Informatik II
Backtracking

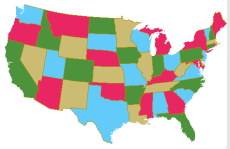
G. Zachmann
Clausthal University, Germany
zach@in.tu-clausthal.de

- Szenario: man hat eine Reihe von Entscheidungen aus einer Menge von Wahlmöglichkeiten zu treffen, wobei
  - man nicht genug Informationen hat, um zu wählen;
  - jede Entscheidung zu einer neuen Menge von Wahlmöglichkeiten führt;
  - eine Folge von Wahlmöglichkeiten (oder auch mehrere) die Lösung des Problems sein können;
  - manche Entscheidungen evtl in eine "Sackgasse" führen.
- Backtracking ist der methodische Weg, verschiedene Folgen von Entscheidungen zu probieren, bis eine gefunden wird, die das Problem löst

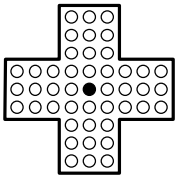
Beispiele

- Ausweg aus einem Labyrinth:
  - finde den Weg vom Eingang zum Ausgang
  - an jeder Kreuzung gibt es 2-3 Entscheidungs-möglichkeiten: gerade aus, rechts, links
  - Algo: s. "Faden der Ariadne" = früheste Beschreibung des Backtracking-Verfahrens

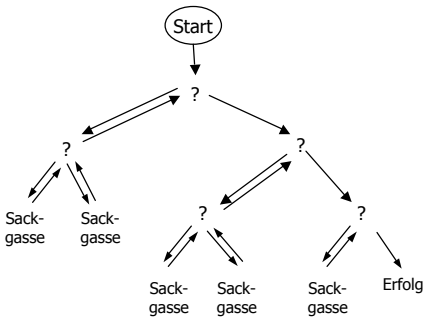
- Färben einer Landkarte:
  - eine Karte soll mit nur 4 Farben gefärbt werden
  - Einschränkung: benachbarte Länder müssen verschieden gefärbt werden



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



Entscheidungsbaum



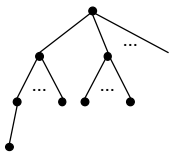
Abstrakter Algorithmus:

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

- Backtracking kann als Suche nach einem speziellen "Zielblatt" eines Baumes gesehen werden
- Achtung: dieser Baum wird beim Backtracking nie real aufgebaut!
- Wesentliche Komponente: 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 Entscheidungsbaum, wobei für jeden Knoten entschieden wird
  - ob er nützlich ist → weiter absteigen;
  - ob er nicht nützlich ist → Rückschritt zu seinem Elternknoten
- Weitere Betrachtungsweise:
  - Der Suchraum wird durch den Baum ausgefüllt
  - Jedes Blatt ist eine Lösung oder keine Lösung
  - der Suchraum wird mit Hilfe von Rekursion und Backtracking durchsucht, um eine Lösung zu finden



Färbungsproblem

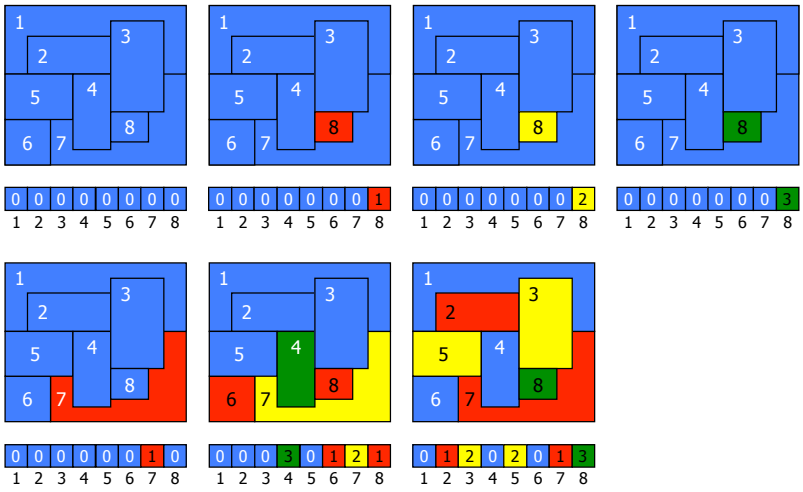
- Vier-Farben-Satz: jede Landkarte in der Ebene lässt sich mit höchstens 4 Farben so färben, daß keine 2 Länder mit gemeinsamer Grenze die gleich Farbe haben
- Datenstruktur muß folgendes speichern:
  - eine Farbe für jedes Land,
  - zu jedem Land alle seine Nachbarn

Historie

- Ursprung:
  - 1852 aufgestellte Vermutung durch Francis Guthrie beim Färben der Karte der Ländereien von England: 4 Farben genügen um die Länder einer Karte so zu färben, daß alle benachbarten Länder unterschiedliche Farben tragen
  - Veröffentlichung des Problems durch Brief des Mathematik-Professors De Morgan an Hamilton
- Beweisfindung:
  - Beweise durch Kempe (1878) und Tait (1880) wurden 1890 bzw. 1891 widerlegt
  - 1890: Formulierung des 5-Farben-Satzes durch Heawood → Existenz einer oberen Schranke bewiesen
  - 1977: Beweis durch Ken Appel und Wolfgang Haken
    - Beweis reduziert die problematischen Fälle von unendlich auf 1.936, später weniger, welche vom Computer geprüft wurden
  - 2004: Entwicklung eines formalen Beweises mittels Beweis-Assistenten COQ durch Werner und Gonthier
    - (S.a.: bewiesenermaßen korrekter Compiler in Coq)
  - Erstes Problem das mittels Computer bewiesen wurde

Erschöpfende Suche

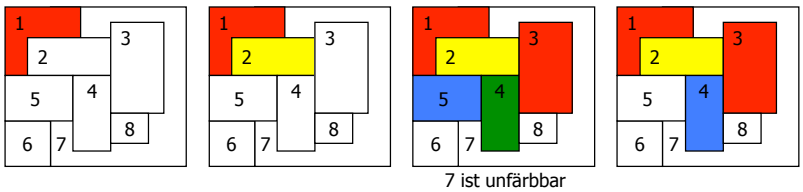
- Länder werden durchnummeriert, dito die Farben
- Erzeugung aller möglichen Färbungen durch „Tacho-Mehode“:
  - jedes Rädchen = ein Land
  - starte mit Zählerstand 0…0
  - drehe rechtes Zählrädchen einmal durch alle Ziffern (= Farben)
  - bei Übergang von 3 → 0 im rechten Rädchen: drehe zweitrechtes Rädchen 1 Ziffer weiter
  - bei jedem Zählerstand: überprüfe Konsistenz (4-Farben-Bedingung)



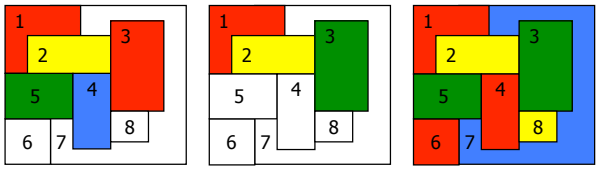
- Nachteile:
  - im ungünstigsten Fall müssen fast alle Kombinationen durchprobiert werden
  - Algorithmus zählt weiter, obwohl klar ist, daß einige Kombinationen übersprungen werden können
  - sehr hoher Zeitaufwand

Backtracking-Lösung

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



7 ist unfärbbar geworden!
→ Backtracking

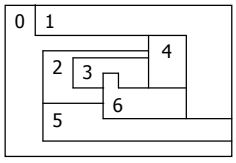


7 wieder unfärbbar

- Vorteil: wesentlich effizienter als erschöpfende Suche
- Nachteil: abhängig von der Nummerierung, d.h. bei ungünstiger Nummerierung ist der Aufwand ebenfalls enorm hoch
- Implementierung:
  - Länder sind Instanzen einer Klasse, deren jede neben den Referenzen der Nachbarländer ein Array der Länge 4 enthält:
    - 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 *i* gefärbt werden
    - Initialisierung mit -1
  - Realisierung der Pfadrückverfolgung durch Rekursion

Vereinfachte 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
```

```
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 i in range( 0, len(map[country]) ):
        if self.colors[ map[country][i] ] == color :
            return False
    return True
```


Kurze Wiederholung

- Es wurden alle Länder rekursiv durchlaufen, begonnen wurde mit Land 0
- Für jedes Land war eine Farbe zu wählen:
  - die Farbe mußte von denen aller Nachbarn verschhieden sein
  - wenn keine Farbe gefunden werden konnte, wurde ein Fehler gemeldet
  - wenn eine Farbe gefunden werden konnte, wurde sie benutzt und mit dem nächsten Land fortgefahren
  - als alle Länder durchlaufen waren (alle wurden gefärbt), wurde Erfolg gemeldet
- Nach der Rückkehr vom obersten Aufruf war der Algorithmus fertig

Komplexität

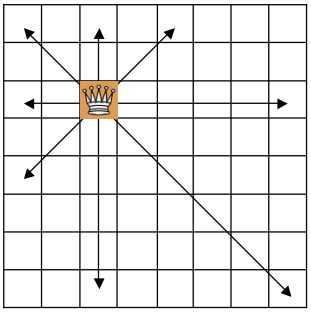
- Exponentieller Aufwand:
  - Anzahl der Kombinationen steigt exponentiell
  - Anzahl der möglichen Sackgassen steigt ebenfalls exponentiell, daher ist Backtracking oftmals deutlich schneller als erschöpfende Suche
- Fällt in Klasse NP-schwer:
  - D.h., es gibt wahrscheinlich keinen effizienten (= polynomiellen)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

Anwendungen

- Stundenplanproblem
  - Veranstaltungen = Länder
  - Veranst., die nicht gleichzeitig ablaufen können, sind "benachbart"
  - Anzahl der Farben = Anzahl der verschiedenen Zeitfenster
- Registerzuweisungs-Probleme
- Bandbreitenzuweisungs-Probleme
- Viele mathematische Probleme lassen sich als Knotenfärbe-Problem formulieren

Das Acht-Damen-Problem

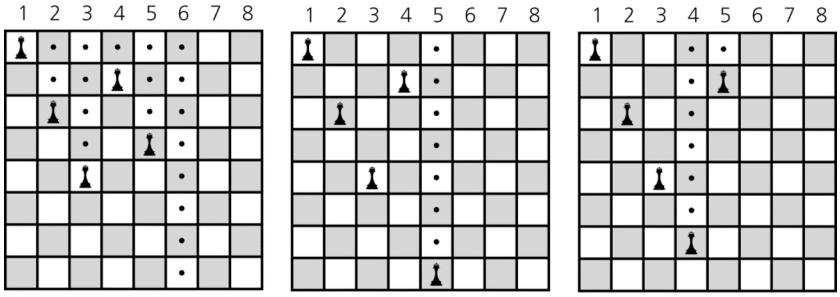
- Problem: stelle 8 Damen so auf ein Schachbrett, daß keine Dame eine andere angreift
  - Schachbrett = 8x8 Felder
  - eine Dame kann jede andere Figur angreifen, die in derselben Zeile, derselben Spalte, oder der Diagonalen steht
  - (die Dame ist die mächtigste Figur beim Schach)



Naive Strategie

- Naive Strategie: alle möglichen Belegungen ausprobieren
  - Anzahl Möglichkeiten = 64·63·62·61·60·59·58·57
  - zu viele → nicht praktikabel
- Einfache Beobachtung:
  - keine Dame kann zusammen mit einer anderen in einer Zeile oder Spalte stehen
  - es müssen nur noch 8! = 40320 Damen-Anordnungen auf Diagonalangriffe überprüft werden

Backtracking-Beispiel



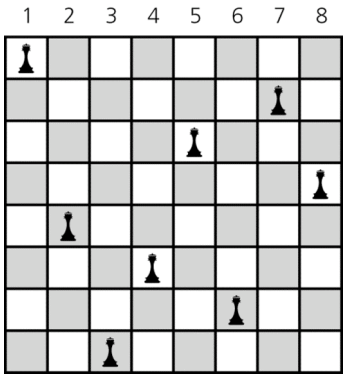
(a) (b) (c)

(a) 5 Damen können sich nicht gegenseitig angreifen, aber jede Dame in Spalte 6 könnte es

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

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

Eine Lösung des Problems



Implementierung

- Verallgemeinerung:
  - Konstruiere alle Lösungen
- Methode:
  - Reduziere n×n-Problem auf (n-1)×n-Problem (eine Zeile weniger)
  - Per Konstruktion können keine 2 Damen in derselben Zeile stehen
  - Ausgabe: Liste von Lösungen; Lösung = Array von Spaltennummern, Index = Zeilennummer


```
# Erzeuge eine Liste von Lösungen auf einem Brett mit
# reihen und spalten.
# Eine Lösung wird durch eine Liste der Spaltenpositionen,
# indiziert durch die Reihennummer, angegeben.
# Die Indizes beginnen mit Null.

def damenproblem( reihen, spalten ):
    if reihen == 0:
        return [[]]
        # Eine Lösung, nämlich: keine Dame
        # auf dem nicht-existierenden Brett
    else:
        return dame_dazu(reihen-1, spalten,
                         damenproblem(reihen-1, spalten) )
```

```
for loesung in damenproblem( 8, 8 ):
   print loesung
```

Erinnerung: dies ist ein Bsp. für eine verschachtelte Rekursion (*"compound recursion"*) [s. 1. Semester]

```
# Probiere für alle gegebene Teilloesung,
# eine Dame in "neue_reihe" und alle Spalten zu stellen.
# Wenn kein Konflikt mit der Teilloesung auftritt,
# ist eine neue Loesung des um eine Reihe erweiterten
# Bretts gefunden.

def dame_dazu(neue_reihe, spalten, vorherige_loesungen):
    neue_loesungen = []
    for loesung in vorherige_loesungen:
        # Versuche jede Spalte von neue_reihe
        for neue_spalte in range(spalten):
            if kein_konflikt( neue_reihe, neue_spalte,
                              loesung ):
                # Kein Konflikt → dieser Versuch ist Lsg
                neue_loesungen.append( loesung +
                                       [neue_spalte] )
    return neue_loesungen
```



```
# Kann Dame in neue_spalte / neue_reihe gestellt werden,
# ohne dass sie eine der schon stehenden D. schlagen kann?

def kein_konflikt( neue_reihe, neue_spalte, loesung ):
    # Stelle sicher, dass neue Dame mit keiner exist.en
    # Dame auf gleicher Spalte oder Diagonale steht
    for reihe in range(neue_reihe):
        if ( loesung[reihe] == neue_spalte  or
                                            # gleiche Spalte
             loesung[reihe] + reihe == neue_spalte + neue_reihe  or
                                            # gleiche Diagonale
             loesung[reihe] - reihe == neue_spalte - neue_reihe  ):
                                            # gleiche Diagonale
            return 0
    return 1
```

Demo

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

Unterhaltsame "Lehreinheit"

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