Verteilte Systeme

Minimale Aufspannende Bäume

Minimale Aufspannende Bäume

Auf einem beliebigen Graph G=(V,E) / Netz soll ein aufspannender UG G'=(V,E) mit E'⊆E mit minimalem Gesamtkantengewicht (Minimal aufspannender Baum, MST) berechnet werden.

Anwendungen:

- Rundruf (Broadcast) /Gruppenruf (Multicast)
- Auswahl an Alternativen in der Netzplanung

Berechnung eines MST:

Idee:

Wir starten mit einem aufspannenden Wald, trivialer Weise $(V,\varnothing)$ und nehmen sukzessive Kanten minimalen Gewichts hinzu, die keine Kreise einführen.
Wir hören damit auf, wenn der aufspannenden Wald genau eine Komponente hat.

Berechnung eines MST:

Verteilte Berechnung:

- Algorithmus von Prim/Dijkstra ist sequentiell
- Algorithmus von Kruskal bietet sich zur Parallelisierung an

Problem:

Durch Parallelisierung können Kreise entstehen

Lösung:

Falls alle Kantengewichte ungleich sind, kann es zu keinen Kreisen kommen.

Neue Gewichte: Vektor ($weigh_{i,j}$, i, j) wobei $i<j$

SyncGHS Algorithmus

Idee:

- Wir bauen den MST Stufenweise auf
- Auf jeder Stufe wird für jede Komponente die ausgehende Kante minimalen Gewichts (MWOE) bestimmt.
- Entlang der MWOEs werden Komponenten vereinigt.

- Jede Komponente des aufspannenden Walds in Stufe k soll mindestens 2^k Knoten haben.
- Für das Erreichen der nächsten Stufe sollen maximal $O(|V|)$ Runden benötigt werden.

SyncGHS Algorithmus

Wie bestimmen wir die MWOE?

- Die Wurzel sendet entlang der Kanten des MST einen Rundruf, verbreitet die Komponeten ID und fordert zur Bestimmung der MWOE auf.
- Jeder Knoten prüft durch Senden einer “test”-Nachricht mit der Komponeten ID für jede seiner Kanten, ob der adjazente Knoten zur gleichen Komponente gehört:
  - Falls ja: Kante kann in Zukunft ignoriert werden
  - Falls nein: kleinstes Kantengewicht zur Wurzel weiterleiten
- Das Gewicht der kleinsten Kante wird an die Wurzel gemeldet, diese gibt dann den Auftrag zum Vereinigen der Komponenten.

SyncGHS Algorithmus

Zustände der Kanten:

```
branch    - Kante ist Teil des MST
rejected  - Kante gehört zur selben Komponente und ist
            nicht im MST
unknown   - Noch nicht klassifiziert.
```

SyncGHS Algorithmus

Wie bestimmen wir Wurzel?

- Phase 0: trivial (n Bäume zu 1 Knoten)
- Phase k: Eine neue Komponente in Stufe k
  besteht aus j>2 Komponenten aus k-1,
  wobei zwei eine gemeinsame MWOE haben, der zu
  dieser Kante adjazente Knoten mit höchster ID
  wird Wurzel.

SyncGHS Algorithmus

Pro Stufe wird jede Komponente mit mindestens einer anderen Komponente vereinigt – Per Induktion folgt:

Jede Komponente des aufspannenden Walds in Stufe k hat mindestens 2^k Knoten.

=> Nach log(|V|) Phasen sind alle Knoten in einer Komponente, wir haben einen MST

Pro Stufe werden über jede Kante maximal zwei “test” Nachrichten versandt

Die Minima entlang des MST der Komponenten weitergeleitet.

Kommunikationskomplexität: O(|V| * log(|V|) + |E|)

Zeitkomplexität: O(|V| * log(|V|))

GHS Algorithmus

Den SyncGHS Algorithmus asynchron auszuführen brächte einige Probleme mit sich:

- Es kann passieren, dass zwei Knoten in einer Komponente sind, dies aber nicht bemerken, weil die Mitteilung über die letzte Komponentenverschmelzung noch nicht angekommen ist.
- Die Kommunikationskomplexität von SyncGHS wird primär dadurch beschränkt, dass die Komponenten phasenweise verschmelzen. Klappt dies nicht mehr: $\Omega(n^2)$
- Es ist unklar wie Komponenten auf “test”-Nachrichten von Komponenten anderen Stufen reagieren sollen.

GHS Algorithmus

Idee:

- Zwei Methoden zum vereinigen von Komponenten:
  - merge - vereinigt zwei Komponenten gleicher Stufe
  - absorb - vereinigt zwei Komponenten verschiedener Stufe

- Anpassung der MWOE Suche

- Anpassung der Stufen

GHS Algorithmus

merge vereinigt <u>genau</u> zwei Komponenten der Stufe k zu einer Komponenten der Stufe k+1.
Die Wurzel wird über die gemeinsame MWOE bestimmt.
Die neue Komponente hat also wie bei SyncGHS mindestens 2^{k+1} Knoten.

absorb vereinigt zwei Komponenten unterschiedlicher Stufe
die neue Komponente behält die Stufe, kID und Wurzel der Komponente höherer Stufe bei.

GHS Algorithmus

Analog zu SyncGHS wird von der Wurzel der Komponente die neue kID und damit die Aufforderung zur Suche einer MWOE verschickt, woraufhin die Knoten ihre Nachbarn testen.

Prozess i testet, ob sein Nachbar j in der selben Komponente ist:

- falls $kID_i = kID_j$:
  - i und j sind in einer Komponente
- falls kID_i != kID_j:
  - falls $lvl_i \leq lvl_j$:
    - i und j sind in unterschiedlichen Komponenten
  - falls $lvl_i > lvl_j$:
    - j muss mit der Antwort warten bis $lvl_i \leq lvl_j$ (!)

GHS Algorithmus

Warum führ “j muss mit der Antwort warten bis $lvl_i \leq lvl_j$” nicht zu Verklemmungen?

Das System kann immer noch Fortschritt machen:

- Es warten nur Komponenten höhere Stufe auf Komponenten kleineren Stufe
- Eine Komponente höherer Stufe kann immer noch eine Komponente kleinerer Stufe absorbieren, falls ihre MWOE zu dieser führt.

GHS Algorithmus

Nachrichten:

```
init           - Startet eine Suche nach der MWOE verteilt kID
                 und Stufe, initial von der Wurzel ausgesandt.
report         - Nachricht, um die kleinste Kante zur Wurzel
                 zurückzumelden

test           - Test, ob der adjazente Knoten in der selben
                 Komponente ist, enthält kID und Stufe
accept/reject - Antwort auf test
```

GHS Algorithmus

Nachrichten (Fortsetzung):

changeroot - Nachricht an den zur gewählten MWOE adjazenten Knoten, eine merge/absorb Operation einzuleiten

connect - Nachricht über die MWOE nachdem changeroot empfangen wurde:

- merge wird ausgelöst, falls beide zur MWOE adjazenten Knoten "connect" geschickt haben
- absorb wird ausgelöst, falls beide Komponenten unterschiedliche Stufe haben.

GHS Algorithmus

Kommunikationskomplexität:

Zwei Typen von Nachrichten:

- test/accept/reject $O(|E|)$
- MEOE-Suche $O(|V| * \log(|V|))$

$$O(|V| * \log(|V|) + |E|)$$

Zeitkomplexität:

$$O(|V| * \log(|V|) * (l+d))$$

Verteilte Systeme

Minimale unabhängige Menge

Minimale unabhängige Menge

Problem:

Eine begrenze Ressource (z.B. Übertragungsmedium) kann nur nicht von benachbarten Stationen geteilt werden. Es sollen aber so viele Stationen wie möglich Zugriff auf die Ressource erhalten

Idee:

Stationen einigen sich, wer die Ressource bekommt

Minimale unabhängige Menge

neues Problem:

Analog zur Koordinatorwahl lässt sich dies nicht deterministisch entscheiden!

neue Lösung:

Um die Symmetrie zu brechen wählen wir die “Gewinner” nichtdeterministisch aus.

Minimale unabhängige Menge (LubyMIS)

1. Runde:

   Jeder Prozess p wählt gleichverteilt zufällig eine Zahl $v \in [0,n^4]$ aus und sendet sie an seine Nachbarn

2. Runde:

   Falls $v > recv_i$ für alle Nachbarn i von p:

   sende “winner” an alle Nachbarn

   setzte p inaktiv

3. Runde:

   Falls “winner” von einen Nachbarn i von p empfangen:

   sende “looser” an alle Nachbarn

   setzte p inaktiv

4. Runde:

   Falls “looser” von einen Nachbarn i von p empfangen:

   entferne alle Kanten von P aus dem Graphen

Minimale unabhängige Menge (LubyMIS)

Terminierung:

Da es sehr unwahrscheinlich ist, dass zwei Prozesse dieselbe Zahl ziehen, werden jede Runde die Knoten inaktiv (Gewinner + Nachbarn/Verlierer)

Zeitkomplexität:

O(log n)

Übung 4 zum 7. Juni 2011

Implementieren Sie den asynchronen GHS-Algorithmus, und gehen Sie dabei wie folgt vor:

- Erweitern Sie die Klassen GHSNode
- Schreiben Sie eine Main-Methode, mit zwei Parametern:

  int n - Anzahl der Nodes

  float p - Wahrscheinlichkeit für die Existenz einer Kante zwischen zwei beliebige Knoten.

  die einen zusammenhängenden Graphen aus n GHSNodes erzeugt und anschließen den Algorithmus startet.
- Die Methoden von GHSNode sollten Meldung über jede Zustandsänderung auf der Standardausgabe machen.