NitscheSahoo - AD Praktikum für 17.10.13 – Aufgabe 2 - Matrix

Das Interface:

```
public interface Matrix {

	void insert(int i, int j, double value);

	int getM();

	int getN();

	double get(int i, int j);

	void copyFrom(Matrix source);

	int memoryUsage();

	int accessCount();

	void resetAccessCount();

	Matrix add(Matrix m);

	Matrix mul(double skalar);

	Matrix mul(Matrix factor);

	Matrix pow(int exponent);

}
```

Begründungen:

Insert: man braucht eine Methode um ein Element an eine entsprechende Position zu packen.
GetM/getN: Simple getter für Matrixdimensionen.
CopyFrom: eine exakte Kopie von einer Matrix im Speicher, da wior nicht nur eine Referenz wollen.

Aufgabe 7 – Platzaufwand der Listenimplementationen

Siehe Datei „ab2_test/Aufgabe7.java“.

Versuch: (Aufgabenteil a, b und d – Vergleich der Matrizenimplementationen)
Erwartung: Wir erwarten, dass wir beim Erhöhen von p um eine Faktor auch einen erhöten Speicherplatzbedarf in beiden Listenimplementationen um den gleichen Faktor erhalten. Wir erwarten, dass ungefähr $p*n^2$ viele Elemente gespeichert sind. Die MatrixArray Implementation braucht n^2 viel Speicher.

Ergebnisse: Siehe Excel Tabelle Sheet „Aufgb2“ -> Aufgabe 7.

Interpretation: Unsere Erwartung hat sich bestätigt. Bei p=0.01 sind im Durchschnitt 10000 Elemente (9946.0) in der Matrix. Bei p=0.05 sehen wir ungefähr 50000 Elemente (50088.2). entsprechend auch ungefähr 100000 Elemente (99693.4) bei p=0.1.

Unsere zweite Vermutung über die durchschnittlich $p*n^2$ viele Elemente bestätigt sich auch.
Für n=1000 bekommen wir $n^2 = 1000000$. Und für z.B. p=0.01 erhalten wir $p*n^2 = 10000$. Dieser Wert ist ungefähr der gemessene Wert 9946.0.
Aus dem Verlgleich der Matrizenimplemntationen folgern wir Schluss, dass bei niedrigen p sich die Listenimplementationen mehr lohnen.

Da beide Matrizenimplementationen mit den gleichen Zufallsmatrizen arbeiten haben beide in den Ergebnissen den gleichen Platzbedarf.

Aufgabenteil c:
Zu einer gegebenen Speicherkapazität k und einer gegebenen Wahrscheinlichkeit p kann man aufgrund der Versuchsergebnisse von Versuch 1 die dazugehörigige Matrizendimension n x n bestimmen die sich in den beiden Listenimplementaionen dennoch speichern lässt. Zu einer Speicherkapazität von k (Anzahl von Elementen) kann man mit der MatrixArray Implementation n x n Matrizen für n kleiner gleich sqrt(k) speichern. Bei den Listenimplentationen ist die Grenze höher. Für ein festes p und einer Kapazität von k (Elementen) können n x n Matrizen der Größe p*sqrt(k) gespeichert werden.

Aufgabe 8 – Zeitaufwand der Listenimplementationen

Siehe Datei „ab2_test/Aufgabe8.java“.
Der Zeitaufwand wird in der Anzahl der Dereferenzierungen gemessen. Das Excel-Diagramm ist außerdem logarithmisch da dort einige Inhalte leichter zu erkennen sind.

Versuch 1:

Erwartung: Wir erwarten, dass die Multiplikation aufgrund $O(n^3)$ langsamer als die Addition $O(n^2)$ sein wird. Außerdem erwarten wir, dass die ArrayList implementation schneller als die reine Listenimpl. ist. Bei höheren p erwarten wir stets längere Laufzeiten.

Ergebnisse: Siehe Excel Tabelle Sheet „Aufgb2“ -> Aufgabe 8.

Interpretation: Wir stellen fest, dass die reine Listenimpl. immer langsamer ist als die ArrayList-Variante. Das liegt daran, dass bei der Listenvariante der Zugriff auf ein Element durch alle Elemente iterieren muss, wobei die ArrayList-Variante nur durch einen Teil aller Listen iterieren muss. Dadurch spart sie sich Zeit. Wir können dabei eine Verbesserung von ungefähr 100x fesstellen. (Das sind zwei Größenordnungen entlang der logarithmischen Skala.)
Die Multiplikation ist bei beiden Implementationen stets langsamer als die dazugehörige Addition.
Außerdem verlängert sich die Zeit beim Erhöhen von p, weil alle Operationen auf mehr Zahlen ungleich 0.0 ausgeführt werden müssen und daher sich die Zugriffsanzahl erhöht.

ListAddAverage und ArrayListMultAverage haben sehr ähnliche Werte(sowie ihre Overheads). Deswegen sind nicht beide Linien zu erkennen. Die Diagramme ist außerdem logarithmisch.

Beim Overhead, welcher als die Anzahl der Dereferenzierungen durch n^2 definiert ist (siehe zweites Diagramm) sehen wir einen Verlauf welcher den AccessCounts im anderen Diagramm entspricht. Die meiste Ersparnis hat man demnach bei der ArrayList implementation gegenüber der Listen- oder Arrayimplmenentation. Außerdem ist der Overhead bei der Addition um zwei Größenordnungen kleiner als bei der Multiplikation.

Aufwandsabschätzung:

Arbeitszeit: 15h gemeinsam
Zeitverteilung: 50% MatrixImplementations - 50% Tests und Experimente mit Diskussion

Vorschau der Javadoc: