Fachhochschule Köln
Campus Gummersbach
FB Informatik

Algorithmische Anwendung

Gruppe : **C-Blau**

Name : **Youssef Aghigha (11024992)**

Youssef Barrou (11022119)

Backtracking Algorithmus:

Überblick

Manchmal muss man einen Schritt zurückgehen, um 2 Schritte nach vorn zu kommen. Nach diesem Prinzip arbeitet das Backtracking-Verfahren.

- ***Man führt einen Algorithmus so lange aus, bis man an eine Grenze stößt.***

- ***Ist das der Fall, kehrt man zum letzten Schritt zurück und testet einen anderen Folgeschritt.***

- ***Versuche, eine gültige Teillösung auf dem Weg zum Ergebnis zu finden.***
- ***Baue auf diese Teillösung den restlichen Weg zum Ziel auf.***

- ***Ist das nicht möglich, versuche eine andere Teillösung zu finden.***

Eine Teillösung kann man sich wie einen Knoten in einem Baum vorstellen. Jeder Knoten hat eine gewisse Anzahl von Ästen, die zu weiteren Knoten führen können. An jedem Ast hängt ein weiterer Baum, welcher mögliche Folgeschritte symbolisiert.

Deshalb ist Backtracking:

- ist eine **systematische** Suchstrategie und findet deshalb immer eine **optimale Lösung**, sofern vorhanden, und sucht höchstens einmal in der gleichen „Sackgasse“
- ist einfach zu implementieren mit **Rekursion**
- macht eine **Tiefensuche** im Lösungsbaum
- hat im schlechtesten Fall eine exponentielle **Laufzeit $O(k^n)$** und ist deswegen primär für kleine Probleme geeignet
- erlaubt Wissen über ein Problem in Form einer **Heuristik** zu nutzen, um den Suchraum einzuschränken und die Suche dadurch zu beschleunigen

Pseudocode für das Backtracking im Baumstil

1. Aktueller Knoten des Baumes ist die Wurzel
2. Prüfe, ob der Knoten Kinder besitzt
    a. Wenn ja: wiederhole Schritt 2 für die Wurzel des linksten (unbesuchten) Teilbaumes
    b. Sonst: prüfe, ob der Knoten die Lösung des Problems repräsentiert
        i. Wenn ja: Beende Algorithmus
        ii. Sonst: Prüfe, ob der aktuelle Knoten einen Elternknoten besitzt
            1. Wenn ja: Wiederhole Schritt 2 für den Elternknoten
            2. Sonst: Beende (keine Lösung)

Wobei die Knoten eine Teillösung repräsentieren und jeder Ast, der von diesem Knoten ausgeht, einen möglichen Folgeschritt. Jedes Backtracking-Problem lässt sich auf diese Art und Weise darstellen.

Pseudocode für rekursives Backtracking

1. Rekursionsanker:
Gibt es keinen möglichen Folgeschritt, dann ist der Algorithmus beendet
Ob eine Lösung gefunden wurde, hängt davon ab, ob die aktuelle Teillösung das Problem löst

2. Rekursionsschritt:
Suche Rekursiv eine Lösung, ausgehend vom nächsten möglichen Folgeschritt.
a. Existiert keine Lösung: Wiederhole Schritt 2
b. Sonst: Gesamtlösung = Bisherige Lösung + Folgeschritte

Das Backtracking-Verfahren ist schneller als die Auswertung aller möglichen Permutationen von Lösungswegen.
Begründung hierfür ist die Tatsache, dass Sackgassen vorzeitig abgebrochen werden können.

<u>Das 8-Damen Problem</u> :

Das 8-Damen-Problem ist ein klassisches Schachproblem

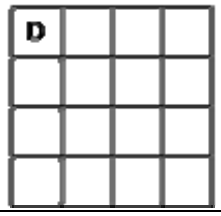
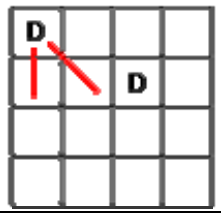
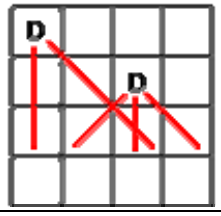
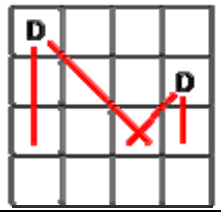
n Damen sollen auf einem nxn-Schachbrett so hingestellt werden, dass sie sich gegenseitig nicht bedrohen. Eine Dame kann beim Schach senkrecht, waagerecht und diagonal so weit ziehen, wie sie will.

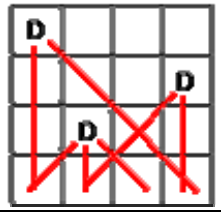
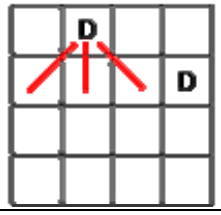
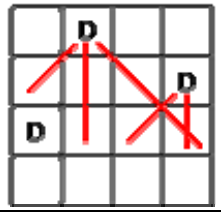
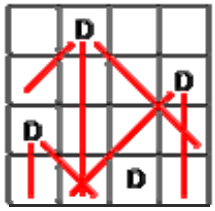
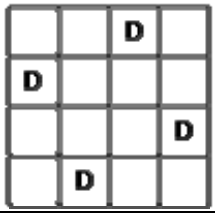
Über das Problem wird berichtet:

"*Dieses Problem tauchte zum ersten Mal 1848 in einer Schachzeitung auf, die von der Berliner Schachgesellschaft herausgegeben wurde. Sie stand auch in der "Illustrierten Zeitung" vom 1. Juni 1850, einer allgemeinen Zeitschrift unter der Rubrik Schach. Dadurch fand sie große Leserschaft.... Es gab aber nun einen Leser (Dr. Nauk), der alle Lösungen gefunden hatte und dieser Mann war von Geburt an blind. der Mathematiker Gauß hatte bis zu diesem Zeitpunkt erst 72 gefunden*"

*Es gibt genau **92** verschiedene Lösungen*.

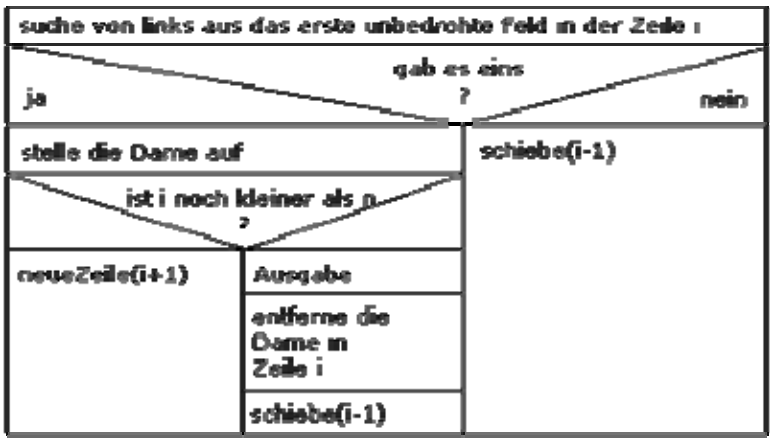
Um den Algorithmus zu verdeutlichen, fangen wir mit 4 Damen auf einem 4x4-Schachbrett an.
(Beim 1x1-Brett ist die Lösung trivial, beim 2x2- und 3x3-Brett läßt sich leicht die Unlösbarkeit zeigen.)

	Wir beginnen mit einer Dame in der ersten Zeile und ersten Spalte. In der ersten Zeile kann nun keine Dame mehr gesetzt werden, da sie bedroht wäre. So kann in jeder Zeile und jeder Spalte nur eine Dame stehen, wir setzen unsere Versuche also in der zweiten Zeile fort.
	Spalte 1 und 2 sind hier bedroht, wir setzen die nächste Dame also in die 3. Spalte.
	Die beiden gesetzten Damen bedrohen nun alle Felder der dritten Zeile, wir gehen eine Zeile zurück und setzen die Dame ein Feld weiter nach rechts.
	Diese Feldbelegung lässt nun eine Dame in Zeile 3 zu, wir setzen sie in die 2. Spalte und prüfen die Bedrohungen in der 4. Zeile.

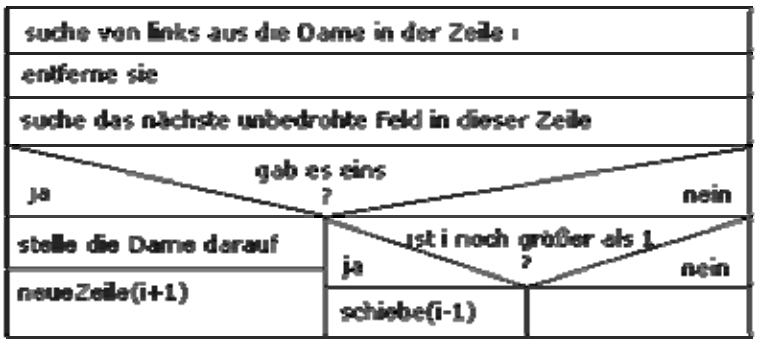
	Leider sind nun in der 4. Zeile alle Felder bedroht, keine Dame ist mehr setzbar. Wir ziehen uns eine Zeile zurück und suchen dort ein weiter rechts liegendes, nicht bedrohtes Feld. Dies gibt es nicht, wir gehen wiederum eine Zeile zurück und suchen einen weiter rechts liegenden Platz für die Dame in der zweiten Zeile. Sie steht schon am rechten Rand, also gehen wir in die erste Zeile und suchen einen neuen Platz für die erste Dame. Das ist leicht, noch kein Feld ist bedroht.
	Diese Setzung lässt uns in der zweiten Zeile nur die 4. Spalte frei. Dorthin setzen wir die Dame und prüfen die Setzmöglichkeiten in der nächsten, der dritten Zeile.
	Wir finden sofort eine nichtbedrohte Position in der ersten Spalte. Nun versuchen wir die 4.Zeile.
	Die dritte Spalte in der 4. Zeile bleibt unbedroht, wir haben eine Lösung gefunden. Ein Teilziel ist erreicht, es sollen aber möglichst alle Lösungen gefunden werden. Wir entfernen die letzte Dame und suchen einen alternativen Platz in der 4. Zeile, den gibt es nicht. Nun ziehen wir uns eine Zeile zurück und versuchen dort einen unbedrohten, anderen Platz für die Dame zu finden, auch dort vergeblich. So landen wir schnell wieder in der ersten Zeile und rücken dort unsere Dame ein Feld weiter, prüfen dann die Möglichkeiten in der nächsten, also zweiten Zeile, dann in der dritten und vierten.
	Wir erhalten die nebenstehende, zweite Lösung, wenden das gleiche Verfahren erneut an, haben aber keine weiteren Erfolge.

Algorithmen zum n-Damen-Problem

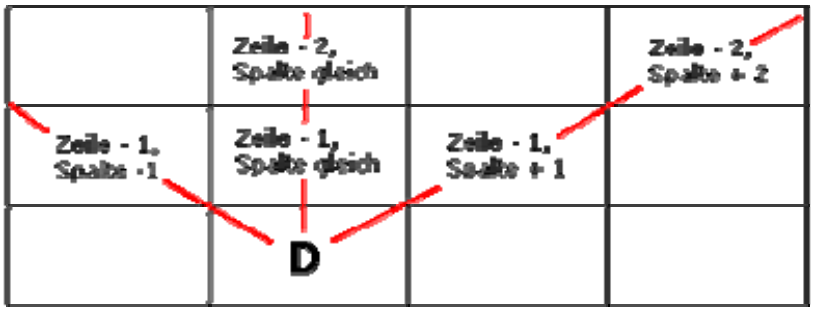
Wir gehen davon aus, dass dies ein Unterprogramm "neueZeile" mit dem Eingabeparameter i, der Zeilennummer, sein wird, das zunächst mit neueZeile(1) aufgerufen wird und sich rekursiv durch die Zeilen vorarbeitet.
Der Algorithmus "schiebe" sucht in einer Zeile die Dame, löscht sie und sucht einen neuen unbedrohten Platz weiter rechts für sie. Ist dies möglich, setzt er sie dorthin und ruft neueZeile mit der folgenden Zeilennummer auf, sonst übergibt er das Verschieben der Dame an die Zeile darüber.



Nun zum Algorithmus von "schiebe". Auch hier wird die Zeilennummer als Parameter i übergeben.



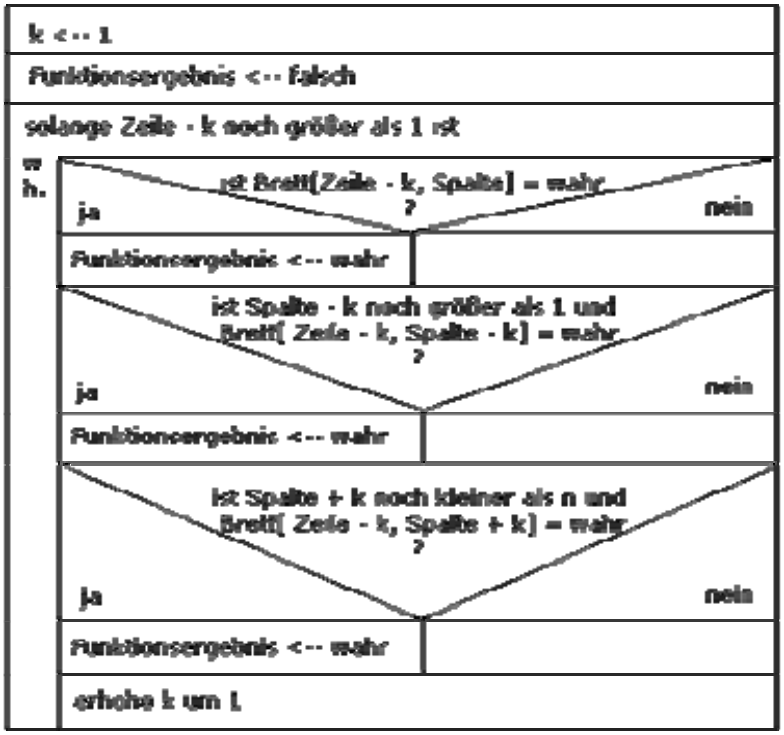
Nun fehlt nur noch die Bestimmung der bedrohten Felder. Da wir in jede Zeile nur eine Dame setzen können, reicht es aus, von dem aktuellen Feld aus eine Dame in der gleichen Spalte zu finden oder in den Diagonalen nach links und rechts oben.



Die gleiche Spalte ist einfach: Spaltennummer bleibt, die Zeilenzahl wird jeweils um eins erniedrigt, bis die erste Zeile erreicht ist. Bei der Diagonalen nach links oben wird jeweils die Spalten- und Zeilennummer eins erniedrigt, dabei wird kontrolliert, dass man die Brettgrenzen nicht überschreitet (Spalte und Zeile mindestens 1). Bei der Diagonalen nach rechts oben entsprechend, Zeilennummer erniedrigen, Spaltennummer erhöhen.

Nur wie erkennen wir eine Dame?
Spätestens hier müssen wir uns Gedanken über die Datenstruktur machen. Die einfachste Möglichkeit ist ein Feld aus Wahrheitswerten, keine Dame ist eine Belegung mit "falsch", Dame gesetzt ändert den Feldinhalt auf "wahr".

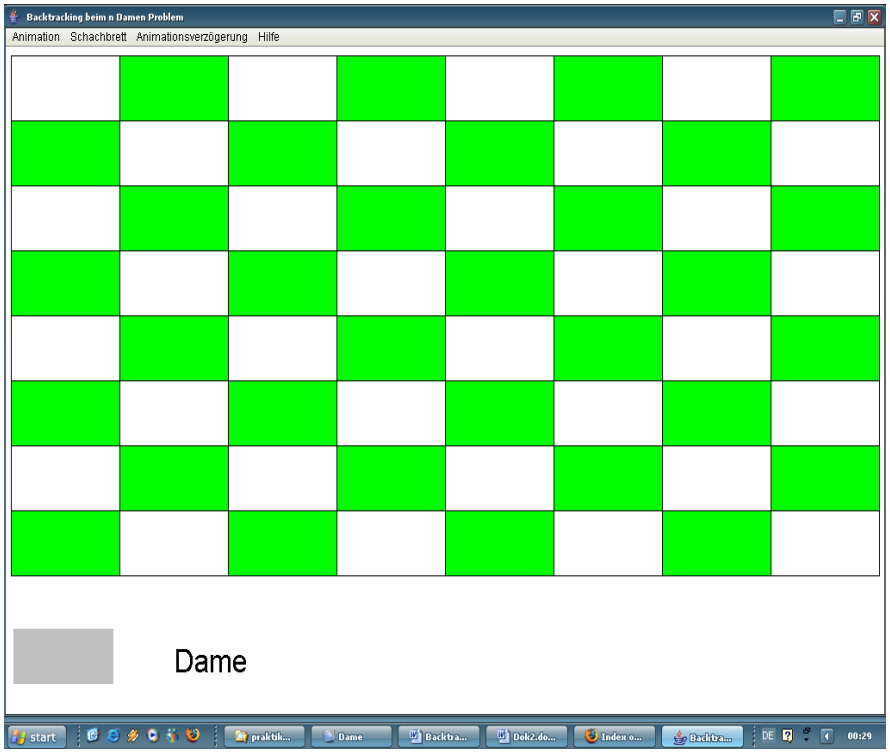
Wir entscheiden uns für eine Funktion "bedroht", die einen Wahrheitswert "wahr" liefert, falls das mit den ganzzähligen Parametern Zeile und Spalte übergebene, potentielle Damenfeld von den bisher gesetzten Damen bedroht ist.



Java-Code zum Einsatz vom Backtracking im 8-Damen-Problem:

```
// Konstante: dame[Spalte] = Zeile = UNDEF, wenn die Spalte keine Dame hat
final int UNDEF = -1;

boolean FindeLoesung(int spalte, DameLsg loesung) {
  // spalte      = aktuelle Schrittzahl und zugleich aktuelle Spalte
  // loesung     = Referenz auf bisherige Teil-Lösung
  int zeile = -1;

  // while(es gibt noch neue Teil-Lösungsschritte)
  while (zeile != n-1) {

 // Wähle einen neuen Teil-Lösungsschritt schritt;
    zeile++;  // Zeilen von oben nach unten der Reihe nach ausprobieren

    // Visualisierung
    visualisiereFindeLoesung(spalte,loesung, spalte, zeile, 4);

    // if (schritt ist gültig), d.h. Dame (zeile, spalte) wird nicht bedroht
    if (loesung.z[zeile] && loesung.d1[zeile+spalte] && loesung.d2[zeile-spalte+n-1]) {

      // Erweitere loesung um schritt
      // Platziere Dame auf (zeile, spalte)
      loesung.dame[spalte]       = zeile;
      loesung.z[zeile]           = false;
      loesung.d1[zeile + spalte]     = false;
      loesung.d2[zeile - spalte  + n - 1] = false;

      // Visualisierung
      visualisiereFindeLoesung(spalte,loesung, spalte, zeile, 1);

      // if (loesung noch nicht vollständig)
      if (spalte != n-1) {

        // rekursiver Aufruf von FindeLoesung
        if (FindeLoesung(spalte+1, loesung)) {

          // Lösung gefunden
          return true;
        }
          else {

          // Wir sind in einer Sackgasse:
          // Mache schritt rückgängig: track back
          loesung.dame[spalte]  = UNDEF;
          loesung.z[zeile]  = true;
          loesung.d1[zeile + spalte]  = true;
          loesung.d2[zeile - spalte  + n - 1] = true;

          // Visualisierung
          visualisiereFindeLoesung(spalte,loesung, spalte, zeile, 2);
        }
      } else return true; // Lösung gefunden -> fertig
    }
  }
  return false; // keine Lösung gefunden
}
```

Screenshots:



Neues Schachbrett

Endaufstellung nach der Lösung

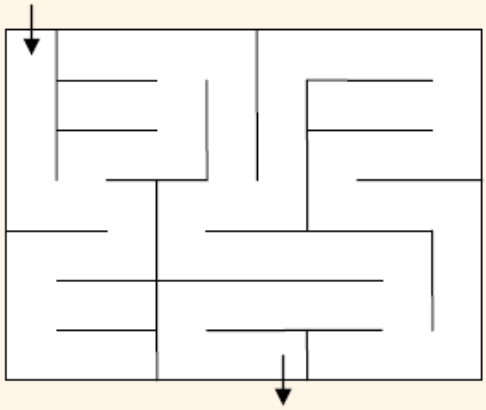
<u>Das Labyrinth-Problem</u> :

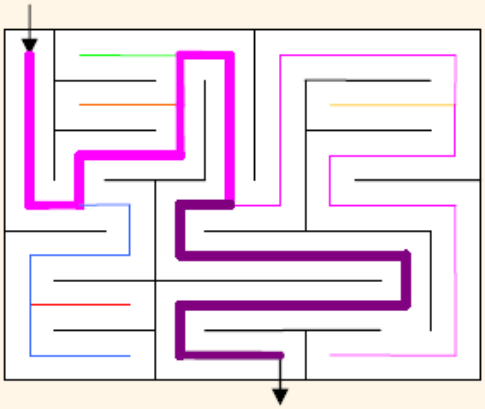
Implementierung einen auf Backtracking beruhenden Algorithmus, der feststellt, ob es in einem Labyrinth einen Weg zum Ausgang gibt und diesen gegebenenfalls im Labyrinth markiert. Am Schluss soll das Labyrinth mit eingezeichnetem Weg mittels GUI ausgegeben werden.

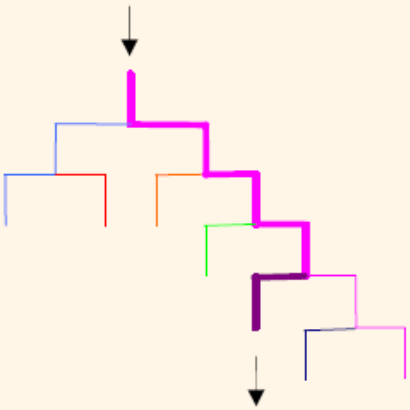
Das Backtrackingverfahren (Rückverfolgung) findet seinen Ursprung in der **griechischen Mythologie**. Der Sage nach gab Ariadne (die Tochter des Königs Minos von Kreta) Theseus (Sohn des Aigeus, Herrscher von Athen), der den Minotaurus im Labyrinth ihres Vaters töten wollte, ein Wollknäuel mit, den Ariadnefaden. Theseus "markierte" damit alle durchwanderten Wege und fand nach erfolgreichem Kampf wieder zu Ariadne zurück, indem er den Faden einfach zurückverfolgte (wieder aufwickelte).

Idee hierbei ist es, eine Traversierungsart zu finden, über die man sicher den Ausgang aus einem Labyrinth findet. Der Algorithmus ist trivial, aber ein schönes Beispiel für das Backtracking.

Wir stellen uns die möglichen Wege durch das Labyrinth wie einen Baum vor. Jede Kreuzung ist ein Knoten und jede mögliche Richtung ein Ast an diesem Knoten. In diesem Beispiel habe ich die einzelnen Wege farbig markiert. Nun traversieren wir den Baum nach einem festen Algorithmus. Irgendwann erreichen wir den Ausgang.

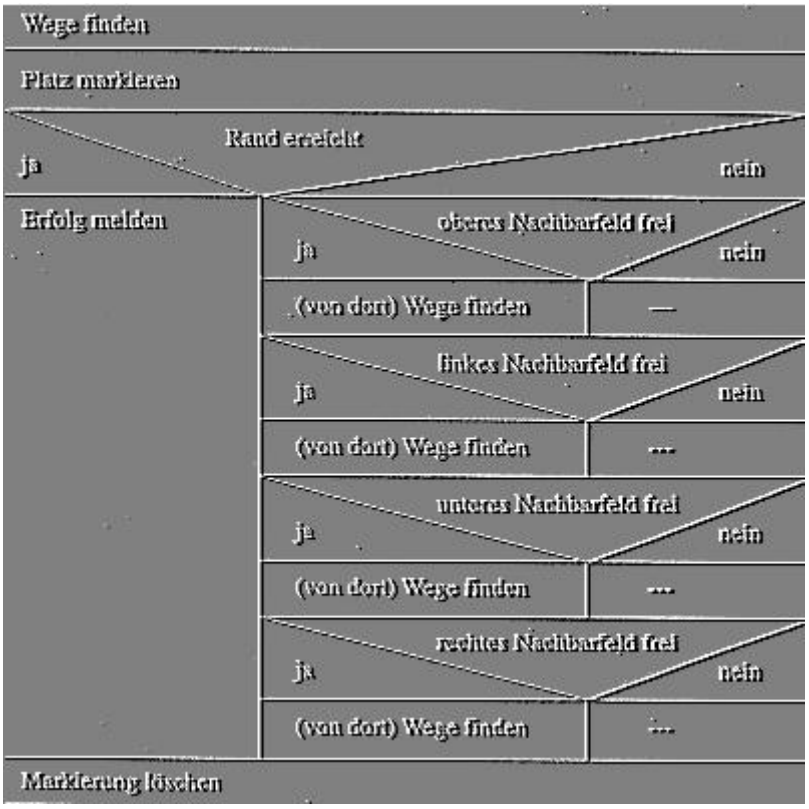






Wir traversieren die Äste von links nach rechts

```
-> L
-> L - L (Sackgasse)
<- L
-> L - R (Sackgasse)
<- L
-> R - L (Sackgasse)
-> R
-> R - R
-> R - R - L (Sackgasse)
<- R - R
-> R - R - R - L (Ausgang)
```



Bei der Tiefensuche werden bei
- max. *k* möglichen Verzweigungen von jeder Teillösung aus und
- einem Lösungsbaum mit maximaler Tiefe von *n*

im schlechtesten Fall $1 + k + k_2 + k_3 + ... + k_n = (k_{n+1} - 1) / (k - 1) =$ **$O(k_n)$** Knoten im Lösungsbaum erweitert.

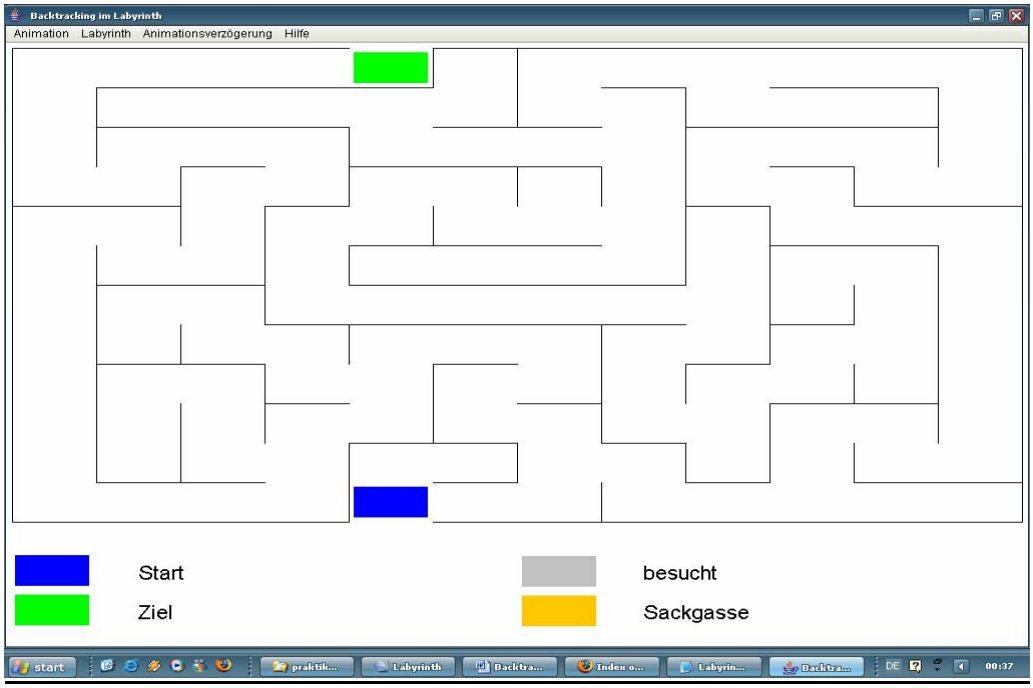
Im Labyrinthbeispiel gab es pro Schritt maximal k=4 mögliche Verzweigungen (oben, rechts, unten oder links). Die maximale Tiefe n entsprach der Weglänge des Lösungsweges.

Java-Code zum Einsatz vom Backtracking im Labyrinth-Problem:

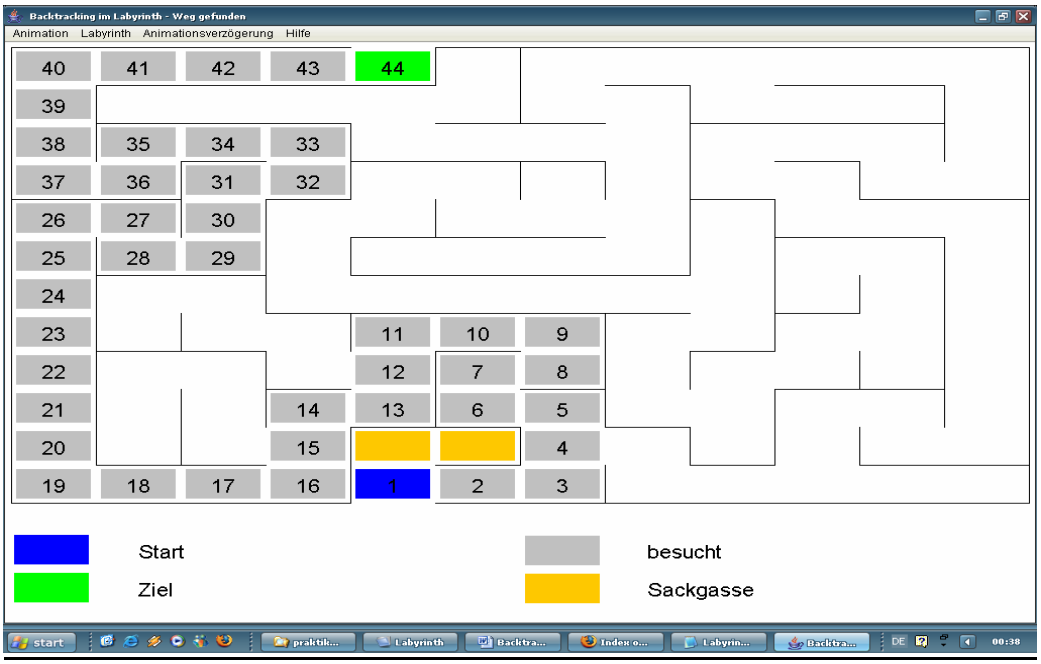
```
final static int LEER = 0, SACKGASSE = -1;  // Feldinformation
final static int[] STEPX = {  0, 1, 0,-1 }; // STEPX,-Y: Schritte
final static int[] STEPY = { -1, 0, 1, 0 }; // in alle vier Richtungen

boolean FindeLoesung(int index, Lsg loesung, int aktX, int aktY) {

  // index       = aktuelle Schrittzahl
  // loesung     = Referenz auf bisherige Teil-Lösung
  // aktX, aktY  = aktuelle Feldposition im Labyrinth
  int schritt = -1;

  // while(es gibt noch neue Teil-Lösungsschritte)
  while (schritt != LINKS) {

    // Wähle einen neuen Teil-Lösungsschritt schritt;
    schritt ++; // Weg nach 1. oben, 2. rechts, 3. unten

    // und 4. links zu erweitern versuchen.
    int neuX  = aktX + STEPX[schritt];
    int neuY  = aktY + STEPY[schritt];

    // Tests, ob schritt gültig ist
    boolean ok = true;

     // Test, ob schritt innerhalb Brett bleibt
    if (neuX < 0 || neuX >= K) ok = false;
    if (neuY < 0 || neuY >= L) ok = false;

    // Test, ob schritt durch Wand führt (sofern innerhalb)
    if (ok && hatWand(aktX,aktY,neuX,neuY,schritt)) ok = false;

     // Test, ob schritt in ein bereits besuchtes Feld führt
    if (ok && loesung.feld[neuX][neuY] != LEER) ok = false;

    // if (schritt ist gültig)
    if (ok) {

      // Erweitere loesung um schritt
      // Markiere neues Feld mit aktueller Schrittzahl
      loesung.feld[neuX][neuY] = index;

      // Visualisierung
      visualisiereFindeLoesung(index,loesung, neuX, neuY, 1);

      // if (loesung noch nicht vollständig)
      if (!ausgangGefunden(neuX, neuY)) {

       // rekursiver Aufruf von FindeLoesung
        if (FindeLoesung(index+1, loesung, neuX, neuY)) {
          // Lösung gefunden
          return true;
        }
          else {
          // Wir sind in einer Sackgasse:
          // Mache schritt rückgängig: track back
          loesung.feld[neuX][neuY] = SACKGASSE;

          // Visualisierung
          visualisiereFindeLoesung(index,loesung, neuX, neuY, 2);
```

```
        visualisiereFindeLoesung(index-1,loesung, aktX, aktY, 3);
       }
     } else return true; // Lösung gefunden -> fertig
   }
  }
  return false; // keine Lösung gefunden
 }
```

Screenshots:



Neues Labyrinth



Endaufstellung nach der Lösung

Das 4-Farben-Problem:

Einleitung

Im Jahre 1852 war der englische Mathematiker Francis Guthrie mit der Aufgabe beschäftigt, eine Karte mit den englischen Grafschaften zu kolorieren. Es war offensichtlich, dass drei Farben nicht ausreichten und man fünf in keinem konstruierten Beispiel brauchte. Die Bedingung dabei war, dass benachbarte Länder farblich unterscheidbar sein sollten.

Francis Guthrie war ein Student der Universität von London und hat das Problem seinem Lehrer (De Morgan) vorgetragen.

De Morgan war nicht in der Lage eine Antwort auf das Problem und hat wiederum die Frage an seinen Kollegen William Rowan Hamilton in Dublin weiter geleitet :

A student of mine asked me today to give him a reason for a fact which I did not know was a fact - and do not yet. He says that if a figure be anyhow divided and the compartments differently coloured so that figures with any portion of common boundary line are differently coloured - four colours may be wanted, but not more - the following is the case in which four colours are wanted. Query cannot a necessity for five or more be invented. ..... If you retort with some very simple case which makes me out a stupid animal, I think I must do as the Sphynx did....

Hamilton und Arthur Calley stellten zum ersten mal 1878 das Problem der mathematischen Gesellschaft vor. Innerhalb nur eines Jahres fand Alfred Kempe einen Beweis für den Satz. Elf Jahre später, 1890, zeigte Percy Heawood, dass Kempes Beweis fehlerhaft war.

- Heinrich Heesch entwickelte in den 1960er und 1970er Jahren Verfahren, um einen Beweis mithilfe des Computers zu suchen.

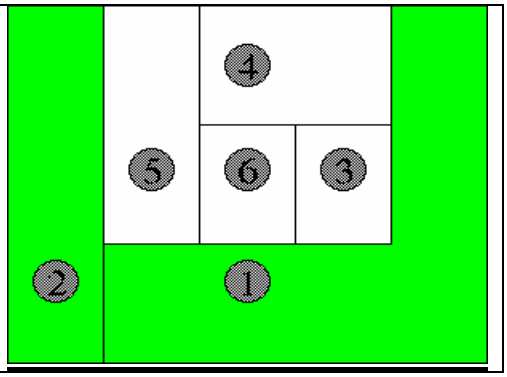
- Darauf aufbauend konnten Ken Appel und Wolfgang Haken 1977 einen solchen finden. Der Beweis reduzierte die Anzahl der problematischen Fälle von Unendlich auf 1.936 (eine spätere Version sogar 1.476), die durch einen Computer einzeln geprüft wurden.
- 1996 konnten Neil Robertson, Daniel Sanders, Paul Seymour und Robin Thomas einen modifizierten Beweis finden, der die Fälle auf 633 reduzierte. Auch diese mussten per Computer geprüft werden.

- *Der Vier-Farben-Satz war das erste große mathematische Problem, das mit Hilfe von Computern gelöst wurde.* Deshalb wurde der Beweis von einigen Mathematikern nicht anerkannt, da er nicht direkt durch einen Menschen nachvollzogen werden kann.

Vierfarbenproblem (Backtracking)

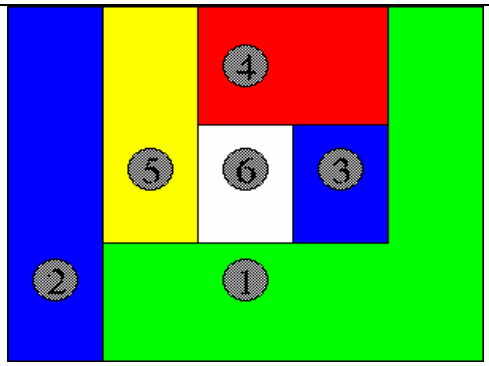
Bei der Wahl der Farbe für ein neues Land müssen mindestens zwei Bedingungen erfüllt sein:

Bedingung 1:
Das neu gefärbte Land grenzt an kein bereits früher gefärbtes Land mit der gleichen Farbe.

Beispiel 1: Bedingung 1 ist verletzt. (Land 1 und 2 haben die gleiche Farbe)	

Bedingung 2:

Durch die Färbung des neuen Landes wird kein angrenzendes, noch nicht gefärbtes Land unfärbbar.

Beispiel 2: Bedingung 2 ist verletzt. (Land 6 ist unfärbbar.)	

Die Bedingungen 1 und 2 müssen zwingend erfüllt sein, damit die Karte korrekt eingefärbt wird. Man bezeichnet sie auch als notwendige Bedingungen.

<u>notwendige Bedingung</u> :

Die Bedingungen 1 und 2 analysieren nur die Situation für die direkt angrenzenden Länder. Es kann durchaus vorkommen, dass die Entscheidung, ein Land mit einer bestimmten Farbe zu belegen, zwar den Bedingungen 1 und 2 genügt, man aber erst später feststellt, dass diese Entscheidung in eine "Sackgasse" führte.

Wir geben jetzt ein Verfahren an, das den schrittweisen Aufbau der Lösung steuert, die Bedingungen 1 und 2 überprüft und schließlich zu einer korrekten Einfärbung der Karte führt:

Wir nummerieren die Länder durch und bezeichnen sie mit L1,...,Ln.

Erinnerung:
Die Farben probieren wir in der Reihenfolge grün, blau, rot, gelb durch.

Algorithmus

i=1

1. **Solange** i<=n
   2. betrachte Land Li
   3. **Wenn** noch nicht alle Farben durchprobiert
      dann
         färbe das Land Li mit der nächsten Farbe.
         Wenn Bedingungen **1** und **2** erfüllt sind
         dann
            i=i+1
            gehe zu **1**.
         sonst
            gehe zu **3**.

      sonst
         entfärbe Land Li
         i=i-1
         gehe zu **2**.

Literaturverzeichnis :

Backtracking (Allgemein):

1- http://www.fhaugsburg.de/informatik/projekte/emiel/vista/deutsch/backtracking.html
2- http://www.laurentianum.de/backtr.htm
3- **Algorithmen & Datenstrukturen – Eine Einführung mit Java –**
(Gunter Saake / Kai-Uwe Sattler) *dpunkt.verlag*
4- Eirund, H. / Müller, B. / Schreiber, G.: „**Formale Beschreibungsverfahren der Informatik**“, B. G. Teubner, Stuttgard Leipzig Wiesbaden, 1. Auflage, 2000;
5- **Algorithmen und Datenstrukturen** (4. Auflage)
(T. Ottmann / P. Widmayer) *Spektrum akademischer Verlag*

Das 8-Damen-Problem:

1. http://www.arstechnica.de/computer/JavaScript/JS11_04.html
2. http://www.hp-gramatke.de/math/german/page0020.htm#8_Damen
3. http://imperium.my-flow.com/viewtopic.php?t=288
4. http://page.inf.fu-berlin.de/~zoppke/E/aufzeichnung.html
5. http://wvs.be.schule.de/faecher/informatik/material/algorithmus/backtracking/backtracking_einfuehrung.html

Das Labyrinth-Problem:

1- http://www.thillm.th.schule.de/pages/schule/faecher/informatik/lpif/programm/llaby.htm
2- http://www.matheprisma.uni-wuppertal.de/Module/4FP/maus.htm

Das 4-Farben-Problem:

- Artikel:

- Neil Robertson, Daniel P. Sanders, Paul Seymour, Robin Thomas: ***A new proof of the four-colour theorem.*** In: *Electronic Research Announcements of the American Mathematical Society.* 2/1996. S. 17-25
- Kenneth Appel, Wolfgang Haken: ***Every Planar Map is Four Colorable****.* Contemp. Math., vol. 98, Amer. Math. Soc., Providence, RI, 1989.

- WWW:

- http://de.wikipedia.org/wiki/Vier-Farben-Satz
- http://www-groups.dcs.st-and.ac.uk/~history/HistTopics/The_four_colour_theorem.html
- http://www.matheprisma.uni-wuppertal.de/Module/4FP/index.htm