Monte Carlo Tree Search for Generating Interactive Data Analysis Interfaces

Yiru Chen, Eugene Wu
Columbia University
yiru.chen@columbia.edu, ewu@cs.columbia.edu

Abstract

Interactive tools like user interfaces help democratize data access for end-users by hiding underlying programming details and exposing the necessary widget interface to users. Since customized interfaces are costly to build, automated interface generation is desirable. SQL is the dominant way to analyze data and there already exists logs to analyze data. Previous work proposed a syntactic approach to analyze structural changes in SQL query logs and automatically generates a set of widgets to express the changes. However, they do not consider layout usability and the sequential order of queries in the log. We propose to adopt Monte Carlo Tree Search(MCTS) to search for the optimal interface that accounts for hierarchical layout as well as the usability in terms of how easy to express the query log.

Introduction

SQL is the dominant language for accessing and analyzing large datasets today. Its expressive power is useful to identify the appropriate queries during ad-hoc analysis (e.g., in a Jupyter notebook). However, it is cumbersome to repeatedly use for the same set of analysis tasks, and inaccessible to many end-users. In contrast, customized interactive interfaces help users quickly accomplish their data analysis tasks by hiding underlying programming complexity and exposing a simple set of visual widgets designed for the tasks. Unfortunately, turning those analysis queries into a reusable interactive interface requires considerable design and programming expertise.

Prior work (**?**) proposed an automatic interface generation method. Given a set of analysis queries, it identifies changes between the abstract syntax trees (AST) of the queries (**??**), and chooses a set of customized interactive widgets (e.g., slider, tabs, buttons) from a predefined library that can express those changes. For instance, if the queries differ by a numeric value (e.g., `a=1`, `a=2`), then it maps the changes (e.g., $1 \rightarrow 2$) to a widget template (e.g., a slider) that can express the different values. It uses a bottom-up approach that enumerates subtree differences between every pair of ASTs, and maps differences at the same path in the AST to a widget.

Although this work has shown promise, it still suffers from a number of limitations. First, it groups subtrees at the same location in the ASTs and matches them to a widget without consideration of the other widgets nor whether the subtrees should be grouped together. Second, it returns a set of widgets that does not account for the interface layout nor constraints such as the screen size. Notably, it does not leverage the body of HCI research that has studied and quantified interface layout and usability (**?**; **?**). Third, it ignores the effort needed to use the interface to express the *sequence* of input queries.

To this end, we describe our preliminary work on a top-down search-based approach towards interface generation that explicitly addresses the above limitations. We propose a `difftree` representation of the input query ASTs whose structure also encodes the interface layout—this represents a state in the search space—and define transition rules that incrementally transform the `difftree`. The search space is extremely large, thus we use Monte Carlo Tree Search (MCTS) to efficiently identify the lowest cost interface. The rest of this paper describes the problem and our current approach, preliminary results, and ongoing research directions.

Problem Overview

Our goal is to take as input a sequence of SQL queries that are part of an analysis task (e.g., from a query log, or provided by a developer during or after an analysis session), and output an interactive data analysis interface that can express the input queries (and likely similar queries not explicitly in the log). Our assumption is that the structural differences between the queries are representative of the types of changes the user wishes to express interactively.

Our approach is to 1) extract syntactic differences between queries, 2) choose interactive widgets that can express those differences as transformations, and 3) design and layout an interative interface. In this work, we leverage existing automatic visualization techniques that recommend visualizations based on a dataset (**?**; **?**), and thus we focus on the joint problem of determining a good layout, and selecting and configuring the appropriate widgets for the layout.

Queries: Similar to (**?**), we model each query as its abstract syntax tree (AST). **??** without the `ANY` node illustrates the simplified ASTs of three queries. Each node (e.g., `Select`,

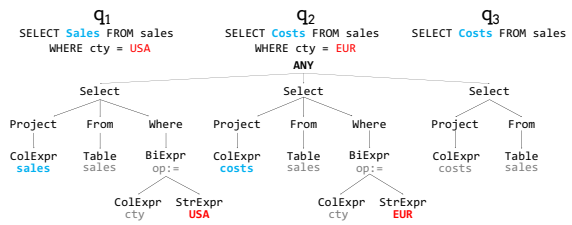

Figure 1: Example ASTs for 3 SQL queries.

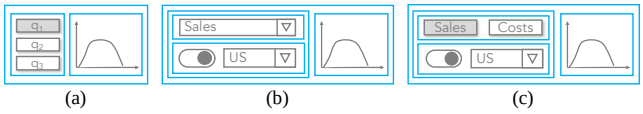

Figure 2: Examples of three interfaces that express the queries in **??**. Blue boxes depict bounding boxes in the layout hierarchy.

`BiExpr`) corresponds to a rule in the query grammar. q_1 and q_2 differ at two nodes: `ColExpr` changed from `sales` to `costs`, while `StrExpr` changed from `USA` to `EUR`. q_3 differs from q_2 by dropping the `WHERE` clause completely.

Interfaces: An interface is a set of visualizations, a set of interactive widgets, and a hierarchical layout of the visualizations and widgets. For ease of discussion, we will describe the case where there is a single visualization. A visualization renders the output of the current query q, and each widget changes q based on the user's interactions (e.g., changing a slider, typing in a text box). When the current query changes, it is re-executed and the results update the visualization.

Layouts: ?? shows three possible interfaces that can express the queries in **??**. The blue boxes represent the bounding boxes in the layout hierarchy; for simplicity, we only depict layouts with widgets to the left of the visualization. For example, **??**(a) vertically organizes three buttons, where clicking on a button loads the corresponding query. (b) uses two dropdowns to change the column and string expressions, respectively, and uses a toggle widget to specify whether the `WHERE` clause should be in the query. The toggle and dropdown for the string expression are organized together because they relate to the same parts of the AST. (c) uses the same layout as (b), but uses the available width to list both column expressions (`Sales`, `Costs`) as buttons organized horizontally.

We represent these layouts using a hierarchical data structure called a *Widget Tree*, where each node corresponds to a layout or interaction widget (**??**). Layout widgets such as vertical and horizontal specify how to organize their children[1], while interaction widgets[2] such as Button and Dropdown are configured with subtrees (e.g., q_1) or values (e.g., 'Sales', 'Costs').

[1]Our layout widgets include: horizontal layout, vertical layout, tabs, and an adder that adds a copy of its child widget to the interface (e.g., to add multiple predicates).

[2]Our interaction widgets include: label, textbox, dropdown, slider, range slider, check boxes, radio buttons, and buttons.

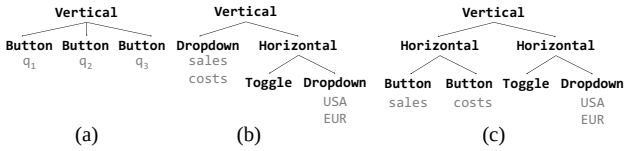

Figure 3: Widget Trees for the interfaces in **??**.

Widgets: We model a widget as a function $w(q, u) \rightarrow q'$, where a user interaction picks u from a domain of possible values, which is then used to change the current query q to a new query q'. To do so, the widget replaces the subtree at a fixed path in q's AST with a new subtree derived from u. For example, let $q = q_1$ be the current query in **??**(a); clicking on the q_2 button replaces the root of q with q_2's AST. In contrast, when the user selects "Costs" from the top drop-down in **??**(b), the `ColExpr` node in the AST will be replaced with a `ColExpr` node whose value is "Costs". Similarly, clicking the toggle widget will swap between the current subtree rooted at the `WHERE` node and an empty subtree that corresponds to the absence of a `WHERE` clause in the query (q_3). Each widget has a fixed size only depending on the domain. For example, the button widget in **??**(c) is used to choose from the domain – 'Sales' and 'Costs'. If the domain is larger, then there are more buttons; if it contains a longer word, the buttons will be wider. To support widgets that vary in size, we discretize the sizes and define a separate widget for each size. For example, for the button widget, we predefine small, medium and large button templates separately.

In short, each widget offers the user a choice from a domain of subtrees, and then places the chosen subtree at a widget-specific fixed location in the current AST. The three layouts primarily differ in the paths and granularities of the subtrees that the widgets replace: layout (a) replaces the root of the current AST, whereas layouts (b) and (c) replace leaves and interior nodes of the AST.

The Interface Generation Problem

Our problem is to identify changes within the input query sequence, and map them to an appropriate widget tree that can be rendered as an interactive interface. The challenge is that the layout and the selected widgets are intertwined with the process of identifying subtree differences between the input query ASTs.

To facilitate this process, we encode the layout and input queries in a `difftree`. Each node in the `difftree` corresponds to a (possibly empty) sequence of AST nodes. There are four node types that encode differences and similarities between the input queries. `ANY` can choose one of its child nodes, `OPT` has a single child that is optional, `MULTI` has a single child that can be chosen zero or more times, and `ALL` requires all of its children to exist. We call `ANY`, `OPT`, `MULTI` *choice nodes*. Note that an AST is a special case of a `difftree`, where each AST node is an `ALL` node.

A given query is expressed as the set of choices made for the choice nodes in the `difftree`. For example, **??** is a `difftree` with the root `ANY`—choosing any of its children is equivalent to one of the input queries. **??** illus-

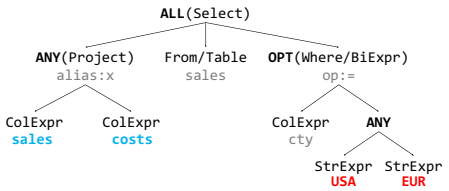

Figure 4: `difftree` for layouts **??**(b,c).

trates the `difftree` for layouts **??**(b,c). `ALL(Select)` states that all queries share the `SELECT` node as well as the `From/Table` nodes. However, the Project clause can be chosen from `Sales` and `Costs`, and the `where` clause is optional. Note that **??** can express more queries than the initial `difftree` in **??**.

Creating Widget Trees: Given a `difftree`, it is straightforward to derive a widget tree that can be rendered. Each *choice node* is mapped to one or more interactive widgets, and `ALL` nodes are mapped to layout widgets if it contains descendant choice nodes.

Cost Function: We quantify the cost of an interface based on the usefulness and appropriateness of the widget tree W (**?**). Each query $q \in Q$ is expressible by selecting the appropriate values for each widget in W. Thus, $U(q_i, q_{i+1}, W)$ models the minimum set of widgets that need to be changed in order to transform q_i into q_{i+1}. $M(\cdot)$ measures whether a selected widget is well-suited for the set of subtrees it expresses. For instance, a slider is well suited to select from a range of numeric values, but not arbitrary subtrees, whereas radio buttons are well suited for a small number of subtrees, but ill-suited for a large number.

$$C(W, Q) = \sum_{q_i \in Q} U(q_i, q_{i+1}, W) + \sum_{w \in W} M(w)$$

For reference, (**?**) only considered appropriateness when selecting widgets, and we borrow their $M(\cdot)$ cost functions. $U(\cdot)$ accounts for the size of the minimum spanning tree that connects the widgets that need to be changed, along with the cost to interact with each of those widgets. The cost function is a linear combination of terms that can be incrementally maintained as we explore the space of `difftrees` and widget trees. We consider a widget tree invalid (has infinite cost) if its size exceeds the output screen's size.

Our Approach

We now describe interface generation as a search problem, and our use of Monte Carlo Tree Search (MCTS) (**?**) to efficiently search the space for a good interface.

Search Space

Each state in the search space is a `difftree`, and the initial state is the list of input queries connected with an `ANY` node as the root. We define a set of transition rules that transform one `difftree` into another (**??**). The intuition is that the initial `difftree` represents a subset of the combinatorial enumeration of all expressible trees, and each rule factors out redundant substructures and variation between the trees. In the diagram, `x,y,z` represent subtrees that are distinguished by their root nodes—the roots of `x` and `x'` are the same, and different than the root of `y`.

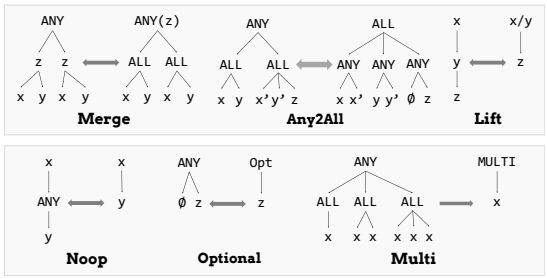

Figure 5: Set of transformation rules.

For instance, `Any2All` finds that `ANY` node's children all have children than can be aligned (`x`→`x'`, `y`→`y'`, `z`→ ∅), and groups the aligned nodes together[3]. With the exception of `Multi`, which replaces subtrees that repeat `x` with a `MULTI` node with a single `x` child, all rules are bidirectional.

Rules can be applied to choice nodes in the `difftree` that satisfy the rule's input pattern. The number of applicable rules for the current `difftree` determines the current search state's fanout. This primarily depends on the number of choice nodes and number of applicable rules for each choice node.

For example, Listing 1 show 10 input queries. The fanout is as high as 50, and a search path can be as long as 100 steps. It is impractical to enumerate the full search space to find the lowest cost `difftree`, and thus we propose the MCTS method described next.

Monte Carlo Tree Search

Monte Carlo Tree Search is used to balance exploration (trying unexplored states) with exploitation (exploring promising states) when searching in a large search space (**?**). In each iteration, it performs a randomized walk of the states, and estimates the reward of the final state. It then maintains a UCT score for each visited state s:

$$UCT^s = \frac{w_i^s}{n_i^s} + c\sqrt{\frac{\ln N_i^s}{n_i^s}}$$

Where w_i^s is the total reward for the state after the i^{th} iteration. The reward at the end of the random walk is added to every state along the path. n_i^s is the number of times the state was visited, N_i^s is the number of times s's parent state was visited, and c is a tunable exploration parameter.

In each iteration, we pick the state with the highest UCT, and perform a random walk of up to 200 steps from all of its immediate neighbor states. For the first iteration, we start with the initial state (e.g., Figure 1). To compute reward, we map the state (a `difftree`) to the lowest cost widget tree. During the search, we randomly assign widgets to the difftree k times and select the lowest cost. The reward is the negated cost. Once the search terminates after a fixed wall clock time, we enumerate all possible widget trees for the final difftree to find the lowest cost interface.

[3] ∅ represents no node.

(a) All queries, wide screen (b) All queries, narrow screen (c) Queries 6-8 (d) Interface w/ low reward (e) Original SDSS interface

Figure 6: (a-d) Generated interfaces from queries in **??**. (e) the pre-existing Sloan Digital Sky Survey search form. Screenshots only show widgets and do not include the visualizations.

Preliminary Results

We now present preliminary results when running our approach on the query log in **??**, which is derived from the Sloan Digital Sky Survey (**?**) query log. We run MCTS for around 1 minute to generate each interface. **??** shows that the layout and widget selections are sensitive to the input queries and screen constraints. (a) uses all queries in **??** as input, and generates a layout for a wider screen. It finds that the queries vary in the attributes that are selected (objid, count), as well as the number of results to return (top), and takes advantage of the wider screen to enumerate them as two sets of radio buttons. In contrast, (b) chooses dropdown widgets due to the narrower screen.

??(c) shows the interface is much simpler when queries 6–8 are use as input. These queries have the same `WHERE` clauses; since the three queries all have a `TOP` clause, the user is only asked to pick the number of rows to return (10, 100, 1000). (d) shows a low-reward interface, and illustrates that that poor interface choices are easily possible. Finally, (e) shows the original SDSS form.

```
1  select top 10 objid from stars
   where u between 0 and 30 and g between 0 and 30 and
         r between 0 and 30 and i between 0 and 30
2  select top 100 objid from galaxies
   where u between 1 and 29 and g between 10 and 30 and
         r between 9 and 30 and i between 3 and 28
3  select top 1000 objid from quasars where ...
4  select count(*) from stars where ...
5  select objid from galaxies where ...
6  select top 10 objid from quasars where ...
7  select top 100 objid from stars where ...
8  select top 1000 objid from galaxies where ...
9  select count(*) from quasars where ...
10 select objid from stars where ...
```

Listing 1: Example queries used in experiments. All queries have the same `WHERE` clause structure; for space considerations, we only show the full queries for the first two.

Ongoing Work

Although we have shown that the top-down approach can generate layout-sensitive interactive interfaces, there are a number of improvements needed for it to be practically useful in terms of functionality and performance.

A current limitation is that some combinations of widget choices may not make semantic sense; one approach is to integrate with a query engine to benefit from its query analysis phase, another is to leverage co-occurrence of subtrees in the query log to identify likely and unlikely combinations of widget choices. This can also inform the search phase. Further, we are extending the widgets to support parameterized sizes—for instance, a button or dropdown can be resized depending on the available screen space.

This work has not been optimized for performance—many of the algorithms perform exhaustive enumeration, and can benefit from optimizations such as parallelization, incremental computation of the `difftree` and cost functions, and search pruning. A key optimization opportunity is to accelerate the transformation rules, which become slow to evaluate as the `difftree` becomes large. Our goal is interactive run-times.

Acknowledgements: This work was supported by NSF IIS 1527765, 1564049, 1845638, and Amazon and Google awards.