Theoretische Grundlagen der Informatik II
Blatt 6

Markus Vieth

10. Dezember 2015

Aufgabe 1

Beh: Travelling-Salesman $\in$ NP:

Bemerkung: TSP ist ein Optimierungsproblem. NP ist aber nur auf Entscheidungsprobleme definiert. Deshalb wird das TSP Entscheidungsproblem, ob die Länge des Kreises kleiner als ein übergebenes k ist, betrachtet.

Bew:

GUESS

Wähle nicht-deterministisch einen Vektor mit $|V| = n$ Kanten und kopiere die erste Kante an das Ende des Vektors.

CHECK

```
for ( int i = 0; i < n ; i++)
  for (int j = i; j < n; j++)
    if (C[i] == C[j])
      return false;    // Teste, ob jeder Knoten nur einmal besucht wird (O(n^2))
int sum = 0;
for ( int i = 0; i < n ; i++)
  sum += d(C[i],C[i+1]);  // Berechne die Länge des Kreises (O(n))
return sum <= k;
```

Laufzeit:

GUESS:

Es wird ein Vektor mit $n+1$ Elementen erzeugt, somit ist die Laufzeit in $\mathcal{O}(n)$.

CHECK:

Die ersten beiden Schleifen durchlaufen $\sum_{i=1}^{n} i \in \mathcal{O}(n^2)$ Elemente, die dritte Schleife durchläuft n Elemente und macht Brechungen, welche konstante Zeit benötigen. Somit liegt die Laufzeit in $\mathcal{O}(n^2)$.

Gesamt:

Die gesamt Laufzeit liegt somit in $\mathcal{O}(n^2) \Rightarrow$ TSP $\in$ NP
Alternativ könnte man auch eine Permutation $\Pi(i)$ nicht-deterministisch wählen, dann wäre nur noch die letzte Schleife notwendig und die Laufzeit wäre in $\mathcal{O}(n)$

```
int sum = 0;
for ( int i = 0; i < n ; i++)
  sum += d(C[Π(i)],C[Π(i+1)]);  // Berechne die Länge des Kreises (O(n))
return sum <= k;
```

q.e.d.

Reduktion: Hamilton-Cycle $\leq_p$ TSP

Wir haben gegeben, einen ungerichteten Graphen $G = (V, E)$.

Vorgehen:

Definiere:

$$d(v,w) = \begin{cases} 0 & \text{für } \{v,w\} \in E \\ 42 & \text{sonst} \end{cases}$$

Übergebe nun den Graphen $G = (V, V \times V)$ mit der Entfernungsfunktion $d(v,w)$ an den TSP Algorithmus. Prüfe, ob das Ergebnis gleich Null ist, wenn ja, gebe JA zurück, wenn das Ergebnis größer Null ist, gib NEIN zurück.

Laufzeit:

1. Bilde eine $V \times V$ Matrix für die Entfernungsfunktion $d(v,w)$ mit den oben beschriebenen Werten. Laufzeit in $\mathcal{O}(n^2)$
2. Nutze den TSP Algorithmus mit $G = (V, V \times V)$ und der oben erstellten Entfernungsmatrix.
3. Prüfe ob das Ergebnis des TSP Algorithmus. Wenn dieses gleich Null ist, gibt JA zurück, sonst NEIN. Laufzeit in $\mathcal{O}(1)$

Alternativ ließe sich auch das oben definierte Entscheidungsproblem verwenden, mit dem selben Graphen und $k = 0$, die Laufzeit würde sich dabei nicht verändern.
Damit liegt die Laufzeit in $\mathcal{O}(n^2)$ und ist somit polynomiell.

Äquivalenzbeweis:

" $\Rightarrow$ " Geg: Hamiliton-Cycle
Existiert mindestens ein Hamiliton-Cycle, so existiert in der oben beschriebenen Entfernungsfunktion ein Weg im TSP, welcher alle Knoten besucht und die Entfernung Null hat.

" $\Leftarrow$ " Geg: TSP
Ist die kleinst mögliche Entfernung im TSP gleich Null, so existieren im Hamilton-Cycle Problem Kanten, so dass jeder Knoten einmal besucht werden kann.

$$\Rightarrow \text{ TSP ist NP-hart}$$

q.e.d

$$\text{TSP} \in \text{NP} \wedge \text{TSP ist NP-hart} \Rightarrow \text{TSP ist NP-voll.}$$

q.e.d

Aufgabe 2

a) Nicht-polynomielle Algorithmen

Nicht-polynomielle Algorithmen, sind Algorithmen, welche nicht in polynomieller Laufzeit laufen. Dazu gehören z.B. Algorithmen mit exponentieller Laufzeit. Oft werden möglichst effiziente nicht-polynomielle Algorithmen für Probleme in NP genutzt, z.B. Backtracking bei TSP oder SAT.

b) Pseudopolynomielle Algorithmen

Ein Algorithmus heißt pseudopolynomiell, wenn seine Laufzeit sowohl von der Eingabelänge n, als auch von dem größten Wert(/der größten Zahl) der Eingabe W abhängig ist, also durch ein Polynom $p(n, W)$ nach oben beschränkt ist. Ein Beispiel ist das Subset-Sum-Problem, für welches ein Algorithmus existiert, welcher ein Laufzeit in $\mathcal{O}(nW)$ hat.

c) Fixed-Parameter Algorithmen

Ein Algorithmus heißt Fixed-Parameter Algorithmus, wenn sich seine Laufzeit durch $f(k)\cdot p(n)$ angeben lassen kann, wobei $p(n)$ ein Polynom mit der Variablen n und $f(k)$ eine berechenbare Funktion mit dem Parameter k meint. Der Parameter k bezieht sich dabei auf eine Eigenschaft des Problems, z.B. die Größe der gesuchten Lösung, die Anzahl der Kanten, die Tiefe/Breite eines Baumes, die Baumweite o.ä. So ist Vertex-Cover in der Lösungsgröße parametrisierbar.

d) Approximationsalgorithmen

Approximationsalgorithmen sind Algorithmen mit polynomieller Laufzeit. Die Lösung eines Approximationsalgorithmus ist im Allgemeinen nicht die optimale Lösung, darf sich aber von den Kosten her nur um einen konstanten Vorfaktor bzw. beim Profit nur um einen konstanten Nenner von der optimalen Lösung unterscheiden. Approximationsalgorithmen eigenen sich für Optimierungsprobleme.

e) Randomisierte-Algorithmen

Randomisierte-Algorithmen trifft während seiner Laufzeit gewisse Entscheidungen zufällig. Ziel ist es, den Algorithmus zu beschleunigen und/oder zu vereinfachen. Ausgenutzt wird, dass ein solcher Algorithmus im Durchschnitt eine gute Laufzeit haben soll. Ungünstige Eingaben werden verhindert, da der Algorithmus selbst bei gleicher Eingabe jedes mal unterschiedlich arbeitet. Beispiele finden sich oft in Sortieralgorithmen, wie dem randomisierten Quicksort.

f) Las-Vegas-Algorithmen

Las-Vegas-Algorithmen sind Randomisierte Algorithmen, welche immer ein richtiges Ergebnis liefern. Ihre Rechenzeit ist der Zufallsvariablen abhängig und kann abhängig von dieser kurz oder lang sein. Ziel ist es, dass der Erwartungswert der Rechenzeit möglichst gering ist. Der bereits erwähnte randomsierte Quicksort Algorithmus gehört hier dazu, da dieser abhängig vom gewählten Pivotelement schneller oder langsamer ist, aber immer am Ende das Array sortiert hat.

g) Monte-Carlo-Algorithmen

Im Gegensatz zu Las-Vegas-Algorithmen dürfen Monte-Carlo-Algorithmen einen Fehler verursachen. Die Wahrscheinlichkeit für diesen muss aber eine obere Schranke ausweisen. Durch mehrfaches Anwenden eines Monte-Carlos-Algorithmus auf das selbe Problem mit verschiedenen Zufallsvariablen senkt die Fehlerwahrscheinlichkeit und ist meist effizienter als ein deterministischer Algorithmus. Man versucht meistens die Fehlerwahrscheinlichkeit soweit wie möglich zu reduzieren, also z.B. auf die Wahrscheinlichkeit, dass der Computer einen Rechenfehler macht. Ein bekanntes Beispiel ist der Miller-Rabin-Test, welcher für eine Zahl entweder sicher ausgibt, das sie „nicht-prim“ ist oder „wahrscheinlich prim“. Wendet man den Miller-Rabin-Test auf einem Rechner häufig genug an (ich meine es wären um die 18 Mal), dann ist es wahrscheinlicher, dass der Computer im weiteren Verlauf der Berechnung einen Fehler macht, als dass die Zahl nicht prim ist, trotz der wiederholten Ausgabe „wahrscheinlich prim“.