Douglas Cunningham, Petra Hofstedt, Klaus Meer, Ingo Schmitt (Hrsg.): INFORMATIK 2015
Lecture Notes in Informatics (LNI), Gesellschaft für Informatik, Bonn 2015

Erzeugung minimaler Spannbäume auf ungerichteten, kantengewichteten Graphen mit den Algorithmen von Kruskal, Prim und Boruvka

Paul Jähne[1]

Abstract: Das Paper analysiert die Laufzeit der Algorithmen von Kruskal, Prim und Borůvka für das Erzeugen minimaler Spannbäume auf ungerichteten, kantengewichteten Graphen und vergleicht die Ergebnisse mit der theoretischen Laufzeit. Dazu werden die Algorithmen kurz vorgestellt und deren Implementierung erläutert. Die Algorithmen sind in der Programmiersprache C umgesetzt und teilweise mit dem MPI-Standard parallelisiert. Darauf folgt eine Beschreibung der für die Tests verwendeten Daten. Für Tests mit Graphen verschiedener Dichte werden generische Graphen fester Größe verwendet. Um Aussagen über Graphen verschiedener Größe zu treffen, kommen die Graphen des neunten DIMACS Implementierungswettbewerbs zum Einsatz. Anhand dieser Graphen wird gezeigt, dass der Algorithmus von Borůvka in der sequentiellen Variante sowohl für verschiedene Dichten als auch unterschiedliche Größen die schnellste Laufzeit und den geringsten Speicherverbrauch erzielt. Mit der gewählten Parallelisierungsstrategie erreicht der Algorithmus von Kruskal für dünne Graphen die beste Laufzeit.

Keywords: minimaler Spannbaum, Laufzeitanalyse, Algorithmus von Kruskal, Algorithmus von Prim, Algortihmus von Borůvka

1 Einleitung

Im Rahmen dieser Arbeit werden die Algorithmen von Kruskal, Prim und Borůvka für die Berechnung von minimalen Spannbäumen verglichen. Ein Spannbaum ist ein Baum aus Kanten eines Graphen, der alle Knoten enthält und kreisfrei ist. Minimal wird ein Spannbaum genannt, wenn die Summe aller Kantengewichte die kleinstmögliche unter allen Spannbäumen des Graphen darstellt.

Minimale Spannbäume werden unter anderem für Näherungslösungen des Rundreiseproblems oder für die Erstellung kostengünstiger Versorgungsnetzwerke eingesetzt. Sie sind auch zur Generierung von Labyrinthen verwendbar.

Es existieren bereits Veröffentlichungen, die die Algorithmen vergleichen. Jedoch sind diese relativ alt und stellen den Quellcode nicht zur Verfügung [MS91][DG98][BH01] oder benutzen einfache Implementierungen der Algorithmen [Po08]. Weiterhin fehlt bei diesen ein Bezug zu realen Daten, da mit zufällig generierten Graphen gearbeitet wird. Diese Lücke soll geschlossen werden. Zusätzlich wird mit den verwendeten Graphen ein größerer Problembereich abgedeckt.

[1] Hochschule für Technik, Wirtschaft und Kultur, Fakultät Informatik, Mathematik und Naturwissenschaftten, Schulstraße 12, 04687 Trebsen, paul.jaehne@gmx.de

2 Algorithmen

Alle verwendeten Verfahren gehören zur Klasse der Greedy-Algorithmen. Diese konstruieren die optimale Gesamtlösung aus Einzelentscheidungen, die zum jeweiligen Zeitpunkt den größten Gewinn versprechen. Bei der Angabe der asymptotischen Laufzeit steht E für die Menge der Kanten und V für die Knotenmenge des Graphen.

Der Algorithmus von Kruskal [Kr56] sucht auf dem gesamten Graphen nach der aktuell besten Lösung. Es handelt sich somit um ein globales Verfahren, das wie folgt abläuft. Zu Beginn erfolgt eine Sortierung der Kanten des Graphen nach ihrem Gewicht. Danach durchläuft der Algorithmus die sortierte Kantenliste. Dabei wird zu jeder Kante geprüft, ob ein Hinzufügen zum aktuellen Spannbaum einen Kreis entstehen lassen und somit die Spannbaumeigenschaft verletzen würde. Ist das nicht der Fall, wird die Kante zum Spannbaum hinzugefügt. Dies wird wiederholt bis alle Kanten abgearbeitet wurden. Um die Kreisfreiheit effizient zu prüfen, können die einzelnen Komponenten des Spannbaums in einer Union-Find-Struktur verwaltet werden. Unter Verwendung dieser wird die Laufzeit des Algorithmus durch das Sortieren der Kanten dominiert. Es ergibt sich somit für ein optimales vergleichsbasiertes Sortierverfahren die Zeitkomplexität $O(|E|\log|E|)$. Dies ist asymptotisch äquivalent zu $O(|E|\log|V|)$.

Im Gegensatz zum Algorithmus von Kruskal sucht der Algorithmus von Prim [Pr57] nur von den bisher gewählten Knoten aus eine neue Kante und arbeitet somit lokal. Am Anfang wird ein beliebiger Knoten gewählt. Danach wird die folgenden Schritte abgearbeitet. Von den verbundenen Knoten werden alle Kanten betrachtet, die zu einem bisher unerreichten Knoten führen. Von jenen wird die Kante mit dem geringsten Gewicht zum Spannbaum hinzugefügt. Dies wird fortgesetzt bis alle Knoten im Spannbaum enthalten sind. Für eine effiziente Implementierung wird eine Prioritätswarteschlange benötigt. Diese enthält zu jedem bisher unerreichten Knoten die Kante mit dem geringsten Gewicht, die zu einem erreichten Knoten führt. Weiterhin wird eine Adjazenzliste genutzt, in der die Nachbarn zu jedem Knoten enthalten sind. Für die Umsetzung der Warteschlange werden Heaps verwendet. Unter Nutzung eines binären Heap ergibt sich eine asymptotische Laufzeit von $O(|E|\log|V|)$ und mit einem Fibonacci-Heap $O(|E|+|V|\log|V|)$.

Der Algorithmus von Borůvka [Bo26] bearbeitet den Graphen folgendermaßen in einer Mischung aus lokalen und globalen Schritten. Beginne mit jedem Knoten als eine eigene Komponente. Finde von jeder Komponente die Kante mit dem geringsten Gewicht, die zu einer anderen Komponente führt. Füge diese Kanten hinzu und vereinige alle so verbundenen Komponenten. Wiederhole dies bis nur noch eine Komponente übrig ist. Somit wird auf dem gesamten Graphen für jede Komponente lokal nach dem aktuell besten Schritt gesucht. Für die Verwaltung der Komponenten kann eine Union-Find-Datenstruktur genutzt werden. Mit dieser ergibt sich eine theoretische Laufzeit von $O(|E|\log|V|)$.

3 Implementierung

Für die Implementierung der vorgestellten Algorithmen wird die Programmiersprache C verwendet. Weiterhin erfolgt eine teilweise Parallelisierung unter Verwendung des Messa-

ge Passing Interface (MPI)-Standard. Dabei handelt es sich um ein nachrichtengekoppeltes Programmiermodell. Dies bedeutet, dass jeder Prozess seinen eigenen Speicherbereich besitzt und zwischen diesen Informationen über Nachrichten ausgetauscht werden. Der Quellcode des Programms ist auf GitHub unter der LGPL in Version 3 verfügbar.[2]

Der Graph ist als C-struct mit einem Array für die Kantenliste und Werten für die Anzahl der Kanten und Knoten repräsentiert. Das Einlesen eines Graphen erfolgt über eine Datei in der zuerst die Anzahl der Knoten und Kanten steht. Danach folgen alle Kanten, die als drei Zahlen repräsentiert werden. Diese stehen für die verbundenen Knoten und das Kantengewicht. Die Knoten sind von Null beginnend durchnummeriert.

Die Implementierung des Algorithmus von Kruskal verwendet als Sortierverfahren Mergesort. Das Verfahren besitzt eine Zeitkomplexität von $O(n \log n)$ und ist ein stabiles Sortierverfahren. Weiterhin kann dieser Teil parallel ausgeführt werden, sodass jeder Prozess einen Teil der Kanten empfängt und sortiert. Anschließend werden die sortierten Teile wieder versendet und schrittweise vereinigt. Der Merge-Algorithmus ist leicht abgewandelt, indem die zweite Hälfte für das Merge in umgekehrter Reihenfolge kopiert wird. Das folgende Abarbeiten der Kantenliste erfolgt sequentiell unter Verwendung einer arraybasierten Union-Find-Struktur. Diese ist mit Pfadverkürzung und Union-by-Rank umgesetzt. Mit dieser wird geprüft, ob die Knoten der Kanten zu verschiedenen Komponenten gehören und somit ein kreisfreies Einfügen möglich ist.

Der Algorithmus von Prim wird mit Heaps umgesetzt. Es steht eine Variante mit einem binären Heap und eine mit einem Fibonacci-Heap zur Verfügung. Der binäre Heap ist als Array umgesetzt und der Fibonacci-Heap mit doppelt verketteten Listen. Außerdem wird eine Adjazenzliste benötigt, um auf die benachbarten Knoten zu jeder Kante zuzugreifen. Beide Varianten laufen gleichermaßen ab. Zuerst wird der Heap mit allen Knoten initialisiert. Begonnen wird beim ersten Knoten. Dieser wird entfernt und der Heap mit den Kanten aus der Adjazenzliste zu dem Knoten aktualisiert. Danach wird in jedem Schritt das Minimum entfernt und der Heap mit den neuen Kanten aktualisiert. Eine Parallelisierung erfolgt nicht, da die Heapstrukturen nicht dafür geeignet sind.

Der Algorithmus von Borůvka besteht aus zwei Teilen, die wiederholt nacheinander ausgeführt werden bis nur noch eine Komponente übrig ist.Der erste Teil ist die Suche aller Kanten mit dem geringstem Gewicht zwischen den Komponenten. Dazu wird die Union-Find-Struktur und ein Array von der Größe der Knotenanzahl genutzt. In dieses wird die Kante mit dem geringsten Kantengewicht eingetragen, die zu dem jeweiligen Knoten führt, sofern beide Knoten in verschiedenen Komponenten enthalten sind. Für die Ausführung mit mehreren Prozessoren wird die Suche parallelisiert. Dafür erhält jeder Prozess einen Teil der Kanten und sucht in diesem. Die Ergebnisse werden in einer Sammeloperation bei einem Prozess zusammengeführt. Dieser verteilt danach das Ergebnis wieder. Im zweiten Teil werden die neu verbundenen Komponenten vereinigt. Dazu durchläuft jeder Prozess das Array mit den kürzesten Knoten und vereinigt die verbundenen Komponenten. Anschließend aktualisiert jeder Prozess seine Komponentenliste.

[2] https://github.com/SethosII/minimum-spanning-tree

4 Daten

Um verschiedene Eigenschaften der Algorithmen zu untersuchen, werden zwei verschiedene Datenquellen verwendet. Mit den realen Daten werden die Eigenschaften für Graphen verschiedener Größe untersucht. Die generischen Daten dienen zur Untersuchung des Verhaltens auf Graphen unterschiedlicher Dichte.

Als Quelle für reale Daten werden die Graphen des 9. Implementierungswettbewerbs des Center for Discrete Mathematics and Theoretical Computer Science (DIMACS) verwendet.[3] Die Graphen beschreiben das Straßennetz der USA bzw. Teile davon. Eine Auflistung der einzelnen Graphen findet sich in Tabelle 1.

Name	Beschreibung	Knoten	Kanten	Dichte
USA	USA komplett	23.947.347	58.333.344	2,03E-7
CTR	Zentral-USA	14.081.816	34.292.496	3,45E-7
W	West-USA	6.262.104	15.248.146	7,77E-7
E	Ost-USA	3.598.623	8.778.114	1,35E-6
LKS	Große Seen	2.758.119	6.885.658	1,81E-6
CAL	Kalifornien und Nevada	1.890.815	4.657.742	2,60E-6
NE	Nordost-USA	1.524.453	3.897.636	3,35E-6
NW	Nordwest-USA	1.207.945	2.840.208	3,89E-6
FLA	Florida	1.070.376	2.712.798	4,73E-6
COL	Colorado	435.666	1.057.066	1,11E-5
BAY	Bucht von San Francisco	321.270	800.172	1,55E-5
NY	New York City	264.346	733.846	2,10E-5

Tab. 1: Auflistung der realen Graphen

Die Graphen besitzen sehr unterschiedliche Größen und decken einen Bereich von 70.000 bis 60.000.000 Kanten ab. Alle Graphen sind dünn, da ihre Dichte zwischen 10^{-5} und 10^{-8} liegt. Die Dichte errechnet sich mit Formel 1.

$$Dichte(G) = \frac{2|E|}{|V|(|V|-1)} \approx \frac{2|E|}{|V|^2} \tag{1}$$

Für Tests während der Entwicklung und zum Vergleich der Algorithmen auf verschieden dichten Graphen werden generischen Graphen verwendet. Diese besitzen ein zufälliges Kantengewicht von 0 bis 99 und sind in Tabelle 2 aufgelistet. Alle haben 10.000 Knoten und unterscheiden sich in der Anzahl der Kanten. Der Gittergraph hat eine zweidimensionale, rasterartige Struktur. Außer am Rand hat jeder Knoten vier Nachbarn. Bei dem Pfadgraph handelt es sich um eine Kette von Knoten, die bis auf den Start- und Endknoten jeweils nur zwei Nachbarn haben. Somit hat dieser die kleinstmögliche Dichte für einen zusammenhängenden Graph.

Der Spannbaum des Gittergraphen kann als Labyrinth interpretiert werden. Das Programm verfügt über die Möglichkeit beliebige rechteckige Labyrinthe zu erzeugen.

[3] http://www.dis.uniroma1.it/challenge9/download.shtml

Name	Beschreibung	Knoten	Kanten	Dichte
FULL	Graph mit Dichte 1	10.000	49.995.000	1,00E0
HALF	Graph mit Dichte 0,5	10.000	24.997.500	5,00E-1
QUAR	Graph mit Dichte 0,25	10.000	12.498.750	2,50E-1
GRID	Gittergraph	10.000	19.800	3,96E-4
PATH	Pfadgraph	10.000	9.999	2,00E-4

Tab. 2: Auflistung der generischen Graphen

5 Laufzeitmessung

Die Laufzeitmessungen erfolgen auf einem Dell R720 Server mit zwei Intel Xeon E5-2680 v2 Prozessoren und 512 GB Arbeitsspeicher. Als Betriebssystem kommt das Community Enterprise Operating System (CentOS) in der Version 6.6 zum Einsatz. Dieses verwendet den Linux-Kernel der Version 2.6.32. Als Compiler dient der GNU C Compiler 4.8.1 und die MPI-Implementierung OpenMPI 1.8.2 für die Parallelisierung.

Es werden für jeden Algorithmus auf allen vorgestellten Daten jeweils 30 Messungen durchgeführt und über diese gemittelt. Für die parallelisierten Varianten werden die gleichen Messungen zusätzlich auf zwei, vier und acht Prozessorkernen ausgeführt. Die Zeitmessung erfolgt über die MPI-Funktion MPI_Wtime.

Abbildung 1 zeigt die sequentielle Laufzeit der Algorithmen auf Graphen unterschiedlicher Größe. Die beste Laufzeit für alle Größen liefert der Algorithmus von Borůvka. Die benötigte Zeit liegt in etwa bei der Hälfte im Vergleich zum Algorithmus von Kruskal. Die Algorithmen von Kruskal und Prim mit binärem Heap sind für kleinere Graphen gleich auf. Mit steigender Größe erreicht der Algorithmus von Kruskal bessere Laufzeiten. Der Algorithmus von Prim unter Verwendung eines Fibonacci-Heaps ist der langsamste. Es wird nur für größere Graphen eine bessere Laufzeit als mit einem binären Heap erzielt.

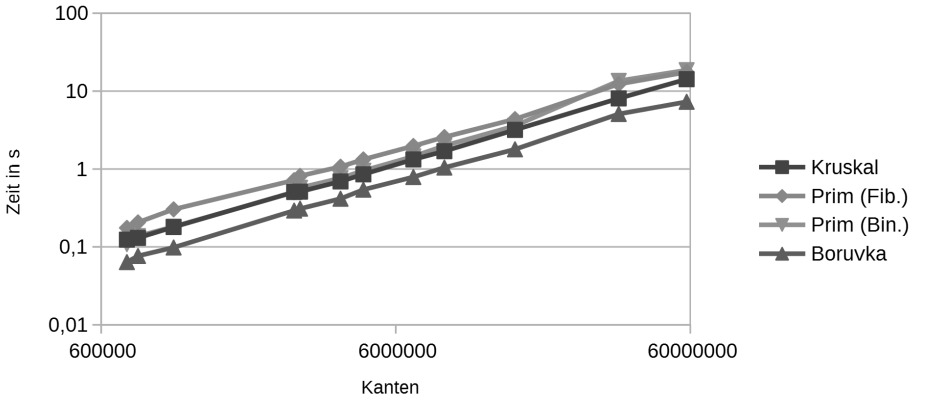

Abb. 1: Laufzeit für reale Daten (sequentiell)

Für den Vergleich der Algorithmen auf unterschiedlich dichten Graphen werden generische Graphen mit der gleichen Knotenanzahl verwendet. Die Ergebnisse sind in Abbildung 2 dargestellt. Bei diesen schneidet der Algorithmus von Kruskal mit zunehmender Dichte schlechter ab. Somit ist dieser Algorithmus für dichte Graphen wenig geeignet. Die Zunahme der Laufzeit für die Algorithmen von Kruskal und Prim ist nahezu linear. Jedoch ist der Anstieg des Algorithmus von Prim geringer. Die Variante mit einem Fibonacci-Heap erreicht keine bessere Laufzeit als die mit einem binären Heap. Der Algorithmus von Borůvka ist deutlich der schnellste. Dieser hat zusätzlich ab einer mittleren Dichte nur eine geringe Zunahme der Laufzeit je dichter der Graph wird. Damit ist der Algorithmus von Borůvka für dichte Graphen am besten geeignet.

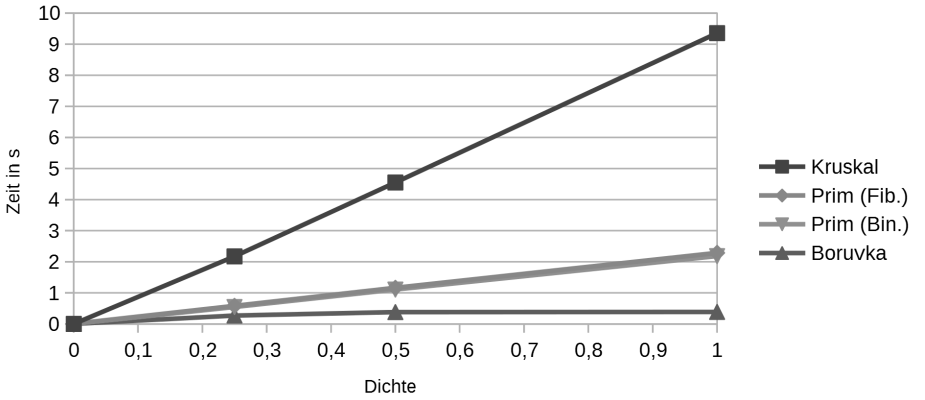

Abb. 2: Laufzeit in Abhängigkeit der Dichte (sequentiell)

Die Ergebnisse stehen im Gegensatz zu den Beobachtungen der eingangs angeführten Veröffentlichungen. So ergab die Veröffentlichung von Bazlamaçci und Hindi [BH01], dass der Algorithmus von Prim sowohl für dichte als auch dünne Graphen am besten geeignet ist und der Algorithmus von Borůvka mehr Zeit benötigt als der Algorithmus von Kruskal.

Im weiteren erfolgt eine Analyse der parallelen Varianten des Algorithmus von Kruskal und Borůvka für die realen und generischen Daten. Dazu wird der Speedup betrachtet. Dieser ergibt sich aus dem Verhältnis der Laufzeiten des sequentiellen und parallelen Algorithmus, wie in Formel 2 beschrieben, wobei P für die Anzahl der Prozessoren steht.

$$Speedup(P) = \frac{T(1)}{T(P)} \tag{2}$$

Die Abbildung 3 zeigt den erzielten Speedup auf Graphen verschiedener Größen unter Verwendung des parallelisierten Algorithmus von Kruskal. Dieser erreicht für alle verwendeten Größen der realen Daten einen akzeptablen Speedup, der unter Verwendung von mehr Prozessoren gesteigert werden kann. Für acht Prozessoren liegt dieser bei ungefähr 2,5. Mit vier und acht Prozessoren liegt die Laufzeit unter der sequentiellen Umsetzung des Algorithmus von Borůvka.

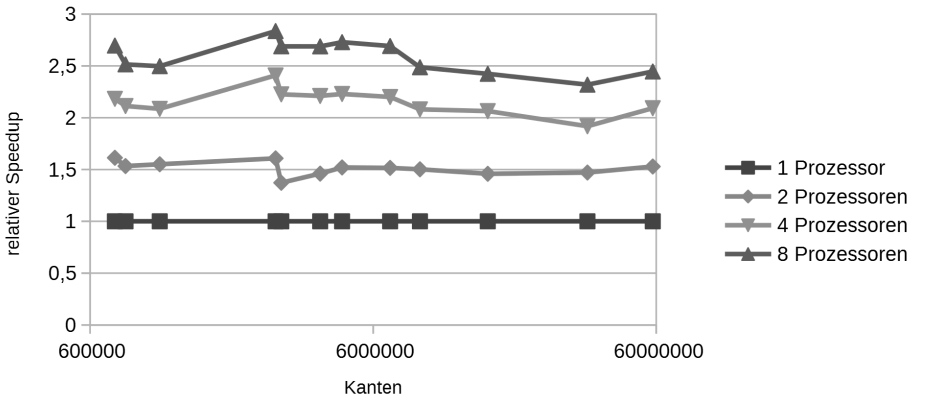

Abb. 3: Relativer Speedup für reale Daten (Kruskal)

Abbildung 4 zeigt den Speedup für den Algorithmus von Kruskal auf verschieden dichten Graphen. Für sehr dichte Graphen wird ein höherer Speedup erreicht als bei dünnen Graphen. Dies liegt daran, dass mit höherer Kantenanzahl der Sortieraufwand schneller steigt als der Kommunikationsaufwand. Unter Verwendung von acht Prozessoren kann ein Speedup nahe 4 erreicht werden. Damit liegt die Laufzeit für sehr dichte Graphen unter Verwendung von acht Prozessoren in etwa auf der Höhe des sequentiellen Algorithmus von Prim für diese Daten.

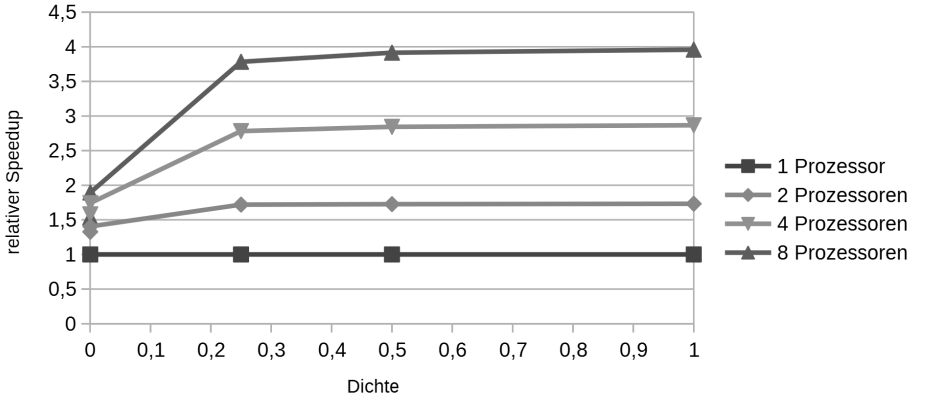

Abb. 4: Speedup in Abhängigkeit der Dichte (Kruskal)

In Abbildung 5 ist der Speedup für Graphen unterschiedlicher Größen unter Verwendung des Algorithmus von Borůvka dargestellt. Für reale Daten erreicht die verwendete Parallelisierung des Algorithmus von Borůvka keinen brauchbaren Speedup. Für zwei Prozessoren ist ein minimaler Zeitgewinn zu verzeichnen. Durch die Hinzunahme weiterer Prozessoren verlangsamt sich die Ausführung jedoch. Damit ist die gewählte Parallelisierungsstrategie für diese Daten nicht sinnvoll.

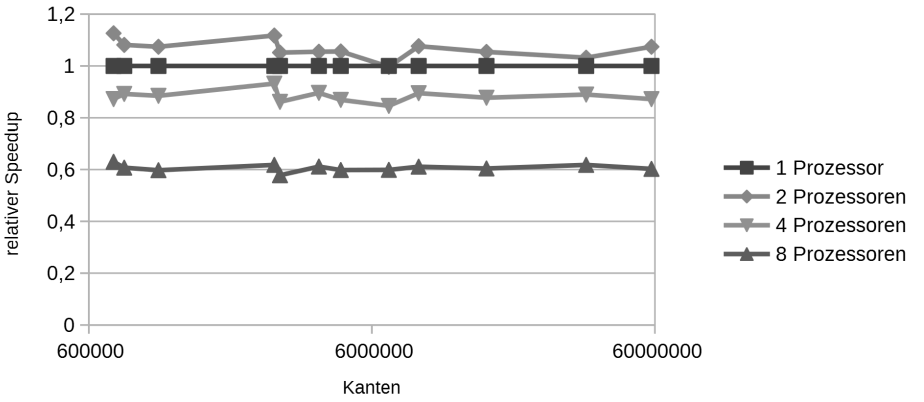

Abb. 5: Speedup für reale Daten (Borůvka)

Abbildung 6 zeigt den Speedup für den Algorithmus von Borůvka auf verschieden dichten Graphen. Für sehr dünne Graphen bestätigt sich das Bild aus den realen Daten, dass die Parallelisierung die Laufzeit verschlechtert. Für höhere Dichten ergibt sich jedoch ein akzeptabler Speedup, der mit steigender Dichte wieder sinkt. Somit kann die verwendete Parallelisierung für Graphen mittlerer Dichte als sinnvoll bezeichnet werden.

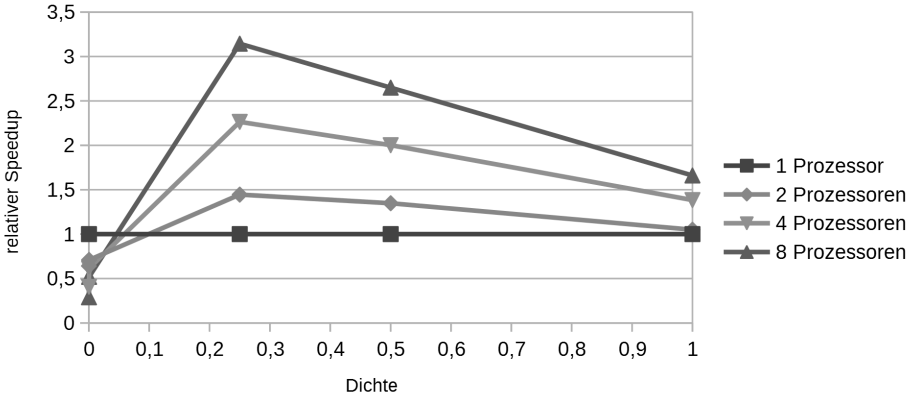

Abb. 6: Speedup in Abhängigkeit der Dichte (Borůvka)

6 Vergleich mit asymptotischer Laufzeit

Um zu überprüfen wie die reale Laufzeit zur asymptotischen in Beziehung steht, wird die Laufzeit der Algorithmen mit der jeweiligen asymptotischen Laufzeit aus Abschnitt 2 skaliert. Das Ergebnis ist in Abbildung 7 zu sehen.

Für alle Algorithmen ergibt sich eine waagerechte Linie. Dies bedeutet, dass die reale Laufzeit in dem Maße wächst, wie durch die asymptotische vorhergesagt. Die Implementierungen des Algorithmus von Prim enthalten Sprünge. Das zeigt an, dass eine leichte Abweichung von der asymptotischen Laufzeit vorliegt.

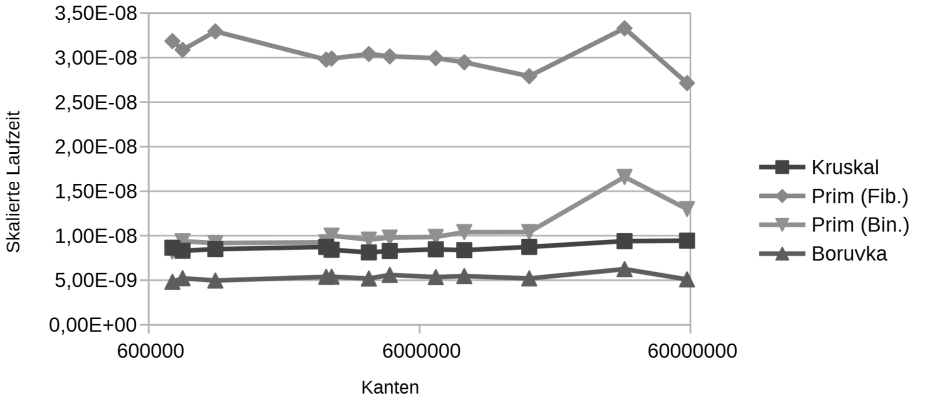

Abb. 7: Vergleich mit asymptotischer Laufzeit

7 Statistische Analyse

Um statistisch belegbare Aussagen für den Vergleich der Algorithmen treffen zu können, wird der gepaarte, zweiseitige t-Test verwendet. Die Resultate mit diesem Test stehen jeweils in Klammern hinter den Aussagen. Irrtumswahrscheinlichkeiten von unter einem Prozent (10^{-2}) sind als höchst signifikant anzusehen.

Anhand des USA-Graphen lassen sich folgende Aussagen treffen. Für die sequentiellen Varianten auf den realen Daten ist der Algorithmus von Borůvka schneller als der Algorithmus von Kruskal (10^{-32}). Dieser wiederum ist schneller als der Algorithmus von Prim in beiden Varianten (Prim Fib.: 10^{-26}, Prim Bin.: 10^{-23}). Der Algorithmus von Kruskal auf realen Daten ist in der parallelen Variante ab mehr als vier Prozessoren schneller als der Algorithmus von Borůvka (10^{-23}). Die Umsetzung des Algorithmus von Prim mit einem Fibonacci-Heap ist für große Graphen schneller als die mit einem binären Heap (10^{-17}). Die parallele Umsetzung des Algorithmus von Kruskal mit vier Prozessoren ist schneller als der sequentielle Algorithmus (10^{-23}).

Am Beispiel des FLA-Graphen kann gezeigt werden, dass die Laufzeiten der Algorithmen von Kruskal und Prim mit binärem Heap für kleine Graphen nicht verschieden sind (0,87). Außerdem kann gezeigt werden, dass der Algorithmus von Prim mit dem Fibonacci-Heap für kleine Graphen langsamer ist als der mit einem binären Heap (10^{-9}). Aus dem FULL-Graphen kann geschlossen werden, dass der Algorithmus von Borůvka für dichte Graphen schneller ist als der Algorithmus von Prim in beiden Varianten (Prim Fib.: 10^{-53}, Prim Bin.: 10^{-54}). Diese sind für dichte Graphen wiederum schneller als der Algorithmus von Kruskal (10^{-48}).

8 Speicherverbrauch

Für die Messung des Speicherverbrauchs wird der USA-Graph verwendet. Dies ist der größte verfügbare reale Graph. Die Messung erfolgt mit einem periodischen Aufruf des Kommandozeilentool ps aller zehn Millisekunden. Die Messwerte beschreiben den maximalen Speicherverbrauch über die gesamte Laufzeit der sequentiellen Varianten. Die Ergebnisse sind in Abbildung 8 zu sehen.

Die Repräsentation des Graphen benötigt im Arbeitsspeicher rund 0,7 GB, was als Referenzwert für die relative Angabe verwendet wird. Es ist zu erkennen, dass der Algorithmus von Borůvka den geringsten Speicherverbrauch hat (1,4 GB). Den meisten Speicher benötigt der Algorithmus von Prim unter Verwendung eines Fibonacci-Heaps (4,6 GB).

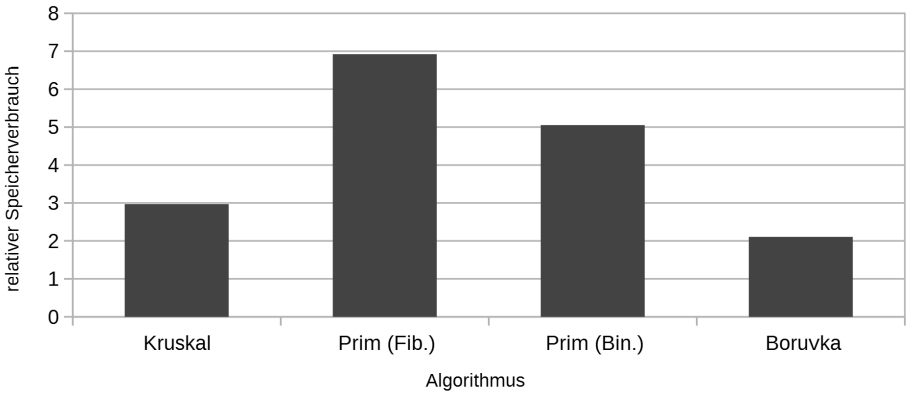

Abb. 8: Vergleich des relativen Speicherverbrauchs mit dem USA-Graphen

9 Zusammenfassung

Das Paper beschreibt und vergleicht die klassischen Algorithmen zur Berechnung eines minimalen Spannbaums. Es wird gezeigt, dass in der sequentiellen Variante der Algorithmus von Borůvka die beste Wahl ist. Dieser hat den geringsten Speicherverbrauch und die geringste Laufzeit für Graphen beliebiger Dichte und Größe. Zusätzlich ist er vergleichsweise einfach zu implementieren.

Die genutzte Parallelisierung des Algorithmus von Borůvka ist nicht geeignet für die Verwendung auf realen Daten und erreichte nur auf Graphen mit mittlerer Dichte einen akzeptablen Speedup. Mit einem speichergekoppelten Programmiermodell könnte die Parallelisierung sinnvoller umgesetzt werden, da der Austausch des Arrays nicht nötig wäre und stattdessen mit atomaren Operationen gearbeitet werden könnte.

Der Algorithmus von Kruskal ist für die realen Daten die zweitbeste Wahl. Durch die eingesetzte Parallelisierung kann mit genügend Prozessoren die Laufzeit des sequentiellen Algorithmus von Borůvka unterboten werden. Jedoch ist der Algorithmus von Kruskal für dichte Graphen ungeeignet.

Der Algorithmus von Prim wurde mit zwei verschiedenen Heaps untersucht. Dabei zeigt sich, dass der vergleichsweise einfache binäre Heap einem Fibonacci-Heap überlegen ist. Die einzige Ausnahme sind extrem große Graphen, bei denen der Fibonacci-Heap schneller ist. Zudem benötigt der Fibonacci-Heap den meisten Speicher und ist schwierig zu implementieren. Für kleine Graphen ist die Umsetzung mit dem binären Heap ähnlich schnell wie der Algorithmus von Kruskal. Bei sehr dichten Graphen stellt der Algorithmus von Prim die zweitbeste Wahl dar.

Insgesamt empfiehlt es sich daher für eine allgemeine Implementierung den Algorithmus von Borůvka zu verwenden.

Literaturverzeichnis

[BH01] Bazlamaçci, C. F.; Hindi, K. S.: Minimum-weight spanning tree algorithms - A survey and empirical study. Computers & Operations Research, 28:767–785, 2001.

[Bo26] Borůvka, O.: O jistém problému minimálním. Práca Moravské Přírodovědecké Společnosti, 3:37–58, 1926.

[DG98] Dehne, F.; Götz, S.: Practical Parallel Algorithms for Minimum Spanning Trees. IEEE Press, S. 366, 1998.

[Kr56] Kruskal, J. B.: On the shortest spanning subtree of a graph and the traveling salesman problem. Proceedings of the American Mathematical Society, 7:48–50, 1956.

[MS91] Moret, B. M. E.; Shapiro, H. D.: An Empirical Analysis of Algorithms for Constructing a Minimum Spanning Tree. In: DIMACS Series in Discrete Mathematics and Theoretical Computer Science. Springer, S. 400–411, 1991.

[Po08] Podsechin, Igor: , Comparing minimum spanning tree algorithms. http://www.aka.fi/Tiedostot/Tiedostot/Viksu/Viksu%202008/IgorPodsechin.pdf, 2008.

[Pr57] Prim, R. C.: Shortest connection networks and some generalizations. Bell System Technical Journal, 36:1389–1401, 1957.