2-Approximation des Pricecollecting-Steinerbaum-Problems

May 10, 2016

1 Motivation

Das vorliegende Projekt findet im Rahmen des Moduls "Parallele Algorithmen und Datenverarbeitung" statt. In dem Projekt geht es darum, einen gegebenen Algorithmus zu parallelisieren, in unserem Fall einen Algorithmus zur Approximation einer Lösung des minimalen Steinerbaum-Problems.

2 Einleitung

Ein Graph G ist beschrieben durch eine Knotenmenge V, eine Kantenmenge E und optional eine Gewichtsfunktion w, die jedem Knoten $v \in V$ eine nichtnegative reelle Zahl zuweist, d.h. $G = (V, E, w)$. Im Folgenden wird davon ausgegangen, dass der gegebene Graph zusammenhängend ist, d.h. es existiert für jedes Paar von Knoten in dem Graphen ein Pfad, der diese verbindet. Ein Graph $S = (V_S, E_S, w)$ ist ein Steinerbaum von einem Graphen $G = (V, E, w)$ für eine Knotenmenge $T \subset V$, wenn S ein Baum ist, alle Blätter von S int T enthalten sind und $T \subseteq V_S$ [**?**]. Die Knoten der Menge T werden Terminale genannte und die Knoten der Menge $V_S \backslash T$ Nichtterminale. Ein Steinerbaum S von G ist minimal, wenn die Summe der Kantengewichte von S unter allen Steinerbäumen S' von G minimal ist.

Die Berechnung minimaler Steinerbäume ist NP-Vollständig. Im folgenden Abschnitt wird ein Algorithmus zur Approximation minimaler Steinerbäume vorgestellt.

3 Verwendeter Algorithmus

Der Algorithmus (**??**) liefert eine 2-Approximation für die Berechnung minimaler Steinerbäume.

4 Implementation

4.1 Repräsentation des Graphen

Zur Repräsentation des Graphen werden zwei Datenstrukturen verwendet.

Input: Gewichteter Graph $G = (V, E, w)$; Menge T terminaler Knoten

1. Berechne den complete distance network $G_D = (T, E_D, w)$.
2. Berechne einen minimalen Spannbaum S_D von G_D.
3. Wandle S_D in einen gewichteten Teilgraphen N um, indem jede Kante aus S_D durch den Korrespondierenden kürzesten Pfad ersetzt wird.
4. Berechne einen minimalen Spannbaum S_N für N.
5. Wandle S_N in einen Steinerbaum um, indem nacheinander jeder Blattknoten entfernt wird, der kein Terminal ist.

Algorithmus 1 : 2-Approximation für die Berechnung minimaler Steinerbäume

Zum einen eine Form (Kanten-Array-Form) die als Eingabe, Ausgabe und zur Berechnung des Spannbaums verwendet wird. Diese besteht aus drei Arrays, $nodeI$, $nodeJ$ und $weights$, sowie der Anzahl der Knoten ($nodeCount$) und der Kanten ($edgeCount$). Der Graph enthält alle Knoten mit Index n mit $0 \leq n < nodeCount$ und alle gerichteten Kanten von Knoten $nodeI[i]$ zu $nodeJ[i]$ $\forall 0 \leq i < edgeCount$. Das Gewicht der Kante i wird dabei durch $weights[i]$ beschrieben.

Zum anderen arbeitet die Dijkstra-Implementation mit einer Darstellung durch Adjazenslisten. Hierbei werden zwei Int-Arrays verwendet. Das erste ($adjacencyList$) ist eine Aneinanderreihung aller Adjazenslisten und hat somit die Länge $edgeCount$. Das zweite ($listStart$) speichert an der Stelle i den Startpunkt der Adjazensliste von Knoten i innerhalb des ersten Arrays. Formal ist die Menge der Kanten also:
$\{(i,j)|0 \leq i < nodeCount, j \in \{adjacencyList[n]|listStart[i] \leq n < endOf(i)\}$
mit $endOf(i) = \begin{cases} listStart[i+1] & \text{falls } i+1 < nodeCount \\ edgeCount & \text{falls } i+1 \geq nodeCount \end{cases}$.

4.1.1 Konvertierung von Kanten-Array-Form in Adjazenslisten-Form

Es wurden drei Algorithmen zur Konvertierung implementiert und auf ihre Effiziens getestet.

ArraySort

Dieser Algorithmus nutzt den bereits implementierten Sortieralgorithmus von Java, wobei der Comperator überschrieben wurde, sodass die Kanten erst entsprechend dem Startknoten und zweitranging nach dem Zielknoten sortiert werden. Dieser Algorithmus arbeitet besonders effizient, falls die Eingabe bereits vorsortiert ist. Die allgemeine Effizienskklasse ist jedoch $O(M^2)$ wobei M der Anzahl der Kanten beschreibt.

CountingSort

Dieser Algorithmus macht sich zu Nutzen, dass es sich beim Sortieren um einen endlichen, abzählbaren Wertebereich ($0 \leq n < N$) handelt. Durch die temporäre überführung in eine Adjazensmatrix wird die Optimale Effizienskklasse $O(N^2)$ mit N der Anzahl der Knoten erreicht. Dies geschieht jedoch auf Kosten

des Speicherbedarfs welcher in jedem Fall N^2 beträgt.
PartlyCountingSort
Da der CountingSort Algorithmus gerade für spärliche Graphen (Diche $< 1\%$) nicht sinnvoll ist, arbeitet dieser Algorithmus etwas dynamischer. Für jeden Knoten wird eine Ajdazensliste in Form einer ArrayList mit den entsprechenden Folgeknoten gefüllt. Jede dieser Listen wird anschließend sortiert und anschließend in das adjacencyList-Array geschrieben. Die Effizienskласse ist $O(N^3)$, jedoch im zu erwartenden Durchschnitt am besten geeignet.

4.1.2 Iteration über Folgeknoten

Um über die Folgeknoten eines Knotens i zu iterieren gibt es verschiedene Möglichkeiten. Die effizienteste nutzt eine for-Schleife über die gesamte Adjazensliste vom Start- bis Endpunkt von i:

```
for(int m=getStartOf(i); m<getEndOf(i); m++) {
    int j = getToNode(m);
    float weight = getWeight(m);
}
```

4.2 Dijkstra

Da der Dijkstra-Algorithmus als Rückgabewert die kürzesten Pfade von einem Startknoten n zu allen anderen Knoten des Graphens besitzt, wurde hierfür eine eigene Klasse entworfen, welche nicht nur als Behälter dient, sondern in Form des Konstruktors auch den Algorithmus implementiert. Der Konstruktor erwartet somit einen Graphen, repräsentiert als AdjacencyList, und einen Startknoten. Zusätzlich können die Targets des Steinerbaums angegeben werden, um die Rechenzeit des Dijkstra's zu reduzieren. In diesem Fall ist jedoch nur noch gewährleistet, dass die kürzesten Pfade vom Startknoten zu allen Targets berechnet wurde und nicht mehr zu allen Knoten.
Um den Speicherbedarf des Ergebnisses zu reduzieren, wird nicht für jeden Knoten der gesamte Pfad gespeichert, sondern nur jeweils der Vorgängerknoten. Der Speicherbedarf sinkt dadurch von $O(N^2)$ auf $O(N)$, jedoch steigt die Zugriffszeit auf den kürzesten Pfad von $O(1)$ auf $O(log(N))$.

4.3 Minimale Spannbäume

Die Ausführungen in diesem Unterabschnitt richten sich nach [**?** , Kapitel 23]. Der in Abschnitt (**??**) beschreibene Algorithmus benötigt an zwei Stellen den minimalen Spannbaum eines Graphen $G = (V, E, w)$. Ein Graph $G_S = (V_S, E_S, w)$ ist ein minimaler Spannbaum für den Graphen G genau dann, wenn G_S eine Teilgraph von G ist, alle Knoten von G enthält und die Summe seiner Kantengewichte minimal ist. D.h. für G_S muss folgendes gelten: $V_S = V, E_S \subseteq E$ und $\sum_{e \in E_S} w(w)$ muss minimal sein.

Ein möglicher Algorithmus für die Berechnung minimaler Spannbäume ist der Algorithmus von Kruskal. Dieser ist in Algorithmus (**??**) dargestellt. Der Kruskal-Algorithmus startet initial mit einem Wald, in dem jeder Baum aus nur einem Knoten besteht. Dann wird sukzessive über alle Kanten (u, v) des Eingebegraphen G in nichtfallender Reihenfolge bezüglich des Kantengewichts

Input: $G = (V, E, w)$ $A = \emptyset$
for $v \in V$ **do**
  MAKE-SET(v)
Sortiere Kanten aus E in nichtfallender Reihenfolge nach dem Gewicht w
for $(u, v) \in E$ *in nichtfallender Reihenfolge nach ihren Gewichten* **do**
  if *FIND-SET(u)* $\neq$ *FIND-SET(v)* **then**
    $A = A \cup \{(u, v)\}$
    UNION(u,v)

Algorithmus 2 : Algorithmus von Kruskal

$w((u, v))$ iteriert. Gehören die Knoten u und v zu verschiedenen Bäumen B_1 und B_2, wird die Kante (u, v) zur Menge A hinzugefügt und die Bäume B_1 und B_2 werden zu einem Baum vereinigt. Am Ende enthält die Menge A die Kanten des minimalen Spannbaums zum Graphen G.

Für die Verwaltung des Waldes wird eine Datenstruktur disjunkter Mengen mit den Operationen MAKE-SET, FIND-SET und UNION benötigt. Die Operation MAKE-SET erstellt eine leere Menge, FIND-SET findet die Menge, die das gegebene Element enthält, und UNION vereinigt zwei Mengen. In unserer Implementation verwenden wir hierfür eine Version, die verkettete Listen für die Repräsentation der Mengen nutzt [**?** , Kapitel 21.2]. Hier wird für jede Menge ein repräsentatives Element ausgeweisen (meist das erste Element in der jeweiligen Liste), durch das jede Menge eindeutig repräsentiert wird. Zusätzlich dazu erhält jedes Element ein Attribut $\mathcal{R}$, mit dem das repräsentative Element der Menge gespeichert wird, zu dem das jeweilige Element gehört. Somit benötigt die Operation FIND-SET(E) konstante Laufzeit, da nur das Attribut $\mathcal{R}$ des Elements E abgefragt werden muss. Nehmen wir für die Operation UNION(M_1, M_2) o.B.d.A. an, dass die Menge M_2 der Menge M_1 hinzugefügt wird. Um dies zu realisieren muss die Liste der Menge M_1 aktualisiert werden und die Attribute $\mathcal{R}$ der Elemente der Menge M_2 müssen auf das repräsentative Element der Menge M_1 gesetzt werden. Hier ist es somit am effizientesten, die kleinere der beiden Mengen M_1 und M_2 zur größeren hinzuzufügen. Schließlich kann die Liste der Menge M_2 gelöscht werden.

Insgesamt besitzt der Algorithmus von Kruskal eine Laufzeit von $\mathcal{O}(|E| \log(|V|))$.

5 Ergebnisse und Diskussion