Prof. Dr. Dieter Rautenbach
Dipl. Math. Philipp Schäfer

WS 2011/2012

Operations Research Praktikum

1 Einleitung

In diesem Praktikum sollen Sie einige einfache Algorithmen insbesondere der diskreten Optimierung implementieren. Typischerweise ist der Weg vom Pseudocode eines Algorithmus bis zu seiner fertigen Implementierung länger als erwartet. Genau mit den Problemen, die auf diesem Weg zu lösen sind, möchte dieses Praktikum Sie konfrontieren.

Zum Implementieren können Sie zwischen folgenden Sprachen wählen:

- Python 2.6 oder 3.0 und
- C oder C++ (so dass es der Compiler gcc 4.4.3 kompilieren kann).

Nach Rücksprache können Sie auch andere Sprachen benutzen, solange Sie sicherstellen, dass wir Ihren Code ausführen können.

Um grundlegende Definitionen und relevante Sätze nachzuschlagen eignen sich zum Beispiel die Bücher von Reinhard Diestel [1], Korte und Vygen [2] und West [3].

2 Aufgabenstellung

Bearbeiten Sie zunächst die Aufgaben, die in den Abschnitten 3, 4 und 5 beschrieben werden. Bearbeiten Sie dann eine der drei Hauptaufgaben beschrieben in den Abschnitten 6, 7 und 8.

Schreiben Sie eine Dokumentation und bereiten Sie einen Vortrag vor, in dem Sie Ihr Programm vorführen und erläutern.

In der Dokumentation beschreiben Sie, welche Schwierigkeiten bzw. Probleme bei der Implementierung über die hier beschriebenen Aufgabenstellungen hinaus auftraten und wie Sie sie gelöst haben, wie die theoretische Laufzeit der gewählten Implementierungen ist, ob und wie man diese Laufzeiten verbessern könnte und wie sich ihre jeweiligen Routinen (der Hauptaufgaben) auf zufälligen Instanzen verhalten.

3 Adjazenzmatrix

Überlegen Sie sich eine geeignete Datenstruktur, um Adjazenzmatrizen eines Graphen zu repräsentieren. Überlegen Sie sich ausgehend davon eine Datenstruktur, die einen Graphen mit Kantengewichten $c : E(G) \to \mathbb{N}$ repräsentiert. Schreiben Sie eine Routine, die einen Graphen mit Kantengewichten (in der von Ihnen gewählten Datenstruktur) als Argument nimmt und diesen in menschenlesbarer Form auf die Standardausgabe ausgibt.

4 Generieren von Zufallsgraphen

Schreiben Sie eine Routine, welche zur Eingabe von

- $n, C \in \mathbb{N}$ und
- $p \in [0, 1]$

die Adjazenzmatrix eines Zufallsgraphen G aus $\mathcal{G}_{n,p}$ mit $V(G) = [n]$ sowie eine zufällige Funktion $c : E(G) \to [C]$ generiert, d.h. jede der $\binom{n}{2}$ möglichen Kanten von G existiert unabhängig mit Wahrscheinlichkeit p.

5 Baumtest

Schreiben Sie eine Routine, welche zur Eingabe der Adjazenzmatrix eines Graphen G testet, ob G ein Baum ist, d.h. ob G ein zusammenhängender Graph ohne Kreise ist.

Falls G kein Wald ist, soll ein Kreis ausgegeben werden. Falls G nicht zusammenhängend ist, soll die Eckenmenge einer Komponente ausgegeben werden. Testen Sie Ihre Routine auf zufälligen Instanzen von Graphen (vgl. Abschnitt 4).

6 Prüfercode

Zu jedem Baum T definiert folgender Algorithmus den sogenannten Prüfercode $f(T)$ des Baumes T.

Input: Ein Baum T mit $V(T) = [n]$.
Output: $f(T) = (a_1, a_2, \ldots, a_{n-2}) \in [n]^{n-2}$.

begin
  for $j = 1$ *to* $n - 2$. **do**
    Sei i die kleinste Endecke aus T;
    $a_i :=$ Nachbar von i in T;
    Lösche i aus T;
  end
  return $f(T) = (a_1, a_2, \ldots, a_{n-2})$;
end

Man kann zeigen, dass f zwischen der Menge aller Bäume T mit $V(T) = [n]$ und der Menge $[n]^{n-2}$ eine Bijektion definiert, d.h. insbesondere gibt es n^{n-2} verschiedene solche Bäume (Satz von Cayley).

6.1 Baum → Prüfercode

Schreiben Sie eine Routine, welche zur Eingabe der Adjazenzmatrix A eines Baumes T mit $V(T) = [n]$ den Prüfercode von T ausgibt.

6.2 Prüfercode → Baum

Schreiben Sie eine Routine, welche zur Eingabe von $n \in \mathbb{N}$ die Adjazenzmatrix eines zufälligen Baumes T mit $V(T) = [n]$ generiert. Erzeugen Sie dazu zufällig gleichverteilt ein Element $(a_1, a_2, \ldots, a_{n-2})$ von $[n]^{n-2}$ und bilden Sie $T = f^{-1}((a_1, a_2, \ldots, a_{n-2}))$. Die Routine soll die Adjazenzmatrix von T ausgeben.

7 Minimum Spanning Trees

Schreiben Sie Programmcode, welcher zur Eingabe

- der Adjazenzmatrix eines Graphen G
- und einer Funktion $c : E(G) \to [C]$

die Adjazenzmatrix eines Minimum Spanning Tree von G bestimmt.

Implementieren Sie dazu Kruskal's Algorithm sowie Prim's Algorithm und testen Sie Ihre Routinen auf zufälligen Instanzen.

Algorithmus 1. Kruskal's Algorithm

Input: *Ein zusammenhängender Graph G und eine Kostenfunktion $c : E \to \mathbb{R}$.*
Output: *Ein* Minimum Spanning Tree *T.*
begin
  Sortiere die Kanten von G so, dass $c(e_1) \leq c(e_2) \leq ... \leq c(e_m)$ gilt;
  $T \leftarrow (V(G), \emptyset)$;
  for $i = 1$ **to** m **do**
    if $(V(G), E(T) \cup \{e_i\})$ *hat keinen Kreis* **then**
      $T \leftarrow (V(G), E(T) \cup \{e_i\})$;
    end
  end
  return T;
end

Algorithmus 2. Prim's Algorithm

Input: *Ein zusammenhängender Graph G und eine Kostenfunktion $c : E(G) \to \mathbb{R}$.*
Output: *Ein* Minimum Spanning Tree *T.*
begin
  Wähle $v \in V(G)$;
  $T \leftarrow (\{v\}, \emptyset)$;
  while $V(T) \neq V(G)$ **do**
    Wähle eine Kante $e = xy$ minimaler Kosten mit $x \in V(T)$ und $y \in V(G) \setminus V(T)$;
    $T \leftarrow (V(T) \cup \{y\}, E(T) \cup \{e\})$;
  end
  return T;
end

8 Lange Kreise

Einen Weg $P : u_0u_1 \dots u_l$ in einem Graphen G nennen wir *inklusionsmaximal*, wenn keine Ecke v von G existiert, für die entweder $vu_0u_1 \dots u_l$ oder $u_0u_1 \dots u_lv$ ein Weg in G ist.

Betrachten Sie folgende Heuristik zur Konstruktion langer Kreise in Graphen.

Algorithmus 3. LONG CYCLE

Input: *Ein Graph G, der kein Wald ist.*
Output: *Ein Kreis C in G.*
begin
  Sei C der Kreis in G, den die Routine aus Abschnitt 5 findet;
  if $n(C) = n(G)$ **then**
    done $\leftarrow 1$;
  else
    done $\leftarrow 0$;
    $P \leftarrow \emptyset$;
  end
  while done $= 0$ **do**
    Sei $P' : u_0u_1 \dots u_l$ ein inklusionsmaximaler Weg in G mit $P \subseteq P'$;
    if $\exists i \in [l] : u_0u_i, u_{i-1}u_l \in E(G)$ **then**
      $C' : v_0 \dots v_lv_0 \leftarrow u_0u_1 \dots u_{i-1}u_lu_{l-1} \dots u_iu_0$;
      if $\exists v_i \in V(C') : \exists w \in V(G) \setminus V(C') : v_iw \in E(G)$ **then**
        $P \leftarrow wv_iv_{i+1} \dots v_lv_0v_1 \dots v_{i-1}$;
      else
        done $\leftarrow 1$;
      end
    else
      done $\leftarrow 1$;
    end
  end
  if $n(C') > n(C)$ **then**
    $C \leftarrow C'$;
  end
  return C;
end

Der Beweis des Satzes von Dirac impliziert, dass obige Heuristik in Graphen G mit $2\delta(G) \geq n(G) \geq 3$ immer einen Hamiltonschen Kreis findet.

Implementieren Sie diese Heuristik. Immer dann, wenn Sie mehrere Wahlmöglichkeiten haben, sollten Sie möglichst zufällig eine der Möglichkeiten wählen. Bei der Konstruktion von P' zum Beispiel könnten Sie den aktuellen Weg jeweils um einen zufälligen Nachbarn einer der Endecken, der außerhalb des Weges liegt, verlängern, bis solch ein Nachbar nicht mehr existiert. Auch bei der Wahl des Paares (v_i, w) sollten Sie unter allen Möglichkeiten zufällig wählen.

Testen Sie Ihre Implementierung auf zufälligen Instanzen aus $\mathcal{G}_{n,p}$. Ab welchem p findet die Heuristik typischerweise Hamiltonsche Kreise in G.

Literatur

[1] Reinhard Diestel, *Graph Theory*, verschiedene Auflagen (auch in Deutsch)

[2] Bernhard Korte und Jens Vygen, *Cominatorial Optimization*, verschiedene Auflagen (auch in Deutsch)

[3] Douglas B. West, *Introduction to Graph Theory*, Prentice Hall, verschiedene Auflagen