Technische Universität München
Fakultät für Informatik
Dr. M. Schneider, Dr. M. Pizka,
Dr. A. Vilbig, Dr. R. Letz,
G. Bauer, B. Zhang

SS 2003
Lösungsvorschläge zu Blatt 8
13. Juni 2003

Übungen zur Vorlesung Einführung in die Informatik IV

m

Aufgabe 33 Erfüllbarkeitsproblem

Der Einfachheit halber nehmen wir an, dass die n aussagenlogischen Variablen V der Formel F durch natürliche Zahlen $1, \ldots, n-1$ dargestellt seien (wie z.B. im sog. DIMACS-Format realisiert).

(a) Pseudocode für Erfüllbarkeit

Zur Darstellung einer Belegung ι verwenden wir ein Feld der Grösse n. Das Verfahren besteht aus zwei Teilprogrammen: *Nichtdeterministisches Raten einer Belegung* und *Auswerten* der Belegung, die hintereinander ausgeführt werden.

Raten einer Belegung
forall $0 \leq i < n$
$\iota(i) := 1$ [] $\iota(i) := -1$
end

Auswerten geschieht durch das folgende funktionale Programm, wobei die Formel F als Eingabeparameter übergeben wird.

Auswerten(F)
if $F \in \mathbb{N}_0$ then
  return $\iota(F)$
elsif $F = \neg G$ then
  return $-$ Auswerten(G)
else $F = (G \vee H)$ then
  return max(Auswerten(G),Auswerten(H))
fi

(b) Nichtdeterministische Turingmaschine $T = \langle T, S, \delta, s_0, S_Z \rangle$ für Erfüllbarkeit

Der Einfachheit halber nehmen wir an, eine Variable i ($\in \mathbb{N}_0$) sei durch das Wort $|\cdots|$ der Länge $i+1$ dargestellt. Das zweite Band wird ausschliesslich zur Speicherung der Belegung verwendet.

Arbeitweise der Turingmaschine (Kopf am Anfang auf erstem Wortsymbol):

Raten und Speichern einer Belegungsfunktion $\{\langle 0, \iota(0) \rangle, \ldots \langle n-1, \iota(n-1) \rangle\}$ auf zweitem Band als Wort $\iota(0) \cdots \iota(n-1)$ wie folgt im (Anfangs-)Zustand s_0:

- wenn ein $|$ gelesen, gehe auf beiden Bändern 1 nach rechts (bleibe in Zustand s_0)
- andernfalls:
  - schreibe nichtdeterministisch **w** oder **f** auf zweites Band

- danach zunächst auf zweitem Band ganz nach links (Unterprogramm)
- dann, falls Bandende auf erstem Band erreicht, in Zustand s_1 und ganz nach links; ansonsten 1 nach rechts und wieder in Zustand s_0

Ersetzen aller $|\cdots|$ der Länge $i+1$ durch ihren Wert $\iota(i)$ im Zustand s_1

- wenn ein $|$ gelesen, gehe auf beiden Bändern 1 nach rechts
- andernfalls ersetze maximales Teilwort der Form $|\cdots|$ links vom Kopf auf erstem Band durch Symbol **w** bzw. **f** vom zweiten Band

Danach in Zustand s_2.

Auswerten der komplexen Formel mittels Simulation der folgenden reduktiven Grammatik auf dem ersten Band:
$\neg\mathbf{w} \to \mathbf{f}$
$\neg\mathbf{f} \to \mathbf{w}$
$(\mathbf{w} \vee \mathbf{w}) \to \mathbf{w}$
$(\mathbf{w} \vee \mathbf{f}) \to \mathbf{w}$
$(\mathbf{f} \vee \mathbf{w}) \to \mathbf{w}$
$(\mathbf{f} \vee \mathbf{f}) \to \mathbf{f}$

Akzeptieren, falls am Ende nur noch **w** auf erstem Band

Aufgabe 34 Rundreiseproblem

(a) Ein simpler und exakter, wenn auch sehr aufwendiger Algorithmus berechnet für alle möglichen Permutationen die Größe

$$gesamtdistanz = (\sum_{i=1}^{n} dist(\pi(i), \pi(i+1))$$

und bestimmt diejenige Permutation mit der geringsten Länge. Unter Verwendung eines Algorithmus zur Berechnung der Permutationen einer Menge von Elementen (siehe etwa Übungsaufgabe 42, Informatik I) ergibt sich dann folgender Pseudocode:

```
fct rundreise = ([1:n, 1:n] array int dist, [1:n] array string v) [1:n] array int:
   var [1:n!][1:n] array int perm := permutationen(v);
   var int minLaenge := gesamtdistanz(perm[1]);
   var int minimalpath := 1;
   for i:= 2 to n! do
            var int aktuelleLaenge := gesamtdistanz(perm[i]);
            if aktuelleLaenge < minLaenge then
                        minimalpath := i;
                        minLaenge := aktuelleLaenge;
            fi
   od
   perm[minimalpath];
```

Das Problem dieses Algorithmus ist offensichtlich: Es müssen alle Permutationen analysiert werden; bei 20 Städten sind das bereits $20! > 2.4 \cdot 10^{17}$. Das heißt, bereits bei einer überschaubaren Zahl von Städten ist eine exorbitante Anzahl von Pfaden zu überprüfen.

(b) Da die exakte Berechnung des minimalen Weges schnell an technische Grenzen stößt, wurden Algorithmen für Näherungslösungen erarbeitet. Eine derartige Methode ist, die Methode des **Nächsten Nachbarn:** Seien $V = \{V_1, ..., V_n\}$ die Knoten.

- Wähle eine beliebige Startecke V_{i_1}.
- Falls $V_{i_1}, ..., V_{i_m}$ konstruiert ist, suche eine Ecke $V_{j_0} \in V' = V \setminus \{V_{i_1}, ..., V_{i_m}\}$ mit

$$dist(V_{j_0}, V_{i_m}) = min\{V_j, V_{i_m} : V_j \in V'\}$$

  und setze $i_{m+1} = j_0$.

Aufgabe 35 Rundreiseproblem vs. Erfüllbarkeitsproblem

(1) Dass die Entscheidungsvariante des Rundreise-Problems in NP ist, sieht man leicht. Man rate einfach einen Weg der Länge $\leq k$ und verifiziere dann mit polynomiellem Aufwand, dass die Kosten $\leq k$ sind.

(2) Typischerweise ist der Beweis der NP-Härte immer der schwierigere Teil. Dazu sucht man am besten nach einem Problem, das bereits als NP-vollständig erkannt wurde und versucht eine Transformation zu finden, die jede Instanz dieses Problem auf eine Instanz des neuen Problems mit polynomiellem Zeitbedarf reduziert, wobei die beiden Probleme bzgl. ihrer Lösbarkeit äquivalent bleiben.

In vielen Fällen sind auch bereits vereinfachte Versionen eines Problems NP-vollständig, wie auch im vorliegenden Beispiel.

Wir betrachten das folgende Problem *Hamilton-Pfad*: Gegeben ein (gerichteter) Graph G, gibt es eine Rundreise durch G.

Wir zeigen, dass die Existenz einer Rundreise durch einen gerichteten Graphen NP-hart ist. Dazu bauen wir auf dem folgenden bereits als NP-vollständig erwiesenen Problem 3-SAT auf: 3-SAT ist die Menge aller aussagenlogischen Formeln der Form $\bigwedge_{i \in \mathbb{N}} \{l_i^1 \vee l_i^2 \vee l_i^3\}$, wobei die l_i^j aussagenlogische Variablen oder deren Negation sind (die Teilformeln $l_i^1 \vee l_i^2 \vee l_i^3$ nennt man auch *Klauseln* und die l_i^j *Literale*).

Wir entwickeln nun eine Transformation, die jede 3-Sat-Formel in ein Rundreiseproblem vom o.g. Typ übersetzt, und zwar mit polynomiellem Zeitbedarf.

Sei F eine Konjunktion von m Klauseln der Länge 3 und $x_1, \ldots, x_r$ die aussagenlogischen Variablen, die in F vorkommen. Wir transformieren F in einen speziellen gerichteten Graphen $G = \langle V, E \rangle$ mit $|V| = r + 6m$. Die r Knoten dienen zur Repräsentation der aussagenlogischen Variablen, weiterhin gibt es m Teilgraphen zu jeweils 6 Knoten, die die Klauseln darstellen.

Für jede Variable x_i gibt es zwei eingehende und zwei ausgehende Kanten, die der Belegung von x_i durch $\mathbf{w}$ bzw. durch $\mathbf{f}$ entsprechen.

Die Klauselkomponenten bestehen aus drei Paaren von Knoten, wobei jedes Paar einem in der Klausel enthaltenen Literal entspricht, aus drei eingehenden und drei ausgehenden externen Kanten sowie aus 9 internen Kanten, wie in Abbildung **??** dargestellt.

Die von einem Variablenknoten, der x_i entspricht, erste ausgehende Kante (die „positive" Kante) führt zur ersten Klausel, die x_i enthält, und dort zu einem Knoten l_j, der dem Literal x_i entspricht. Die Klauselkomponente wird vom Knoten l'_j aus verlassen und führt zur nächsten Klausel, die x_i enthält, und so weiter, bis alle Klauseln durchlaufen sind, die x_i enthalten. Die von der letzten solchen Klausel ausgehende Kante ist die erste (die „positive") zum Variablenknoten, der x_{i+1}

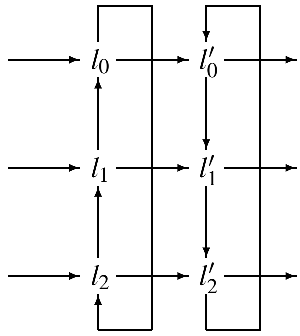

Abbildung 1: Klauselkomponente der 3-SAT-HAMILTON-Transformation

entspricht, führende. Die zweite (die „negative“) von einem Variablenknoten x_i ausgehende Kante führt analog zur ersten Klausel, die $\neg x_i$ enthält. Die entsprechende von der letzten Klausel, die $\neg x_i$ enthält, wegführende Kante ist die zweite (die „negative“) Kante, die zum Variablenknoten x_{i+1} führt. Auf dieselbe Weise wird schließlich der Variablenknoten x_n mit dem Variablenknoten x_1 verbunden.

Korrektheit der Transformation: Wir müssen zeigen, daß die Konjunktion der Klauseln in F erfüllbar ist genau dann, wenn es eine Rundreise durch den entsprechenden gerichteten Graphen gibt.

„$\Rightarrow$“: Sei ι ein Modell für die Konjunktion der Klauseln in F. Wir konstruieren zunächst einen Weg W, der alle Variablenknoten und Klauselkomponenten besucht, evtl. fehlende Knoten werden am Schluß durch eine lokale Reparatur in den Weg integriert. Dazu folgen wir von jedem Variablenknoten x_i aus der positiven Kante, wenn $\iota(x_i) = \mathbf{w}$, und ansonsten der negativen Kante. Die Klauselkomponenten passieren wir zunächst auf dem kürzesten Weg. Da in jeder Klausel mindestens ein Literal l vorkommt mit $\iota(l) = \mathbf{w}$, besucht W nach Definition des Graphen jeden Variabenknoten genau einmal, jede Klauselkomponente mindestens einmal und jeden internen Klauselknoten höchstens einmal. Mittels einer lokalen Modifikation des Weges durch die Klauselkomponenten lassen sich evtl. fehlende Klauselknoten wie folgt integrieren.

(a) Falls eine Klauselkomponente lediglich einmal durchlaufen wird via der Knoten l_i und l'_i, dann modifizieren wird den internen Weg $l_i \longrightarrow l'_i$ wie folgt:

$$l_i \longrightarrow l_{i\hat{+}2} \longrightarrow l_{i\hat{+}1} \longrightarrow l'_{i\hat{+}1} \longrightarrow l'_{i\hat{+}2} \longrightarrow l'_i\,,$$

wobei $\hat{+}$ die Addition modulo 3 ist.

(b) Falls die Klauselkomponente zweimal durchlaufen wird und die Knoten l_i und l'_i nicht besucht werden, dann modifizieren wir das Wegstück $l_{i\hat{+}1} \longrightarrow l'_{i\hat{+}1}$ wie folgt:

$$l_{i\hat{+}1} \longrightarrow l_i \longrightarrow l'_i \longrightarrow l'_{i\hat{+}1}\,.$$

Dadurch lassen sich alle fehlenden Knoten integrieren und wir haben eine Rundreise erhalten.

„$\Leftarrow$“: Sei W eine Rundreise durch den gerichteten Graphen, der F entspricht. Da jeder Variabenknoten x_i genau einmal passiert wird, können wir eine Interpretation ι wie folgt definieren: $\iota(x_i) = \mathbf{w}$, wenn x_i auf der positiven Kante verlassen wird und ansonsten $\iota(x_i) = \mathbf{f}$. Da jede Klauselkomponente durchlaufen wird, gibt es nach Definition des Graphen in jeder Klausel von F ein Literal l mit $\iota(l) = \mathbf{w}$. Damit ist ι ein Modell für alle Klauseln in F.

Weiterhin kann der F entsprechende Graph mit polynomiellem Zeitbedarf erstellt werden. q.e.d.

Es bleibt noch zu zeigen, dass wir vom gerichteten Rundreiseproblem zum Traveling-Salesman kommen können, was wir in zwei Schritten zeigen.

Zunächst zeigen wir, dass das Rundreise-Problem auch für ungerichtete Graphen (kurz: *Graphen*) NP-hart ist. Wir benutzen die in Abbildung **??** gezeigte Transformation θ von gerichteten Graphen in Graphen, bei der pro Knoten v_i zwei neue Knoten v_i^1 und v_i^2 sowie die entsprechenden Kanten hinzukommen.

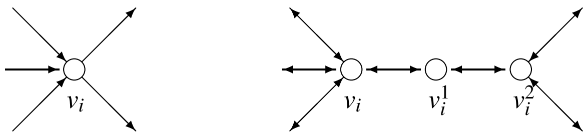

Abbildung 2: Transformation von gerichteten Graphen in Graphen

Aus einer Rundreise durch einen gerichteten Graphen G läßt sich unmittelbar eine Rundreise durch den entsprechenden Graphen $\theta(G)$ gewinnen. Falls es andererseits eine Rundreise durch $\theta(G)$ gibt, dann muß diese, um jeweils die Knoten v_i, v_i^1 und v_i^2 genau einmal zu passieren, entweder das Wegstück $v_i \longrightarrow v_i^1 \longrightarrow v_i^2$ oder $v_i^2 \longrightarrow v_i^1 \longrightarrow v_i$ enthalten. Da die Kantenrelation von $\theta(G)$ symmetrisch ist, gibt es eine Rundreise W, die $v_i \longrightarrow v_i^1 \longrightarrow v_i^2$ enthält. Nach Konstruktion läßt sich dann durch Projektion von W auf G eine Rundreise durch G erhalten. q.e.d.

Schliesslich übersetzen wir das letztere Rundreise-Problem in das Traveling-Salesman-Problem. Wir transformieren einen Graphen $G = \langle V, E\rangle$ wie folgt in den gewichteten Graphen $G' = \langle V, V \times V, w\rangle$, wobei $w : V \times V \to \mathbb{N}_0$ wie folgt definiert ist: $w(v_i, v_j) = 1$, falls $\langle v_i, v_j\rangle \in E$, und sonst: $w(v_i, v_j) = 2$. Die Kostengrenze k sei $|V|$. Offenbar gibt es genau dann eine Rundreise durch G, wenn es eine Rundreise durch G' gibt, sodaß die Summe der Kantengewichte $\leq |V|$ ist. q.e.d.

Übrigens folgt aus dem Beweis, dass das Traveling-Salesman-Problem bereits für Abstandsfunktionen $V^2 \to \{1, 2\}$ NP-vollständig ist.

Aufgabe 36 (P) Rundreiseproblem

```
public class TSP {
    private final static int INF = Integer.MAX_VALUE; // infinite distance
    private int numNodes = 3;                         // number of nodes in graph
    // matrix representation of distance from node_i to node_j
    // uses non-equidistant graph for somewhat more interesting results
    private int[][] distance = {{0, 3, 7},
                                {1, 0, 4},
                                {2, 6, 0}};

    // Calculate static example with 3 nodes and 6 paths
    // Returns shortest path as permutation with sequence of node indices
    public Permutation shortestPathStatic() {
        int minLength = INF;                              // initial minimal length
        int numPaths = 6;                                 // number of different paths
        Permutation spath = null;                         // shortest path
        Permutation[] paths = new Permutation[numPaths];// array of paths
        // Enter all possible permutations of node indices => 3! different paths
        for (int k = 0; k < numPaths; k++)
            paths[k] = new Permutation();
        paths[0].add(0); paths[0].add(1); paths[0].add(2);
        paths[1].add(0); paths[1].add(2); paths[1].add(1);
        paths[2].add(1); paths[2].add(0); paths[2].add(2);
        paths[3].add(1); paths[3].add(2); paths[3].add(0);
```

```
        paths[4].add(2); paths[4].add(0); paths[4].add(1);
        paths[5].add(2); paths[5].add(1); paths[5].add(0);
        for (int i = 0; i < numPaths; i++) {
            // Determine length of current path and compare with previous results
            int length = pathLength(paths[i]);
            System.out.println("Checking path " + paths[i] +
                               " with length " + length);
            if (length < minLength) {
                // Remember new minimal length and copy path
                minLength = length;
                spath = paths[i];
            }
        }
        return spath;
    }

    // Calculate shortest path for arbitrary graphs (not suitable for large graphs!)
    // Returns shortest path as integer array with sequence of node indices
    public Permutation shortestPath() {
        Permutation spath = null;
        int minLength = INF;
        // Use generator to determine all possible permutations of node indices
        PermutationGenerator gen = new PermutationGenerator();
        Set permutations = gen.generate(numNodes);
        Iterator it = permutations.iterator();
        while (it.hasNext()) {
            // Compare length of path to current minimal length
            Permutation path = (Permutation) it.next();
            int length = pathLength(path);
            System.out.println("Checking path " + path + " with length " + length);
            if (length < minLength) {
                minLength = length;
                spath = path;
            }
        }
        return spath;
    }

    // Calculate length for given path represented as permutation of node indices
    private int pathLength(Permutation path) {
        int result = 0;
        for (int i = 0; i < numNodes; i++) {
            // Account for wrap-around from last node to first node in path
            int from = path.get(i);
            int to = path.get((i+1) % numNodes);
            result += distance[from][to];
        }
        return result;
    }

    // Read graph from text file with number of nodes and distances in format
```

```
    // line 1:   n
    // line 2:   d_00 d_01 ... d_0n
    // line n+1: d_n0 d_n1 ... d_nn
    public void readGraph(String fileName) {
        String buffer = new String();
        int row = 0;
        numNodes = 0;
        try {
            BufferedReader reader = new BufferedReader(new FileReader(fileName));
            while (reader.ready()) {
                buffer = reader.readLine();
                if (numNodes == 0) {
                    numNodes = Integer.parseInt(buffer);
                    distance = new int[numNodes][numNodes];
                } else {
                    int column = 0;
                    StringTokenizer tokenizer = new StringTokenizer(buffer);
                    while (tokenizer.hasMoreTokens()) {
                        String tmp = tokenizer.nextToken();
                        distance[row][column] = Integer.parseInt(tmp);
                        column++;
                    }
                    row++;
                }
            }
            reader.close();
        }
        catch (IOException ex) {
            ex.printStackTrace();
            System.exit(1);
        }
    }

    public static void main(String[] args) {
        TSP tsp = new TSP();
        System.out.println("*** Static test ***");
        Permutation spath = tsp.shortestPathStatic();
        int length = tsp.pathLength(spath);
        System.out.println("-> Shortest path: " + spath + " with length " + length);
        System.out.println("*** Dynamic test ***");
        tsp.readGraph("graph.txt");
        spath = tsp.shortestPath();
        length = tsp.pathLength(spath);
        System.out.println("-> Shortest path: " + spath + " with length " + length);
    }
}
```