2-Approximation des Pricecollecting-Steinerbaum-Problems

June 14, 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.

Ein interessanter Anwendungsfall von Steinerbäumen sind Protein-Protein-Interaktionsnetzwerke in der Biologie. In einem solchen Netzwerk reräsentieren die Knoten Proteine und die Gewichte der Kanten beschreiben die Wahrscheinlichkeit, mit der das jeweilige Paar Proteine miteinander interagiert. Solche Netzwerke sind in der Regel sehr groß, so dass man häufig an Teilgraphen dieser Netzwerke interessiert ist. Diese Teilgraphen können durch Steinerbäume beschrieben werden, indem man eine Menge terminaler Knoten festlegt (siehe unten).

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 $St = (V_{St}, E_{St}, w)$ ist ein Steinerbaum von einem Graphen $G = (V, E, w)$ für eine Knotenmenge $S \subset V$, wenn St ein Baum ist, alle Blätter von St in S enthalten sind und $S \subseteq V_S$[3].

Die Knoten der Menge S werden Terminale genannte und die Knoten der Menge $V_S \setminus S$ Nichtterminale. Ein Steinerbaum St von G ist minimal, wenn die Summe der Kantengewichte von St 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 (1) liefert eine 2-Approximation für die Berechnung minimaler Steinerbäume wobei Knotenpreise gegen Kantengewichte aufgewogen werden.

Wir berufen uns hierbei auf den Algorithmus von Akhmedov et al.[2], den wir hier in leicht korrigierter Form ebenfalls benutzen.

Der Algorithmus von Akhmedov et al. enthält so wie er veröffentlicht wurde einen Fehler in Phase I. Wird die Schleife wie dort angegeben solange durchlaufen, wie $C' \leq C$, entsteht eine Endlosschleife und der Algorithmus terminiert nicht.

Input: Ungerichteter Graph mit Kantengewichten $G = (V, E, w)$;
Menge S terminaler Knoten; Preise für alle Knoten in S.

1. **Initialisierung:**
   (a) Wir setzen die Variable C auf ∞.
   (b) Wir setzen die Variable C' auf die Summe aller Kantengewichte im gesamten Graphen.
2. **Phase I:**
   Solange $C' < C$ tue folgendes:
   (a) C wird auf C' gesetzt.
   (b) Wir berechnen zunächst ein complete distance network und konstruieren dann einen Graphen $G' = (V', E', w)$. $V' = S$ und Elemente von E' korrespondieren jeweils zu den kürzesten Pfaden aus G.
   (c) Mittels des Kruskal-Algorithmus (2) berechnen wir einen minimal spanning tree von G' (T).
   (d) Wir setzen C' auf die Summe aller Kantengewichte in T.
   (e) T wird zurück in den Originalgraphen G eingebettet, wir erhalten einen Subgraphen T'.
   (f) S wird auf alle Knoten in T' gesetzt.
3. **Phase II:**
   Solange noch nicht alle Blätter v auf T' gepruned wurden, tue folgendes:
   (a) Berechne die Verbindungskosten cc_v von v.
   (b) Wenn Preis $p_v < cc_v$, dann prune v aus dem Baum.

Algorithmus 1 : 2-Approximation für die Berechnung minimaler Steinerbäume

4 Implementation

4.1 Repräsentation des Graphen

Zur Repräsentation des Graphen werden zwei Datenstrukturen verwendet.

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 $nodeJ[i]$ $\forall i \,.\, 0 \leq i < edgeCount$. 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 $endOf(i) = \begin{cases} listStart[i+1] & \text{falls } i+1 < nodeCount \\ edgeCount & \text{falls } i+1 \geq nodeCount \end{cases}$.

Es ist anzumerken, dass in unserer Implementierung im Verlauf des Algorithmus die Knotenanzahl des Graphen immer konstant bleibt. Einige Knoten sind hier nicht mit des Rest des Graphen verbunden (haben keine Kanten die zu ihnen hin oder von ihnen weg führen), werden jedoch für die Knotenanzahl noch mitgezählt.

Dies wurde so implementiert, damit wir über die Knotenanzahl eines Graphen eindeutig einzelne Knoten referenzieren können (d.h. Knoten 4 im Originalgraph ist auch immer noch Knoten 4 in bearbeiteten Graphen, auch wenn dieser Knoten inzwischen z.B. beim Pruning entfernt wurde). So umegehen wir komplexe Neuberechnungen des Graphen und der Knoten. Die hierbei entstehenden Laufzeitkosten sind vernachlässigbar.

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

Es wurden drei Algorithmen zur Konvertierung implementiert und auf ihre Effizienz 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 Effizienzklasse ist jedoch $\mathcal{O}(M^2)$ 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 Effizienzklasse $\mathcal{O}(N^2)$ 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 (Dichte $< 1\%$) 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 Effizienzklasse ist $\mathcal{O}(N^3)$, 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 $\mathcal{O}(N^2)$ auf $\mathcal{O}(N)$, jedoch steigt die Zugriffszeit auf den kürzesten Pfad von $\mathcal{O}(1)$ auf $\mathcal{O}(\log N)$.

Für zusammenhängende Graphen hat der Dijkstra-Algorithmus eine Laufzeit von $\mathcal{O}(|E| \log |V|)$. Durch die Verwendung eines Fibonacci-Heaps für die Realisierung der Prioritätswarteschlange kann die Laufzeit auf $\mathcal{O}(|V| \log |V| + |E|)$ reduziert werden.

4.3 Minimale Spannbäume

Die Ausführungen in diesem Unterabschnitt richten sich nach [1, Kapitel 23]. Der in Abschnitt (3) 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 ein

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 $\sum_{e \in E_S} w(e)$ muss minimal sein.

Ein möglicher Algorithmus für die Berechnung minimaler Spannbäume ist der Algorithmus von Kruskal. Dieser ist in Algorithmus (2) dargestellt. Der

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,
 d.h. in dieser Sortierung gilt für alle Kantenpaare e_i und e_j: $e_i \leq e_j$,
 falls $w(e_i) \leq w(e_j)$
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

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 $w((u, v))$ 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 [1, Kapitel 21.2]. Hier wird für jede Menge ein repräsentatives Element ausgewiesen (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 hat der Algorithmus von Kruskal eine Laufzeit von $\mathcal{O}(|E| \log(|V|))$.

4.4 Price-collecting

Das Pricecollecting in Phase II von Algorithmus 1 hat als Grundlage, dass für jeden Knoten im Ergebnisbaum von Phase I die Verbindungskosten (das heißt

das Kantengewicht der einzigen Kante dieses Knotens) abgefragt und mit dem Preis für den entsprechenden Knoten verglichen wird.

Sind die Verbindungskosten größer als der Preis, so wird der Knoten aus dem Baum entfernt (*gepruned*). Sind die Verbidungskosten gleich oder geringer dem Preis, so passiert nichts.

Dies wird so lange durchgeführt, bis es keine Knoten mehr gibt, die noch gepruned werden könnten. Hierbei ist zu beachten, dass nachdem ein Knoten entfernt wurde, eventuell andere Knoten zusätzliche Kandidaten für pruning werden.

4.4.1 Pruning

Beim Vorgang des Entfernen eines Knotens aus einem Baum (genannt *pruning*), wird in unserer Implementierung ein neuer Graph basierend auf einem Inputgraph zurück gegeben. Beide Graphen haben die gleiche Anzahl Knoten, allerdings werden alle Kanten von und zu dem zu entfernenden Knoten gelöscht.

Es ist anzumerken, dass für diese Zwecke vorgesehen ist, dass nur Knoten mit einer einzigen Kante entfernt werden können.

```
public static AdjacencyList prune(AdjacencyList g, int node)
{
    assert g.getDegree(node) == 1;
    int newEdgeCount = g.getEdgeCount() - 2;

    int[] newI   = new int[newEdgeCount];
    int[] newJ   = new int[newEdgeCount];
    float[] newW = new float[newEdgeCount];

    int k = 0;

    for (int i = 0; i < g.getEdgeCount(); i++)
    {
        if (!(g.getFromNode(i) == node || g.getToNode(i) == node))
        {
            newI[k] = g.getFromNode(i);
            newJ[k] = g.getToNode(i);
            newW[k] = g.getWeight(i);
            k++;
        }
    }

    AdjacencyList h = new AdjacencyList(g.getNodeCount(),
                                        newEdgeCount,
                                        newI, newJ, newW);
    return h;
}
```

5 Ergebnisse und Diskussion

References

[1] T.H. Cormen. *Algorithmen: eine Einführung.* Oldenbourg, 2010.

[2] Ivo Kwee Murodzhon Akhmedov and Roberto Montemann. A fast heur istic for the prize - collecting s teiner tree problem. *Lecture N otes in Management Science*, 6:207–216, 2014.

[3] Hans Jürgen Prömel and Angelika Steger. *The Steiner tree problem: a tour through graphs, algorithms, and complexity.* Springer Science & Business Media, 2002.