$
$\mathcal{T}[\![\ \texttt{[}\ S_{1,1}, S_{1,2} \cdots, S_{1,n}; \cdots; S_{m,1}, S_{m,2} \cdots S_{m,n}\ \texttt{]}\ ]\!]$	$=$	$\texttt{if } \mathcal{T}[\![S_{1,1}]\!] = \mathcal{T}[\![S_{1,2}]\!] = \cdots \texttt{ then}$ $\texttt{Matrix}(m, n, \mathcal{T}[\![S_{1,1}]\!]) \texttt{ else } \bot$
$\mathcal{T}[\![\textit{atom}/S]\!]$	$=$	$\texttt{Node}(\mathcal{T}[\![S]\!])$
$\mathcal{T}[\![\textit{atom} \texttt{ -> } /S\ \textit{atom}]\!]$	$=$	$\texttt{DEdge}(\mathcal{T}[\![S]\!])$
$\mathcal{T}[\![\textit{atom} \texttt{ -- } /S\ \textit{atom}]\!]$	$=$	$\texttt{UEdge}(\mathcal{T}[\![S]\!])$
$\mathcal{T}[\![\textit{graph}([A_1, \cdots, A_n], [E_1, \cdots, E_m])]\!]$	$=$	$\texttt{if } \mathcal{T}[\![A_1]\!] = \mathcal{T}[\![A_2]\!] = \cdots = \texttt{Node}(\alpha)$ $\texttt{and } \mathcal{T}[\![E_1]\!] = \mathcal{T}[\![E_2]\!] = \cdots = \texttt{UEdge}(\beta) \texttt{ then}$ $\texttt{Graph}(\alpha, \beta) \texttt{ else } \bot$
$\mathcal{T}[\![\textit{digraph}([A_1, \cdots, A_n], [E_1, \cdots, E_m])]\!]$	$=$	$\texttt{if } \mathcal{T}[\![A_1]\!] = \mathcal{T}[\![A_2]\!] = \cdots = \texttt{Node}(\alpha)$ $\texttt{and } \mathcal{T}[\![E_1]\!] = \mathcal{T}[\![E_2]\!] = \cdots = \texttt{DEdge}(\beta) \texttt{ then}$ $\texttt{Digraph}(\alpha, \beta) \texttt{ else } \bot$
$\mathcal{T}[\![\textit{grid}[G_1, G_2, \cdots]]\!]$	$=$	$\texttt{Grid}()$

Table 2: Type assignments.

(Transitive closure). The query

```
([3,7],[5,1,6]) ==> [5,1,6,3,7]
```

asks what function maps the lists `[3,7]` and `[5,1,6]` to the list `[5,1,6,3,7]` (List append).

The recursive structure of the grammar allows queries to be deeply nested, although this is fairly uncommon. For example, the query

```
[a->/[1]b,a->/[3,4]c] ==> [1,3,4]
```

looks for an algorithm that maps a tree to a list of integers, where each tree edge is labeled with a set of integers.

Table 3 gives some example queries.

3.1 Query Types

In Section 4 we will introduce *query transformations*, operations that map queries to queries with different representations, while maintaining semantic equivalence. Furthermore, in Section 5 we will present a search algorithm that makes extensive use of pre-computation to speed up searching. This algorithm relies on queries, checklets, and query transformations all being *typed*. Table 2 shows the type assigments that map the primitives of the concrete syntax of Table 1 into types.

For example, the query $\ulcorner$`([1,2],[3,4])==>[4,6]`$\urcorner$ has the type

$\ulcorner\texttt{Map(Pair(Vector(}\mathbb{I}\texttt{),Vector(}\mathbb{I}\texttt{)),Vector(}\mathbb{I}\texttt{))}\urcorner$

and the query $\ulcorner$`([a/3,b/2,c/1],[a->b,a->c])`$\urcorner$ has the type

$\ulcorner\texttt{Pair(Vector(Node(}\mathbb{I}\texttt{)),Vector(DEdge(}\mathbb{0}\texttt{)))}\urcorner$.

Type expressions also allow variables, written α, β, γ. These are used to describe the signatures of polymorphic checklets and transformations, as in the generic sorting checklet in Figure 3(a).

AλgoVista's type system is actually somewhat richer than the type assignments above would indicate. Special internal types `Set`, `Linked`, `Graph`, `Digraph`, `DAG`, `Tree`, and `List` have been introduced to make searching more efficient. For example, a checklet writer can declare a `setUnion` checklet either using the `Vector` type

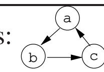
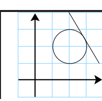
#	QL query	Query explanation
①	`[3, 5, 7, 8] ==> 23` **Query result:** ⟨`Vector reduction.`⟩	
②	`[a->b,b->c,c->a]` **Query result:** ⟨`Strongly connected graph`⟩	What kind of graph is this:  ?
③	`[m,a,d,a,m]` **Query result:** ⟨`Palindrome sequence`⟩	
④	`[a->c,a->d,b->c,d->c,d->b] ==> [a,d,b,c]` **Query result:** ⟨`Topological sort`⟩	List of edges representation.
⑤	`[0,0,1,1;0,0,1,0;0,0,0,0;0,1,1,0] ==> [1,4,2,3]` **Query result:** ⟨`Topological sort`⟩	Adjacency matrix representation.
⑥	`[a->[c,d],b->[c],c->[],d->[c,b]] ==> [a,d,b,c]` **Query result:** ⟨`Topological sort`⟩	List of neighbors representation.
⑦	`[n,k,l,i]==>[i,k,l,n]` **Query result:** ⟨`Sorting`⟩	
⑧	`[a--/3 b,a--/2 d,a--/1 c,c--/1 b,b--/1 d,d--/4 c] ==> 5` **Query result:** ⟨`Traveling Salesman Problem`⟩	
⑨	`[(3,8),(2,4)] ==> (-26,28)` **Query result:** ⟨`Complex multiplication`⟩	
⑨	`grid[circle(<2,2>,1),line(<2,4>,<3.8,1>)]` **Query result:** ⟨`Tangent of a Circle`⟩	

Table 3: Example AλgoVista queries.

```
checklet setUnionA (input ⇒ output)
   signature Map(Pair(Vector(α),Vector(α)),Vector(α))
      ...
```

or using the Set type

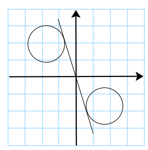
```
checklet setUnionB (input ⇒ output)
   signature Map(Pair(Set(α),Set(α)),Set(α))
      ...
```

Both will work equally well, but `setUnionB`'s signature hints to the search engine that it will fail for any inputs that are not sets (i.e. vectors that have duplicate elements). For example, the search engine would hand off the query

⌜`([1,2],[2,2,3])==>[1,2,3]`⌝

to `setUnionA` but not to `setUnionB` since `[2,2,3]` is not a set. Similarly, a checklet that declares that it takes a `DAG` as input will never be handed a graph with a cycle. Note that an AλgoVista user has to learn no special syntax to use these types. Rather, the *query transformations* that will be introduced in Section 4 and Appendix B take care of transforming queries from general types (such as `Vector`) to more specific types (such as `Set`) when this is legal.

Devising type assignments for geometric queries has proven not to be as straight-forward as for the other syntactic elements. One possibility is a very precise type system where every different collection of geometric elements yields a different type signature. For example, the grid



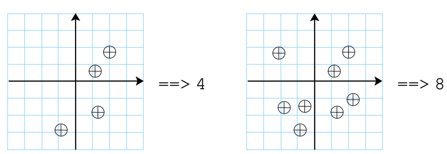
could get the type

⌜`Grid([Circle(),Circle(),Line()]`⌝

or, possibly,

⌜`Grid([Circle(),Line()]`⌝

However, we would not want to be *too* precise. This would, for example, lead to the queries



(that look for algorithms that count the number of points in the plane) to have different signatures. For these reasons, for geometric queries we have chosen a single, weak type: every cartesian grid gets the same type `Grid()`, regardless of the elements on the grid. Since we currently only have a small number of geometric checklets in our database, this serves our current purposes well. As the database grows, geometric queries may start to take a long time to service, and a more precise type system may be warranted.

4 Query Transformations

Early on in the design of AλgoVista we realized that there is often a representational gap between a user's query and the checklet that is designed to match this query. For example, there are any number of reasonable ways for a user to express a topological sorting query, including representing the input graph as a *list of edges*, an *adjacency matrix*, or a *list of neighbors*. These queries are shown in Figure 3 ④–⑥. The corresponding topological sorting checklet, on the other hand, might expect the input graph only in a matrix form.

This problem appears even for very simple queries. For example, in the query

⌜([3,7],[5,1,6])==>[5,1,6,3,7]⌝

the user was looking for information about the ⟨List append⟩ problem but gave the input pair of lists in the "wrong" order.

AλgoVista provides a set of *query transformations* that will automatically convert queries between common representations. For example, given the topological sorting query in Figure 3 ④, AλgoVista would automatically produce the queries in Figure 3 ④–⑥, all of which would be matched against the checklets in the checklet database. Similarly, the query

⌜([3,7],[5,1,6])==>[5,1,6,3,7]⌝

would be transformed into

⌜([5,1,6],[3,7])==>[5,1,6,3,7]⌝

allowing it to be accepted by the ListAppend checklet. We will refer to the transformed query as a *query mutation*.

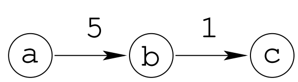
The search algorithm from Section 2 has to be extended to accommodate the transformation of queries. The exhaustive search algorithm in Figure 5 generates all possible mutations of the incoming query (by applying any combination of query transformations to it) and submits these to every available checklet. In Section 5 we will present an alternative algorithm that uses extensive pre-computation to speed up the search.

Appendix B lists the transformations currently in use by the search engine. For example, the `Int2Flt` transformation promotes a integer to a float and the `FlipPair` transformations swaps the elements of a pair. These transformations will form the basis for future examples, and are illustrated in Figure 6.

`List2VectorB` and many other transformations in Figure B are concerned with transforming between different representations of various linked structures.

4.1 A Query Transformation Example

Consider a user who wants to submit a query containing the list of integers ⌜[5,1]⌝. There are several ways to represent this list, and our user decides on a linked-list representation with edge-weights:



.

In our textual query language this would be expressed as

⌜[a->/5b,b->/1c]⌝.

However, a particular checklet might expect the list of integers to be given in a vector representation:

⌜[5,1]⌝.

Figure 7 shows how AλgoVista's transformation engine would transform the original linked representation to the vector representation (using the transformation `List2VectorB`). Further transformations will turn the original query into pairs of integers, booleans, and reals. The basic AλgoVista search algorithm in Figure 5 would hand off all twelve query mutations to all the checklets in the checklet database.

5 Query Optimization

In Figure 5 we described a straight-forward algorithm that employs exhaustive search to submit every possible mutation of a query to every checklet in the checklet database. Obviously, with dozens of transformations and hundreds of checklets this procedure will be prohibitively expensive.

To see how type-analysis can help us speed up the search, consider a situation where we have two checklets `FloatExp` and `FloatAdd` `FloatExp` checks for real exponentiation and `FloatAdd` checks for real addition, We also have two transformations `Int2Flt` and `FlipPair` where `Int2Flt` promotes an integer to a real and `FlipPair` commutes a pair. These are illustrated in Figure 6.

Suppose the input query is ⌜(2.0,2)==>4.0⌝. This input has a signature of

$$\texttt{Map(Pair(}\mathbb{F}\texttt{,}\mathbb{I}\texttt{),}\mathbb{F}\texttt{)},$$

and therefore can be tested immediately against the `FloatExp` checklet. Similarly, by applying the `Int2Flt` transformation, the query can be transformed into ⌜`(2.0,2.0)==>4.0`⌝, which matches the signature of `FloatAdd`, and therefore can be submitted to that checklet.

It is trivial to determine that the query ⌜`true==>1`⌝ (which has the type $\texttt{Map(}\mathbb{B}\texttt{,}\mathbb{I}\texttt{)}$) cannot match any of our (current) checklets, regardless of which transformations are applied. Still, the algorithm in Section 2 would apply all possible combinations of transformations to ⌜`true==>1`⌝ and submit any generated query mutation to every checklet in the database.

We will next show how precomputing *viable* transformations can speed up searching by eliminating any such useless transformations.

5.1 Fast Checking by Precomputation

Whenever a new checklet is added to the database, AλgoVista generates a new search procedure $\mathcal{S}_{\mathcal{T},\mathcal{C}}$ automatically. This procedure is hardcoded to handle exactly the set of transformations $\mathcal{T}$ which are available in the database of transformations, and the set of checklets $\mathcal{C}$ which are currently available in the checklet database. $\mathcal{S}_{\mathcal{T},\mathcal{C}}$ is constructed such that given an input query q whose type is $\mathcal{T}[\![q]\!]$, $\mathcal{S}_{\mathcal{T},\mathcal{C}}$ will apply exactly those combinations of transformations to q that will result in *viable* mutated queries. A query is *viable* if it is correctly typed for checking by at least one checklet.

In other words, AλgoVista's optimized search procedure $\mathcal{S}_{\mathcal{T},\mathcal{C}}$ will never perform a useless transformation, one that could not possibly lead to a mutated query correctly typed for some checklet.

In order to apply transformations and to test checklets efficiently, AλgoVista determines the signature of an input query upon its arrival. Given the query's signature, AλgoVista knows exactly which, if any, checklets to test, and which, if any, transformations to apply. Furthermore, AλgoVista knows the exact signature of each newly-generated query because it knows the input query signature and how the transformation will transform the signature. For example, AλgoVista knows that applying the `FlipPair` transform to

$$\texttt{Map(Pair(}\mathbb{F}\texttt{,}\mathbb{I}\texttt{),}\mathbb{F}\texttt{)}$$

will yield

$$\texttt{Map(Pair(}\mathbb{I}\texttt{,}\mathbb{F}\texttt{),}\mathbb{F}\texttt{)}.$$

This observation yields a very simple, but highly optimized architecture that applies transformations and tests checklets based on signatures, in which there is one function per signature responsible for all the operations that affect queries of that signature. Each function has three parts: verifying the originality of the query, testing all matching checklets, and generating isomorphic queries by applying transformations. All generated queries are simply handed off to the function that handles their signature.

For the given checklets and transformations above, the function that handles the signature $\texttt{Map(Pair(}\mathbb{F}\texttt{,}\mathbb{I}\texttt{),}\mathbb{F}\texttt{)}$ would look like this:

```
set FI_F_AlreadySeen;
function FI_F(query Q) {
   if Q in FI_F_AlreadySeen then return;
   insert Q into FI_F_AlreadySeen;

   Check if the FloatExp-query accepts Q;

   Apply Int2Flt (whose signature is
   Map(𝕀,𝔽)) to Q, yielding Q' (whose
   signature is Map(Pair(𝔽,𝔽),𝔽));

   Call FF_F(Q');
}
```

The set `FI_F_AlreadySeen` prevents the same query mutation from being produced more than once, as in this example:

$$
\begin{array}{lll}
\texttt{(1,2)==>3} & \stackrel{\text{FlipPair}}{\Rightarrow} & \texttt{(2,1)==>3} \\
 & \stackrel{\text{FlipPair}}{\Rightarrow} & \texttt{(1,2)==>3} \\
 & \stackrel{\text{FlipPair}}{\Rightarrow} & \texttt{(2,1)==>3} \\
 & \Rightarrow & \cdots
\end{array}
$$

The only non-trivial aspect of the generated function is knowing which transformations can be applied to a given signature, and *where*. For instance, given the query signature,

$$\texttt{Map(Pair(Pair(}\mathbb{I}\texttt{,}\mathbb{F}\texttt{),Pair(}\mathbb{F}\texttt{,}\mathbb{I}\texttt{)))}$$

it is possible to apply the `FlipPair` transformation at any of the three `Pair`s in the query—even the nested ones.

In addition to the signature-specific functions, it is also necessary to generate a large decision tree that determines the signature of the original query before that query is dispatched to the appropriate function.

5.2 The Query Signature Graph

Figure 8 is a graphical representation of the functions that would be generated for the checklets and transformations in our running example. The nodes depict the signature-bound functions and the edges show transformations from one signature to another. The shaded nodes are those nodes that have associated checklets.

To construct this query signature graph we start with those signatures accepted by checklets — they are trivially acceptable. Then, for all of those signatures, we apply the *inverted* transformations wherever possible. I.e., at each step of this process we determine those signatures that are one transformation away from the given acceptable signature. By repeatedly applying these inverted transformations, all acceptable query transformations can be discovered and the graph can be constructed.

There is, however, one unfortunate complication to this architecture. Consider the following example:

$$
\begin{array}{ll}
\multicolumn{2}{l}{\texttt{[a->b,b->c]}} \\
\stackrel{\text{Vector2VectorPair}}{\Rightarrow} & \texttt{([a,b,c],[a->b,b->c])} \\
\stackrel{\text{Vector2VectorPair}}{\Rightarrow} & \texttt{([a,b,c],([a,b,c],[a->b,b->c]))} \\
\stackrel{\text{Vector2VectorPair}}{\Rightarrow} & \texttt{([a,b,c],([a,b,c],([a,b,c],} \\
 & \qquad\qquad\qquad\texttt{[a->b,b->c])))} \\
\Rightarrow & \cdots
\end{array}
$$

In this particular example, the query ⌜`[a->b,b->c]`⌝ (representing a linked list $\langle a,b,c\rangle$) is transformed into ⌜`([a,b,c],[a->b,b->c])`⌝. This is the standard AλgoVista representation of a linked structure, a pair of a node-list and an edge-list. However, the `Vector2VectorPair` transformation can be re-applied to the edge-list in the transformed query, *ad infinitum*.

As it turns out, with any sufficiently rich set of transformations, it is always possible to generate an infinite number of signatures.

To avoid this problem, and to bound the number of signatures, we put a limit on the number of transformations that will be applied to any query. Typical values for this limit is four to six. This would seem to limit the usefulness of precomputed search, but in practice this is not so. First of all, the exhaustive search algorithm in Section 2 is still available to those users who are willing to trade a longer response-time for a more complete response. Secondly, very deep chains of transformations will often transform a query beyond recognition, resulting in spurious query results that have little meaning to the user.

The generation of the decision tree and all of the signature-specific functions is done automatically at checklet upload-time by a small Icon program [19].

5.3 Simple Indexed Checking

Figure 9 shows a third search algorithm, a simple extension of the exhaustive algorithm. The idea is to create an *index* mapping a particular query signature to the list of checklets that have this signature. `search(`Q`)` generates the same set of mutations of Q as the exhaustive algorithm does, but only applies these to checklets with the appropriate signature. As we will see in Section 9, this simple algorithm often performs well in practice.

6 Checklet Construction

To extend the AλgoVista database with new problem classifications, a user downloads a checklet template, modifies and tests it, and uploads the new checklet into the server where it is added to the checklet database. The template is a Java class consisting of four methods that the user should fill in: `Description()` gives a short title of the problem, `ProtoExamples()` provides a list of accepting example queries, and `References()` gives links to web resources for the problem. The main method is `Check()` which takes a parsed query as input and returns `true` if the query could be an instance of the problem the checklet checks for.

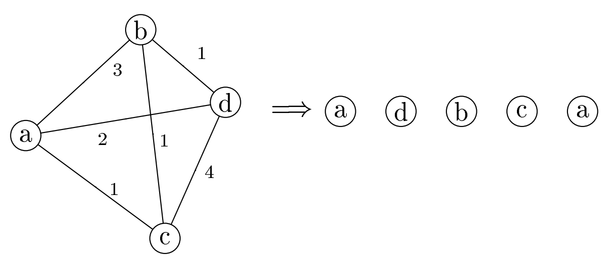
Figure 10 shows a simple sorting checklet that runs in expected linear time.

Figure 11 shows a checklet that classifies Eulerian graphs. It suffices to check that the graph is undirected, connected, and that each node is of degree 2. The only difference between a checklet that classifies a problem and a checklet that classifies a combinatorial structure lies in their input arguments: the former takes an *input⇒output* structure as argument, whereas the latter only takes the combinatorial structure itself. In fact, it is possible to write checklets to classify *any* mathematical object, as long as it can be expressed in AλgoVista's query language. It would, for example, be easy to add checklets to AλgoVista that classify mathematical constants, much as *Plouffe's Inverter* [25] does.

6.1 Checking hard problems

Much research has gone into the search for efficient result checkers for many classes of problems. In some cases, efficient result checkers are easy to construct. For example, let $P(x)$ return a factor of the composite integer x. This is generally thought to be a computationally difficult problem. However, checking the correctness of a result returned by P is trivial; it only requires one division. On the other hand, let $P(x)$ return a least-cost traveling salesman tour of the weighted graph x. Checking that a given tour is actually a minimum-cost tour is as expensive as finding the tour itself.[2]

For example, consider the following query:



⌜`[a--/3b,a--/1c,a--/2d,b--/1c,b--/1d,c--/4d]==>[a,d,b,c,a]`⌝.

A TSP checklet would have to compute a minimum cost tour of the graph and compare this to the cost of the tour suggested by the query. The checklet should accept only if the costs are the same. For even moderately large graphs this would be prohibitively expensive.

Instead, for hard problems we are often satisfied writing *relaxed* checklets. These are checklets that "cheat", either in order to run faster or to avoid having to implement a complex algorithm. For example, instead of computing a minimum cost tour, a TSP checklet could use a quick-and-dirty approximation algorithm and accept whenever the cost of the user's tour is "close enough" to the cost of this approximate tour. A tour based on the minimum spanning tree of the graph, for example, is within a factor of 2 of optimal and can be computed in $\Theta(V^2)$ time [14].

Of course, a relaxed checklet trades precision for efficiency: it *accepts* more often than it should. This is often a reasonable trade-off since – like all search engines – we expect AλgoVista to sometimes return false positives.

If fact, we can think of a hierarchy of progressively more relaxed TSP checklets, starting with the most expensive and exact algorithm, and ending with one that will always accept the *input⇒output* query:

1. Accept only if the *output* forms a minimum cost tour of the *input* graph.

[2] Note that this optimization formulation of TSP is different from the decision problem normally found in textbooks. In the decision problem $\mathrm{TSP}_d = \langle G,k,T\rangle$ we are asked to verify that the tour T is a hamiltonian cycle of G with cost of most k. This can certainly be done in polynomial time. In the optimization problem $\mathrm{TSP}_o = \langle G,T\rangle$, on the other hand, we are asked to verify that T is a minimum cost tour of G, which is hard.

2. Accept only if the *output* tour has a cost that is close enough to that of an approximate MST-based tour of the *input* graph.
3. Accept only when the *output* forms a hamiltonian cycle of the *input* graph.
4. Accept only when the *input* is a weighted complete graph and the *output* is a list of nodes.
5. Accept always.

Another point on this spectrum is *spot-checking* [17], a recent development in result checking that allows hard problems to be checked probabilistically.

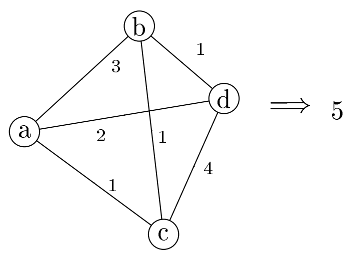
Just as checklet writers can cheat with relaxed checklets, AλgoVista users can cheat by submitting relaxed queries. From a user's point of view they would like to submit the simplest (least specific) query possible that will produce the answer they are looking for, and just a few false positives. For example, the user would be very pleased if their relaxed query ⌜`[a--/1b]==>[a,b,a]`⌝ was specific enough to return TSP and only a few "wrong" hits.

Point 4 in the list above is an interesting case which we call *signature search*, and which is supported by AλgoVista. If a user enters the query ⌜`[a--/1b]==>[a]`⌝ in signature search mode they will receive a list of all problems in the database which map a weighted undirected graph to a list of nodes. If only a few problems have this signature such relaxed queries can be very effective as well as extremely fast to compute. We will discuss this further in Section 11.1.

It is worth noting that the queries that users submit to AλgoVista are almost always extremely short. Therefore, in many cases checklets do not have to worry about being efficient; the simplest, most straight-forward algorithm will often be adequate.

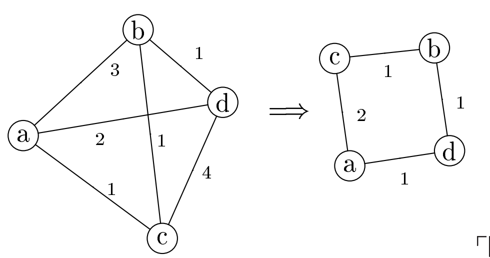
6.2 Query variations

Most problems come in different flavors, and AλgoVista needs to contain checklets for all of them. A user looking for the TSP problem could enter the query above, but they might just as well enter



⌜`[a--/3b,a--/1c,a--/2d,b--/1c,b--/1d,c--/4d]==>5`⌝.

(i.e., "the length of a minimum tour of this graph is 5"), or



⌜`[a--/3b,a--/1c,a--/2d,b--/1c,b--/1d,c--/4d]==>`
`[a--/1d,d--/1b,b--/1c,c--/2a]`⌝.

and so forth.

Certain query variations are taken care of by AλgoVista itself, through its transformation engine. For example, AλgoVista will automatically convert graphs between different data representations, such as *list of edges*, *adjacency matrix*, and *adjacency lists*. As a result, if the TSP checklet expects its input graph in the form of an adjacency matrix but the user submits a query using a list of edges representation, the transformation engine would automatically make the appropriate conversion and the checklet would still accept. For cases when the built-in transformations are not enough, the checklet writer has to provide the appropriate variations.

6.3 Checking problems on reals

Checklets that operate on floating-point numbers present their own set of problems concerning floating-point equality. For example, which, if any, of the queries

$$\ulcorner 2.0 \Rightarrow 1.4142135623 \urcorner,$$
$$\ulcorner 2.00000 \Rightarrow 1.4140 \urcorner, \text{ and}$$
$$\ulcorner 2.0 \Rightarrow 1.0 \urcorner$$

should a *floating-point square root* checklet accept? In all cases, the right hand side *is* an approximation of $\sqrt{2}$, but just how accurate should the approximation be in order to be acceptable to the checklet? Checklets can, of course, define their own equality primitives, but AλgoVista provides a default heuristic that works well in most situations: floating-point comparisons are done in the *minimum* precision of any floating-point number in the input query. Hence,

$$\ulcorner 2.0 \Rightarrow 1.4142135623 \urcorner$$

will accept (since $1.4142135623^2 = 1.9999999997 \approx 2.0$ when comparing with a precision of one decimal digit), but

$$\ulcorner 2.00000 \Rightarrow 1.4140 \urcorner$$

will not (since $1.4140^2 = 1.999396 \not\approx 2.0000$ when comparing with four digits' precision).

For example, in the checklet below, the standard AλgoVista function `Float.equals()` will compare `s` and `t*t` in the minimum of the precisions of `s` and `t`:

```
public class FltSqrt extends Float_Float {
   ...
   public boolean Check (
        double s, double t,
        int sprec, int tprec) throws Throwable {
      return Float.equals(s, t*t, sprec, tprec);
   }
}
```

7 Visual Queries

The textual query language described in Section 3 is extremely simple. However, our experience has been that many casual users of the AλgoVista search engine do not take the time to study the available documentation to be able to produce correct queries. The following examples are some actual queries from our query log.

Many users will start by entering a few keywords, thinking that, like most other search engines, AλgoVista is keyword based:

⌜`create heap`⌝
⌜`quadratic knapsack problem`⌝
⌜`computer science`⌝

In spite of the simplicity of the query language many queries are still ill formed:

⌜`(1)==>3 `⌝
⌜`(1,2,3,4,5)==>3 `⌝
⌜`()------>()`⌝
⌜`a + b = c`⌝
⌜`array => heap`⌝

Some queries just make very little sense at all:

⌜`7,5,5,6,9,1 Sort these numbers`⌝
⌜`How to implement the dijkstra's algorithm in java??`⌝

Occasionally, someone will do a “vanity search” to see if their name is in the database:

⌜`viet anh nguyen`⌝

One user even drew a pictorial query using ASCII graphics:

```
       *
      / \
     /   \
    /     \
   *       *
  / \     /
 /   \   /
*     * *
```

From these examples and others like it in our query logs, we have drawn the conclusion that for a specialized search engine such as AλgoVista to become successful its user interface has to be so simple and self-explanatory that virtually no user training is necessary. For this reason we designed a visual query language and a graphical user interface that allows users to *draw* their queries rather than enter them textually.

The design of the visual query language and the user interface is based on the following principles:

1. Give the user complete freedom in drawing her query, because no web user will take the time to learn complex visual grammars.

2. When there is more than one possible parse of a visual query let the user herself choose the appropriate one. The user will have no desire to learn any semantic constraints that may have been embedded in the interface.

3. Make each visual query a learning experience. The user may not have the time or patience to read the documentation describing the textual query language, but we can force them to learn it surreptitiously by showing them the textual query corresponding to each visual query.

4. Provide plenty of example queries that the user can learn from, modify, and experiment with.

7.1 The Visual Query Language

AλgoVista's visual query language is closely modeled on its textual counterpart. A user constructs a query by dragging primitive elements from a *template region* on the user interface (Figure 2) onto the drawing canvas. Atoms are modeled by named circles, links by the obvious lines and arrows, booleans, matrices, and the `==>`-arrow by themselves, and numbers are entered by clicking and typing. Geometric queries are entered by drawing circles, ellipses, points, lines, and polygons directly on a Cartesian grid. There are, however, no obvious visual counterparts to the pairs and element-lists of the textual language. These are instead inferred from the positioning of the visual elements.

For example, instead of entering a traveling salesman query textually:

```
[a--/3 b,a--/2 d,a--/1 c,c--/1 b,b--/1 d,d--/4 c] ==> 5
```

it could instead be drawn like in Figure 2. The textual query is inferred from the drawing, and the English prose query is derived from the textual query.

Recursive queries could be handled in a variety of ways. Many graph editors parse node and edge labels by *proximity*; that is, a graphical element is inferred to be the label of a node a if it is “close enough” to a. This puts a heavy burden on the parser as well as on the casual user who needs to have some understanding of the principles under which the parser operates.

We have instead opted for a much more lightweight solution: if the user double-clicks on an atom or link a new, simpler, drawing window opens up, allowing the user to enter the sub-drawing. This strategy has the advantage of both being simple to implement and trivial to explain to the user. It is now easy to create arbitrarily complex queries, where, for example, the nodes of a graph could be labeled with lists of trees, whose edges are labeled with.... See Figure 12 for an example.

7.2 Parsing Visual Queries

Because of the limited set of graphical elements that AλgoVista supports, parsing visual queries is relatively straight-forward. The `==>`-arrow separates inputs from outputs, which means that anything to the left of the arrow is an element of the input, anything to the right belongs to the output.

As we have seen, recursive queries are created by the user explicitly opening up atoms and links (by double-clicking on them) and drawing the label in a sub-editor pane. This limited amount of structure editing greatly simplifies parsing, since it is always clear if an element is the label of an atom or link, rather than a free-standing element.

The main challenge in parsing visual queries is that grouping of elements is not always evident from the drawing. Consider the queries in Figure 13. There are four connected components and it is unclear from the query which of those, if any, form sub-groups. For example, the nodes `u`, `s`, and `t` could form a (dis-connected) graph, or `s` and `t` could form one graph and `u` another.

Many visual parsers use proximity to infer element grouping. In a web-situation where many casual users will visit a site for only a minute or two, there simply is no time to explain what heuristics the parser employs. So, again, we prefer a lightweight and user-centric solution. We will simply guess the user's intentions, and then report back what that guess was. In this case, the parser's first guess (shown at the top left of Figure 13) was that each connected component is a unique argument in the query. It indicates this by drawing a convex hull (red lines) around each component.

If, however, the user's intentions were different, she can simply ask the parser to produce a different parse, by clicking on a PARSE-button one or more times. This will cycle through all the different possible parses of the query, each described visually (using convex hulls), formally (using a textual AλgoVista query), and informally (using English prose). For example, it could be that the user had planned for the node `u` to belong to a three-node graph (bottom left of Figure 13) or for the two integer elements to be part of a pair (top right of Figure 13), or for `u` and `456` to form a pair (bottom right of Figure 13).

In most cases there are few possible parses and it is immediately clear to the user which is the one she is looking for. To cut down the number of possible parses we can exploit the fact that, as described in Section 3.1, AλgoVista queries are typed. During parsing we may find that certain groupings of elements do not typecheck, in which case we never present them to the user.

The above discussion is summarized by the algorithm in Figure 14. `parse` generates a sequence of possible interpretations of the graphical elements on the canvas. We first separate the input from the output elements, and then construct a set of connected components for each. We then generate all possible type-correct queries by merging adjacent connected components. Finally, each query is translated to English and presented to the user, along with the textual query and a convex hull around each component.

8 Security Issues

To the best of our knowledge, AλgoVista is the first web site to allow arbitrary users to upload executable code into its database. Obviously, there are a number of security issues that have to be addressed.

Figure 15 shows some examples of hostile checklets. Figure 15 (a) shows an overly general checklet `evil1` that was uploaded in an attempt to promote someone's web site. Regardless of the input query, `evil1` will always accept and return a link to the bogus site. Checklet `evil2` in Figure 15 (b) launches a *denial-of-service* attack by stealing as many CPU cycles or as much memory as possible. Checklet `evil3` in Figure 15 (c) attempts to compromise the security of the AλgoVista server by reading from or writing to the local file system.

AλgoVista checklets are written in Java and are executed with the same security privileges as Java applets. This allows us to rely on Java's built-in security features to prevent checklets from compromising the security of the AλgoVista server.

Denial-of-service attacks [21] are more difficult to deal with. While time-outs are used to stop checklets from stealing too many CPU cycles, Java provides no straight-forward way to limit the dynamic memory allocation of a process. Some recent results (for example, Binder [5]) implement resource accounting using bytecode modification, but this is not yet common place.

It is unclear whether there are any strong technical means to prevent attacks by overly general checklets. The same problem plagues keyword search engines such as AltaVista where unscrupulous users will

> "submit pages with numerous keywords, or with keywords unrelated to the real content of the page" [1].

in order to promote their own web-pages. Currently, we require every checklet to provide a list of *accepting examples*:

```
checklet intAdd ((int a, int b) ⇒ int c)
   examples (0,0)==>0, (5,6)==>11
   if (a+b)=c then accept ···
```

When a checklet is uploaded the AλgoVista server ensures that

a) the checklet accepts every one of the example queries it has provided, and

b) the checklet only accepts a *small fraction* of all the example queries provided for all other checklets in the coop.

While not foolproof, these policies provide a reasonable level of security.

9 Evaluation

In order to be useful a search engine should

1. be simple to use,
2. have a reasonable response time,
3. return results that are "useful" to the user, and
4. return few false positive results.

Obviously, AλgoVista is not as easy to use as a keyword-based search engine, since queries have to be expressed in a semi-formal manner. As we will see below, most AλgoVista queries can be answered in a few seconds. "Usefulness" is a more difficult property to measure. In case of AλgoVista we could define it as "the ability to compute relevant results that the user could not have found using other, simpler, search tools." In other words, is the extra effort on the part of the user to formulate *input*==>*output* queries worth the trouble? So far, we have performed no formal surveys of actual users' experience with the system.

9.1 Query Times

Table 4 shows the search times for some typical queries. The times were collected by running each query five times and averaging the wall clock times of the last four runs. The reason for discarding the first measurement is that Java start-up times are quite significant and unpredictable. Furthermore, in web applications such as this one, programs are typically pre-loaded into (a large) primary memory and queries are fielded without any disk accesses. The measurements were collected on ...

Timing measurements. Times are in seconds. Anomalous measurements are due to rounding errors and inadequate timer resolution. The measurements were collected on a lightly loaded Sun Ultra 10 workstation with a 333 MHz UltraSPARC-IIi CPU and 256 MB of main memory, running AλgoVista on Sun JDK 1.2.1.

For each algorithm, Table 4 gives five columns that show the *mutation level*, the *number of transformations executed during the search*, the *the number of generated mutations*, the *number of checklets executed*, the *number of generated query results*, and the *average wall clock time of the query*, in milliseconds.

From Table 4 we can draw the conclusion that, even though the precomputed search will in general execute fewer transformations and checklets than the other algorithms, in practice the indexed algorithm is faster. This is due to the large size of the internal tables used by the precomputed algorithm. For exampple, Table 5 shows that, for mutation level 5, almost 22MB are used to store the signature graph. This graph is realized as a two-dimensional integer array. Such arrays are stored non-contiguously on the heap in Java, and thus exhibit poor locality. We believe that using standard table compression techniques as well as recoding the table indexing part of AλgoVista in a lower level language such as C, would improve performance significantly.

We expect that as the system grows with more checklets and query transformations, the performance of the precomputed search algorithm will exceed that of the other algorithms. The reason is that the worst-case execution time of the exhaustive and indexed algorithms for a query Q is

$$O(\texttt{\#mutations}(Q) \times \texttt{\#checklets})$$

	Exhaustive					Indexed					Precomputed				
M	**ET**	**GM**	**EC**	**GR**	**WT**	**ET**	**GM**	**EC**	**GR**	**WT**	**ET**	**GM**	**EC**	**GR**	**WT**
`((1,2),(3,4))==>(4,6)`															
1	10	1	360	2	34	10	1	41	2	6	15	20	58	2	49
2	90	45	16116	2	893	90	45	727	2	98	182	84	361	3	180
3	618	1727	153579	3	†	2258	1727	42114	3	2886	539	100	425	3	1131
4	5662	62150	143569	1	†	5662	62370	146614	3	†	682	100	425	3	5082
5	5662	62370	143121	1	†	5662	62274	147639	3	†					
`(1,2)==>3`															
1	5	1	360	3	33	5	1	39	3	6	6	4	57	3	41
2	31	10	3579	3	193	31	10	142	3	18	28	8	98	4	102
3	119	44	15687	4	810	119	44	498	4	83	31	8	98	4	313
4	119	44	15687	4	819	119	44	498	4	79	31	8	98	4	1068
5	119	44	15687	4	817	119	44	498	4	82					
`Graph([e,d,c,a,b],[e--a,e--d,c--d,c--b,a--b])`															
1	32	13	4490	18	326	32	13	67	18	102	0	1	13	8	75
2	32	13	4490	18	326	32	13	67	18	102	0	1	13	8	96
3	32	13	4490	18	328	32	13	67	18	105	0	1	13	8	252
4	32	13	4490	18	327	32	13	67	18	99	0	1	13	8	857
5	32	13	4490	18	329	32	13	67	18	97					
`[e--a,e--d,c--d,c--b,a--b]`															
1	49	20	6905	18	453	49	20	232	18	111	0	1	1	0	37
2	49	20	6905	18	460	49	20	232	18	111	8	3	3	0	63
3	49	20	6905	18	454	49	20	232	18	111	8	3	3	0	220
4	49	20	6905	18	458	49	20	232	18	111	8	3	3	0	835
5	49	20	6905	18	460	49	20	232	18	110					
`[n,k,l,i]==>[i,k,l,n]`															
1	3	1	360	2	37	3	1	21	2	6	0	1	12	2	41
2	12	6	2150	2	126	12	6	116	2	19	3	2	22	2	77
3	12	6	2150	2	124	12	6	116	2	21	3	2	22	2	285
4	12	6	2150	2	163	12	6	116	2	19	3	2	22	2	1073
5	12	6	2150	2	125	12	6	116	2	19					

Table 4: Timing results of the three proposed search algorithms on some representative queries. M=*mutation level*, ET=*number of executed transformations*, GM=*number of generated mutations*, EC=*number of executed checklets*, GR=*number of generated results*, WT=*wall clock time (milliseconds)*, †=*search timed out after 20 seconds*.

level	table sizes (bytes)	checklet signatures	query signatures
1	15,990	83	552
2	96,676	83	2,469
3	488,862	83	9,949
4	4,621,228	83	39,495
5	21,555,016	83	159,318

Table 5: Table sizes for the precomputed search algorithm, and the number of unique query and checklet signatures for different mutation levels.

while the execution time of the precomputed search algorithm is

$$O(\texttt{\#viable\_mutations}(Q)),$$

where we expect

$$\texttt{\#viable\_mutations}(Q) \ll \texttt{\#mutations}(Q).$$

9.2 The Visual Interface

AλgoVista's textual interface and its underlying database engine have been operational for some time. The design of the graphical interface was initiated by the fact that our query logs indicated that users had trouble forming correct textual queries. We have yet to perform a formal usability study so we cannot conclusively say whether users will favor one interface over another. Our own experience with the graphical interface, however, has allowed us to make the following observations:

1. For large, disconnected, inputs, the number of possible parses grows exponentially. For example, consider the following query, which a user might have entered to look for information about palindromes:

   m a d a m

   This query generates a large number of parses, making it time consuming for the user to navigate to the desired one. For most queries this is not an issue, since they tend to contain a small number of connected components. For queries with many components, the textual search interface often works better. For example, the following is an unambiguous palindrome query:

   ```
   [m, a, d, a, m]
   ```

2. Generally, a query is easy to express if the query language provides the appropriate primitives. For example, since AλgoVista provides node and link primitives it is straight-forward to create queries involving graphs. On the other hand, classifying three-dimensional problems in the Computational Geometry domain is difficult, since AλgoVista currently only supports two-dimensional geometric queries.

10 Related Work

A number of web sites, for example the *CRC Dictionary* [6], Skiena's Stony Brook Algorithms Repository [30], and the *Encyclopedia of Mathematics* [34], already provide encyclopedic information on algorithms, data structures, and mathematical results. These sites typically classify and organize algorithms hierarchically. Like all encyclopedias, however, they are of no use to someone unfamiliar with the terminology of the field they are investigating.

AλgoVista provides an alternative mode of searching these repositories (to which we link heavily). Most importantly, we allow users to express their problem in their own terms, as an example of its required behavior. We then attempt to match this behavior (through the judicious use of query transformations) against the known behavior of problems in our database.

More relevant to the present research is *Sloane's On-Line Encyclopedia of Integer Sequences* [31]. This search service allows users to look up number sequences without knowing their name. For example, if a user entered the sequence ⌜1,2,3,5,8,13,21,34⌝, the server would respond with "Fibonacci numbers." It is interesting to note that, although many of the entries in the database include a program or formula to generate the sequences, these programs do not seem to be used in searching the database. Similar search services are *Plouffe's Inverter* [25] where one can classify interesting numbers and the *Encyclopedia of Combinatorial Structures* [24].

MELDEX [27, 2] is an online index of tunes which can be searched by example. A user queries the database by singing (or humming, or playing) a few notes which are then matched against a database of 9,400 folk songs.

A number of search engines on the web use query-by-example for text documents and bibliographic data. WebBird [33], for example, allows users to enter the URL of a page and it will return "similar documents by following citation paths that pass through those given documents." Search engines such as Google have similar features allowing users to narrow their search results by looking for documents similar to ones the engine has already returned.

Much work has been done on developing systems for classifying and searching for reusable software components. These tend to use complex semantic classification schemes (for example the LaSSIE [15] system) or, at the opposite end of the spectrum, simple query-by-keyword.

Inductive Logic Programming (ILP) [4] is a branch of Machine Learning. One application of ILP has been the automatic synthesis of programs from examples and counter-examples. For example, given a language of list-manipulation primitives (`car, cdr, cons, and null`) and a set of examples

```
append([],[],[]).
append([1],[2],[1,2]).
append([1,2],[3,4],[1,2,3,4]).
```

an ILP system might synthesize the following Prolog-program for the `append` predicate:

```
append(A, B, B) :- null(A).
append(A,B,C) :-   car(A, X), cdr(A, Y),
                   append(Y, B, C1),
                   cons(X, C1, C).
```

Obviously, this application of ILP is far more ambitious than AλgoVista. While both ILP and AλgoVista produce programs from *input⇒output* examples, ILP *synthesizes* them while AλgoVista just retrieves them from its database. The ILP approach is, of course, very attractive (we would all like to have our programs written for us!), but has proven not to be particularly useful in practice. For example, in order to synthesize *Quicksort* from an input of sorting examples, a typical ILP system would first have to be taught *Partition* from a set of examples that split an array in two halves around a pivot element:

```
partition(3,[],[],[]).
partition(5,[6],[],[6]).
partition(7,[6],[6],[]).
partition(5,[6,3,7,9,1],[3,1],[6,7,9]).
```

AλgoVista is essentially a *reverse definition* dictionary for Computer Science terminology. Rather than looking up a term to find its definition (as one would in a normal dictionary), a reverse definition dictionary allows you to look up the term given its definition or an example. The *DUDEN* [12] series of pictorial dictionaries is one example: to find out what that strange stringed musical instrument with a hand-crank and keys is called, you scan the *musical instruments* pages until you find the matching picture of the *hurdy-gurdy*. Another example is *The Describer's Dictionary* [18] where one can look up ⌜mixture of gypsum or limestone with sand and water and sometimes hair used primarily for walls and ceilings⌝ to find that this concoction is called *plaster*.

Rekers [26] provides a nice overview of graphical parsing algorithms. They note that most graph parsing algorithms are worst-case exponential. The paper also presents a new multi-stage graph parsing method with separate phases for determining object locations and spatial relationships, and a final grammar-based rewrite phase. In a web-based visual interface such complex, and potentially slow, parsing methods are unacceptable.

In [22], a visual interface to a CASE tool is presented, where boolean queries are constructed in a syntax-directed fashion. Users proceed top-down from a "root" query, iteratively expanding non-terminal nodes until the query is complete.

Novice (as well as expert!) users typically find syntax-directed iterative refinement cumbersome to use. There is a reason why programmers prefer free-form editors like `emacs` over syntax-directed ones, even though the latter ensures that only correct programs can be constructed. For this reason, AλgoVista's visual interface is a mostly free-form graph editor, and syntax-directed editing is reserved for recursive edits.

The web ought to present many opportunities for introducing more people to direct-manipulation interfaces. However, we have found few such examples. Marmotta [10], a graphical front-end to online databases, is an exception.

11 Searching Using Query-By-Example

We are constantly expanding the AλgoVista database of checklets and query transformations. We are also adding new (non-keyword based) methods of searching for information related to Computer Science. For example, future versions of AλgoVista will have the ability to search for abstract data-types and data structures by example, and by *complexity*. For example, it would be useful to be able to ask:

> *"show me all data structures that support the operation*
>
> ⌜`[(a,5),(h,10),(g,2),(f,9)]==>g`⌝
>
> *(*`extract-min`*) (in* $O(\log n)$ *worst-case time) and the operation*
>
> ⌜`([(a,5),(h,10),(g,2),(f,9)],h)==>[(a,5),(h,9),(g,2),(f,9)]`⌝
>
> *(*`decrease-key`*) (in* $O(1)$ *worst-case time)"*

and have AλgoVista return links to information about *relaxed heaps* [16]. We are also planning to integrate the large databases of boolean functions used in the VLSI community [3, 11] into AλgoVista. These libraries are typically searched by hashing or by signature and would fit in well with our current search strategies.

A more long-range goal is to extend AλgoVista to search for combinations of problems. For example, if AλgoVista knows about sorting and binary search, and is given a query

⌜`(45,[1,6,45,9,33])==>true`⌝

it should answer

> *"This looks like an instance of the* `search` *problem. You can either solve this in* $O(n)$ *using a* `linear search` *or if you combine* `sorting` *and* `binary search` *you can do the search in* $O(\log n)$ *time."*

Unfortunately, program checking does not appear to be strong enough to support this type of query.

We are also interested in applying the ideas developed for AλgoVista — in particular the query-by-example search engine and the web-based graphical query interface — to areas outside of Computer Science. For this reason we are working on transforming the current implementation into a generic toolkit that can be easily specialized to different application domains.

Below we outline three novel *query-by-example* strategies which are being added to AλgoVista, namely searching checklets by signature, searching the Java APIs by signature, and searching for complex regular expressions by example.

11.1 Searching Checklets by Signature

A simple variant of AλgoVista looks only at the type signature of a query when trying to match it to checklets. The signature of each checklet is determined at upload time (a checklet that accepts queries of the type ⌜`(1,2)==>3`⌝ would have the signature ⌜`Map(Pair(Int,Int),Int)`⌝, for example), and stored with the checklet in the database. When a user submits a query such as ⌜`(1,2)==>3`⌝ it is trivial for AλgoVista to determine the type of the query and match it against the signatures of each of the checklets in the database.

Such *quick-and-dirty* searches are favored by users because queries are easier to form and the search is much faster. The disadvantage is that the search is much less precise and may therefore generate many false positive results.

There are actually two ways of searching AλgoVista by signature. The easiest one (and the one that most users would use) is simply to issue an AλgoVista query such as ⌜`([1,3],[2,4])==>[1,2,3,4]`⌝, selecting *search-by-signature* rather than *search-by-value*. AλgoVista returns a list of all the problems in the database that map a pair of integer vectors to an integer vector, for example list-append and list-merge. But, what if we want to look for problems that map a pair of *any* two vectors to a vector, regardless of what type of data they contain? AλgoVista will also accept queries directly in a signature format, such as

```
Map(Pair(Vector(Any(a)),Vector(Any(a))),Vector(Any(a))).
```

In this query ⌜`Any(a)`⌝ represents an arbitrary type which we've called "a". The query will match any problem that maps a pair of vectors to a vector, as long as all three vectors have the same element types.

11.2 Searching the Java API by Signature

`JavaDoc` is a standard Java tool which produces HTML from stylized comments in a set of Java classes. The resulting HTML files essentially constitute a small hierarchical database which can be traversed starting at the top of the class tree. The standard Java library, for example, can be browsed at `http://java.sun.com/products/jdk/1.3/docs/api`.

AλgoVista has the ability to extract from JavaDoc-generated HTML the names and signatures of Java methods, as well as the HTML labels of each method. This information is then indexed and can be searched by signature from the AλgoVista homepage.

Searching by signature is remarkably powerful and convenient, particularly when faced with questions such as

> I wonder if the Java class library has a method that converts a string to an array of bytes?

Submitting the query

```
(string)==>byte[]
```

to AλgoVista will return a link directly to Sun's documentation for

⌜`java.lang.String.getBytes()`⌝.

This method could just as well have been declared in `java.lang.Byte`, and it could just as well have been named `getByteArray` or `string2bytes`, and if it wasn't for AλgoVista there would be no way for a programmer to locate this method other than through a more or less random traversal of Sun's HTML documentation.

AλgoVista has a very simple query language for searching the Java APIs, with a syntax similar to the one used when classifying algorithmic problems and combinatorial structures. A query consists of a tuple of one or more input types and one output type:

$$(type_{i_1}, type_{i_2}, \cdots, type_{i_n}) \quad ==> \quad type_o.$$

Each *type* can be

1. the name of a Java built-in type (`int`, `float`, etc.),
2. the name of a Java class (`String`, `java.lang.String`, etc.),
3. a type followed by a pair of brackets (`int[]`, `java.lang.String[]`, etc.) to indicate an array parameter,
4. `*` (matching zero or more arbitrary types),
5. `?` (matching exactly one arbitrary type), or
6. `+` (matching one or more arbitrary types).

One small complication arises due to the fact that Java has both static and virtual methods. Consider a method `append` that concatenates two strings into a third string. Either of the following two declarations would be possible:

```
static String append (String first, String second) {
   return first + second;
}

String append (String second) {
   return this + second;
}
```

In the static case both arguments are explicit (the call ⌜`append("1st","2nd")`⌝ would append the strings `"1st"` and `"2nd"`), while in the virtual case the first string argument is implicit in `this` (a corresponding call would be ⌜`"1st".append("2nd")`⌝. In AλgoVista we treat virtual methods as if they were static, i.e. we make `this` an explicit parameter. This allows a user to enter the query

```
(string,string)==>string
```

when searching for an `append` method without having to know if the actual method is static or virtual.

11.3 Searching for Regular Expressions by Example

While many regular expressions are straight-forward to construct, others are definitely non-trivial for novice as well as experienced programmers. Consider, for example, the following RE which matches Uniform Resource Identifiers (URIs):

`^(([^:/?#]+):)?(//([^/?#]*))?([^?#]*)(\?([^#]*))?(#(.*))?`.

An AλgoVista user, on the other hand, only has to submit a query consisting of a random URL, such as `http://algovista.com` and AλgoVista will respond

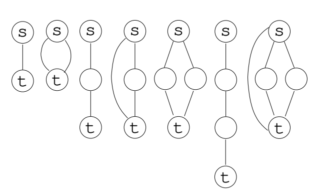
> "This looks like a URI which can be matched by the regular expression
> `^(([^:/?#]+):)?(//([^/?#]*))?([^?#]*)(\?([^#]*))?(#(.*))?`,"

12 Summary

AλgoVista provides a unique resource to programmers to enable them to discover descriptions and implementations of algorithms without knowing theoretical nomenclature. It is important to note that while AλgoVista does not solve all problems for all programmers all of the time, it does solve some problems for which there are no other available resources. For example, if a programmer is looking for information related to *Fibonacci Heaps*, then a Google keyword query ⌜`fibonacci heap`⌝ is the best and simplest strategy. However, if this programmer is in need of a *priority queue* but is unaware of this term, then there is no way for her to form an accurate keyword-based query. It should, however, be possible for the programmer to form an AλgoVista query by providing an example of how a priority queue would work on a particular input.

In fact, the inspiration for the design of AλgoVista came from personal experiences the authors had being unable to classify problems and combinatorial structures:

1. Working on the design of graph-coloring register allocation algorithms, the second author showed his theoretician colleague Sampath Kannan the following graphs:

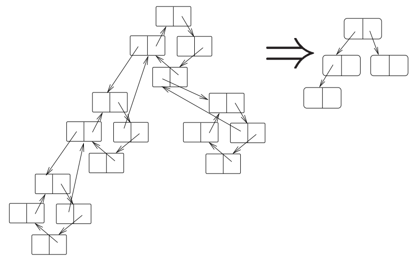


"Do these graphs mean anything to you?" Todd asked.

"Sure," Prof. Kannan replied, "they're series-parallel graphs."

This was the beginning of a collaboration which resulted in a paper in the *Journal of Algorithms* [20].

2. In a similar episode, the first author showed his theoretician colleague Clark Thomborson the following graph-transformation:



"Do you know what I am doing here?" Christian asked.

"Sure," Prof. Thomborson soon replied, "you're shrinking the biconnected components of the underlying graph."

This result became an important part of a joint paper on software watermarking [13].

It is our hope that AλgoVista will prove to be a useful "virtual theoretician" that working programmers can turn to with a problem, quickly sketch it out — visually or textually depending on the nature of the problem — and quickly receive a useful answer.

It is our belief that there are many cases when query-by-keyword is inadequate and where query-by-example is much more intuitive and fruitful. For example, Google keyword queries are language sensitive; queries in languages other than English will only return a fraction of the information available on the web. It is also our belief that — because of the nature of the web and the people who use it — complex formal query languages will never become popular. An interesting alternative is graphical query interfaces such as the one developed for AλgoVista. The generic query-by-example toolkit that we are currently developing based on the AλgoVista system will hopfully make it easy to construct such interfaces for a variety of domains.

Acknowledgments: Will Evans pointed out the relationship between checklets and program checking. Dengfeng Gao, Jessica Miller, Travis Singleton, Zhanwen Li, Gregg Herres, Ursula McDuffee, Zhen Wang, Matthew Johnson, Thaddeus Dombrowski, Kelly Heffner, and Gergely Kota worked on AλgoVista's graphical interface and implemented most of the checklets in the current database. Sean Davey implemented the Java API signature search algorithm. Stephen Kobourov provided much helpful discussion. We thank them all.

References

[1] AltaVista – Adding pages or URLs to the index. `http://www.altavista.com/cgi-bin/query?pg=addurl`.

[2] David Bainbridge. Meldex: A web-based melodic index search service. *Computing in Musicology*, 11:223–230, 1998. Melodic Comparison: Concepts, Procedures, and Applications.

[3] Luca Benini and Giovanni De Micheli. A survey of boolean matching techniques for library binding. *Design Automation of Electronic Systems*, 2(3):193–226, 1997.

[4] Francesco Bergadano and Daniele Gunetti. *Inductive Logic Programming – From Machine Learning to Software Engineering*. MIT Press, 1995. ISBN 0-262-02393-8.

[5] W. Binder, J. Hulaas, A. Villazn, and R. Vidal. Portable resource control in Java: Application to mobile agent security. *Electronic Notes in Theoretical Computer Science*, 63, 2002.

[6] Paul E. Black. Algorithms, data structures, and problems – terms and definitions for the CRC dictionary of computer science, engineering and technology. `http://hissa.ncsl.nist.gov/~black/CRCDict`.

[7] Manuel Blum. Program checking. In Somenath Biswas and Kesav V. Nori, editors, *Proceedings of Foundations of Software Technology and Theoretical Computer Science*, volume 560 of *LNCS*, pages 1–9, Berlin, Germany, December 1991. Springer.

[8] Manuel Blum. Program result checking: A new approach to making programs more reliable. In Svante Carlsson Andrzej Lingas, Rolf G. Karlsson, editor, *Automata, Languages and Programming, 20th International Colloquium*, volume 700 of *Lecture Notes in Computer Science*, pages 1–14, Lund, Sweden, 5–9 July 1993. Springer-Verlag.

[9] Manuel Blum and Sampath Kannan. Designing programs that check their work. *Journal of the ACM*, 42(1):269–291, January 1995.

[10] Fabrizio Capobianco, Mauro Mosconi, and Lorenzo Pagnin. Progressive http-based querying of remote databases within the Marmotta iconic VQS. In *VL'95*, 1995.

[11] Jovanka Ciric and Carl Sechen. Efficient canonical form for boolean matching of complex functions in large libraries. In *Proceedings of the 2001 IEEE/ACM international conference on Computer-aided design*, pages 610–617. IEEE Press, 2001.

[12] Michael Clark and Bernadette Mohan. *The Oxford–DUDEN Pictorial English Dictionary*. Oxford University Press, 1995. ISBN 0-19-861311-3.

[13] Christian Collberg and Clark Thomborson. Software watermarking: Models and dynamic embeddings. In *POPL'99*, San Antonio, TX, January 1999. `http://www.cs.arizona.edu/~collberg/Research/Publications/CollbergThomb%orson99a`.

[14] T. H. Cormen, C. E. Leiserson, and R. L. Rivest. *Introduction to Algorithms*. McGraw-Hill, 1990.

[15] P. Devanbu, R. J. Brachman, P. G. Selfridge, and B. W. Ballard. LaSSIE: A knowledge-based software information system. In *Proceedings of the 12th International Conference on Software Engineering*, pages 249–261, 1990.

[16] James R. Driscoll, Harold N. Gabow, Ruth Shrairman, and Robert E. Tarjan. Relaxed heaps: an alternative to fibonacci heaps with applications to parallel computation. *Commun. ACM*, 31(11):1343–1354, 1988.

[17] Funda Ergün, Sampath Kannan, S. Ravi Kumar, Ronitt Rubinfeld, and Mahesh Vishwanathan. Spot-checkers. In *Proceedings of the 30th Annual ACM Symposium on Theory of Computing (STOC-98)*, pages 259–268, New York, May 23–26 1998. ACM Press.

[18] David Grambs. *The Describer's Dictionary*. W. W. Norton & Company, 1995. ISBN 0-393-31265-8.

[19] Ralph E. Griswold and Madge T. Griswold. *The Icon Programming Language*. Prentice Hall, Englewood Cliffs, NJ, 2 edition, 1990.

[20] Sampath Kannan and Todd A. Proebsting. Register allocation in structured programs. *Journal of Algorithms*, 29(2):223–237, November 1998.

[21] Zheng YL Leiwo J. A method to implement a denial of service protection base. In *INFORMATION SECURITY AND PRIVACY*, volume 1270 of *LNCS*, pages 90–101, Berlin, Germany, 1997. Springer.

[22] Hui Liu. A visual interface for querying a CASE repository. In *VL'95*, 1995.

[23] Mehlhorn and Naher. From algorithms to working programs: On the use of program checking in LEDA. In *MFCS: Symposium on Mathematical Foundations of Computer Science*, 1998.

[24] Stéphanie Petit. Encyclopedia of combinatorial structures. `http://algo.inria.fr/encyclopedia`.

[25] Simon Plouffe. Plouffe's inverter. `http://www.lacim.uqam.ca/pi`.

[26] J. Rekers and A. Schür. A graph grammar approach to graphical parsing. In *VL'95*, 1995.

[27] David Bainbridge Rodger J. McNab, Lloyd A. Smith and Ian H. Witten. The new zealand digital library MELody inDEX. *D-Lib Magazine*, May 1997. `http://www.dlib.org/dlib/may97/meldex/05witten.html`.

[28] Ronitt Rubinfeld. Batch checking with applications to linear functions. *INFORMATION PROCESSING LETTERS*, 42(2):77–80, May 1992.

[29] Ronitt Rubinfeld. Designing checkers for programs that run in parallel. *ALGORITHMICA*, 15(4):287–301, April 1996.

[30] Steven S. Skiena. Who is interested in algorithms and why? Lessons from the Stony Brook algorithms repository. In *Worskop on Algorithm Engineering, WAE'98*, 1998.

[31] Neil J. A. Sloane. Sloane's on-line encyclopedia of integer sequences. `http://www.research.att.com/~njas/sequences/index.html`.

[32] Hal Wasserman and Manuel Blum. Software reliability via run-time result-checking. *Journal of the ACM*, 44(6):826–849, November 1997.

[33] Webbird. `http://ai.iit.nrc.ca/II_public/WebBird/resQueryByExample.html`.

[34] Eric Weisstein. Encyclopedia of mathematics. `http://www.treasure-troves.com/math`.

A The AλgoVista Database of Problem Descriptions

3-Colorable Graph, A Basis for the Null Space of the Matrix, A Binomial Tree, Absolute value, Achromaitc Number problem, Ackerman's Function, Activity-Selection Problem, Addition of Polynomials, Addition of Rational Numbers, Addition of two n-D vectors, A directed Graph and its longest path, Adjugate Matrix, All pairs shortest path, Altitude Of A Triangle, Altitude of Trapizoid, A mapping from a Graph to a set representing the Max-Cut problem, A mapping identifying a directed graph and its longest path, An Acute Angle, Angle Bisector, Angle of a Regular Polygon, Angle Sum Formula of a Regular Polygon, Angle Sum of a Polygon, Annulus, A pair containing a graph and its max cut set, Arbitrated Digraph, Arc cosine, Area of a Circle, Area of a Rectangle, Area of a Trapezoid, Area of a Triangle, Area of Parallelogram, Area of Triangle, Arithmetic Sequence, Articulation points, A Symmetric Graph, Average number, Average number of a list of floating point numbers, Average number of a list of integers, AVL Tree, Balancek Complete Bipartite Subgraph, Bandwidth, Bandwidth Problem, Biconnected Graph, Binary logical and, Binary logical or, Bipartite Graph, Central Angle, Central Angle Regular Polygon, Chord Of Circle, Circle, Circumcenter of Polygon, Circumcircle of Triangle, Circumference, Circumference of a Circle, Clique, Clique problem, Collapsed Graph, Combination, Common External Tangent, Common Internal Tangent, Compare less-than, Complementary Angles, Complete graph, Complex Addition, Concave, Concentric Circles, Congruent Circles, Connected graph, Continuous Knap Sack Problem, Contraction, Contraction of Circles, Convex, Convex Combination, Convex Combination of two points, Cosine of the angle, Covering by Clique Problem, Cross, Cross Product, Cross Product of two 3-D vectors, Cubic Subgraph Problem, DAG, Decagon, Degree Bounded Connected Subgraph, Derangement of an ordered set, Determinant of a Matrix, Diagonal, Diameter Of Circle, Digon, Directed Acyclic Graph (DAG), Directed Graph, Directed Linked List, Directed Tree, Disjoint Paths Problem, Distance between two points, Division of two complex numbers, Domatic Number problem, Dominating Set Problem, Dot Product of an n-D vector, Duality Principle, Equality of Polygon, Equivalence Classes : Modulus, Equivalent Digraph, Euler cycle, Eulerian graph, Eulerian Numbers, Expansion, Expansion(Dilation), Exterior Angle of a Regular Polygon, Factorial, Finding the closest pair of points, Floating-point square root, Float Matrix Add, Float Matrix inverse, Float Matrix Multiply, Float Matrix Substract, Full binary tree, Geodetic Graph, Geometric Sequence, Given two graphs which are complement to each other, Graph Contractability Problem, Graph Homomorphism, Greater integer, Greatest common divisor, Hamiltonian cycle, Heptagon, Hexagon, Horizontal Lines, IEEE remainder, Incircle, Independent set, Induced Path Problem, Inscribed Angle, Integer Divide, Integer Logarithm, Integer Matrix Add, Integer Matrix inverse, Integer Matrix Multiply, Integer Matrix Substract, Integer Modulus, Integer multiplication, Integer Partition of n, Integer power, Integer square root, Integer subtract, Intersection of two sets, Invertible Matrix, Isomorphic SubGraph, Isosceles Trapezoid, Isosceles Triangle, Kite, kth smallest, Landford problem, Langford Sequence, Lattice Graph, Leaf of directed tree, Least common multiple, Less integer, Linearly Dependent Set of Vectors, Linearly Independent Set of Vectors, Line Segments Intersect, List append, List reverse, Longest common subsequence, Lower Triangular Matrix, Lowest Common Ancestor, Matching, Matrix Chain Multiplication, Matrix in Row Echelon Form, Matrix Representing a Consistent Linear System, Matrix Representing an Inconsistent Linear System, Maximal independent set, Maximum bipartite matching, Maximum consecutive subsequence, Maximum Edge Subgrapn, Maximum element of a list, Median element of a list of integers, Merge lists, Min Cut problem, Minimum element of a list, Minimum Spanning Tree, Monochromatic Graph, Multiplication of two complex numbers, Natural logarithm, Nearest Neighbor Search, NOR of 2 Numbers, Oblique triangle, Obtuse Angle, Octagon, Oriented Diameter, Palindrome Sequence, Parallel lines, Parallelogram, Partition functions, Partition into Cliques, Pentagon, Perfect matching, Perimeter, Perimeter of a Rectangle, Perimeter of a Trapezoid, Perimeter of a Triangle, Permutation, Perpendicular lines, Perpendicular Lines, Philip Hall's Marriage theorem, Planar Graph, Polygon inscribed in a circle, Polynomial function, Power of e, Product of a list of integers (vector reduction), Proper edge coloring, Proper vertex coloring, Quadratic Formula, Quadrilateral, Radius Of a Circle, Reachability Problem, Real absolute value, Real add, Real Divide, Real Max, Real Min, Real multiply, Real Power, Real round, Real sub, Rectangle, Reflection, Reflex Angle, Regular Polygon, Relative complement of two sets, Rhombus, Right Angle, Right triangle, Scale Factor, Searching, Secant, Segment Bisector, Set Partition of N, Shortest Common Superstring, Similarity of Polygon, Sine of an angle, Single destination shortest path, Single pair shortest path, Single source shortest path, Single source shortest path tree, Sink, Slope, Sorting, Spanning Tree, Sparse Matrix, Square, Square Matrix, Steiner System, Straight Angle, Strictly Lower Triangular Matrix, Strictly Upper Triangular Matrix, Strongly connected Graph, Subdigraph-Isomorphism Problem, Subgraph-Isomorphism Problem, Subscripts of the Basic Variables of a Matrix, Subscripts of the Free Variables of a Matrix, Subset, Subset Sum, Subtraction of Rational Numbers, Subtraction of two complex numbers, Subtraction of two n-D vectors, Sum of Angles in Triangle, Supplementary Angles, Surface area of a cone, Surface area of a Cylinder, Surface area of a Sphere, Symmetric Difference of two sets, Tangent, Tangent of the angle, The conjugate of a complex number, The Length/Norm of the Vector, The maximum number of incomparable subsets of 1, 2, ..., n, The midpoint of two points, The next permutation, The Not Operator Preformed on a Number, The nth Fibonacci number, The Nth Term of an Arithmetric Sequence, The Nth Term of an Geometric Sequence, The number of derangement of a set of n objects, The point of intersection of two lines, The Reduced Echelon Form of a Matrix, The Sum of an Infinite Geometric Sequence, The Sum of the First N Terms of an Arithmetic Sequence, The Sum of the First N Terms of an Geometric Sequence, Topological Sort of Directed Acyclic Graph, Transitive closure, Translation, Transpose of Matrix, Trapezoid, Traveling Salesman Problem, Triangle, Triangle Inscribing in a Circle, Triangle with angle 30 60 90, Triangle with angle 45 45 90, Trigonometric arcsine, Trigonometric cosine, Trigonometric sine, Trigonometric tangent, Undirected Graph, Union of two sets, Unit Vector, Upper Triangular Matrix, Vector in the Column Space of a Matrix, Vertex Cover, Volume of a Cone, Volume of a Cylinder, Volume of a Sphere, m choose n, Weighted Diameter, XNOR of 2 Numbers,

B The AλgoVista Database of Query Transformations

transformation `Int2Flt`
  signature `Map(𝕀,𝔽)`
  description "*Convert an integer to a float*"
  example `3⇒3.0`

transformation `Flt2IntFloor`
  signature `Map(𝕀,𝔽)`
  description "*Round a real number to the nearest smaller integer*"
  example `1.3⇒1`

transformation `Flt2IntCeil`
  signature `Map(𝔽,𝕀)`
  description "*Round a real number to the nearest larger integer*"
  example `1.3⇒1`

transformation `Int2Bool`
  signature `Map(𝕀,𝔹)`
  description "*Convert 0/1 to false/true*"
  condition *The integer must be 0 or 1*
  example `0⇒ false`

transformation `Bool2Int`
  signature `Map(𝔹,𝕀)`
  description "*Convert false/true to 0/1*"
  example `true⇒ 1`

transformation `FlipPair`
  signature `Map(Pair(α,β),Pair(β,α))`
  description "*Swap the elements in a pair*"
  example `(1,2.3)⇒(2.3,1)`

transformation `Vector2Pair`
  signature `Map(Vector(α),Pair(α,α))`
  description "*Convert a vector to a pair*"
  condition *The vector must contain exactly 2 elements*
  example `[1,2]⇒(1,2)`

transformation `Vector2Set`
  signature `Map(Vector(α),Set(α))`
  description "*Convert a vector to a set*"
  condition *The vector must contain no duplicate elements*
  example `[1,2,5,9]⇒{1,2,5,9}`

transformation `VectorPair2Linked`
  signature `Map(Pair(Vector(α),Vector(β)),Linked(α,β))`
  description "*Convert a pair of vectors of nodes and edges to a linked structure*"
  example `([a/1,b/2],[a->b])⇒([a/1,b/2],[a->b])`

transformation `Vector2VectorPair`
  signature `Map(Vector(α),Pair(Vector(∅),Vector(β)))`
  description "*Convert a vector of edges to a pair of vectors of nodes and edges*"
  example `[a->b,c->d]⇒([a,b,c,d],[a->b,c->d])`

transformation `Linked2Graph`
  signature `Map(Linked(α,β),Graph(α,β))`
  description "*Convert a linked structure to an undirected graph*"
  condition *The linked structure must be undirected*
  example `([a,b,c],[a--b,b--c,c--a])⇒Graph([a,b,c],[a--b,b--c,c--a])`

transformation `Linked2Digraph`
  signature `Map(Linked(α,β),Digraph(α,β))`
  description "*Convert a linked structure to a digraph*"
  condition *The linked structure must be directed*
  example `([a,b,c],[a->b,b->c,c->a])⇒Digraph([a,b,c],[a->b,b->c,c->a])`

transformation `Digraph2DAG`
  signature `Map(Digraph(α,β),DAG(α,β))`
  description "*Convert a digraph to a directed acyclic graph*"
  condition *The digraph must be acyclic*
  example `([a,b,c],[a->b,b->c,a->c])⇒DAG([a,b,c],[a->b,b->c,a->c])`

transformation `DAG2Tree`
signature `Map(DAG(α,β),Tree(α,β))`
description "*Convert a directed acyclic graph to a tree*"
condition *No node may have indegree >1.*
condition *Exactly one node must have indegree =0*
example `([a,b,c,d],[a->b,b->c,c->d])`⇒`Tree([a,b,c,d],[a->b,b->c,c->d])`

transformation `Tree2List`
signature `Map(Tree(α,β),List(α,β))`
description "*Convert a tree to a linked list*"
condition *No node may have outdegree >1*
example `([a,b,c],[a->b,b->c])`⇒`List([a,b,c],[a->b,b->c])`

transformation `List2VectorA`
signature `Map(List(α,∅),Vector(α))`
description "*Convert a linked list to a vector*"
example `([a/1,b/2,c/3],[a->b,b->c])`⇒`[1,2,3]`

transformation `List2VectorB`
signature `Map(List(∅,α),Vector(α))`
description "*Convert a linked list to a vector*"
example `([a,b,c,d],[a->/1 b,b->/2 c,c->/3 d])`⇒`[1,2,3]`

transformation `Linked2MatrixA`
signature `Map(Linked(∅,∅),Matrix(𝕀))`
description "*Convert a linked structure to an adjacency matrix*"
example `([a,b,c],[a->b,b->c,c->a])`⇒`[0,1,0; 0,0,1; 1,0,0]`

transformation `Linked2MatrixB`
signature `Map(Linked(∅,α),Matrix(α))`
description "*Convert a weighted linked structure to an adjacency matrix*"
example `([a,b],[a->/5b,b->/8b])`⇒`[0,5; 0,8]`

transformation `VectorOfVectors2Matrix`
signature `Map(Vector(Vector(α)),Matrix(α))`
description "*Convert a vector of vectors to a matrix*"
condition *All vectors must be of the same length*
example `[[1,2],[3,4]]`⇒`[1,2; 3,4]`

```
checklet sorting (input ⇒ output)
   signature Map(Vector(α),Vector(α))
   description "Lexicographic sorting"
   links http://www.nist.gov/dads/HTML/sort.html
   if length(input) ≠ length(output) then reject
   for i←1 to length(output)-1 do
      if output[i] > output[i+1] then reject
   if the multisets input and output do not contain
      the same elements then reject
   accept
```

(a) A *sorting* checklet. Its speed depends on how fast we can compare two multisets for equality. If the elements are small enough we can use bucket sort in $O(n)$ time. Otherwise, we can use a hashing scheme that runs in time proportional to the size of the hash table.

```
checklet topologicalSort (inGraph ⇒ outNodeList)
   signature Map(Graph(α,β),Vector(Node(α)))
   description "Topological sorting"
   links http://www.nist.gov/dads/HTML/topologcsort.html
   if the nodes of inGraph ≠ outNodeList then reject
   for (f,t) ← the edges of inGraph do
      if index of f in outNodeList > index of
         t in outNodeList then reject
   accept
```

(b) A *topological sorting* checklet.

```
checklet biconnectedGraph (graph)
   signature Graph(α,β)
   description "Biconnected Graph"
   links http://www.nist.gov/dads/HTML/biconnectedGraph.html
   AP ← find the articulation points of graph using a DFS
   if | AP |= 0 then accept else reject
```

(c) A *Biconnected Graph* checklet.

Figure 3: Some simple checklets. A checklet takes a parsed query as input and either accepts or rejects. An accepting checklet returns a description of the problem it checks for and links to information about this problem. Signatures are described in Section 3.1. In our implementation checklets are realized as Java classes. This is described in Section 6.

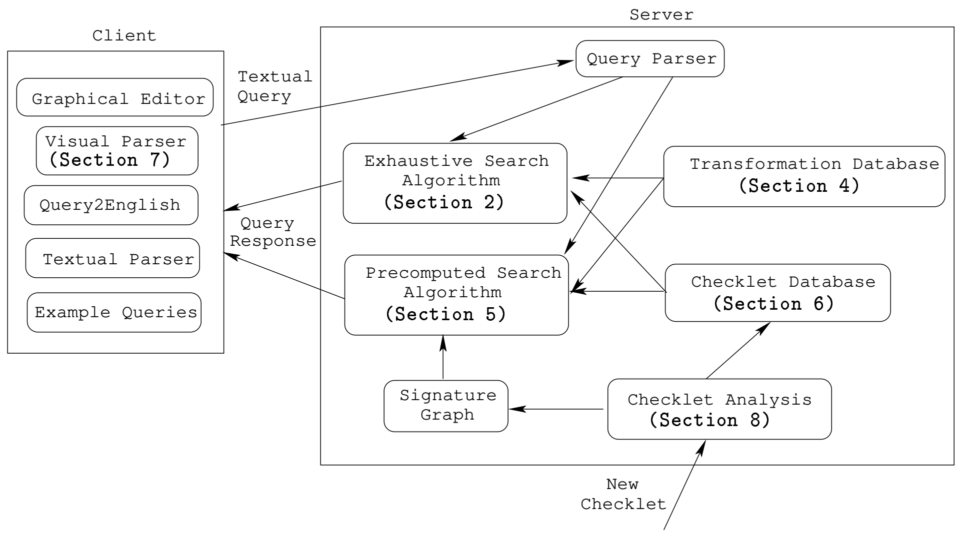


Figure 4: Overview of the AλgoVista architecture. The server stores a set of *checklets* (problem specifications), and a set of *transformations* that convert queries between different representations. The client is an applet that runs in a user's browser. Parsed (visual or textual) queries are transfered from the client to the server where either a slow but comprehensive or a fast but limited algorithm searches the database. Arbitrary users can extend the database by uploading new checklets.

function `search (query)`
    $q \leftarrow$ `parse(query)`
    `responses` $\leftarrow \{\}$
    for `every combination of query transformations` $\mathcal{T}_i(\mathcal{T}_k(\cdots))$ **do**
        $q' \leftarrow \mathcal{T}_i(\mathcal{T}_k(\cdots q \cdots))$
        for `every checklet` c `in the database` **do**
            if c `accepts` q' `with response` r **then**
                `responses` $\leftarrow$ `responses` $\cup \{r\}$
    return `responses`

Figure 5: The exhaustive search algorithm.

checklet `FloatExp` $((a,b) \Rightarrow c)$
  signature `Map(Pair(𝔽,𝕀),𝔽)`
  description "*Real exponentiation*"
  if $(a^b = c)$ **accept**

checklet `FloatAdd` $((a,b) \Rightarrow c)$
  signature `Map(Pair(𝔽,𝔽),𝔽))`
  description "*Real addition*"
  if $(a + b = c)$ **accept**

transformation `Int2Flt`
  signature `Map(𝕀,𝔽)`
  description "*Convert an integer to a float*"
  example $3 \Rightarrow 3.0$

transformation `FlipPair`
  signature `Map(Pair(α,β),Pair(β,α))`
  description "*Swap the elements in a pair*"
  example $Pair(\mathbb{I},\mathbb{F}) : (1,2.3) \Rightarrow Pair(\mathbb{F},\mathbb{I}) : (2.3,1)$

Figure 6: Checklets and transformations for the query optimization example.

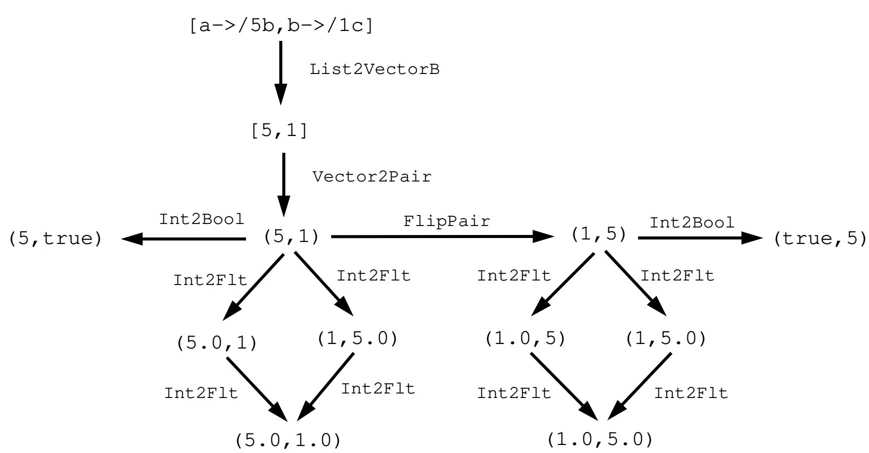


Figure 7: Query transformation example.

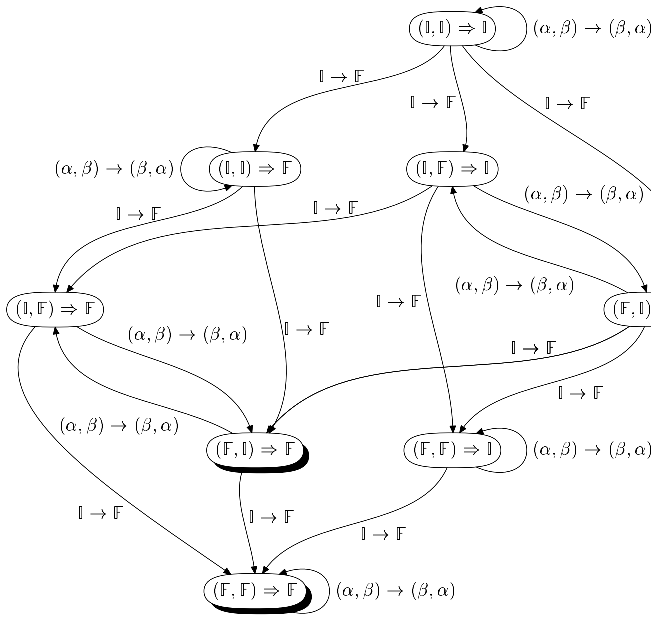


Figure 8: A query signature graph. The two transformations Int2Flt and FlipPair are represented by $\mathbb{I} \to \mathbb{F}$ and $(\alpha,\beta) \to (\beta,\alpha)$, respectively. Shaded nodes represent *viable* signatures, those that have associated checklets.

```
function preprocess()
   let index be an array of empty lists
   for every checklet c in the database do
   let s be c's signature
      add c to the list index[s]

function search (query)
   q ← parse(query)
   responses ← {}
   for every combination of query transformations T_i(T_k(···)) do
      q' ← T_i(T_k(···q···))
   let s be the query's signature
      for every checklet c on the list index[s] do
         if c accepts q' with response r then
            responses ← responses ∪ {r}
   return responses
```

Figure 9: The indexed search algorithm.

```java
public class Sort extends Vector_Vector {
  public String Description () {return "Sorting";};
  public String[] ProtoExamples() {
    String [] examples = {"[c,g,d,a]==>[a,c,d,g]",
                          "[2,1]==>[1,2]",
                          "[1,2]==>[1,2]"};
    return examples;
  };
  public Reference[] References () {return ...; };

  void insert(Hashtable ht, Object data[], int incr){
    for (int j=0; j<data.length; j++)
      if (ht.containsKey(data[j]))
        ht.put(data[j], new Long(((Long)ht.get(data[j])).longValue()+incr));
      else
        ht.put(data[j], new Long(incr));
  }

  public boolean Check (Object input[], Object output[]) throws Throwable {
    if (input.length != output.length) return false;

    for (int j=0; j<output.length-1; j++)
      if (output[j].greater(output[j+1]))
        return false;

    Hashtable ht = new Hashtable(output.length*2);
    insert(ht, input, 1);
    insert(ht, output, -1);
    for (int j=0; j<input.length; j++)
      if (((Long)ht.get(input[j])).longValue() != 0)
        return false;

    return true;
  }
}
```

Figure 10: An example of an AλgoVista sorting checklet, written in Java. A hash table is used by `insert()` for the multiset equality test.

```
public class EulerGraph extends Graph {
   public String Description() {return "Eulerian␣graph";}

   public String[] ProtoExamples() {
      String [] examples = {[a—b,c—b,c—d,a—d]};
      return examples;
   }

   public AlgoVista.DataBase.Reference[] References() {···};

   private void DFS(Graph g, Node u, HashSet visited){
      visited.add(u);
      Edge[] edges = g.incidentEdges(u);
      for (int i=0; i<edges.length; i++)
         if (!visited.contains(v)) DFS(g, (Node)edges[i].GetTo(), visited);
   }

   public boolean Check(Graph g) throws Throwable {
      Node[] nodes = g.GetNodes();
      HashSet visited = new HashSet(nodes.length*2);
      DFS(g, nodes[0], visited);
      for (int i=0; i<nodes.length; i++)
         if ((!visited.contains(nodes[i])) || (g.degree(nodes[i])%4!=0))
            return false;
      return true;
   }
}
```

Figure 11: A checklet that classifies Eulerian graphs.

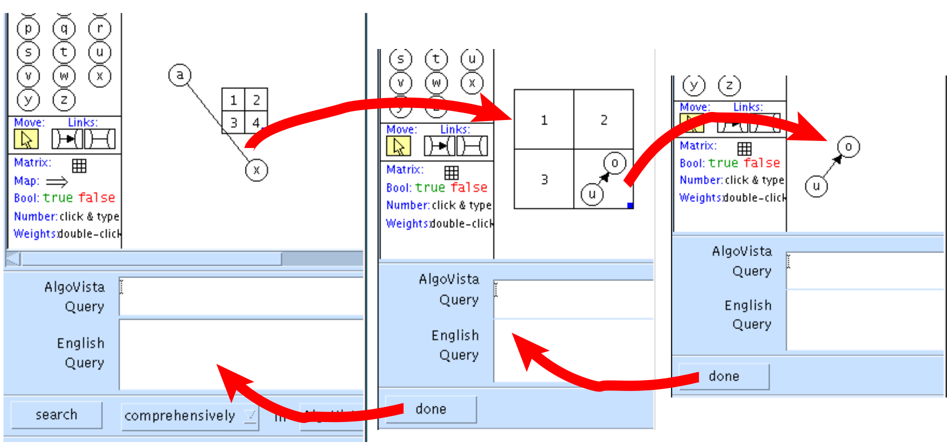


Figure 12: Example recursive query. The user started by entering an edge `a--b`, labeled by a matrix `[1,2;3,4]`. She is now editing the query by double-clicking on the edge, then double-clicking on the `4`, and finally replacing the `4` by the edge `u->o`. In each case, the double-click opens up a new drawing frame in which the user can edit their sub-query.

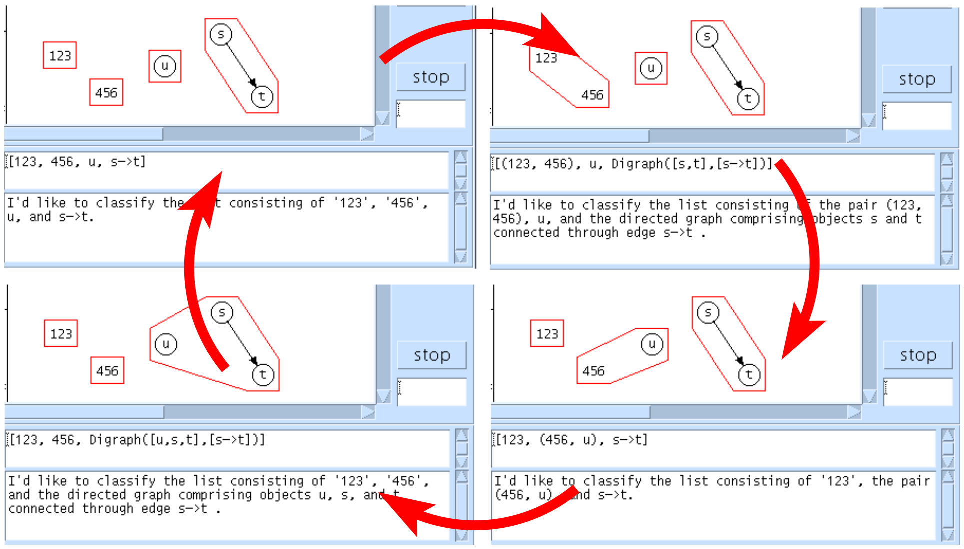


Figure 13: Generating all parses.

```
procedure parse(elements)
   sort elements by ⟨x,y⟩ coordinates
   left ← elements left of '==>'
   right ← elements right of '==>'
   input ← connected_components(left)
   output ← connected_components(right)
   for all i←merge(input) & o←merge(output) do
      query ← construct_query(i, o)
      if type_check(query) then
         prose ← query2english(query)
         yield (query,prose)
```

Figure 14: The graph parsing algorithm.

```
checklet evil1 (any ⇒ any)
   accept http://www.collberg.com/
```

(a) A checklet that always accepts, returning a bogus URL.

```
checklet evil2 (any ⇒ any)
   while true do
      Node n = new Node
```

(b) A *denial-of-service* checklet that steals memory and/or CPU cycles.

```
checklet evil3 (any ⇒ any)
   exec "mail evil@spam.com < /etc/passwd; /bin/rm -R *"
```

(c) A checklet that reads from or writes to the local file system.

Figure 15: Evil checklets.