On CS Undergrad Teaching and Visualization Research

Part 1 (teaching)
A Mini-lecture on sorting in different programming paradigms

Part 2 (research)
Visual Data Analysis: real users, real data, right methods

Michael Sedlmair
Visualization and Data Analysis group (VDA)
Part 1: Teaching

A Mini-lecture on Sorting in different programming paradigms
Assumptions wrt previous knowledge

• Target audience:
  • Mainly 2nd term CS students,
  • some 1st term

• Previous knowledge of
  • Programming: Java
  • Algorithms: Complexity analysis, data structures, recursion

• Learning goals
  • Repetition/introduction of sorting algorithms
  • First contact with programming paradigms, specifically imperative VS. declarative programming
Outline

• Sorting algorithms
  • Selection Sort
  • Quicksort

• Sorting in different programming paradigms
  • Imperative VS. declarative
  • Quicksort in Java (imperative)
  • Quicksort in Haskell (declarative/functional)
Sorting algorithms
Sorting
Sorting

- One of the standard algorithmic problems in Computer Science
- Goal: Put an unsorted list of elements $u$ in ascending/descending order.
- Example:
  $[7,3,3,1,5]$ $\rightarrow$ $[1,3,3,5,7]$
- (Note: special case of sorting numbers in ascending order)
Sorting algorithms

• Simple sorts
  • **Selection Sort**
  • Insertion Sort
  • Bubble Sort
  • ...

• More efficient sorts
  • **Quicksort**
  • Merge Sort
  • Heap Sort
  • Radixsort
  • ...

Selection Sort

• Split list into 2 parts:
  • $L$, sorted part of the list (left)
  • $R$, rest of the list (right)
• At the beginning $L = \emptyset$
• Identify the smallest element in $R$ and swap it with the first element of $R$
• Done when $|R| = 1$
Selection Sort

71 19 2 87 92 5 66 45
Selection Sort

\[ 71 \ 19 \ 2 \ 87 \ 92 \ 5 \ 66 \ 45 \]
Selection Sort

71 19 2 87 92 5 66 45

2 19 71 87 92 5 66 45
Selection Sort

$\begin{array}{ccccccc}
71 & 19 & 2 & 87 & 92 & 5 & 66 & 45 \\
\end{array}$

$\begin{array}{ccccccc}
2 & 19 & 71 & 87 & 92 & 5 & 66 & 45 \\
\end{array}$

$\begin{array}{ccccccc}
2 & 5 & 71 & 87 & 92 & 19 & 66 & 45 \\
\end{array}$
Selection Sort

\[ \begin{array}{ccccccc}
71 & 19 & 2 & 87 & 92 & 5 & 66 & 45 \\
\end{array} \]

\[ \begin{array}{ccccccc}
2 & 19 & 71 & 87 & 92 & 5 & 66 & 45 \\
\end{array} \]

\[ \begin{array}{ccccccc}
2 & 5 & 71 & 87 & 92 & 19 & 66 & 45 \\
\end{array} \]

\[ \begin{array}{ccccccc}
2 & 5 & 19 & 87 & 92 & 71 & 66 & 45 \\
\end{array} \]
Selection Sort

![Diagram of Selection Sort algorithm with numbers arranged in a grid and arrows pointing to the selected elements. The numbers are 71, 19, 2, 87, 92, 5, 66, 45, and the process of selecting the smallest element in each iteration is illustrated with arrows pointing to the selected numbers.](image-url)
Selection Sort

Initial list: 71, 19, 2, 87, 92, 5, 66, 45

1. Find the smallest element: 2
2. Swap it with the first element: 2, 19, 71, 87, 92, 5, 66, 45
3. Find the next smallest element: 5
4. Swap it with the second element: 2, 5, 71, 87, 92, 19, 66, 45
5. Find the next smallest element: 19
6. Swap it with the third element: 2, 5, 19, 87, 92, 71, 66, 45
7. Find the next smallest element: 45
8. Swap it with the fourth element: 2, 5, 19, 45, 92, 71, 66, 87
9. Final list: 2, 5, 19, 45, 66, 71, 92, 87
Selection Sort

71 19 2 87 92 5 66 45

2 19 71 87 92 5 66 45

2 5 71 87 92 19 66 45

2 5 19 87 92 71 66 45

2 5 19 45 92 71 66 87

2 5 19 45 66 71 92 87

2 5 19 45 66 71 92 87
Selection Sort: Strengths / Weaknesses

- **Strengths**
  - easy to understand
  - “in-place” (memory usage: $n$)
Selection Sort: Strengths / Weaknesses

• **Strengths**
  • easy to understand
  • “in-place” (memory usage: $n$)

• **Biggest Weakness**
  • Expensive for large lists: $O(n^2)$
Quicksort

• Divide-and-conquer algorithms

• Determine an arbitrary pivot element \( u_p \) (e.g., last element of a list)

• Swap elements in the list, so that
  • all elements left of \( u_p \) are smaller than \( u_p \)
  • all elements right of \( u_p \) are greater than \( u_p \)

• The pivot \( u_p \) is then already at its final position

• Recursively sort left and right parts of the list
Quicksort

• Moving the pivot:
  • Go from left to right until you find an element $u_i$ with $\text{value}_i > \text{value}_p$
  • Go from right to left until you find an element $u_j$ with $\text{value}_j < \text{value}_p$
  • Swap these elements
  • Repeat until $i$ and $j$ meet
  • Swap pivot element with the element at position $i$ if $\text{value}_p < \text{value}_i$
Quicksort

71 19 2 87 92 5 66 45
Quicksort

71 19 2 87 92 5 66 45

71 19 2 87 92 5 66 45
Quicksort
Quicksort
Quicksort
Quicksort: Strength / Weaknesses

• Strength:
  • Average complexity: $O(n \cdot \log_2(n))$
  • “in-place” memory
Quicksort: Strength / Weaknesses

• **Strength:**
  • Average complexity: $O(n \cdot \log_2(n))$
  • “in-place” memory

• **Weaknesses:**
  • Worst case complexity: $O(n^2)$
  • not efficient for small lists
Sorting in different programming paradigms
(using the example Quicksort)
Programming paradigms

• Fundamental style or “way” of programming

• Some examples

  • Imperative — Give an explicit sequence of commands
  • Declarative — Programs specify the result, not how to get it.
  • Object-Oriented — Computation is effected by sending messages to objects; objects have state and behavior.
  • ...

• Not mutually exclusive concepts! Specifically, languages can follow more than one paradigm
Imperative VS. declarative
Imperative VS. declarative

• Step-by-step recipe
• Describe the how: Control flow in imperative programming is explicit; commands show how the computation takes place.
• Examples: Java, C++, C, …
public static void qsort (int[] val, int low, int high) {
    if (high - low > 0) {
        //get pivot element from the end of the list
        int pivot = high;
        //initialize pointers
        int l = low, h = high - 1;
        while (l < h) {
            //set pointers
            while ((l < h) && (val[l] <= val[pivot])) {
                l++;
            }
            while ((l < h) && (val[h] >= val[pivot])) {
                h--;
            }
            //swap values
            if (l != h) {
                swap(val, l, h);
            } else if (val[l] > val[pivot]) {
                swap(val, l, pivot);
            }
        }
        //recursion
        quicksort(val, low, l - 1);
        quicksort(val, l + 1, high);
    }
}
public static void qsort (int[] val, int low, int high) {
    if (high - low > 0){
        //get pivot element from the end of the list
        int pivot = high;

        //initialize pointers
        int l = low, h = high - 1;

        while (l < h) {
            //set pointers
            while ((l < h) && (val[l] <= val[pivot])) {
                l++;
            }
            while ((l < h) && (val[h] >= val[pivot])) {
                h--;
            }

            //swap values
            if (l != h) {
                swap(val, l, h);
            } else if (val[l] > val[pivot]) {
                swap(val, l, pivot);
            }
        }

        //recursion
        quicksort(val, low, l - 1);
        quicksort(val, l + 1, high);
    }
}
public static void qsort (int[] val, int low, int high) {
    if (high - low > 0){
        //get pivot element from the end of the list
        int pivot = high;
        //initialize pointers
        int l = low, h = high - 1;
        while (l < h) {
            //set pointers
            while ((l < h) && (val[l] <= val[pivot])) {
                l++;
            }
            while ((l < h) && (val[h] >= val[pivot])) {
                h--;
            }
            //swap values
            if (l != h) {
                swap(val, l, h);
            } else if (val[l] > val[pivot]) {
                swap(val, l, pivot);
            }
        }
        //recursion
        quicksort(val, low, l - 1);
        quicksort(val, l + 1, high);
    }
}
public static void qsort (int[] val, int low, int high) {

    if (high - low > 0) {
        //get pivot element from the end of the list
        int pivot = high;

        //initialize pointers
        int l = low, h = high - 1;

        while (l < h) {
            //set pointers
            while ((l < h) && (val[l] <= val[pivot])) {
                l++;
            }
            while ((l < h) && (val[h] >= val[pivot])) {
                h--;
            }

            //swap values
            if (l != h) {
                swap(val, l, h);
            } else if (val[l] > val[pivot]) {
                swap(val, l, pivot);
            }
        }

        //recursion
        quicksort(val, low, l - 1);
        quicksort(val, l + 1, high);
    }
}
public static void qsort (int[] val, int low, int high) {

  if (high - low > 0) {
    //get pivot element from the end of the list
    int pivot = high;

    //initialize pointers
    int l = low, h = high - 1;

    while (l < h) {
      //set pointers
      while ((l < h) && (val[l] <= val[pivot])) {
        l++;
      }
      while ((l < h) && (val[h] >= val[pivot])) {
        h--;
      }

      //swap values
      if (l != h) {
        swap(val, l, h);
      } else if (val[l] > val[pivot]) {
        swap(val, l, pivot);
      }
    }

    //recursion
    quicksort(val, low, l - 1);
    quicksort(val, l + 1, high);
  }
}
public static void qsort (int[] val, int low, int high) {
    if (high - low > 0) {
        // get pivot element from the end of the list
        int pivot = high;

        // initialize pointers
        int l = low, h = high - 1;

        while (l < h) {
            // set pointers
            while (((l < h) && (val[l] <= val[pivot]))) {
                l++;
            }
            while (((l < h) && (val[h] >= val[pivot]))) {
                h--;
            }

            // swap values
            if (l != h) {
                swap(val, l, h);
            } else if (val[l] > val[pivot]) {
                swap(val, l, pivot);
            }
        }

        // recursion
        quicksort(val, low, l - 1);
        quicksort(val, l + 1, high);
    }
}
public static void qsort (int[] val, int low, int high) {
    if (high - low > 0) {
        //get pivot element from the end of the list
        int pivot = high;

        //initialize pointers
        int l = low, h = high - 1;

        while (l < h) {
            //set pointers
            while ((l < h) && (val[l] <= val[pivot])) {
                l++;
            }
            while ((l < h) && (val[h] >= val[pivot])) {
                h--;
            }

            //swap values
            if (l != h) {
                swap(val, l, h);
            } else if (val[l] > val[pivot]) {
                swap(val, l, pivot);
            }
        }

        //recursion
        quicksort(val, low, l - 1);
        quicksort(val, l + 1, high);
    }
}
Imperative VS. declarative

• **What** not the **how**

• “if you can describe the problem, you have solved the problem” [Brian Tarbox]

• Higher-level abstraction

• Main benefit:
  • Code is much shorter
  • Less error-prone

• **Example:** Haskell (functional)
Declarative/functional: Quicksort in Haskell

qsort [] = []
qsort (p:xs) = qsort lesser ++ [p] ++ qsort greater
where
    lesser = [ y | y <- xs, y < p ]
    greater = [ y | y <- xs, y >= p ]

- [] is the empty list
- (p:xs)
- [p] is a list with one element
- ++ concatenation of two list
- lesser / greater filters the list
Summary Mini-Lecture

• Sorting
  • One of the canonical algorithmic problems in Computer Science
  • Many algorithms exist
  • Examples: Selection Sort (simple), Quicksort (more efficient)

• Programming paradigms
  • Imperative VS. declarative
  • Quicksort in Java VS. Quicksort in Haskell
Further references

• Quicksort in 42 different programming languages:  
  https://en.wikibooks.org/wiki/Algorithm_Implementation/Sorting/Quicksort

• Books
  • Tate, Bruce A. (2010), Seven Languages in Seven Weeks: A Pragmatic Guide to Learning Programming Languages (1st ed.), Pragmatic Bookshelf.

• Other lectures & special thanks (!)
  • Konzepte der Informatik, 2013, Prof. Deussen  
    https://streaming.uni-konstanz.de/lectures/wintersemester-2013/inf-11700-20132/
  • Konzepte der Programmierung, 2014, Prof. Scholl  
    https://streaming.uni-konstanz.de/lectures/sommersemester-2014/inf-12070-20141/
  • ACM curriculum  
Part 2: Research

Visual Data Analysis: real users, real data, right methods
data has become ubiquitous

visualization & data analysis
My Focus

• **real users/real data**
  • problem-driven: design studies
  • technique-driven: high-dimensional data analysis

• **right methods**
  • innovate and refine research methods for visual data analysis
real users/real data:

Applied visualization projects (design studies)

High-dimensional data analysis techniques

right methods:

Novel and refined research methods
real users/real data

Applied visualization projects (design studies)

High-dimensional data analysis techniques

right methods:

Novel and refined research methods
real users/real data:

Cardiogram: Visual Analytics for Automotive Engineers

more and more electronics...
... enabled by in-car communication networks

~70 Controllers / Car
**Problem Characterization**

- **data**: recorded traces (15k messages/sec)
- **task**: finding errors
- **process**: test drives
- **current practices**: mainly textual lists
Design Requirements

• **Main Challenges**
  
  • handling masses of test traces (large and many)
  
  • understanding correlation between trace and car behavior
  
  • ...

Cardiogram overview
Cardiogram
state machine pre-analysis
State Machines: Error Detection

Correct \quad \quad Error
Cardiogram
create/select state machines

Multiple State Machines
- State Machine 1
- State Machine 2
- State Machine 3
... (dozens)
Cardiogram

data abstraction/reduction

Trace

State Machine Pre-Analysis

Visualization

from each state machine

Verification Tag
error / warning / ok

Transition List
time: state x → state y ...
Hey Günter,

I think I know the reason for this. Please contact me so we can talk about it.

Helmut
WARN ! Neg Antwort - Grund 21
OK ! Anfrage FirstFrame
OK ! Pos Antwort (PC)
OK ! Pos Antwort (SF)
OK ! Ueberwachung aktiv
OK ! Pos Antwort (FF)
OK ! Anfrage SingleFrame
INIT ! Functional_Request_7B_1st

Key: Gunther, I think I know the reason for this. Please contact me so we can talk about it. Helmut.
Longitudinal field study

1 year, 15 engineers
Longitudinal field study results

• externalization and sharing of expert knowledge
Longitudinal field study results

- externalization and sharing of expert knowledge
- complete coverage of traces vs. sparse sampling
Longitudinal field study results

- externalization and sharing of expert knowledge
- complete coverage of traces vs. sparse sampling
- understand behavioral correlations

<table>
<thead>
<tr>
<th>Door 1</th>
<th>Door 2</th>
<th>Door 3</th>
<th>Door 4</th>
</tr>
</thead>
<tbody>
<tr>
<td>closed</td>
<td>closed</td>
<td>closed</td>
<td>closed</td>
</tr>
<tr>
<td>open</td>
<td>open</td>
<td>open</td>
<td>open</td>
</tr>
</tbody>
</table>

- Overpressure error: ok
Longitudinal field study results

- externalization and sharing of expert knowledge
- complete coverage of traces vs. sparse sampling
- understand behavioral correlations

deployed and adopted
how to?

“A design study is a project in which visualization researchers analyze a specific real-world problem faced by domain experts, design a visualization system that supports solving this problem, validate the design, and reflect about lessons learned in order to refine visualization design guidelines.”

[Sedlmair et al.: Design Study Methodology: Reflections from the Trenches and the Stacks, InfoVis 2012]
design studies: long and winding road with many pitfalls!
right methods:

Design Study Methodology: Reflection from the Trenches and the Stacks

M. Sedlmair, M. Meyer, T. Munzner. TVCG (Proc. InfoVis 2012). **Best paper honorable mention.**
main contributions

guiding other researchers with ...

Definitions

9-stage framework

32 pitfalls
future work

short-term
1. Design studies

• real users/real data:
  • contribute to their fields
  • improve our guidelines

• right methods:
  • validate, improve, extend design study methodology
Researchers in this domain build models of how different anatomical shapes are demarcated. The computational model here is therefore a segmentation algorithm that segment a medical image. The goal is to find input parameter configurations that lead to "good" segmentation of a molecular image quantification of dynamic positron emission tomography (PET) data. Such data is difficult to segment due to severe noise and partial volume effects. Our user therefore developed an extension to the random walk segmentation method by adding input parameters that account for desirable criteria such as data fidelity, shape prior, intensity prior, and regularization.

In order to attain a superior segmentation quality, a proper configuration of input parameters, our user specified additional numerical parameters. Our users' model can be used to describe individual anatomic regions such as the cerebellum and putamen. Labels can be found in [1].

Figure 3.4 Example: Fuel Cell Prototyping

A fuel cell takes hydrogen and oxygen as input and produces outputs of electric current. Affordable, high-performance fuel cells have the potential to enable more environmentally friendly means of transport by greatly reducing emissions. A reliable synthetic model that can simulate different design options than real prototyping, such as current density, temperature, or relative humidity variance across the geometry of the cell stack, is desirable to visually sort or relative humidity variance across the geometry of the cell stack.

In the following, we describe abstractions for tasks and requirements that generalize across the three use cases we considered in this study. These challenges were not supported by our user's current practices. Yet by visual inspection, it becomes apparent that the segmentation quality does not decay too quickly for slight changes to the chosen parameter settings. A proper understanding of the sensitivity of certain parameter settings is necessary. Hence, extracting the space of solutions that lead to 'good' segmentation results enables the user to pick a robust solution within that space. These requirements were also particularly interested in getting a deeper understanding of the behavior of the cell stack, such as current density, temperature, pressure, concentration, and produce outputs of different plots of various physical sensitivities.


Design studies

Data-driven Journalism

• **users**: journalists
• **tasks**: story creation
• **data**: text over time, e.g., parliament speeches

FFG-funded VALiD project (690,000 Euro)
Aigner (PI), **Sedlmair (Co-PI)**, Ausserhofer (Co-PI), Popper (Co-PI)
2. High-dim data analysis techniques

• real users/real data:
  • understand techniques in real circumstances
  • build better techniques

• right method
  • combine approaches from HCI, Vis, Psychology, and Machine Learning
High-dim techniques: visual quality measures

• real users/real data

• right method
  [Sedlmair, Aupetit, Data-driven Evaluation of Visual Quality Measures, EuroVis 2015]

• new techniques
  [Aupetit, Sedlmair: Learning human visual judgment of class separation in colored scatterplots for Visual Analytics, NIPS 2015 - in submission]
High-dim techniques:

erother ... 

• clustering
• dimension reduction
• sensitivity analysis
• ...
future work

long-term
Vision:

The “cookbook” for Visual Data Analysis

abstract data & task

↓

data analysis technique
The “cookbook” for Visual Data Analysis

blocks and guidelines

design decision patterns
ongoing project:
with Heidi Lam & Miriah Meyer
The “cookbook” for Visual Data Analysis

natural science vs. design research

vast amounts of real users/real data work necessary

[Sir Francis Bacon 1561-1626]
Thanks to you & to …

VDA Group @ University of Vienna

Many, many other collaborators

…
Questions?

On CS Undergrad Teaching and Visualization Research

1. Sorting in different programming paradigms

2. Visual Data Analysis: real users, real data, right methods

Michael Sedlmair
Visualization and Data Analysis group (VDA)