Ein Generator für Core-Graphen

Autoren: Sergej Müller und Thorsten Vogel

6. Juli 2006

Betreuer: Michael Baur

Institut für Theoretische Informatik
Lehrstuhl Prof. Dr. D. Wagner
Universität Karlsruhe (TH)

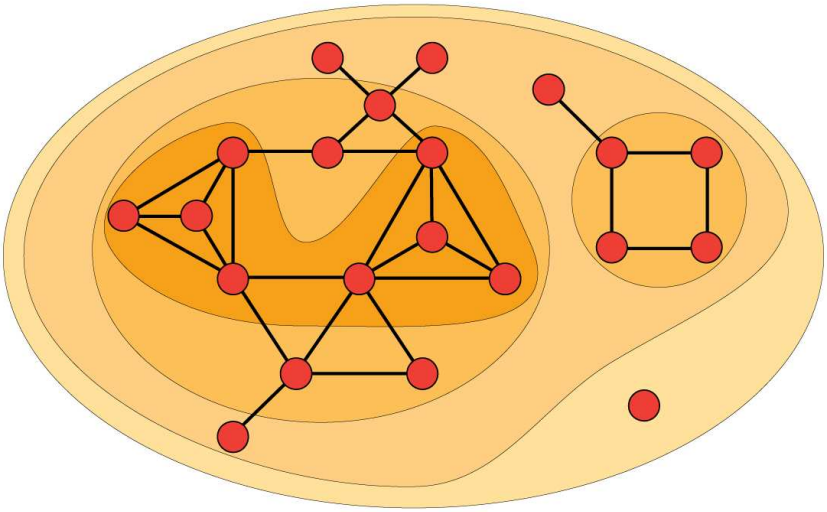
Zusammenfassung

Durch Dekomposition von Graphen, beruhend auf so genannte p-Cores, lassen sich ihre strukturellen Eigenschaften leichter erkennen. In diesem Dokument beschreiben wir verschiedene Verfahren, um Graphen mit vorgegebenem maximalen p-Core zu generieren. Dabei untersuchen wir zuerst einige einfache Verfahren und stellen anschließend ein rekursives Verfahren vor, das versucht, das durch das iterative Wachstum eines Graphen entstehende lokale Verhalten auszunutzen.

1 Einleitung

In dieser Arbeit untersuchen wir Algorithmen zum Generieren von Graphen, die die Eigenschaft haben, einen vorgegebenen maximalen p-Core zu enthalten. Dabei bauen wir auf die Arbeit von V. Batagelj und M. Zaversnik [2] auf. Die in diesem Dokument verwendeten Begriffe und Definitionen stellen wir im nachfolgendem Kapitel 2 vor. In Kapitel 3 beschäftigen wir uns zunächst mit naiven Verfahren. Mit den daraus gewonnenen Erkenntnissen stellen wir in Kapitel 4 einen iterativen Algorithmus vor, der sich in die zwei Teilschritte Strict-Core und Party-People einteilen lässt. Kapitel 5 beschäftigt sich letztendlich mit unseren in diesem Praktikum gewonnenen Erfahrungen und geht auf einige weitergehende Ideen bezüglich des vorgestellten Algorithmus ein.

Abbildung 1: Core-Dekomposition eines Graphen. Die verschiedenen p-Cores sind durch Farben markiert.



2 Definitionen

Kommen wir nun zu den in diesem Dokument verwendeten Begriffen.

In unserer Arbeit betrachten wir stehts ungerichtete Graphen $G = (V, E)$. Die Nachbarn eines Knotens $v \in V$ werden mit $N(v)$ bezeichnet; Δ bezeichnet den maximalen Knotengrad. Bevor wir eine genaue Definition zu p-Cores angeben, benötigen wir eine Funktion, die jedem Knoten eines Graphen sein Gewicht zuordnet.

Definition 1 (Gewichtsfunktion)**.**

- Eine Gewichtsfunktion p ordnet einem Knoten $v \in V$ in Abhängigkeit von einer Teilmenge $\bar{V} \subset V$ ein Gewicht zu, also $p : V \times \mathcal{P}(V) \to \mathbb{R}$.
- Eine Gewichtsfunktion p ist genau dann *monoton*, wenn gilt: $V_1 \subset V_2 \Rightarrow \forall v \in V : p(v, V_1) \leq p(v, V_2)$.
- Beschränkt sich eine Gewichtsfunktion nur auf die Nachbarn eines Knoten und sich selbst, nennen wir diese Gewichtsfunktion *lokale* Gewichtsfunktion:
  Eine Gewichtsfunktion p ist genau dann lokal, wenn gilt: $p(v, \bar{V}) = p(v, N(v, \bar{V}))$, wobei gilt:
  $N(v, \bar{V}) = N(v) \cap \bar{V}, \bar{V} \subseteq V$.

Damit lassen sich nun p-Cores definieren:

Definition 2 (p-Cores)**.** Sei $G = (V, E)$ ein Graph. Ein Teilgraph $H_G = (V_H, E_H)$, $V_H \subseteq V$, $E_H = E_{|V_H \times V_H}$ von G ist ein p-Core vom Level t, falls

- $\forall v \in V_H : t \leq p(v, V_H)$
- $|V_H|$ ist maximal
- p ist Gewichtsfunktion.

Um uns leichter ausdrücken zu können, führen wir an dieser Stelle den Begriff Core-Nummer ein.

Definition 3 (Core-Nummer)**.** Die *Core-Nummer* $core(v)$ eines Knotens v ist die Nummer des maximalen Cores, zu dem v gehört.

Die nachfolgenden naiven Verfahren in Kap. 3 nutzen zur Berechnung der Core-Nummern der Knoten eines Graphen einen 2001 von V. Batagelj und M. Zaversnik [2] vorgestellten Algorithmus (Algorithmus 1). Die Idee besteht darin, immer den Knoten mit dem kleinsten akutellen Wert zu betrachten. Um dies zu erreichen, wird ein Heap, sortiert nach den Knotengewichten, verwendet.

Algorithm 1 Bestimmen der Core-Nummern aller Knoten eines Graphen.

```
V̄ ↩ V
for v ∈ V do
  p[v] ↩ p(v, N(v, V̄))
end for
erstelle Heap(v, p)
while |heap| > 0 do
  entferne erstes Element top aus V̄
  core[top] ↩ p[top]
  for v ∈ N(top, V̄) do
    p[v] ↩ max(p[top], p(v, N(v, V̄)))
    aktualisiere Heap(v, p)
  end for
end while
```

Dieser Algorithmus bestimmt die *Core*-Nummern aller Knoten in einer Laufzeit von $O(m \cdot max(p, log(n)))$.

In den nachfolgenden Verfahren beschränken wir uns auf die lokale Gewichtsfunktion $p(v, U) = deg_U(v)$, die jedem Knoten seinen Grad zuordnet. Hier ist $p = O(\Delta)$.

3 Naive Verfahren

Unser Ziel ist es, die Generatoren möglichst allgemein zu halten. Deshalb beschränken wir sie auf folgende Eingabe-Parameter:

- Anzahl der Knoten n
- gewünschte Core-Nummer k.

Eine offensichtliche Bedingung für diese Parameter ist $n \geq k + 1$.

Schauen wir uns nun einige einfache Ansätze an, die trotz nicht zufriedenstellender Resultate zu wichtigen Erkenntnissen für unsere ausgereifteren Verfahren führten.

3.1 Expansion/Reduction

3.1.1 Idee

Eine einfache Methode einen Graphen mit gewünschter maximaler Core-Nummer zu generieren ist es, von einem leeren Graphen auszugehen und n Knoten und m Kanten zufällig iterativ hinzuzufügen. Dieses Modell ist allgemein als $G_{n,m}$ bekannt. Nach jeder Iteration wird die Core-Nummern neu berechnet. Sobald ein Knoten die max. Core-Nummer erreicht, terminiert der Algorithmus.

Eine Variation davon ist es, von einem vollständigen Graphen mit n Knoten auszugehen und aus diesem so lange zufällig iterativ Kanten zu entfernt, bis die gewünschte Core-Nummer k erreicht ist. Algorithmus 2 zeigt diesen Algorithmus als Pseudo-Code.

Algorithm 2 *reduction*(n, k)
Erstellt einen vollständigen Graph mit n Knoten. Entfernt so lange Kanten, bis die maximale Core-Nummer k beträgt.

```
erstelle vollstaendigen Graph mit n Knoten

while maximale Core-Nr > k do
  kante ↩ waehle zufaellige Kante
  entferne kante aus Graph
  maximale Core-Nr ↩ max(berechne Core-Nummern)
end while
```

3.1.2 Beobachtungen

Natürlich hat dieser Ansatz ein schlechtes Laufzeitverhalten - schließlich werden die Core-Nummern pro hinzugefügter bzw. entfernter Kante für alle Knoten neu berechnet. Die zweite Variante benötigt zudem anfangs noch mehr Speicher als der endgültige Graph.

Aber hier lässt sich bereits folgendes erkennen: Bei jeder Iteration werden nicht alle, sondern nur die maximale Core-Nummer im Graphen benötigt. Des weiteren fällt bei der Betrachtung der entstehenden Graphen auf, dass unabhängig von der verwendeten Variante alle generierten Graphen dazu neigen, einen großen Core mit einer hohen Core-Nummer, oftmals der gewünschten maximalen, zu enthalten.

Bereits an dieser Stelle wird dem aufmerksamen Leser klar, dass eine Einsparung der wiederholten Berechnung aller Core-Nummer eine dramatische Verbesserung bringen würde. In Kapitel 4 gehen wir auf dieses Thema etwas genauer ein. Das Problem der hier beobachteten großen Cores wird auch in Kapitel 4 etwas näher erläutert.

3.2 Preferential Attachment

3.2.1 Idee

Ausgangspunkt ist hier ein Graph $G = (V, E)$ mit $V = \emptyset, E = \emptyset$. Es werden iterativ n Knoten mit jeweils m Kanten hinzugefügt. Die Kanten werden bevorzugt zu Knoten mit hohem Grad gelegt. Eine bekannte Umsetzung dieses Verfahren ist der Barabasi-Albert-Algorithmus. Algorithmus 3 zeigt eine weitere Implementierung dieser Idee.

Algorithm 3 preferential attachment(n, m)
Erstellt n Knoten mit jeweils m Kanten, die sich bevorzugt zu Knoten mit hohem Grad verbinden.

```
for i = 1, ..., n do
  neuerKnoten ↩ erstelle Knoten
  for j = 1, ..., m do
    sortierteKnoten[] ↩ sortiere Knoten nach Grad
    for k = n, ..., 1 do
      if Zufallsexperiment erfolgreich then
        erstelle Kante von neuerKnoten zu sortierteKnoten[k]
        break
      end if
    end for
  end for
end for
```

3.2.2 Beobachtungen

Es ist offensichtlich, dass dieser Algorithmus (abhängig von den Zufallsvariablen) dazu neigt, einzelne wenige Knoten mit einer großen Anzahl an Kanten zu generieren. Die restlichen Knoten sind sehr nahe am minimalen Grad. Die Überlegungen zu verhindern, dass der Grad einzelner Knoten immer größer wird, führten zur folgenden Variante.

3.2.3 Verbesserung: Budget Attachment

Ausgangspunkt ist wieder ein Graph $G = (V, E)$ mit $V = \emptyset, E = \emptyset$, zu dem iterativ Knoten hinzugefügt werden. Neue Kanten werden wie bei *Preferential Attachment* bevorzugt zu Knoten mit hohem Grad gelegt. Zusätzlich wird der Grad, den ein Knoten maximal erreichen kann, durch ein Budget beschränkt (siehe Algorithmus 4).

3.2.4 Probleme

Diese Modifikation brachten leider auch keine großen Verbesserungen. Außerdem fehlt bei dieser Art der Generatoren die Möglichkeit, den maximalen Core als Parameter zu übergeben. Somit ist dieser Ansatz für unser Vorhaben alles andere als gut geeignet und wird aus diesem Grunde nicht mehr weiter verfolgt.

Algorithm 4 budget attachment(n, m)
Erstellt n Knoten mit jeweils m Kanten, die sich bevorzugt zu Knoten mit hohem Grad verbinden. Beschränkung durch begrenztes Budget.

```
for i = 1,...,n do
  neuerKnoten ↩ erstelle Knoten
  for j = 1,...,Anzahl Kanten pro Knoten do
    sortierteKnoten[] ↩ nach Grad sortierte Knoten
    for k = n,...,1 do
      if budget(neuerKnoten)-grad(sortierteKnoten[k])>0 then
        verbinde neuerKnoten mit sortierteKnoten[k]
        budget(neuerKnoten)↩ (budget(neuerKnoten)-
        grad(sortierteKnoten[k]))
      end if
    end for
  end for
end for
```

3.3 Erkenntnisse

Bewaffnet mit obigen Erkenntnissen motivieren wir im folgenden Kapitel einen etwas anderen Ansatz. Halten wir an dieser Stelle noch ein mal fest, worauf bei einem neuen Verfahren zu achten ist:

- **Ausnutzung der Lokalität**
  Beim iterativen Hinzufügen von Knoten/Kanten ändert sich die Core-Nummer oft nur in einer bestimmten lokalen Umgebung. Ein Beispiel hierzu wäre ein vollständiger Graph G mit $k+1$ Knoten, welchem ein weiterer Knoten mit $m < k$ Kanten hinzugefügt wird. Solange $m < k$ ist, ändert sich nur die Core-Nummer des neuen Knotens. Abb. 2 verdeutlicht diese Beobachtung. Wesentlich seltener kommt es vor, dass sich ein neuer Knoten global auf die Core-Nummern des gesamten Graphen auswirkt. Abb. 3 zeigt das obige Beispiel - jedoch werden hier $k+1$ Kanten vom neuen Knoten gelegt.

- **Core-Nummer + 1**
  Da wir uns in dieser Arbeit auf die lokale Gewichtsfunktion $p(v, \bar{V}) = \deg(v)$ beschränken, ändert sich beim Hinzufügen eines neuen Knotens in einen vorhandenen Graphen die Core-Nummer jedes Knotens maximal um 1.

- **Teilgraph mit $k+1$ Knoten und Core-Nummer k**
  Betrachtet man die Core-Graphen etwas genauer, so stellt man fest, dass sie sich in zwei Teilgraphen unterteilen lassen. Zum einen enthält jeder Core-Graph nach Definition einen zusammenhängenden Teilgraphen mit mindestens $k+1$ Knoten deren Core-Nummer genau k ist. Zum anderen besteht der Rest dieses Graphen aus $n-(k+1)$ Knoten, deren Core-Nummer kleiner k ist.

Abbildung 2: Das Entstehen einer nicht notwendigen Kante.
Hinzufügen von zwei Kanten - nur die Core-Nummer des neuen Knotens ändert sich.

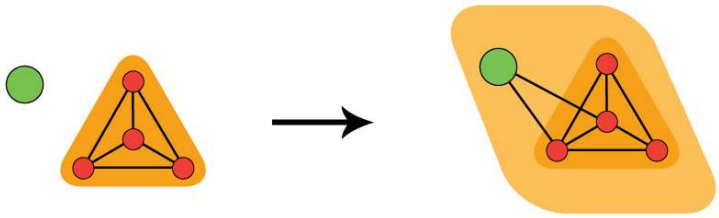
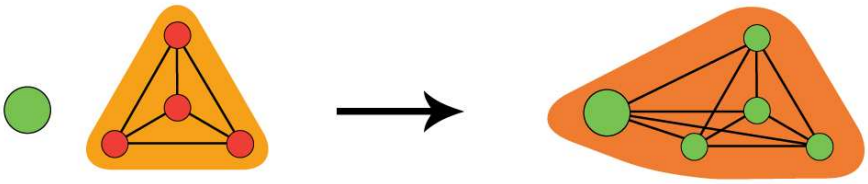


Abbildung 3: Das Entstehen einer nicht notwendigen Kante.
Hinzufügen von vier Kanten - die Core-Nummern aller Knoten ändert sich.



4 Strict-Core und Party-People

4.1 Idee

Der Versuch diese Beobachtungen unter ein Dach zu bekommen, führte zu folgendem Verfahren, bei dem das Generieren des Graphen in zwei Schritte aufgeteilt wird:

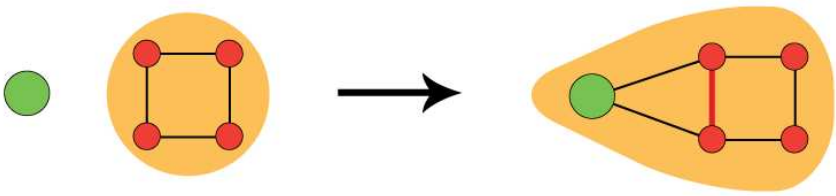
- Generiere einen Graphen mit mindestens $k+1$ Knoten, die alle Core-Nummer k haben (*Strict-Core*).
- Ergänze diesen Graphen iterativ auf n Knoten. Setze dabei auf eine rekursive Berechnung der Core-Nummer der lokal betroffenen Knoten (*Party-People*).

4.2 Strict-Core

In diesem Abschnitt konzentrieren wir uns auf das Generieren des bereits oben motivierten Teilgraphen, den wir ab hier *Strict-Core* nennen werden. Ein Strict-Core ist wie folgt definiert:

Definition 4 (Strict-Core)**.** Sei $G = (V, E)$ ein Graph. Ein Teilgraph $H_G = (V_H, E_H)$, $V_H \subseteq V$, $E_H = E_{|V \times V}$ von G ist ein k-Strict-Core, falls

Abbildung 4: Das Entstehen einer nicht notwendigen Kante.



(A) $\forall v \in V_H : core(v) = k$ und

(B) $|V_H| \geq k + 1$.

Um einen k-Strict-Core zu erhalten, geht man wie folgt vor:

Erstelle zuerst den kleinst möglichen k-Strict-Core. Nach Voraussetzung **(B)** besteht dieser aus mindestens $k+1$ Knoten und ist nach Voraussetzung **(A)** ein vollständiger Graph.

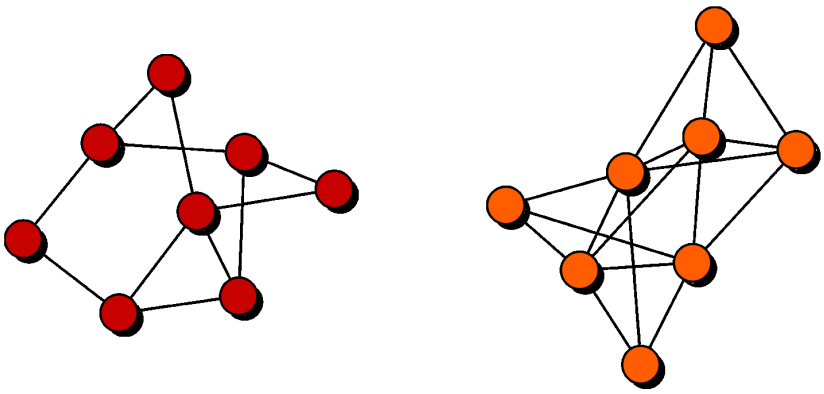
Im nächsten Schritt wird dieser kleinste k-Strict-Core iterativ mit weiteren Knoten ergänzt. Von jedem neuen Knoten werden genau k Kanten zum bestehenden Graphen gelegt. Durch dieses Verfahren bleiben die Bedingungen eines Strict-Cores erhalten. Bestehende Kanten können nun möglicherweise entfernt werden, ohne dass der Graph zu einem $(k-1)$-Strict-Core wird. Diese Kanten liegen an den Verbindungsknoten der neuen Kanten mit dem bestehenden Graphen. Da wir alle möglichen k-Strict-Core Graphen generieren können wollen, würfeln wir an dieser Stelle, ob diese unwesentlichen Kanten beibehalten oder aus dem Graphen entfernt werden. Abb. 4 veranschaulicht das Entstehen einer solchen Kante (rot).

Man beachte, dass durch dieses Vorgehen zu keinem Zeitpunkt die Berechnung der einzelnen Core-Nummern nötig ist, da diese beim gesamten Verfahren eine Invariant ist. Somit lässt sich ein relativ großer Teil eines Core-Graphen in deutlich besserer Laufzeit als bei den ersten Versuchen erstellen. Algorithmus 5 zeigt das Verfahren etwas übersichtlicher. In Abb. 5 sieht man einen mit diesem Verfahren generierten 2-Strict-Core. und einen möglichen 3-Strict-Core.

Algorithm 5 Strict-Core(n, k)
Erstellt einen Strict-Core mit n Knoten und einem k-Core.

```
generiere vollständigen Graph mit k + 1 Knoten
for i = 1, ..., n − (k + 1) do
    erstelle neuen Knoten
    Lege Kante zu k zufällig gewählten Knoten
    ermittle überflüssige Kanten
    entferne zufällige Anzahl überflüssiger Kanten
end for
```

Abbildung 5: Zwei Strict-Cores: (links) $k = 2$, (rechts) $k = 3$.



4.3 Party-People

Wie bereits motiviert, wollen wir nun einen Graph G, der mit dem Strict-Core-Algorithmus erstellt wurde, iterativ mit neuen Knoten anreichern. Dabei wollen wir insbesondere von den ersten zwei Merkmalen aus unserer Motivation Gebrauch machen.

Nachdem ein neuer Knoten v zu G hinzugefügt wurde, werden iterativ Kanten von v aus zu zufällig gewählten Knoten gelegt. Jede neue Kante im Graphen könnte eine mögliche Veränderung der Core-Nummer der Knoten nach sich ziehen. Diese Veränderung wird durch die von der neuen Kante betroffenen Knoten initiierte. Der Knoten v könnte somit zu einem Knoten $\bar{v}$ mit $core(\bar{v}) = core(v) + 1$ aufsteigen. Dieses Aufsteigen wollen wir zu Gunsten der Anschaulichkeit ab jetzt eine „z-Party feiern“ nennen, wobei $z \leq core(v) + 1$ ist. Wird also eine neue Kante von v zu einem anderen Knoten gelegt, so versucht v eine z-Party zu veranstalten. Aus der Definition einen k-Cores lässt sich leicht herleiten, dass für solch eine z-Party mindestens z Kanten zu z Nachbarknoten (im Nachfolgenden als „Gäste“ bezeichnet) mit je einer Core-Nummer von mindestens z gebraucht werden. Aus diesem Anlass verschickt der Knoten v nun Einladungen zu einer z-Party an alle seine Nachbarn. Folgende Situationen sind nun möglich:

- **Knoten v bekommt mindestens z Zusagen:**
  die z-Party kann stattfinden und v steigt in den Core mit Nummer $core(\bar{v}) = core(v) + 1$ auf.
- **Knoten v erhält weniger als z Zusagen:**
  es findet keine z-Party statt. Die Core-Nummer von v bleibt unverändert.

Kann ein Knoten an einer Party teilnehmen, so wird dies auf einer Einladungsliste mitprotokolliert. Die Antwort eines Knoten auf eine Einladung sei dabei wie folgt definiert:

Ein Knoten v kann an einer z-Party teilnehmen, falls

- $core(v) \geq z$ ist, oder
- Knoten v steht auf der Einladungsliste, oder
- $core(v) = z - 1$ und v bekommt selbst z Zusagen für eine z-Party (Rekursionsabstieg).

Kann eine z-Party nicht statt finden, da nicht genügend Gäste zusagen, muss die Party abgesagt werden. Alle betroffenen Gäste werden von der Einladungsliste gestrichen.

Zur Veranschaulichung zeigt Algorithmus 6 den Pseudo-Code der rekursiven Einladungsfunktion und Abb. 6 ein mögliches Szenario. Es folgt nun eine kurze Beschreibung der einzelnen Schritte:

Im *ersten Schritt* wird in den Graphen mit den Knoten **A**,**B**,**C**,**D** ein neuer Knoten **X** hinzugefügt. *Schritt 2* zeigt, dass eine neue Kante zu Knoten **C** gelegt wurde. Darauf verschickt Knoten **X** eine Einladung zu einer 1-Party an Knoten **C**. Dieser Antwortet unverzüglich mit einer Zusage. Im *Schritt 3* wird nun eine neue Kante zum Knoten **A** gelegt. Knoten **A** kann jedoch nicht direkt auf die Einladung zu einer 2-Party von Knoten **X** antworten, da seine Core-Nummer eins ist. Also versucht Knoten **A** mindestens 2 Zusagen zu einer 2-Party zu bekommen. Knoten **B** sagt unverzüglich zu. Auch Knoten **X** wird gefragt. Da dieser auch eine 2-Party feiern wollte sagt auch **X** zu (*Schritt 4*). Somit kann Knoten **A** wie in *Schritt 5* dargestellt eine 2-Party feiern und seine Core-Nummer wird auf zwei erhöht. Dies hat zur Folge, dass nun auch **A** auf die frühere Einladung von **X** zusagen kann. Knoten **C** antwortet wiederum auf die Einladung von **X** mit einer positiven Antwort (*Schritt 6*) und die Core-Nummer von Knoten **X** kann nun um eins erhöht werden.

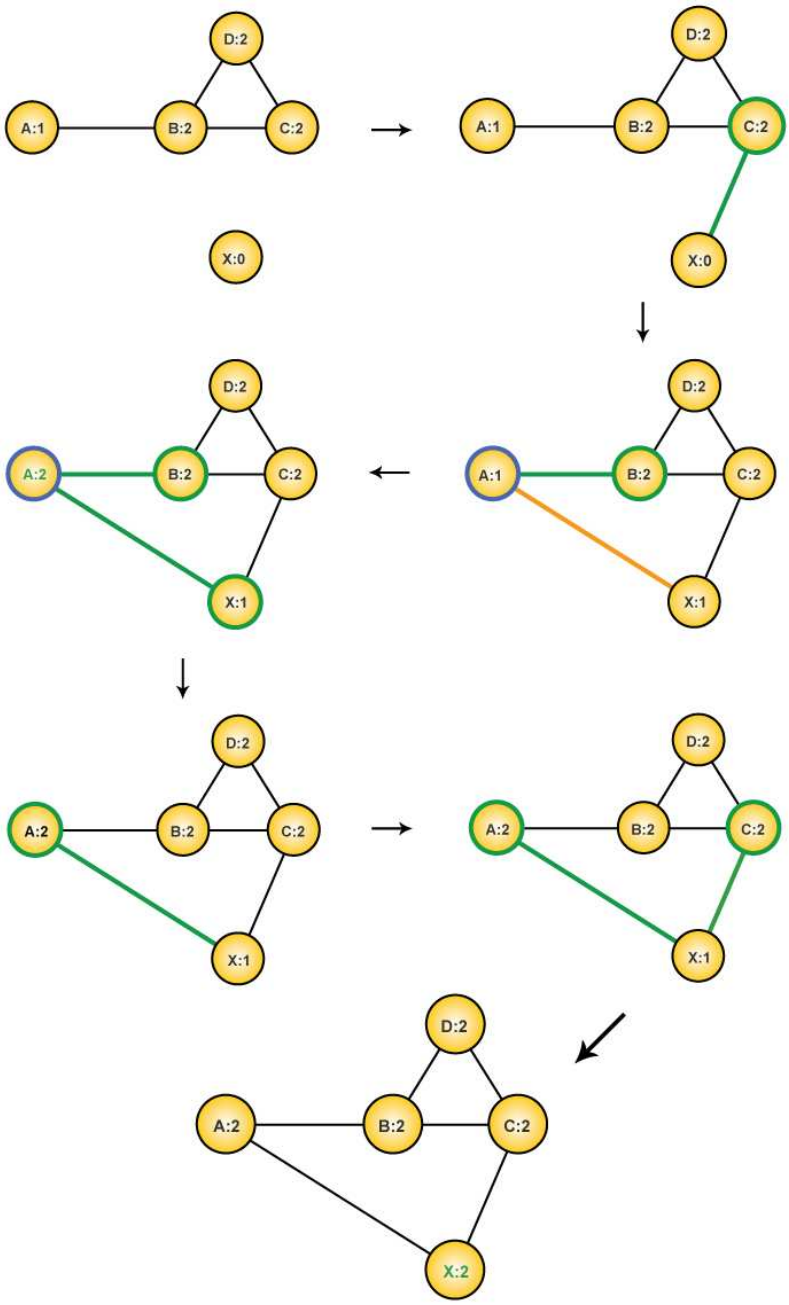
5 Fazit

5.1 Ergebnisse

Nach einigen Versuchen ist es uns in dieser Arbeit gelungen, ein Verfahren vorzustellen, das auf iterative Anreicherung eines Core-Graphen setzt und die Core-Nummern rekursiv berechnet.

Ein Problem, das uns bereits in den ersten Verfahren aufgefallen ist, blieb jedoch auch bei diesem Verfahren erhalten. Die Graphen tendieren dazu, große Cores mit einer hohen Anzahl an Knoten zu enthalten. Im Laufe unserer Arbeit stießen wir auf die Beschreibung so genannter Huge-Cores [4], die dieses Phänomen bei zufalls-generierten Graphen im Detail untersuchen. Da eine genauere Erklärung an dieser Stelle unseren Rahmen sprengen würde, sei der interessierte Leser hier auf die unten aufgeführten Referenzen diesbezüglich verwiesen.

Abbildung 6: Party-People: Aufbau einer 2-Party für Knoten X. Die Label zeigen die aktuellen Core-Nummern.



Algorithm 6 ladeEin(v, z)
Rekursive Einladungsfunktion. Der Knoten v versucht eine z-Party zu feiern.

```
if (core(v) ≥ k) oder (v ist schon eingeladen) then
  return k
end if

markiere k als eingeladen

for all Nachbarn als Nachbar do
  moeglicheCoreNr ↩ ladeEin(Nachbar, k)
  if moeglicheCoreNr = k then
    Zusagen ↩ Zusagen + 1
  end if
end for

if Zusagen ≥ k then
  core(v) ↩ k
  return k
else
  entferne eingeladene Knoten von Einladungsliste und setze deren Core-
  Nummern zurueck
  return core(v)
end if
```

5.2 Ausblick

Natürlich kann das von uns in dieser Arbeit vorgestellt Verfahren weiter verbessert und ausgebaut werden. Einige Ideen, die wir während unserer Arbeit diesbezüglich hatten, jedoch aus Zeitgründen nicht weiterverfolgen konnten, sollen an dieser Stelle festgehalten werden.

Huge Cores

Wie bereits beschrieben neigen fast alle von uns untersuchten Verfahren dazu, einen Graphen zu generieren, der aus einem einzigen großen Core, der fast alle Knoten enthält, besteht. Eine Möglichkeit dies unter Kontrolle zu bekommen ist es, die max. Core-Nummern einzelner Knoten bereits im Vorfeld festzulegen.

Aufwand

Um den Algorithmus Party-People auch für sehr große Graphen praktikabel zu machen, muss die Anzahl der Rekursionsaufrufe (Einladungen) weiter reduziert werden. Einen guten Anfang ist die Überarbeitung der Einladungsliste. Hier werden bei einer Absage alle Knoten bis zum initiierenden Knoten entfernt. Dies hat zur Folge, dass viele Rekursionen mehrfach berechnet werden, obwohl sich das Verhalten nur in den einzelnen Blättern unterscheidet.

Verbindung mehrerer Strict-Cores

Eine Variante des Party-People Generators, deren genauere Untersuchung interessant erscheint, ist es, mehrere Strict-Cores miteinander zu einem zusammenhängendem Graphen zu verbinden. Somit hätte man eine genaue Kontrolle über die Hierarchie der Cores in einem Graphen. Party-People scheint für diese Verschmelzung eine gute Wahl zu sein.

Literatur

[1] S. B. Seidman. Network structure and minimum degree. *Social Networks, 5, 269-287,* 1983.

[2] V. Batagelj, M. Zaversnik. Generalized cores, 2001.

[3] T. Luczak. Size and connectivity of the k-core of a random graph, 1991.

[4] B. Pittel, J. Spencer, N. Wormald. Sudden emergence of a giant k-core in a random graph, 1996.