Algorithm-Engineering Projekt
Graphenproblem

Paul Vincent Guigas
Mat.-Nr: 70018

Das folgende Projekt thematisiert das Graphenproblem. Im genaueren wird der Prim Algorithmus und dessen Effizienz in zwei verschiedenen Implementationen verglichen. Ziel ist es, mithilfe von konkreten Echtweltdaten den Algorithmus so zu optimieren, dass er sowohl für Benchmarkdaten als auch Echtweltdaten einen Minimalen Spannwald möglichst effizient erzeugt. Die Implementation des Problems erfolgt in Java.

Mathematisch Formale Beschreibung des Problems

Begriffserklärung:

- Ein Graph G = (V,E) ist ein Paar bestehend aus einer endlichen Menge von Knoten V und einer endlichen Menge von Kanten E.
- Ein ungerichteter Graph G = (V,E) ist definiert für jede $E \subseteq \{\{u,v\} \in V\}$.
  E kann dabei als symmetrische Relation $E \subseteq V \times V$ angesehen werden und liegt in der Form $e \in E = (u,v)$ vor.
- Ein ungerichteter Graph G heißt zusammenhängend, falls es zu je zwei beliebigen Knoten {u,v} ∈ V einen ungerichteten Weg in G mit u als Startknoten und v als Endknoten gibt
- Ein Graph G heißt gewichtet, wenn jeder Kante e ∈ E ein Gewicht w(e) zugeordnet ist.
- Ein Spannwald eines Graphen G ist eine Untermenge an Kanten E die alle Knoten V miteinander verbinden. Der Spannbaum ist dabei zusammenhängend
- Ein Minimaler Spannwald T ist ein Spannwald eines Graphen G, dessen Gewicht w(T) = $\forall e \in T \sum w(e)$ minimal unter allen möglichen Spannwäldern von G ist.

Der Prim-Algorithmus ist ein Greedy Algorithmus der einen Minimalen Spannwald in einem zusammenhängenden, ungerichteten, gewichteten Graphen G berechnet. Dabei geht der Algorithmus so vor, dass er von einem Startknoten ausgehend Schritt für Schritt Kanten zum Minimalen Spannwald hinzufügt bis dieser zusammenhängend ist. Der Greedy-Algorithmus wählt dabei immer die Kante mit dem geringsten Gewicht aus den vom MST erreichbaren Kanten aus. Diese wird dann zum MST hinzugefügt, sodass eine weiterer Knoten aus G mit dem MST erreicht werden kann.

Als Echtwelt Anwendungsszenario wurde der Datensatz “Distance Between Capital Cities” des Autors Kristian Skrede Gleditsch gewählt. (Quelle: http://privatewww.essex.ac.uk/~ksg/data-5.html)
Dieser enthält eine Reihe von Hauptstädten und die Entfernungen dieser zueinander. Für die Bearbeitung innerhalb des Projektes wurden die Indexe angepasst sowie der Datensatz entsprechend ergänzt, sodass ein zusammenhängender Graph aus diesem erstellt werden kann. Der angepasste Datensatz sowie die dazugehörige Legende ist im Anhang beigefügt.
Werden diese Daten nun innerhalb des Problems verarbeitet, so ist das Resultat ein Minimaler Spannwald aller Hauptstädte, sprich die kürzeste Route zwischen allen Haupstädten.

Implementation des Algorithmus

Um das Problem innerhalb eines Algorithmus zu bearbeiten wurde die Datenstruktur des WeightedEdgeArray (siehe http://www.cs.cmu.edu/~pbbs/test/benchmarks/graphIO.html) implementiert. Diese wird in folgender Datenstruktur abgebildet:

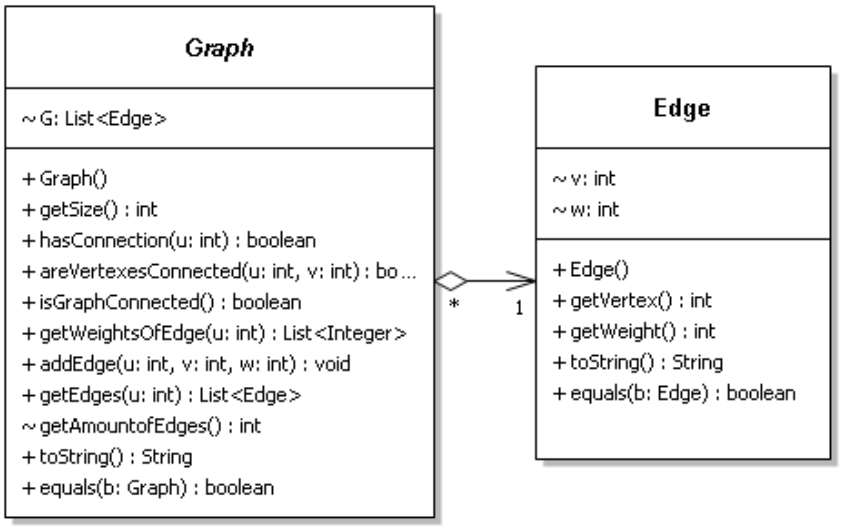

Abbildung 1: UML-Diagramm Datenformat

Hierbei werden die Knoten innerhalb der Klasse Graph beschrieben, die zu einem Knoten zugehörigen Kanten werden in der Klasse Edge abgebildet.

Die formale Beschreibung des implementierten Prim-Algorithmus lautet wiefolgt:

PRIM (Spannbaum input_ST vom Typ Graph)

1. **Rückgabewert**: MST vom Typ Graph
2. Erstelle neuen Baum MST, der Größe input_ST.getSize()
3. Startwert ← Erster Knoten (n=0)
4. Betrachte Nachbarn des ersten Knotens
5. Wähle Kante mit geringstem Gewicht
6. Füge die Kante zum MST hinzu
7. **while** MST nicht zusammenhängend
8. {
9. Formatiere die Warteschlange
10. Durchlaufe die bereits zum MST hinzugefügten Knoten
11. Untersuche input_ST:
12. Füge alle von den bereits hinzugefügten Knoten ausgehenden Kanten in die Warteschlange
13. Wähle aus der Warteschlange die Kante mit dem geringsten Gewicht
14. Füge die Kante zum MST

}

return MST

Abbildung 2: Erste Implementierung des Prim-Algorithmus

Zum Testen des Algorithmus wurde eine eigene Benchmark-Klasse implementiert

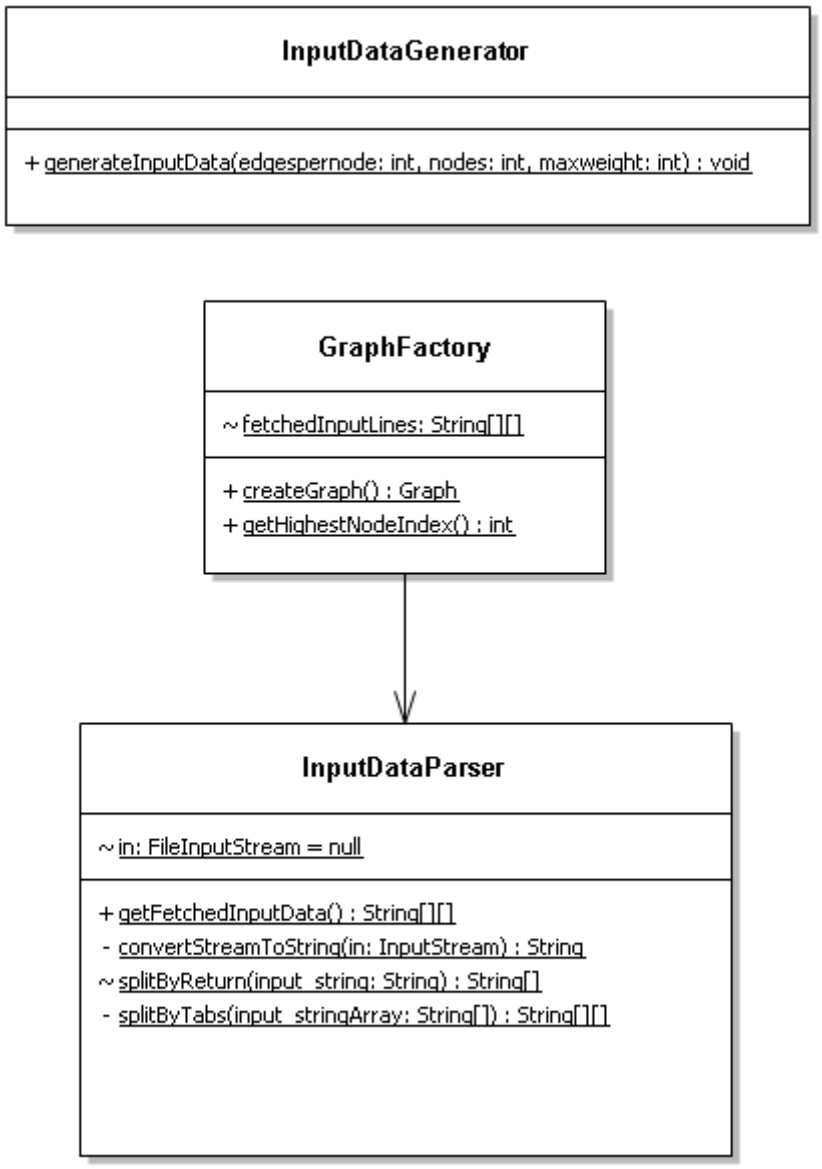

Abbildung 3: Klassen zum Auslesen der Daten aus einer input.txt sowie Generierung von Benchmarkdaten

Der InputDataGenerator kann mit der Methode generateInputData(…) eine Menge von Kanten im WeightedEdgeArray-Format in eine input.txt schreiben. Dabei wird angegeben, wie viele Knoten der Datensatz umfassen soll und wie viele Kanten pro Knoten existieren. Danach kann der Inhalt dieser input.txt mithilfe der GraphFactory.createGraph() Methode ausgelesen und in das Graph-Datenformat überführt werden.

Laufzeitmessung mit Benchmarkdaten

Beim Testen des Algorithmus wurden 2 Ansätze gewählt: So wurde ein bei Gleichbleibender Anzahl der Kanten im Graphen G einmal ein Ansatz mit geringer und mit hoher Knotenzahl gewählt. Die Anzahl der Probleminstanzen beträgt n= 15.

Ansatz A: (Geringe Knotenanzahl mit hoher Kantenanzahl pro Knoten)

Anzahl der Knoten	Gesamtanzahl der Kanten	Durchschnittliche Zeit	Standardabweichung
10	2222	0,5	0,612372436
20	8442	3,125	0,695970545
30	18662	12,875	0,78062475
40	32882	37,0625	1,67588596
50	51102	87,3125	4,208900539
60	73322	185,75	15,07274029
70	99542	366,6875	39,07479806
80	129762	664	124,338751
90	163982	961,375	190,9492325
100	202202	1419,4375	240,8552389

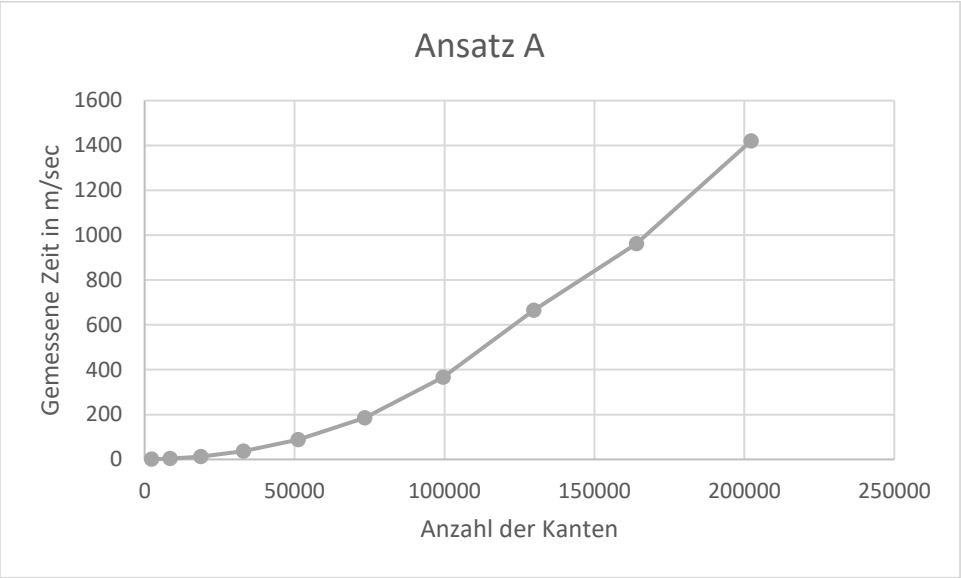

Abbildung 4: Alter Algorithmus - Geringe Knotenanzahl mit hoher Kantenanzahl pro Knoten

Ansatz B: (Hohe Knotenanzahl mit geringer Kantenanzahl pro Knoten)

Anzahl der Knoten	Gesamtanzahl der Kanten	Durchschnittliche Zeit	Standardabweichung
100	2222	19,75	6,72216483
200	8442	256,4375	53,37130403
300	18662	1244,9375	353,0739067
400	32882	3109,875	675,6737263
500	51102	7064,625	1007,491431
600	73322	12953,75	966,8002314
700	99542	22256,625	1026,728291
800	129762	37723,0625	1367,474793
900	163982	57493,0625	1361,688312
1000	202202	85971,9375	1433,758586

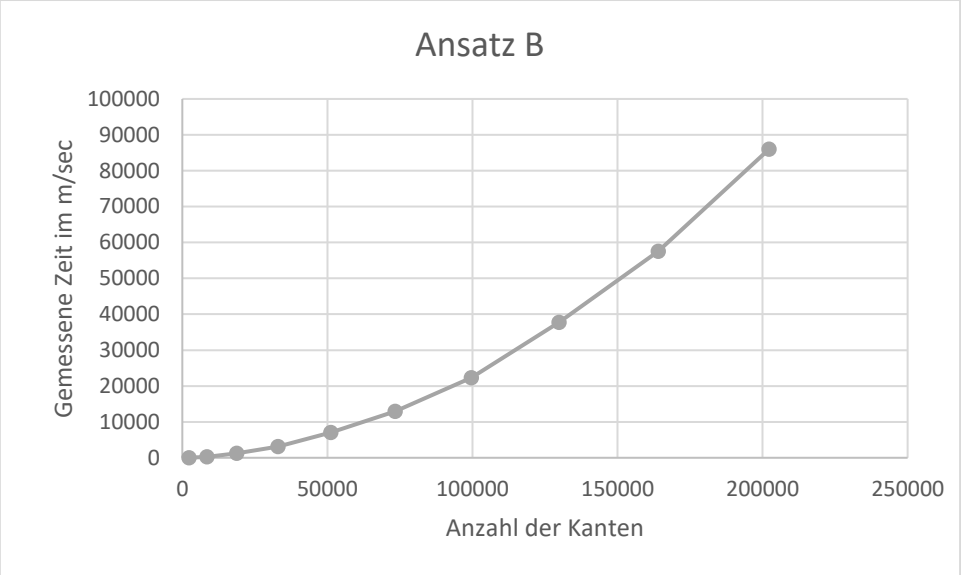

Abbildung 5: Alter Algorithmus - Hohe Knotenanzahl mit geringer Kantenanzahl pro Knoten

Somit sind aus dem Datensatz 2 Dinge zu schließen:

- Der die Kurven wachsen exponentiell, somit lässt sich als asymptotische Laufzeit O(n^2) annehmen.
- Ein entscheidender Faktor bei der Laufzeit ist die Anzahl der Knoten.

Beim Betrachten des Codesnippets ergibt sich folgendes Optimierungspotential:

```
// OldMstBuilder.DoThePrim()

// Äußere Schleife durchläuft die Bereits zum MST hinzugefügten Knoten
// Wenn sich viele Knoten im Spannwald befinden, so wird diese
// Schleife auch entsprechend oft durchlaufen und defacto verlängert
// dies die Laufzeit
(Integer node : usedNodes)
{
      // Referenz A: Schleife durchläuft Kanten des Knoten
      for(Edge e : input_graph.getEdges(node))
      {
            // Referenz B: Zielknoten schon besucht?
            if(!usedNodes.contains(e.getVertex()))
            {
                  int[] newEdge = {node,e.getVertex(),e.getWeight()};
                  possibleEdges.add(newEdge);
            }
      }
}
```

Abbildung 6: Codesnippet des alten Algorithmus

Anwendung der Echtweltdaten

Als zusätzlicher Schritt wurden die Echtweltdaten in den Algorithmus eingespeist. Der daraus resultierende Graph hat eine Knotenzahl von V=216 und eine Kantenzahl von E=34106. Die Analyse mittels eines Profilers erbrachte folgendes Ergebnis:

Name	Total Time	Total Time (CPU)
main	1.201 ms (100%)	1.064 ms (100%)
main.**main** (String[])	1.054 ms (87,8%)	1.054 ms (99,1%)
OldMstBuilder.**DoThePrim** (Graph, int)	848 ms (70,6%)	848 ms (79,7%)
java.util.LinkedList.**contains** (Object)	718 ms (59,8%)	718 ms (67,5%)
Self time	59,8 ms (5%)	59,8 ms (5,6%)
Graph.**getEdges** (int)	40,1 ms (3,3%)	40,1 ms (3,8%)
OldMstBuilder.**getLowestEdge** (java.util.List)	10,0 ms (0,8%)	10,0 ms (0,9%)
java.util.LinkedList.**clear** ()	10,0 ms (0,8%)	10,0 ms (0,9%)
java.lang.System.**currentTimeMillis[native]** ()	9,99 ms (0,8%)	9,99 ms (0,9%)
GraphFactory.**createGraph** ()	206 ms (17,2%)	206 ms (19,4%)
Self time	0,0 ms (0%)	0,0 ms (0%)
org.netbeans.lib.profiler.server.ProfilerServer.**activate** (String, int, int, int)	137 ms (11,4%)	0,0 ms (0%)
sun.launcher.LauncherHelper.**checkAndLoadMain** (boolean, int, String)	9,99 ms (0,8%)	9,99 ms (0,9%)
Reference Handler	641 ms (100%)	9,71 ms (100%)
Finalizer	0,0 ms (-%)	0,0 ms (-%)
DestroyJavaVM	0,0 ms (-%)	0,0 ms (-%)

Abbildung 7: Ergebnis des Profilers für die Echtweltdaten - Alter Algorithmus

Der Algorithmus rechnete genau 848 Millisekunden. Vergleicht man dies mit den Werten aus Tabelle Ansatz A und B sehen wir, dass die Zeit überdurchschnittlich lange für Vergleichbare Kantenanzahl (siehe Ansatz A) oder vergleichbare Knotenanzahl (siehe Ansatz B) braucht. Die Echtwelt Daten bilden hier also einen Datensatz ab, der sehr viele Kanten bei moderater Knotenanzahl enthält.
Wie wir Abbildung XXX entnehmen können wird im Algorithmus ein Großteil der Zeit mit dem Aufruf java.util.LinkedList.contains verbracht. Wie in Abbildung 6 – Referenz A zu erkennen ist wird für eine Hohe Kantenzahl pro Knoten die Schleife auch entsprechend oft besucht. Folglich wird die contains-Methode bei Referenz B sehr oft aufgerufen, was die Erhöhung der Laufzeit erklärt.

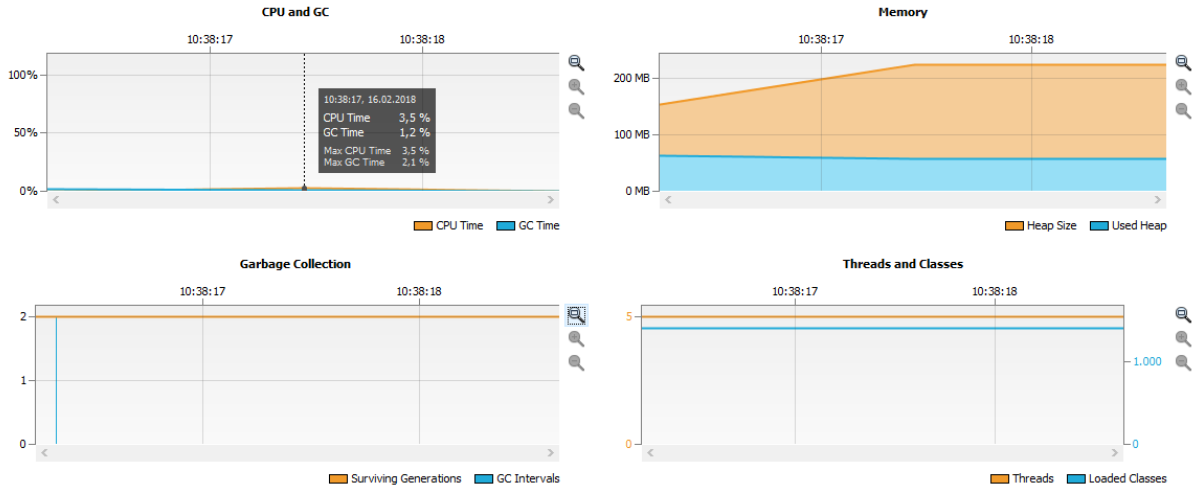

Abbildung 8: Speicherauslastung für die Echtweltdaten - Alter Algorithmus

Die schnell anwachsende Warteschlange lässt den benötigten Speicher schnell Anwachsen, somit muss eine Optimierung hin zu einer selektiveren Warteschlange vollzogen werden.

Verbesserung des Algorithmus

ImprovedPRIM (Spannbaum input_ST vom Typ Graph)

15. **Rückgabewert**: MST vom Typ Graph
16. Erstelle neuen Baum MST, der Größe input_ST.getSize()
17. Startwert ← Erster Knoten (n=0)
18. Betrachte Nachbarn des ersten Knotens
19. Wähle Kante mit geringstem Gewicht
20. Füge die Kante zum MST hinzu
21. **while** MST nicht zusammenhängend
22. {
23. Untersuche alle Kanten des zuletzt zum MST hinzugefügten Knotens (*nextNode*):
24. **IF(** *Kante und Zielknoten noch nicht in Warteschlange***)**
25. Füge Kante zur Warteschlange hinzu
26. **ELSE IF(** *Zielknoten der Kante schon in Warteschlange aber aktuell untersuchte Kante hat kleineres Gewicht***)**
27. Ersetze Kante in der Warteschlange
28. Wähle aus der Warteschlange die Kante mit dem geringsten Gewicht
29. Füge die Kante zum MST
30. Entferne die Kante aus der Warteschlange
31. Setze den entsprechenden Zielknoten als den zuletzt zu MST hinzugefügten Knoten (*nextNode*)
32. }
33. **return** MST

Abbildung 9: Zweite und verbesserte Implementierung des Prim-Algorithmus

Im verbesserten Algorithmus wird die Warteschlange nur dann erweitert, wenn der Zielknoten einer Kante sich nicht in der Warteschlange befindet. Sollte sich jedoch der Zielknoten schon in der Warteschlange befinden, so wird die entsprechende Kante im Falle einer neuen Kante geringeren Gewichtes ersetzt. Dadurch wird verhindert, dass die Warteschlange unnötig aufgebläht wird.

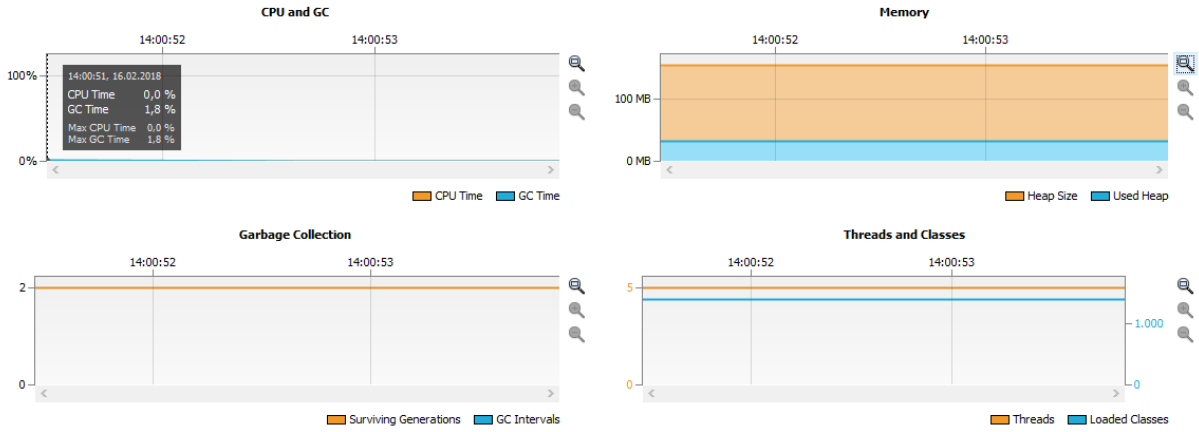

Abbildung 10: Speicherauslastung für die Echtweltdaten - Neuer Algorithmus

Dies belegt auch die Speicherauslastung. Diese hat sich für die Echtwelt Testdaten im Vergleich zum alten Algorithmus fast halbiert.

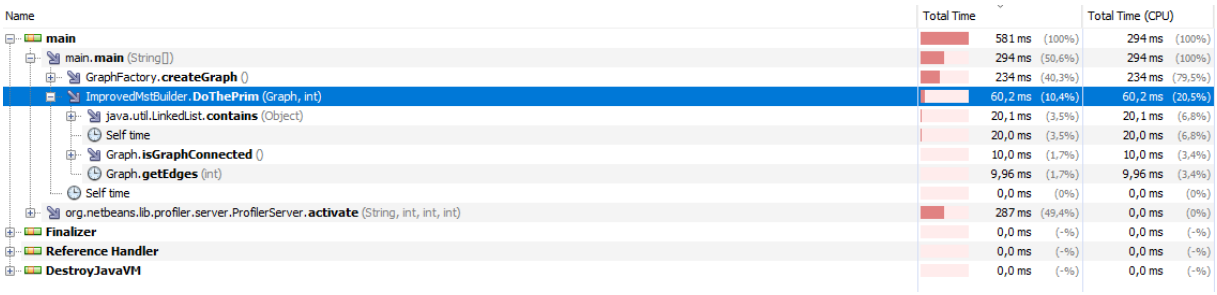

Abbildung 11:: Ergebnis des Profilers für die Echtweltdaten - Neuer Algorithmus

Folglich ändert sich auch die Laufzeit spürbar, so konnte diese auf 60ms reduziert werden.

Der neue Algorithmus wurde wie der alte auch mit 2 Ansätzen getestet. Die Anzahl der Probleminstanzen beträgt n= 15.

Ansatz A: (Geringe Knotenanzahl mit hoher Kantenanzahl pro Knoten)

Knotenanzahl	Gesamtanzahl der Kanten	Durchschnittliche Zeit	Standardabweichung
10	2222	0,375	0,484122918
20	8442	0,5	0,612372436
30	18662	1,3125	0,463512405
40	32882	2,3125	0,463512405
50	51102	4,4375	0,496078371
60	73322	8,3125	0,681794507
70	99542	14,1875	0,634305723
80	129762	23,25	3,913118961
90	163982	31,5	2,031009601
100	202202	43,0625	1,818954026

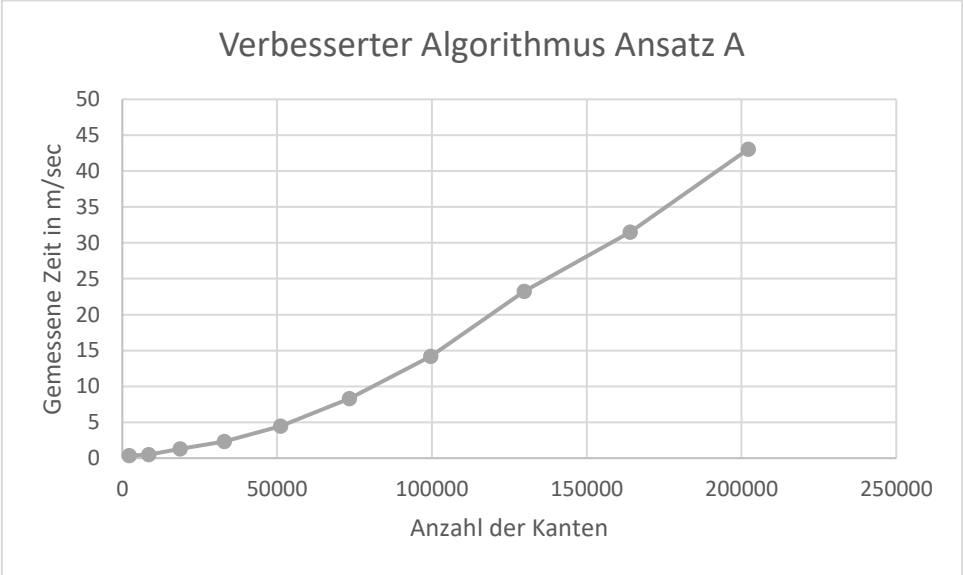

Abbildung 12: Geringe Knotenanzahl mit hoher Kantenanzahl pro Knoten - Neuer Algorithmus

Ansatz B: (Hohe Knotenanzahl mit geringer Kantenanzahl pro Knoten)

Knotenanzahl	Gesamtanzahl der Kanten	Durchschnittliche Zeit	Standardabweichung
100	2222	1,75	2,487468593
200	8442	4,5	2,979093822
300	18662	16,3125	0,463512405
400	32882	42,375	0,59947894
500	51102	87,8125	0,881670999
600	73322	163,3125	1,609299149
700	99542	272,5625	2,573148606
800	129762	445,5625	27,47036756
900	163982	660,9375	17,11895422
1000	202202	960,75	15,60248378

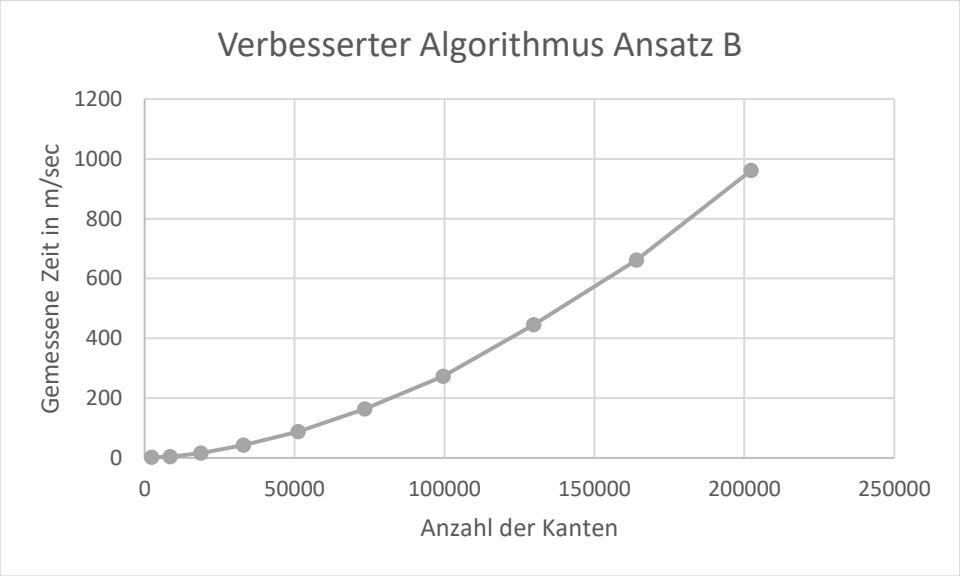

Abbildung 13: Hohe Knotenanzahl mit geringer Kantenanzahl pro Knoten - Neuer Algorithmus

Zwar ist die Laufzeit immer noch exponentiell, jedoch konnte durch die Verbesserung der Warteschlange die Laufzeit drastisch reduziert werden.

Vergleich der Algorithmen

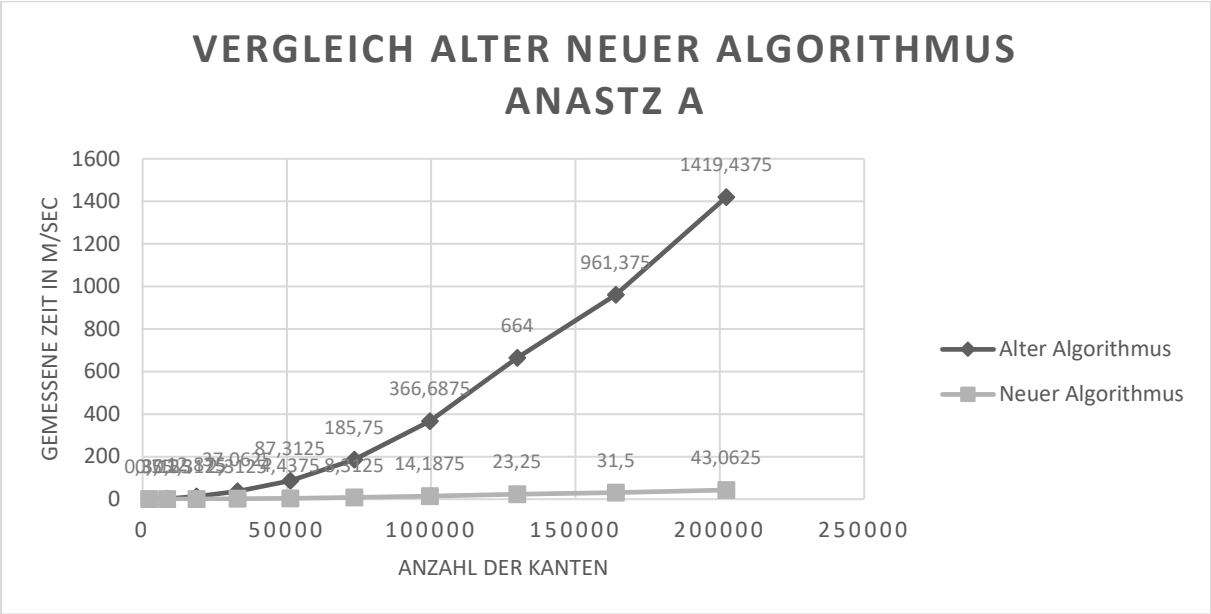

Abbildung 14: Geringe Knotenanzahl mit hoher Kantenanzahl pro Knoten - Vergleich Neu/Alt

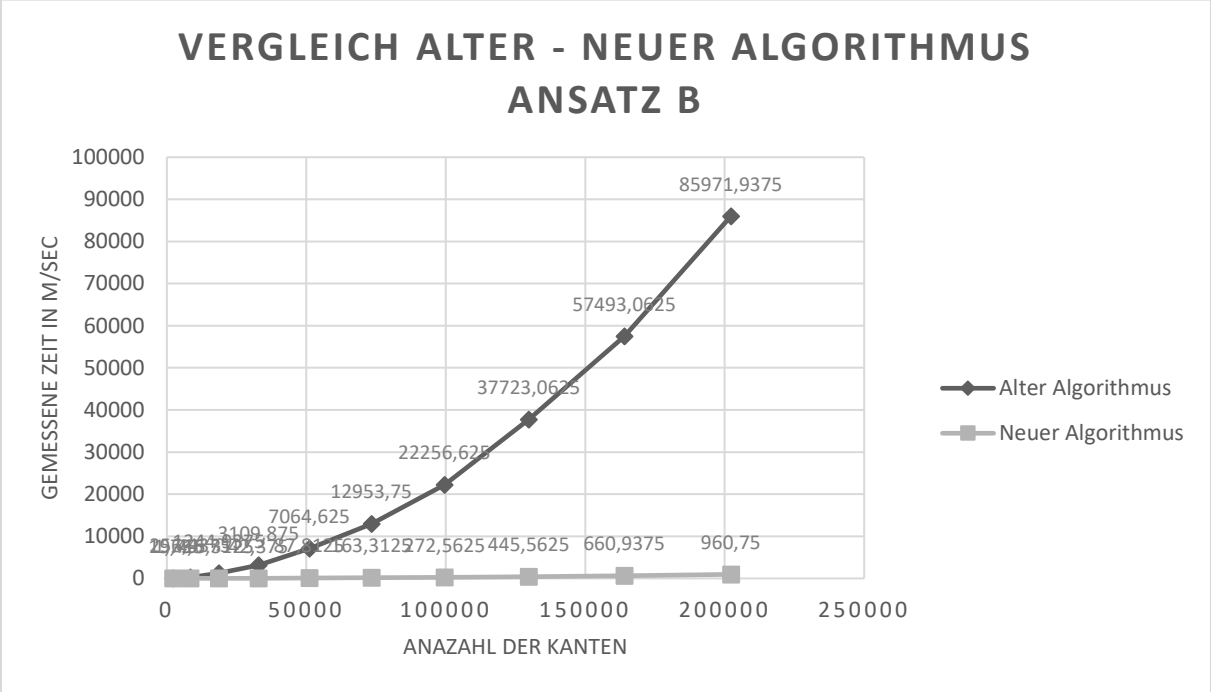

Abbildung 15: Hohe Knotenanzahl mit geringer Kantenanzahl pro Knoten - Vergleich Neu/Alt

Wie den Grafiken zu entnehmen ist, ist der neue Algorithmus erheblich schneller. Er sollte bei allen Problemen, sei es Benchmark- als auch Echtweltdaten, verwendet werden.

Quellen

Kristian Skrede Gleditsch: Distance Between Capital Cities	http://privatewww.essex.ac.uk/~ksg/data-5.html
Steven S. Skiena: The Algorithm Design Manual ; 2008; ISBN: 978-1-84800-069-8	Buch
Alexandra Schwartz: Einführung in die Graphentheorie ; Uni Würzburg	http://www.mathematik.uni-wuerzburg.de/~schwartz/Lehre/Graphentheorie/SkriptGraphentheorieSchwartz.pdf
Wikipedia.org: Zusammenhang (Graphentheorie)	https://de.wikipedia.org/wiki/Zusammenhang_(Graphentheorie)