GKA Lösungsdokumentation Aufgabe 2

von Chau Vu, Leonhard Pahlke und Jannik Sturhann (GKAP04_TeamC)

Aufgabenaufteilung:
- Chau Vu (Kruskal + Tests)
- Jannik Sturhann (Prim + Tests)
- Leonhard Pahlke (Graph Generator + Tests)

Bearbeitungszeitraum:
- 15.05.2019 – 19.05.2019 Prim Algorithmus und Tests
- 10.05.2019 – 17.05.2019 Graph Generator und Tests
- 14.05.2019 – 19.05.2019 Kruskal und Tests

Quellenangaben:
- Graphstream core - Stellt Graph Datenstruktur bereitstellt, die visuell dargestellt werden kann und auf der gearbeitet werden kann. (github.com/graphstream/gs-core)
- GraphStream JavaFX - Bietet Schnittstelle zwischen GraphStream und JavaFx Oberflächen (github.com/graphstream/gs-ui-javafx)
- FibonacciHeap Klasse - *Keith Schwarz (htiek@cs.stanford.edu)*
- *DisjointSet Klasse - GraphStream library*

Inhaltsverzeichnis

Kruskal Algorithm

Definition Minimum Spanning tree (MST)
Die Teilmenge der Kanten, die alle Scheitelpunkte im Diagramm verbinden und das minimale Gesamtgewicht haben. Wir benutzen Kruskal Algorithmus, um MST zu finden.

Disjoint Sets:

Definition: eine Datenstruktur, die alle Elemente verfolgt, die durch eine Anzahl disjunkter (nicht verbundener) Teilmengen getrennt sind.

Die Funktionen von DisjointSet():

add(E)
Fügt der Struktur eine neue Gruppe hinzu, die nur Kanten enthält. Wenn es bereits zu einigen der disjunkten Mengen gehört, passiert nichts

inSameSet(e1,e2)
Prüft, ob zwei Elemente zur gleichen Menge gehören.

union(e1,e2)
Beispiel : union(edge1, edge2) // Verbinde zwei Mengen zu einer einzigen Teilmenge

- Vereinigung der Menge mit edge1 und der Menge mit edge2.
- Nach dieser Operation gibt inSameSet (e1, e2) true zurück.
- Ob edge1 oder edge2 gehören zu keiner Menge oder sind es bereits gehören zu der gleichen Menge, passiert nichts.

Die Operation union() benutzt die Operation join() und root(), um die Wurzeln der Bäume zu bestimmen, zu denen x und y gehören.
Die Operation join() mit der Rangzuordnung verknüpft den kürzeren Baum immer mit der Wurzel des größeren Baums. Daher ist der resultierende Baum nicht höher als die Originale, es sei denn, sie sind gleich hoch. In diesem Fall ist der resultierende Baum einen Knoten höher.
Die Operation root() folgt der Kette der übergeordneten Zeiger von x bis zu einem Stammelement, dessen übergeordnetes Element es selbst ist. Dieses Wurzelelement ist das repräsentative Mitglied der Menge, zu der x gehört, und kann x selbst sein.
Durch die Pfadkomprimierung wird die Struktur des Baums geglättet, indem mit root () die einzelnen Knoten auf den Stamm verweisen. Dies ist gültig, da jedes Element, das auf dem Weg zu einer Wurzel besucht wird, Teil derselben Menge ist.

Algorithm

Der Kruskal-Algorithmus zum Auffinden des Spanning Tree mit minimalen Kosten verwendet den Greedy-Ansatz.

```
        KRUSKAL(G):
1 MST = ∅ // List to keep all the edges in MST
2 foreach v ∈ G.V:
3   DisjointSet = MAKE-DISJOINT-SET(v)
4 foreach (u, v) in G.E ordered by weight(u, v), increasing:
5   if UNION(FIND-SET(u), FIND-SET(v)) von DisjointSet :
6     MST = MST U {(u, v)}
7   if MST.size == G.getNodeCount -1:
8         break;
9 return MST
```

Implementierung

- Sortieren Sie die Diagrammkanten nach ihren Gewichten.
- Fügen Sie dem MST Kanten von der Kante mit dem kleinsten Gewicht bis zur Kante mit dem größten Gewicht hinzu.
- Fügen Sie nur Kanten hinzu, die keinen Zyklus bilden, Kanten, die nur nicht verbundene Komponenten verbinden.
- Wenn die Anzahl der Knoten in MST ist gleich als die Anzahl der Knoten in der Graph -1, dann enden wir den Prozess.
- Wir geben den MST aus.

Laufzeit des Algorithmus

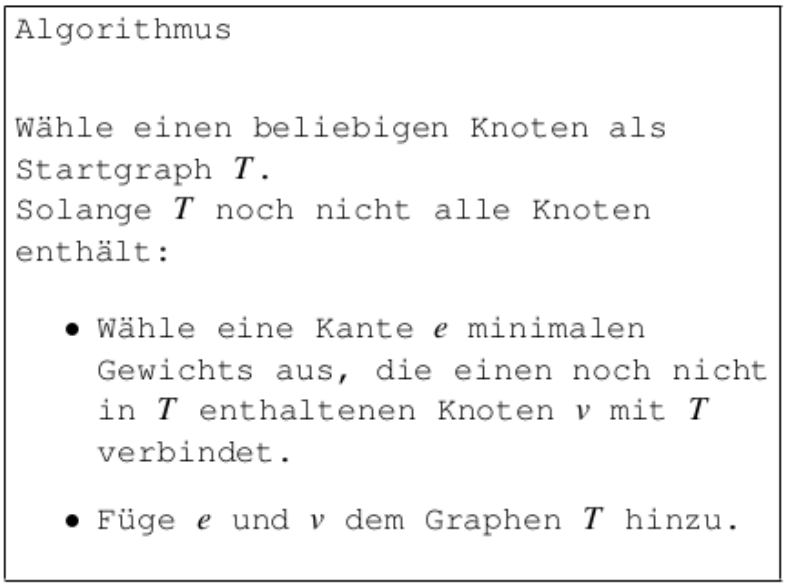
Es kann gezeigt werden, dass der Kruskal-Algorithmus in der Zeit O (E log E) oder äquivalent in der Zeit O (E log V) abläuft, wobei E die Anzahl der Kanten im Diagramm und V die Anzahl der Eckpunkte ist, alle mit einfachen Datenstrukturen

Wir können diese Grenze wie folgt erreichen: Zuerst sortieren wir die Kanten nach Gewicht unter Verwendung einer Vergleichssortierung in O (E log E) -Zeit.
Als Nächstes verwenden wir eine disjunkte Datenstruktur, um zu verfolgen, welche Knoten sich in welchen Set befinden. Wir müssen O (N) Operationen ausführen, da wir in jeder Iteration einen Knoten mit dem Spanning Tree (MST) verbinden, zwei Suchoperationen und möglicherweise eine Vereinigung für jede Kante.

Prim Algorithmus

Definition

Der Prim Algorithmus ist dafür gedacht ein minmales Gerüst in einem zusammenhängen ungerichteten Graphen zu finden. In einem Graphen G wird ein beliebiger Knoten v ausgesucht. Dieser wird zum leeren Spannbaum S hinzugefügt. Es werden alle an v anliegende Knoten markiert, die nicht in S enthalten sind und über eine Kante mit v verbunden sind. Der Wert der Markierung ist das Gewicht der verbindenden Kante. Dann wird der Knoten mit der kleinsten Markierung gewählt und samt Kante, die den Knoten zu v verbindet zu S hinzugefügt. Für den Knoten werden wieder alle Kanten markiert, die nicht in S enthalten sind. Dieses Vorgehen wird solange wiederholt, bis alle Knoten des Graphen G in S enthalten sind.



Implementierung

Wir haben uns dafür entschieden, für die Implementierung einen Fibonacci Heap zu verwenden. Dieser funktioniert ähnlich, wie eine Liste, allerdings wird jedem Wert ein Schlüssel zugeordnet, nach dem die Liste sortiert wird. Als Schlüssel verwenden wir die Markierungen der Knoten. Da nun eine Ordnung auf der Liste definiert ist, können wir sehr schnell den Knoten mit der kleinsten, bzw. der größten Markierung finden. Den Fibonacci Heap haben wir nicht selber implementiert sondern von *Keith Schwarz (htiek@cs.stanford.edu)*übernommen.

In der Initialisierung des Algorithmus fügen wir alle Knoten in den Fibonacci Heap ein und markieren diese mit positiver Unendlichkeit. Einen Knoten markieren wir mit dem maximalen Wert von Double. Das machen wir, damit, wenn wir später die Knoten aus dem Heap entfernen, und einen finden, der mit positiver Unendlichkeit markiert ist, wissen, dass der Graph nicht zusammenhängend ist und den Algorithmus abbrechen können. (Denn angrenzende Knoten werden immer mit einer Zahl markiert)

Wir entfernen den Knoten mit der Kleinsten Markierung und prüfen ob die Markierung gleich positive Unendlichkeit ist. Falls die der Fall ist brechen wir ab → Graph nicht zusammenhängend.

Andernfalls fügen wir die Kante, die den Knoten mit dem bestehenden Spannbaum verbindet zur Liste der Kanten des Spannbaumes hinzu.

Dann holen wir uns alle Kanten und Knoten, die an den gerade hinzugefügten Knoten angrenzen und noch im Fibonacci Heap vorhanden sind (also noch nicht im Spannbaum). Wir prüfen, ob der Schlüssel, der für ein Knoten im Fibonacci Heap hinterlegt ist, größer ist, als das Gewicht der Kante, das den Knoten mit dem bestehenden Spannbaum verbindet. Ist dies der Fall, aktualisieren wir den Schlüssel im Fibonacci Heap zu dem Gewicht der verbindenden Kante.

Dieses Vorgehen wiederholen wir in einer Schleife so lang, bis der Fibonacci Heap leer ist oder wir einen Knoten finden, der mit positiver Unendlichkeit markiert ist.

Zuletzt können wir über die Liste der Kanten des Spannbaumes das Gesamtgewicht des Spannbaumes berechnen und diesen auch ggf. zusammen mit den Knoten des Ausgangsgraphen als GraphStream Graph aufbauen.

GraphGenerator

Definition

Der *GraphGenerator* erstellt einen zusammenhängenden Graphen. Knoten werden dabei zufällig miteinander verbunden.

Implementierung

Die Idee ist zunächst alle Knoten in eine Liste *"unconnected"* zu packen. In dieser Liste sind jetzt alle Knoten des Graphen enthalten.
In einer weiteren Liste *"connected"* sind alle Knoten, die über Kanten mit dem Graphen verbunden sind. Diese Liste ist anfangs leer.

Wir nehmen nun einen zufälligen Knoten aus der *"unconnected"* Liste und bilden eine Kante zu einem Knoten in der *"connected"* Liste. Der Knoten der *"unconnected"* List wird das dieser Liste gelöscht und in die "connected" Liste gepackt.
Wenn die Liste *"unconnected"* leer ist, dann können wir sicher sein, dass der Graph zusammenhängend ist.

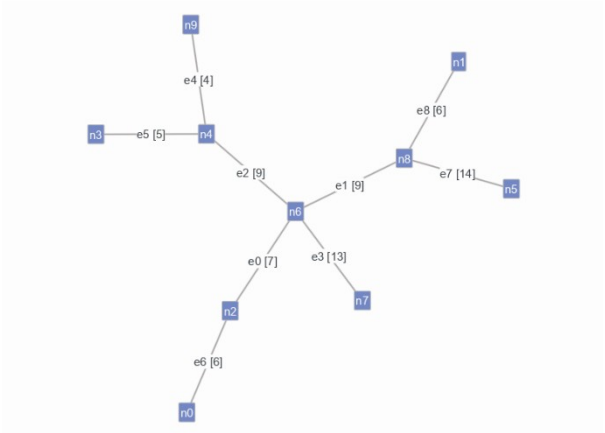


Abbildung 1 Graph nachdem Liste "unconnected" leer ist

Wenn jetzt noch Kanten erstellt werden sollen, dann nehmen wir zwei zufällig gewählte Knoten aus dem Graphen, die noch nicht verbunden sind und erstellen eine Kante. Fügt man beim oberen Graphen so weitere Kanten hinzu, dann könnte folgender Graph dabei entstehen.

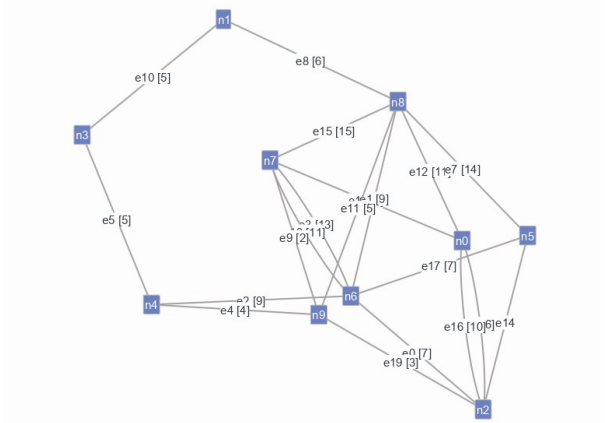


Abbildung 2 Graph nachdem restliche Kanten hinzugefügt worden sind

GraphGenerator Test

Wir haben folgende Eigenschaften des Graphens getestet.

1. Anzahl der Übergebenden Kanten und Knoten sind im Graph zu finden
2. Knoten und Kanten Gewichtungen sind nicht höher als die Maximale Gewichtung
3. Graph ist zusammenhängend

Zu 1. Wir prüfen ob im Graph ein Knoten / Kante auf dem Index (Anzahl der Kanten / Knoten – 1) liegt.
Zu 2. Wir traversieren über den Graphen und schauen, ob die derzeitige Kante und Knoten0, Knoten1 eine Gewichtung < Maximale Gewichtung hat.
Zu 3. Wir wählen ein Knoten aus dem Graphen und schauen, ob alle weiteren Knoten des Graphens mit diesem verbunden sind.