Aufgabenblatt 4

Pseudocode der NTM

Um das CLIQUE Problem zu entscheiden benutzen wir eine Turingmaschine mit 2 Bändern. Im ersten Schritt werden alle Möglichkeiten betrachtet mit n Knoten jeweils einen davon mit allen anderen zu verbinden. Dazu erstellen wir Turingmaschinen mit den Inputs $A(v_i)$, $A(v_i)\#A(v_j)$, $A(v_i)\#A(v_j)\#A(v_l)$, $\ldots$, wobei die Indizes von v hier jeweils paarweise verschieden sein sollen. Das bedeutet $\sum_{k=1}^{n}\binom{n}{k} = 2^n$ viele Turingmaschinen wobei k die Anzahl der aneinander gereihten Listen sein soll. Diese Möglichkeiten werden auf das zweite Band geschrieben und am ersten Band bleibt der Originalinput stehen. (Die leere Turingmaschine wird sofort verworfen)

Danach wird die Binärzahl, die ganz rechts im Eingabeband steht, dupliziert und auf das Band 2 geschrieben ganz nach rechts geschrieben. Im nächsten Schritt wird die Anzahl der Knotenlisten, welche sich auf dem Band 2 befinden, abgezählt indem wir die kopierte zahl k vom Band 1 als Hilfe benutzen. Wenn die Anzahl ungleich k ist, dann wird die TM verworfen, weil eine Clique mit der Größe k nur dann existieren kann, wenn in einer Menge von Knoten jeder mit jedem verbunden ist. (daher wenn sie vollständig ist) Dadurch stellen wir fest, dass es hierbei vermutlich um eine Clique handeln könnte.

Der letzte Schritt ist es um zu überprüfen, ob in jeder Knotenliste, welche sich auf dem Band 2 befindet, k Knoten vorkommen. Dazu nehme man einen Knoten v von der ersten Knotenliste und überprüfe, ob dieser Knoten v in allen anderen Knotenlisten enthalten ist.

Notation. $|A(v_1)\#\ldots A(v_i)\#\#A(v_n)\#| = l_g$, $|\#bin(k)| = l_k$, Inputlänge: $l_g + l_k = l$, n …Anzahl der Knoten im Graph.

Listing 1: Pseudocode zur NTM

```
# iter1 ... Iterator for tape 1
# iter2 ... Iterator for tape 2

# START GUESS and CHECK

counter = 1
while A(v_i) not A(v_n)      # Oracle, Create 2^n TM'S
  new TM[A(v_i)][counter]
    do COPY to tape2
  new TM[A(v_i)][counter]
    do JUMP
  counter++

# COPY bin(k) to tape 2
duplicate( bin(k), bin_2(k), tape1.end() )

# CHECK LENGTH
if |A(v_i) on tape2| != bin_2(k) then
  delete this.TM

# DELETE RIGHT
```

```
loop = l_g + l_k
while loop < l_g + 2 * l_k
  tape1[loop] = blank
  loop++

# COPY LEFT VERTEX
amount = bin(k)

# CHECK IF 0
while bin(k) > 0
  # DO NEXT VERTEX
  if |tape2[A(v_1)]| > 0 then
    cut( tape2[A(v_1)][1], tape1.end() )
    vertex = tape1[last_bin]
  else
    delete this.TM

  # CHECK VERTEX
  i = 2
  isInAll = True
  while i <= amount
    loop = 1
    isThere = False
    while loop <= |tape2[A(v_i)]|
      if vertex == tape2[A(v_i)][loop] then
        isThere = True
        break
      loop++

    if isThere == False then
      isInAll = False
      break

    # TAKE NEXT VERTEX
    i++

  # DECREMENT COUNTER
  if isInAll == True
    bin(k)--

# EXIT
```

Platzkomplexität

Um die Platzkomplexität zu bestimmen, betrachten wir zuerst die Codierung von den einzelnen Knoten an. Wir nehmen an, dass wir im Graph n Knoten haben. Jeder Knoten ist in Binärdarstellung kodiert, d.h. jeder Knoten hat eine Länge von $l = \lceil log_2 n \rceil$ mit der optimalen Kodierung. Nehmen wir an, dass wir einen Vollständigen Graphen K_n hätte, welcher wiederum eine Max-Clique darstellt. Laut der Kodierung von dem Graph braucht man n Zahlen für jede Knotenliste und Damit jeder Knoten mit jedem verbunden ist, braucht man dazu noch n Knotenlisten.
Die Kodierung für einen Vollständigen Graphen würde folgendermaßen ausschauen:

$$bin(v_1) - bin(q_{(1,1)}) - bin(q_{(1,2)}) - \cdots - bin(q_{(1,n-1)}) \# \ldots \# bin(v_n) - bin(q_{(n,1)}) - bin(q_{(n,2)}) - \cdots - bin(q_{(n,n-1)}) \#\# bin(k)$$

Der maximale Speicher wäre dann:

$$S_T(n) = n * (n + n * l) = n * (n + n * \lceil log_2 n \rceil) = n^2 + n^2 * \lceil log_2 n \rceil = \mathcal{O}(n^2 * log n)$$

Wir haben gezeigt, dass $S_{CLIQUE}(n) \in NSPACE$.

Zeitkomplexität

Für die Zeit werden wir uns wieder Beispiel mit dem vollständigen Graph betrachten, welches in der Platzkomplexität behandelt worden ist (Da dies ja der worst case ist).

START GUESS and CHECK Als erstes erzeugen wir 2^n Turingmaschinen um eine NTM zu simulieren. Für die Zeitanalyse entscheidend ist allerdings nur die Schleife, die über den gesamten Input iteriert. Dieser hat, wie bereits in der Speicherbedarfanalyse erläutert $n^2 \cdots \log(n)$ Länge. Damit folgt

$$t_1(n) = \mathcal{O}(n^2 \cdot \log n)$$

.

CHECK LENGTH Im nächsten Schritt wird nur eine Binärzahl dupliziert.

$$t_2(n) = \mathcal{O}(\log n)$$

DELETE RIGHT Danach wird der rechte Teil auf dem Band 1 gelöscht.

$$t_3(n) = \mathcal{O}(\log n)$$

COPY LEFT VERTEX Der letzte Schritt ist zu überprüfen, ob eine clique existiert. Für diese Berechnung ist das Kopieren eines Knoten notwendig. Dies ist in

$$t_4(n) = \mathcal{O}(\log n)$$

möglich.

Als letztes werden über beide Bänder für jeden zu überprüfenden Knoten (also maximal n viele) von links nach rechts und nochmal von rechts nach links iteriert, damit man überprüft ob der Knoten v in jeder Knotenliste enthalten ist.

$$t_5(n) = \mathcal{O}(n \cdot n^2 \cdot log n) = \mathcal{O}(n^3 \cdot \log n)$$

Die gesamte Zeitkomplexität ist dann gegeben durch:

$$T(n) = t_1(n) + t_2(n) + t_3(n) + t_4(n) + t_5(n) = \mathcal{O}(n^2 \cdot \log n) + \mathcal{O}(\log n) + \mathcal{O}(\log n) + \mathcal{O}(\log n) + \mathcal{O}(n^3 \cdot \log n) = \mathcal{O}(n^3 \cdot \log n)$$

Wir haben also gezeigt, dass $CLIQUE \in NP$ ist, da wir eine NTM haben, die $CLIQUE$ in polynomieller Zeit entscheiden kann.

Aufgabenblatt 5

Zeigen Sie, dass P abgeschlossen gegenüber der Kleenschen Hülle ist.

Sei $A \in P$ eine Sprache. Wir zeigen nun, dass $A^* \in P$. Da $A \in P$ wissen wir, dass eine deterministische Turingmaschine M_A existiert, die A in polynomieller Zeit entscheidet. Wir werden nun mit Hilfe von M_A, A^* in polynomieller Zeit wie folgt entscheiden:
Zunächst bemerken wir, dass $w \in A^*$ dann und nur dann wenn, genau eine der Folgenden Eigenschaften gilt,

1. $w = \epsilon$
2. $w \in A$
3. $\exists u, v : w = uv \quad v, u \in A^*$,

also entweder w ist das 'leere' Wort oder ein Wort aus A selbst oder zerlegbar in zwei Worte aus A^*.
Sei nun $w = a_1 \dots a_n, a_i \in \Sigma$ und bezeichne w_{ij} den Substring, der mit a_i beginnt und mit a_j endet. Wir erzeugen im folgenden Algorithmus eine Tabelle, sodass

$$table(i,j) = \begin{cases} 1, & w_{i,j} \in A^* \\ 0, & \text{sonst.} \end{cases} \tag{1}$$

1: **if** $w = \epsilon$ **then** accept
2: **else**
3:   **for** $l \leftarrow 1, n$ **do**
4:     **for** $i \leftarrow 1, n-(l-1)$ **do**
5:       $j \leftarrow i + l - 1$
6:       Run M_A on $w_{i,j}$
7:       **if** M_A accepts $w_{i,j}$ **then** $table(i,j) := 1$
8:       **else**
9:         **for** $k \leftarrow i, j-1$ **do**
10:           **if** $table(i,k) = 1$ and $table(k+1,j) = 1$ **then** $table(i,j) = 1$
11: **if** $table(1,n) = 1$ **then** accept
12: **else** reject

Algorithm 1: Entscheider

Wir betrachten also alle möglichen Substrings von w, beginnend mit Substrings der Länge $1, 2, \dots$ bis zum Substring der Länge n, also w selbst. Zunächst testet der Algorithmus in Zeile 5, ob der Substring selbst entscheidbar ist, falls nicht, so nutzen wir die oben erwähnte Eigenschaft 3 in Zeile 8-10 um zu sehen ob das Wort in zwei von M_A entscheidbare Worte zerlegbar ist. Letztendlich bleibt nur noch zu überprüfen, ob $table(1,n) = 1$ um zu entscheiden ob $w \in A^*$.
Wir haben also einen Algorithmus, der mit Hilfe von M_A entscheiden kann ob $w \in A^*$ oder nicht. Bleibt nur noch zu Überprüfen ob er dies auch in polynomieller Zeit tut. Offensichtlich sind die drei For-Schleifen entscheidend für die Laufzeit im Algorithmus. Die dritte For-Schleife ruft M_A und berücksichtigt man, dass M_A in polynomieller Zeit entscheidet, also in $\mathbb{O}(n^k), k \geq 0$ so ergibt sich insgesamt eine Laufzeit von

$$\begin{aligned} T(n) &= \mathbb{O}(n)\mathbb{O}(n)(\mathbb{O}(n^k) + \mathbb{O}(n)) \\ &= \mathbb{O}(n^{2+\max k,1}) \end{aligned} \tag{2}$$

Bemerkung. Dies ist ein Polynom und somit ist $A^* \in P$. □

Aufgabenblatt 6

Zeigen Sie, das SOLITAIRE in NP enthalten ist.

Implementierung in Jflap

Fünf Zustände haben hierfür ausgereicht. Die NTM wird in zwei Teile gespaltet. Im ersten Bereich wird Guess and Check durchgeführt und im zweiten werden die Reihen miteinander verglichen.

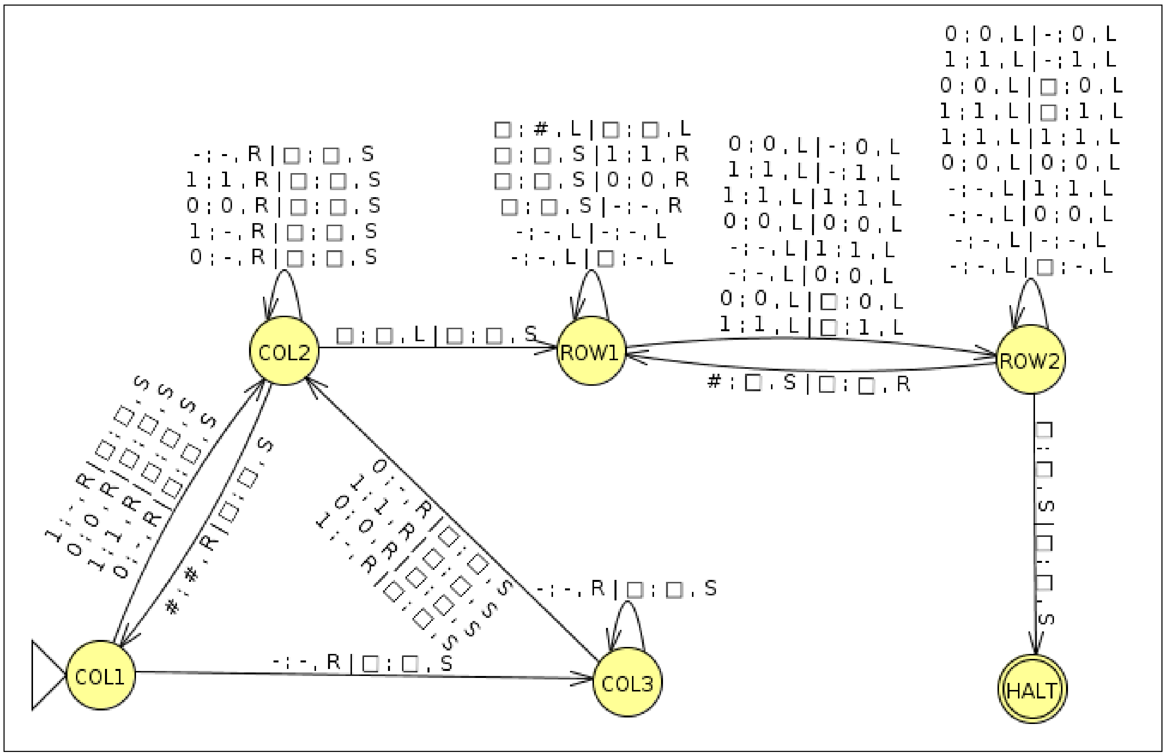

Abbildung 1: Turingmaschine

Pseudocode

Die NTM baut bei den Zuständen (COL1-3) auf das Prinzip von Guess and Check auf. Dabei wird der ganze Input nach der Reihe nach durchgegegangen. Die NTM schreibt zu Beginn eine 0 auf das zweite Band und haltet. Währenddessen wird das erste Band durchlaufen und alle Möglichkeiten ausprobiert, indem man alle 2^{n-1} Möglichkeiten pro Reihe auf neuen TMs durchprobiert.
Falls in einer Reihe nur '-' stehen, dann wird diese NTM verworfen, da dies nicht nach den Regeln von Solitaire ist. Die NTM führt diesen Vorgang bis zum Ende der Eingabe fort.

In der Zeile 2 des Pseudocodes finden wir eine big While-Schleife die 3 weitere innere Schleifen behinhaltet.

$$\langle G\rangle = \overbrace{\underbrace{Z_1}_{\substack{\longleftarrow\\ \longrightarrow\\ \longleftarrow}} \# \underbrace{Z_2}_{\substack{\longleftarrow\\ \longrightarrow\\ \longleftarrow}} \# \underbrace{Z_3}_{\substack{\longleftarrow\\ \longrightarrow\\ \longleftarrow}} \# \qquad \underbrace{Z_m}}^{\text{1.big while}\longrightarrow} \quad \begin{array}{l}\text{1.1 inner while}\longleftarrow\\ \text{1.2 inner while}\longrightarrow\\ \text{1.3 inner while}\longleftarrow\end{array}$$

Wenn nun das Ende erreicht wird, arbeitet die NTM von hinten nach vorne Reihe nach Reihe ab. Sie kopiert die letzte Reihe auf das zweite Band und vergleicht sie mit den anderen Reihen.

Aufgabenblatt 6 continued on next page...

Rootband	0	1	0	1	0	1
Helpband	-	-	1	0	0	1
Resulat auf dem Helpband	0	1	verwerfe		0	1

Die NTM verwirft, sobald sie eine ungültige Konfiguration findet, oder haltet wenn sie alle Reihen verglichen hat.

Listing 2: Pseudocode zur NTM

```
###################### Erster Teil #########################
while Z_i not Z_m                                       # 1. big loop
  # ROW1 creates guess and check
  while iter1 != square and iter1 != '#'                # 1.1 inner loop
    helpband[Z_i][1] = '0'
    # create TM for each element in row to simulate all outcomes
    new TM[Z_i][iter1] # consits of rootband and helpband
    iter1++

  # ROW2 goes back to the left on all new Turing machines
  while iter1 != square and iter1 != '#'                # 1.2 inner loop
    iter1--

  # ROW3 checks if row is empty,
  # if not confirm to the rules of solitaire and deletes this TM
  while iter1 == '-'                                    # 1.3 inner loop
    if iter1 == '1' or iter1 == '0' then
    change_state_to ROW1
    iter1++

  if helpband_k[iter1 - 1] != '1' then
    delete TM[Z_i]
Z_i++

###################### Zeiter Teil ########################
# COL1 copy                                             # 2 big loop
while rootband[iter1] != '#'
  helpband[iter2] = rootband[iter1]
  iter1--
  iter2--

goto (rootband.end() and helpband.end())

# compares rootband and helpband Z_m
#ROW1 and ROW2
  for Z_i = Z_m-1 downto Z_1                            # 3 big loop
    for iter1 = n downto 1                              # 3.1 inner loop
      if helpband[iter1] == '-' and rootband[Z_i][iter1] == '0' then
        helpband[iter1] = '0'

      if helpband[iter1] == '-' and rootband[Z_i][iter1] == '1' then
        helpband[iter1] = '1'

      if helpband[iter1] == '1' and rootband[Z_i][iter1] == '0' then
        delete TM[Z_i]

      if helpband[iter1] == '0' and rootband[Z_i][iter1] == '1' then
        delete TM[Z_i]
```

2. Ansalyse der Laufzeit

Für unsere Laufzeitanalyse berücksichtigen wir hauptsächlich die while-Schleifen, da diese die größte Bedeutung haben als Operationen wie Zuweisungen, Abfragen, etc ... (Man geht bei diesen Operationen davon aus, dass sie in konstanter Zeit ausgeführt werden können).

Um die Analyse zu vereinfachen Teilen wir das Programm in 2 Teile:

1. Analysieren wir zunächst den ersten Teil des Pseudocodes: Wir betrachten nun zunächst die erste while-Schleife “1. big loop”. Diese Schleife iteriert über alle Zeilen des Spielfeldes und arbeitet somit in $\mathbb{O}(m)$ Zeit. (Zur Erinnerung: Das Spielbrett besitzt $m \times n$ Felder)

Nun werden in dieser Schleife 3 weitere Schleifen, “1.1 inner loop”, “1.2 inner loop” und “1.3 inner loop” ausgeführt. Betrachten wir zunächste die Erste der Drei:

Wir erzeugen für Elemente in Reihe Z_i, in der wir uns gerade befinden 2^{n-1} Turingmaschinen, um eine nichtdeterministische Turingmaschine zu simulieren. (Man beachte, dass dies für die Laufzeitanalyse nicht bedeutsam ist, da nur die Simulation der NTM exponentielle Laufzeit besitzt!)

In den 2 nachfolgenden Inneren Schleifen arbeitet die NTM alle erzeugten Turingmaschinen parallel ab. In diesen 2 Schleifen wird lediglich der Lesekopf zurückgeshiftet und dann die Regeln des SOLITAIRE, in der Reihe Z_i, überprüft. Da jede Zeile genau n Elemente besitzt, ergibt sich somit für den ersten Teil der Analyse eine Laufzeit von

$$T_1(n) = \mathbb{O}(m) * (\mathbb{O}(n) + \mathbb{O}(n) + \mathbb{O}(n)) = \mathbb{O}(mn) \tag{1}$$

2. Analyse des zweiten Teils: Zunächst bemerken wir, dass die einzelne while-Schleife “2. big loop”, nur vom Hauptband aufs Hilfsband kopiert, also $\mathbb{O}(nm)$ Zeit benötigt.(Gleiches gilt für das goto)

Betrachten wir nun die for-Schleife, “3. big loop”:

Wir iterieren wieder über die Zeilen und dann über die Elemente in der Zeile also ergibt sich ingesamt $\mathbb{O}(mn)$ als Laufzeit hier. (Die if Abfragen können wie oben bereits erwähnt in konstanter Zeit abgefragt werden)

Insgesamt ergibt das für den zweiten Teil

$$T_2(n) = \mathbb{O}(nm) + \mathbb{O}(nm) + \mathbb{O}(nm) = \mathbb{O}(nm). \tag{2}$$

Die gesamte Laufzeit ist somit

$$T(n) = T_1(n) + T_2(n) = \mathbb{O}(nm) + \mathbb{O}(nm) = \mathbb{O}(nm). \tag{3}$$

Bemerkung. Wenn wir OBdA annehmen $m \leq n$ (ansonsten drehe das Spielbrett) folgt $\mathbb{O}(nm) \leq \mathbb{O}(n^2)$ und somit ist die Laufzeit durch ein Polynom 2ten Grades gegeben, daher eine polynomielle Laufzeit.

Damit haben wir also eine NTM, die in polynomieller Laufzeit SOLITAIRE entscheidet und somit folgt, dass die Sprache SOLITAIRE in NP enthalten ist. □