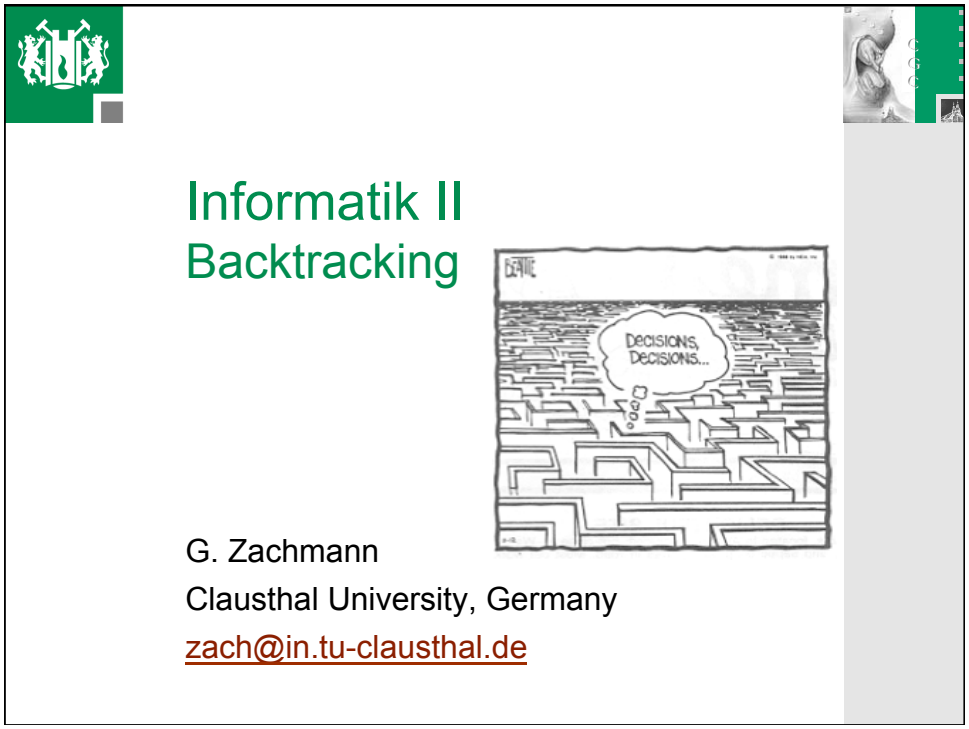
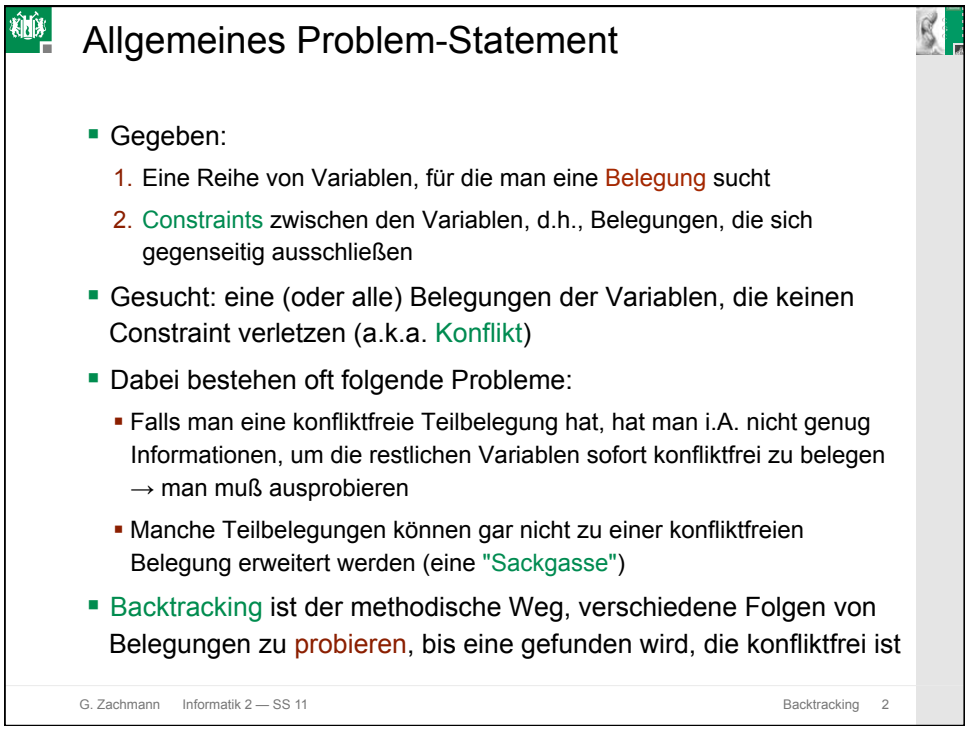


Informatik II
Backtracking

G. Zachmann
Clausthal University, Germany
zach@in.tu-clausthal.de



Allgemeines Problem-Statement

- Gegeben:
  1. Eine Reihe von Variablen, für die man eine Belegung sucht
  2. Constraints zwischen den Variablen, d.h., Belegungen, die sich gegenseitig ausschließen
- Gesucht: eine (oder alle) Belegungen der Variablen, die keinen Constraint verletzen (a.k.a. Konflikt)
- Dabei bestehen oft folgende Probleme:
  - Falls man eine konfliktfreie Teilbelegung hat, hat man i.A. nicht genug Informationen, um die restlichen Variablen sofort konfliktfrei zu belegen → man muß ausprobieren
  - Manche Teilbelegungen können gar nicht zu einer konfliktfreien Belegung erweitert werden (eine "Sackgasse")
- Backtracking ist der methodische Weg, verschiedene Folgen von Belegungen zu probieren, bis eine gefunden wird, die konfliktfrei ist

Chatsworth Garden Maze

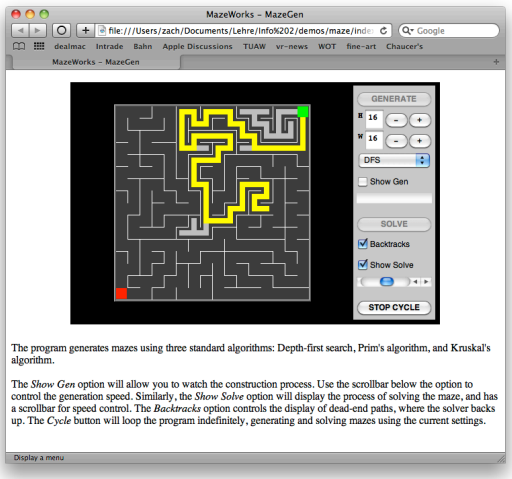
Beispiele

- Ausweg aus einem Labyrinth:
  - Finde einen Weg vom Startpunkt zum Zielpunkt
  - An jeder Kreuzung gibt es 2-3 Entscheidungsmöglichkeiten: geradeaus, rechts, links
  - "Faden der Ariadne" = früheste Beschreibung des Backtracking-Verfahrens

Theseus in the Labyrinth
Sir Edward Burne-Jones (1862)

Ariadne
John William Waterhouse (1898)

- Der Algorithmus ist so einfach, dass eine Demo reicht:

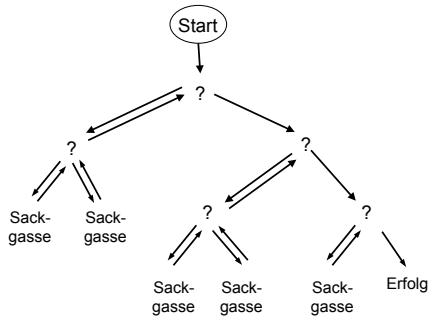


http://www.mazeworks.com/mazegen/

- Solitaire (Brettspiel):
  - Alle Löcher, bis auf eins, sind mit Stiften besetzt
  - Man darf mit einem Stift über einen anderen springen
  - Übersprungene Stifte werden entfernt
  - Ziel: alle Stifte, bis auf den letzten, zu entfernen

- Internet-Browsing

Der abstrakte Backtracking-Algorithmus

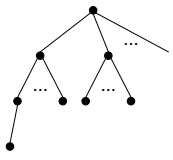



```
explore( node ):
if node ist Blatt:
    if node == Zielknoten:
        return "Erfolg"
    else:
        return "Sackgasse"
foreach Kind c von node:
    explore( c )
    if c erfolgreich:
        return "Erfolg"
return "Sackgasse"
```

- Backtracking kann als Suche nach einem speziellen "Zielblatt" in einem sog. Such-Baum gesehen werden
- Achtung: dieser Baum wird beim Backtracking nie real aufgebaut!
- Wesentliche Komponente: ein Kriterium, so daß Sackgassen möglichst weit oben erkannt werden
- Realisierung der Pfadrückverfolgung durch Rekursion

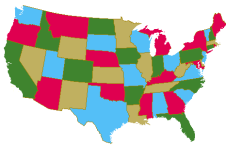
Zusammenhang zu Depth-First-Search

- Ein Knoten wird als nützlich bezeichnet, falls er zu einer Lösung führen könnte, sonst nicht nützlich
- Hauptidee: Backtracking besteht aus der Anwendung von DFS auf den Such-Baum, wobei für jeden Knoten entschieden wird
  - ob er nützlich ist → weiter absteigen;
  - ob er nicht nützlich ist → zurück zum Vaterknoten (*back-tracking*)
- Weitere Betrachtungsweise:
  - Der Suchraum wird durch den Baum ausgefüllt und mit Hilfe von Rekursion und Backtracking durchsucht, um eine Lösung zu finden



Das Färbungsproblem

- Färben einer Landkarte:
  - Eine Karte soll mit nur 4 Farben gefärbt werden
  - Bedingung: benachbarte Länder (gemeinsame Grenze) müssen verschieden gefärbt werden

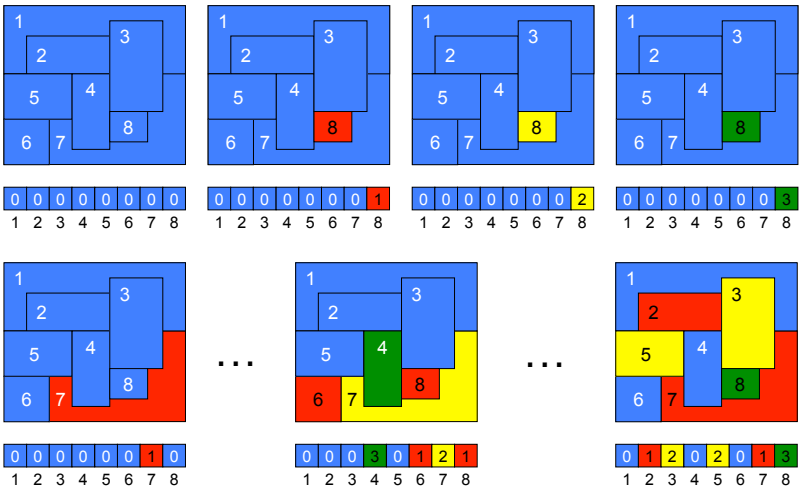


- Der Vier-Farben-Satz [1976]:
  Jede Landkarte in der Ebene lässt sich mit höchstens 4 Farben so färben, daß keine 2 Länder mit gemeinsamer Grenze die gleiche Farbe haben.

Naive Lösung: erschöpfende Suche (*exhaustive search*)

- Länder werden durchnummeriert, dito die Farben
- Erzeugung aller möglichen Färbungen durch die "Tacho-Mehode":

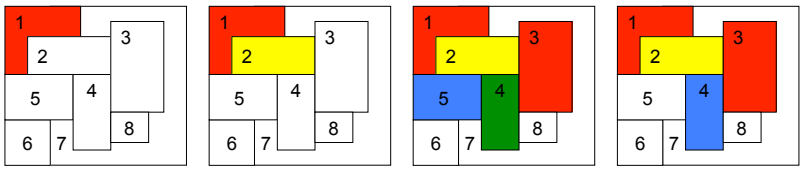
  - Jedes Rädchen = ein Land
  - Starte mit Zählerstand 0…0
  - Drehe rechtes Zählrädchen einmal durch alle Ziffern (= Farben)
  - Bei Übergang von 3 → 0 im rechten Rädchen: drehe zweitrechtes Rädchen 1 Ziffer weiter
  - Bei jedem Zählerstand: überprüfe Konsistenz (4-Farben-Bedingung)



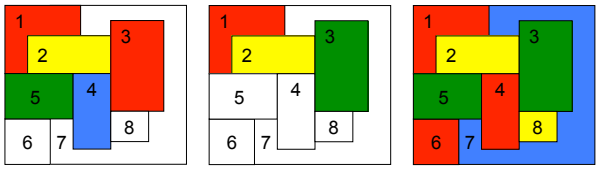
- Nachteile:
  - Im ungünstigsten Fall müssen fast alle Kombinationen durchprobiert werden
    - Im average case müssen die Hälfte aller Kombinationen probiert werden
  - Der Algorithmus zählt weiter, obwohl klar ist, daß einige Kombinationen übersprungen werden können

Eine Backtracking-Lösung

- Idee: ein Land färben und dann versuchen, das nächste zu färben, so daß folgende Bedingungen erfüllt sind:
  - Die benachbarten, bereits gefärbten Länder haben unterschiedliche Farben
  - Deines der benachbarten, noch ungefärbten Länder wird unfärbbar
- Diese lokalen Bedingungen können erfüllt sein, aber dennoch zu keiner Lösung führen (Sackgasse)



7 ist unfärbbar geworden!
→ Backtracking

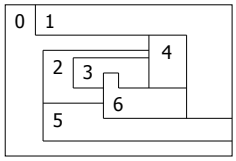


7 wieder unfärbbar

- Vorteil: wesentlich effizienter als erschöpfende Suche
- Nachteil: abhängig von der Nummerierung, d.h. bei ungünstiger Nummerierung ist der Aufwand ebenfalls enorm hoch
- Die Datenstruktur muß folgendes speichern:
  - Länder sind Instanzen einer Klasse **`Land`**
  - Zu jedem Land alle seine Nachbarn (Liste von Zeigern)
  - Ein Array der Länge 4 :
    - Wert 1 an der Stelle *i* = Land ist in der Farbe *i* gefärbt
    - Wert -1 = Land kann mit Farbe *i* gefärbt werden
    - 0 = Land kann nicht mit der Farbe *i* gefärbt werden
    - Initialisiere alle Stellen (= Farben) mit -1

Beispiel-Implementierung in Python

- Achtung:
  - Im folgenden wird die "Unfärbbar"-Heuristik weggelassen
  - Es gibt nur 1 Klasse für alle Länder zusammen
  - Bessere Implementierung à la vorige Folie → Übungsaufgabe
- Datenstruktur:
  - Klasse **`Map`** mit 2 internen Arrays
  - Array **`colors`**, wobei **`colors[i]`** die Farbe des *i*-ten Landes ist
  - Array **`neighbors`** mit den benachbarten Ländern
    - Beispiel: **`neighbors[5][3] == 8`** bedeutet: Land 8 ist zu Land 5 benachbart (hier ist es zufällig das dritte Land in der Liste der zu Land 5 benachbarten Länder)




```
class Map(object):
    def __init__(self):
        self.neighbors = 7 * [None]
        self.neighbors[0] = [ 1,4,2,5 ]
        self.neighbors[1] = [ 0,4,6,5 ]
        self.neighbors[2] = [ 0,4,3,6,5 ]
        self.neighbors[3] = [ 2,4,6 ]
        self.neighbors[4] = [ 0,1,6,3,2 ]
        self.neighbors[5] = [ 2,6,1,0 ]
        self.neighbors[6] = [ 2,3,4,1,5 ]
        self.colors = 7 * [None]
```



```
def main():
    map = Map()
    print map.explore(0, "red" )
    print map.colors
```

```
# Fortsetzung von class Map

def explore( self, country, color ):
    if country >= len( self.neighbors ):
        return True
    if okToColor( country, color ):
        self.colors[country] = color
        for c in [ "red", "yellow", "green", "blue" ]:
            if explore( country+1, c ):
                return True
    return False
```

```
# Teste, ob irgend ein Nachbar von 'country'
# die Farbe 'color' hat

def okToColor( self, country, color ):
    for n in map[country]:
        if self.colors[ map[country][n] ] == color :
            return False
    return True
```


Erläuterung

- Es werden alle Länder rekursiv durchlaufen, begonnen wird mit Land 0
- Für jedes Land wird eine Farbe gewählt:
  - Die Farbe muß von denen aller Nachbarn verschieden sein
  - Wenn keine Farbe gefunden werden kann, wird ein Fehler gemeldet
  - Wenn eine Farbe gefunden wird, wird sie benutzt und mit dem nächsten Land fortgefahren
  - Wenn alle Länder durchlaufen sind (alle wurden gefärbt), wird Erfolg gemeldet
- Nach der Rückkehr vom obersten Aufruf ist der Algorithmus fertig

Komplexität

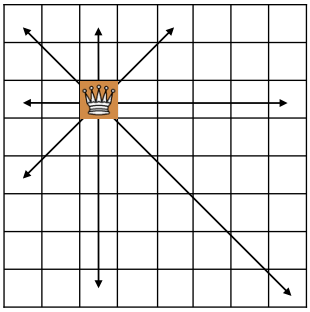
- Exponentieller Aufwand:
  - Anzahl der Kombinationen steigt exponentiell
  - Anzahl der möglichen Sackgassen steigt ebenfalls exponentiell, daher ist Backtracking oftmals deutlich schneller als erschöpfende Suche
- Fällt in die Klasse NP-vollständig:
  - D.h., es gibt wahrscheinlich keinen effizienten (= polynomiellen) Algorithmus

Anwendungen

- Stundenplanproblem
  - Veranstaltungen = Länder
  - Veranst., die nicht gleichzeitig ablaufen können, sind "benachbart"
  - Anzahl der Farben, die zum Einfärben benötigt werden = Anzahl der verschiedenen Zeitfenster (minimale Anzahl = *chromatic number* des Graphen)
- Registerzuweisungs-Probleme
- Bandbreitenzuweisungs-Probleme
- Viele mathematische Probleme lassen sich als Knotenfärbe-Problem formulieren

Das Acht-Damen-Problem

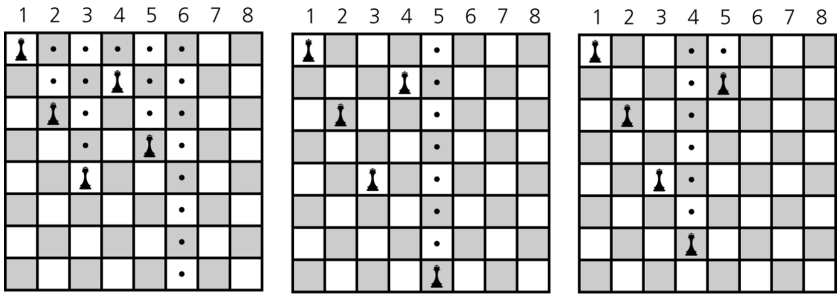
- Aufgabe: stelle 8 Damen so auf ein Schachbrett, daß keine Dame eine andere angreift
  - Schachbrett = 8x8 Felder
  - Eine Dame kann jede andere Figur angreifen, die in derselben Zeile, derselben Spalte, oder der Diagonalen steht
    - (Die Dame ist die mächtigste Figur beim Schach)



Die naive Strategie

- Naive Strategie: alle möglichen Belegungen ausprobieren
  - Anzahl Möglichkeiten = 64·63·62·61·60·59·58·57
  - Zu viele → nicht praktikabel
- Idee:
  - Plaziere von vornherein nur eine Dame pro Spalte
  - Starte mit der ersten Spalte; suche für jede eine Zeile, die noch nicht belegt ist
  - Es müssen nur noch 8! = 40 320 Damen-Anordnungen auf Diagonalangriffe überprüft werden

Beispiel

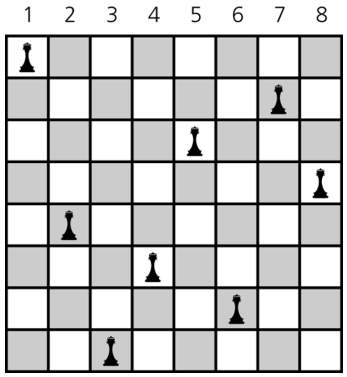


a) Die 5 Damen können sich nicht gegenseitig angreifen, aber jede Position in Spalte 6 wird bedroht

b) Bachtracking zu Spalte 5 um ein anderes Feld für die Dame zu probieren

c) Backtracking zu Spalte 4 um ein anderes Feld für die Dame zu probieren und dann Spalte 5 erneut zu betrachten

Eine Lösung des Problems



Der Algorithmus im Detail (hier zeilenweise)

```
# Teste, ob Dame in (neue_reihe,neue_spalte) in Konflikt mit
# einer der schon stehenden Damen in 'loesung' ist.
# loesung = Array mit neue_reihe-1 Eintraegen,
# wobei in Zeile i eine Dame auf Feld (i,loesung[i]) steht.

def konflikt( neue_reihe, neue_spalte, loesung ):
    # Stelle sicher, dass neue Dame mit keiner existierenden
    # Dame auf gleicher Spalte oder Diagonale steht
    for reihe in 0,…,neue_reihe-1:
        if ( loesung[reihe] == neue_spalte    # gleiche Spalte
             or
             (neue_reihe - reihe) +           # gleiche Diagonale
             (neue_spalte - loesung[reihe]) == 0
             or
             (neue_reihe - reihe) -           # gleiche Diagonale
             (neue_spalte - loesung[reihe]) == 0
           ):
               return True                    # Konflikt
    return False # alles OK
```



```
# Berechne alle Loesungen fuer ein Brett der Groesse
# neue_reihe x spalten.
# teilloesungen = alle Loesungen fuer
# (neue_reihe-1) x spalten
# Eine loesung = definiert wie in Fkt konflikt()
# Jedes Array in teilloesungen hat (neue_reihe-1)
# Eintraege, jedes Array im Output hat neue_reihe Elem.

def dame_dazu( neue_reihe, spalten, teilloesungen ):
    neue_loesungen = []
    for loesung in teilloesungen:
        # versuche jede Spalte von neue_reihe
        for neue_spalte in 0,…,spalten:
            if not konflikt( neue_reihe, neue_spalte,
                             loesung ):
                neue_loesung = loesung + [neue_spalte]
                neue_loesungen.append( neue_loesung )
    return neue_loesungen
```

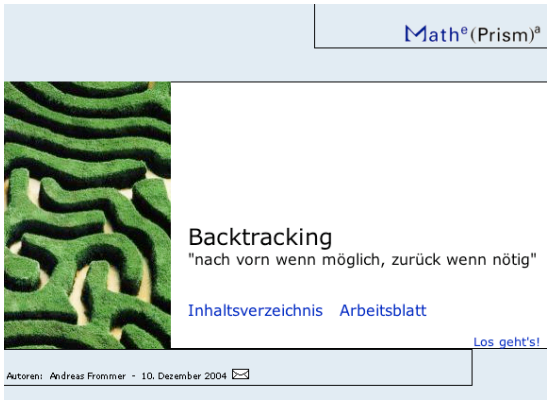
```
# Erzeuge eine Liste von Lösungen auf einem Brett der
# Groesse (reihen x spalten)
# Eine Lösung wird durch eine Liste der Spaltenpositionen,
# indiziert durch die Reihennummer, angegeben.
# Die Indizes beginnen mit 0!

def damenproblem( reihen, spalten ):
   if reihen <= 0 or spalten <= 0:
       return [ [] ]
       # Ende der Rekursion; Lösung =
       # keine Dame auf dem nicht-existierenden Brett
   else:
       teilloesungen = damenproblem( reihen-1, spalten )
       return dame_dazu(reihen-1, spalten, teilloesungen )
```


Demo

http://www.faust.fr.bw.schule.de/mhb/backtrack/achtdamen/eight.htm

Eine unterhaltsame Lehreinheit zum Thema



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

Abschließende Bemerkungen

- Aufgaben für Backtracking sind i.A. nicht Optimierungsaufgaben, sondern Ja-/Nein-Aufgaben (Entscheidungsprobleme)
- Backtracking = trial & error
  - Verwenden meistens Rekursion
- Falls Backtracking keine Lösung findet → es gibt keine Lösung!
- Die Algorithmentechnik "Backtracking" stammt von Edsger Dijkstra
  - Das 8-Damen-Problem diente zur Illustration des "structured programming" (1972)