Informatik ⊂⊃ – Blatt 10

Rasmus Diederichsen

13. Juli 2014

Aufgabe 10.1

a)

$\mathcal{A}$ berechnet die Funktion $f(x) = 3^{3^x}$, für $x > 0$.

b)

Das uniforme Kostenmaß ist nur sinnvoll, wenn primitive Operationen (Addition, Multiplikation) wirklich in $\mathcal{O}(1)$ durchgeführt werden können. Für reale Computer kann dies je nach Architektur nur für Zahlen $\leq 2^{32}$ oder 2^{64} passieren, da die Register keine größeren Zahlen fassen können.

Die Werte, die y in $\mathcal{A}$ annimmt, laufen jedoch schon für relativ kleine x ($x \lessapprox 3.366$ bei 64 Bit Integralzahlen) über diesen Wertebereich und sind dann nicht mehr in 32 oder 64 Bit darstellbar. Die arithmetischen Operationen sind dann nicht mehr atomar und der Aufwand wächst mit der Eingabegröße, ist also nicht mehr konstant.

c)

Da der Wert von y in jedem Schleifendurchlauf mit 3 potenziert wird, braucht Schleifendurchlauf i $3(3^i \log 3)^2$ viele Schritte. Die Gesamtzahl an Schritten ist somit

$$f(x) = \sum_{i=1}^{x} 3(\log_2 3^{3^i})^2 = \sum_{i=1}^{x} 3(3^i \log_2 3)^2 \in \mathcal{O}\left(3^{2x}\right)$$

Aufgabe 10.2

a)

Das zugehörige Entscheidungsproblem ist

$$\text{Gibt es ein } x \in \mathbb{R}^n, \text{ sodass } c^T x \geq k \text{ mit } Ax \leq b?$$

b)

Optimierungsalgorithmus

```
def optimize(c, A, b):
    import numpy as np
    upperLim = np.sum(c) # groesser geht's nicht
    lowerLim = 0

    return binarySearch(0, upperLim)

def 𝒜(k):
        ... # entscheide ob max >= k ist mit c, A, b

    def binarySearch(lower, upper):
        if (upper == lower) return upper
        if (𝒜(upper)):
            return upper # wenn es ein Element >= upper gibt, muss es
                == upper sein
        if (not 𝒜(lower + 1)):
            return lower # wenn es kein Element > lower gibt, muss das
                 maximum == lower sein
        if(𝒜(lower + (upper + lower) / 2)): # weiter in oberer
             Haelfte suchen
            return binarySearch(lower + (upper + lower) / 2, upper)
        else: # weiter in unterer Haelfte suchen
            return binarySearch(lower, upper - (upper + lower) / 2)
```

Im Algorithmus wird der Suchraum in jeder Iteration halbiert, es kann also maximal $\log_2 n$ viele rekursive Aufrufe geben, und auch das nur, falls $c = \{1\}^n$. Für jeden Aufruf von `binarySearch` wird einmal $\mathcal{A}$ aufgerufen, die Gesamtlaufzeit ist somit $\in \mathcal{O}\left(t_{\mathcal{A}}(n, m) \log_2 n\right)$. Ist man auch an dem zu dieser Lösung führenden x interessiert, so errechnet man dies aus den n Ungleichungen $Ax \leq b$, mithilfe von $c^T x = k$.

Aufgabe 10.3

a)

Man untersucht nacheinander alle Terme T_i darauf, ob sie ein Literal und seine Negation enthalten. Falls nein, so kann man alle Literale so belegen, dass sie `true` sind und man terminiert, da nur ein T_i `true` sein muss. Falls ja, so geht man weiter zur nächsten Klausel.

Ist auch der letzte Term unerfüllbar (der worst case), so hat man alle Literale nur einmal angeschaut, da es pro Literal nur eine Möglichkeit gibt, dass es `true` wird. Der Aufwand ist also maximal linear.

b)

Das Theorem gälte, wenn die Überführung einer beliebigen Formel in DNF auch in polynomieller Zeit durchführbar wäre. Dieses Problem ist jedoch (anscheinend)

bewiesenermaßen *NP*-schwer, und daher nur in P, wenn es auch in *NP* ist und $P = NP$ gilt. Dies ist nicht bewiesen, daher gilt der Beweis nicht.

Aufgabe 10.4

(1) $(x_{Stannis,1} \wedge x_{Sansa,1}) \vee (\neg x_{Stannis,1} \wedge \neg x_{Sansa,1})$

(2) $\bigwedge_t \bigvee_{v_1 \neq v_2} \neg(x_{v_1,t} \vee x_{v_2,t})$

→ Für jeden Zeitpunkt müssen mindestens zwei ungleiche Personen *nicht* in Tyrions Gemächern sein, über die anderen wird keine Aussage getroffen.

Es resultieren folgende Klauseln:

$(\neg(x_{Stannis,1} \vee x_{Sansa,1}) \vee \neg(x_{Stannis,1} \vee x_{Shae,1}) \vee \neg(x_{Stannis,1} \vee x_{Sandor,1}) \vee \neg(x_{Sansa,1} \vee x_{Shae,1}) \vee \neg(x_{Sansa,1} \vee x_{Sandor,1}) \vee \neg(x_{Shae,1} \vee x_{Sandor,1})) \wedge (\neg(x_{Stannis,2} \vee x_{Sansa,2}) \vee \neg(x_{Stannis,2} \vee x_{Shae,2}) \vee \neg(x_{Stannis,2} \vee x_{Sandor,2}) \vee \neg(x_{Sansa,2} \vee x_{Shae,2}) \vee \neg(x_{Sansa,2} \vee x_{Sandor,2}) \vee \neg(x_{Shae,2} \vee x_{Sandor,2})) \wedge (\neg(x_{Stannis,3} \vee x_{Sansa,3}) \vee \neg(x_{Stannis,3} \vee x_{Shae,3}) \vee \neg(x_{Stannis,3} \vee x_{Sandor,3}) \vee \neg(x_{Sansa,3} \vee x_{Shae,3}) \vee \neg(x_{Sansa,3} \vee x_{Sandor,3}) \vee \neg(x_{Shae,3} \vee x_{Sandor,3}))$

(3) $\bigwedge_{t_1 \neq t_2} \neg(x_{Stannis,t_1} \wedge x_{Stannis,t_2}) \wedge \neg(x_{Shae,t_1} \wedge x_{Shae,t_2})$

Es resultieren die Klauseln:

$\neg(x_{Stannis,1} \wedge x_{Stannis,2}) \wedge \neg(x_{Shae,1} \wedge x_{Shae,2}) \wedge \neg(x_{Stannis,1} \wedge x_{Stannis,3}) \wedge \neg(x_{Shae,1} \wedge x_{Shae,3}) \wedge \neg(x_{Stannis,2} \wedge x_{Stannis,3}) \wedge \neg(x_{Shae,2} \wedge x_{Shae,3})$

(4) $(x_{Sansa,1} \wedge x_{Sansa,2}) \vee \neg(x_{Sansa,1} \vee x_{Sansa,2}) \wedge \neg x_{Sansa,3}$

(5) $(x_{Sansa,2} \wedge x_{Shae,2}) \vee \neg(x_{Sansa,2} \vee x_{Shae,2})$

→ Sansa und Shae müssen sich entweder beide in Tyrions Gemächern aufgehalten haben, oder sie waren beide nicht dort. Zwar müssten sie dann *beide* am selben anderen Ort gewesen sein, dies spielt hier aber keine Rolle und ist nicht direkt darstellbar, da die einzigen Orte Tyrions Gemächer und Nicht-Tyrions-Gemächer sind.

(6) $x_{Shae,2}$

(7) $\bigwedge_t x_{Shae,t} \rightarrow y_t$. Dies lässt sich umformen zu $\bigwedge_t \neg x_{Shae,t} \vee y_t$. Es resultieren die Formeln

$(\neg x_{Shae,1} \vee y_1) \wedge (\neg x_{Shae,2} \vee y_2) \wedge (\neg x_{Shae,3} \vee y_3)$

(8*) $\bigwedge_{t_2 > t_1} \neg y_{t_1} \rightarrow \neg y_{t_2}$, was äquivalent ist zu $\bigwedge_{t_2 > t_1} y_{t_1} \vee \neg y_{t_2}$. Es resultieren die Klauseln

$(y_1 \vee \neg y_2) \wedge (y_1 \vee \neg y_3) \wedge (y_2 \vee \neg y_3)$

Wir sind nun in der Position, einige Schlussfolgerungen zu machen.

1. Wegen (6) wissen wir, dass $x_{Shae,2} = \texttt{true}$.
2. Wegen (7) und 1. wissen wir, dass $y_2 = \texttt{true}$.
3. Wegen (5) und 1. wissen wir, dass nicht nur $x_{Shae,2} = \texttt{true}$, sondern auch $x_{Sansa,2} = \texttt{true}$, da beide am gleichen Ort waren.
4. Wegen (4) und 3. wissen wir, dass $x_{Sansa,1} = \texttt{true}$.
5. Wegen (1) und 4. wissen wir, dass $x_{Stannis,1} = \texttt{true}$, da beide am gleichen Ort waren.
6. Wegen (3) und 5. wissen wir, dass $x_{Stannis,2} = \texttt{false}$.
7. Wegen (2) und 3. wissen wir auch, dass $x_{Sandor,2} = \texttt{false}$ Alle Konjunkte müssen wahr sein, im zweiten Konjunkt sind jedoch wegen 3. alle Disjunkte `false`, die Shae und Sansa enthalten. Also muss $\neg(x_{Stannis,2} \vee x_{Sandor,2})$ `true` sein, mithin $x_{Stannis,2} = \texttt{false}$ und $x_{Sandor,2} = \texttt{false}$.
8. Wegen (3) und 6. wissen wir, dass $x_{Stannis,3} = \texttt{false}$.
9. Wegen (3) und 1. wissen wir, dass $x_{Shae,3} = \texttt{false}$.
10. Wegen (4) wissen wir, dass $x_{Sansa,3} = \texttt{false}$
11. Der einzige, der zur Stunde 3 bei Tyrion gewesen sein kann, war Sandor.

Der Mörder muss also Sandor Clegane sein.

Aufgabe 10.5

Aufgabe 10.6

Für $\chi = \text{SAT}$ ist die Lösung relativ simpel. Sei n die Anzahl der vorkommenden Variablen. Die folgende Funktion gibt nach Eingabe einer Klauselmenge mithilfe von $\mathcal{A}_\chi$ einen Zeugen für $\mathcal{I}^*$ in Form einer Assoziationsliste zurück.

Hierbei sei `solvedp` ein Prädikat, dass bestimmt, ob ein Satz Formeln mit gegebenen Variablenzuweisungen bereits erfüllt ist, und `variables` retourniere die Liste von Variablen der SAT-Instanz.

```
(defun witness (problem)
   (labels ((solve (formulas cur_assignments)
      (cond ((solvedp formulas cur_assignments) cur_assignments)
            ((not (A_χ formulas cur_assignments)) nil)
            (t (or (solve formulas
                          (cons (cons (find-if #'unassignedp
                                               (variables formulas)
                                                   )
                                      t)
                                cur_assignments))
                   (solve formulas
```

```
11                          (cons (cons (find-if #'unassignedp
12                                              (variables formulas)
                                                    )
13                                      nil)
14                                cur_assignments)))))))
15        (solve problem nil)))
```

In Worten geht man wie folgt vor. Man betrachtet nacheinander alle in $\mathcal{I}^*$ vorkommenden Variablen $x_1, \ldots, x_n$. Man setzt nun die aktuelle Variable x_i auf `true`, reduziert die Klauselmenge, sodass sie wieder nur Variablen enthält und prüft, ob das um eine Variable reduzierte Problem noch immer lösbar ist.

Falls ja, so behält man die soeben getroffene Zuweisung und wendet sich der nächsten freien Variable zu.

Falls nein, so wählt man stattdessen $x_i =$ `false`, reduziert die Klauselmenge und geht zur nächsten Variable (das reduzierte Problem ist nun auf jeden Fall lösbar, da wir bereits wissen, dass die vorliegende eine Ja-Instanz ist).

Irgendwann ist man beim trivialen Problem `true` angekommen und weiß nun, dass die gespeicherten Zuweisungen einen Zeugen von $\mathcal{I}^*$ darstellen. Ob es noch andere gibt, weiß man nicht.

Dieses Schema ließe sich theoretisch auch auf andere Probleme anwenden. Diese könnte man entweder auf SAT reduzieren (weil SAT *NP*-vollständig ist), oder die Reduktion des Problems geschieht auf andere Weise. Im Graph Coloring-Problem würde man einem beliebigen Knoten eine Farbe zuordnen und ihn aus dem Graphen entfernen und für alle Nachbarknoten seine Farbe aus der Menge erlaubter Farben entfernen. Wiederum prüft man mit $\mathcal{A}_\chi$, ob der geschrumpfte Graph noch colorierbar ist. Falls ja, geht man vor wie mit dem ersten Knoten, falls nein, so backtracked man und wählt die nächste Farbe für den ersten Knoten.