UNIVERSITY OF TARTU
FACULTY OF MATHEMATICS AND COMPUTER SCIENCE
Institute of Computer Science

Anna Aljanaki

Illustrative Applications on Algorithms and Data Structures

Bachelor thesis (4 CP)

Supervisor: Ain Isotamm

Author: ……………………………………… “….” juuni 2008
Supervisor: ………………………………….. “….” juuni 2008
Professor: …………………………………... “….” juuni 2008

TARTU 2008

Contents

Introduction....3
Goal....3
Target audience and prerequisites....3
Outline....4
Chapter 1....5
Web-based education from algorithms study viewpoint....5
1.1 Computer-based training in study of algorithms....5
1.2 Application implementation....6
Chapter 2....8
Realisation....8
2.1 Sorting methods....8
Bubble sort....10
Insertion sort....10
Divide-and-conquer sorting....11
Quicksort....11
Mergesort....13
Heapsort....14
2.2 Graph algorithms....15
Dijkstra's algorithm....16
Floyd-Warshall's algorithm....17
Kruskal's algorithm....18
Prim's algorithm....19
2.3 Text algorithm....19
Knuth-Morris-Pratt's algorithm....20
2.4 Data structures....21
AVL-tree....21
Summary....23
Bibliography:....25
Appendix....26

Introduction

The aim of this work is to create an illustrative interactive teaching material, which can be used as a supplementary training aid for the course „Algorithms and data structures". For years this course has constituted a classical part of computer science teaching curriculum. There are numerous schoolbooks concerning algorithms, numeric theory, compexity theory and data structures, among the classical ones can be mentioned *Donald Knuth*'s „The Art of Computer Programming" [5], *Thomas H. Cormen*'s „Introduction to Algorithms" [1]. There are also books, considering problems of implementation of algorithms in different programming languages, usually these are such widespread languages as *C++*, *Java*, *Delphi*. In the University of Tartu the above-mentioned course belongs to many different study curricula, such as: computer science, information technology, genetics, mathematical statistics and molecular engeneering. Such a multiplicity derives from wide adoptation of algorithms and data structures in scientific research and practical applications.

Goal

By definition of *Thomas H. Cormen,* „An algorithm is any well-defined computational procedure that takes some value, or set of values, as input and produces some value, or set of values, as output" [1]. Algorithms are applied in all fields of computer science, helping to solve different practical problems more effectively. In genetics they are necessary to determine sequence of chemical base pairs, that make DNA, data analysis; in networking to find good routes for data traveling, searching through huge amounts of data; in cryptography find application numerical algorithms and number theory.

The present work's goal is to illustrate work of selected algorithms on examples. It should help student to understand principles of algorithms in general and of their implementation in Java in particular.

Target audience and prerequisites

This work will be of particular interest to computer science and information technology students, and also for those students, regardless of their study curriculum, who are taking part in course „Algorithms and Data Structures". Most useful it will be to students,

who are familiar with *Java*, because most concrete examples (algorithm implementations) in the work are given in *Java*.

Outline

The work consists of two chapters. In the first chapter will be discussed the distinctive features of interactive self-learning online on example of concept of learning objects. Also the structure of program outline will be given.

In second chapter the overview of application constructed will be discussed, with short description of every algorithm included.

Chapter 1

Web-based education from algorithms study viewpoint

This chapter gives basic overview of computer-based learning advantages and relevance and appropriateness as viewed from investigated standpoint. The developed application will be discussed.

1.1 Computer-based training in study of algorithms

At present days computer-based learning is gaining more and more popularity. It is expected that using contemporary technologies, internet development and computer science evolution will make learning and teaching more effective, attractive and accessible. The most appealing to learners seem such advantages of e-learning, as compact information presentation, visuality, easy and convienient search possibility, twenty-four-hour access.

And, moreover, especially if the case in point is studying algorithms and data structures, computer-based training can be considered most natural way of learning. Firstly, execution of algorithm on computer is faultless, secondly, concrete implementations in programming languages can be involved.

To describe the learning entity in computer-based course, frequently a term learning object is used. Giving a definition to this term is no trivial task. The most popular and most cited definition is *IEEE* own: „any entity, digital or non-digital, that may be used for learning, education or training " [3]. However, for the present day this definition is much too wide: it seems more appropriate to constraint the term to digital environment only. This is how defines learning objects *Mike Sosteric*: „A LO is a digital file (image, movie, etc.) intended to be used for pedagogical purposes, which includes, either internally or via association, suggestions on the appropriate context within which to use the object." [2], *Wikipedia*: „A learning object is a resource, usually digital and web-based, that can be used and re-used to support learning. Learning objects offer a new conceptualization of the learning process: rather than the traditional "several hour chunk", they provide smaller, self-contained, re-usable units of learning" [9].

The concept of learning object turns out to be suitable when dealing with studying algorithms. In this case building a single application is not that appropriate. Separate learning

objects allow to achieve the goal of implementation independent programs, preserving compositional unity.

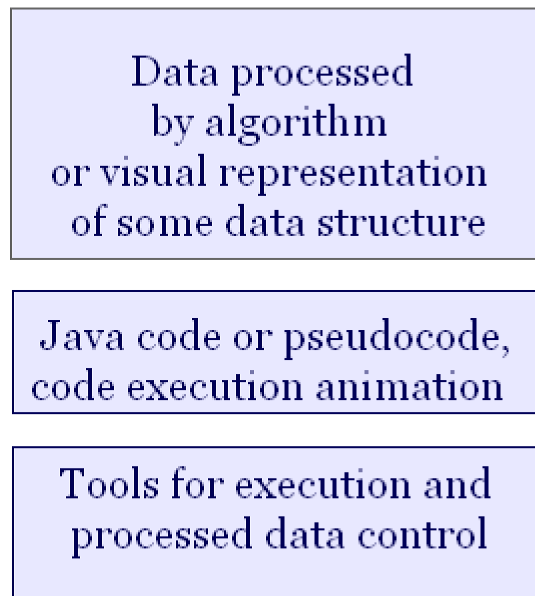
1.2 Application implementation

When designing the program, user convenience has been taken as a guideline. For reasons discussed in the previous chapter, it was decided that webpage form is most appropriate in present case. Technology choice proved to be not a simple task with no straightforward solution. Examining educational applications, one can make a conclusion, that most of them are applets. However, applets have many considerable drawbacks, which make them quite an unattractive tool. Applets load noticeably slowly, have quite a big size, especially if external libraries, for instance for graph layout etc. are used. *Java* graphical tools are insufficiently elaborated.

For these reasons it was decided in favour of *Flash* technology. *Flash* has excellent graphic tools, *Flash*-files have minimal weight. In 2007 new powerful object-oriented language, based on *ECMAScript*, was introduced: *ActionScript* 3.0. The code is now executed on *ActionScript* Virtual Machine 2. Simultaneously the goal was to learn and investigate new language possibilities.

In digital environment learning objects need some contextual information, so learning objects must be accompanied with background information. Metadata function is accomplished with present text.

A component architecture scheme of educational applications is presented on the following picture:



The upper sector of application may contain an array of integers, weighted graph, matrix or text. This is the input of algorithm. Code panel can contain as *Java* code, so and pseudocode, depending on the data processed. If *Java API* provides common implementation for the data structure, such as arrays or strings, then *Java* code is used, if data structure is more comlicated (graphs, trees), than pseudocode is given.

This work does not aspire to any broad coverage of algorithms, only the main ones have been implemented. The choice was determined by classical books on algorithms and data structures, such as *T. H. Cormen* “Introduction to algorithms” [1] and *D. Knuth*, “The Art of Computer Programming”[5]. Also *J. Kiho* schoolbook [7], [8] was taken in consideration.

Chapter 2

Realisation

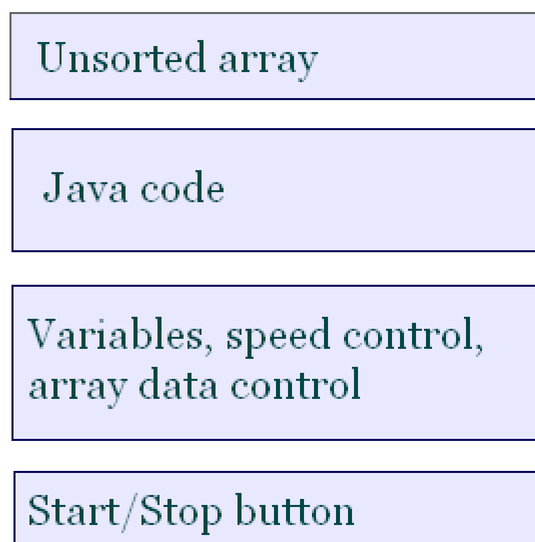
This chapter provides necessary theoretical background to the program. Every algorithm, chosen for the educational application, is discussed in detail. Also the code, used in application for illustration, is provided along with necessary comments.

The implemented algorithms can be divided into three groups:

- Sorting methods
- Graph algorithms (shortest path finding, minimum spanning tree determining)
- Text search
- Data structures

2.1 Sorting methods

There have been chosen five sorting algorithms: bubble sort, insertion sort, quick sort, merge sort and heap sort. The structure of applications concerned is presented on the following picture:



The upper panel contains an array of 10 elements. By default they are filled randomly. The next panel contains code in Java. Code is colored in IDE *Eclipse*-like way. Code comments are not present in the animation due to lack of space. However, they are present in the explainatory text and in the current text. In general, code consists of one method only. In some cases there can also be some auxiliary methods. Next part contains variables, which change during program execution (such as loop variables, in some cases, if they are essential for programm execution path, temporary variables, holding array data). There is also slider to control execution speed. The speed is changed at runtime (live dragging is enabled). Lastly, there is a checkbox, where one can determine the input data. By default, as it was mentioned above, random values are chosen. User can change them to backward sorted array, or input values by himself. For most sorting algorithms backward order is the worst case, so it is of particular interest for investigating how algorithm behaves in it. It may seem, that algorithms rarely face worst case input in practice. Actually, this is not true. For instance, in database search the worst case would occur, if information is not present. The result looks like that:

pivot

4	18	15	24	40	36	28	44	47	47

```
void quickSort(int[] array){
   Stack<Integer> S = new Stack<Integer>();
   int l = 0;   int r = a.length-1;
   S.push(l); S.push(r);
   while (!S.empty()){
     r = S.pop(); l = S.pop();
     if (r > l){
        int i = partition(a,l,r);
        S.push(l); S.push(i-1);}
        S.push(i+1);   S.push(r);
     }
   }
}
int partition(int a[], int left, int right) {
int pivot = a[left];  int p = left;
   for (int r = left + 1; r<=right;r++){
      if (a[r]<pivot){
        a[p] = a[r]; a[r] = a[p+1]; a[p+1] = pivot; p++;
      }
   }
return p;
}
```

Variables:
r = 6   left = 0   p = 0
l = 0   right = 6

Speed: 3 s

Random
Random
Backward
User input

STOP

This is a picture of program in action. The highlighted line is currently executed. Between two lines of code there will be delay 3 seconds. Array is filled with random values. Current assignments of the variables are shown in the „Variables“ section.

Bubble sort

Bubble sort is probably the simpliest sorting algorithm known. The principle of bubble sort is looping through array n^2 times, exchanging every two adjacent elements if needed. The name of the algorithm derives from the similarity of array elements, gradually rising to the top, to the bubbles of air in the water.

The algorithm was included in order to provide background for comparison with other, more efficient algorithms. Bubble sort's complexity is $O(n^2)$.

The code in Java, used in the illustrative flash application:

```
void bubbleSort(int[] array) {
         //path through array n times
          for (int i=0; i < array.length-1; i++) {
              for (int j=0; j < array.length-1-i; j++){
                //swap to elements if needed
                   if (array[j] > array[j+1]) {
                       int temp = array[j+1];
                       array[j+1] = array[j];
                       array[j] = temp;
                   }
               }
            }
        }
```

Bubble sort does best on a nearly sorted array. If number of elements is quite big, it is extremly inefficient. It's Java implementation is also extremely inefficient, because of large quantity of writing operations needed.

Insertion sort

The idea of insertion sort is looping through array, putting consequent element on it's correct place on every step. That means, if algorithm has reached *n*-s element, elements from 0 to *n*-1 are already in sorted. The following code was used:

```
public static void insertionsort(int[] a) {
      for (int i = 0; i < a.length; i++) { // path through array
            for (int j = i; j > 0; j--) { // go down from i to first element
                  if (a[j] < a[j - 1]) { // swap two elements if needed
                        int temp = a[j];
                        a[j] = a[j - 1];
                        a[j - 1] = temp;
                  }
            }
      }
}
```

Insertion sort's worst case complexity is $\Theta(n^2)$. Conducting complexity analysis, we see, that algorithm's complexity is polynomial:

$$T(n) = c_1 n + c_2(n-1) + c_3(n - 1) + \frac{c_4 n(n+1)}{2} - 1) + \frac{c_5 n(n-1)}{2} - 1) + \frac{c_6 n(n-1)}{2} - 1) + c_7(n-1) = (\frac{c_5}{2} + \frac{c_6}{2} + \frac{c_7}{2})n^2 + (c_1 + c_2 + c_4 + \frac{c_5}{2} + \frac{c_6}{2} + \frac{c_7}{2})n - (c_2 + c_4 + c_5 + c_7).$$

However, in practice insertion sort is twice as fast as bubblesort. Insertion sort is easy to understand and implement, and in its average case it does with $n^2/4$ comparisons. Due to the fact, that insertion sort starts sorting from first element upwards, it can start before the whole array is loaded completely. In some cases, if an array gradually grows, insertion sort can be useful. It is stable sort, which means, that the order of equal elements never changes.

The best case for insertion sort is nearly sorted array, the worst backwards sorted one.

Divide-and-conquer methods of sorting

The next three algorithms discussed follow divide-and-conquer approach. It means, they break the problem into several subproblems and deal with each separately, and then combine solution into solution for original problem. Such algorithms are by their nature recursive. However, as far as the following algorithms are most suitable for huge sorting array holging big amount of data, due to problems of recursion it was decided to provide iterative implementations.

Quicksort

The quicksort is one of the most popular sorting algorithms. It was developed by C. A. R. Hoare in 1960. In average case quicksort makes O(n *log* n) comparisons, however, in worst case – $\Theta(n^2)$. The principle of quicksort is:

- chose one element – pivot
- sort two parts of array so that in one part all the elements are smaller than pivot, in other all the elements are bigger (partition)
- apply quicksort to both parts

Quicksort is not stable sort. For several reasons, the nonrecursive version of quicksort has been chosen: firstly, recursive quicksort often faces stack overflow problem, secondly, it proved to perform slower, and thirdly, non-recursive algorithm is easier to visualise and understand from visualization.

The most important part of the algorithm is partitioning. The worst case partitioning recurrence occurs, when one subproblem arises as with $n - 1$ elements, so and with 0 elements. Since call on an array of 0 elements just returns, $T(0) = \Theta(1)$. Recurrence calls:

$$T(n) = \Theta(n) + T(n - 1) + T(0) = \Theta(n) + T(n - 1)$$

In best partitioning case both partitions produced are equal. In this case $T(n) = O(n \lg n)$.

Java code:

```
void quickSort(int[] array) {
	Stack<Integer> S = new Stack<Integer>();//stack for recursive calls
	int l = 0;                              // the start position
	int r = array.length - 1;               // the end position
	S.push(l);
	S.push(r);
	while (!S.empty()) {                    // if there are still calls
		r = S.pop();
		l = S.pop();
		if (r > l) {                    // if end is bigger than start
			int i = partition(array, l, r);
			if (i - l > r - i) {
				S.push(l);
				S.push(i - 1);
			}
			S.push(i + 1);
			S.push(r);
			if (r - i >= i - l) {
				S.push(l);
				S.push(i - 1);
			}
		}
	}
}

int partition(int a[], int left, int right) {
	int pivot = a[left]; // element which will serve as separator
	int p = left;
	for (int r = left + 1; r <= right; r++) {   //path through partition
		if (a[r] < pivot) {
			a[p] = a[r];      // swap next element with pivot if needed
			a[r] = a[p + 1];
			a[p + 1] = pivot;
			p++;
		}
	}
	return p;
}
```

This implementation is based on *Sedgewick's* in “Algorithms in Java“ [4].

Mergesort

Mergesort was invented by *John von Neumann* in 1945. It is a stable sort. Merge sort's best use is for sorting linked lists.

Mergesort’s principle can be described like that:

- If the array consists of 0 or 1 element, then it is sorted
- Divide array into two subarrays and apply merge sort recursively
- Merge subarrays back into original array

The idea of mergesort is based on the fact, that smaller list is easier to sort, than big one. Merge function takes only one path through both arrays to unify them. Mergesort’s complexity is Θ(n log n).

Mergesort's implementation provides possibility to view what is going on in an auxiliary array. For that an additional array of 10 values is provided. At the picture red are elements of two partes of array, which are being merged.

a: 9 10 7 8 6 5 4 3 2 1

aux: 10 9 8 7 6 5 4 3 2 1

```
void mergesort (int[] a) {
  int l=0, r = a.length -1;
  int[] aux = new int[a.length];
  for (int m = 1; m <= r-l; m = m+m)
    for (int i = l; i <= r-m; i += m+m)
      merge(a, i, i+m-1, Math.min(i+m+m-1, r), aux);
  }
void merge(int[] a, int l, int m, int r, int[] aux)
 { int i, j;
  for (i = m+1; i > l; i--) aux[i-1] = a[i-1];
  for (j = m; j < r; j++) aux[r+m-j] = a[j+1];
  for (int k = l; k <= r; k++)
   if (aux[j] < aux[i])
     a[k] = aux[j--]; else a[k] = aux[i++];
 }
```

Variables:
i= 4
m= 0

Speed: 1 s

Backward

STOP

Recursive merge sort versions make 2*n*-1 recursive calls, producing large overhead. This is why, and also for similar reasons as discussed in case of quicksort a non-recursive version of merge sort has been chosen.

Code:

```
void mergesort(int[] a) {
      int l = 0, r = a.length – 1; // start and end of partition
      int[] aux = new int[a.length];//array for holding subarrays  temporarily
      for (int m = 1; m <= r - l; m = m + m) //
            for (int i = l; i <= r - m; i += m + m)
                  merge(a, i, i + m - 1, Math.min(i + m + m - 1, r), aux);
}

void merge(int[] a, int l, int m, int r, int[] aux) {
      int i, j;
      for (i = m + 1; i > l; i--)
            aux[i - 1] = a[i - 1];
            for (j = m; j < r; j++)
                  aux[r + m - j] = a[j + 1];
            for (int k = l; k <= r; k++)
                  if (aux[j] < aux[i])
                        a[k] = aux[j--];
                  else
                        a[k] = aux[i++];
            }
```

Merge procedure takes time Θ (n). Recurrence for the worst case running time T(n) = Θ(n log n).

Implementation of algorithm is based on *Sedgewick*'s book "Algorithms in Java" [4].

Heapsort

Heapsort is an unstable sorting algorithm, which uses auxiliary data structure – heap. The binary heap data structure is a nearly complete binary tree. Heapsort uses two operations on a heap: node insertion and root deletion. On every pass the root, which is the largest (or smallest – in minheap) value is extracted untill the heap is empty. After each root deletion operation heap must be rebuilt to return preserve heap properties. It's worst case is already sorted array, best case reversed array.

Java code:

```
void heapsort(int a[]) {
      // build a heap on basis of array a
      heapify(a);
      //consequently remove root from heap
      for (int i = a.length - 1; i > 0; i--) {
            int temp = a[i];
            a[i] = a[0];
            a[0] = temp;
           //rebuild heap
            sift(a, i, 0);
      }
}
```

```
void sift(int a[], int size, int element) {
      int i, j;
      j = element;
      do {
            i = j;
            // if element 2*i +1 is in array and bigger than j, swap
            if ((2 * i + 1) < size && a[2 * i + 1] > a[j])
                  j = 2 * i + 1;
           // if element 2*i +2 is in array than j, swap
            if ((2 * i + 2) < size && a[2 * i + 2] > a[j])
                  j = 2 * i + 2;
            int temp = a[i];
            a[i] = a[j];
            a[j] = temp;
      } while (i != j);
}

void heapify(int a[]) {
      for (int i = a.length - 1; i >= 0; i--)
            //for every element i put in correct place in a heap
            sift(a, a.length, i);
}
```

Heapsort's complexity is O(n log n), because heapify() takes O(n) time, and each of n-1 calls to sift() take O(lg n).

2.2 Graph algorithms

In this chapter the graph algorithms will be discussed. There have been chosen four of them: two shortest path finding algorithms: Floyd-Warshall's and Dijkstra's, and two minimal spanning tree determining algorithms: Kruskal's and Prim's. All the graphs, used in this section as input, are weighted, as far as it is demanded by chosen algorithms. Let's look at the example application.

The graph in the upper section is an input data. (See picture on next page). User can modify weights of edges by inputting values into boxes. Such a possibility is provided, because it is the only graph algorithm, which uses difficult calculations on edge weights. The other three do not have choice of input data implemented, because the function of finding edge with minimal weight is trivial. As in previous section, there is speed control, which is currently set to one second. The element a[5][2] in matrix A is currently highlighted. On the first path of external cycle its value was determined to be 8.

Floyd-Warshall' s algorithm implementation in action:

```
for (int k=1;k<=6;k++){
   for (int i=1;i<=6;i++){
      for (int j=1;j<=6;j++){
        if (a[i][k]+a[k][j]<a[i][j]){
           a[i][j] = a[i][k]+a[k][j];
           b[i][j] =b[i][k];
        }
      }
   }
```

A

	1	2	3	4	5	6
1	0	2	∞	∞	∞	∞
2	∞	0	∞	3	∞	∞
3	∞	1	0	∞	∞	∞
4	∞	∞	∞	0	1	∞
5	6	8	∞	∞	0	5
6	∞	∞	3	7	∞	0

B

	1	2	3	4	5	6
1	1	2	3	4	5	6
2	1	2	3	4	5	6
3	1	2	3	4	5	6
4	1	2	3	4	5	6
5	1	1	3	4	5	6
6	1	2	3	4	5	6

Variables:
i= 5
j= 2
k= 1

Speed: 1 s

STOP

This is the only application, concerning graphs, where code is written in Java. It follows from the fact, that *Floyd-Warshall* algorithm uses for graph distance matrix representation auxiliary data structure – two-dimensional arrays, there is no direct addressation of graph data structure.

Dijkstra's algorithm

Dijkstra's algorithm was invented by Dutch computer scientist *E. Dijkstra* in 1959. It solves the single-source shortest path problem, which means, that it finds path from one vertice to all the other vertices. An input graph is constrained with a condition, that all edge weights must be nonnegative, it must be weigted and directed. The principle of algorithm can be described like that:

1) Put the source vertice in Opened list.
2) For all vertices in Opened list, check all the incident edges and choose the shortest.
3) Add the connected vertice to Opened list, continue from second point.

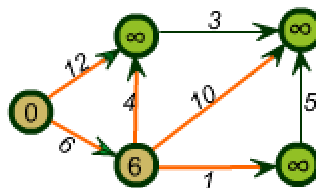
In pseudocode:

```
S= Ø
```

```
Q = V(G)
 while Q ≠ 0
     do u = extract min(Q)
       S=S U {u}
         for all v є Adjacency(u)
          do relax ( u,v,w)
```

Here S is a set of vertices, whose final shortest path weights from the source have already been determined. Q is min-priority queue of vertices by their distance values.

As far as Dijkstra's algorithm always uses lightest edge in V, it uses a greedy strategy. Operation extract min(Q) is a simple search through list Q, which takes linear time. Dijkstra’s complexity is $O(|V|^2 + E) = O(|V|^2)$, where V are graph vertices.




```
S= Ø
Q = V(G)
 while Q ≠ 0
     do u = extract min(Q)
       S=S U {u}
         for all v є Adjacency(u)
          do relax ( u,v,w)
```

Dijkstra's algorithm in action.

Floyd-Warshall's algorithm

The Floyd-Warshall's algorithm finds shortest path from every node to every other node, computing its length. Algorithm uses matrix, where on [i][j] place there is distance from A_i to A_j, if there is an edge between them, or ∞-sign, if there is no edge. In order to determine the order of vertices we need to traverse to get from Ai to A_j, B matrix is maintained. It contains the number of first intermediary vertice. Floyd-Warshall’s complexity is $\Theta(n^3)$, where n is quantity of graph's nodes.

Code:

```
for (int k = 1; k <= 6; k++) {
      for (int i = 1; i <= 6; i++) {
            for (int j = 1; j <= 6; j++) {
                  if (a[i][k] + a[k][j] < a[i][j])
                  a[i][j] = a[i][k] + a[k][j];
                  b[i][j] = b[i][k];
                  }
            }
      }
}
```

Kruskal's algorithm

Algorithm has been first published in 1956 year by american mathematician *Joseph Kruskal*. It finds a minimum spanning tree for a connected weighted graph. Kruskal's algorithm is an example of a greedy algorithm.

Principle of work:

- create a forest F, where each vertex in the graph is a separate trees
- create a set S containing all the edges in the graph
- while S is not empty, remove edge with minimum weight
- if edge connects different trees, add it to forest, otherwise discard

Pseudocode:

```
S= Ø
F= E(G)
 while F ≠ 0
     do u = extract min(F)
       if  u connects different trees
         S=S U {u}
```

Kruskal's algorithm running time is O(E lg V), where E are graph edges, V are vertices.

Prim's algorithm

Algorithm was discovered in 1930 by mathematician *Vojtech Jarnik* and in 1957 independently by computer scientist *Robert C. Prim* and rediscovered by *Edsger Dijkstra* in 1959. It is somtimes called *DJP* algorithm, the *Jarnik* algorithm, or the *Prim-Jarnik* algorithm.

Its principle of work:

- Choose an arbitary node
- repeat until nodes remain:
    - Choose edge from chosen node with minimal weight
    - Add edge to spanning tree

Pseudocode:

```
S= Ø
O =  Ø
F= E(G)
 while F ≠ 0
   do O = O U extract node(F)
     z = relax adjacent(u)
     S=S U {min(z)}
```

O is a set of opened nodes, S are vertices, added to the spanning tree, F is a list of all graph vertices.The performance of Prim's algorithm depends on implementation of min-priority queue F. In different implementation extraction operation can from O(1) up to O(E) time. For instance, if we use *Fibonacci* heap to implement it, Prim's execution time is O(V lg V + E).

2.3 Text algorithm

Text search algorithms are used in text editors to find the pattern in a text or in genetics to search for pattern in DNA sequence. The brute-force algorithm to find a match in a string would be to compare first letter of pattern to letter *i* of test, if they don't match, than move increase *i*, if they match, than compare the second letter with *i*+1 and so on. This algorithm execution takes O(*n***m*) time, where *n* is number of characters in text and *m* is number of characters in a pattern. Of course, with big texts this is not acceptable.

Knuth-Morris-Pratt's algorithm

The algorithm was discovered by *Donald Knuth* and *Vaughan Pratt* and independently by *J. H. Morris* in 1977, but all the three published it together. The Knuth–

Morris–Pratt algorithm is a linear time algorithm to search for occurrences of a pattern within a main text. The algorithm first computes failure-function, or partial-match table, or prefix function, which determines, where we should start to search for match again in case of mismatch. Algorithm's complexity is O(n+m), where n is text length and m is pattern length.

```
//search function. Returns -1, if the pattern is not found, otherwise
retuns the index, of the pattern
int match(String text, String pattern, int[] failure) {
      int j = 0;
      if (text.length() == 0)   //if there is no text
            return -1;

      //untill there is no more text
      for (int i = 0; i < text.length(); i++){
            // if a mismatch occurs, use failure function
            while (j > 0 && pattern.charAt(j) != text.charAt(i)) {
                  j = failure[j - 1];
            }
            // if character matches, check next character
            if (pattern.charAt(j) == text.charAt(i)) {
                  j++;
            }
            // the whole pattern matches - the result is found
            if (j == pattern.length()) {
                  matchPoint = i - pattern.length() + 1;
                  return matchPoint;
            }
      }
      return -1;
}
 int[] computeFailure(String pattern) {
      int j = 0;
      int[] failure = new int[pattern.length()];
      for (int i = 1; i < pattern.length(); i++) {
            while (j > 0 && pattern.charAt(j) != pattern.charAt(i)) {
                  j = failure[j - 1];
            }
            if (pattern.charAt(j) == pattern.charAt(i)) {
                  j++;
            }
            failure[i] = j;
      }
      return failure;
}
```

The prefix function embodies knowledge about how the pattern matches against itself. Is there are repeating substrings, then in case of mismatch the pattern can be shifted to the beginning of previous coinciding substring. Prefix function is determined as follows: "For pattern $P[1 .. m]$, prefix function is a function $\pi : \{1,2, \ldots , m\} \rightarrow \{0,1, \ldots , m - 1\}$, such that $\pi [q] = \max\{k:k<q \text{ and } P_k \,]\, P_q\}$". [1]

On the following picture there is application in action. Text «testsearchstring» and pattern «str» is default algorithm's input. The failure function Java code is not provided in animation due to lack of space.

t e s t s e a r c h s t r i n g

s t r

failure: 0 0 0

```
int match(String text, String pattern, int[] failure) {
int j = 0;
if (text.length() == 0) return -1;
for (int i = 0; i < text.length(); i++) {
while (j>0 && pattern.charAt(j)!= text.charAt(i)) j=failure[j-1];
if (pattern.charAt(j) == text.charAt(i)) j++;
if (j == pattern.length()) return i-pattern.length()+1;
}
return -1;
}
```

Variables:
i=4
j=2

Speed: 3 s

Text:

Pattern:

STOP

2.4 Data structures

This sections contains implementations of two versions of binary tree: AVL-tree and B-tree.

AVL-tree

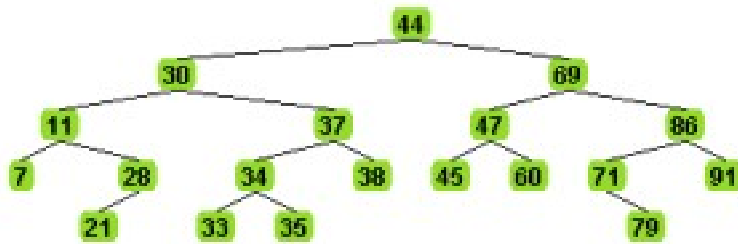
AVL-tree is a self-balancing binary search tree. It is constrained to several conditions:

- The height of two child subtrees differ at most by one.
- Both subtrees are AVL-trees.

AVL-tree is named after its inventors, *G. M. Adelson-Velsky* and *E. M. Landis*. AVL-trees support insertion, removal and search operation, the first two are accompanied with tree balancing, if needed. Operation cost is O(log *n*). *Java* applet implements AVL-tree with possibility to conduct these operations.

The next picture shows how application works. By default the tree is filled with 20 random values. User can input some value in a field in the lower left corner and press «add»

or «remove» button. If user is attempting to add existing value, nothing is done, the same for removing inexistent one. If the field is empty, random value is added.



Inserting 34 .Left rotating 35. Right rotating 33.

34 Add Remove

Code implementation is based on *Weiss's* in [10].

Summary

In course of this work the educational applications, which explain the principles of work of some algorithms, have been created. For this purpose flash technology has been used. The outcome is a webpage.

The developed application can be used along with such classical textbooks on algorithms as a visual training aid, because the short explaination provided along with every algorithm is not enough to understand profoundly complexity analysis.

Algoritmide ja andmestruktuuride illustratiivseid rakendusi

Anna Aljanaki, bakalaureusetöö

Resümee

Selle töö käigus loodi hariduslik programm, mis illustreerib mõnede väljavalitud algoritmide tööd. Kasutati *Flash*-tehnoloogiat. Tulemuseks on saadud veebilehekülg.

Saadud programmi võib kasutada visuaalse abivahendina koos klassikaliste algoritmide õpikutega. Iseseisvaks interaktiivseks õpikuks see ei sobi, sest seletavat teksti ei piisa selleks, et täielikult süveneda algoritmi või andmestruktuuri olemusse.

Bibliography:

[1] *Thomas H. Cormen, Charles E. Leiserson, Ronald L. Rivest, Clifford Stein.* Introduction to Algorithms: second edition. Massachusetts, 2001.

[2] *Sosteric M. Hesemeier S.* A first step towards a theory of learning objects // Online education using learning objects. McGreal Rory. NY, 2004.

[3] Draft Standard for Learning Object Metadata. IEEE Standard 1484.12.1, New York: Institute of Electrical and Electronics Engineers, http://ltsc.ieee.org/wg12/files/LOM_1484_12_1_v1_Final_Draft.pdf, last visited on 15.06.08

[4] *Sedgewick R.* Algorithms in Java. Michigan University Press, 2003.

[5] *Knuth D.* The Art of Computer Programming. Boston, 2007.

[6] *Weiss A. M.* Data Structures and Algorithm Analysis. Floida University Press, 2006.

[7] *Kiho J.* Algoritmid ja andmestruktuurid. Tartu, 2003.

[8] *Kiho J.* Algoritmid ja andmestruktuurid: ülesannete kogu. Tartu, 2005.

[9] Wikipedia http://en.wikipedia.org/wiki/Learning_object, last visited 18.06.08

[10] *Weiss A. M.* Data structures and algorithm analysis in Java. Addison – Wesley, 2007

Appendix

CD with program.

The contents of the CD:
- folder „Program“
- README.txt
- Current document „Illustrative applications on algorithms and data structures.pdf“