Densely Connected Biclusters

Stefan Dresselhaus, Thomas Pajenkamp

Parallele Algorithmen und Datenverarbeitung
Programmierprojekt im Wintersemester 2013/14

Universität Bielefeld – Technische Fakultät

29. März 2014

Inhaltsverzeichnis

Todo list

1 Zielsetzung des Projekts

Im Rahmen dieses Programmierprojekts wurde ein Programm entworfen und entwickelt, um Densely Connected Biclusters, im weiteren DCB, in einem biologischen Netzwerk zu ermitteln. Bei DCB handelt es sich um Teilgraphen eines Netzwerks, dessen Knoten untereinander hoch vernetzt sind und Objekte mit ähnlichen Eigenschaften repräsentieren.
Die Suche nach DCB ist ein NP-schweres Problem. Da mit einem geeigneten Algorithmus jedoch voneinander unabhängige Lösungspfade einzeln verfolgt werden können, ist das Problem gut für eine parallele Berechnung geeignet, wodurch die Gesamtlaufzeit stark reduziert werden kann. Zudem sind real verwendete biologische Netze üblicherweise nur schwach vernetzt. Daher können durch die Forderung nach einer hohen Konnektivität der DCB viele Lösungskandidaten schnell ausgeschlossen werden und die schlimmstenfalls nichtdeterministisch polynomielle Laufzeit findet kaum Anwendung.

1.1 Densely Connected Biclusters

Ausgangsbasis ist ein ungerichteter ungewichteter Graph $G = (V, E)$, dessen Knoten $n \in V$ mit jeweils p Attributen versehen sind. Jedem Knoten n ist zu jedem Attribut i ein numerischer Wert a_{ni} zugewiesen.
Ein DCB $D_k = (V_k, E_k)$ ist ein Teilgraph von G, der durch die Parameter $\alpha \in [0, 1]$, $\delta \in \mathbb{N}$ und $\omega \in \mathbb{R}^p$ beschränkt wird und die folgende drei Eigenschaften erfüllt.

- Der Teilgraph ist zusammenhängend.
- Die Dichte des Teilgraphen unterschreitet einen Schwellenwert α nicht, also $\frac{2 \cdot |E_k|}{|V_k|(|V_k|-1)} \geq \alpha$.
- Für mindestens δ Attribute liegen die Werte aller Knoten des Teilgraphen höchstens ω_i auseinander. Anders ausgedrückt

$$\delta \leq \left| \left\{ 1 \leq i \leq p \;\middle|\; \omega_k \geq \left(\max_{n \in V_k} a_{ni} - \min_{n \in V_k} a_{ni} \right) \right\} \right|.$$

2 Wahl der Programmiersprache und Konzept der Parallelisierung

Die Wahl der Programmiersprache zur Verwirklichung des Projekts beeinflusst stark die Methoden der Programmierung und die Art der Parallelisierung. Klassischerweise werden für sequentielle und parallele Programme gleichermaßen imperative Sprachen wie C(++), Fortran oder auch Java verwendet, wofür Erweiterungen zur parallelen Programmierung existieren oder einige Werkzeuge direkt in die Sprache eingebaut sind. Bekannte Ansätze hierfür sind MPI und openMP.
Unser Projekt geht in eine etwas andere Richtung. Bei imperativer Programmierung muss ein großes Augenmerk auf die Vermeidung unerwünschter wechselseitiger Beeinflussungen verschiedener Threads und Prozesse gelegt werden, die fehlerhafte Rechenergebnisse zur Folge haben. Außerdem muss bei der Thread-/Prozesskommunikation immer die Gefahr von Verklemmungen beachtet werden, die schlimmstenfalls zu einem kompletten Stillstand der Programmausführung führen. Beide Probleme sind schwierig zu detektieren und zu lokalisieren.
Die genannten klassischen Probleme des Parallelrechnens können mit pur funktionaler Programmierung gut vermieden werden. Nebenbedingungen treten in pur funktionalem Programmcode (einen korrekten Compiler/Interpreter vorausgesetzt) garantiert nicht auf. Da das DCB-Problem bis auf das Einlesen der Eingabedaten und die Ausgabe pur funktional realisierbar ist, ist

es optimal für eine derartige Implementierung geeignet Die konkrete Wahl der funktionalen Programmiersprache fiel auf *Haskell*.

Für Haskell wurden Bibliotheken entwickelt, die eine einfache und effiziente Programmierung paralleler Programme erlauben. Wir verwenden das Paket *parallel* in Verbindung mit *repa*-Arrays. Durch *parallel* können geeignete Algorithmen mit wenig Aufwand, aufgeteilt werden. Diese Funktionsaufrufe werden unevaluiert in ein Array gepackt und dort von freien Threads abgearbeitet. Diese Technik nennt man Work-Stealing und die noch nicht ausgewerteten Funktionen werden in Haskell *Sparks* genannt. Man kann sich dies als einen auf den Funktionsaufruf beschränkten light-weight Thread vorstellen - mit weniger Overhead. Die *repa*-Arrays bieten Funktionen, um die einzelnen Arrayelemente parallel zu berechnen. Mit diesen Techniken lässt sich sequentieller Programmcode einfach parallelisieren, da hierfür nur wenige Änderungen erforderlich sind. Es müssen lediglich die Berechnungsfunktionen an die parallelisierende Funktion übergeben und die Funktion zur Auswertung der Arrayelemente ausgetauscht werden.

Zwei wichtige Punkte müssen dennoch beachtet werden. Zum einen verwendet Haskell das Konzept *Lazy Evaluation*. Befehle werden immer nur soweit berechnet, wie sie an anderer Stelle benötigt werden. Dadurch entstehen manchmal zur Laufzeit große Bäume nur teilweise ausgewerteter Befehle, welche die Ausführungszeit durch eine hohe Garbage-Collector-Auslastung stark negativ beeinflussen. Es muss demnach darauf geachtet werden, die Berechnung später ohnehin erforderlicher Funktionen frühzeitig zu erzwingen. Zum anderen ist die Anzahl der Sparks standardmäßig nicht begrenzt, sodass auch hier zu große Arrays entstehen können, deren Abarbeitung allerdings im Verlaufe des Programms durch o.g. Lazy Evaluation evtl. gar nicht erforderlich ist. Daher beschränken wir die Anzahl der möglichen Sparks (und somit der maximal Möglichen Worker-Threads) auf 1000. Erwähnenswert ist noch, dass diese Technik *nicht* von Hyper-Threading profitiert (da nichtmal mehr ein Kontextwechsel der Threads nötig ist) und wir somit 1000 „echte“ Kerne für eine maximale Auslastung benötigen. Die obere Grenze wird hier dann eher durch Amdahls Gesetz, denn durch die verfügbaren Kerne beschränkt.

3 Der Algorithmus

Der DCB-Algorithmus besteht aus einer Vorverarbeitungsphase, in der Cluster-Seeds aus 2 jeweils verbundenen Knoten generiert werden und einer anschließenden Expansion dieser Seeds unter Berücksichtigung der in 1.1 vorgestellten Nebenbedingungen (Constraints). Diese Cluster (im Folgenden Graphen genannt) bestehen zu Anfang aus genau 2 Knoten, die sämtliche Bedingungen erfüllen. Eine erste Optimierung findet nun statt, da es hiernach auch Knoten geben kann, die nicht zur initialen Bildung der Graphen begetragen haben. Diese können im Folgenden komplett ausgeschlossen werden, da die einzige Bedingung, die diese nicht erfüllen konnten eine Attributs-Bedingung sein muss[1]. Folglich würde eine Hinzufügung dieses Knotens zu einem beliebigen Graphen diesen unweigerlich auch insgeamt gegen selbige Attributsbedingung verstossen lassen. Alle hiervon betroffenen Knoten können somit aus der Adjazienzmatrix gelöscht werden.
Wir exportieren 2 Funktionen nach außen, die in der Lage sind den Graphen zu expandieren: `step` und `maxDCB`. `step` expandiert alle Graphen und liefert diese - allerdings verliert man somit alle Graphen, die nicht expandiert werden konnten. Dies hat den Zweck, dass man eine gewisse mindestzahl an Knoten im Graphen hat. Da die Seeds mit 2 Knoten beginnen, man aber z. B. alle DCB mit 4 Knoten oder mehr haben möchte, kann man `step` so häufig aufrufen, dass alle Graphen mit weniger Knoten gar nicht zurückgegeben werden. Wir verwenden dies in unserem

[1] Ein Graph aus 2 verbundenen Knoten ist immer maximal dicht und zusammenhängend.

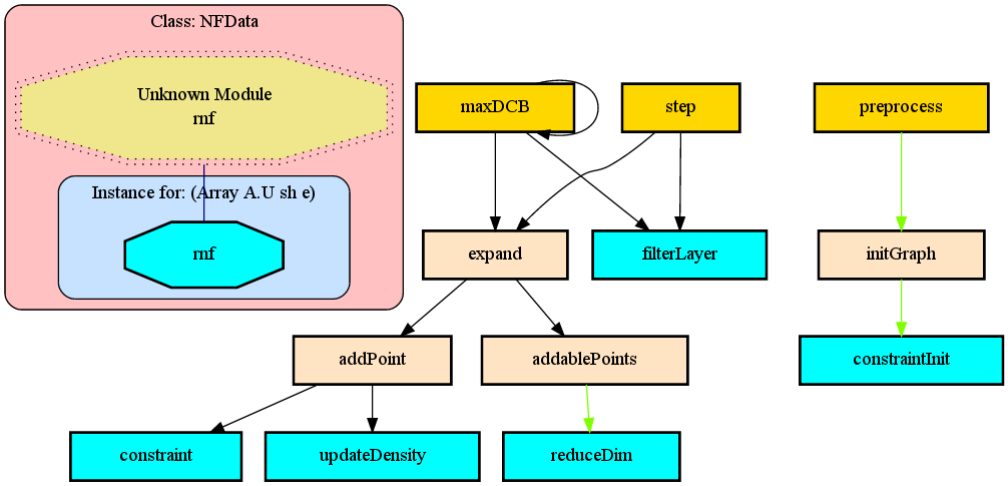

Abbildung 1: Übersicht über die Hierachie des DCB-Moduls. Gelbe Funktionen sind nach außen hin sichtbar.

Algorithmus einmalig, da wir nur Cluster mit 3 oder mehr Knoten zurückliefern.
Die Funktion maxDCB übernimmt die eigentliche Arbeit, sodass wir diese im Detail besprechen werden. Zunächst jedoch geben wir einen kleinen Überblick über die hilfs-Funktionen im Hintergrund:

filterLayer filtert eine Menge an Graphen, indem es Duplikate herausfiltert.
Laufzeit: $\mathcal{O}(n \log n)$

constraint überprüft, ob der Graph noch die Constraints erfüllt und wenn ja, wie diese aussehen.
Laufzeit: $\mathcal{O}(n^2)$ korrekt?

updateDensity errechnet die Änderung der Dichte des Graphen anhand des hinzuzufügenden Punktes.
Laufzeit: $\mathcal{O}(n \cdot m)$ bei m Knoten im Ursprungsgraph

reduceDim ist eine interne Hilfsfunktion, die eine Dimension einer Array-Shape verwirft.
Laufzeit: $\mathcal{O}(1)$

addablePoints traversiert die Adjazienzmatrix und liefert bei gegebenem Graphen alle potentiell hinzufügbaren Knoten zurück.
Laufzeit: $\mathcal{O}(n \cdot m)$ bei m Knoten im Ursprungsgraphen.

addPoint fügt einen neuen Knoten zu einem bestehenden Graphen hinzu, indem es zunächst ein `updateDensity` macht und anschließend mittels `constraint` überprüft, ob alle Nebenbedingungen noch erfüllt sind.
Laufzeit: $\mathcal{O}(n^2 + n \cdot m)$, falls die Density-Constraint erfüllt blebit, $\mathcal{O}(n \cdot m)$, falls nicht.

expand wendet `addPoint` auf alle Ergebnisse von `addablePoints` an.

Laufzeit-Analyse ist doof ... ich überseh sicher was oder mache Fehler. Auch kann ich mit O-Notation nicht zum Ausdruck bringen, was ich eigentlich sagen wollte ... nämlich, dass der Algo zwar in n*n*n läuft, aber wir z.b. bei addPoint in n*m liegen KÖNNEN, wenn die sterne gut stehen..

```
1  -- | Calculates all maximum DCB. A maximum DCB is a densely connected bicluster that
2  --   cannot be expanded by any additional node without breaking the constraints.
3  --
4  --   This function does return the seed graphs themselves if they cannot be expanded.
5  maxDCB :: [Graph] -> Adj -> Attr -> Density -> MaxDivergence -> Int -> [Graph]
6  maxDCB [] _ _ _ _ _ = []
7  maxDCB gs adj attr dens maxDiv minHit =
8      let next = L.map (expand adj attr dens maxDiv minHit) gs
9                       +|| (parBuffer 1000 rdeepseq)
10         (maximal, expandable) = part (\_ rm -> rm == []) (zip gs next)
11         expandable' = filterLayer $ concat expandable
12         -- Divide solutions into expandable solutions and maximum solutions.
13         -- Expandable solutions yield a result via the 'expand' function.
14     in maxDCB expandable' adj attr dens maxDiv minHit L.++ maximal
15     -- append maximum solutions of prospective function calls and maximum solutions
16     -- of this iteration
```

Listing 1: Die maxDCB-Funktion

Der rekursive Funktionsaufruf findet in Zeile 14 statt. Hier werden rekursiv alle expandierbaren Möglichkeiten evaluiert und alle nicht weiter expandierbaren Graphen angehängt. In Zeile 8/9 wird die Expansion auf den Eingabegraphen `gs` parallel in einem Buffer von höchstens 1000 parallelen Anweisungen ausgeführt. Die Strategie, welche wir für die parallele Evaluation verwenden ist `rdeepseq`, was bedeutet, dass diese Graphen nachher vollständig evaluiert vorliegen und nicht (z.B. durch lazy-evaluation) nach-evaluiert werden müssen.

Anschließend partitionieren wir die expandierten Graphen in bereits maximal Expandierte und in weiter expandierbare (Z. 10). Letztere filtern wir noch (Z. 11) nach duplikaten, um eine redundante Expansion (und damit einen erhöhten Rechenaufwand) zu vermeiden.
Zurückgeliefert werden somit alle Graphen, die maximal expandiert sind.

4 Ausführung und Auswertung

Amdahls Gesetz, Minskys Vermutung
Nach jedem Erweiterungsschritt: Sammeln und Aufgaben neu verteilen → Kommunikation

5 Fazit

Wir sind toll.