Beispiel NPC: SAT-Problem (1)

SAT-Problem: Erfüllbarkeit aussagenlogischer Ausdrücke

- SAT für *formula satisfiability*
- Frage: Ist ein gegebener aussagenlogischer Ausdruck erfüllbar oder nicht.
- Wichtiges Problem: erstes Problem von dem gezeigt wurde, daß es **NP**-vollständig ist.

Ein aussagenlogischer Ausdruck besteht aus:

1. n Booleschen Variablen $v_1, v_2, \ldots, v_n$
2. Boolesche Operatoren: $\wedge$ (Konjunktion), $\vee$ (Disjunktion), $\neg$ (Negation), $\rightarrow$ (Implikation) und $\leftrightarrow$ (Äquivalenz)
3. Klammern

Beispiele:

$$F_1 : \neg(B \rightarrow C) \wedge (D \vee (A \rightarrow \neg B))$$

$$F_2 : \neg((A \wedge B) \rightarrow C) \wedge (C \vee (A \rightarrow \neg B))$$

Beispiel NPC: SAT-Problem (2)

SAT-Problem.

- Geg.: Aussagenlogischer Ausdruck F mit n Variablen
- Ges.: Ist F erfüllbar oder nicht.

Algorithmus:

- Bilde die Menge aller möglichen Variablenbelegungen (n-Tupel) und teste auf Erfüllbarkeit

```
function SAT-SOLVER (IN:formula F with n variables)
  // build all 2^n possible assignments
  A:={(true_1,true_2,…,true_n),(false_1,true_2,…,true_n),...,
      (false_1,false_2,…,false_n)};
  for each tuple T in A do
    if F is satisfied by T
      return true;
  end for
  return false;
end
```

Beispiel NPC: SAT-Problem (3)

Aufwand von SAT-SOLVER:

- Dieser einfache Algorithmus hat eine worst-case Laufzeit von **O**(2^n) und hat somit **exponentielles** Wachstum.
- Beispiel 100 Variable: $2^{100} \sim 10^{30}$.
- 1 TB = 10^{12} Byte, d.h. Menge A ist nicht speicherbar unabhängig von der Codierung der n-Tupel, d.h. n-Tupel müssen in der for-Schleife generiert werden.
  Anm.: Weltweite Speichervolumina werden üblicherweise in Exabyte (10^{18}) angegeben!
- Bei Nichterfüllbarkeit ist eine Terminierung nicht absehbar.

Lösung der Beispiele:

- F_1 ist erfüllbar mit (false,true,false)
- F_2 ist nicht erfüllbar

Ähnliche Probleme unterschiedl. Komplexität

1. **Shortest-Path-Problem vs. Longest-Path-Problem.**
   - Kürzester Pfad zwischen 2 Knoten eines Graphen ist in polynomieller Zeit lösbar (siehe Dijkstra´s Algorithmus).
   - Längster Pfad zwischen zwei Knoten eines Graphen ist ein **NP**-vollständiges Problem.
2. **Euler-Kreis vs. Hamiltonscher Kreis.**
   - Euler-Kreis: Ein Pfad durch einen Graphen G, der alle Kanten genau einmal enthält und wieder zum Anfangspunkt zurückkehrt.
     Polynomielles Problem.
   - Hamiltonscher Kreis: Ein Pfad durch einen Graphen G, der jeden Knoten genau einmal besucht und wieder zum Anfangspunkt zurückkehrt.
     NP-vollständiges Problem.

Weitere NP-vollständige Probleme

1. **Traveling Salesman Problem**.
   Gegeben ist ein Graph mit Knoten, die Städten entsprechen, und Kanten, welche die Kosten angeben, um von einer Stadt in die andere zu gelangen. Gesucht ist ein Weg mit minimalen Kosten, der alle Knoten besucht und zum Ausgangspunkt zurückkehrt.

2. **Subset-Sum Problem.**
   Geg.: Menge natürlicher Zahlen $\mathbf{S} \subset \mathbb{N}$. Zahl $\mathbf{t} \in \mathbb{N}$.
   Ges.: Gibt es eine Teilmenge $\mathbf{S}' \subseteq \mathbf{S}$ deren Elemente aufsummiert $\mathbf{t}$ ergibt.

Optimierungsprobleme und Entscheidungsprobleme

- Theorie zur NP-Vollständigkeit basiert auf *Entscheidungs-probleme*.

- **Shortest-path**. *Optimierungsproblem*
  Geg.: u,v
  Ges.: Pfad von u nach v mit min. Kanten k.

- Ein Optimierungsproblem kann in ein Entscheidungsproblem umgewandelt werden.

- **Path.** *Entscheidungsproblem*
  Geg.: u,v,k
  Frage: Existiert ein Pfad von u nach v mit max. k Kanten?

Lösung NP-vollständiger Probleme

Approximationsverfahren.

- Viele **NP**-vollständige Probleme haben praktische Bedeutung und Lösungen sind gesucht.
- Optimale Lösung: NP-vollständig und nicht möglich
- Gesucht: Gute Lösung (nahe dem Optimum) in polynomialer Zeit

- **Approximative Algorithmen:**
  - Es werden Algorithmen entwickelt, die mit polynomialem Aufwand z.B. unter Verwendung von Heuristiken (suboptimale) Lösungen finden.

Optimale vs. Näherungslösung (1)

Bereits bekannt:

Subset-Sum Problem. (*Entscheidungsproblem*, NP-*vollständig*)

Geg.: Menge $S \subset \mathbb{N}$ mit $|S| = n$ und Zahl $t \in \mathbb{N}$.

Frage: Gibt es eine Teilmenge $S' \subseteq S$ deren Elemente aufsummiert t ergibt?

Neu:

Optimierungsproblem zu diesem Entscheidungsproblem:

Gesucht ist jene Teilmenge S' mit max. Summe $\leq t$

(**Rucksackproblem**).

Algorithmus zur optimale Lösung des Rucksackproblems:

1. Bilde alle 2^n Teilmengen und berechne die Summen s_i
2. Wähle als opt. Lösung die Summe $s_i \leq t$ mit $t - s_i =$ minimal

exponentieller Aufwand! (wie bereits gesagt: $2^{100} \sim 10^{30}$)

Optimale vs. Näherungslösung (2)

Algorithmus Rucksackproblem.

Geg.: $S = \{ x_1, x_2, \ldots, x_n \}$, $\text{sum} \leq t$.

Konstruktion der Teilmengen: *iterativer Ansatz*

$L_0 = \{ 0 \}$ $\quad 1=2^0$

$L_1 = L_0 \cup (L_0+x_1) = \{ 0, 0+x_1 \}$ $\quad 2=2^1$

$L_2 = L_1 \cup (L_1+x_2) = \{ 0, 0+x_1, 0+x_2, 0+x_1+x_2 \}$ $\quad 4=2^2$

$\vdots$

L_n

(Notation L+x: zu jedem Element der Menge L wird x addiert.)

```
function optimal_rucksack (S,t)
  n:=|S|; L0:={0};
  for i:= 1 to n
    Li:=Li-1 ∪ (Li-1+xi);
    Li:=Li-{all elements>t};
  end for
return largest number in Ln;
end
```

Optimale vs. Näherungslösung (3)

Polynomiale Näherungslösung.

Problem: Mengen werden zu groß und müssen reduziert werden.

Idee: Wenn zwei Werte nahe beisammen liegen, müssen für eine Approximation nicht beide Werte weiterverfolgt werden.

Funktion **trim**: es kann ein Element y gelöscht werden, wenn ein Element z existiert, so dass: $\frac{y}{1+\varepsilon} \leq z \leq y \; mit \; 0 < \varepsilon < 1$ wählbar;

bei guter Wahl von ε polynomiell (siehe T.H. Cormen, C.E. Leiserson, R.L. Rivest und C. Stein: Introduction to Algorithms,3rd ed., MIT Press, 2009)

```
function approx_rucksack (S,t,ε)
  n:=|S|; L0:={0};
  for i:=1 to n
    Li:=Li-1 ∪ (Li-1+xi);
    Li:=trim(Li,ε);
    Li:=Li-{all elements>t};
  end for
return largest number in Ln; // nahe opt. Lösung?
end
```

Top 500 Supercomputer - www.top500.org

Tihane-2 – Aktuelle Nummer 1 (6/2013 – 11/2015 akt.)

National Super Computer Center, Guangzhou, China

Anm.:

FLOP/s = *floating point operations per second*

Giga=10^9, Tera=10^{12}, Peta=10^{15},

Exa =10^{18}, Zetta =10^{21},

$2^{100} \sim 10^{30}$

Intel Xeon multicore processors + Intel Xeon Phi accelerators
~55 Petaflop/s theoret.max. und ~34 Petaflop/s für Benchmark-Test
Cores: 3.120.000, Memory: ~ 1 PetaB
Power: ~18 MWatt

Top 500 Supercomputer – www.top500.org, Scientific Computing – Simulationen

RANK	SITE	SYSTEM	CORES	RMAX (TFLOP/S)	RPEAK (TFLOP/S)	POWER (KW)
1	National Super Computer Center in Guangzhou China	**Tianhe-2 (MilkyWay-2)** - TH-IVB-FEP Cluster, Intel Xeon E5-2692 12C 2.200GHz, TH Express-2, Intel Xeon Phi 31S1P NUDT	3,120,000	33,862.7	54,902.4	17,808
2	DOE/SC/Oak Ridge National Laboratory United States	**Titan** - Cray XK7 , Opteron 6274 16C 2.200GHz, Cray Gemini interconnect, NVIDIA K20x Cray Inc.	560,640	17,590.0	27,112.5	8,209
3	DOE/NNSA/LLNL United States	**Sequoia** - BlueGene/Q, Power BQC 16C 1.60 GHz, Custom IBM	1,572,864	17,173.2	20,132.7	7,890
4	RIKEN Advanced Institute for Computational Science (AICS) Japan	K computer, SPARC64 VIIIfx 2.0GHz, Tofu interconnect Fujitsu	705,024	10,510.0	11,280.4	12,659.9
5	DOE/SC/Argonne National Laboratory United States	**Mira** - BlueGene/Q, Power BQC 16C 1.60GHz, Custom IBM	786,432	8,586.6	10,066.3	3,945
85	Vienna Scientific Cluster Austria	**VSC-3** - Oil blade server, Intel Xeon E5-2650v2 8C 2.6GHz, Intel TrueScale Infiniband ClusterVision	32,768	596.0	681.6	450

VSC-3 Nov 2015 (current ranking): rank 137

Analyse von Algorithmen

Schritt: von

berechenbaren Problemen

zu

Algorithmen

- Die Probleme, die wir betrachten, haben eine natürliche Größe, wie z.B. die zu verarbeitende Datenmenge *n* (**Größe des Inputs**). Die benötigte Ressource wird dann als Funktion der Inputgröße *n* beschrieben.

- Der Vorteil dieses Ansatzes ist:
  - Algorithmen ohne Laufzeitmessung vergleichbar.

Häufig auftretende Funktionen für Laufzeit (1)

Gegeben sei die Inputgröße n :

- c : konstanter Faktor unabhängig von n.
- n : lineares Wachstum (für Problemgröße n eine gute Situation).
- n ld n : logarithmisches Wachstum mit linearem Faktor, auch für große n noch anwendbar.
- n^2 : quadratisches Wachstum, z.B. doppelt verschachtelte Schleife.
- n^3 : kubisches Wachstum, z.B. dreifach verschachtelte Schleife, für große n problematisch.
- 2^n : exponentielles Wachstum, in der Praxis nicht mehr berechenbar.

Häufig auftretende Funktionen für Laufzeit (2)

In der Praxis fallen viele Algorithmen in 2 Klassen:

- Algorithmen mit **polynomieller Laufzeit $O(n^k)$** oder mit **exponentieller Laufzeit $O(2^n)$**

Gibt es auch Funktionen, die zwischen n^k und 2^n liegen?

- *ja* – $n^{\text{ld } n}$ (wächst schneller als n^k , aber langsamer als 2^n)

Einwand: $O(n^{100})$ ist doch auch nicht traktabel und ähnlich einem exponentiellem Aufwand ?

- Polynome solch hohen Grades treten in der Praxis kaum/nicht auf.
- Algorithmen mit hoch-gradigem polynomiellen Aufwand können üblicherweise durch effizientere Algorithmen mit niedrigerem Grad ersetzt werden.

Beispiel Laufzeiten (1)

Beispiel 1.

- Um die Größenordnungen der verschiedenen Funktionen zu veranschaulichen, berechnen wir einige Werte.

  Dazu wollen wir annehmen, dass ein Rechenschritt eine Millisekunde (ms) benötigt. Die Definition eines Rechenschritts ist vom Algorithmus abhängig und muss eine sinnvolle Bewertung eines Algorithmus ermöglichen; z.B bei einem Sortieralgorithmus könnte ein Rechenschritt aus dem Laden zweier Datensätze und einem Vergleich der Schlüsselwerte bestehen.

  Tab. 1 gibt den Zeitbedarf für Algorithmen mit unterschiedlichen Aufwand für Problemgrößen zwischen 10 und einer Million an; d.h. z.B. bei einer Problemgröße von 10 benötigt man 10, 33, 10^2, 10^3, 2^{10} Rechenschritte.

Beispiel Laufzeiten (2)

Tab. 1:

Problemgröße	n	n ld n	n^2	n^3	2^n
10	10 ms	33 ms	100 ms	1 s	1,02 s
100	100 ms	0,7 s	10 s	16,6 min	4E+19 a
1.000	1 s	10 s	16,6 min	11,6 d	3,3E+290 a
10.000	10 s	2,2 min	27,7 h	31,7 a	#NUM!
100.000	1,6 min	27,7 min	3,9 m	31709 a	#NUM!
1.000.000	16,6 min	5,5 h	31,7 a	31E+06 a	#NUM!

Legende:

ms…10^{-3}s, s…Sekunde, min…Minute, d…Tag, m…Monat, a…Jahr

#NUM! … Zahl für Tabellenkalkulator zu gross

Beachte: Zahl der Sekunden seit dem Urknall wird auf 10^{18} geschätzt.

Beispiel Laufzeiten (3)

Beispiel 2.

- Nun wollen wir die umgekehrte Fragestellung beantworten. Welche Problemgrößen kann man bei den unterschiedlichen Laufzeitverhalten innerhalb einer Sekunde/Minute/Stunde lösen. In **Tab. 2** nehmen wir wieder 10^{-3} s für einen Rechenschritt an. In **Tab. 3** werden die Problemgrößen für einen 10-mal so schnellen Rechner angegeben, d.h. der Rechner benötigt für einen Rechenschritt 10^{-4} s. **Tab. 4** gibt die Steigerung zwischen Tab. 2 und Tab. 3 an.
- Tab. 4 zeigt, dass trotz 10-facher Steigerung der Rechenleistung kaum die Problemgröße vergrößert werden konnte.

Beispiel Laufzeiten (4)

Tab. 2: Rechenschritt 10^{-3} s

	n	n ld n	n^2	n^3	2^n
1 sec	1000	140	32	10	10
1 min	60000	4895	245	39	16
1 h	3600000	204000	1897	153	22

Tab. 3: Rechenschritt 10^{-4} s (10-mal so schnell)

	n	n ld n	n^2	n^3	2^n
1 sec	10000	1003	100	22	13
1 min	600000	39300	775	84	19
1 h	36000000	1737000	6000	330	25

Tab. 4: Problemgrößensteigerung von Tab. 2 auf Tab. 3

	n	n ld n	n^2	n^3	2^n	
1 sec	10	7,2	3,2	2,2	1,3	(+3)
1 min	10	8,0	3,2	2,2	1,2	(+3)
1 h	10	8,5	3,2	2,2	1,2	(+3)

Laufzeitberechnung

- Exakte Laufzeitberechnungen sehr aufwendig und üblicherweise nicht wert des Aufwandes.
- Abstraktionen helfen die Berechnung der Laufzeit zu vereinfachen, jedoch die Aussagekraft nicht zu schmälern; z.B. für Laufzeit an^2+bn+c sind niedere Terme für große n nicht signifikant und kann durch n^2 approximiert werden.
- Man unterscheidet zwischen worst-case, average-case und best-case Laufzeit. Üblicherweise wird die worst-case Laufzeit betrachtet.
- Interessant ist die Laufzeit von Algorithmen für große Inputs (asymptotische Laufzeit). Dafür muss das asymptotische Wachstum von Funktionen näher betrachtet werden.