B T R A C K

Backtracking-Algorithmen

© Herbert Paukert

BACKTRACKING - PROGRAMME

[01] Die Grundlagen

Es gibt im Leben Probleme, welche nicht mit Hilfe von Berechnungsformeln direkt lösbar sind. Zwei Problemstellungen sollen diesen Sachverhalt illustrieren:

Problem [P1.1]: Ein Labyrinth besteht aus vielen Sackgassen und einigen Wegen, die ins Freie führen. Gesucht ist EIN solcher Weg in die Freiheit.

Problem [P2.1]: Ein Wegnetz besteht aus einer Menge von Orten und aus verschieden langen Verbindungswegen zwischen diesen. Gesucht ist EIN möglicher Weg, der von einem bestimmten Startort zu einem bestimmten Zielort führt.

In solchen Problemsituationen wird der Lösungsweg am besten durch "Versuch und Irrtum" (trial and error) ermittelt. Dazu wählen wir im praktischen Leben aus der Menge unserer Verhaltensmöglichkeiten nur jene aus, welche einen **Erfolg versprechen**. Führt ein solcher Versuch dann nicht zum Erfolg, so wird er ausgeschieden und die nächste Erfolg versprechende Verhaltensmöglichkeit ausgeführt. Dieses Verfahren wird so lange wiederholt, bis entweder ein tatsächlicher Erfolg (d.h. eine Lösung) eingetreten ist oder alle Möglichkeiten erschöpft sind.

Das Verfahren kann häufig rekursiv formuliert werden. Dabei stellt sich die Lösungssuche als ein Prozess des Versuchens und Nachprüfens dar, der einen Baum von untergeordneten Problemen aufbaut und durchläuft. Bei vielen Problemstellungen wächst dieser Suchbaum sehr schnell. Eine genaue Beschreibung rekursiver Techniken findet der Leser in den entsprechenden Kapiteln.

Unsere beiden Problemstellungen können noch erweitert werden:

Problem [P1.2]: Im Labyrinth sollen ALLE Wege, welche in die Freiheit führen, gefunden und davon der kürzeste ermittelt werden.

Problem [P2.2]: Im Wegnetz sollen ALLE Verbindungswege zwischen einem Startort und einem Zielort gefunden und davon der kürzeste ermittelt werden.

Vor allem das zweite Problem ist für den Gütertransport auf unseren Verkehrswegen von großer wirtschaftlicher Bedeutung. Dabei werden wieder, wie oben dargestellt, alle Erfolg versprechenden Wege rekursiv durchlaufen. Jene, die zum Ziel führen, werden gespeichert, alle anderen ausgeschieden. In der Menge der erfolgreichen Möglichkeiten wird sodann die beste ermittelt (d.h. in unserem Fall der kürzeste Lösungsweg).

Dieses allgemeine Verfahren zur Lösung von Problemen wird *BACKTRACKING* genannt. Das charakteristische Merkmal solcher Strategien besteht in Folgendem: Man versucht wiederholt Schritte in Richtung des Ziels und zeichnet sie auf. Stellt sich später heraus, dass sie in eine Sackgasse führten, so macht man sie rückgängig und beginnt einen Weg in eine neue Richtung. Am Ende hat man entweder alle möglichen Wege durchlaufen oder voher schon das Ziel erreicht.

In unseren Beispielen gibt es immer nur endlich viele Verhaltensmöglichkeiten (i = 1,2, ..., N). Am Beginn des Lösungsverfahrens wird in der Programmierpraxis eine logische Kennvariable auf "Falsch" gesetzt. So lange nur Misserfolge eintreten, bleibt ihr Wert auf "Falsch". Tritt im Verlaufe des Backtrackings ein Erfolg ein, wird also das Ziel erreicht, dann erhält diese Kennvariable den Wert "Wahr". Will man nur EINEN Lösungsweg finden, dann kann das Backtracking abgebrochen werden. Will man jedoch ALLE Lösungswege finden, dann wird es so lange fortgesetzt, bis alle vorgegebenen Verhaltensmöglichkeiten durchlaufen sind.

Das allgemeine Muster solcher Backtracking-Verfahren ist in nachfolgender Prozedur schematisch dargestellt.

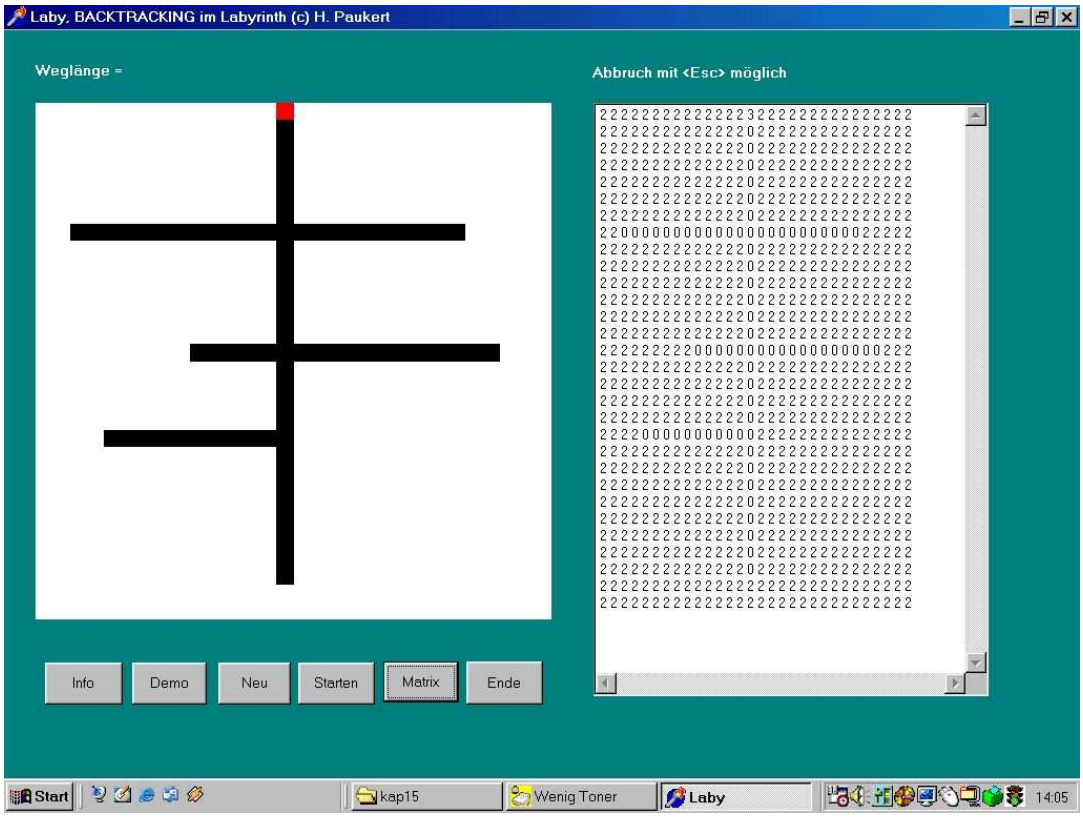
```
PROCEDURE  Versuch(Möglichkeit i);
BEGIN
   IF  alle Möglichkeiten erschöpft (i > N)
      THEN  Verlasse die Prozedur;
   IF  Möglichkeit i  erfolgreich
      THEN  Speichere die Möglichkeit;
            Setze eine Kennvariable auf True;
            (Verlasse die Prozedur);
   IF  Möglichkeit i  Erfolg versprechend
      THEN  Versuch(Möglichkeit i +1)
      ELSE  Scheide die Möglichkeit aus;
END
```

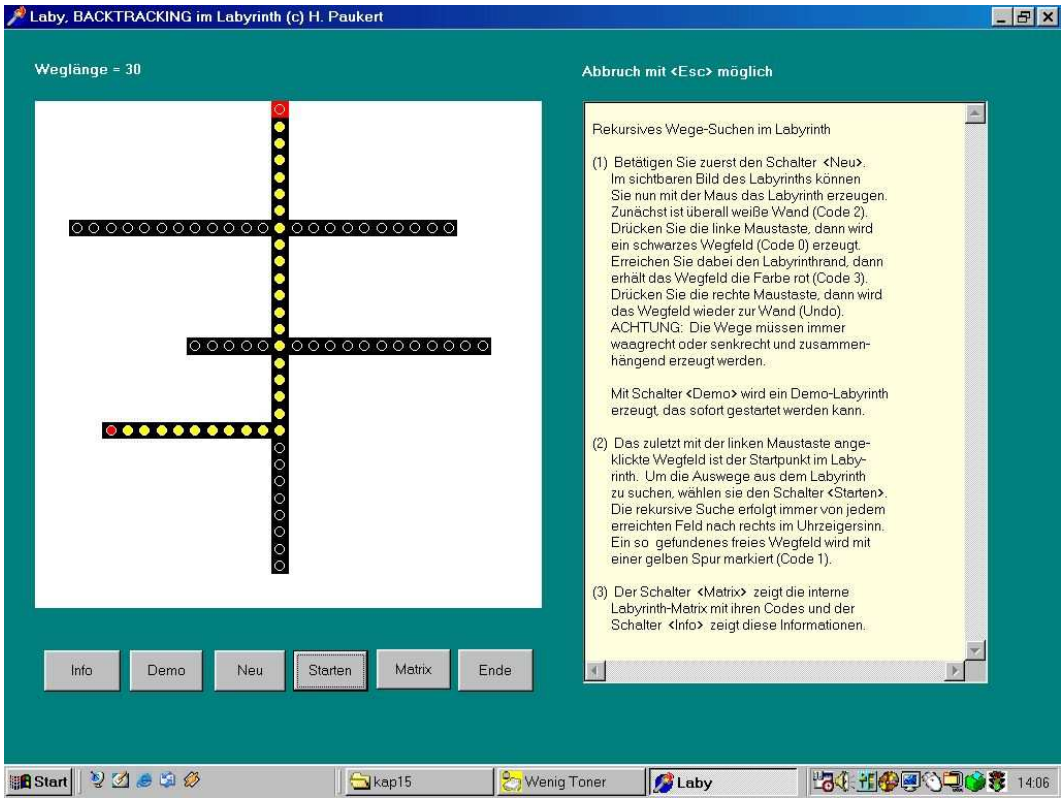
Die Prozedur ***Versuch*** ist rekursiv, denn sie ruft sich selbst auf. Dabei gibt es zwei Möglichkeiten für eine Abbruchbedingung. Entweder wird die Prozedur beim Eintreten des ersten Erfolges verlassen oder wenn alle Möglichkeiten durchgespielt worden sind, d.h. der Parameter ***i*** größer ***N*** ist. Im ersten Fall wird nur EIN Lösungsweg aufgezeichnet, im zweiten Fall ALLE Lösungswege.

Im Hauptprogramm wird die Prozedur mit der Anweisung ***Versuch****(Möglichkeit **1**)* aufgerufen. Vor diesem Aufruf muss die Kennvariable auf *False* gesetzt werden. Nach der Beendigung der Prozedur wird an der Kennvariablen Erfolg oder Misserfolg des Verfahrens abgelesen. Ist ihr Wert *True*, dann wurde das Ziel erreicht. Ist ihr Wert *False*, dann wurde das Ziel nicht erreicht.

[02] Backtracking im Labyrinth (*laby*)

Das Programm ***Laby*** stellt eine Lösung des einfachen Labyrinthproblems [P1.1] dar. Die folgende Abbildung zeigt das Formular:





Für das Labyrinth wird im Hauptspeicher eine zweidimensionale Matrix *Laby* (*array [1..Max, 1..Max] of byte*) reserviert. Die Konstante *Max*, welche die Seitenlänge des Labyrinths bestimmt, ist mit 30 begrenzt. Die Felder der Matrix *Laby[i,k]* können mit vier Ziffern (*0,1,2,3*) belegt werden. ***2*** ist der Code für eine undurchdringliche Wand. ***0*** ist der Code für ein freies Feld eines Weges. ***1*** markiert eine Spur, d.h., dieses Wegfeld wurde beschritten. Der Code ***3*** kennzeichnet die Endfelder von Wegen, welche am Rand des Labyrinths liegen und in die Freiheit führen.

Visualisiert wird die Labyrinth-Matrix durch eine Image-Komponente (0,0,XMax,YMax) im Formular. Dieses Labyrinth-Bild wird in Rechtecke unterteilt, sodass einem Labyrinthfeld *Laby[i,k]* genau ein Rechteck entspricht. Dabei sind den Zifferncodes *0, 1, 2, 3* in den Labyrinthfeldern die Rechtecksfarben *schwarz, gelb, weiß, rot* zugeordnet.

Zuerst muss über den Schaltknopf <*Neu*> das gesamte Labyrinth mit dem Wandcode *2* (Farbe *weiß*) belegt werden. Das leistet die Prozedur *Initialisieren*. Sodann können mit Hilfe der Maus in der Image-Komponente die Wege im Labyrinth erzeugt werden. Die Betätigung der linken Maustaste erzeugt ein *schwarzes* Rechteck im Labyrinth-Bild und den Wegcode *0* im entsprechenden Feld der Labyrinth-Matrix. Mit der rechten Maustaste wird alles wieder rückgängig gemacht, d.h. aus Weg wird wieder Wand. Bei Erreichen des Labyrinthrandes erhält ein Feld den Code *3*. Dadurch wird ein Ausgang des Labyrinths gekennzeichnet. Unbedingt zu beachten ist, dass die Wege immer waagrecht oder senkrecht und zusammenhängend erzeugt werden. Diese Entwicklung des Labyrinths ist in der Prozedur *Erzeugen* programmiert.

Die Kernroutine des Programms ist die rekursive Prozedur ***Weg_Suchen***. Sie entspricht in modifizierter Form dem allgemeinen Backtracking-Schema, welches früher bereits erklärt wurde. Dabei befindet man sich zunächst auf einem freien Wegfeld (*Laby[i,k] = 0*). Nun werden, nach rechts beginnend und im Uhrzeigersinn drehend, die vier Nachbarfelder überprüft, ob sie ebenfalls freie Wegfelder sind, d.h. ob sie einen Erfolg versprechen. Wenn das der Fall ist, dann wird das Verfahren dort rekursiv fortgesetzt. Handelt es sich dabei um eine Sackgasse, dann wird zurück gegangen und die nächste Richtung überprüft. Handelt es sich hingegen um einen Ausgang in die Freiheit, dann wird die Prozedur abgebrochen. In der logischen Kennvariablen *AusFlag* wird Erfolg (*True*) oder Misserfolg (*False*) des Verfahrens festgehalten.

Als Anfangspunkt *Start* kann jedes freie Wegfeld mit der linken Maustaste angeklickt werden. Mit dem Schaltknopf <*Start*> beginnt dann die Wegsuche im Labyrinthfeld *Laby[Start.y,Start.x*]. Dem Startfeld wird, so wie auch den Labyrinthausgängen, die Farbe *rot* zugewiesen.

Zur Markierung eines Wegfeldes (Spurencode 1, *gelb*) dient die Prozedur *Markieren*. Mit der Prozedur *Spur_Löschen* werden alle Markierungen wieder entfernt (freier Wegcode 0, *schwarz*).

Mit dem Schaltknopf <*Info*> wird ein Memo für Hilfsinformationen angezeigt. Außerdem kann hier eine Verzögerungszeit eingegeben werden, wodurch die Geschwindigkeit des Wegsuchens gesteuert wird. Mit dem Schaltknopf <*Demo*> wird ein schon eingerichtetes Labyrinth angezeigt. Mit dem Schaltknopf <*Matrix*> wird in einem zweiten Memo die Labyrinth-Matrix dargestellt. Mit dem Schaltknopf <*Beenden*> kann schließlich das Programm beendet werden.

Anzumerken ist noch, dass in der Matrix *Laby[i,k]* der erste Index ***i*** die Zeile angibt (im Image der ***y***-Wert) und der zweite Index ***k*** die Spalte angibt (im Image der ***x***-Wert).

Listing der Unit "*laby_u.pas*"

```
unit laby_u;
{ Backtracking im Labyrinth (c) Herbert Paukert }
{ Suchen EINES Lösungsweges aus dem Labyrinth   }

interface
uses
  Windows, Messages, SysUtils, Classes, Graphics,
  Controls, Forms, Dialogs, ExtCtrls, StdCtrls;

type
  TForm1 = class(TForm)
    Memo1  : TMemo;                  // Informationsfeld
    Memo2  : TMemo;                  // Matrixfeld
    Image1 : TImage;                 // Labyrinth-Bild
    Label1 : TLabel;                 // Anzeige der Weglänge
    Label2 : TLabel;                 // Informationstext
    Button1: TButton;                // <Info>
    Button2: TButton;                // <Demo>
    Button3: TButton;                // <Laby Neu>
    Button4: TButton;                // <Starten>
    Button5: TButton;                // <Matrix>
    Button6: TButton;                // <Beenden>

    procedure FormCreate(Sender: TObject);
    procedure Image1MouseDown(Sender: TObject; Button: TMouseButton;
                              Shift: TShiftState; X, Y: Integer);
    procedure Image1MouseMove(Sender: TObject; Shift: TShiftState;
                              X,Y: Integer);
    procedure Button1Click(Sender: TObject);
    procedure Button2Click(Sender: TObject);
    procedure Button3Click(Sender: TObject);
    procedure Button4Click(Sender: TObject);
    procedure Button5Click(Sender: TObject);
    procedure Button6Click(Sender: TObject);
    private { Private declarations }
    public  { Public declarations }
  end;

var Form1: TForm1;

implementation
{$R *.DFM}

const Zeit : Integer = 10;           // Verzögerungszeit in Millisekunden
      Max  = 30;                     // Seitenlänge der Labyrinth-Matrix

      Frei = 0;                      // freier Weg (schwarz)
      Spur = 1;                      // Wegspur (gelb)
      Wand = 2;                      // feste Wand (weiß)
      Aus  = 3;                      // Ausgang (rot)

      Farbe : array[0..3] of TColor = (clBlack,clYellow,clWhite,clRed);
```

```
type  TLaby = array[1..Max,1..Max] of Byte;

      // Der erste Index entspricht den Zeilen (i bzw. y)
      // Der zweite Index entspricht den Spalten (k bzw. x)

                                            // Demo-Labyrinth-Matrix
const Laby0 : TLaby =
      ((2,2,2,2,2,2,2,2,2,2,2,2,2,2,3,2,2,2,2,2,2,2,2,2,2,2,2,2,2,2),
       (2,2,2,2,2,2,2,2,2,2,2,2,2,2,0,2,2,2,2,2,2,2,2,2,2,2,2,2,2,2),
       (2,2,2,2,2,2,2,2,2,2,2,2,2,2,0,2,2,2,2,2,2,2,2,2,2,2,2,2,2,2),
       (2,2,2,2,2,2,2,2,2,2,2,2,2,2,0,2,2,2,2,2,2,2,2,2,2,2,2,2,2,2),
       (2,2,2,2,2,2,2,2,2,2,2,2,2,2,0,2,2,2,2,2,2,2,2,2,2,2,2,2,2,2),
       (2,2,2,2,2,2,2,2,2,2,2,2,2,2,0,2,2,2,2,2,2,2,2,2,2,2,2,2,2,2),
       (2,2,2,2,2,2,2,2,2,2,2,2,2,2,0,2,2,2,2,2,2,2,2,2,2,2,2,2,2,2),
       (2,2,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,2,2,2,2,2),
       (2,2,2,2,2,2,2,2,2,2,2,2,2,2,0,2,2,2,2,2,2,2,2,2,2,2,2,2,2,2),
       (2,2,2,2,2,2,2,2,2,2,2,2,2,2,0,2,2,2,2,2,2,2,2,2,2,2,2,2,2,2),
       (2,2,2,2,2,2,2,2,2,2,2,2,2,2,0,2,2,2,2,2,2,2,2,2,2,2,2,2,2,2),
       (2,2,2,2,2,2,2,2,2,2,2,2,2,2,0,2,2,2,2,2,2,2,2,2,2,2,2,2,2,2),
       (2,2,2,2,2,2,2,2,2,2,2,2,2,2,0,2,2,2,2,2,2,2,2,2,2,2,2,2,2,2),
       (2,2,2,2,2,2,2,2,2,2,2,2,2,2,0,2,2,2,2,2,2,2,2,2,2,2,2,2,2,2),
       (2,2,2,2,2,2,2,2,2,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,2,2,2),
       (2,2,2,2,2,2,2,2,2,2,2,2,2,2,0,2,2,2,2,2,2,2,2,2,2,2,2,2,2,2),
       (2,2,2,2,2,2,2,2,2,2,2,2,2,2,0,2,2,2,2,2,2,2,2,2,2,2,2,2,2,2),
       (2,2,2,2,2,2,2,2,2,2,2,2,2,2,0,2,2,2,2,2,2,2,2,2,2,2,2,2,2,2),
       (2,2,2,2,2,2,2,2,2,2,2,2,2,2,0,2,2,2,2,2,2,2,2,2,2,2,2,2,2,2),
       (2,2,2,2,0,0,0,0,0,0,0,0,0,0,0,2,2,2,2,2,2,2,2,2,2,2,2,2,2,2),
       (2,2,2,2,2,2,2,2,2,2,2,2,2,2,0,2,2,2,2,2,2,2,2,2,2,2,2,2,2,2),
       (2,2,2,2,2,2,2,2,2,2,2,2,2,2,0,2,2,2,2,2,2,2,2,2,2,2,2,2,2,2),
       (2,2,2,2,2,2,2,2,2,2,2,2,2,2,0,2,2,2,2,2,2,2,2,2,2,2,2,2,2,2),
       (2,2,2,2,2,2,2,2,2,2,2,2,2,2,0,2,2,2,2,2,2,2,2,2,2,2,2,2,2,2),
       (2,2,2,2,2,2,2,2,2,2,2,2,2,2,0,2,2,2,2,2,2,2,2,2,2,2,2,2,2,2),
       (2,2,2,2,2,2,2,2,2,2,2,2,2,2,0,2,2,2,2,2,2,2,2,2,2,2,2,2,2,2),
       (2,2,2,2,2,2,2,2,2,2,2,2,2,2,0,2,2,2,2,2,2,2,2,2,2,2,2,2,2,2),
       (2,2,2,2,2,2,2,2,2,2,2,2,2,2,0,2,2,2,2,2,2,2,2,2,2,2,2,2,2,2),
       (2,2,2,2,2,2,2,2,2,2,2,2,2,2,2,2,2,2,2,2,2,2,2,2,2,2,2,2,2,2),
       (2,2,2,2,2,2,2,2,2,2,2,2,2,2,2,2,2,2,2,2,2,2,2,2,2,2,2,2,2,2));

var   Laby : TLaby;                   // Labyrinth-Matrix
      XMax,YMax : Integer;            // Abmessungen des Labyrinth-Bildes
      a,b  : Real;                    // Abmessungen eines Labyrith-Bild-Feldes
      r    : Integer;                 // Radius des Markierungskreises (Wegspur)
      Start: TPoint;                  // Startpunkt in der Labyrinth-Matrix
      Len  : Integer;                 // Länge eines gefundenen Weges
      WegStr : String;                // Ausgabestring der gefundenen Weglänge
      AusFlag: Boolean;               // Kennzeichen für gefundenen Ausweg
      BreakFlag: Boolean;             // Kennzeichen für Abbruch mit <Escape>
      X0,Y0: Integer;                 // gemerkte Mausposition


procedure Warten(Zeit: Integer);
{ Unterbricht das Programm um 'Zeit' MilliSekunden }
var Zeit0 : Integer;
begin
  Zeit0 := GetTickCount;              // Internen Zeitzähler aufrufen
  repeat
     Application.ProcessMessages;     // ermöglicht Systembotschaften
  until (GetTickCount - Zeit0) > Zeit;
end;

procedure Initialisieren;
{ Belegt die Labyrinthplätze mit dem Wandcode }
{ und zeichnet das grafische Labyrinth-Bild.  }
var i,k : Integer;
begin
  for i := 1 to Max do
    for k := 1 to Max do
        Laby[i,k] := Wand;

  with Form1.Image1.Canvas do begin
    Brush.Style := bsSolid;
    Brush.Color := Farbe[Wand];
    Pen.Color   := Farbe[Wand];
    Rectangle(0,0,XMax,YMax);
  end;
end;
```

```
procedure Kopieren;
{  Kopiert die komplette Demo-Matrix in das Labyrinth }
var Code,i,k : Integer;
    P,Q : TPoint;  // linkes,obere und rechtes,untere Eck eines Labyrinthfeldes
begin
  for i := 1 to Max do
    for k := 1 to Max do begin
        Laby[i,k] := Laby0[i,k];
        Code := Laby[i,k];
        P.x := Round((k-1)*a); P.y := Round((i-1)*b);
        Q.x := Round(k*a); Q.y := Round(i*b);
        with Form1.Image1.Canvas do begin
          Brush.Style := bsSolid;
          Brush.Color := Farbe[Code];
          Pen.Color   := Farbe[Code];
          Rectangle(P.x,P.y,Q.x,Q.y);
        end;
    end;
end;

procedure Erzeugen(Code,X,Y : Integer);
{ Übernimmt die Mauskoordinaten X,Y bei Mausklick und belegt das entsprechende }
{ Labyrinthfeld mit dem Code für Frei (linker Klick) oder Wand (rechter Klick) }
var i,k : Integer;
    P,Q : TPoint;  // linkes,obere und rechtes,untere Eck eines Labyrinthfeldes
begin
  for i := 1 to Max do
    for k := 1 to Max do begin
      P.x := Round((k-1)*a); P.y := Round((i-1)*b);
      Q.x := Round(k*a); Q.y := Round(i*b);
      if (X >= P.x) and (X < Q.x) and(Y >= P.y) and (Y < Q.y) then begin
         Laby[i,k] := Code;
         with Form1.Image1.Canvas do begin
           Brush.Color := Farbe[Code];
           Pen.Color   := Farbe[Code];
           Rectangle(P.x,P.y,Q.x,Q.y);
         end;
         if Laby[i,k] = Frei then begin                  // Startpunkt
            Start.x := k; Start.y := i;
         end;
         if (i=1) or (i=Max) or (k=1) or (k=Max) then    // Ausgang
            if (Code = Frei) then Laby[i,k] := Aus;
      end;
    end;
end;

procedure Markieren(Code,i,k: Integer);
{ Markiert ein Wegfeld mit einem Kreis, dessen Farbe durch Code bestimmt ist }
var M : TPoint;
begin
  M.x := Round((k-1)*a + a/2); M.y := Round((i-1)*b + b/2);
  with Form1.Image1.Canvas do begin
       Brush.Color := Farbe[Code];
       Pen.Color   := Farbe[Code];
       if (k = Start.x) and (i = Start.y) then begin
          Brush.Color := Farbe[Aus];
          Pen.Color   := Farbe[Aus];
       end;
       Ellipse(M.x-r,M.y-r,M.x+r,M.y+r);
  end;
end;

procedure Weg_Suchen(i,k: Integer);
{ Rekursives Suchen des Ausgangs im Labyrinth ab der Position (i/k) }
begin
  Warten(Zeit);
  if GetAsyncKeyState(VK_ESCAPE) <> 0 then BreakFlag := True;
  if BreakFlag then Exit;

  case Laby[i,k] of
       Spur: begin end;                 // nichts tun
       Wand: begin end;                 // nichts tun
       Aus : begin                      // Ausgang erreicht
               Markieren(Aus,i,k);
               Inc(Len);
               WegStr  := IntTostr(Len); // Weglänge aufzeichnen
               AusFlag := True;
             end;
```

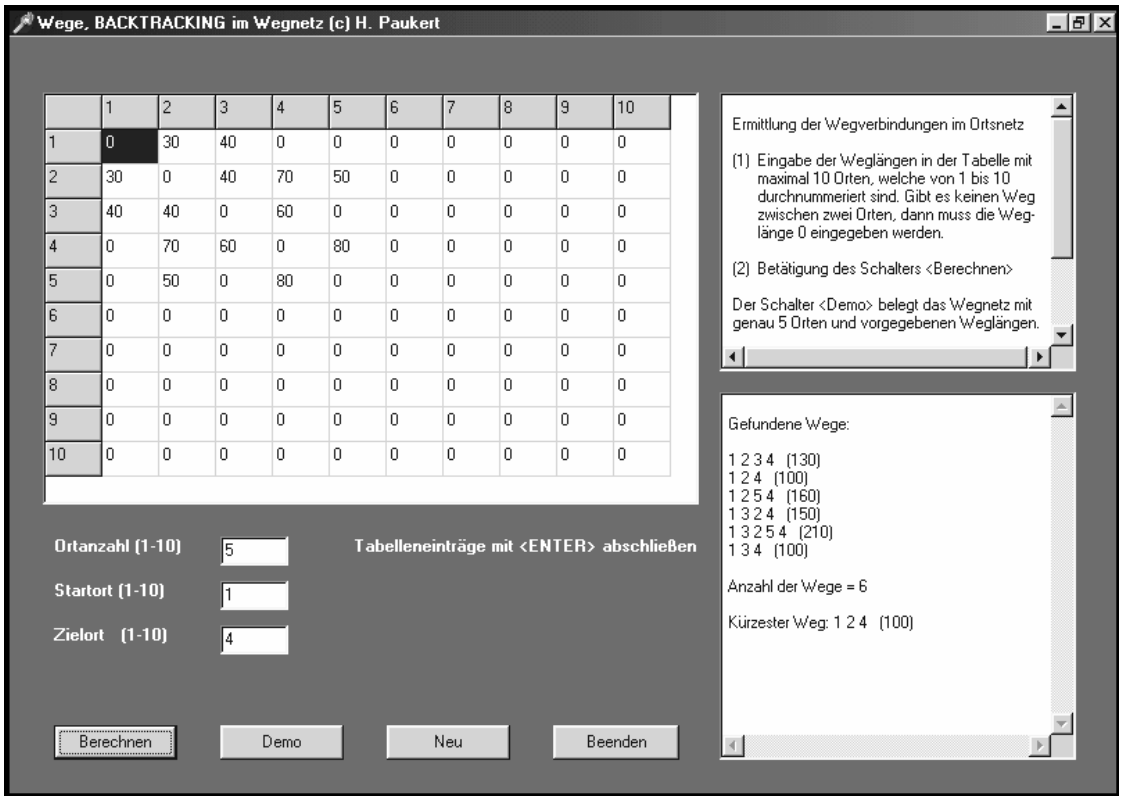
```
       Frei: begin                        // nächstes freie Wegfeld

               if AusFlag then Exit;      // EIN Weg in die Freiheit erreicht

               Laby[i,k] := Spur; Markieren(Spur,i,k); Inc(Len);
                                          // Wegspur markieren
               Weg_Suchen(i,k+1);         // Versuch nach rechts
               Weg_Suchen(i+1,k);         // Versuch nach unten
               Weg_Suchen(i,k-1);         // Versuch nach links
               Weg_Suchen(i-1,k);         // Versuch nach oben
                                          // Wegspur löschen
               If Not AusFlag then begin
                  Laby[i,k] := Frei; Markieren(Frei,i,k); Dec(Len);
               end;

             end;
  end; { of case }
end;

procedure Spur_Loeschen;
{ Markierte Spuren im Labyrinth-Bild löschen }
var i,k : Integer;
begin
  for i := 1 to Max do
      for k := 1 to Max do
          if (Laby[i,k] = Spur) then begin
             Laby[i,k] := Frei;
             Markieren(Frei,i,k);
          end;
end;

procedure FitForm(F :TForm);
// Anpassung des Formulares an die Monitorauflösung
begin
  with F do begin
    if (Screen.Width<>1024) then ScaleBy(Screen.Width,1024);
    if (Font.PixelsPerInch<>120) then ScaleBy(120,Font.PixelsPerInch);
    WindowState := wsMaximized;
  end;
end;

procedure TForm1.FormCreate(Sender: TObject);
{ Formular initialisieren }
begin
  FitForm(Form1);
  XMax := Image1.ClientWidth;
  YMax := Image1.ClientHeight;
  a := XMax / Max;
  b := YMax / Max;
  if a > b then r := Round(b/3) else r := Round(a/3);
end;

procedure TForm1.Image1MouseDown(Sender: TObject; Button: TMouseButton;
                                 Shift: TShiftState; X, Y: Integer);
{ Merken der Mausposition und ein Labyrinthfeld erzeugen/löschen }
begin
  if Button = mbLeft then Erzeugen(Frei,X,Y);
  if Button = mbRight then Erzeugen(Wand,X,Y);
  X0 := X;
  Y0 := Y;
end;

procedure TForm1.Image1MouseMove(Sender: TObject; Shift: TShiftState;
                                 X,Y: Integer);
{ Labyrinth-Erzeugung/Löschung mit horizontalen oder vertikalen Mausbewegungen }
var n: Integer;
begin
  if shift = [ssLeft] then
     if ((X > X0-a) and (X < X0+a)) or
        ((Y > Y0-b) and (Y < Y0+b)) then
              for n := Y0 to Y do Erzeugen(Frei,X,Y);
  if shift = [ssRight] then
     if ((X > X0-a) and (X < X0+a)) or
        ((Y > Y0-b) and (Y < Y0+b)) then
              for n := Y0 to Y do Erzeugen(Wand,X,Y);
end;
```

```
procedure TForm1.Button1Click(Sender: TObject);
{ Informationen in einem Memofeld anzeigen und Geschwindigkeit einstellen}
var S : String;
    Code : Integer;
begin
  Memo2.Visible := False;
  Memo1.Visible := True;
  S := InputBox(' Verzögerung in MilliSekunden (0..1000)','','10');
  val(S,Zeit,Code);
  if (Code < 0) or (Code > 1000) then Zeit := 10;
end;

procedure TForm1.Button2Click(Sender: TObject);
{ Demo-Labyrinth einrichten }
begin
  Label1.Caption := 'Weglänge = ';
  Kopieren;
  Start.y := 20;
  Start.x := 5;
end;

procedure TForm1.Button3Click(Sender: TObject);
{ Neues Labyrinth initialisieren }
begin
  Initialisieren;
end;

procedure TForm1.Button4Click(Sender: TObject);
{ Backtracking starten }
begin
  Label1.Caption := '';
  Len := 0;
  WegStr  := '';
  AusFlag := False;
  BreakFlag := False;
  Spur_Loeschen;

  Weg_Suchen(Start.y,Start.x);

  if AusFlag then Label1.Caption := 'Weglänge = ' + WegStr
             else Label1.Caption := 'Keinen Ausweg gefunden!';
end;

procedure TForm1.Button5Click(Sender: TObject);
{ Matrix in einem Memofeld anzeigen }
var i, k : Integer;
    S, T : String;
begin
  Memo1.Visible := False;
  Memo2.Visible := True;
  Memo2.Clear;
  for i := 1 to Max do begin
      S := '';
      for k := 1 to max do begin
        str(Laby[i,k]:2,T); S := S + T;
      end;
      Memo2.Lines.Add(S);
  end;
end;

procedure TForm1.Button6Click(Sender: TObject);
{ Programm beenden }
begin
   Application.Terminate;
end;

end.
```

Anmerkung: Auf der Begleit-CD befindet sich eine zweite Version des Labyrinthprogrammes "***laby2***". Damit wird das *Problem [P1.2]* gelöst, d.h., es werden im Labyrinth ALLE Wege gesucht und zusätzlich der kürzeste Weg ermittelt.

[03] Backtracking im Wegnetz (*wege*)

Das Programm "***wege***" stellt eine Lösung des erweiterten Wegproblems [P2.2] dar. Die folgende Abbildung zeigt das Formular.



	1	2	3	4	5	6	7	8	9	10
1	0	30	40	0	0	0	0	0	0	0
2	30	0	40	70	50	0	0	0	0	0
3	40	40	0	60	0	0	0	0	0	0
4	0	70	60	0	80	0	0	0	0	0
5	0	50	0	80	0	0	0	0	0	0
6	0	0	0	0	0	0	0	0	0	0
7	0	0	0	0	0	0	0	0	0	0
8	0	0	0	0	0	0	0	0	0	0
9	0	0	0	0	0	0	0	0	0	0
10	0	0	0	0	0	0	0	0	0	0

Für die Menge der Orte mit ihren Verbindungswegen wird im Hauptspeicher eine zweidimensionale Matrix (*array[1..Max,1..Max] of word*) reserviert. Die Konstante *Max*, welche die Gesamtanzahl der Orte bestimmt, ist mit 10 begrenzt. Die einzelnen Orte entsprechen den Zeilen- bzw. Spaltennummern der Matrix (1,2,3, ..., 10). Die Felder der Matrix werden mit den Längen der direkten Verbindungswege zweier Orte belegt, beispielsweise ist *Matrix[x,y]* die Weglänge zwischen Ort ***x*** und Ort ***y***. Der Wert 0 bedeutet, dass es keine direkte Verbindung der beiden Orte gibt. Diese Matrix ist symmetrisch, weil der direkte Weg von ***x*** nach ***y*** derselbe ist, wie der Weg von ***y*** nach ***x***. In der Hauptdiagonale der Matrix stehen nur die Werte 0, weil der Weg von ***x*** nach ***x*** natürlich die Länge 0 hat.

Visualisiert wird die Wege-Matrix (bzw. das Ortsnetz) durch eine entsprechend eingerichtete Tabellen-Komponente (*StringGrid*) im Formular. Hier können die verschiedenen Weglängen eingegeben werden. Die Prozedur *TableToMatrix* kopiert die Tabelleneinträge in die Wege-Matrix. Die Prozedur *MatrixToTable* leistet das Umgekehrte.

Nach der Eingabe der Weglängen in der Tabelle müssen noch die aktuelle Anzahl der Orte (*Anz*), der Startort (*Start*) und der Zielort (*Ziel*) eingegeben werden. Das erfolgt in einfacher Weise über *Edit*-Komponenten.

Die Kernroutine des Programms ist die rekursive Prozedur ***Weg_Suchen***. Sie entspricht in modifizierter Form dem allgemeinen Backtracking-Schema, welches früher bereits erklärt wurde. In einer Wiederholungsschleife werden alle Orte durchlaufen. Nur wenn ein Verbindungsweg zwischen dem Startort und dem aktuellen Ort in der Schleife existiert und der Ort nicht bereits begangen worden ist (was in der Variablen *Menge* festgehalten ist), wird fortgeschritten.

Falls der Zielort erreicht ist, erfolgt eine Ausgabe des Weges. Andernfalls wird die Prozedur neuerlich rekursiv aufgerufen. Das Verfahren ist beendet, wenn alle vorgegebenen Orte durchlaufen sind. Die Ausgabe der zum Zielort führenden Wege erfolgt in der Prozedur *Weg_Ausgeben*. Dort wird auch jede Weglänge überprüft und der Weg mit der kürzesten Länge abgespeichert. Am Ende des Programms wird zusätzlich noch dieser kürzeste Weg ausgegeben. Der Aufruf der Kernroutine *Weg_Suchen* wird über den Schaltknopf <*Berechnen*> ausgelöst.

Mit dem Schaltknopf <*Demo*> wird die Tabelle mit 5 Orten und den entsprechend vorgegebenen Verbindungswegen belegt. Mit dem Schaltknopf <*Neu*> kann die Tabelle initialisiert werden, d.h. alle ihre Felder erhalten den Wert 0 zugewiesen. Mit dem Schaltknopf <*Beenden*> wird schließlich das Programm beendet.

Grundsätzlich erscheint es ratsam, dass vor jeder Eingabe eines neuen Wegnetzes in der Tabelle die Netzstruktur wohl überlegt wird. Dazu kann man die einzelnen Orte als Punkte auf einem Blatt Papier markieren und die gewünschten Verbindungswege als Linien dazwischen einzeichnen. Neben den Linien können auch ihre Weglängen aufgeschrieben werden. Auf diese Weise gewinnt man einen guten Überblick über das Wegnetz und eine Überprüfung des Programmes ist dadurch leicht möglich.

Listing der Unit *"wege_u.pas"*

```
unit wege_u;

{ Backtracking im Wegnetz (c) Herbert Paukert }
{ Suchen aller Lösungswege und Ermittlung des }
{ kürzesten Weges.                            }

interface

uses Windows, Messages, SysUtils, Classes, Graphics,
     Controls, Forms, Dialogs, Grids, StdCtrls;

type
  TForm1 = class(TForm)
    Label1: TLabel;               // Ortsanzahl (1-10)
    Label2: TLabel;               // Startort (1-10)
    Label3: TLabel;               // Zielort (1-10)
    Label4: TLabel;               // Eingaben mit <ENTER> abschließen
    Edit1:  TEdit;                // Eingabe der Ortsanzahl ANZ
    Edit2:  TEdit;                // Eingabe des Startortes START
    Edit3:  TEdit;                // Eingabe des Zielortes ZIEL
    StringGrid1: TStringGrid;     // Tabelle zur Eingabe der Verbindungswege
    Memo1:  TMemo;                // Memo für Hilfsinformationen
    Memo2:  TMemo;                // Ausgabe der gefundenen Wege

    Button1: TButton;             // <Berechnen>
    Button2: TButton;             // <Demo>
    Button3: TButton;             // <Neu>
    Button4: TButton;             // <Beenden>

    procedure Button1Click(Sender: TObject);
    procedure Button2Click(Sender: TObject);
    procedure Button3Click(Sender: TObject);
    procedure Button4Click(Sender: TObject);
    procedure FormCreate(Sender: TObject);
    procedure FormKeyPress(Sender: TObject; var Key: Char);
    procedure StringGrid1KeyPress(Sender: TObject; var Key: Char);
  private { Private declarations }
  public  { Public declarations }
end;

var Form1: TForm1;

implementation
{$R *.DFM}

const Max = 10;                           // maximale Anzahl der Orte
      Anz : Word = 5;                     // aktuelle Anzahl der Orte
```

```
type  TWeg = array[1..255] of Byte;             // Liste der Orte eines Weges
      TMat = array[1..Max,1..Max] of Word;      // Ortsnetz mit Entfernungen

const Matrix : TMat = ((00,30,40,00,00,00,00,00,00,00),   // Ortsnetz,
                       (30,00,40,70,50,00,00,00,00,00),   // initialisiert
                       (40,40,00,60,00,00,00,00,00,00),   // als DEMO!
                       (00,70,60,00,80,00,00,00,00,00),
                       (00,50,00,80,00,00,00,00,00,00),
                       (00,00,00,00,00,00,00,00,00,00),
                       (00,00,00,00,00,00,00,00,00,00),
                       (00,00,00,00,00,00,00,00,00,00),
                       (00,00,00,00,00,00,00,00,00,00),
                       (00,00,00,00,00,00,00,00,00,00));

var   Matrix1: TMat;              // Kopie der DEMO-Matrix

      Weg    : TWeg;              // aktueller Weg
      WegMin : TWeg;              // kürzester Weg
      OrtMin : Word;              // Ortsanzahl im kürzesten Weg
      LenMin : Word;              // Weglänge des kürzesten Weg
      WegAnz : Word;              // Zähler der gefundenen Wege

      Start  : Byte;              // Startort
      Ziel   : Byte;              // Zielort
      Menge  : Set of Byte;       // Hilfsvariable zur Wegmarkierung


procedure MatrixToTable(N: Word);
{ Kopiert das Wegnetz von der Matrix in die Tabelle }
{ mit N als Zeilen- bzw. Spalten-Anzahl             }
var x,y : Integer;
    Zahl: Word;
    S   : String;
begin
  for y := 1 to N do
    for x := 1 to N do begin
      Zahl := Matrix[x,y]; Str(Zahl,S);
      Form1.StringGrid1.Cells[x,y] := S;
    end;
end;

procedure TableToMatrix(N: Word);
{ Kopiert das Wegnetz aus der Tabelle in die Matrix }
{ mit N als Zeilen- bzw. Spalten-Anzahl             }
var x,y : Integer;
    Code: Integer;
    Zahl: Word;
    S   : String;
begin
  for y := 1 to N do
    for x := 1 to N do begin
      S := Form1.StringGrid1.Cells[x,y];
      val(S,Zahl,Code);
      Matrix[x,y] := Zahl;
    end;
end;

procedure TableInit;
{ Initialisiert die Tabelle und alles andere mit Null }
var x,y : Integer;
    Ort : Byte;
begin

  for x := 1 to 10 do
      for y := 1 to 10 do
          Form1.Stringgrid1.Cells[x,y] := '0';
  for Ort := 1 to 255 do begin
      Weg[Ort]    := 0;
      WegMin[Ort] := 0;
  end;

  WegAnz := 0;
  OrtMin := 0;
  LenMin := 32767;  // die kürzeste Weglänge wird anfangs mit einer sehr großen Zahl belegt
  Menge  := [];

end;
```

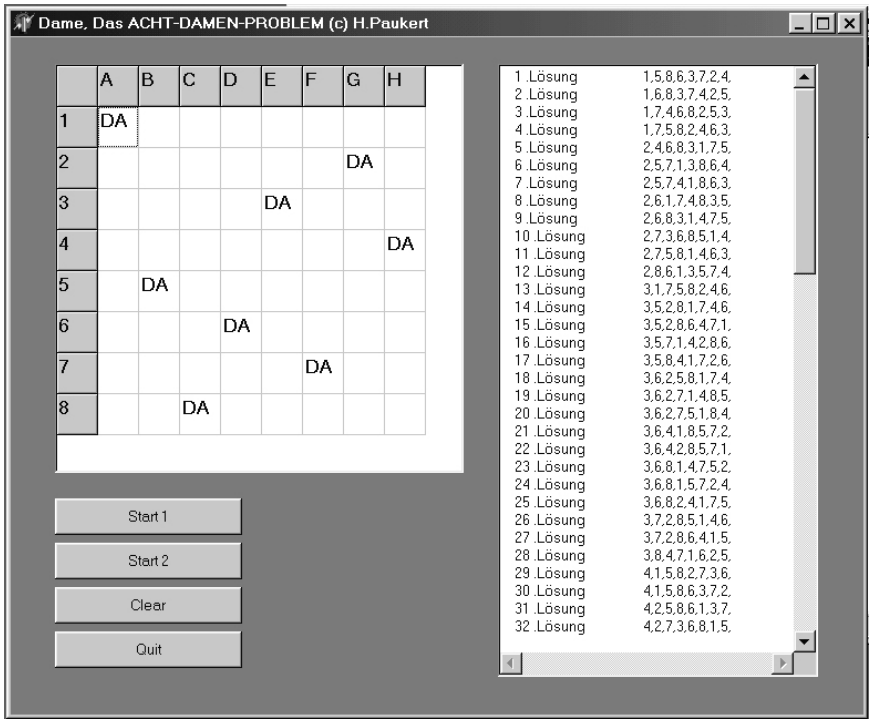
```
procedure FitForm(F :TForm);
// Anpassung des Formulares an die Monitorauflösung
begin
  with F do begin
    if (Screen.Width<>1024) then ScaleBy(Screen.Width,1024);
    if (Font.PixelsPerInch<>120) then ScaleBy(120,Font.PixelsPerInch);
    WindowState := wsMaximized;
  end;
end;


procedure TForm1.FormCreate(Sender: TObject);
{ Formular-Initialisierungen }
var x,y : Integer;
begin
  FitForm(Form1);
  KeyPreview := True;
  with StringGrid1 do begin
    for x := 1 to 10 do Cells[x,0] := IntToStr(x);
    for y := 1 to 10 do Cells[0,y] := IntToStr(y);
  end;
  Matrix1 := Matrix;
  TableInit;
  Anz := 5;
  MatrixToTable(Anz);
  Edit1.Text := IntToStr(Anz);
  Memo2.Clear;
  Memo2.Lines.Add(' ');
  Memo2.Lines.Add(' Gefundene Wege: ');
  Memo2.Lines.Add(' ');
end;


procedure TForm1.FormKeyPress(Sender: TObject; var Key: Char);
{ Mit der <Enter>-Taste zur nächsten Eingabe springen }
begin
  if (Sender=Edit1) and (Key=#13) then Edit2.SetFocus;
  if (Sender=Edit2) and (Key=#13) then Edit3.SetFocus;
  if (Sender=Edit3) and (Key=#13) then Button1.SetFocus;
end;


procedure TForm1.StringGrid1KeyPress(Sender: TObject; var Key: Char);
{ Sichere und symmetrische Dateneingabe in der Tabelle mittels <Enter> }
var S : String;
    Zahl : Word;
    x,y,Code : Integer;
begin
  with StringGrid1 do begin
   if Key = #13 then begin
      if (Col = Row) then Cells[Col,Row] := '0';    // Diagonalfelder sind Null
      x := Col;
      y := Row;
      S := Cells[x,y];
      val(S,Zahl,Code);

      if (Code > 0) then Cells[x,y] := '';

      if (Code = 0) then begin
         Cells[y,x] := S;                           // Symmetrische Felder sind gleich
         x := x + 1;
         if x > 10 then begin
            x := 1;
            y := y + 1;
         end;
         if y > 10 then begin
            y := 1;
         end;
      end;

      Col := x;
      Row := y;
   end;
  end;
end;
```

```
procedure Weg_Ausgeben(N: Word);
{ Ausgabe eines gefundenen Weges mit N als Ortszähler }
var Ort : Byte;
    Len : Integer;
    S   : String;
begin
  Inc(WegAnz);
  Inc(N);
  Weg[N] := Ziel;
  S := ' ';
  Len := 0;

  for Ort := 1 to N-1 do begin
      Len := Len + Matrix[Weg[Ort],Weg[Ort+1]];
      S := S + IntToStr(Weg[Ort]) + ' ';
  end;

  S := S + IntToStr(Weg[N]) + ' ';
  if Len < LenMin then begin
     LenMin := Len;
     OrtMin := N;
     WegMin := Weg;
  end;
  S := S + '  (' + IntToStr(Len) + ')';
  Form1.Memo2.Lines.Add(S);
end;


procedure Weg_Suchen(N: Word; Start, Ziel: Byte);
{ Rekursive Wegbestimmung von Start zu Ziel mit N als Ortszähler }
var Ort : Byte;
begin

  for Ort := 1 to Anz do begin

      if (Matrix[Start,Ort]<>0) and Not (Ort in Menge) then begin
         Menge  := Menge + [Start];
         Weg[N] := Start;
         if Ort <> Ziel then Weg_Suchen(N+1,Ort,Ziel)
                        else Weg_Ausgeben(N);
      end;

      Menge := Menge - [Start];

  end;

end;

procedure TForm1.Button1Click(Sender: TObject);
{ Ortanzahl, Start und Ziel eingeben   }
{ Wege rekursiv ermitteln und ausgeben }
var Ort : Byte;
    A,S : String;
begin
  WegAnz := 0;
  OrtMin := 0;
  LenMin := 32767;
  Menge  := [];
  Anz   := StrToInt(Edit1.Text);
  Start := StrToInt(Edit2.Text);
  Ziel  := StrToInt(Edit3.Text);
  TableToMatrix(Anz);
  Weg_Suchen(1,Start,Ziel);
  A := ' Anzahl der Wege = ' + IntToStr(WegAnz);
  Memo2.Lines.Add(' ');
  Memo2.Lines.Add(A);
  Memo2.Lines.Add(' ');
  S := ' Kürzester Weg: ';
  if WegAnz > 0 then begin
     for Ort := 1 to OrtMin do S := S + IntToStr(WegMin[Ort])+ ' ';
     S := S + '  (' + IntToStr(LenMin) + ')';
     Memo2.Lines.Add(S);
  end;
  Memo2.Lines.Add(' ');
end;
```

```
procedure TForm1.Button2Click(Sender: TObject);
{ Wegnetz mit Demo-Matrix initialisieren }
begin
  TableInit;
  Anz := 5;
  Matrix := Matrix1;
  MatrixToTable(Anz);
  Edit1.Text := IntToStr(Anz);
  Edit2.Text := '';
  Edit3.Text := '';
  Memo2.Clear;
  Memo2.Lines.Add(' ');
  Memo2.Lines.Add(' Gefundene Wege: ');
  Memo2.Lines.Add(' ');
  Edit2.SetFocus;
end;


procedure TForm1.Button3Click(Sender: TObject);
{ Wegnetz völlig neu (d.h. mit 0) initialisieren }
begin
  TableInit;
  Edit1.Text := '';
  Edit2.Text := '';
  Edit3.Text := '';
  Memo2.Clear;
  Memo2.Lines.Add(' ');
  Memo2.Lines.Add(' Gefundene Wege: ');
  Memo2.Lines.Add(' ');
  StringGrid1.SetFocus;
end;


procedure TForm1.Button4Click(Sender: TObject);
{ Programm beenden }
begin
  Application.Terminate;
end;


end.
```

[04] Das Acht-Damen-Problem (*dame*)

Acht Damen sind auf einem leeren Schachbrett so aufzustellen, dass keine Dame eine andere Dame bedroht. Im Programm "***dame***" wird dieses Problem gelöst.



Interessant ist, dass dieses Problem bereits 1850 von dem berühmten Mathematiker C.F. GAUSS untersucht, jedoch nicht vollständig gelöst worden ist. Wie alle ähnlichen Probleme dieser Art, ist auch diese Aufgabe analytisch durch Berechnungsformeln nicht zu lösen. Auch hier können BACKTRACKING-Algorithmen erfolgreich angewandt werden. Ihre exakte Durchführung und Protokollierung erfordert großen Aufwand an Sorgfalt und Geduld. Aber genau diese Merkmale besitzen Computer in einem wesentlich höheren Ausmaß als Menschen. Erst durch den Einsatz von Computern gewinnen solche Algorithmen an Bedeutung. Sie werden u.a. auch bei der Programmierung von Spielstrategien benutzt.

Für die Lösung des vorliegenden Problems benötigt man zunächst die elementare Schachregel, dass die Dame alle jene Figuren bedroht, die sich auf dem Schachbrett in derselben Spalte, Zeile oder Diagonale befinden. Daraus folgt, dass jede Spalte höchstens eine Dame enthalten kann. Es sei I der Spaltenindex einer Dame und X[I] speichert den Zeilenindex J der I-ten Dame. Dabei gilt: (1 <= X[I] <= 8). J sei der Zeilenindex, sodass mit (I,J) jedes Feld am Schachbrett lokalisiert wird. Jedes solche Feld ist der Schnittpunkt einer Spalte mit einer Zeile, aber auch von zwei Diagonalen mit den Richtungen / und \. Es gilt dabei für alle Felder in der K-ten /-Diagonale, dass (I+J) konstant ist, während für die K-te \-Diagonale (I-J) konstant bleibt. Die booleschen Arrays D1[K] mit (2 <= K <= 16) und D2[K] mit (-7 <= K <= +7) geben an, dass die jeweilige Diagonale frei von einer Dame ist. Dabei ist K entweder (I+J) oder (I-J), je nach Diagonalenrichtung. Die zusätzliche, boolesche Variable ERFOLG signalisiert einen erfolgreichen Aufstellungsversuch einer Dame.

Datenstruktur

```
I: Integer;          {Spaltenindex}
J: Integer;          {Zeilenindex}
ERFOLG: Boolean;     {nur True, wenn Dame erfolgreich platziert}
```

```
X : array[ 1.. 8]  of Integer     {Zeilenposition der I-ten Dame}
Z : array[ 1.. 8]  of Boolean;    {nur True, wenn J-te Zeile frei}
D1: array[ 2..16]  of Boolean;    {nur True, wenn K-te /-Diagonale frei}
D2: array[-7.. 7]  of Boolean;    {nur True, wenn K-te \-Diagonale frei}
```

Der Befehl "*Setze Dame*" kann durch folgende Anweisungen ausgeführt werden:

```
X[I]    := J;
Z[J]    := False;
D1[I+J] := False;
D2[I-J] := False;
```

Der Befehl "*Entferne Dame*" kann durch folgende Anweisungen ausgeführt werden:

```
X[I]    := J;
Z[J]    := True;
D1[I+J] := True;
D2[I-J] := True;
```

Die Bedingung "*Feld frei*" heißt dann, dass das Feld (I,J), die I-te Spalte, die J-te Zeile und die zwei zugehörigen Diagonalen von Damen unbesetzt sind. Es gilt: *Z[J] and D1[I+J] and D2[I-J]*.

Mit dieser Datenstruktur lässt sich ein allgemeines Lösungsverfahren formulieren:

```
Modul VERSUCH (mit Dame in Spalte I);
   Initialisiere die Zeilenposition der Dame (J:=0);
   Wiederhole bis (ERFOLG erreicht) oder (alle ZEILEN durchlaufen)
      Gehe zur nächsten Zeile (J:=J+1);
      Initialisiere Erfolglosigkeit (ERFOLG:=False);
      Wenn FELD FREI, dann
           SETZE DAME auf Position (I/J);
           wenn (I < 8), dann
                rekursiver Aufruf von VERSUCH (mit Dame in Spalte I+1);
                wenn (kein ERFOLG), dann ENTFERNE DAME
                                    andernfalls markiere den ERFOLG;
   Ende der Wiederholung;
Ende des Moduls;
```

Das Programm realisiert genau diesen Algorithmus. Als Resultat erhält man unten stehende Aufstellungen für die Damen. Die Zahlen in der Klammer sind die Spalten (I) und Zeilen (J=X[I]).

1-te Dame: (1/1), 2-te Dame: (2/5), 3-te Dame: (3/8), 4-te Dame: (4/6)
5-te Dame: (5/3), 6-te Dame: (6/7), 7-te Dame: (7/2), 8-te Dame: (8/4)
Diese Lösung in Kurzschreibweise lautet: (1,5,8,6,3,7,2,4).

Interessant ist eine Erweiterung des Problems, wobei nicht nur EINE Lösung gefunden werden soll, sondern ALLE Lösungen. Diese Erweiterung liefert 92 Lösungen und könnte folgendermaßen aussehen:

```
Modul FINDEN (mit Dame in Spalte I);
   Wiederhole ZEILE J von 1 bis 8
      Wenn FELD FREI, dann
           SETZE DAME auf Position (I/J);
           wenn (I < 8), dann
                 rekursiver Aufruf von VERSUCH (mit Dame in Spalte I+1),
                 andernfalls Ausgabe der Lösung;
           ENTFERNE DAME;
   Ende der Wiederholung;
Ende des Moduls;
```

Die ersten drei Lösungen des erweiterten Damenproblems seien angeführt. (Die Zahlen in der Klammer sind nur die Zeilenpositionen der Damen, welche fortlaufend in den Spalten 1, 2 bis 8 positioniert sind): ..... (1,5,8,6,3,7,2,4); (1,6,8,3,7,4,2,5); (1,7,4,6,8,2,5,3) .....

```
unit dame_u;
// Dame, das ACHT-DAMEN-PROBLEM (c) H.Paukert

interface
uses Windows, Messages, SysUtils, Classes, Graphics,
     Controls, Forms, Dialogs, StdCtrls, Grids;
type
  TForm1 = class(TForm)
    StringGrid1: TStringGrid;
    Memo1: TMemo;
    Button1: TButton;
    Button2: TButton;
    Button3: TButton;
    Button4: TButton;
    procedure Button1Click(Sender: TObject);
    procedure Button3Click(Sender: TObject);
    procedure Button2Click(Sender: TObject);
    procedure FormActivate(Sender: TObject);
    procedure Button4Click(Sender: TObject);
  private { Private-Deklarationen }
  public  { Public-Deklarationen }
  end;

var  Form1: TForm1;

implementation
{$R *.DFM}

CONST DAME = 'DA';
      ZEIT : INTEGER = 20;

VAR   S : STRING;
      N : INTEGER;        { VERSUCHS-ZÄHLER }
      I : INTEGER;        { SPALTENINDEX FÜR DIE I-TE DAME }
      ERFOLG : BOOLEAN;   { TRUE = ERFOLGREICHE PLATZIERUNG, SONST FALSE }
      X : ARRAY[ 1.. 8] OF INTEGER;   { ZEILENPOSITION DER I-TEN DAME }
      Z : ARRAY[ 1.. 8] OF BOOLEAN;   { TRUE: J-TE ZEILE FREI, SONST FALSE }
      D1: ARRAY[ 2..16] OF BOOLEAN;   { K-TE /-DIAGONALE FREI, SONST FALSE }
      D2: ARRAY[-7.. 7] OF BOOLEAN;   { K-TE \-DIAGONALE FREI, SONST FALSE }

procedure Pause(Zeit: Integer);
// Pause in MilliSekunden
var Zeit1: Integer;
begin
  Zeit1 := GetTickCount;
  repeat
    Application.ProcessMessages;
  until (GetTickCount - Zeit1 > Zeit);
end;

procedure VERSUCH(I: Integer; var ERFOLG: Boolean);
// Rekursives Unterprogramm zum Suchen EINER Lösung
var  J : Integer;
begin
   J := 0;
   repeat
      J := J + 1;
      ERFOLG := FALSE;
      if (Z[J] AND D1[I+J] AND D2[I-J]) then
         begin
           X[I] := J; Z[J] := FALSE; D1[I+J] := FALSE; D2[I-J] := FALSE;
           Form1.StringGrid1.Cells[I,(X[I])] := Dame;
           Pause(ZEIT);
           if (I < 8) then
              begin
              VERSUCH(I+1,ERFOLG);
              if NOT ERFOLG then
                 begin
                   Z[J] := TRUE; D1[I+J] := TRUE; D2[I-J] := TRUE;
                   Form1.StringGrid1.Cells[I,(X[I])] := ' ';
                   Pause(ZEIT);
                 end
              end
           else ERFOLG := TRUE;
         end;
   until ERFOLG or (J=8);
end;
```

```
procedure FINDEN(I: integer);
// Rekursives Unterprogramm zum Finden ALLER Lösungen
var  J,K : integer;
begin
   for J := 1 to 8 do begin
      if Z[J] and D1[I+J] and D2[I-J] then
         begin
           X[I] := J; Z[J] := FALSE; D1[I+J] := FALSE; D2[I-J] := FALSE;
           if (I < 8) then FINDEN(I+1)
                      else begin
                           N := N + 1;
                           S := '    ' + IntToStr(N) + ' .Lösung' + #9;
                           for K := 1 to 8 do S := S + IntToStr(X[K])+',';
                           Form1.Memo1.Lines.Add(S);
                      end;
           Z[J] := TRUE; D1[I+J] := TRUE; D2[I-J] := TRUE;
         end;
   end;
end;

procedure TForm1.FormActivate(Sender: TObject);
// Initialisierungen
var x,y: Integer;
begin
  if (Screen.Width<>1024) then ScaleBy(Screen.Width,1024);
  if (Font.PixelsPerInch<>120) then ScaleBy(120,Font.PixelsPerInch);
  WindowState := wsMaximized;
  With Form1.StringGrid1 do begin
    For x := 1 to 9 do Cells[x,0] := Char(64+x);
    For y := 1 to 9 do Cells[0,y] := IntToStr(y);
  end;
end;

procedure TForm1.Button1Click(Sender: TObject);
// EINE Lösung suchen
var i : Integer;
begin
  for i := 1 to 8 do Z[i] := True;
  for i := 2 to 16 do D1[i] := True;
  for i := -7 to 7 do D2[i] := True;
  ERFOLG := False;
  VERSUCH(1,ERFOLG);
end;

procedure TForm1.Button2Click(Sender: TObject);
// ALLE Lösungen finden
var i : Integer;
begin
  Form1.Memo1.Clear;
  for i := 1 to 8 do Z[i] := True;
  for i := 2 to 16 do D1[i] := True;
  for i := -7 to 7 do D2[i] := True;
  N := 0;
  FINDEN(1);
end;

procedure TForm1.Button3Click(Sender: TObject);
// Felder löschen
var x,y : Integer;
begin
  Memo1.Clear;
  With Form1.StringGrid1 do begin
    For x := 1 to 8 do
      For y := 1 to 8 do
        Cells[x,y] := ' ';
  end;
end;

procedure TForm1.Button4Click(Sender: TObject);
// Programm beenden
begin
  Application.Terminate;
end;

end.
```