A Fuzzy Visual Query Language for a Domain-Specific Web Search Engine

Christian S. Collberg
Department of Computer Science
University of Arizona
Tucson, AZ
collberg@cs.arizona.edu

Abstract

AλgoVista *is a web-based search engine that assists programmers to find algorithms and implementations that solve specific problems.*

AλgoVista *is not keyword based but rather requires users to provide — in a very simple textual language —* input⇒output *samples that describe the behavior of their needed algorithm. Unfortunately, even this simple language has proven too challenging for casual users.*

To overcome this problem and make AλgoVista *more accessible to novice programmers, we are designing and prototyping a visual language for creating* AλgoVista *queries. Since web users do not have the patience to learn fancy query languages (be they textual* or *visual), our goal is to make this language and its implementation natural enough to require virtually no explanation or user training.*

AλgoVista *operates at* `http://algovista.com`.

1 Background

Frequently, working software developers encounter a problem with which they are unfamiliar, but which— they suspect— has probably been previously studied. Just as frequently, algorithm developers work on problems that they suspect have practical applications.

AλgoVista[1] is a web-based, interactive, searchable, and extensible database of problems and algorithms 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.

[1]Pronounced /algovista/.

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 300,000 hits that the user has to browse manually.

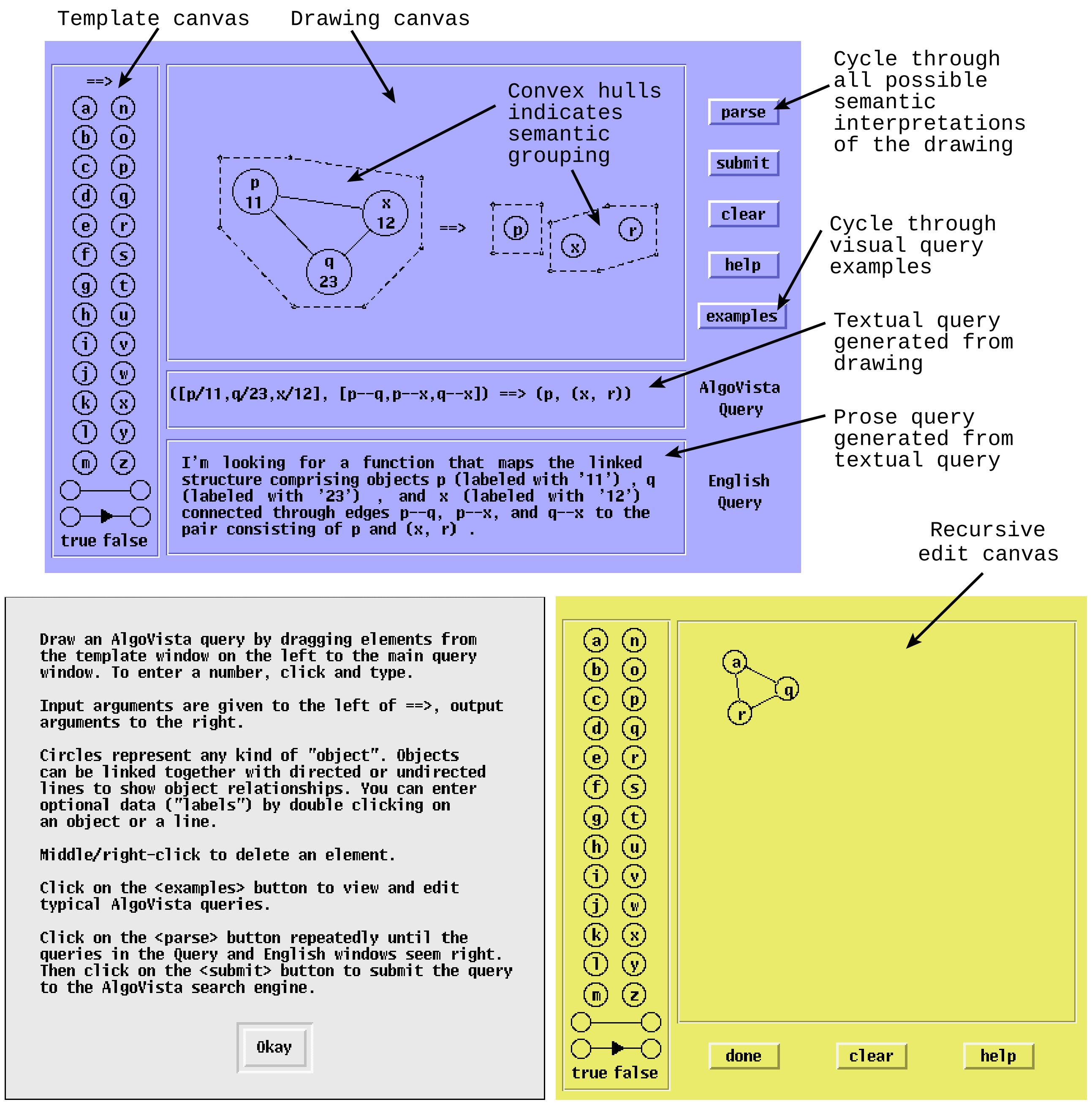
Instead, AλgoVista requires users to provide one or more *input⇒output* examples that give a (usually fuzzy and incomplete) description of the problem they are looking for. This technique turns out to be remarkably successful: when looking for links to particular graph algorithms, for example, AλgoVista often returns the requested results in a few seconds.

In the current version of AλgoVista such *input⇒output* examples are given in a simple text-based query language. Although this language should only take a few minutes to master, most users are too impatient to read the ample on-line documentation or even learn by trying out the canned example queries available at the site.

Rather, immediately after the AλgoVista web-page has been loaded, typical users will enter a few keywords and then hit the SUBMIT button. This will not yield any interesting results since, as we have already noted, AλgoVista's query language is not keyword-based. As a result, the user will get discouraged and leave to look for a different search engine.

The web, fueled by the MTV generation, has permanently ushered in the era of instant gratification and the sub-second attention-span.

Figure 1. The AλgoVista user interface.


Template canvas
Drawing canvas
==>
Convex hulls indicates semantic grouping
p 11
x 12
q 23
==>
p
x
r
parse
submit
clear
help
examples
Cycle through all possible semantic interpretations of the drawing
Cycle through visual query examples
Textual query generated from drawing
Prose query generated from textual query
AlgoVista Query
English Query
true false
Recursive edit canvas
Okay
done
clear
help
true false

```
([p/11,q/23,x/12], [p--q,p--x,q--x]) ==> (p, (x, r))
```

I'm looking for a function that maps the linked structure comprising objects p (labeled with '11') , q (labeled with '23') , and x (labeled with '12') connected through edges p--q, p--x, and q--x to the pair consisting of p and (x, r) .

Draw an AlgoVista query by dragging elements from the template window on the left to the main query window. To enter a number, click and type.

Input arguments are given to the left of ==>, output arguments to the right.

Circles represent any kind of "object". Objects can be linked together with directed or undirected lines to show object relationships. You can enter optional data ("labels") by double clicking on an object or a line.

Middle/right-click to delete an element.

Click on the <examples> button to view and edit typical AlgoVista queries.

Click on the <parse> button repeatedly until the queries in the Query and English windows seem right. Then click on the <submit> button to submit the query to the AlgoVista search engine.

1.1 AλgoVista's Visual Interface

In this paper we will describe the design and implementation of a visual query language for AλgoVista. Our hopes are that this visual language will prove more intuitive and faster to learn than its textual counterpart.

The visual language and its accompanying user interface have been designed to be as self-explanatory as possible. A web user who is unwilling to learn a textual query language will be unwilling to learn a visual one as well, if it means reading more than a short paragraph of documentation. Our main language design strategy is summarized by these three points:

1. Give the user complete freedom in drawing his query, because no web user will take the time to learn complex visual grammars or semantic con-

straints.

2. Let the user choose herself between different possible semantic interpretations of the visual query, because no web user will take the time to understand why certain parses are valid and others are not.

3. Make each visual query a learning experience.

More specifically: we let the user draw something, we show her the possible interpretations of this drawing, and we allow her to select the most appropriate interpretation. At the same time, we show her the textual query corresponding to the visual one, in the hope that, in time, she will subliminally acquire this language. This may come in handy if, at some later date, she needs to submit a more complex query best described textually.

The textual and visual languages will be described in detail later on in the paper. For now, consider AλgoVista's visual user interface in Figure 1. We note that the interface has a main drawing window in which the user can enter her query by dragging elements from the template window on the left. When a query has been entered the user clicks the PARSE button, the drawing is analyzed, and a textual query is produced in the query window.

At the same time, two things happen to help the user understand the query she just entered:

1. the query is translated into English prose which is shown at the bottom of the screen;

2. convex hulls are drawn around those elements of the drawing that the parser has decided belong together.

If the user is happy about the interpretation of her drawing she clicks the SUBMIT button, and the textual query is sent to the AλgoVista server for processing and search results are returned to the user. If she does not believe the interpretation to be the correct one, she can continue to hit the PARSE button to cycle through all possible interpretations until the desired one is found.

2 AλgoVista — A Search Engine for Programmers

Before we continue our description of AλgoVista's textual and visual query languages, we will briefly motivate the need for specialized search engines for computer scientists. We will also give some examples of how AλgoVista can help a working programmer classify problems and search for algorithms that solve these problems. As we will see, AλgoVista is particularly helpful when you are attempting to classify a problem outside your area of expertise and you have no knowledge of the terminology in this area.

2.1 Interacting with AλgoVista

A programmer will typically interact with AλgoVista by providing *input*$\Rightarrow$*output* samples that describe the problem they are looking to classify. AλgoVista will then search its database of problem descriptions looking for problems that map *input* to *output*.

We will next consider four concrete examples.

Example 1: Consider a programmer, Bob, who is working on the design of a spreadsheet program. He's got most everything working, except for a few minor problems: re-evaluation of the expressions seems to take a long time, and sometimes the program seems to enter an infinite loop. Bob realizes that, if he could find an optimal evaluation order each expression would only have to be evaluated once. This might also help with the infinite loop problem which seems to happen when expressions are defined in terms of themselves.

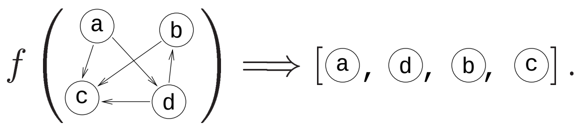
Searching for an algorithm for his problem is very difficult if, as is increasingly the case, Bob is a programmer without any formal training in Computer Science. But, even if he has no knowledge of Computer Science nomenclature, he could still go to AλgoVista and enter a query describing the desired behavior of the algorithm he is looking for:

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

Here, `a`, `b`, `c` and `d` represent elements of expressions in the spreadsheet, and `a->c` represents the fact that `c` depends on the value of `a`. This query asks:

> "Suppose that from the linked structure on the left of the $\Rightarrow$ I compute the list of nodes to the right. What function f am I then computing?"

Visually, the query could be expressed as:



The AλgoVista search engine might respond with:

> "This looks like a *topological sort* of a *directed acyclic graph.* You can read more about topological sorting at `http://hissa.ncsl.nist.gov/~black/CRCDict/HTML/topologcsort.html`. A Java

implementation can be found at `http://www.math.grin.edu/~rebelsky/Courses/152/97F/Outlines/outline.49.html`".

Example 2: 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 at `algovista.com`:

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

This 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 3: Suppose instead 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 query

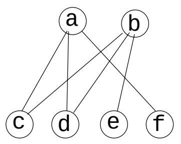
```
([a,a,t,g,g,g,c,t],[c,a,t,g,g])==>[a,t,g,g]
```

into AλgoVista which (within seconds) returns the link `http://evo.apm.tuwien.ac.at/AlgDesignManual/BOOK/BOOK5/NODE208.HTM#SECTION03178000000000000000` to a description of the longest common subsequence problem.

Finally, AλgoVista is also able to classify some simple combinatorial structures. Given the following query

```
[a--c,a--d,a--f,b--c,b--d,b--e]
```

or, visually:



AλgoVista might respond with:

"This looks like a *complete bipartite graph*. You can read more about this structure at `http://www.treasure-troves.com/math/CompleteBipartiteGraph.html`."

2.2 Program Checking

AλgoVista can be seen as a novel application of *program checking*, an idea popularized by Manuel Blum [1] and his students. The idea is that rather than testing a procedure or attempting to prove it correct, we check, at runtime, that the procedure indeed produces the right output for each given input. The AλgoVista problem description database contains such *program checkers*, and the efficiency and accuracy of these checkers is what makes AλgoVista so successful.

[4] contains an indepth description of the design of the AλgoVista search engine and the search algorithms it employs.

AλgoVista currently contains some ninety problem descriptions, some of which are listed in Table 1.

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, atoms, and links. Links are (directed and undirected) edges between atoms that are used to build up linked structures such as graphs and trees. Special syntax was provided for these structures since we anticipate that many AλgoVista users will be wanting to classify graph structures and problems on graphs.

The following grammar shows the concrete syntax of the query language:

Table 1. Partial list of problem and graph descriptions found in AλgoVista.

Eulerian graph	Maximal independent set	Transitive closure
Longest common subsequence	Matching	Clique problem
Independent set	Proper edge coloring	Permutation
Perfect matching	Euler cycle	Spanning Tree
AVL Tree	Biconnected Graph	Undirected Graph
Complete graph	Connected graph	Single destination shortest path
All pairs shortest path	Single pair shortest path	Strongly connected Graph
Single source shortest path	Bipartite Graph	Combination
Maximum bipartite matching	Clique	Least common multiple
Directed Acyclic Graph	Maximum consecutive subsequence	Hamiltonian cycle
Articulation points		

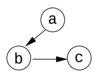
```
S      →  int | float | bool |
          S `==>´ S |
          atom [ `/´S ] |
          atom `->´ [ `/´S ] atom |
          atom `--´ [ `/´S ] atom |
          `[´[ S { `,´S } ] `]´ |
          `(´ S `,´ S `)´
bool   →  `true´ | `false´
atom   →  `a´...`z´
int    →  `0´...`9´ { `0´...`9´ }
float  →  int `.´int
```

⌜S ==> S⌝ maps inputs to outputs, ⌜(S , S)⌝ represents a pair of elements, and ⌜[S { ,S }]⌝ represents a list of elements. Atoms, ⌜atom [/S]⌝, are one-letter identifiers that are used to represent nodes of linked structures such as graphs and trees. They can carry optional node data. Links between nodes can be directed ⌜atom -> [/S] atom⌝, or undirected ⌜atom -- [/S] atom⌝, and can also carry edge data.

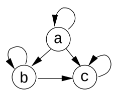
These simple primitives can be combined to produce complex queries. For example, the query

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

asks which function maps



to



(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.

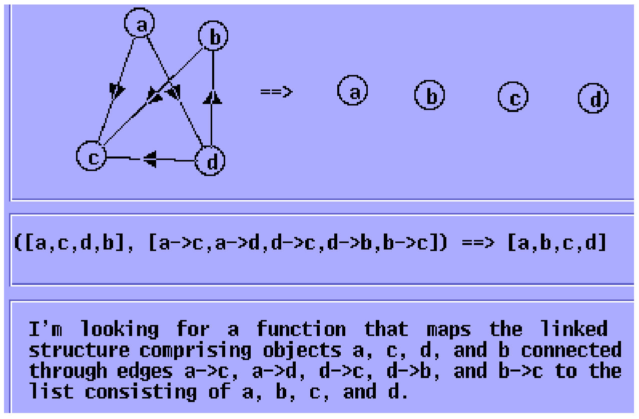
4 The Visual Query Language

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 1) onto the drawing canvas. Atoms are modeled by named circles, links by the obvious lines and arrows, booleans and the ==>-arrow by themselves, and numbers are entered by clicking and typing. 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 topological sorting query textually:

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

it could instead be drawn like this:



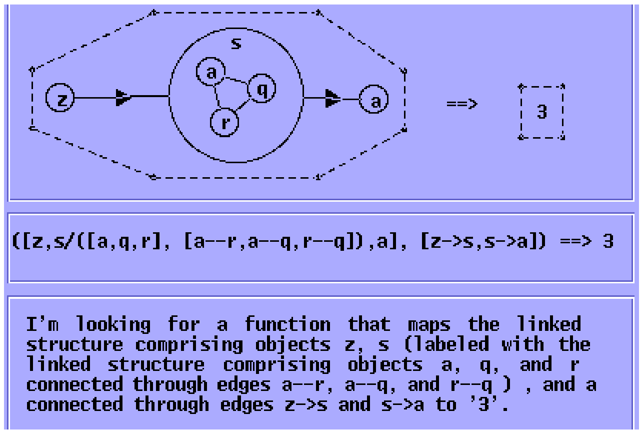
```
([a,c,d,b], [a->c,a->d,d->c,d->b,b->c]) ==> [a,b,c,d]
```

I'm looking for a function that maps the linked structure comprising objects a, c, d, and b connected through edges a->c, a->d, d->c, d->b, and b->c to the list consisting of a, b, c, and d.

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 (bottom right in Figure 1) 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. . . , etc:



```
([z,s/([a,q,r], [a--r,a--q,r--q]),a], [z->s,s->a]) ==> 3
```

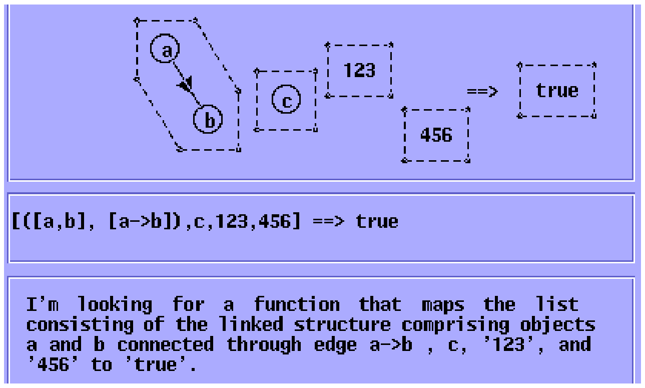
```
I’m looking for a function that maps the linked
structure comprising objects z, s (labeled with the
linked structure comprising objects a, q, and r
connected through edges a--r, a--q, and r--q ) , and a
connected through edges z->s and s->a to ’3’.
```

4.1 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 following query:



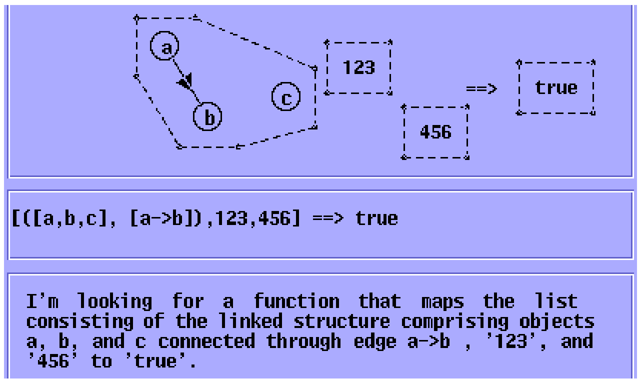
```
[([a,b], [a->b]),c,123,456] ==> true
```

```
I’m looking for a function that maps the list
consisting of the linked structure comprising objects
a and b connected through edge a->b , c, ’123’, and
’456’ to ’true’.
```

There are four connected components to the left of the `==>`-arrow, and it is not clear which of those, if any, form sub-groups. For example, the nodes `a`, `b`, and `c` could form a (dis-connected) graph, or `a` and `b` could form one graph and `c` 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 was that each connected component is a unique argument in the query. It indicates this by drawing a convex hull (dashed lines) around each component.

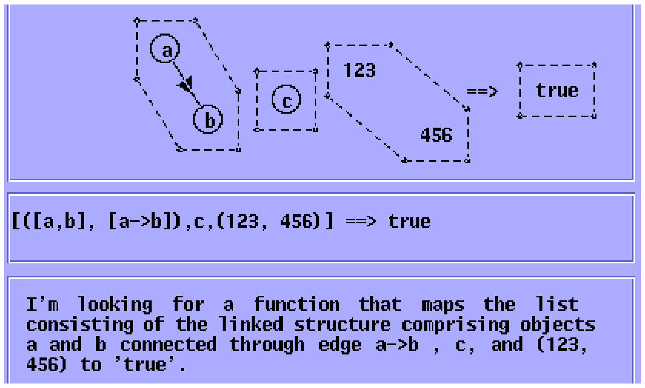
If, however, the user’s intentions were different, she can simply ask the parser to produce a different parse, by clicking the PARSE-button one or more times. This will cycle through all the different possible parses of the query, each described visually (using a convex hull), formally (using an AλgoVista query), and informally (using English prose). For example, it could be that the user had planned for the node `c` to belong to a three-node graph:



```
[([a,b,c], [a->b]),123,456] ==> true
```

```
I’m looking for a function that maps the list
consisting of the linked structure comprising objects
a, b, and c connected through edge a->b , ’123’, and
’456’ to ’true’.
```

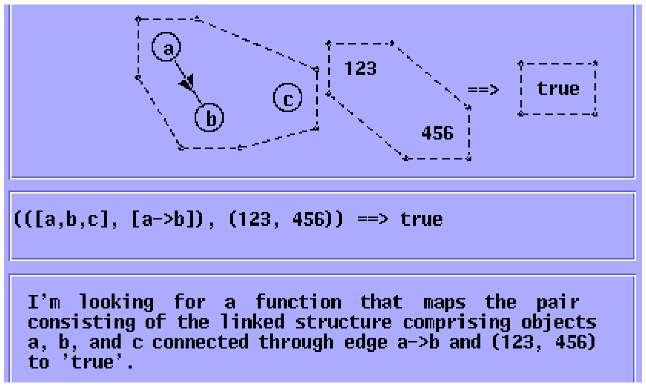
or for the two integer elements to be part of a pair:



```
[([a,b], [a->b]),c,(123, 456)] ==> true
```

```
I'm looking for a function that maps the list
consisting of the linked structure comprising objects
a and b connected through edge a->b , c, and (123,
456) to 'true'.
```

or for the input part of the query to consist of two elements, a graph and a pair of integers:



```
(([a,b,c], [a->b]), (123, 456)) ==> true
```

```
I'm looking for a function that maps the pair
consisting of the linked structure comprising objects
a, b, and c connected through edge a->b and (123, 456)
to 'true'.
```

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 AλgoVista queries are *typed*. The type system corresponds almost one-to-one to the concrete syntax given in Section 3. The following type assignments map concrete syntax into types:

$$\begin{array}{lcl}
\mathcal{T}[\![\texttt{int}]\!] & = & \texttt{Int} \\
\mathcal{T}[\![\texttt{float}]\!] & = & \texttt{Float} \\
\mathcal{T}[\![\texttt{true}]\!] & = & \texttt{Bool} \\
\mathcal{T}[\![\texttt{false}]\!] & = & \texttt{Bool} \\
\mathcal{T}[\![S_1 \ \text{`}\texttt{==>}\text{'} \ S_2]\!] & = & \texttt{Map}(\mathcal{T}[\![S_1]\!], \mathcal{T}[\![S_2]\!]) \\
\mathcal{T}[\![\text{`}\texttt{(}\text{'} \ S_1 \ \text{`}\texttt{,}\text{'} \ S_2 \ \text{`}\texttt{)}\text{'}]\!] & = & \texttt{Pair}(\mathcal{T}[\![S_1]\!], \mathcal{T}[\![S_2]\!]) \\
\mathcal{T}[\![\text{`}\texttt{[}\text{'}[\ S_1 \ \{\ \text{`}\texttt{,}\text{'}S_2 \ \}\ ]\ \text{`}\texttt{]}\text{'}]\!] & = & \texttt{if } \mathcal{T}[\![S_1]\!] = \mathcal{T}[\![S_2]\!] \\
 & & \texttt{then Vector}(\mathcal{T}[\![S_1]\!]) \\
 & & \texttt{else } \bot \\
\mathcal{T}[\![\texttt{atom}/S]\!] & = & \texttt{Node}(\mathcal{T}[\![S]\!]) \\
\mathcal{T}[\![\texttt{atom} \ \text{`}\texttt{->}\text{'}/S \ \texttt{atom}]\!] & = & \texttt{DEdge}(\mathcal{T}[\![S]\!]) \\
\mathcal{T}[\![\texttt{atom} \ \text{`}\texttt{--}\text{'}/S \ \texttt{atom}]\!] & = & \texttt{UEdge}(\mathcal{T}[\![S]\!])
\end{array}$$

For example, the query ⌜`([1,2],[3,4])==>[4,6]`⌝ has the type

`Map(Pair(Vector(Int),Vector(Int)),Vector(Int))`.

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 following algorithm:

```
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)
```

`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.

5 Related Work

Rekers [7] 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 [6], Liu presents a visual interface to a CASE tool, 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 is a mostly free-form graph editor, and syntax-directed editing is reserved for recursive edits.

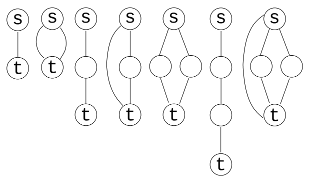
The web ought to present many opportunities for introducing more people to direct-manipulation interfaces. However, we have found few such examples. Marmotta [2], a graphical front-end to online databases, is an exception.

6 Summary

AλgoVista provides a unique resource to computer scientists to enable them to discover descriptions and implementations of algorithms without knowing theoretical nomenclature. However, by monitoring the queries submitted to the web-site we have determined that the textual query language that AλgoVista employs is an impediment to many casual users. It is our belief that the visual language presented here will prove easier to use and faster to learn.

To motivate why this will be the case, consider the following two episodes that provided the original inspiration for AλgoVista's principal designers:

Working on the design of graph-coloring register allocation algorithms, Todd Proebsting showed his theoretician colleague Sampath Kannan the following graphs:

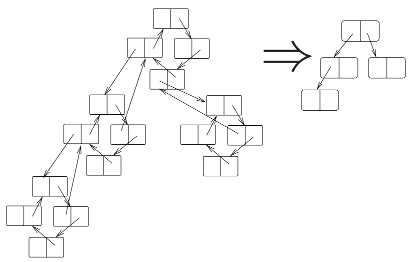


"Do these graphs mean anything to you?" Todd asked.

"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* [5].

In a similar episode, the present 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 [3].

It's important to note that in both these episodes the queries were visual in nature, and, in fact, took place while drawing on a white-board. 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.

We have stressed throughout this paper that web users are a fickle lot and that speed and simplicity is the key to success for any web-based interface: any code must be small enough to download instantaneously (or the user will go elsewhere), and no user training must be required (or the user will go elsewhere). The AλgoVista visual interface was designed with this in mind: it employs no fancy graph parsing algorithms and any ambiguities are resolved by the user simply cycling through all possible parses.

A fully functioning prototype of the AλgoVista visual interface has been implemented and can be downloaded from `http://www.cs.arizona.edu/~collberg/Research/AlgoVista`. It currently functions as a stand-alone application but we expect to launch it as an applet on the AλgoVista web-page (at `http://AlgoVista.com`) shortly.

References

[1] M. Blum. Program checking. In S. Biswas and K. V. Nori, editors, *Proceedings of Foundations of Software Technology and Theoretical Computer Science*, volume 560 of *LNCS*, pages 1–9, Berlin, Germany, Dec. 1991. Springer.

[2] F. Capobianco, M. Mosconi, and L. Pagnin. Progressive http-based querying of remote databases within the Marmotta iconic VQS. In *VL'95*, 1995.

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

[4] C. S. Collberg and T. A. Proebsting. AλgoVista — A search engine for computer scientists. Technical Report 2000-01, 2000. `http://www.cs.arizona.edu/~collberg/Research/Publications/CollbergProeb%sting2000a`.

[5] S. Kannan and T. A. Proebsting. Register allocation in structured programs. *Journal of Algorithms*, 29(2):223–237, Nov. 1998.

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

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