

# **Praxisblock 2**

## **Team 7 - Christian Grafe & Marco Fredl - Team 7**

Gliederung

- 0 VORWORT
- I Installation
- II Entwicklungsphase
- III Umsetzung
- IV Fazit
- V Quellen
- VI Quelltext

## **Vorwort**

In diesem Dokument geht es um das Seminar, welches die Verwaltungsinformatiker 04/07 mit Herrn Wunderratsch vom 25.09 - 29.09.06 abgehalten haben. Die Thematik die behandelt wurde - ist hochgradig schwierig.

Es geht um die Lösung für ein Backtracking Problem. Als Beispiel hierfür verwenden wir das Rätsel "Der richtige Weg". Hierbei muss man einen Weg mit Zahlen folgen und die betretenen Zahlen aufsummieren!

Sollte man diesen Algorithmus gefunden haben, soll man das ganze noch in ein grafisches Interface packen. Wir nehmen hierfür die Swing Library in Zusammenhang mit Java und Windows. Als Programme dienen uns Windows 2000, Firefox und Eclipse.

Also dann auf ins Backtracking immer getreu dem Motto: "nach vorn wenn möglich, zurück wenn nötig"

## **Installation**

So los gehts..

1. Punkt Windows:

Festplatte rein und fertig - Ok das war vorinstalliert!

2. Punkt Firefox für die Recherche:

[www.firefox-browser.de/](http://www.firefox-browser.de/) gehen und die Installerdatei runterladen. Dann installieren -> Bester Browser für Recherchen

3. Punkt Eclipse:

[www.eclipse.org](http://www.eclipse.org) gehen und die Installerdatei runterladen. Installieren -> neues Projekt -> Testen

# Entwicklungsphase

## Theorie Backtracking

### Prinzip

Der Algorithmus verfährt nach dem Prinzip

**wenn möglich vorwärts, sonst rückwärts**

Dieses Prinzip lässt sich auch bei vielen anderen Problemen anwenden.

### Voraussetzungen:

- Das Problem lässt sich durch eine Folge von Entscheidungen lösen.
- Zu jedem Zeitpunkt stehen nur endlich viele Entscheidungen zur Wahl.
- Manchmal - aber nicht immer - ist sofort erkennbar, dass eine mögliche Entscheidung falsch ist.

Nicht jede falsche Entscheidung ist sofort als eine solche erkennbar. Es ist z.B. falsch, in eine Sackgasse einzubiegen. Das erkennt man aber erst später, nämlich wenn man merkt, dass es nicht weiter geht.

Man kann nun folgendermaßen eine zur Lösung führende Entscheidungsfolge aufbauen:

### Algorithmus Backtracking:

```
solange Lösung noch nicht gefunden
  falls es jetzt eine noch nicht probierte, mögl. Entscheidung gibt
    falls diese nicht erkennbar falsch ist
      verlängere die Entscheidungsfolge um diese Entscheidung
      {'treffe die Entscheidung'}
    sonst
      entferne die letzte Entscheidung aus der Entscheidungsfolge
      {'nimm die zuletzt getroffene Entscheidung zurück'}
```

Diese Theorie trifft genau zu und so ist es bewiesen, dass wir ein Backtracking Problem haben!

Hier die Zusammenfassung für die Programmierung:

Funktion FindeLoesung (Stufe, Vektor)

1. wiederhole, solange es noch neue Teil-Lösungsschritte gibt:

a) wähle einen neuen Teil-Lösungsschritt;

b) falls Wahl gültig ist:

I) erweitere Vektor um Wahl;

II) falls Vektor vollständig ist, return true; // Lösung gefunden!

sonst:

falls (FindeLoesung(Stufe+1, Vektor)) return true; // Lösung!

sonst mache Wahl rückgängig; // Sackgasse (Backtracking)!

2. Gibt es keinen neuen Teil-Lösungsschritt: return false // Keine Lösung!

Nun wollen wir uns mal an die Umsetzung machen!

## Umsetzung

Die Umsetzung war das Herzstück unserer eigentlichen Arbeit. Wir haben uns für eine Modulare Projektaufteilung entschieden. Einer übernahm das Design und die Benutzerinterfaceschnittstelle und der andere den eigentlichen Algorithmus.

Zuerst wollten wir es farbig und sehr schön designen, als wir jedoch gemerkt haben das die Performance arg leidet sind wir auf schlanke und zweckgebundene Designstruktur umgeschwenkt. Bei beiden Bereichen taten sich große Schwierigkeiten auf:

### Algorithmus:

Das größte aller Probleme war der eigentliche BacktrackingAlgorithmus! Wir haben uns bei Wikipedia ausführlich damit beschäftigt und letztendlich als Grundbaustein einen ähnlichen genommen ( Das Springer Problem ) Allerdings klingt das jetzt leichter als es eigentlich war. Es musste trotzdem noch die ganze Logik angepasst machen. Dabei tat sich ein riesiges Problem auf, welches erst nach 3 Stunden Arbeit gelöst werden konnte. Die Übergabe der eigentlichen Wege sprich Lösungen an das Hauptprogramm.

Schlussendlich haben wir uns für einen Vektor entschieden, in dem wir die Lösungen eingespeichert haben und der Global im Programm verfügbar ist! Somit sind wir Übergabeproblemen und Pointerexceptions aus dem Weg gegangen. Leider ist hier die Performance ziemlich gesunken! Deswegen haben wir den übergebenen Weg in eine neu eingeführte Zwischenvariable (richtiger\_weg) gespeichert und auf diese dann den Max oder Min Weg gesetzt! Somit haben wir eine Kopie erzwungen und sind dem Pointerproblem erneut aus dem Weg gegangen. Ausserdem wurde durch den Vektorverzicht die Performance um ca. 20% gesteigert.

=> 160 Zeilen Quellcode Algorithmus und bei der Summe 400 weniger als 0,5 Sekunden Laufzeit!

### Design:

Bei dem Design haben wir uns, aus mehreren Gründen, für eine einfache Darstellung entschieden. Da wir mit Java programmiert haben, lag es nahe mit der Klasse Swing zu arbeiten. Das Hauptfenster besteht aus einem JFrame, in das ein JPanel eingebunden wurde. In dem JFrame ist auch die Menüleiste wiederzufinden, die aus drei Menüoberpunkten besteht. Der erste Oberpunkt ist "Programm" der einen Unterpunkt "Beenden" hat. Der nächste Oberpunkt "Berechnungen" beinhaltet drei Unterpunkte. Der Unterpunkt "neu berechnen" öffnet ein Popup in das der Wert für die Zielsumme eingegeben werden muss und anschließend erscheint ein weiteres Popup mit der Anzahl der Lösungen. Der nächste Unterpunkt "min. Züge berechnen" zeigt die Anzahl der minimalsten Züge in einem Popup an und zeichnet danach die Tabelle mit dem Lösungsweg. Der letzte Unterpunkt "max. Züge berechnen" zeigt die Anzahl der maximalen Züge in einem Popup an und zeichnet danach die Tabelle mit dem Lösungsweg. Der dritte Oberpunkt "?" beinhaltet einen Unterpunkt "About", dort werden wir nur namentlich erwähnt. Im Panel ist der Inhalt, sprich die Tabellen mit den Lösungswegen zu finden. Die einzelnen Lösungswege, z.B. kürzester oder längster, schreiben jeweils das Panel neu, somit nur immer eine Tabelle zu sehen ist.

### "Hochzeit":

Am Dienstag Nachmittag war es dann soweit - sowohl Algorithmus als auch das Design waren bereit für die Vereinigung.

**Bei der Vereinigung der Codefragmente gab es nur wenige Fehler (wir hatten eigentlich mehr erwartet)**

Nennenswert ist auf jeden Fall, dass der Algorithmus umgeschrieben werden musste, so dass er mehrere Durchläufe beherrscht! Die globalen statischen Variablen müssen auf 0 reinitialisiert werden und der Wert muss neu eingelesen werden. Sonst wird der Code immer wieder mit falschen Daten hochgeladen und die Werte verdoppeln sich.

Ein weiteres Problem war die Repaint Funktion, da diese immer wieder zu falschen Ergebnissen geführt hat! Durch die Ersetzung in den `updateUI()`; Befehl haben wir das Problem gelöst.

Bei der Menüstruktur mussten wir uns ein geeignetes System überlegen um die Benutzerschnittstelle optimal auszunutzen und unnötige Fehler von Anwenderseite her zu unterbinden - Unsere Lösung: Wenig Menüpunkte, Einmalige Neuberechnung zulässig!

## Fazit

Es war eine echt harte Nuss die wir zu knacken hatten, aber durch viel Recherche und Unterstützung von lieben Kommilitonen haben wir sie bewältigt und sind stolz auf uns.

Im Vergleich mit den anderen Teams haben wir gesehen, dass die Entwicklung mit Java und vor allem Eclipse am leichtesten war, allerdings von Haus aus mit einigen Performance Problemen verbunden war. Es ist aus unserer Sicht trotzdem empfehlenswert mit unseren Mitteln zu arbeiten, weil es sehr leicht möglich ist den Code mittels Eclipse zu optimieren und zu bearbeiten.

Vor allem die **leichte** Grafische Entwicklung mittels Swing ist herauszuheben!

## Quellen

<http://de.wikipedia.org/wiki/Backtracking>

<http://www.google.de> (generell für Java Probleme)

<http://www.matheprisma.uni-wuppertal.de/Module/BackTr/index.htm>

## Quelltext

```
import java.awt.*;
import java.awt.event.ActionEvent;
import java.awt.event.ActionListener;
import java.util.Vector;
import javax.swing.table.DefaultTableModel;
import javax.swing.table.TableModel;
import javax.swing.JTable;

import javax.swing.*;

public class DRW_1_0 extends JFrame {

    // Algorithmus

    private final static int m = 5; // Grösse des Feldes mxn

    private static final int n = 11;

    static int spielfeld[][] = { { 0, 8, 6, 86, 23, 51, 8, 25, 18, 51, 64 },
        { 9, 55, 41, 2, 32, 9, 9, 99, 89, 52, 89 },
        { 17, 9, 33, 7, 2, 43, 5, 4, 30, 44, 45 },
        { 32, 12, 5, 10, 51, 7, 32, 56, 54, 19, 54 },
        { 14, 46, 3, 39, 8, 29, 65, 10, 6, 81, 0 } };
}
```

```

// Die Arrays a und b enthalten die vier Himmelsrichtungen als relative
// Werte

private final static int[] a = { +1, 0, -1, 0 }, b = { 0, +1, 0, -1 };

// Legt die gesamte zu erreichende Summe fest
public static int zielsumme = 400;

static int[][] h = new int[m][n]; // Weg in der Matrix

private static int[][] zug_min; // Speicherung für Minimalen Zug
private static int[][] zug_max; // Speicherung für Maximalen Zug

static int anzahl_min_zuege = 0;

static int anzahl_max_zuege = 0;

private static long anzLoesungen = 0;

private static Vector loesungen = new Vector();

private static int zwischensumme = 0;

static int x0 = 0, y0 = 0; // Startposition auf 0, 0 setzen

// Grafik

// Fenster erzeugen
static JFrame f = new JFrame("Backtracking - Der richtige Weg");

static JPanel inhalt = new JPanel(new FlowLayout());

static TableModel model;

static void start() {

    // Größe setzen und position bestimmen
    f.setSize(900, 200);
    f.setLocation(100, 100);

    // Look and Feel für Windows anschalten siehe Funktion
    lookandfeel();

    // Bei Schließen des Fensters -> beenden des Programms
    f.setDefaultCloseOperation(JFrame.EXIT_ON_CLOSE);

    /*
    * // Bild in Programm einfügen String filename = "logo.jpg "; Image
    * image = Toolkit.getDefaultToolkit().getImage(filename);
    * f.setIconImage(image);
    */

    // Layout Manager auf BorderLayout-Setzen
    f.getContentPane().setLayout(new BorderLayout());

    // Tabelle zum Start anzeigen lassen
    print();
}

```

```

// Funktionsaufruf zum erstellen des Menüs
menue();

/*
 * // Scrollbar erstellen und einbinden JScrollBar sb = new
 * JScrollBar(JScrollBar.VERTICAL, 0, 255, 0, 255);
 * f.getContentPane().add(sb, BorderLayout.EAST);
 */

// Fenster anzeigen
f.setVisible(true);
}

```

```

static void menue() {

```

```

    // Menüleiste Erstellen
    JMenuBar jmb = new JMenuBar();

```

```

    // Menüpunkte erstellen und Mnemonics einfügen
    JMenu jm1 = new JMenu("Programm");
    jm1.setMnemonic('P');
    JMenu jm2 = new JMenu("Berechnungen");
    jm2.setMnemonic('B');
    JMenu jm3 = new JMenu("?");
    jm2.setMnemonic('?');

```

```

// Menüunterpunkte erstellen
JMenuItem jm1i2 = new JMenuItem("Beenden", 'B');
JMenuItem jm2i1 = new JMenuItem("neu berechnen", 'n');
JMenuItem jm2i2 = new JMenuItem("min. Züge berechnen", 'm');
JMenuItem jm2i3 = new JMenuItem("max. Züge berechnen", 'x');
JMenuItem jm3i1 = new JMenuItem("About", 'a');

```

```

// Menüstruktur erstellen

```

```

// Oberpunkte
jmb.add(jm1);
jmb.add(jm2);
jmb.add(jm3);

```

```

// Unterpunkte zuordnen
jm1.add(jm1i2);
jm2.add(jm2i1);
jm2.add(jm2i2);
jm2.add(jm2i3);
jm3.add(jm3i1);

```

```

// Menü einbinden
f.getContentPane().add(jmb, BorderLayout.NORTH);

```

```

// Funktion des Beenden Buttons
jm1i2.addActionListener(new ActionListener() {
    public void actionPerformed(ActionEvent e) {
        System.exit(0);
    }
});

```

```

// Funktion des Button "Berechne"
jm2i1.addActionListener(new ActionListener() {

```

```

public void actionPerformed(ActionEvent e) {
    // Reinitialisierung der Variablen
    anzahl_min_zuege = 0;
    anzahl_max_zuege = 0;
    anzLoesungen = 0;
    zwischensumme = 0;
    x0 = 0;
    y0 = 0;

    // Einlesen des zu erreichenden Wertes
    String s = JOptionPane.showInputDialog("Bitte Wert eingeben");
    if (s.equals("")) {
        s = "0";
    }
    zielsumme = Integer.parseInt(s);

    anz_loesung();
    inhalt.updateUI();
}
});

// Funktion des Button "Anzeigen der minimalen Züge"
jm2i2.addActionListener(new ActionListener() {
    public void actionPerformed(ActionEvent e) {
        print_min_zuege();
        inhalt.updateUI();
    }
});

// Funktion des Button "Anzeigen der maximalen Züge"
jm2i3.addActionListener(new ActionListener() {
    public void actionPerformed(ActionEvent e) {
        print_max_zuege();
        inhalt.updateUI();
    }
});

//Funktion des Button "Anzeigen der minimalen Züge"
jm3i1.addActionListener(new ActionListener() {
    public void actionPerformed(ActionEvent e) {
        String about = "Programmiert von und durch:\n\nChristian Grafe und Marco
Fredl\n\n\nCopyright CLX Design 2006 ©";
        JOptionPane.showMessageDialog(f, about);
    }
});
}

// Popup der Anzahl der Lösungen

static void anz_loesung() {

    h[x0][y0] = 1; // Erstes Feld von Anfang an als markiert setzen
    DRW(2, x0, y0); // rekursives Backtracking starten

    String s = "Anzahl der Lösungen: " + anzLoesungen;

```

```

OptionPane.showMessageDialog(f, s);
// JTextArea anzloesung = new JTextArea("Anzahl der Lösungen:
// "+anzLoesungen +"\n");

}

// Erstellen der Tabelle mit den Anfangswerten

static void print() {

    // Integer Array für Tabelle in String Array umwandeln

    String str[][] = new String[5][11];
    for (int j = 0; j < 5; j++) {

        for (int i = 0; i < 11; i++) {
            str[j][i] = Integer.toString(spielfeld[j][i]);
        }
    }

    String[] columnNames = { "", "", "", "", "", "", "", "", "", "", "" };

    // Erstellen der Tabelle

    model = new DefaultTableModel(str, columnNames);

    JTable table = new JTable(model) {
        public boolean isCellEditable(int x, int y) {
            return false;
        }
    };

    inhalt.add(table);

    f.getContentPane().add(BorderLayout.CENTER, inhalt);

}

// Popup mit Anzahl der minimalsten Zügen und Anzeige des Weges

static void print_min_zuege() {

    // Integer Array für Tabelle in String Array umwandeln

    String str[][] = new String[5][11];
    for (int j = 0; j < 5; j++) {

        for (int i = 0; i < 11; i++) {
            str[j][i] = Integer.toString(zug_min[j][i]) + " (" + Integer.toString(spielfeld[j][i]) + ")";
            if(str[j][i].startsWith("0"))
                str[j][i]="";
        }
    }

    String[] columnNames = { "", "", "", "", "", "", "", "", "", "", "" };

    // Erstellen der Tabelle

    model = new DefaultTableModel(str, columnNames);

```

```

JTable table_min = new JTable(model) {
    public boolean isCellEditable(int x, int y) {
        return false;
    }
};

inhalt.removeAll();
inhalt.add(table_min);

String s = "Anzahl der minimalsten Züge: " + anzahl_min_zuege;
JOptionPane.showMessageDialog(f, s);

f.getContentPane().add(BorderLayout.CENTER, inhalt);
}

// Popup mit Anzahl der maximalsten Zügen und Anzeige des Weges
static void print_max_zuege() {

    // Integer Array für Tabelle in String Array umwandeln

    String str[][] = new String[5][11];
    for (int j = 0; j < 5; j++) {

        for (int i = 0; i < 11; i++) {
            str[j][i] = Integer.toString(zug_max[j][i]) + " (" + Integer.toString(spielfeld[j][i]) + ")";
            if(str[j][i].startsWith("0"))
                str[j][i]="";
        }
    }

    String[] columnNames = { "", "", "", "", "", "", "", "", "", "", "" };

    // Erstellen der Tabelle

    model = new DefaultTableModel(str, columnNames);

    JTable table_max = new JTable(model) {
        public boolean isCellEditable(int x, int y) {
            return false;
        }
    };
    inhalt.removeAll();
    inhalt.add(table_max);

    String s = "Anzahl der maximalsten Züge: " + anzahl_max_zuege;
    JOptionPane.showMessageDialog(f, s);

    f.getContentPane().add(BorderLayout.CENTER, inhalt);
}

// Accelerator Funktion für die Strg Befehle
private static void setCtrlAccelerator(JMenuItem mi, char acc) {
    KeyStroke ks = KeyStroke.getKeyStroke(acc, Event.CTRL_MASK);
    mi.setAccelerator(ks);
}

static void lookandfeel() {

```

```

// WINDOWS LOOK AND FEEL ANSCHALTEN
try {
    UIManager
        .setLookAndFeel("com.sun.java.swing.plaf.windows.WindowsLookAndFeel");
} catch (ClassNotFoundException e) {
    e.printStackTrace();
} catch (InstantiationException e) {
    e.printStackTrace();
} catch (IllegalAccessException e) {
    e.printStackTrace();
} catch (UnsupportedLookAndFeelException e) {
    e.printStackTrace();
}
}

// Algorithmus
private static void DRW(int zugNr, int x, int y) {
    // = Der richtige Weg

    for (int k = 0; k < 4; k++) {

        int u = x + a[k], v = y + b[k];

        // Zug prüfen: gültig, wenn Position im Spielfeld und Feld noch
        // nicht besucht
        if (u >= 0 && u < m && v >= 0 && v < n && h[u][v] == 0) { // h[u][v]==
            // 0
            // meint
            // unbesucht

            h[u][v] = zugNr; // der eigentliche Zug! Nummer wird in Array
            // speichern
            zwischensumme = zwischensumme + spielfeld[u][v];

            if (zwischensumme <= zielsumme && (!(u == m - 1 && v == n - 1)))
                DRW(zugNr + 1, u, v); // Rekursion!

            if (zwischensumme == zielsumme && (u == (m - 1)
                && (v == (n - 1)))) {
                loesunggefunden(h, zugNr);
                h[u][v] = 0; // Zug rückgängig machen = Backtracking!
                continue;
            } else {
                h[u][v] = 0; // Zug rückgängig machen = Backtracking!
                zwischensumme = zwischensumme - spielfeld[u][v];
                continue;
            }
        }
    }
}

private static void loesunggefunden(int[][] weg, int anzahl_zuege) {
    anzLoesungen++; // Lösungen zählen

    if (anzahl_min_zuege == 0)
        anzahl_min_zuege = anzahl_zuege + 1;
}

```

```

int[][] richtiger_weg = new int[m][n];

for (int zeile = 0; zeile < weg.length; zeile++) {
    for (int spalte = 0; spalte < weg[zeile].length; spalte++) {
        richtiger_weg[zeile][spalte] = weg[zeile][spalte];
    }
}
loesungen.addElement(richtiger_weg);

if (anzahl_zuege > anzahl_max_zuege) {
    zug_max = richtiger_weg;
    anzahl_max_zuege = anzahl_zuege;
} else if (anzahl_zuege < anzahl_min_zuege) {
    zug_min = richtiger_weg;
    anzahl_min_zuege = anzahl_zuege;
}
}

public static void main(String[] args) {

    for (int i = 0; i < m; i++)
        for (int j = 0; j < n; j++)
            h[i][j] = 0; // Flags auf 0 setzen

    start();
}
}

```