

Algorithmen & Datenstrukturen

Backtracking

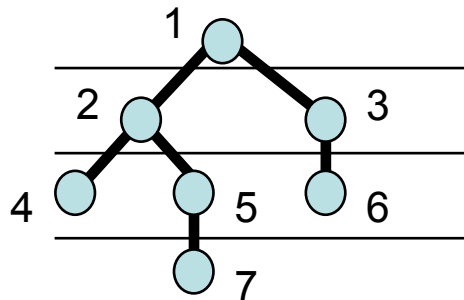
Wolfgang Auer

- Für viele Probleme
 - gibt es keinen spezifischen Algorithmus, der das Problem löst
 - gibt es keinen Algorithmus, der für praxisnahe Problemgrößen in akzeptabler Zeit eine Lösung ergibt
- Versuche dennoch eine Lösung zu finden, beruhen auf dem *Trial and Error* Prinzip. D.h. man versucht durch Probieren eine Lösung zu finden!

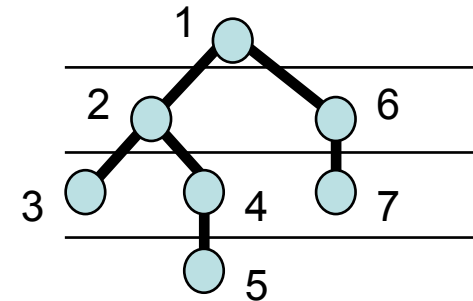
- *Backtracking* (Zurückverfolgung) ist eine
 - *systematische Art der Suche*
 - *in einem vorgegebenen Lösungsraum*

- Führt eine Teillösung in eine
 - *Sackgasse*, dann wird der
 - jeweils letzte Schritt *rückgängig* gemacht („back-tracking“)

■ Alternative Ansätze

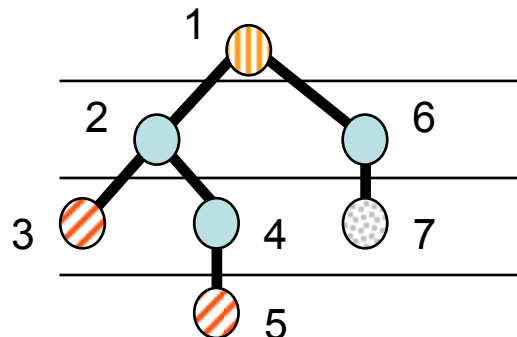





Ebene für Ebene wird durchsucht ⇒
Breitensuche



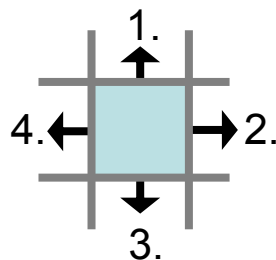
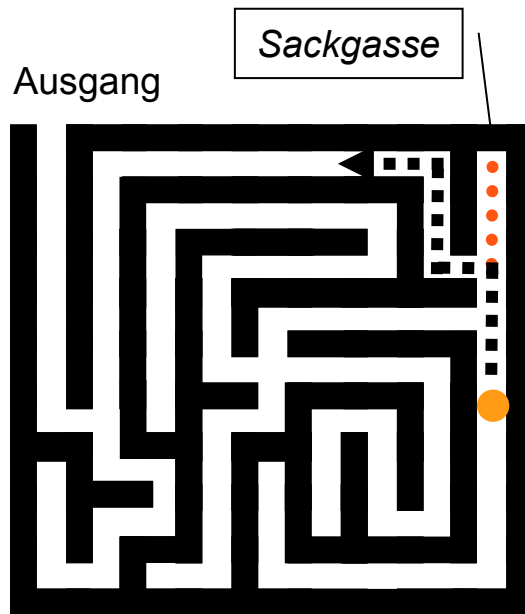
Möglichst schnell in die Tiefe ⇒
Tiefensuche

■ Backtracking ist Tiefensuche!



-  Start
-  Sackgasse
-  Lösung

Weg aus einem Labyrinth (1)

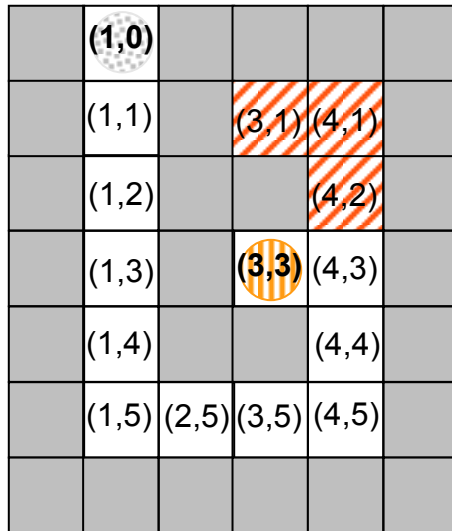


■ Strategie

- Vom aktuellen Feld **systematisch** nach 1. oben, 2. rechts, 3. unten und 4. links abzweigen
- **Markiere** die besuchten Felder
- In **Sackgasse** solange Züge zurücknehmen bis Zug auf ein nicht bereits besuchtes Feld möglich

Weg aus einem Labyrinth (2)

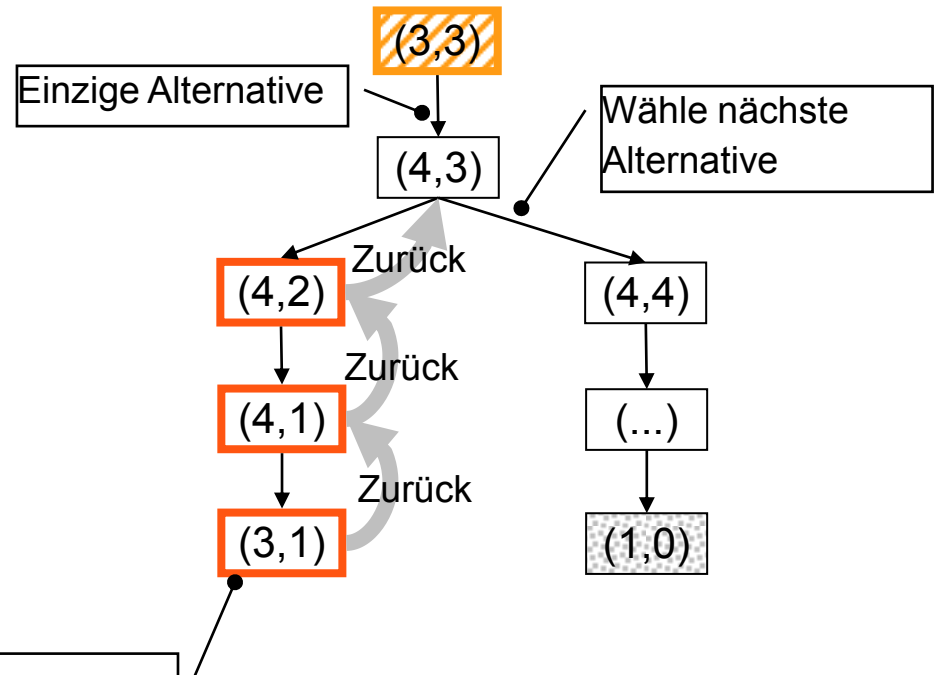
(0,0)



Start = (3,3)

Nur Wände oder bereits
besuchte Nachbarn \Rightarrow **Sackgasse**

■ Lösungsbaum



Weg aus dem Labyrinth

Pseudo Code

```
global char labyrinth[][]
global Direction pfad []
global int schritte //Anzahl der Schritte auf dem Pfad
findeLösung(pos) {
    markiere labyrinth[pos] als besucht

    if (pos == Position eines Ausgangs) {
        return true //Lösung wurde gefunden
    }
    while (es gibt noch unbesuchte Nachbarn von pos) {
        nPos = nächstes unbesuchtes Nachbarfeld in N0SW-Reihenfolge
        pfad[schritte] = Richtung des nächstes Nachbarfeld
        schritte = schritte + 1

        if (findeLösung(nPos) == true) {
            //Lösung für Teilproblem wurde gefunden
            return true
        } else { //Wir sind in einer Sackgasse
            schritte = schritte - 1
        }
    } // end while
    //Es gibt keine Schrittfolge, die zu einer Lösung führt
    return false
}
```

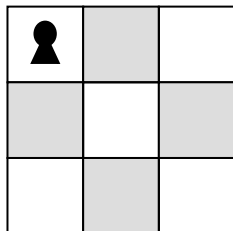
Allgemeiner Backtracking Algorithmus

```
findeLösung( ↓index ↓↑lösung) {  
    while (es gibt noch neue Teilschritte) {  
        wähle den nächsten neuen Teilschritt schritt  
        erweitere lösung um schritt  
  
        if (lösung vollständig) {  
            return true //Lösung wurde gefunden  
        } else {  
            if (findeLösung(index + 1, lösung) == true) {  
                //Lösung für Teilproblem wurde gefunden  
                return true  
            } else { //Wir sind in einer Sackgasse  
                mache schritt rückgängig  
            }  
        }  
    } // end while  
  
    //Es gibt keine Schrittfolge, die zu einer Lösung  
    führt  
    return false  
}
```

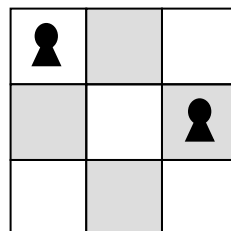

n-Damen Problem (1)

- **Geg:** Schachbrett mit $n \times n$ Feldern und n Damen.
- **Ges:** Alle Damen so platzieren, sodass sich diese gegenseitig nicht schlagen können
- **Idee:** Eine Dame bedroht alle Figuren in der gleichen Zeile/Spalte und entlang den Diagonalen
⇒ pro Zeile 1 Dame, pro Spalte 1 Dame
- **Lösungsschritt**
Suche in der jeweils nächsten Zeile eine Position, die von keiner der bisher gesetzten Figuren bedroht wird

■ 1. Schritt



■ 2. Schritt



■ ...

n-Damen Problem

Algorithmus (Pseudo code)

```
PlaceQueen( ↓row ↓↑nrOfQueensPlaced) {  
    for all (columns col) {  
        if (QueenFits(col, row)) {  
            place Queen at (col|row)  
            nrOfQueensPlaced++  
  
            if (PlaceQueen(row + 1, nrOfQueensPlaced) == true) {  
                //Solution found  
                return true  
            } else { //Unable to place Queen => Backtracking  
                remove Queen from (col|row)  
                nrOfQueensPlaced--  
            } // end if  
        } // end if  
    } // end while  
  
    return (nrOfQueensPlaced == boardSize)  
  
}
```

- Bei der Tiefensuche werden bei
 - max. k möglichen Verzweigungen von jeder Teillösung aus
 - einem Lösungsbaum mit maximaler Tiefe von n im schlechtesten Fall
- $1 + k + k^2 + k^3 + \dots + k^n = (k^{n+1} - 1) / (k - 1) = \mathbf{O(k^n)}$
Alternativen untersucht
 \Rightarrow **exponentielle Laufzeit**

Anwendung von Wissen über das Problem
führt zu wesentlichen Verbesserungen!

- Bisher
Systematische Suche, aber wenig intelligent
- Verbesserung
Verwendung von zusätzlichen Wissen

Bei jeder Entscheidung welcher Schritt als nächster ausgeführt werden soll, wird der viel-versprechendste gewählt.

⇒ Erhöhung der Wahrscheinlichkeit schneller eine Lösung zu finden

Heuristiken sind „Strategien, die *mit höherer Wahrscheinlichkeit* (jedoch *ohne Garantie*) das Auffinden einer *Lösung beschleunigen* sollen.“

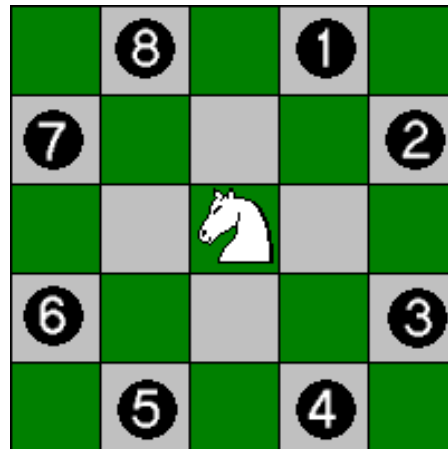
(Quelle: Schüler-Duden „Die Informatik“, S. 236, Bibliograph. Institut, Mannheim/Wien/Zürich, 1986)

Springerproblem (1)

- Das Problem:

- Ein Springer soll jedes Feld eines Schachbretts genau einmal besuchen

Ein **Springer** kann max. **8 mögliche Orte** anspringen



Quelle: Thomas Dübendorfer

- Zug ist **gültig**, wenn das neue Feld
 - innerhalb des Spielfeldes liegt
 - unbesucht ist
- Besuchsreihenfolge wird gespeichert

Springerproblem (2)

- **Backtracking Algorithmus:**
 - Systematisch gemäß obiger Reihenfolge springen
 - Bei „Sackgassen“ Sprünge zurücknehmen (Backtracking)
 - Springerweg, sobald Weglänge = Anzahl Felder

36	31	22	19	4	29
23	18	3	30	9	20
32	35	24	21	28	5
17	2	33	8	13	10
34	25	12	15	6	27
1	16	7	26	11	14

Die Abbildung zeigt einen Springerweg auf einem Schachbrett der Grösse 6x6.

Begonnen wurde unten links.

- Es muss immer auf das Feld gesprungen werden, welches am **schlechtesten** erreichbar ist.
- **Maß** für schlechte Erreichbarkeit eines Feldes:
 - Anzahl unbesuchter Felder, die man von diesem in einem Sprung erreichen kann. Je weniger, um so schlechter erreichbar.
- **Wir bauen diese Regel ein:**
 - Anstatt in der vorgegebenen Reihenfolge zu springen, berechnen wir jeweils die Erreichbarkeit aller vom aktuellen Ort anspringbarer Felder und springen zuerst zum am schlechtest erreichbaren.

- Auswirkungen
 - Springerweg für 8x8 in unter 1 Sekunde.
 - Springerweg für 50x50 in wenigen Sekunden.
 - Springerweg ab ca. 60x60 ziemlich langsam.
- Grund
 - Bis 56x56 Felder brauchen wir dank der Heuristik von Warnsdorf keinen einzigen Backtracking-Schritt zu machen. Der Aufwand zur Bestimmung der Erreichbarkeiten der Felder macht sich also mehr als bezahlt.

- Im Gegensatz zum herkömmlichen Backtracking werden alle Lösungen gesucht

```
findeAlleLösungen( ↓index ↓↑lösung) {  
    while (es gibt noch neue Teilschritte gibt) {  
        wähle den nächsten Teilschritt schritt  
        erweitere lösung um schritt  
  
        if (lösung vollständig) {  
            Verarbeite Lösung  
        } else {  
            findeAlleLösungen(index + 1, lösung)  
        }  
        mache schritt rückgängig  
    } // end while  
}
```