Übungsblatt 09

Til Mohr, 405959
Andrés Montoya, 405409
Marc Ludevid Wulf, 405401

Aufgabe 4

```
1   Input G=(V,E)
2
3   // Get optimal tour cost
4   def getTourCostOpt(G=(V,E)):
5       n = size(V)
6       E = sort(E)
7       E' = getLast(E,n) // Returns Set with last n elements in E
8       opt = sumWeights(E')
9       while true:
10          b = opt − 1
11          if TSP-E(G,b)):
12              opt = b
13          else:
14              break
15      return opt
16
17  // Calc Graph, which only contains the tour
18  opt = getTourCostOpt(G)
19  for e ∈ E:
20      G' = (V, remove(E,e))
21      if TSP-E(G',opt):
22          G = G'
23
24  // Calulate the path
25  v_start = getFirst(V)
26  K = {v_start}
27  e = getRandomEdgeFrom(E, v_start)
28  E = remove(E, e)
29  while true:
30      u = getNeighbourOf(E, getLast(K)) // get the only neighbour
31      K = add(K, u) // add u to K
32
33
34  return K
```

Laufzeit:
Die Methode `getTourCostOpt(G)` ist offensichtlich in polynomieller Zeit berechenbar. Das sortieren von E geht in quadratischer Zeit, das Lesen der letzten n Elemente in linearer Zeit, genauso das Aufsummieren der Kosten von den Kanten ein `E'`. Die Schleife in dieser Methode wird auch nur höchstens `opt` mal ausgeführt (abhängig von den Kantengewichten), da wir `opt` mit jedem Schleifendurchlauf dekrementieren. TSP-E`(G,b)` können wir ja auch in polynomieller Zeit berechnen.
Die Schleife im 2. Teil des Algorithmus wird für jede Kante in `E` ausgeführt. Also kann es nur maximal $n \cdot n$ Schleifendurchläufe geben. Offensichtlich ist auch jeder Schleifendurchlauf in polynomieller Zeit berechenbar.
Die Schleife im 3. Teil des Algorithmus wird genau n mal ausgeführt, da wir im Prinzip über jeden Knoten in `V` iterieren.
Damit ist der gesamte Algorithmus in polynomieller Zeit berechenbar.

Korrektheit:
Die Methode bestimmt für einen übergebenen Graphen `G` die Optimale Tour Kosten des TSP. Da jede Tour in `G` genau n viele Knoten besitzt, läuft diese Tour dann auch

über genau n vielen Kanten aus `E`. Da die Tour also höchstens über die n schwersten Kanten laufen kann, setzen wir unsere optimale Kosten auf die Summe der Kosten dieser Kanten. Nun überprüfen wir, ob es auch Touren gibt mit geringeren Gesamtkosten. Dazu könnten wir natürlich jede mögliche n-Kombination an Kanten überprüfen, doch es ist viel simpler, unsere bisherigen optimalen Kosten zu verringern (möglich, da alle Kosten in $\mathbb{N}$ liegen). So können wir schrittweise die Optimalen Tourkosten von `G` bestimmen.
In dem 2. Teil des Algorithmus bestimmen wir mithilfe der obigen Methode die optimalen Kosten von einer Tour in `G`. Anschließend überprüfen wir für jede Kante in `E`, ob diese für die Optimale Tour relevant ist. Falls nicht, so löschen wir diese. Mit diesem Teil minimieren wir also den Graphen `G`, sodass dieser nur noch die Kanten der optimalen Tour enthält.
Im letzten Teil müssen wir noch eine Reihenfolge der Knoten finden, in der die optimale Tour laufen kann. Dazu bestimmen wir uns einen Startknoten v_{start}, und gehen dann die Kanten der Knoten nacheinander entlang.
Also können wir mithilfe von TSP-E auch TSP bestimmen.

Til Mohr, 405959
Andrés Montoya, 405409
Marc Ludevid Wulf, 405401

Aufgabe 5

Hierzu nehmen wir uns eine Formel φ in 3-KNF mit den Klauseln k_i. Jedes k_i in φ ist daher der Form $(x_i \vee y_i \vee z_i)$. Nun teilen wir jedes k_i in zwei weitere Klauseln k_i' und k_i'', wobei wir auch noch für jede Klausel eine Variable c_i, und für alle Klauseln eine weitere Variable f einführen.
Aus k_i wird also $k_i' \wedge k_i'' = (x_i \vee y_i \vee c_i) \wedge (\bar{c_i} \vee z_i \vee f)$. Diese Konstruktion können wir in Linearer Zeit erstellen.

Korrektheit:

3-SAT $\Rightarrow$ NOT-ALL-EQUAL-SAT:
Sei φ aus 3-SAT mit gegebener Belegung der Variablen. Dann ist jede Klausel k_i der Form $(x_i \vee y_i \vee z_i)$.
Hieraus konstruieren wir eine neue aussagenlogische Formel φ' wie oben beschrieben, übernehmen dieselbe Belegung für die übernommenen Variablen und setzen f auf 0. Anschließend müssen wir noch jedes c_i belegen.

- Falls x_i oder y_i wahr ist, so setzen wir c_i auf 0. Damit ist zum einen die Klausel k_i' wahr, zum anderen gibt es in dieser Klausel mindestens sowohl ein wahres, als auch ein falsches Literal. Desweiteren ist dadurch (unabhängig von z_i) auch k_i'' wahr, da dort $\bar{c_i}$ enthalten ist. Auch diese Klausel besitzt sowohl ein wahres, als auch ein falsches Literal (f).
- Falls x_i und y_i falsch sind, muss z_i wahr sein. Dann setzen wir c_i auf 1. Damit ist die erste Klausel (k_i') wahr und besitzt auch wieder mindestens ein wahres, als auch ein falsches Literal. Die Klausel k_i' ist aufgrund von z_i auch wahr, und besitzt auch wieder ein wahres und ein falsches Literal (f).

Also können wir mit der erfüllenden Belegung von φ auch eine erfüllende Belegung von φ' konstruieren, sodass $\varphi' \in$NOT-ALL-EQUAL-SAT.

NOT-ALL-EQUAL-SAT $\Rightarrow$ 3-SAT
Sei φ' aus NOT-ALL-EQUAL-SAT mit der obig beschriebenen Form. φ' hat also eine erfüllende Belegung, wobei jede Klausel sowohl ein wahres, als auch ein falsches Literal besitzt. Hier kann es jedoch vorkommen, dass f 0 ist, aber auch 1. Wir finden nun eine Belegung für φ:

- Fall $y = 0$:
  Dann muss z_i 1 sein, oder c_i 0, damit k_i'' wahr ist. Ist z_i 1, dann ist offensichtlich k_i auch wahr.
  Ist z_i 0, so muss c_i 0 sein. Daraus folgt, dass x_i oder y_i wahr ist (da es sonst keine erfüllende Belegung für φ' gäbe). Also ist offensichtlich k_i auch wahr.
- Fall $y = 1$:
  Dann muss z_i 0 sein, oder c_i 1, damit diese Klausel zwei Literale mit unterschiedlichen Wahrheitswerten besitzt. Jetzt negieren wir die Belegung aller Variablen x, y, z für alle Klauseln.
  War z_i ursprünglich 0, dann ist z_i nach der Negation 1, weshalb k_i offensichtlich wahr ist.
  War z_i ursprünglich 1, dann ist c_i 1. Dann muss x_i oder y_i 0 sein, da sonst die Klausel k_i' keine Literale mit unterschiedlichen Wahrheitswerten hätte. Nach der Negation der Belegung ist also k_i wahr.

Also können wir für $\varphi' \in$NOT-ALL-EQUAL-SAT zeigen, dass $\varphi \in$3-SAT gilt.

Til Mohr, 405959
Andrés Montoya, 405409
Marc Ludevid Wulf, 405401

Aufgabe 6

(a) Hierzu bauen wir uns einen Verifizierer, welches als Zertifikat einen Vektor $x \in \{0,1\}^n$ übergeben bekommt, welches einfach codiert werden kann als $x_1...x_n$.
Unser Verifizierer überprüft nun Folgende Dinge:

- Eingabe korrekt formatiert (Linearer Zeitaufwand in n).
- Überstimmt die Vektordimension n von x überein mit der gegebenen Matrix A und dem Vektor b? (höchstens Quadratischer Zeitaufwand in $n \cdot m$, je nach Codierung der Matrix A)
- Berechnung von Ax (Quadratischer Zeitaufwand in $n \cdot m$)
- Abgleich, ob $Ax \geq b$ (Linearer Zeitaufwand in m).

Damit läuft der Verifizierer in polynomieller Zeit und $\{-1,0,1\}$-RESTRICTED INTEGER PROGRAMM ist in NP.

(b) Gesucht ist eine polynomielle Funktion die Instanzen von 3-SAT auf Instanzen von $\{-1,0,1\}$-RESTRICTED INTEGER PROGRAMMING (RIP) abbildet.
Für ein 3-SAT Problem mit k Variablen $y_0...y_{k-1}$ seien diese in $x \in \{0,1\}^{2k}$ kodiert als $x_{2i} = 1$ falls $y_i = True$ und $x_{2i+1} = 1$ falls $y_i = False$. Nachher werden wir sicherstellen, dass immer genau eins der beiden 1 ist, da eine Variable immer nur entweder $True$ oder $False$ sein kann.

Kodierung der Klauseln von 3-SAT:
Die ersten m Zeilen von A kodieren die m Klauseln, wobei $a_{j,2i}$ genau dann 1 ist wenn y_i in der j-ten Klausel vorkommt und $a_{j,2i+1}$ genau dann wenn $\overline{y}_i$ in der j-ten Klausel vorkommt. Alle anderen Einträge in der Zeile sind 0.
Um sicherzustellen, dass aus jeder Klausel immer mindestens eine Variable $True$ ist setzen wir $b_j = 1$ für alle $0 \leq j < m$.
Nun müssen wir noch sicherstellen, dass eine Variable immer nur $True$ oder $False$ sein kann. Dafür definieren wir weitere k Zeilen in A wobei $a_{m+i,2i}$ und $a_{m+i,2i+1}$ für $0 \leq i < k$ auf 1 gesetzt werden und alle anderen Einträge auf 0. Die Einträge $b_m...b_{m+k-1}$ werden wie oben auf 1 gesetzt.
Dies sichert zwar, dass mindestens eins von den beiden Einträgen x_{2i} und x_{2i+1} auf 1 gesetzt ist, aber es kann sein, dass beide gesetzt sind. Deswegen definieren wir weitere k Zeilen in A wobei $a_{m+k+i,2i}$ und $a_{m+k+i,2i+1}$ für $0 \leq i < k$ auf -1 gesetzt werden. Alle weiteren Einträge in A werden wieder auf 0 gesetzt und die Einträge $b_{m+k}...b_{m+2k-1}$ werden dieses Mal auf -1 gesetzt.
Dank der $2k$ hinzugefügten Zeilen in A ist nun sichergestellt das jede Lösung x auch wieder zu einer Lösung des 3-SAT Problems dekodiert werden kann, da jeder Variable eindeutig ein Wahrheitswert zugeordnet werden kann.

Diese Beschriebene Kodierung kann in linearer Zeit über die Länge des gegebenen 3-SAT Problems durchgeführt werden, da das Eingabewort nur ein Mal durchlaufen werden muss. Das Anhängen der weiteren $2k$ Zeilen ist ebenfalls in linearer Zeit, dieses Mal linear über die Anzahl an Variablen, durchgeführt werden. Also hat der Algorithmus eine polynomielle Zeit.

Korrektheit:

- $True$-Instanz von 3-SAT $\Rightarrow$ $True$-Instanz von RIP:
  Für eine gegebene $True$-Instanz von 3-SAT mit der Lösung $y = y_0...y_k$ gilt, dass dann auch das nach der oben vorgestellten Kodierung die Lösung $x = x_0...x_{2k}$ existiert die die Ungleichung $Ax \geq b$ erfüllt, da zum einen die

ersten m Zeilen der Ungleichung erfüllt, da in jeder Klausel des gegebenen Problems mit der dazugehörigen Lösung ein $True$ vorhanden ist. Auch werden auf jeden Fall die weiteren $2k$ Zeilen erfüllt sein, falls die Kodierung richtig durchgeführt wurde.

- $True$-Instanz von RIP $\Rightarrow$ $True$-Instanz von 3-SAT:
  Für eine gegebene $True$-Instanz von RIP mit der Lösung $x = x_0...x_{2k}$ gilt, dass man die Lösung zu der Lösung $y = y_0...y_k$ für das 3-SAT Problem dekodieren kann. Diese Lösung erfüllt die 3-SAT Aussage da in jeder der Klauseln mindestens eine der Variablen $True$ auswerten wird. Grund dafür ist die Kodierung in A.

Damit ist bewiesen, dass $\{-1, 0, 1\}$-RESTRICTED INTEGER PROGRAMMING NP-SCHWER ist.