Grundlagen Theoretischer Informatik 3

SoSe 2012 in Trier

Henning Fernau
Universität Trier
fernau@uni-trier.de

Grundlagen Theoretischer Informatik 3 Gesamtübersicht

- Organisatorisches; Einführung
- Algorithmenanalyse: Komplexität und Korrektheit
- Zur Behandlung NP-schwerer Probleme: Ausblicke
- (Randomisierte Algorithmen und ihre Analyse)

Dynamisches Programmieren

Wichtige Technik zur praktischen Umsetzung rekursiver Lösungsideen.

Bsp.: Rekursive Idee zum Lösen schwieriger graphentheoretischer Probleme auf Bäumen:
Würde ich unter gewissen Bedingungen optimale Lösungen für die Sohnknoten kennen, so auch für den Vaterknoten.

Kubisches Parsen von Typ-2-Grammatiken in Chomsky-NF (GTI 2).
Berechnen regulärer Ausdrücke aus Automaten (GTI 1).

Grundidee: Berechne zunächst “einfache Fälle” und löse immer schwierigere durch Rückgriff auf bereits berechnete Lösungen.

Der Algorithmus von Warshall (Floyd 62 / Kleene 56 / Warshall 62 / Roy 59)

Berechnung der reflexiv-transitiven Hülle R^* von $R \subseteq X \times X$, $X = \{1, \ldots, n\}$.
Dabei sei R als Boolesches $n \times n$-Array abgespeichert (mit Grenzen $1 \ldots n$) und S enthalte die Lösung:

(a) Initialisierung:

Für i **von** 1 **bis** n:
    Für j **von** 1 **bis** n:
        $S(i,j) \leftarrow R(i,j) \vee (i = j)$

(b) Hüllenberechnung:

Für k **von** 2 **bis** n:
    Für i **von** 1 **bis** n:
        Für j **von** 1 **bis** n:
            $S(i,j) \leftarrow S(i,j) \vee (S(i,k) \wedge S(k,j))$

Gedanken zur Korrektheit:
R kann als Kantenrelation eines gerichteten Graphen gedeutet werden.
R^* drückt Erreichbarkeit aus:
$(u,v) \in R^*$ gdw. es gibt einen (evtl. leeren) Pfad von u nach v im Graphen.
Invarianten:
Vor jedem Eintritt in den Schleifenrumpf der äußeren FOR-Schleife zur Hüllenberechnung gilt:
(1) Falls $(u,v) \in S$, so gibt es einen Pfad von u nach v.
(2) Gibt es einen Pfad der Länge höchstens $(k-2)$ von u nach v, so gilt $(u,v) \in S$.

Der Algorithmus von Warshall: déja vu??

Folgende “sauberere Variante” sollte klarmachen, wie der Algorithmus mit dem Kleene’schen Algorithmus zur Berechnung regulärer Ausdrücke aus Automaten zusammenhängt.
Wo steckt die “rekursive Idee”?

Initialisierung:
Für i **von** 1 **bis** n:
    Für j **von** 1 **bis** n:
        $S(i,j,1) \leftarrow R(i,j) \vee (i = j)$
Hüllenberechnung:
Für k **von** 2 **bis** n:
    Für i **von** 1 **bis** n bzw. $k-1$:
        Für j **von** 1 **bis** n bzw. $k-1$:
            $S(i,j,k) \leftarrow S(i,j,k-1) \vee (S(i,k,k-1) \wedge S(k,j,k-1))$

NP-Härte: Wiederholungen und Ergänzungen

NP: Klasse von Problemen (formal: Sprachklasse), die mit nichtdeterministischen Turing-Maschinen in Polynomzeit gelöst werden können.

Anstelle von Turing-Maschinen können auch WHILE-Programme als Sprachakzeptoren dienen:
Erinnerung: Wörter "sind" Zahlen (logarithmische Kosten!),
Akzeptanz bei "Ausgabe" von **true**.

Einfachere Vorstellung:
WHILE-Programme (wie bislang betrachtet) mit einer Zusatzfunktion **guess**,
die einen Booleschen Wert zurückliefert.
Der Aufruf dieser Zusatzfunktion benötigt lediglich einen Zeitschritt.
Ein Wort w (also eine Zahl) wird von so einem WHILE-Programm akzeptiert,
wenn es geeignete Rückgaben von **guess** gibt,
sodass das Programm hält und **true** zurückgibt.

NP-Härte

SAT gehört zu den schwierigsten Problemen in NP.
Das heißt: Wenn SAT in P läge, so auch alle anderen Probleme aus NP.

Ein Problem A heißt NP-*hart*, wenn Folgendes gilt:
Kann man $A \in P$ zeigen, so lägen alle NP-Probleme in P.
Wie beweist man NP-Härte für ein Problem A?
Man zeigt: Wäre es möglich, A in Polynomzeit zu lösen, so wäre es auch möglich, ein bekanntermaßen NP-hartes Problem B in Polynomzeit zu lösen und mithin alle Probleme aus NP.
Man *reduziert* B auf A, i.Z.: $B \leq_P A$.

Formaler: Seien A und B Sprachen über einem Alphabet E.
Dann heißt B auf A *polynomial reduzierbar*,
wenn es eine in Polynomzeit berechenbare Funktion $f : E^* \to E^*$ gibt,
so dass für alle $w \in E^*$ gilt: $w \in B \iff f(w) \in A$. Schreibweise: $B \leq_p A$.

Wie schwierig sind Reduktionen?

Kreise im Graphen:
—Pfad, der in dem Knoten endet, in dem er auch beginnt
—Knoten $x_1, \ldots x_m$ mit $x_m = x_1$ und $(x_i, x_{i+1}) \in E$ für $1 \leq i < m$ (bzw. $\{x_i, x_{i+1}\} \in E$ bei ungerichteten Graphen).

- GERICHTETER HAMILTON-KREIS:
  Gegeben: gerichteter Graph $G = (V, E)$.
  Frage: Gibt es in G Kreis, der alle Knoten genau einmal berührt, d.h. gibt es Permutation $(x_1, \ldots, x_n)$ von V, für die $x_1, \ldots, x_n, x_1$ ein Kreis ist?

- UNGERICHTETER HAMILTON-KREIS:
  Gegeben:ungerichteter Graph $G = (V, E)$.
  Frage: Gibt es in G Kreis, der alle Knoten genau einmal berührt?

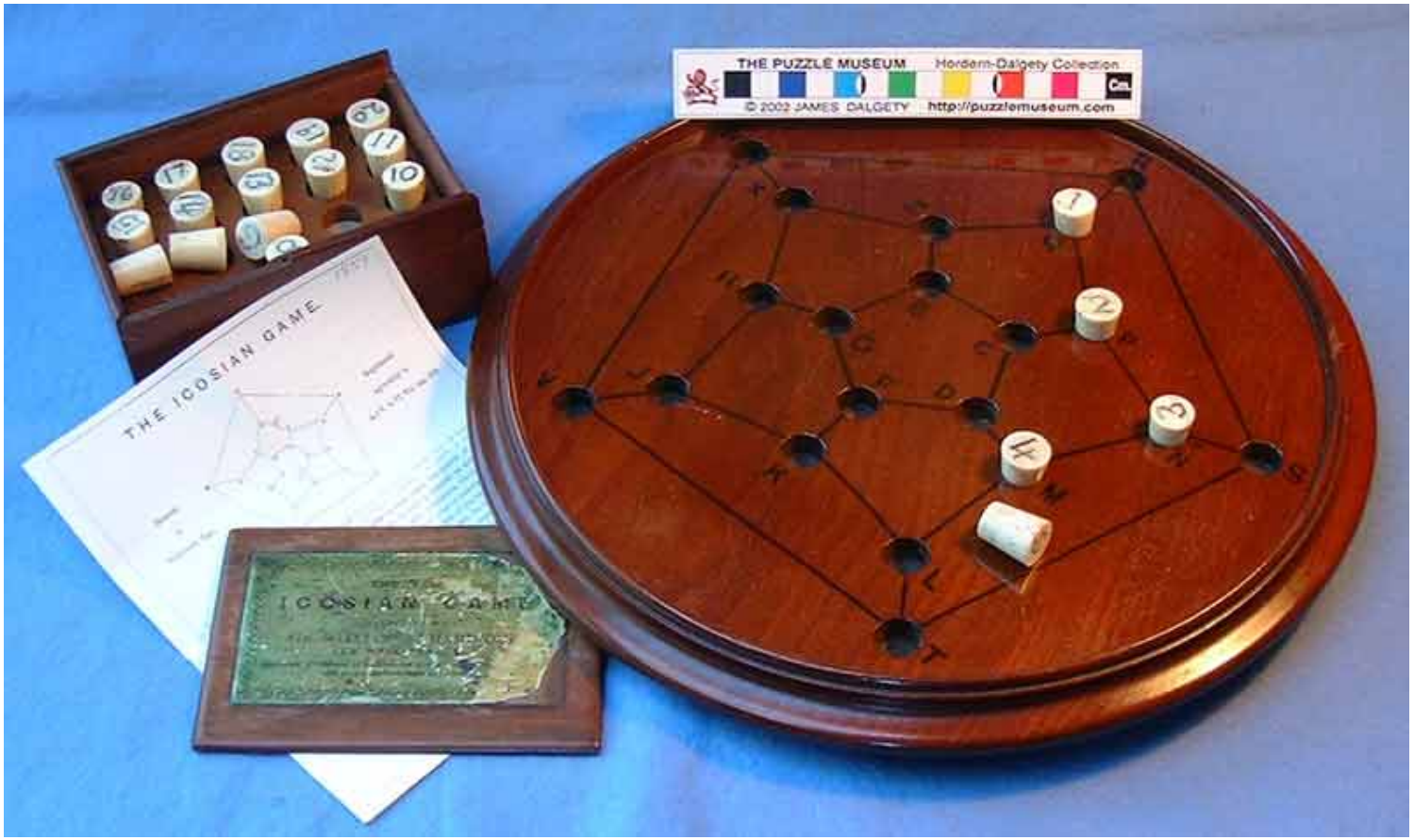
Ein Bei-Spiel... Rösselsprung

42	57	44	9	40	21	46	7
55	10	41	58	45	8	39	20
12	43	56	61	22	59	6	47
63	54	11	30	25	28	19	38
32	13	62	27	60	23	48	5
53	64	31	24	29	26	37	18
14	33	2	51	16	35	4	49
1	52	15	34	3	50	17	36

Diese Springer-Tour wurde von Euler gefunden.

Sie entspricht einem Hamiltonschen Kreis auf einem geeignet definierten (ungerichteten) “Springer-Graphen”.

Einschub: Mathematische Spiele; Gelderwerb für Hamilton

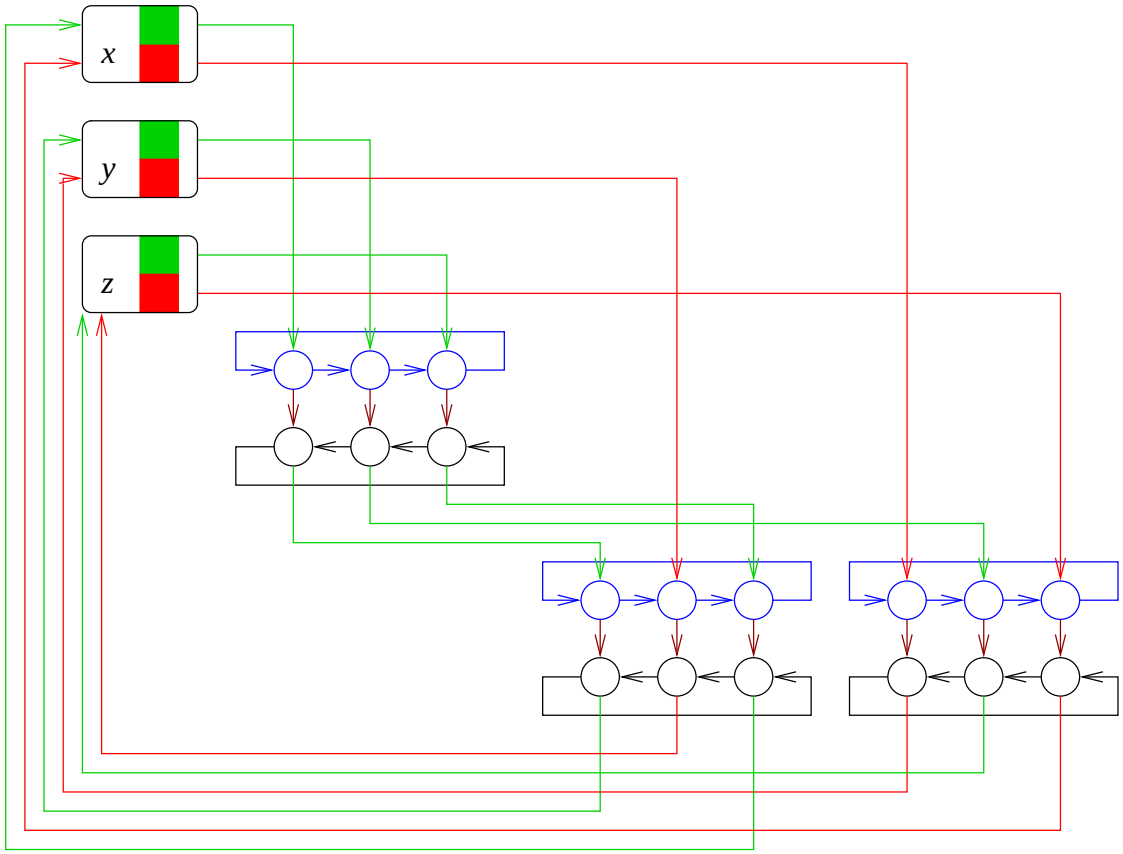


Eine aufwändige Reduktion

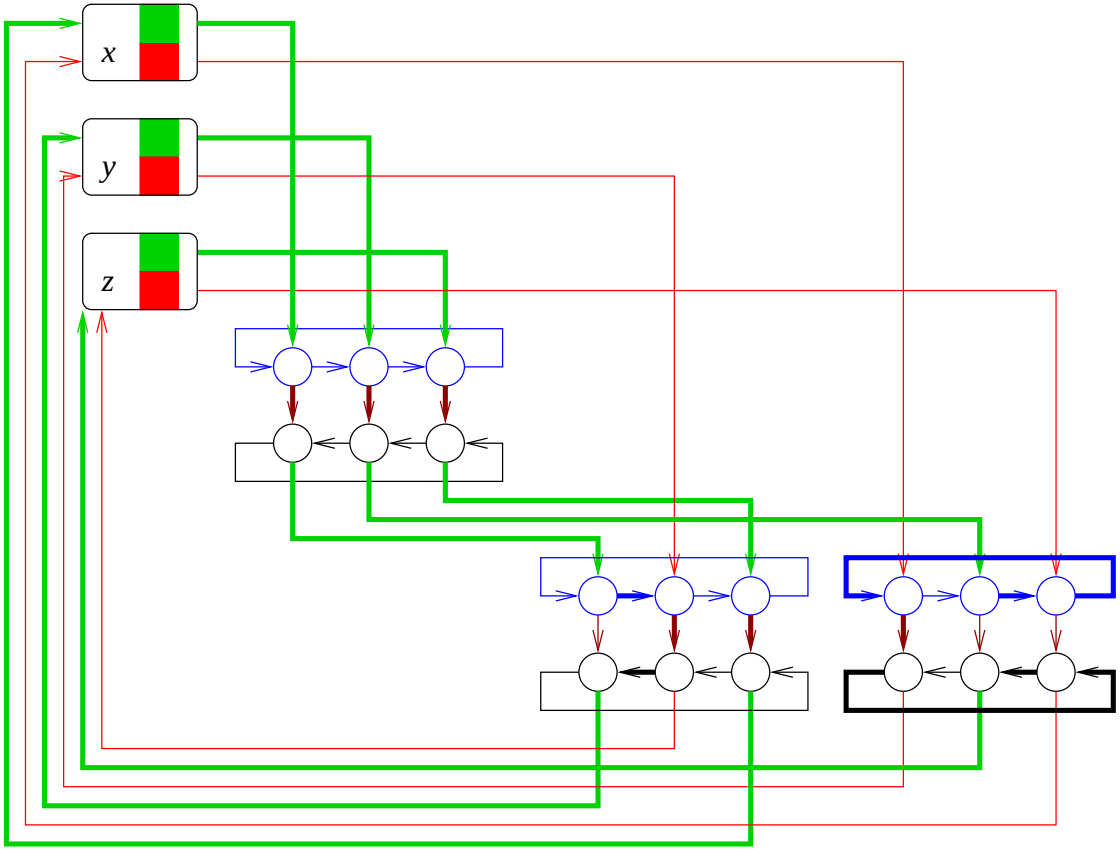
Satz: $3 - \mathrm{SAT}' \leq_{\mathrm{p}} \texttt{GERICHTETER HAMILTON} - \texttt{KREIS}$

Der formalere Beweis wird in der Vorlesung Komplexitätstheorie im Master-Studium gezeigt.
Die nächsten zwei Folien sollen die Konstruktion lediglich andeuten.
Wichtig wieder:
Bausteine (Gadgets) für Variablenauswahl und Klauselauswertung.

Ein Beispiel für die Konstruktion $w = ((x \vee y \vee z) \wedge (x \vee \bar{y} \vee z) \wedge (\bar{x} \vee y \vee \bar{z}))$



Eine Lösung für das Beispiel $\phi(x) = \phi(y) = \phi(z) = 1$.



TRAVELING SALESMAN:

Gegeben: ungerichteter Graph $G = (V, E)$, Kostenfunktion $f : E \to \mathbb{N}$, Zahl $k \in \mathbb{N}$.

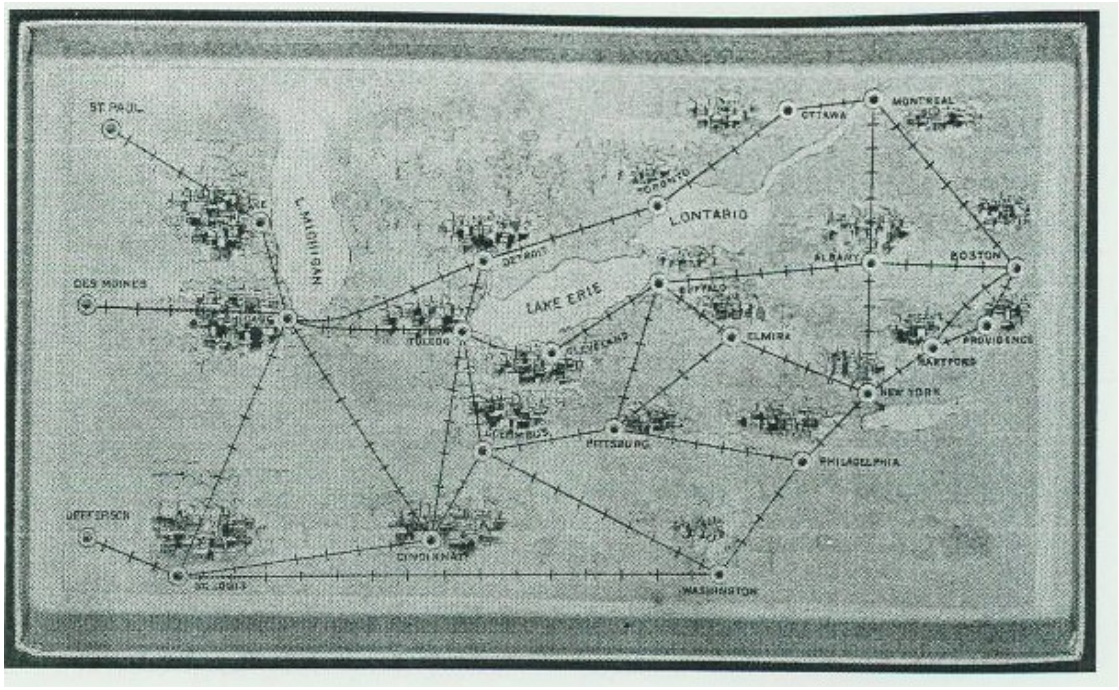
Frage: Gibt es in G Kreis $x_1, \ldots x_n, x_1$, der alle Knoten mindestens einmal berührt, sodass die Summe aller $f(e)$ für die Kanten e im Kreis höchstens k ergibt, d.h. folgender Ungleichung genügt?

$$\sum_{1 \leq i \leq n} f(x_i, x_{i+1}) \leq k$$

Hinweis: Bei TSP sind "Feinheiten" wie "Mehrfachbesuche" von Knoten unerheblich.

Satz: TSP (insbesondere ohne Mehrfachbesuche) ist NP-vollständig.
UNGERICHTETER HAMILTON-KREIS kann nämlich trivial als TSP (ohne Mehrfachbesuche) kodiert werden.

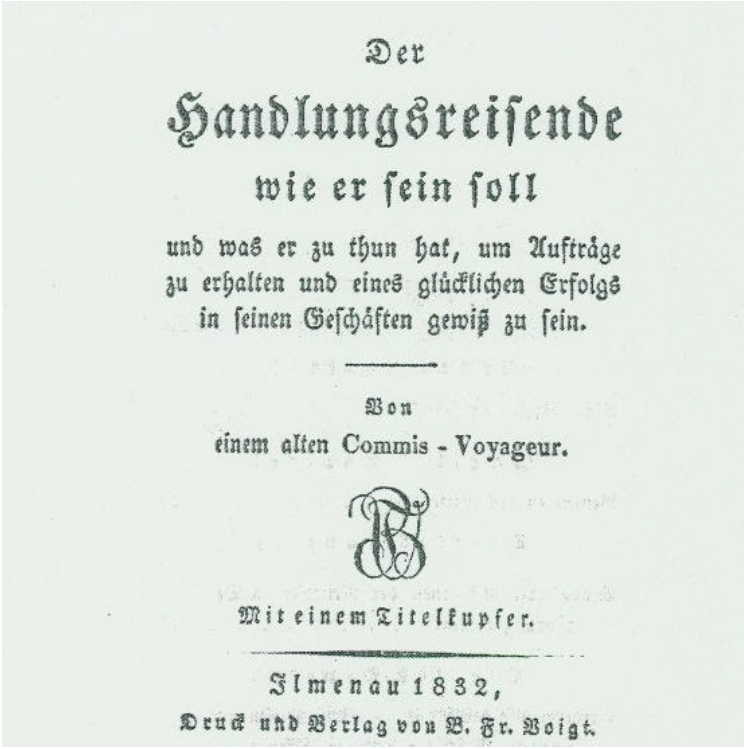
Noch ein Spiel. . . NP-Härte als Geschäftsidee!



Das Spiel “Commercial Traveller”.

Offensichtlich (WO?) mit Mehrfachbesuchen gemeint.

Vom harten Geschäft der Handelsreisenden zeugen alte Berichte:



Der

Handlungsreisende

wie er sein soll

und was er zu thun hat, um Aufträge zu erhalten und eines glücklichen Erfolgs in seinen Geschäften gewiß zu sein.

Von

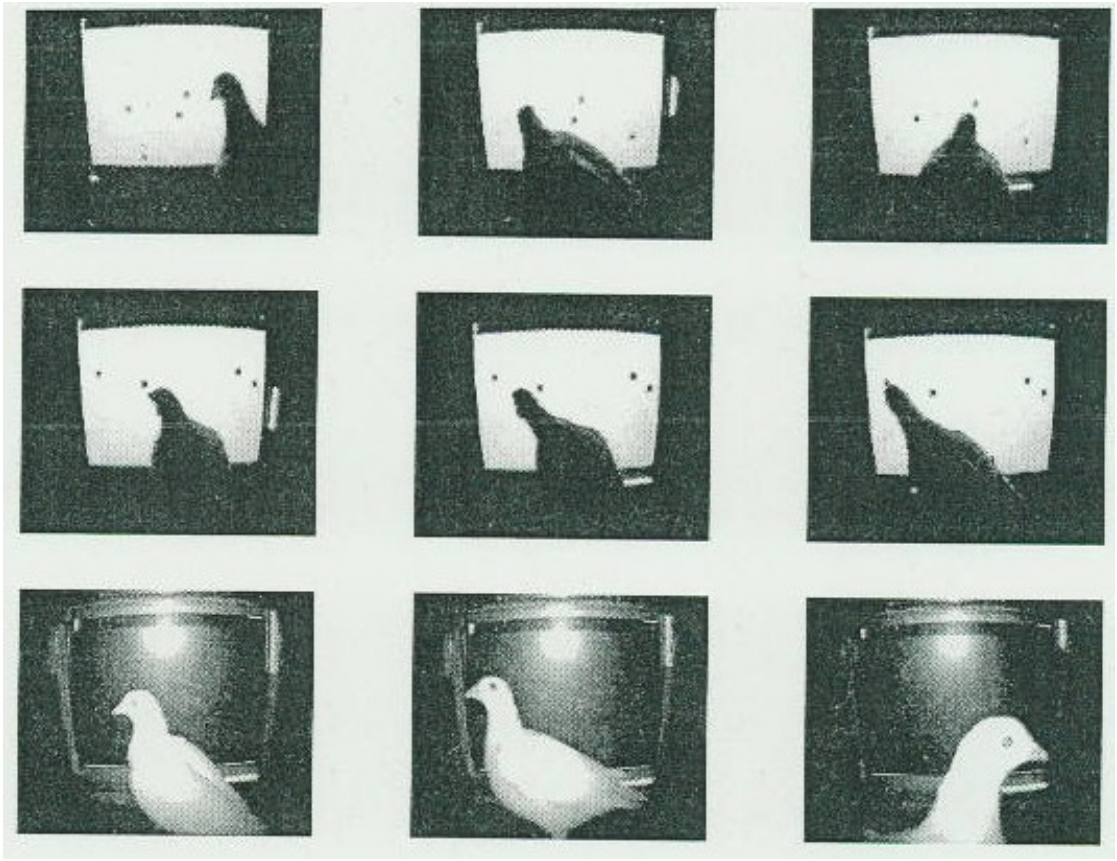
einem alten Commis - Voyageur.

Mit einem Titelkupfer.

Ilmenau 1832,

Druck und Verlag von B. Fr. Voigt.

Schnellste Visiten interessanter Punkte . . . ein Taubenspiel



Wie löst man nun TSP ?

Naiver Algorithmus:

- Untersuche alle Permutationen (d.h. Laufzeit $\Omega(n!)$)...
- Triviale Verbesserung: Starte immer in Stadt 1 (d.h. $\Omega((n-1)!)$)

Laufzeit (geschätzt, bei 1.000.000 Permutationen pro Sekunde):

n	Zeit
10	0.3 s
11	3.6 s
12	40 s
13	8 m
14	1 h 43 m
15	1 d
16	15 d
17	240 d
18	11 y
19	202 y
20	3836 y

Uminterpretation von TSP:

- Betrachte **TSP** als (kantenbewerteten) Graphen
- Knotenmenge $V = \{1, \ldots, n\}$
- Kantenmenge $E = V \times V$, Bewertung: $m_{i,j}$ für Kante $(i, j) \rightsquigarrow$ Matrix M

Teil-Lösungen für **TSP**:

- zu Knoten i und $S \subset V$ setze

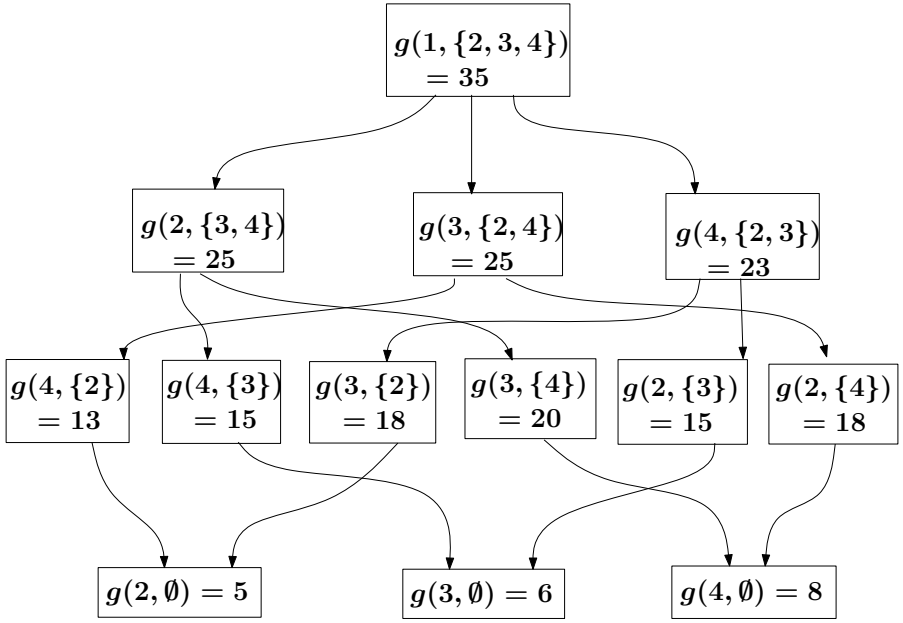
$$g(i, S) = \quad \begin{array}{l} \text{Gewicht des besten Weges von } i \text{ nach } 1, \\ \text{der jedes } j \in S \text{ genau einmal berührt} \end{array}$$

- Dann ist $g(1, V \setminus \{1\})$ gesuchte Lösung des TSP
- Rekursive Beschreibung von g:

$$g(i, S) = \begin{cases} m_{i,1} & S = \emptyset \\ \min_{j \in S} \left(m_{i,j} + g(j, S \setminus \{j\}) \right) & S \neq \emptyset \end{cases}$$

Beispiel:

$$M = \begin{bmatrix} 0 & 10 & 15 & 20 \\ 5 & 0 & 9 & 10 \\ 6 & 13 & 0 & 12 \\ 8 & 8 & 9 & 0 \end{bmatrix}$$



Rekursive Implementierung $\rightsquigarrow$ 'überflüssige' Auswertungen von g!

‘Dynamisches Programm’ für TSP:

- Bottom-Up Auswertung
- Speicherung der Funktionswerte in Tabelle

Iterativ programmiert:

```
FOR i:=2 TO n DO g[i,∅]=m[i,1];
FOR k:=1 TO n−2 DO
    FOR S, |S|=k, 1∉S DO
        FOR i ∈{2,...,n}\S DO
           berechne g[i,S] nach Formel
berechne g[1,{2,...,n}] nach Formel
```

Aufwand:

- Speicherplatz: Tabelle mit $< n \cdot 2^n$ Einträgen
- Zeit: (Größe der Tabelle) $\cdot$ (Aufwand pro Eintrag), d.h. $n^2 \cdot 2^n$

(Hilfsmittel zum Progammieren von Mengen in `C++`, z.B. `bitset`)

Gegenüberstellung

Programme zur Implementierung solcher Formeln:
1) naiv rekursiv
2) mit dynamischem Programmieren

Alternative zu 2): Sukzessiver Tabellenaufbau durch Rekursion;
Vorteil: Nur die tatsächlich benötigten Tabelleneinträge werden berechnet.
Diese Vorgehensweise wird *Memoisierung* genannt.
In anderem Zusammenhang wird auch von *Lazy Evaluation* gesprochen.

Exkurs: Bitsets in C++

Quelle: www.cplusplus.com/reference/stl/bitset/

```
Member functions
(constructor)           Construct bitset (public member function)
applicable operator     Bitset operators (functions)

Bit access:
operator[]              Access bit (public member function)

Bit operations:
set                     Set bits (public member function)
reset                   Reset bits (public member function)
flip                    Flip bits (public member function)

Bitset operations:
to_ulong                Convert to unsigned long integer (public member function)
to_string               Convert to string (public member function)
count                   Count bits set (public member function)
size                    Return size (public member function)
test                    Return bit value (public member function)
any                     Test if any bit is set (public member function)
none                    Test if no bit is set (public member function)
```

```
/* 1) Naiv rekursiv */
#include <bitset>
#include <iostream>
#include <iomanip>
using namespace std;
/* Achtung: V=0,1,...,n-1
   Modifikation: Nicht pi(1)=1 sondern pi(n)=n, damit S nur aus {0,1,...,n-2}
*/
unsigned long formel(
        unsigned int n, unsigned int m[],
        unsigned int i, bitset<30> S){
  bitset<30> S2;
  unsigned int minimum=2000000000, m2;
  if (S.none()) return m[i+n*(n-1)] ;
  else {
     for (unsigned j=0; j<n-1; j++){ if (S.test(j)){
           S2 = S;   S2.set(j,0);
           m2 = m[i+n*j] + formel(n,m,j,S2);
           if (m2 < minimum) minimum = m2;
       }
     }
     return minimum;
  }
};
```

```
int main(){
  unsigned int n;
  cin >> n;
  unsigned int zpn = 1<<(n-1);
  unsigned int m[n*n];
  cout << "Parameter: "<< n <<", Teilmengen: "<< zpn;

  for ( unsigned int i=0; i<n;   i++ )
      for ( unsigned int j=0; j<n;j++)  m[i+n*j]     = 4;
  for ( unsigned int i=0; i<n;   i++ )  m[i+n*i]     = 0;
  for ( unsigned int i=0; i<n-1; i++ )  m[i+n*(i+1)] = 2;
  m[n-1 + n*0] = 2;

  cout << "Resultat: "<<formel(n,m,n-1, zpn-1);
}
```

```
/* 2) Dynamisches Programm */
#include <bitset>
#include <iostream>
#include <iomanip>
using namespace std;
/* Achtung: V=0,1,...,n-1
   Modifikation: Nicht pi(1)=1 sondern pi(n)=n, damit S nur aus {0,1,...,n-2}
*/
unsigned long formel(
        unsigned int n, unsigned int m[], unsigned int g[],
        unsigned int i, bitset<30> S){
  bitset<30> S2;
  unsigned int minimum=2000000000, m2;
  if (S.none()) return m[i+n*(n-1)] ;
  else {
     for (unsigned j=0; j<n-1; j++){ if (S.test(j)){
           S2 = S;   S2.set(j,0);
           m2 = m[i+n*j] + g[j+n*S2.to_ulong()];
           if (m2 < minimum) minimum = m2;
       }
     }
     return minimum;
  }
};
```

```
int main(){
  unsigned int n;
  cin >> n;
  unsigned int zpn = 1<<(n-1);
  unsigned int m[n*n], g[n*zpn];
  cout << "Parameter: "<< n <<", Teilmengen: "<< zpn;
  for ( unsigned int i=0; i<n;   i++ )
      for ( unsigned int j=0; j<n;j++)  m[i+n*j]     = 4;
  for ( unsigned int i=0; i<n;   i++ )  m[i+n*i]     = 0;
  for ( unsigned int i=0; i<n-1; i++ )  m[i+n*(i+1)] = 2;
  m[n-1 + n*0] = 2;

  for(unsigned int i=0; i<n-1;i++)
     g[i+n*0] = formel(n,m,g,i,0);
  for (unsigned int k=1; k<= n-2; k++){
    cout << k<< "...";
    for (unsigned int iS=1; iS< zpn;iS+=1) {
       bitset<30> S(iS);
       if (S.count()==k) {
         for ( unsigned int i=0; i<n-1;i++)
           if (!S.test(i)){
                g[i+n*iS]=formel(n,m,g,i,iS);
    } } } }
  cout << "Resultat: "<<formel(n,m,n-1, zpn-1);
}
```

Vergleich der Laufzeiten:

(Intel CoreDuo T2400, 1.83GHz, `g++ -O2`, `ulimit -s unlimited`)

	rekursiver Algorithmus	dyn.Prog, Zeit	dyn.Prog, Speicher
n	$(\approx (n-1)!\ )$	$(\approx n^2 \cdot 2^n)$	$(\approx n \cdot 2^n)$
10	0.01 s	$<$ 0.01 s	?
11	0.5 s	$<$ 0.01 s	?
12	6 s	$<$ 0.01 s	?
13	74 s	$<$ 0.01 s	?
14	1000 s	0.015 s	?
16	?	0.05 s	5 MB
18	?	0.3 s	12 MB
20	?	1.2 s	43 MB
22	?	6 s	180 MB
24	?	28 s	770 MB

Leichte Reduktion des Speichers auf Kosten der Zeit möglich:
—für $g(i, S)$ nur $g(j, S')$ mit $|S'| = |S| + 1$ benötigt
—ca. *halber* Speicherbedarf bei $n = 24$ ...

RUCKSACK

- *Gegeben:* Natürliche Zahlen $a_1, a_2, ..., a_k, A; w_1, \ldots, w_k, W \in \mathbb{N}$
- *Gefragt:* Gibt es eine Teilmenge $J \subseteq \{1, 2, ..., k\}$ mit

$$\sum_{i \in J} a_i \geq A \quad \wedge \quad \sum_{i \in J} w_i \leq W$$

Spezieller: SUBSET SUM

- *Gegeben:* Natürliche Zahlen $a_1, a_2, ..., a_k, b \in \mathbb{N}$
- *Gefragt:* Gibt es eine Teilmenge $J \subseteq \{1, 2, ..., k\}$ mit

$$\sum_{i \in J} a_i = b$$

Satz: SUBSET SUM und somit RUCKSACK ist NP-vollständig.

Die Mitgliedschaft in NP ergibt sich durch einfaches Raten der Menge J.

Zum Härte-Beweis reduzieren wir 3-SAT auf SUBSET SUM.
Die formale Darstellung des Beweises wäre wieder zu aufwändig für diese Veranstaltung.
Eine Beispielskizze soll einen Eindruck vermitteln.

Beispiel:

$$F = (x_1 \vee \neg x_3 \vee x_5) \wedge (\neg x_1 \vee x_4 \vee x_5) \wedge (\neg x_2 \vee \neg x_2 \vee \neg x_5)$$

mit $m = 3$, $n = 5$ und

$v_1 = 100\ 10000$	$v'_1 = 010\ 10000$
$v_2 = 000\ 01000$	$v'_2 = 002\ 01000$
$v_3 = 000\ 00100$	$v'_3 = 100\ 00100$
$v_4 = 010\ 00010$	$v'_4 = 000\ 00010$
$v_5 = 110\ 00001$	$v'_5 = 001\ 00001$
$c_1 = 100\ 00000$	$d_1 = 200\ 00000$
$c_2 = 010\ 00000$	$d_2 = 020\ 00000$
$c_3 = 001\ 00000$	$d_3 = 002\ 00000$

Die Zahl b ist gegeben durch $4...41...1$ (m-mal die Zahl 4, n-mal die 1 im Dezimalsystem).
Eine geeignete Zahlauswahl entspricht nun einer erfüllenden Belegung.
Die Zahlen c_i bzw. d_i dienen lediglich zum “Auffüllen”.

Dynamisches Programmieren; weiteres Beispiel: Rucksack-Problem

- Gegeben: $W \in \mathbb{N}$, Vektoren $(a_1, ..., a_k), (w_1, ..., w_k) \in \mathbb{N}^k$
- Interpretation: k Objekte mit
  - Werten/Kosten/Gewinne (a-Vektor)
  - Gewichten (w-Vektor)
  - (Gewichts-)Kapazität W
- Gesucht: Wie groß kann der 'Wert' $\sum_{i \in J} a_i$ einer Teilmenge J von Objekten werden, deren Gewicht $\sum_{i \in J} w_i$ nicht größer als die Schranke W?
- Formal: bestimme

$$\max\{\sum_{i \in J} a_i \mid J \subseteq \{1, ..., k\} \wedge \sum_{i \in J} w_i \leq W\}$$

Formulierung von Teil-Lösungen:

- bei $k = 1$ (Wert a_1, Gewicht w_1, Schranke W):
  Lösung trivial:
  - a_1, falls $w_1 \leq W$
  - 0, falls $w_1 > W$
- bei $k > 1$ Objekten: Teste zwei Möglichkeiten:
  - Objekt k gehört zu optimaler Menge J:
    J nutzt Teil-Lösung $J' \subseteq \{1, ..., k-1\}$ mit Schranke $W - w_k$
  - Objekt k gehört nicht zu optimaler Menge J:
    J ist identisch zu Teil-Lösung $J' \subseteq \{1, ..., k-1\}$ mit Schranke W,

Daher setze

$$a(i,h) := \left\{ \begin{array}{l} \text{maximaler Lösungswert} \\ \text{für erste } i \text{ Objekte bei Gew.Schranke } h \end{array} \right.$$

mit rekursiver Formulierung

$$a(i,h) = \left\{ \begin{array}{ll} 0 & i = 0 \\ a(i-1,h) & i > 0 \wedge h < w_i \\ \max \left\{ \begin{array}{l} a(i-1,h), \\ a(i-1,h-w_i) + a_i \end{array} \right\} & \text{sonst} \end{array} \right.$$

Dynamisches Programm dazu:

- verwende Tabelle für $a(i,h)$ mit $0 \leq i < k$ und $0 \leq h \leq W$

%

```
FOR i:=0 TO k-1 DO
    FOR h:=0 TO W DO
            berechne a(i,h) nach Formel
berechne a(k,W) nach Formel
```

- Speicherplatz: $k \cdot (W+1)$ (optimierbar zu $2(W+1)$)
- Zeitaufwand: $k \cdot (W+1) \cdot$ (Aufwand für Formelauswertung)

Uniformes Kostenmaß:

- Aufwand $O(k \cdot W)$, d.h. linear in k
- Bei 'beschränktem' W ist `Rucksack` also **polynomiell**!

Wichtig also: betrachte auch W!

- falls W b Bits hat: Aufwand $O(k \cdot 2^b)$, d.h. **exponentiell** in b
- vgl. Reduktion 3-KNF $\mapsto$ `RUCKSACK`
  - erzeugt Rucksackproblem mit "vielen Bits"
  - dynamisches Programm hat hier exponentielle Komplexität!

Alternative Vorgehensweise:

- Betrachte Teil-Lösungen für Kosten (und nicht für Gewichte)!
- definiere $g(i, \nu) :=$ minimales Gewicht für Mengen von Objekten aus $\{1, ..., i\}$, die zu den Kosten ν führen
- dann wieder rekursiv $g(i, \nu)$ über $g(j, \nu')$ mit $j < i$ definierbar
- setze dabei insbesondere $g(0, 0) = 0$ und $g(0, \nu) = \infty$ für $\nu > 0$
- Einzelheiten: Übung...