Uebungsstunde Theoretische Informatik

Thomas Briner, brinerth@iiic.ethz.ch, http://briner.computerscience.ch/ti

27.6.03

1 Nachbesprechung Uebung 7 (?)

1.1 CNF, DNF

1.1.1 Algo for 3-CNF-SAT in $\mathcal{O}(2^d n)$

Alle möglichen Belegungen anschauen: $\mathcal{O}(2^d)$
Für jede Belegung testen, ob die Formel erfüellt ist: $\mathcal{O}(n)$
$\rightarrow \mathcal{O}(2^d n)$

1.1.2 Satisfiability of DNF

Um die ganze Formel wahr zu machen, muss nur eine einzige Klausel wahr sein.

Um eine ganze Klausel wahr zu machen, darf kein Litteral sowohl positiv als auch negiert vorkommen.

Das ist aber alles, was ich testen muss. Ich muss nicht über die Belegungen iterieren. Deshalb ist das durchaus in polynomieller Zeit zu machen. Die Mulö sagt, es gehe in $\mathcal{O}(n)$.

1.1.3 Falsifiability of DNF

Da gibt es eine ganz elegante Lösung: Falsifiability bedeutet: es gibt keine Belegung um die Formel ϕ falsch zu machen. Wenn ich nun die ganze Formel negiere $\neg\phi$, dann darf die nie erfüllbar sein. Wenn ich nun das $\neg$ in die Formel reinziehe gibt das eine CNF und ich bin beim SAT Problem auf CNF.

1.2 $HC \leq_p TSP$

Das mit dem Coding war offensichtlich verwirrend. Gemeint war, dass man den Inputstring angibt, den die Turing Maschine frisst, die das TSP Problem lösen kann. Also wirklich das, was auf dem Band beim Start der Maschine stehen könnte. Dafür brauchts den Graph in irgendeiner Art codiert, z.B. als Liste der Kanten mit dem jeweiligen Gewicht und die obere Grenze für das TSP Problem.

Z.B. $21\#pq5\#ps2\#\ldots$

Um den HC über den Umweg TSP zu lösen, muss man sich überlegen, auf welchem Input welches Problem arbeitet:
So hat HC keine gewichteten Kanten, TSP schon. HC braucht nur den Graph als Input, TSP

zusätzlich noch die obere Grenze.

HC via TSP ginge also z.B. so:

- Algo C:
  Jede Kante im Graph für das HC wird mit 1 gewichtet. Als bound wird $|V|$ festgesetzt. $\mathcal{O}(n^2)$
- Algo D:
  Wenn TSP mit dieser Bound gelöst werden kann, dann gibt es auch einen HC im ursprünglichen ungewichteten Graph. $\mathcal{O}(1)$

und damit ist es geschafft.

2 Reduction Training

2.1 Zeige, dass Element Uniqueness als upper bound $\mathcal{O}(n \log n)$ hat (aus Vordip)

Definition Element Uniqueness:
Gegeben sind n Elemente und ein Gleichheitstest.Entscheide, ob alle Elemente paarweise voneinander verschieden sind.

Reduktionsbeweis:
Um Element Uniqueness zu lösen könnte einen Umweg nehmen, indem ich zuerst sortiere, dann immer die zwei benachbarten auf Gleichheit teste.

$\rightarrow ElementUniqueness \leq_p Sortieren$

Uebersetzungsalgorithmen:

- Algo C:
  Input Element Uniqueness ist eine Liste von Elementen. Die genau gleiche Liste kann auch als Input für das Sortieren benützt werden $\rightarrow$ nichts notwendig. $\mathcal{O}(1)$
- Algo D:
  Für jedes Paar von benachbarten Elementen nach dem Sortieren wird Gleichheit getestet. Das braucht also noch $\mathcal{O}(n)$.

Für den ganzen Umweg gilt also $\mathcal{O}(1) + \mathcal{O}(n \log n) + \mathcal{O}(n) = \mathcal{O}(n \log n))$

Damit ist die obere Grenze bewiesen.

2.2 Zeige, dass Clique-but-one NP-complete ist (aus Vordip)

Definition: k-Clique-but-one ist das Problem, eine Clique der Grösse k zu finden, bei der genau eine Kante für eine k-Clique fehlt.

Idee: Wir versuchen ein anderes NP-completes Problem auf Clique-but-one zu reduzieren. Dann müssen wir nur noch zeigen, dass Clique-but-one selber auch in NP liegt und das Ganze ist geschafft.

Reduktion von Clique auf Clique-but-one:

$\rightarrow Clique \leq_p Clique - but - one$

Vorgehen, um Problem k-Clique über Umweg l-Clique-but-one zu lösen:

- Als Input kommt ein Graph G=(V,E).
- Ich füge zwei neue Punkte hinzu, die zu allen alten Punkten eine Verbindung haben, aber keine untereinander.

  Zum alten Graph existiert genau dann eine k-Clique, wenn zum neuen eine (k+2)-Clique-but-one existiert.
- Uebersetzungsalgorithmen:
  - Algo C:
    Der Graph wird um diese zwei Punkte und die oben erwähnten Kanten erweitert. Das geschieht in $\mathcal{O}(n)$.
  - Algo D: Wenn eine (k+2)-Clique-but-one im neuen Graph existiert, dann gibts auch eine k-Clique im alten Graph. $\mathcal{O}(1)$

Jetzt noch zeigen, dass k-Clique-but-one selber in NP liegt:
Gegeben ein Witness, d.h. eine Liste der Knoten, die in k-Clique-but-one dabei sind, kann in polynomieller Zeit überprüft werden, ob das eine Clique-but-one ist.
Damit gilt nun:

Auch Clique-but-one ist NP-complete.

3 Markov Training

3.1 Write a markov algorithm that increments a binary number by one (aus Vordip)

Wichtig: Es ist nichts angegeben, wie die Zahl codiert ist. Das heisst wir müssen zuerst spezifizieren ob MSB first oder LSB first ist.

Mögliches Vorgehen: Um das Ganze völlig klar zu verstehen und keine Fälle zu vergessen, würde ich es zuerst einmal in Pseudocode hinschreiben:

```
PROCEDURE REVERSE (ARRAY OF INT:a)

    int i=a.length()-1;    // a[i] zeigt auf hinterste Stelle
    boolean finished = false;
    while (i!=-1 && !finished) {
        if (a[i]=1) {
            a[i]=0; i=i-1; // "Behalte 1" um eine Stelle weiterschieben
        }
        else {
            a[i]=1; finished=true;// Algo fertig
        }
    }
    if (i==-1) {           // falls Zahl um 1 wachsen muss
    neue Stelle vorne erzeugen;
    eine 1 dort hinschreiben;
}
```

Und jetzt in Markov:
zuerst Alphabet festlegen: (Das was im Input und im Output vorkommen darf)

$A = \{0, 1\}$

dann Marker, Klammer offen lassen, nach Bedarf mehr dazufügen
$M = \{$

dann Regeln:

Faustregel: Im normalem Ablauf vorgehen, aber die Regeln von unten her auffüllen, d.h. mit der untersten beginnen.

1. Hinterstes Zeichen finden:

   Weil nur sequentiell (=vorne beginnend) Zugriff: Von vorne nach hinten arbeiten

- Erzeuge einen Marker:
  $\epsilon \rightarrow \alpha$
- Benötigten Marker in Markeralphabet eintragen
- Marker nach hinten schieben = jedes Alphabetsymbol überspringen:
  $\alpha 0 \rightarrow 0\alpha$
  $\alpha 1 \rightarrow 1\alpha$

  Wenn diese Regeln erledigt sind, dann ist α zuhinterst.

2. Jetzt würde der eigentliche Algo beginnen: Nach vorne gehen, bis an einer Stelle eine 0 gefunden oder zuvorderst angekommen.

   Achtung: Dafür brauche ich wieder einen Marker um anzuzeigen, bei welcher Stelle ich gerade bin. Kann ich dafür wieder das α brauchen? $\rightarrow$ Heikel, im Zweifelsfall: Neuer Marker β

   β bei Markerliste hinzufügen und das α in ein β umwandeln.
   $\alpha \rightarrow \beta$

   Invariante: Alle Zeichen hinter dem β wurden schon behandelt und solange es das β gibt, ist der Uebertrag noch nicht erledigt.

3. Jetzt mit dem β nach vorne gehen.

   - Falls das β hinter einer 0 steht, mache ich eine 1 daraus und bin fertig $\rightarrow$ Stoppregel.

     $0\beta \neg 1$

     Vorsicht: Immer wenn eine Stoppregel steht, muss ich sicher sein, dass zu diesem Zeitpunkt in jedem Fall alle Marker weggeräumt sind und nur noch Alphabetsymbole im String stehen.
   - Falls das β hinter einer 1 steht, mache ich eine 0 daraus und schiebe das β um eine Stelle nach vorne.

     $1\beta \rightarrow \beta 0$
   - Falls ich zuvorderst angekommen bin und immer noch ein β habe, muss ich noch eine Stelle anhängen und zwar eine 1 und dann bin ich fertig $\rightarrow$ Stoppregel.

     $\beta \neg 1$

     Wie merke ich, dass ich zuvorderst angekommen bin? Wenn weder eine 1 noch eine 0 vor dem β steht. Das kann ich machen, indem ich diese Regel nach den beiden vorhergehenden platziere.

Und das wars jetzt eigentlich. Aber funktioniert es jetzt auch wirklich? Vor allem an der Reihenfolge sind jetzt eventuell noch Aenderungen nötig $\rightarrow$ Versuch, alle repräsentativen Testfälle zu finden und spiel sie durch.

In diesem Fall wären das wohl mindestens diese Fälle: 100, 101, 11, 011.

Wir müessen tatsäechlich noch etwas ändern: Die Regel $\alpha \to \beta$ darf erst nach dem α nach hinten schieben kommen, sonst gehts nicht!

Hier noch einmal der ganze Algorithmus:

$$\begin{array}{l}
\mathcal{A} = \{0, 1\} \\
\mathcal{M} = \{\alpha, \beta\}
\end{array}$$

$$\begin{array}{rrcl}
0: & 1\beta & \to & \beta 0 \\
1: & 0\beta & \dashv & 1 \\
2: & \beta & \dashv & 1 \\
3: & \alpha 0 & \to & 0\alpha \\
4: & \alpha 1 & \to & 1\alpha \\
5: & \alpha & \to & \beta \\
6: & \varepsilon & \to & \alpha
\end{array}$$

3.2 Write a markov algorithm that calculates modulo 3 of a number in unary notation

Unäre Notation bedeutet: soviele 1en, wie die Zahl gross ist.

Wie würdest du einem Kind, das noch nicht teilen kann, die modulo 3-Berechnung erkläeren? $\to$ so oft 3 abzäehlen, bis es nicht mehr geht. Das ist genau das, was unser Markov Algorithmus auch macht.

Es braucht nur eine einzige Regel:

$$111 \to \epsilon$$

Wenn der Algorithmus anhält, steht nur noch der Rest modulo 3 da.

3.3 Write a markov algorithm that reverses a string

Idee: Zeichen vorne wegnehmen und an den Schluss des ursprünglichen Wortes schieben. Wenn ich vorne beginne, wird dadurch die Reihenfolge invertiert.

Mögliches Vorgehen:

- Marker erzeugen, der anzeigt, bis wohin die Zeichen schon nach hinten geschoben wurden: α

  $\varepsilon \rightarrow \alpha$

- Immer nächste Zeichen nach α in ParallelAlphabet 0', 1' übersetzen, dazugehörige 0, 1 löschen

  $\alpha 1 \rightarrow \alpha 1'$
  $\alpha 0 \rightarrow \alpha 0'$

- Zeichen Parallelalphabet nach hinten schieben

  $1'1 \rightarrow 11'$
  $1'0 \rightarrow 01'$
  $0'1 \rightarrow 10'$
  $0'0 \rightarrow 00'$

- Wenn wieder ein α vorne eingefüegt wurde, heisst das, es hat keine Zeichen mehr zum nach hinten schicken.

  Jetzt müssen nur noch die Marker gelöscht werden und die Zeichen des Parallel Alphabets zurück übersetzt werden.
  Dafür brauche ich aber noch einen zweiten Marker β, der beim nach hinten Wandern aufräumt.

  $\alpha\alpha \rightarrow \beta$

  $\beta 0' \rightarrow 0\beta$
  $\beta 1' \rightarrow 1\beta$
  $\beta \dashv \varepsilon$

- Nun muss nur noch die Reihenfolge richtig gestellt werden und das wars:

  $\mathcal{A} = \{0, 1\}$
  $\mathcal{M} = \{0', 1', \alpha, \beta\}$

$$
\begin{array}{rrcl}
0: & \beta 0' & \rightarrow & 0\beta \\
1: & \beta 1' & \rightarrow & 1\beta \\
2: & \beta & \dashv & \varepsilon \\
3: & \alpha\alpha & \rightarrow & \beta \\
4: & 1'1 & \rightarrow & 11' \\
5: & 1'0 & \rightarrow & 01' \\
6: & 0'1 & \rightarrow & 10' \\
7: & 0'0 & \rightarrow & 00' \\
8: & \alpha 1 & \rightarrow & \alpha 1' \\
9: & \alpha 0 & \rightarrow & \alpha 0' \\
10: & \varepsilon & \rightarrow & \alpha
\end{array}
$$