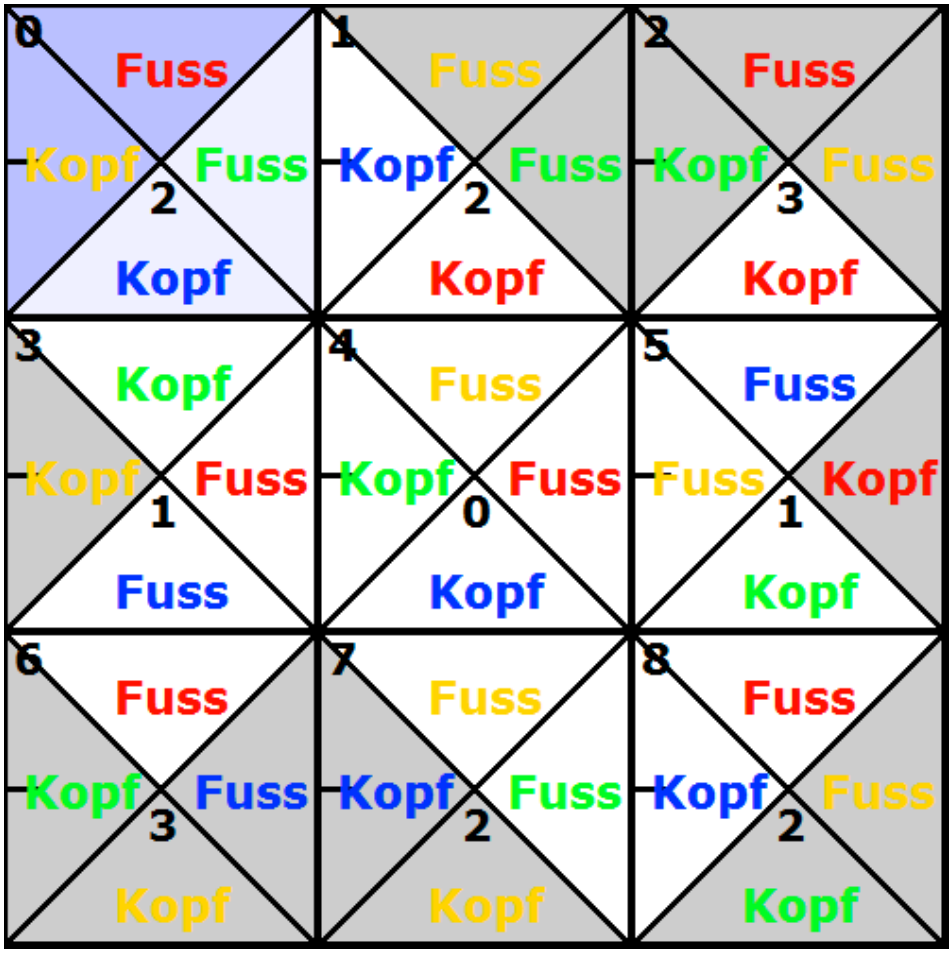
KOMBINATORISCHE PROBLEME DER ANORDNUNG VON SPIELKARTEN UND IHRE REALISIERUNG IN DELPHI



Facharbeit in Informatik

Benedikt Vogler

Herbst 2011 - Frühjahr 2012

Inhaltsverzeichnis

VORWORT

Eine Wahl für das Facharbeitsthema, die sich für mich stellte, war die Lösung von kombinatorischen Problemen mittels Algorithmen. Aus privatem Interesse fing ich im Sommer 2011 das Kartenspiel in JavaScript an zu programmieren. Das Kartenspiel ließ sich gut mit den Fragen und Problemen des (schon in den Anfangsphasen abgebrochenen) Projekts vereinen, so dass die Entwicklung wieder aufgenommen wurde. Neben dem Kartenspiel erwog ich noch ein anderes kombinatorisches Holzspiel umzusetzen. Im Nachhinein hätte ich lieber das Holzspiel genommen, da es weniger komplex ist. Mit dem Abschluss der Facharbeit beende ich eine fast halbjährige intensive Beschäftigung mit dem Thema.

EINLEITUNG

Die Umsetzung von Welten, Medien oder Problemen in digitale Formen ist ein großer Teil der Informatik. Videospiele, Animation, Robotik und Kombinatorik sind einige Beispiele für solche Umsetzungen. Für die Arbeit wurde ein Programm in Delphi entwickelt, welches ein kombinatorisches Problem in die Kernelemente zerlegt, simuliert und versucht zu lösen. Es wir der Aufbau der wichtigsten Komponenten des Programms gezeigt, da der Umfang des Programms den Rahmen der Facharbeit übersteigt.

PROBLEMSTELLUNG

Das Spiel

Das Kartenspiel ist ein Knobelspiel mit neun quadratischen Karten. Bekannt ist es unter dem Namen „das verflixte ... -Spiel“ oder „die verflixten ...“. Inhalt des Namens können z.B. Hexe, Spechte, Hunde, Fische, Piraten oder Schildkröten sein. Ein Alternativname ist „Artus Puzzle“ oder bloß „Legespiel“. Dem Autor liegt ein original Spiel („Das verflixte Hexen-Spiel“) aus dem Jahre 1979 von Artus Games vor.

Fotos von einem originalen Spiel der Firma „Artus Games“

Spielregeln

Der Spieler hat neun Karten, die in einem 3x3-Feld beliebig angeordnet werden. Jede Karte hat an jeder Kante eine Grafik, die entweder ein Vorder- oder Hinterteil in einer von vier verschiedenen Farben zeigt. Ziel des Spiels ist es nun, die Karten so zu drehen und zu tauschen, dass jede Karte im Spielfeld einen passenden Partner hat, d.h. es müssen je Vorder- und Hinterteil in der gleichen Farbe aneinander gelegt werden.

Das Knobelproblem

Es gibt, je nachdem, welche Karten in dem Spiel enthalten sind, unterschiedlich viele Lösungen. Bei dem hier verwendeten Spiel gibt es insgesamt acht Lösungen, wobei sechs davon durch Symmetrie entstehen[1]. Insgesamt gibt es bei neun Karten und einem Feld von 3x3 Größe 95.126.814.720 Legemöglichkeiten. Diese Zahl ergibt sich aus folgender Formel:

$$f(n)=n!\cdot 4^n;\ n\in\mathbb{Z}_0^+$$

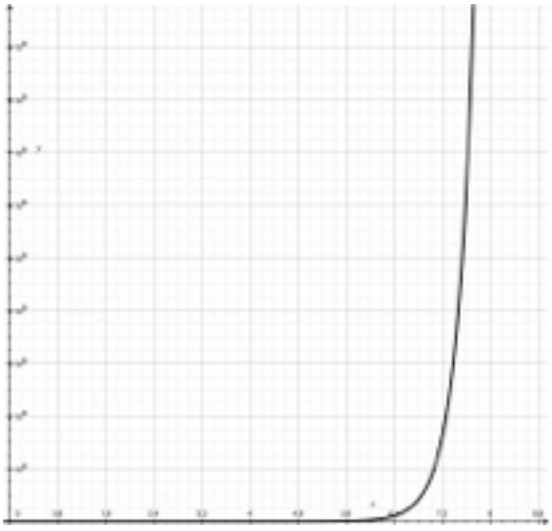
f: Anzahl Möglichkeiten; n: Anzahl der Karten

Für f(9) erhält man:

$$f(9)=9!\cdot 4^9=95.126.814.720$$

Herleitung

Die erste Karte kann auf neun mögliche Feldern platziert werden. Die zweite auf acht Felder, die dritte auf sieben usw. Daraus ergeben sich 9![2] Legemöglichkeiten. Jede Karte kann wiederum in vier verschiedenen Orientierungen liegen, durch drehen der Karten ergeben sich also 4^9 Möglichkeiten. Allgemein kann 9 auch durch *n* ersetzt werden, um die Anzahl der Kombinationen für eine beliebige Anzahl Karten zu ermitteln.



Die Funktion steigt so stark an (Durchschnittliche Steigung beträgt 2.929.777.777,8), dass der Graph, hier bei einer Skalierung der Y-Achse in 10^8er Schritten (!), parallel verlaufend zur X-Achse erscheint und dann parabelähnlich ansteigt.

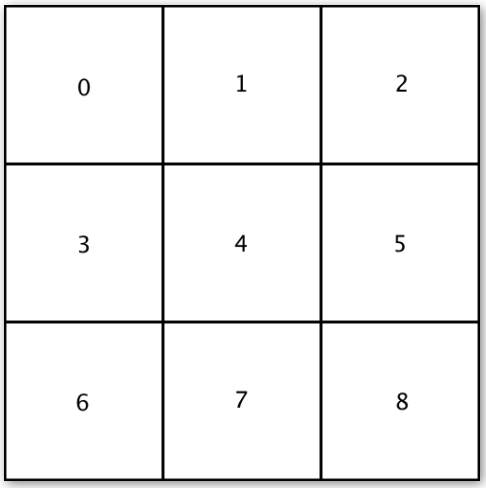
---

[1] Jede Lösung kann drei mal gedreht werden, da das Feld quadratisch ist.

[2] *9! entspricht der Zahl 1*2*3*4*5*6*7*8*9, was man Fakultät nennt*

Zum besseren Verständnis

Im Folgenden wird zwischen den Begriffen „Karte“ und „Position“ unterschieden. Es gibt neun verschiedene Karten, die an neun Positionen (0-8) liegen. Die Karten hingegen können frei auf die Positionen verteilt werden. Damit liegt Karte 0 nicht zwangsläufig in der oberen linken Ecke, allerdings befindet sich Position 0 dort.



Die Positionsnummerierung, bei 0 startend.

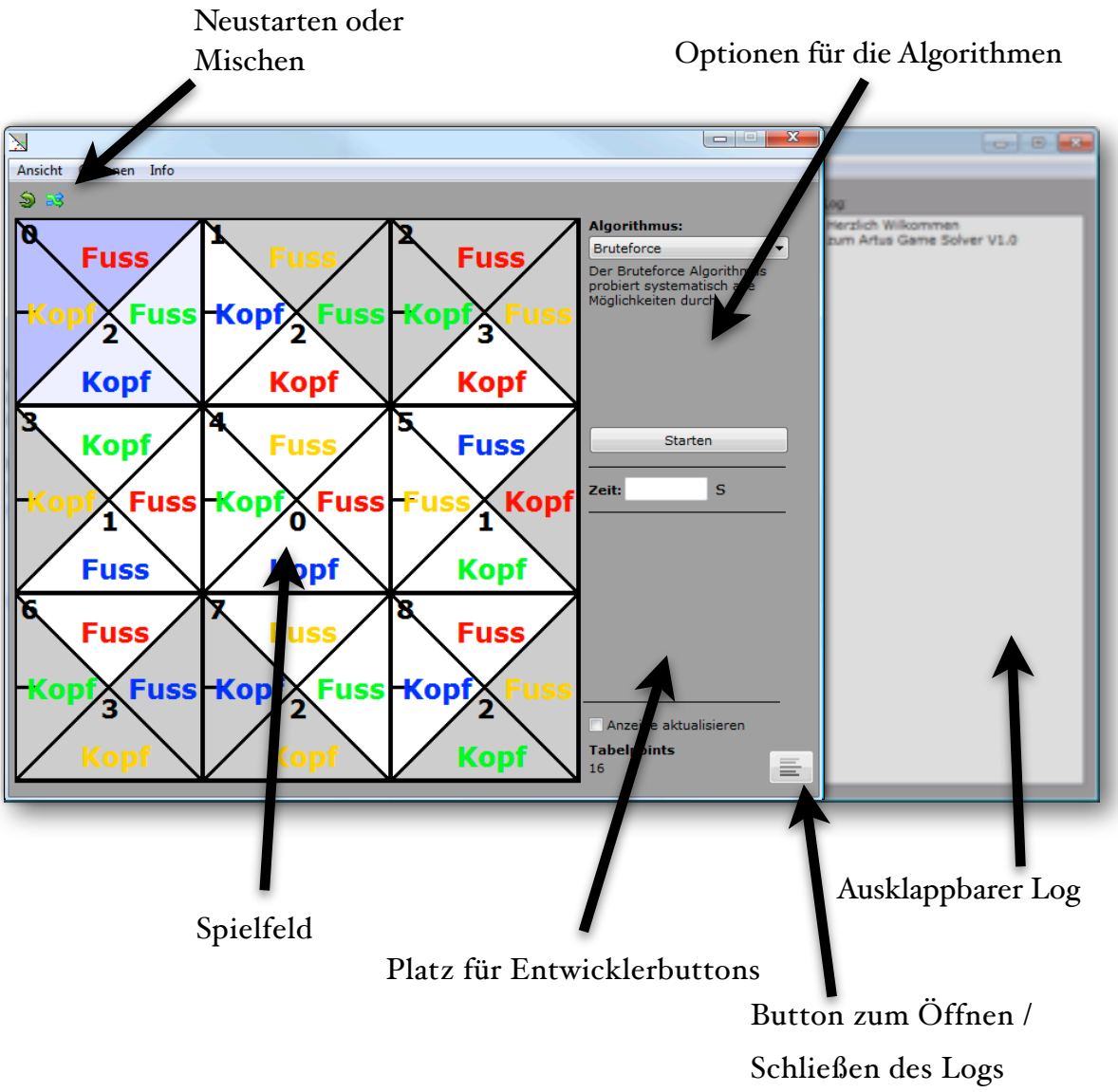
Zur besseren Identifizierung von Code erscheinen Codezitate kleiner und in einer anderen Schriftart. Beispiel: `a := 0`. Längere Codezitate erscheinen eingerahmt.

Funktionsaufrufe werden kursiv und ohne Parameter dargestellt.
Beispiel für einen Aufruf einer Funktion: *tablepoints()*.

Des weiteren bezeichnet das Wort „Tablepoints“ (entsprechend der später gezeigten Funktion *tablepoints()*), die Summe der „Passzahlen“. Eine Passzahl ist die Summe der Kanten einer Karte, die mit den Nachbarkarten passend liegen. Ist die Summe 36, passen alle Kanten und das Feld ist gelöst.

DAS PROGRAMM

Das GUI



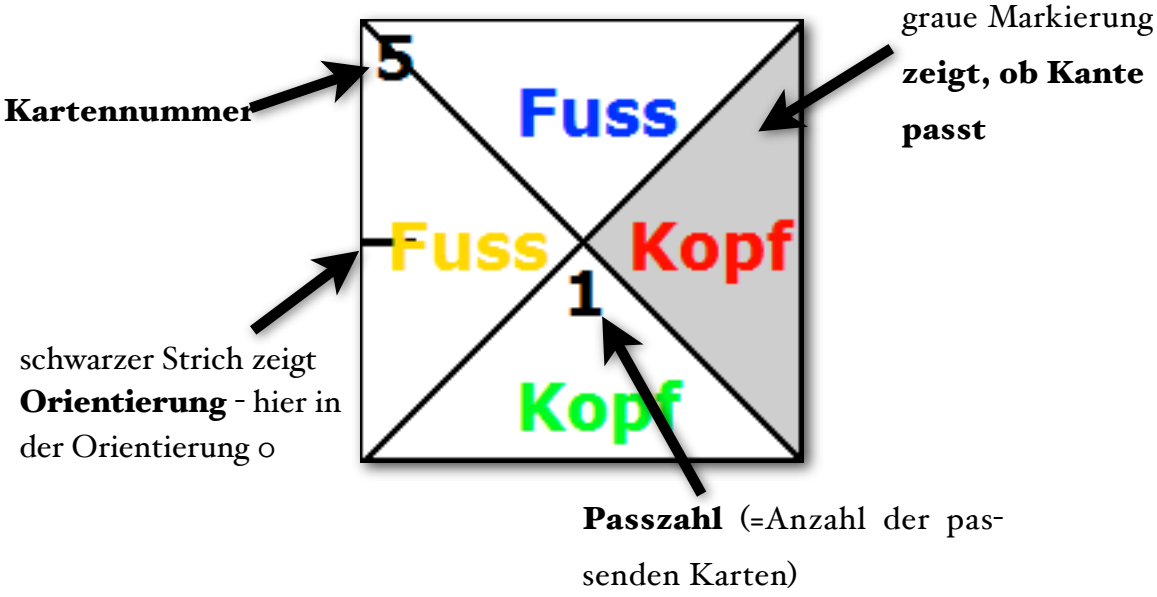
Screenshot des Hauptfensters

Zentraler Inhalt des Formulars ist die Darstellung des Spielfeldes (Zeichnen mit Canvas auf Image). Die Menüleiste bietet zur rechten Spalte weitere Optionen an, wie Programmoptionen, Darstellungsoptionen und Informationsfenster.

Damit die Schaltflächen vom Design besser zum dem jeweilige Betriebsystem passen, wird das XP-Manifest benutzt.

Wählt der Benutzer einen Algorithmus aus der Liste, so wechselt auch die Beschreibung und die Einstellungen darunter werden angepasst.

Das Spielfeld zeigt eine stilisierte Form der Karten. Passt eine Seite, wird der Hintergrund grau markiert. Mit einem Klick auf eine Karte lässt eine sich markieren und mit einem weiteren Klick drehen oder per Drag & Drop mit anderen Karten tauschen.



Darstellung einer Karte auf dem Spielfeld

Programmarchitektur

Das Programm besteht aus vier Klassen, wovon jede eine eigene Unit hat. Zu den vier Klassen existiert noch die Unit mUeber, die für das Informationsfenster zuständig ist.

Das Formular bildet die oberste Ebene. Ihr ist das Spielfeld untergeordnet, welches ein geerbtes Objekt vom Typ *TImage* ist. Über den Konstruktor wird das Log und das Label für die Tablepoints in eine Kennt-Beziehung mit dem Spielfeld gesetzt:

```
constructor
TSpielfeld.Create(AOwner:TComponent;KTablepointslabel:TLabel;KLog:TList
Box);
```

aus mSpielfeld.pas

Jetzt hat das Spielfeld auch Zugriff auf diese Objekte und kann z.B. neue Logeinträge hinzufügen. Über die Prozedur *Restart()* werden nun im constructor die Karten erstellt und anschließend die Passzahlen aktualisiert.

```
procedure TSpielfeld.Restart();
//Neustart des Feldes
var i:integer;
begin
  for i := 0 to 8 do FPositionen[i] := TKarte.Create(i);
  for i := 0 to 8 do refreshPasszahl(i);
end;
```

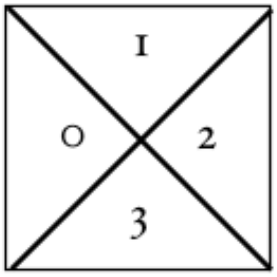
aus mSpielfeld.pas

Der Konstruktor der Karten enthält die Karteninformationen und ruft wiederum den Konstruktor der Kanten auf:

```
constructor TKarte.Create(number:integer);
begin
    FNummer := number;
    FOrientierung := 0;
    //kantenobjekte erzeugen
    FKante[0] := TKante.Create();
    FKante[1] := TKante.Create();
    FKante[2] := TKante.Create();
    FKante[3] := TKante.Create();
    //je nachdem welche Karte erzeugt wird
    case number of
       0:begin
            FKante[0].setFarbe(0);
            FKante[0].setForm(1);
            FKante[1].setFarbe(1);
            FKante[1].setForm(0);
            FKante[2].setFarbe(3);
            FKante[2].setForm(0);
            FKante[3].setFarbe(2);
            FKante[3].setForm(1);
       end;
[...]
```

aus mKarte.pas

Um Kanten identifizieren zu können werden diese nummeriert. Die Positionen sind von 0 bis 8 durchnummeriert, die Orientierungen der Karten im Uhrzeigersinn von 0-3, links mit 0 startend.

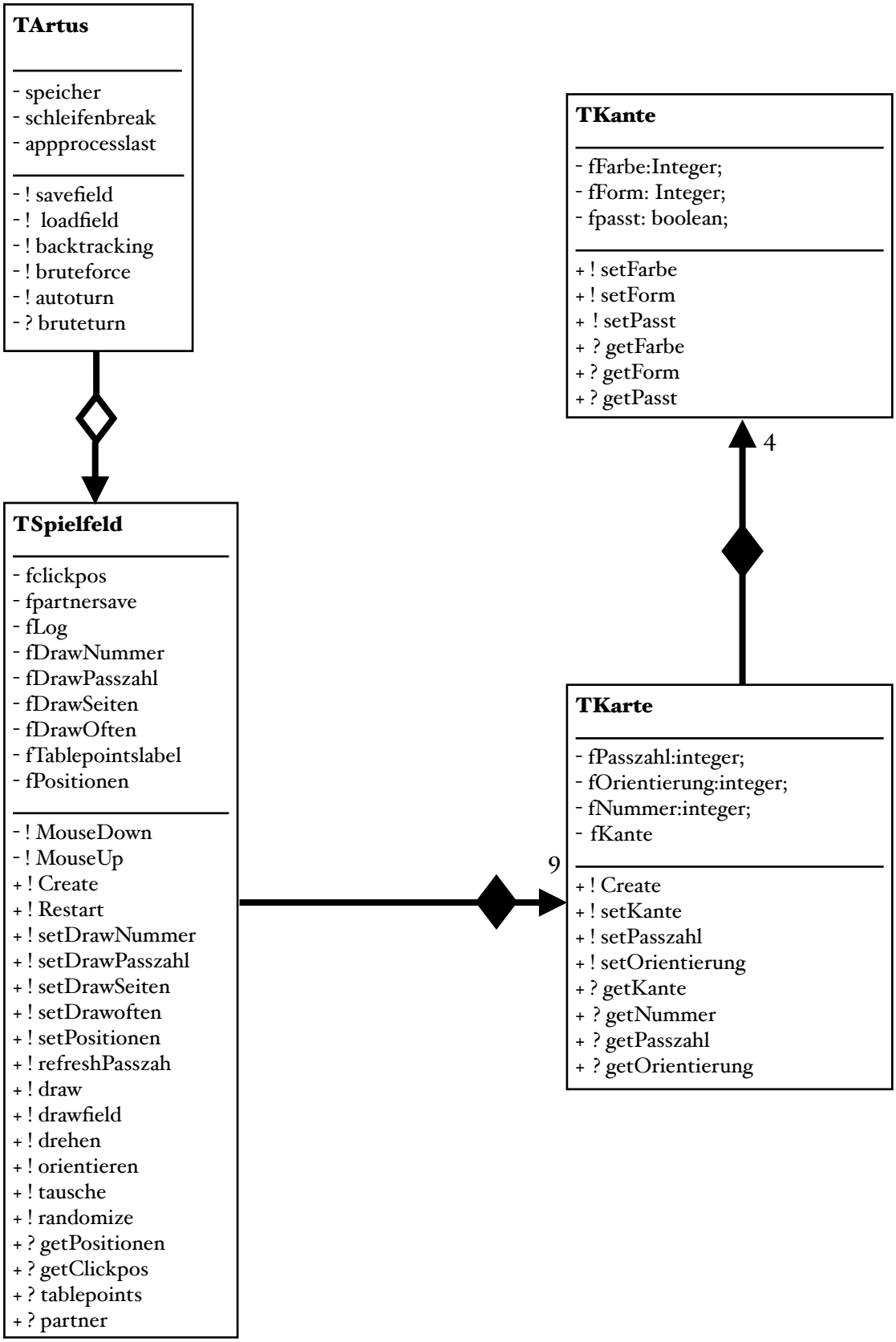


die Lagennummerierug

Arbeitet man mit diesem Nummierungssystem bietet sich an, Zahlen zu addieren und subtrahieren, um gewünschte Kanten anzusprechen. Allerdings sind Zahlen außerhalb des Zahlenbereiches von 0 bis 3nicht kompatibel Abhilfe bringt Modulo 4. Kante 4 entspricht 0 (da 4 mod 4 = 0), Kante 5 entspricht 1 (da 5 mod 4 = 1) usw.. Das gleiche gilt für die Absolutbeträge der negativen Zahlen [Abs(-5 mod 4)=1].

Klassendiagramm in UML

Aus Gründen der Übersichtlichkeit sind Parameter, Datentypen usw. nicht angegeben. Sie können im Programmcode eingesehen werden.



Wichtige Prozeduren und Funktionen

Exemplarisch werden nun die wichtigsten Prozeduren und Funktionen vorgestellt.

PARTNER

```
function TSpielfeld.partner(pos:integer):TPartnerarray;
//liefert Array mit allen umliegenden Karten,
[partner][0]=orientierung;[partner][1]=position
var partners:TPartnerarray;
    curside:integer;
begin
   if (Fpartnersave[pos][0][0] <> -1)//wenn im Speicher, dann ausgeben
       then Result := Fpartnersave[pos]
       else begin
               //Partnerkarten ermitteln
               setLength(partners,4);
               curside := 0;

               if pos mod 3 <> 0 //nur die am linken Rand nicht
                  then begin
                          partners[curside][0]:= 0;//Karte links
                          partners[curside][1]:= pos-1;//Position
                          curside := curside + 1;
                       end;
               if pos >= 3 //alle mittleren und unteren Karten
                  then begin
                          partners[curside][0] := 1;//Karte links
                          partners[curside][1] := pos-3;//Position
                          curside := curside + 1;
                       end;
               if pos mod 3 - 2 <> 0 //nur die am rechten Rand nicht
                  then begin
                          partners[curside][0] := 2;//Karte rechts
                          partners[curside][1] := pos+1;//Position
                          curside:=curside + 1;
                       end;
               if pos <= 5//nur die oberen nicht
                  then begin
                          partners[curside][0] := 3;//Karte unten
                          partners[curside][1] := pos+3;//Position
                          curside := curside + 1;
                        end;
               setLength(partners, curside);//array kürzen, wenn es
weniger Partner gibt
               Fpartnersave[pos] := partners;//speichern
               Result := partners;
            end;
end;
```

aus mSpielfeld.pas

Sollen die angrenzenden Positionen einer Position ermittelt werden, wird die Funktion *partner()* aufgerufen.

Die Funktion *partner()* liefert als Rückgabewert TPartnerarray, ein dynamisches Array[3] mit einem weiteren Array als Inhalt (mehrdimensionales Array). Die erste Dimension bezeichnet die Partner und die zweite Dimension die Partnerinformationen. Karte 1 hat z.B. drei Partner: Einen an der rechten, linken und unteren Kante. Das ergibt für partner(1) den Rückgabewert:

`Partner(1):TPartnerarray`	`[0]`	`[1]`	`[2]`
`[0] (Kante)`	`0 (=links)`	`2 (=rechts)`	`3 (=unten)`
`[1] (Position)`	`0`	`2`	`4`

In FPartnersave werden diese Informationen für jede Position gespeichert, da diese konstant sind. FPartnersave ist ein dreidimensionales Array, was als erste Dimmension die neun Positionen enthält. Wird jetzt partner(pos) aufgerufen und `Fpartnersave[pos][0][0] <> -1` ist wahr, dann wird der dort gespeicherte Wert zurück gegeben (vgl. Tabelle), denn im Konstruktor des Spielfeldes, wird zunächst jede Position im Array ([0-8][0][0]) mit -1 initialisiert. Die Speicherung der Konstanten beschleunigt das Programm. Bei jeder Kantenüberprüfung[4] müssten die Informationen über die zu prüfenden Kanten neu ermittelt werden, dies geschieht aber dadurch insgesamt nur neun mal.

---

[3] ein dynamisches Array ist ein Array, was keine vorher festgelegte Länge hat. Während der Laufzeit kann die Länge mit *setlength(<array oder string>,<neue Länge>:integer)* je nach Bedarf verändert werden.

[4] vergleich zweier Kanten in *refreshPasszahl()*

REFRESHPASSZAHL

```
procedure TSpielfeld.refreshPasszahl(pos:integer);
var partnerlist : TPartnerarray;
    passzahl, i :integer;
    kante1, kante2:TKante;
begin
   //funktionsaufruf zwischenspeichern
   partnerlist := partner(pos);
   passzahl := 4;
   for i := 0 to 3 do
Fpositionen[pos].getKante(i).setPasst(true);//alle erst mal auf true
   for i := 0 to length(partnerlist) - 1 do
       begin
           //beide kanten zwischenspeichern
           kante1 := Fpositionen[pos].getKante(partnerlist[i][0]);
           kante2 :=
Fpositionen[partnerlist[i][1]].getKante((partnerlist[i][0]+2) mod 4);

           if (kante1.getFarbe <> kante2.getFarbe) or (kante1.getForm =
kante2.getForm)
              then begin
                      //passt nicht? dann passzahl verringern und kante
so eintragen
                      passzahl := passzahl - 1;
                      kante1.setPasst(false);
                   end;
       end;
   Fpositionen[pos].setPasszahl(passzahl);//speichern
end;
```

aus mSpielfeld.pas

Die Prozedur ermittelt die Passzahl, also die Anzahl der passenden Seiten, einer Position.

Als erstes wird das Partnerarray über `partner(pos)` ermittelt und in der Variable `partnerlist` gespeichert, damit nicht bei jedem späteren Schleifendurchlauf die Prozedur *partner()* erneut aufgerufen werden muss. Danach wird das Feld `FPasst` einer jeden Kante über eine for-Schleife auf true gesetzt und danach auf nicht passende Seiten, mithilfe von *partner*() geprüft. Kanten, die am Spielfeldrand liegen, passen damit automatisch. Die Überprüfung der Kanten geschieht in einer for-Schleife von 0 bis `length(partnerlist)`, also abhängig von der Anzahl der Partner. Die nun zu vergleichenden Kanten werden in den Variablen kante1 und kante2 gespeichert.

In der if-Abfrage wird geprüft, ob `kante1` und `kante2` nicht passen. Passen diese nicht, wird `passzahl` verringert und das Feld „FPasst“ der Kanten über den Setter `setPasst()` auf `false` gesetzt.

Zuletzt wird die Passzahl gespeichert.

D R E H E N

```
procedure TSpielfeld.drehen(pos:integer);
//dreht Karte an der Position pos im Uhrzeigersinn
var tempSide:TKante;
begin
   tempSide := Fpositionen[pos].getKante(0);//temporärer Speicher
   Fpositionen[pos].setKante(0,Fpositionen[pos].getkante(3));
   Fpositionen[pos].setKante(3,Fpositionen[pos].getkante(2));
   Fpositionen[pos].setKante(2,Fpositionen[pos].getkante(1));
   Fpositionen[pos].setKante(1,tempSide);

   //orientierung aktualisieren
   Fpositionen[pos].setOrientierung((Fpositionen[pos].getOrientierung +
1) mod 4);
   refreshPasszahl(pos);
   if FDrawoften then draw(pos);
end;
```

aus mSpielfeld.pas

Um eine Karte drehen zu können, müssen die Seiten miteinander getauscht werden. Dies geschieht wie beim Tausch zweier Variablen, nur hier mit vier. Man benötigt eine Variable als temporären Speicher.

`tempSide` ist der Zwischenspeicher der nun mit Kante 0 gefüllt wird. Alle anderen kanten werden nun über setKante, mit den Daten der vorherigen gefüllt.

Anschließend wird die aktuelle Orientierung erhöht und über `mod 4` in das System 0-3 gebracht:

```
Fpositionen[pos].setOrientierung((Fpositionen[pos].getOrientierung + 1) mod 4);
```

Die Karte wurde verändert, deswegen muss die Karte aktualisiert werden. Falls Zeichnen aktiviert ist, wird sie neu gezeichnet.

ORIENTIEREN

```
procedure TSpielfeld.orientieren(pos:integer;orient:integer);
var old:array[0..3] of TKante;
    i:integer;
begin
  //Karte, unabhängig der Orientierung, speichern
  for i := 0 to 3 do
      old[i] := Fpositionen[pos].getKante((i +
Fpositionen[pos].getOrientierung) mod 4);
  for i := 0 to 3
      do Fpositionen[pos].setKante(i,old[(i-orient+4) mod
4]);//wiederherstellen
  //Position aktualisieren
  Fpositionen[pos].setOrientierung(orient);
  refreshPasszahl(pos);
  if FDrawoften then draw(pos)
end;
```

aus mSpielfeld.pas

Manchmal müssen die Karten nicht nur im Uhrzeigersinn gedreht werden, sondern direkt in eine bestimmte Orientierung gebracht werden. Es würde sich hier anbieten, die Karten so oft zu drehen bis sie richtig liege, dies ist aber zu ressourcenaufwendig. Ähnlich wie beim Drehen, muss eine Kante zuvor gespeichert werden. Hier werden direkt alle Kanten gespeichert, dass diese im Array `old[0..3]` von 0 bis 3 sortiert sind. Interessant ist die Berechnung des Inhaltes des Arrays.

`Fpositionen[pos].getKante((i + Fpositionen[pos].getOrientierung) mod 4);` gibt die Kante `i` aus, also wie in der 0-Orientierung. Geladen wird die Kante `i` mit `(i-orient+4) mod 4`, bei gewünschter neuer Orientierung `orient`. Nun wird die Orientierung gesetzt und die Karte aktualisiert und wenn gewünscht gezeichnet.

D R A W

```
procedure TSpielfeld.draw(pos:integer);
  function formtoword(form:integer):string;
  //wandelt die Zahl der Form in Wort um
  begin
    if form = 1
       then Result := 'Kopf'
       else Result := 'Fuss';
  end;
  function colortohex(color:integer):TColor;
  //farbe in BGR (umgedrehtes RGB-Format) umwandeln
  begin
     case color of
          0: Result := $00DAFF;
          1: Result := $0000ff;
          2: Result := $ff0000;
          3: Result := $00ff00;
          else Result := $00DAFF;//damit compiler nicht warnt
     end;
  end;
begin
   application.processmessages;
   Canvas.Pen.Width := 3;
   //Hintergrund
   if Fclickpos = pos
     then Canvas.Brush.Color := $fff0f0
     else Canvas.Brush.Color := $ffffff;
   Canvas.Rectangle(pos mod 3 * 166,pos div 3 * 166,pos mod 3 * 166 + 166,
pos div 3 * 166 + 166);
   //Striche
   Canvas.moveTo(pos mod 3*166, pos div 3 *166);//oben links
   Canvas.lineTo(pos mod 3*166+166, pos div 3 *166 + 166);//unten rechts
   Canvas.moveTo(pos mod 3*166+166, pos div 3 *166);//oben rechts
   Canvas.lineTo(pos mod 3*166, pos div 3 *166 + 166);//unten links

   //Mausauswahl blau machen
   if Fclickpos = pos
     then Canvas.Brush.Color := $ffbfbf
     else Canvas.Brush.Color := $cfcfcf;
   if FDrawSeiten
      then begin
              if Fpositionen[pos].getKante(0).getPasst
                 then Canvas.FloodFill(pos mod 3*166+10,pos div 3
*166+83,$000000,fsBorder);
              if Fpositionen[pos].getKante(1).getPasst
                 then Canvas.FloodFill(pos mod 3*166+83,pos div 3
*166+20,$000000,fsBorder);
              if Fpositionen[pos].getKante(2).getPasst
                 then Canvas.FloodFill(pos mod 3*166+100,pos div 3
*166+83,$000000,fsBorder);
              if Fpositionen[pos].getKante(3).getPasst
                 then Canvas.FloodFill(pos mod 3*166+83,pos div 3
*166+100,$000000,fsBorder);
           end;
   //Lagestrich zeichen
   case Fpositionen[pos].getOrientierung of//kante wählen
        0: begin
             Canvas.moveTo(pos mod 3*166, pos div 3 * 166 + 83);//mitte
links
             Canvas.lineTo(pos mod 3*166 + 20, pos div 3 * 166 + 83);
```

```
          end;
        1: begin
             Canvas.moveTo(pos mod 3*166 + 83, pos div 3 * 166);//mitte
oben
             Canvas.lineTo(pos mod 3*166 + 83, pos div 3 * 166 + 20);
          end;
        2: begin
             Canvas.moveTo(pos mod 3*166 + 146, pos div 3 * 166 +
83);//mitte rechts
             Canvas.lineTo(pos mod 3*166 + 166, pos div 3 * 166 + 83);
          end;
        3: begin
             Canvas.moveTo(pos mod 3*166 + 83, pos div 3 * 166 +
146);//mitte unten
             Canvas.lineTo(pos mod 3*166 + 83, pos div 3 * 166 + 166);
          end;
   end;

   //Wörter+Zahlen
   Canvas.Font.Name := 'Verdana';
   Canvas.Font.Size := 18;
   Canvas.Font.Style := [fsbold];
   Canvas.Brush.Style := bsclear;

   Canvas.Font.Color :=
colortohex(Fpositionen[pos].getKante(0).getFarbe());
   Canvas.Textout(pos mod 3*166+10, pos div 3
*166+70,formtoword(Fpositionen[pos].getKante(0).getForm));

   Canvas.Font.Color := colortohex(Fpositionen[pos].getKante(1).getFarbe);
   Canvas.Textout(pos mod 3*166+58, pos div 3
*166+20,formtoword(Fpositionen[pos].getKante(1).getForm));

   Canvas.Font.Color := colortohex(Fpositionen[pos].getKante(2).getFarbe);
   Canvas.Textout(pos mod 3*166+100, pos div 3 *166+70,
formtoword(Fpositionen[pos].getKante(2).getForm));

   Canvas.Font.Color := colortohex(Fpositionen[pos].getKante(3).getFarbe);
   Canvas.Textout(pos mod 3*166+58, pos div 3 *166+130,
formtoword(Fpositionen[pos].getKante(3).getForm));

   //Passzahl
   Canvas.Font.Size := 17;
   Canvas.Font.Color := clblack;
   if FDrawPasszahl
      then Canvas.Textout(pos mod 3*166+76, pos div 3
*166+88,IntToStr(Fpositionen[pos].getPasszahl));
   if FDrawNummer
      then Canvas.Textout(pos mod 3*166+5, pos div 3
*166,IntToStr(Fpositionen[pos].getNummer));
end;
```

aus mSpielfeld.pas

Sollen Karten gezeichnet werden ruft man die Prozedur *draw()* mit dem Parameter pos (welche Position soll gezeichnet werden?) auf. Das ganze Feld wird mit der Prozedur *drawfield()* gezeichnet.

Erst wird die Farbe eingestellt, dann der Hintergrund weiß gezeichnet und die Flächenumrandungen gezeichnet. Danach wird für jede Seite überprüft, ob sie passt. Passt eine Seite, wird diese grau markiert. Das selbe Verfahren passiert mit dem Strich der die Orientierung angibt. Am Ende wird die Form und die restlichen Informationen auf die Karte geschrieben. Text gibt man so aus:

```
Canvas.Textout(X:integer, Y: Integer, Text:String);
```

formtoword() und *colortohex()* wandeln die Zahlendaten der Positionen in Strings um, so dass diese farblich markiert und als Wort ausgegeben werden können. Die beiden untergeordneten Prozeduren werden nur in *draw()* benötigt und sind deswegen nur lokal erreichbar.

Jede obere linke Kante einer Position lässt sich mit `pos mod 3 * 166` definieren. Das Feld ist 500x500 Pixel groß und damit quadtraditsch, also gilt das Ergebnis für die X- und Y-Koordinate. Mit `pos mod 3` wird ermittelt, ob es die nullte, erste oder zweite Spalte ist. Diese Zahl muss man anschließend mit einem drittel (166Px) multiplizieren, um das richtige Ergebnis zu erhalten. Soll ein Objekt z.B. in der Mitte positioniert werden ist das `pos mod 3 * 166 + 88`.

LÖSUNGSALGORITHMEN

Ein Algorithmus ist eine Folge von Handlungsanweisungen um ein bestimmtes Problem zu lösen.[5] Manche bekannte Algorithmen lassen schon aus ihrem Namen erkennen, welche Aufgabe sie Lösen (Bsp. „Quicksort“). Die Geschwindigkeit ist oft ein entscheidender Aspekt, deshalb benötigt es Schätzungen. Die Schätzung erfolgt durch die Hochrechnung einer Messzeit für einige Durchgänge auf die maximalen Durchgänge.

Einige Algorithmen machen zur Lösung des Spiels Sinn, andere weniger. Es werden hier einige Möglichkeiten zur Lösung vorgestellt.

Der Zufallsalgorithmus

```
procedure TArtus.randomsolve(mintable:integer);
//lößt mittels blindem Legen
var partner1, partner2: Integer;
begin
  //solange die minimalen Punkte nicht erreich wurden
   while Spielfeld.tablepoints < mintable do
         begin
            //wenn aktiviert, Programm auf Ereignisse reagieren lassen
            if miProcess.Checked then application.processmessages;
            //wenn die Schleife über das Formular abgebrochen wird den
Algorithmus verlassen
            if schleifenbreak
               then begin
                      schleifenbreak := False;
                      exit;
                   end;
            //wiederhole Schleifendruchgänge bis die positionen
unterschiedlich sind
            repeat
               partner1 := Random(9);
               partner2 := Random(9);
            until partner1 <> partner2;
            //Karten vertauschen und drehen
            Spielfeld.tausche(partner1,partner2);
            Spielfeld.orientieren(partner1,Random(4));
            Spielfeld.orientieren(partner2,Random(4));
         end;
end;
```

aus mArtus.pas

---

5 vgl. Forsythe, Keenan, Organick (u.a.) Problemanalyse und Programmieren, Braunschweig, 1975, S. 1

Der Zufallsalgorithmus platziert alle Karten an einer zufälligen Position in einer zufälligen Lage, bis eine zuvor eingestellte Anzahl Tablepoints erreicht wird. Dies geschieht mit zwei Schleifen. Die äußere While-Schleife ist für den Fortgang zuständig (solange die Tablepoints kleiner als `mintable` sind) und die innere Repeat-Schleife sucht nach nicht gleichen Partnern, damit nicht mit sich selber getauscht wird.

Es braucht durchschnittlich 11,9 Milliarden Versuche[6], bis ein Feld vollständig richtig liegt. Die Zeitschätzung[7] brachte ein Ergebnis von 2,68h. Die Zeitschätzung liefert ein vage Zeitschätzung indem 50.000 mal Durchgänge des Algorithmus simuliert werden.

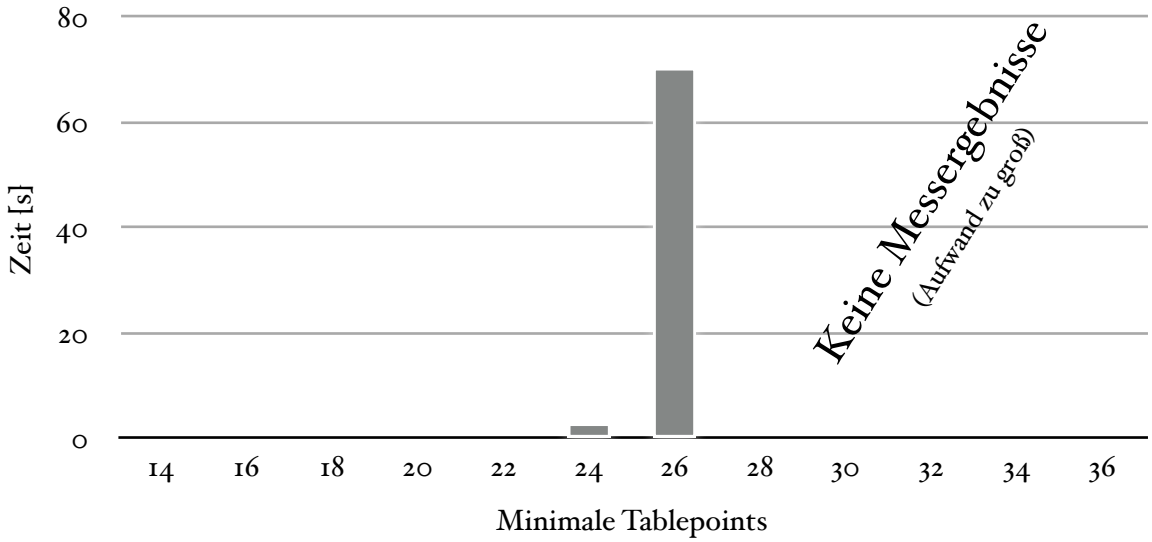
Bei 8 möglichen Lösungen lässt sich die Gesamtzeit dann mit

Gesamtzeit [s] = Zeit [s] * f(9) / 8 / Durchgänge

berechnen.

Der Parameter `mintable` vermindert das gewünschte Ergebnis nicht auf 36 zu erreichende Punkte für das gesamte Feld, sondern auf einen selbst gewünschten Wert. Sobald die Punkte des Feldes größer als `mintable` sind, bricht die Schleife ab.

Die durchschnittliche benötigte Zeit kann man in einem Diagramm darstellen.



Lösungen unter 60s Wartezeit erhält man noch bei mintable=26.

---

[6] f(9) / 8 = 1189085184O

[7] *btZeitschaetzenClick()*

Der verbesserte Zufallsalgorithmus

```
procedure TArtus.randomswitchsolve(mintable:integer);
//legt blind aber dreht mithilfe von autoturn oder bruteturn
var partner1, partner2, trys, trys2, max_trys:integer;
begin
   max_trys := 10;
   trys := 0;
   while Spielfeld.tablepoints < mintable do
   begin
      if miProcess.Checked then application.processmessages;
      //wenn abgebrochen werden soll
      if schleifenbreak
         then begin
                schleifenbreak := False;
                exit;
              end;
      //nach max_trys versuchen das Feld neu starten
      Inc(trys);
      if (trys > max_trys)
         then begin
                 Spielfeld.restart;
                 trys := 0
              end;
      repeat
         //mit trys2 endlosschleife verhindern, wenn Feld schon gelößt ist
         trys2 := 0;
         repeat
            inc(trys2);
            partner1 := Random(9)
         until (Spielfeld.getPositionen(partner1).getPasszahl < 4) or (trys2 > 500);
         trys2 := 0;
         repeat
            inc(trys2);
            partner2 := Random(9)
         until (Spielfeld.getPositionen(partner2).getPasszahl < 4) or (trys2 > 500);
      until partner1 <> partner2;
      Spielfeld.tausche(partner1,partner2);
      //je nach Auswahl Drehverfahren wählen
      if rgDrehalgorithmus.ItemIndex=1
         then autoturn
         else bruteturn;
   end;
end;
```

aus mArtus.pas

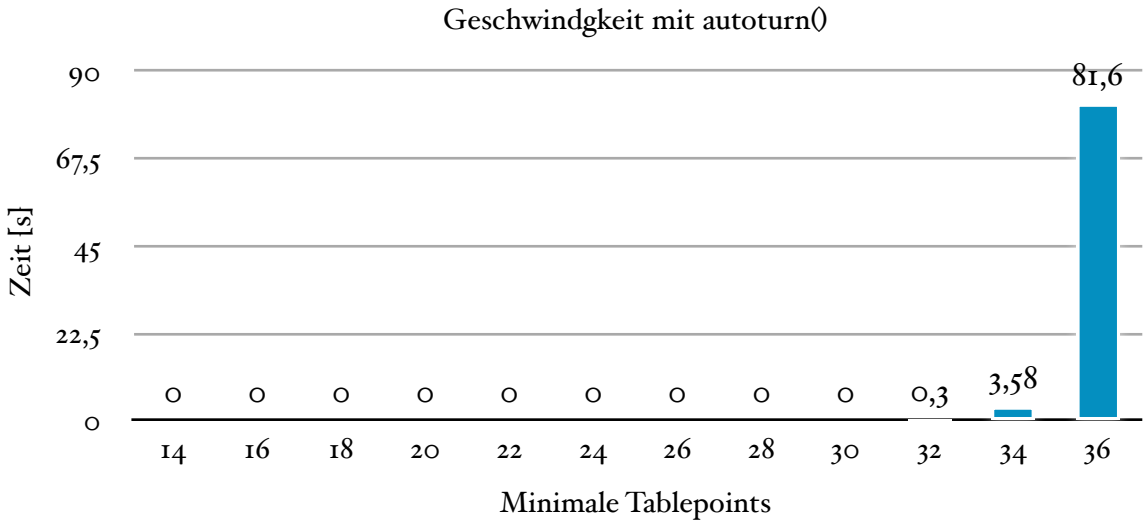
Die Prozedur *randomswitchsolve()* basiert auf *randomsolv*e(). Die Verbesserung besteht darin, dass die zufällig ausgewählten Karten nicht schon vier richtige Kanten haben dürfen. Weiterhin werden anstatt einer zufälligen Orientierung der Karten diese mit *bruteturn()* oder *autoturn()* gedreht. Sollte der Algorithmus sich festgefahren haben, wird nach 10 Versuchen neu gestartet. Dies kann passieren, wenn

etwa drei Karten mit der Passzahl < 4 übrig bleiben, die nun immer wieder miteinander vertauscht werden, aber damit keine Lösung möglich ist.

Eine Methode, um die Zeit zu schätzen ist hier nicht möglich.

Die erwarteten Durchgänge können hier aber nicht ohne großen mathematischen oder rechnerischen Aufwand bestimmt werden.

Die Messergebnisse zeigen eine starke Zeitverbesserung, wenn *autoturn()* zum drehen verwendet wird.

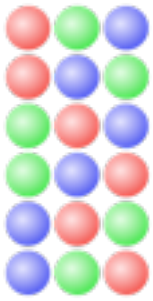


Die Brute-Force-Methode

Ein Algorithmus nach der Brute-Force-Methode ist ein Algorithmus der alle Möglichkeiten systematisch überprüft. Ein einfaches Anwendungsbeispiel für so eine Methode ist das herausfinden von Kennwörten.

Eine Systematische Prüfung aller Möglichkeiten geht mit Hilfe von zwei Schritten. Erst tauscht man jede Karte mit jeder anderen. Dies nennt man Permutation. Nach jedem Tausch probiert man alle möglichen Orientierungen in dieser Kartenkonstellation aus.

Hier wird ein unsortierter Permutationsalgorithmus nach Alexander Bogomolyn verwendet:



Permutationen dreier Kugeln

(http://de.wikipedia.org/wiki/Datei:Permutations_RGB.svg)

```
#include <stdio.h>


void print(const int *v, const int size)
{
  if (v != 0) {
    for (int i = 0; i < size; i++) {
      printf("%4d", v[i] );
    }
    printf("\n");
  }
} // print


void visit(int *Value, int N, int k)
{
  static level = -1;
  level = level+1; Value[k] = level;

  if (level == N)
    print(Value, N);
  else
    for (int i = 0; i < N; i++)
      if (Value[i] == 0)
        visit(Value, N, i);

  level = level-1; Value[k] = 0;
}


main()
{
  const int N = 4;
  int Value[N];
  for (int i = 0; i < N; i++) {
    Value[i] = 0;
  }
  visit(Value, N, 0);
}
```

http://www.bearcave.com/random_hacks/permute.html, letzter Zugriff 12.03.2012 18:50 Uhr

Dieser gibt alle Permutationen nacheinander aus. Nach jeder Permutation müssen alle 262.144 Orientierungen geprüft werden. Der Algorithmus wurde nach Delphi übersetzt und die Ausgabe *print()* durch eine Funktion ersetzt, die alle Orientierungen auch mit einer Brute-Force-Methode prüft (*bruteturn()*). Sollte eine Lösung gefunden worden sein, stoppt der Algorithmus.

Der Algorithmus ist erstaunlicherweise sehr langsam. Bei der Zeitschätzung wird die Zeit für *bruteturn()* gemessen und mit 9! multipliziert und anschließend durch die Lösungsanzahl dividiert. Die erwartete Zeit beträgt hier 3,85 Tage.

Bei der Zeitschätzung wird angenommen, dass nach einem Achtel aller Kombinationen die erste Möglichkeit gefunden wird, ein Fund kann aber auch schon früher oder später erfolgen.

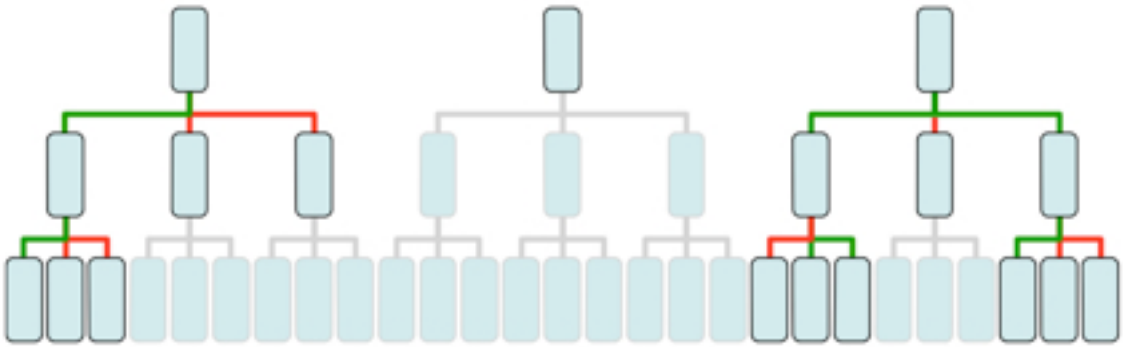
Rekursives Backtracking

```
procedure TArtus.backtracking();
    function rekursion_backtracking(karte:integer):boolean;
    var span, startseite:integer;
    begin
       //extra für Darstellung
       Spielfeld.setclickpos(karte);
       //falls an letzter Karte angelangt, abbrechen und true zurück geben
       if karte >= 9
          then begin
                 Result := true;
                 exit;
               end;
       span := 1;
       //so lannge bis karte+partner über die letzte Position hinaus ist
       repeat
          //startseite Speichern
          startseite := Spielfeld.getPositionen(karte).getOrientierung;
          //drehen bis die Karte alle Orientierungen hatte oder selber
passt+nächste Karte
          repeat
             //vor der Prüfung aktualisieren
             Spielfeld.refreshPasszahl(karte);
             //falls kante links und oben passt sowie die nachbarkarte
passt, true zurück liefern
             if (Spielfeld.getPositionen(karte).getKante(0).getPasst)
                and (Spielfeld.getPositionen(karte).getKante(1).getPasst)
                and (rekursion_backtracking(karte+1))
                then begin
                       Result := true;
                       exit;
                     end;
             //drehen
             Spielfeld.drehen(karte);
          until startseite =
Spielfeld.getPositionen(karte).getOrientierung;
          //wenn partner größer als die Nachbarkarte, dann tauschen
          if span > 1 then Spielfeld.tausche(karte,karte+span);
          //tauschkarte erhöhen
          inc(span);
       until karte+span >= 9;
    end;
begin
   Spielfeld.Restart;
   rekursion_backtracking(0);
end;
```

aus mArtus.pas

Backtracking ist ein Prinzip, ähnlich der Brute-Force-Methode, dabei werden aber nur alle Wege abgegangen, die zum Ziel führen können. Geht es an einer Stelle nicht mehr weiter, geht man einen Schritt zurück und macht dort weiter.[8]

In einem Baumdiagramm würde dies zum Beispiel so aussehen:



nicht passende Fälle Rot, passende Grün, nicht geprüfte Grau, maximal drei Unterknoten (Würde das Diagramm alle Permutationen abbilden, wären an der untersten Ebene 368.220 Knoten).

Alle grauen Fälle werden nicht geprüft. Bei 95. Mrd. Möglichkeiten spart diese immense Mengen von Rechnungen, was zu einer sehr hohen Geschwindigkeit führt.

Durch Rekursion lässt sich hier ein Backtrackingverfahren relativ einfach implementieren;

```
[...]
if (Spielfeld.getPositionen(karte).getKante(0).getPasst)
                and
(Spielfeld.getPositionen(karte).getKante(1).getPasst)
                and (rekursion_backtracking(karte+1))
                then begin
                       Result := true;
                       exit;
                     end;
            //drehen
            Spielfeld.drehen(karte);
[...]
```

Die if-Abfrage enthält einen rekursiven Funktionsaufruf. Liegt nun die nächste Position falsch (`rekursion_backtracking(karte+1)`)wird gedreht oder danach getauscht, das heißt man geht im Baumdiagramm eine Ebene aufwärts. Geben

[8] http://moritz.faui2k3.org/de/backtracking, letzter Zugriff am 12.03.2012 um 15:50 Uhr

alle Rekursionsebenen *true* zurück, passiert nichts mehr und der Algorithmus beendet sich damit selbst.

Eine Zeitschätzung ist hier nicht nötig, da der Algorithmus schneller ist, als mit der herkömmlichen Weise die Zeit messbar ist.

Weitere Algorithmen

Man könnte weitere Algorithmen schreiben, indem man durch Analyse der Karten prüft, wo diese überhaupt liegen können. Die *autoturn()* Prozedur sollte ursprünglich Teil eins Algorithmus werden, der schlechte Karten zuerst tauscht. Ansätze finden dazu finden sich dazu in der JavaScript-Version.

Zur Problemlösung lässt sich auch ein Simulated Annealing-Algorithmus[9] verwenden[10].

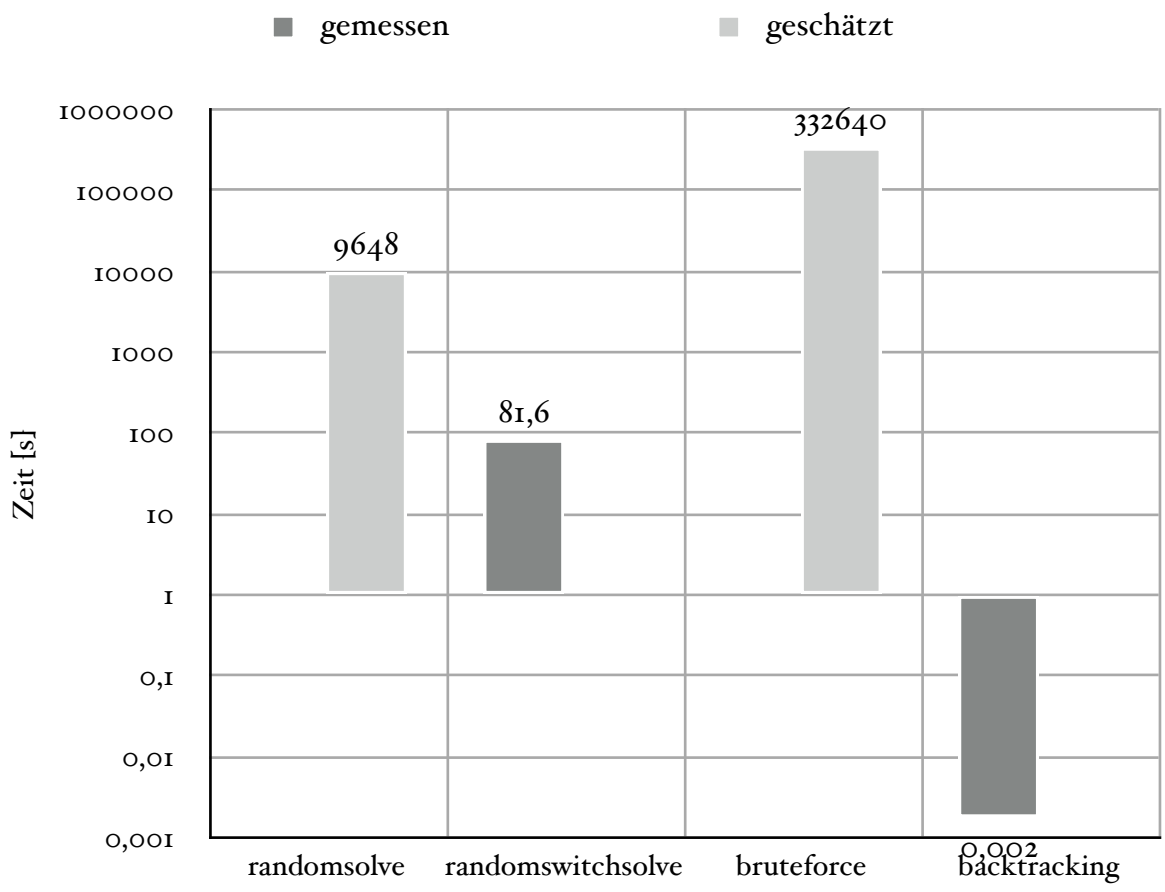
Algorithmen im Vergleich

Die Algorithmen lassen sich nach der geschätzten und gemessenen Geschwindigkeit vergleichen. Am schnellsten ist der rekursive Backtrackingalgorithmus. Danach folgt der *randomswitchsolve()* und *randomsolve()*. Brute-Force ist bei etwa 12. Milliarden Fällen, der ineffizienteste. Unerwarteterweise ist ein komplett zufälliger Algorithmus schneller als ein linearer. Der *bruteforce()* Algorithmus muss, durch die Linearität bedingt, mehr Schleifendurchläufe und Überprüfungen für Zählvariablen machen, welche viel Zeit kosten. Hier sind die Daten im Diagramm dargeestellt

---

[9] ein an der Natur orientiertes Optimierungsverfahren bei dem auch negative Ergebnisse zählen
vgl. http://isgwww.cs.uni-magdeburg.de/sim/vilab/2003/presentations/martin.pdf, letzter Zugriff am 12.03.2012 um 16 Uhr

[10] http://www.ateus.ch/FGMath/ProbAlg/HPuzzle.html, letzter Zugriff am 12.03.2012 um 18:45 Uhr


gemessen
geschätzt
1000000
100000
10000
1000
100
10
1
0,1
0,01
0,001
Zeit [s]
9648
81,6
332640
0,002
randomsolve
randomswitchsolve
bruteforce
backtracking

ARBEITSPROZESSBERICHT

Am 13. Januar 2012 wurde das Thema verbindlich festgelegt.

Letzter Abgabetermin war der 14. März 2012.

Zuerst habe ich das Programm in JavaScript geschrieben und mich parallel dazu zu Algorithmen informiert. Die Architektur wurde zuerst vorgenommen. Nachdem die Daten importiert waren und ausgegeben werden konnten, entwickelte ich die Verwaltung der Daten. Nachdem es Probleme bei der Entwicklung der Prozeduren gab, wurde die graphische Ausgabe um die Lage erweitert und zum leichteren Verständnis des Feldes, die Markierungen der passenden Seiten durch eine graue Markierung ermöglicht. Später wurde der Aufbau der Architektur nochmals verbessert, was nur einen logischen Vorteil hatte und die Portierung in Delphi erleichterte. Als letztes widmete ich mich der Umsetzung der Algorithmen.

In Delphi legte ich nun das GUI als Erstes fest. Der Code musste nun übersetzt werden. Die Darstellung und einfache, intuitive Bedienung benötigte weitere Schaltflächen, die nach und nach hinzugefügt wurden. Nach der Übersetzung des Codes schrieb ich den schriftlichen Teil. Der schriftliche Teil benötigte noch Grafiken, die erstellt werden mussten. Für die Diagramme wurden Messungen erstellt, die nach kleinen Optimierungen am Code, alle unter gleichen Bedingungen durchgeführt werden mussten. Die benutzen Hilfsmittel und Quellen mussten schließlich aufgelistet werden werden.

Am 13. März wurde die CD gebrannt.

VERWENDETE HILFSMITTEL

Editoren

Panic „Coda“ V. 1.7.5

Allan Odgaard „TextMate“ V. 1.5.10 (1631)

Borland „Delphi“ Personal V. 7.0 (4.453)

Browser

Google Inc. „Google Chrome“ V. 16.0.912.77

Apple Inc. „Safari“ V. 5.1.3 (7534.53.10)

Weitere Hilfsmittel

Apple Inc. „Grapher„ V. 2.2 (43)

Taschenrechner: Casio „fx-9860GII“

Pixelmator Team Ltd. „Pixelmator“ V.2.0.1 (10659)

Spencer Kimbell, Peter Mattis und das GIMP Team „Gimp“ 2.6.12

Programm getestet mit

Computer: 2,26 GHz Intel Core 2 Duo, 4GB 1067 MHz DDR3 RAM

Betriebsysteme:

Windows 7 (virtuell mit Parrallels Dekstop 7)

Windows 7 (nativ)

Windows XP (nativ)

QUELLENVERZEICHNIS

Forsythe, Keenan, Organick (u.a.) Problemanalyse und Programmieren, Braunschweig, 1975

http://www.bearcave.com/random_hacks/permute.html, letzter Zugriff am 12.03.2012 um 18:50 Uhr

http://moritz.faui2k3.org/de/backtracking, letzter Zugriff am 12.03.2012 um 15:50 Uhr

http://isgwww.cs.uni-magdeburg.de/sim/vilab/2003/presentations/martin.pdf, letzter Zugriff am 12.03.2012 um 16 Uhr

http://www.ateus.ch/FGMath/ProbAlg/HPuzzle.html, letzter Zugriff am 12.03.2012 um 18:45 Uhr

Ich erkläre, dass ich die Facharbeit ohne fremde Hilfe angefertigt und nur die im Literaturverzeichnis angeführten Quellen und Hilfsmittel benutzt habe.

Benedikt Vogler