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 opt == getTourCostOpt(G'):
22          G = G'
23
24  // Calulate the path
25  v_start = getFirst(V)
26  K = {v_start}
27  while true:
28      u = getNeighbourOf(getLast(K), K) // sets u to the neighbour of the last
            element in K, which is not in K
29      K = add(K, u) // add u to K
30
31
32  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.

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 die Variable $f = 0$ einführen.
Aus k_i wird also $k_i' \wedge k_i'' = (x_i \vee y_i \vee c_i) \wedge (\bar{c_i} \vee z_i \vee f)$. Hierbei wird c_i auf $\neg(x_i \vee y_i)$ gesetzt.
Diese Konstruktion können wir in Linearer Zeit erstellen.

Korrektheit:
Sei φ in 3-KNF 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.

- 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, aber z_1 wahr, 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).
- Falls keines der drei Literale wahr ist, so gibt es ja keine Erfüllende Belegung der Variablen, weshalb die gesamte Formel nicht in 3-SAT ist. Trotzdem setzen wir hier ja c_i auf 1, weshalb ja die Klausel k_i' war ist. Jedoch ist die Klausel k_i'' folglich nicht wahr, weshalb die resultierende Formel auch nicht in NOT-ALL-EQUAL-SAT ist.

Damit ist der Wahrheitsgrad von φ' immer genau derselbe wie von φ für jede Belegung.
Hat also φ keine erfüllende Belegung (nicht in 3-SAT), so hat auch φ' keine erfüllende Belegung (nicht in NOT-ALL-EQUAL-SAT).
Hat aber φ eine erfüllende Belegung (ist in 3-SAT), so hat φ' eine erfüllende Belegung, die die gleichen Variablen gleich belegt, und weitere Variablen so belegt, dass jede Klausel mindestens ein wahres und ein falsches Literal besitzt (ist in NOT-ALL-EQUAL-SAT).

Übungsblatt 09

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 n).

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

(b)