Bericht zum Praktikum Algorithmen 2 von Konstantin Bachem (2430076)

Ich habe mich dazu entschieden, den Algorithmus PK2 aus dem Paper „A Batch Algorithm for Maintaining a Topological Order“ zu implementieren [Pearce & Kelly 2010]. Ich fand, dass der Algorithmus im Paper relativ schlecht erklärt wurde (für Leute, die keine Ahnung von dem Thema haben). Beispielsweise hatten mich die mathematischen Definitionen im Paper verwirrt und der Algorithmus wurde rückwärts erklärt. Allerdings war der Pseudo-Code im Paper sehr einfach implementierbar. Ich habe das ganze erst in Python implementiert, da ich mich in Python besser auskenne. Dann habe ich den Code mithilfe von ChatGPT in C++ übersetzt und dann zum Laufen gebracht und optimiert.

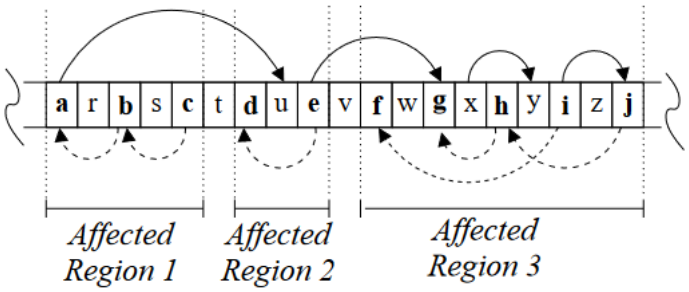
Algorithmus

Eine topologische Sortierung ist eine Reihenfolge der Knoten, bei der für jeden Knoten alle seine Nachfolger später vorkommen. Das Ziel ist es, die Knoten in eine Reihenfolge zu bringen, in der die Kanten immer nur „nach rechts“ zeigen.

Im Folgenden erkläre ich einen Algorithmus, um effizient mehrere Kanten zu einer topologischen Sortierung hinzuzufügen.

ADD_EDGE

Hier ist eine topologische Sortierung:



Wir möchten zu dieser topologischen Sortierung jetzt ein Batch von neuen Kanten hinzufügen. Wenn eine neue Kante die topologische Sortierung nicht invalidiert (also im Bild von links nach rechts geht) können wir sie direkt hinzufügen. Die anderen neuen Kanten, welche die Sortierung invalidieren (im Bild gestrichelt), können wir in sogenannte affected regions zusammenfassen. Diese affected regions können unabhängig voneinander behandelt werden und wenn alle affected regions topologisch sortiert sind, sind wir fertig. Dabei können alle Knoten, die nicht in

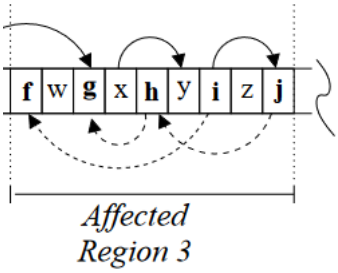
„affected regions“ sind, ignoriert werden. Der Algorithmus zur Berechnung der affected regions ist meiner Meinung nach trivial. Wir sortieren die Kanten absteigend nach dem Index, an dem sie starten und fügen dann die Kanten zu einer affected region hinzu, solange sie mit dieser überlappen. Im Paper ist das in der Methode ADD_EDGE(B) (Algorithm 4) und in meinem Programm heißt diese Methode insertedges(). In der Theorie könnte man die affected regions gleichzeitig mit Multithreading topologisch sortieren, allerdings habe ich das nicht gemacht.

Der nächste Schritt ist das Sortieren der affected regions, welcher der eigentliche Kern des Algorithmus ist. Das Sortieren der affected regions besteht aus zwei Schritten.

DISCOVER

Der erste Schritt ist „Algorithm 3 DISCOVER(B ⊆ E)“ im Paper und bei mir die Methode „discover“. Ich habe in meinem Algorithmus hier ein sort weglassen können, da die Kanten bereits von der vorherigen Methode sortiert wurden. In diesem Schritt werden mithilfe von Depth-First Traversal (DFS) ausgehend von den Enden der neuen Kanten alle Knoten, die in eine andere Reihenfolge gebracht werden müssen, als unbesetzt („vacant“) markiert und in die Liste Q geschrieben. Da DFS eine topologische Sortierung erzeugt, sind die Knoten in Q topologisch sortiert. Zusätzlich wird sich in Q noch gemerkt, nach welchem Knoten die Knoten später eingefügt werden müssen. Außerdem darf Q jeden Knoten nur einmal enthalten.

Wir haben beispielsweise folgende affected region:



Die neuen Kanten sind [(j,h),(i,f),(h,g)]. (Die Kanten wurden nach Index von ihrem Ende sortiert).

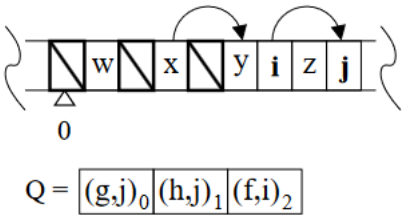
Wir starten mit der neuen Kante von j nach h. Das Ende dieser Kante ist h. Also starten wir DFS(h) und bekommen [g,h]. Dabei werden g und h als unbesetzt markiert. Um die topologische Sortierung zu korrigieren, müssen g und h nach j (dem Start der neuen Kante) eingefügt werden. Auch das merken wir uns in Q. Q sieht also so aus: [(g,j),(h,j)].

Als nächstes haben wir die neue Kante von i nach f. Das Ende dieser Kante ist f. Also starten wir DFS(f) und bekommen [f]. Dabei wird f als unbesetzt markiert. Um

die topologische Sortierung zu korrigieren, muss f nach i eingefügt werden. Auch das merken wir uns in Q. Q sieht jetzt also so aus: [(g,j),(h,j),(f,i)].

Als nächstes haben wir die neue Kante von h nach g. Das Ende dieser Kante ist g. Also starten wir DFS(g). g ist bereits markiert, was bedeutet, dass es bereits in Q ist. Deshalb wird es nicht in Q eingefügt.

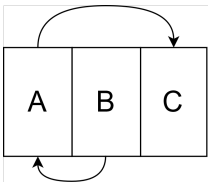
Das Ergebnis sieht dann so aus: (die null mit dem Pfeil kann ignoriert werden; ich habe das Bild für ein anderes Beispiel aus dem Paper geklaut)



Man kann das ganze folgendermaßen interpretieren:

Wie bereits erwähnt, ist Q topologisch sortiert, was bedeutet, dass in der Sortierung f -> h -> g (also f vor h und h vor g) sein muss (Q ist rückwärts). Um die topologische Sortierung aufzuräumen, muss f nach i eingefügt werden. Also wxyif…. Dann muss h nach j eingefügt werden. Also wxyifzjh…. Und danach muss g nach j (und g nach h wegen h -> g) eingefügt werden. Also wxyifzjhg….

Ich habe DFS im Beispiel vereinfacht. Im Algorithmus nimmt DFS zusätzlich zu einem Knoten noch einen upperbound. Alle Knoten über diesem upperbound sind entweder bereits in Q oder sind richtig rum sortiert.

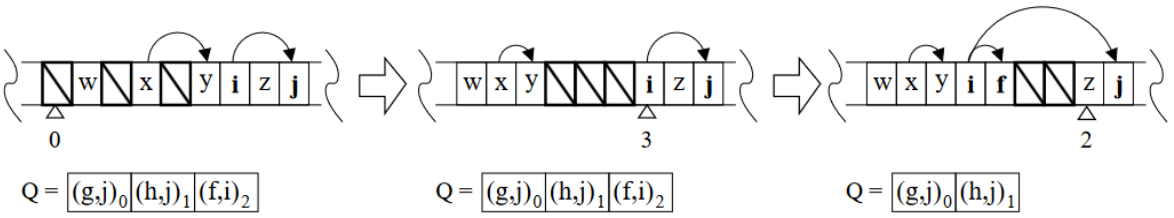


Q währe für dieses Beispiel ohne upperbound [(c,b),(a,b)]. Also a nach b und c nach a und b. Da c schon an der richtigen stelle ist, kann (c,b) ignoriert werden.

SHIFT

Der zweite Schritt ist „Algorithm 2 SHIFT(i, Q)“ im Paper und bei mir shift(i,Q). i ist hier der Index des Starts der affected region und Q ist das Ergebnis von „discovery“. Meine Version ist anders als die aus dem Paper da ich diese verwirrend fand. Allerdings funktioniert sie genau gleich. Ich habe ein Python Programm geschrieben (shiftprocess.py) was den Algorithmus Schritt für Schritt visualisiert.

Ich habe schon erklärt, wie Q strukturiert ist. Shift nimmt jetzt Q und die affected region mit den markierten Knoten und sortiert diese auf Basis von Q.



shift schiebt die unbesetzten Nodes nach links. Dabei beobachten wir immer das letzte Element von Q. In diesem Fall (f,i), was bedeutet, dass f nach i eingefügt werden muss. Wenn wir also bei i angekommen sind und i verschoben haben, fügen wir f ein usw.

Hier ist die Visualisierung meines Programms:

nodetoinsert=f, insertafter=i sind das letzte Element von Q und blaue Knoten sind als unbesetzt markiert. Grün hinterlegt bedeutet, dass dieser Bereich schon fertig ist. Die graue Region kann ignoriert werden. Wir gehen jetzt also durch die affected region und verschieben den aktuellen Knoten (Magenta) oder wir ignorieren ihn, wenn er als unbesetzt maskiert ist.

```
nodetoinsert=f, insertafter=i
wwgxhyizj
 ^ ^
shifting x

nodetoinsert=f, insertafter=i
wxgxhyizj
  ^ ^
ignoring h

nodetoinsert=f, insertafter=i
wxgxhyizj
  ^  ^
```

Wenn wir einen Knoten verschieben, welcher in Q als Einfügeposition (insertafter) ist, fügen wir danach den Knoten aus Q hinzu.

```
nodetoinsert=h, insertafter=j
wxyifzizj
      ^ ^
shifting j

nodetoinsert=h, insertafter=j
wxyifzjzj
       ^^
inserting h

nodetoinsert=g, insertafter=j
```

```
wxyifzjhj
        ^
inserting g

nodetoinsert=None, insertafter=None
wxyifzjhg
         ^
```

Wenn Q leer ist, sind wir fertig mit der affected region.

Implementierungsdetails

Mein gesamter relevanter C++ Code ist in topord.cpp und fullinserttimingtestargs.cpp und testcorrectness.cpp.

Hier ist einfacher code, der einen zufälligen azyklischen Graph erzeugt und diesen in eine topologische Sortierung einfügt. Dabei wird t alle insertsintervall Kanten mit der Methode insertedges() topologisch sortiert (d.h. wir haben eine Batch Size von insertsintervall). Die Methode addedge(start ,stop) fügt die Kante also nur in eine interne Warteliste ein und insertedges() fügt die Kanten von dieser Warteliste in die topologische Sortierung ein.

```
auto [randnodes, randedges]= makeGraph(nodenum,maxedgenum);
topologicalordering<int> t;
t.reserve(randnodes.size());
for(int i=0;i<randnodes.size();i++)
    t.addNode(randnodes[i]);

for(int i = 0; i < randedges.size(); ++i) {
    if (i % insertsintervall == 0) {
        t.insertedges();
    }

    auto [start, stop] = randedges[i];
    t.addedge(start, stop);

}
```

Die eigentliche topologische Sortierung kann mit dem [] Operator abgerufen werden. Hierbei muss man beachten, dass die Sortierung nur bei einem Aufruf von insertedges neu berechnet wird. Beispielsweise:

```
for(int i=0;i<t.size();i++){
    std::cout<<t[i];
}
std::cout<<std::endl;
```

Dieser Code ist relativ ähnlich zu meinem Testcode in fullinserttimingtestargs.cpp. Man kann das reserve weglassen, allerdings ist das Ganze dann langsamer und man kann auch t.addNode weglassen. Allerdings sind dann in t nur Knoten, welche mit addedge hinzugefügt wurden.

Intern sieht topologicalordering vereinfacht so aus:

```
template <typename T>
class topologicalordering{
        std::vector<Node<T>*> ordinv;
        std::vector<Edge<T>> newedges;
        std::unordered_map< T, Node<T>* > ValueToNode;
        int usednodes=0;
...
}

template <typename T>
class Node {
    bool vacant = false;
    bool onStack = false;
    int ord;
    std::vector<Node<T>*> children;
    T value;

    Node(int ord,T value) : ord(ord), value(value) {}
};
```

Der eigentliche Graph wird in Node gespeichert. „ordinv“ ist die topologische Sortierung als Liste von Nodes. „ord“ zeigt an welchem Index die Node in „ordinv“ ist. „newedges“ ist die Warteliste für neue, noch nicht eingefügte Kanten. Wenn wir addNode(T start,T end) aufrufen, rufen wir es mit Knoten vom Typ T auf anstatt mit Typ Node. Um die Node für einen Knoten zu bekommen, benutze ich „ValueToNode“.

„usednodes“ gehört zu einer Erweiterung des Algorithmus welche dafür sorgt das Knoten, die keine ausgehenden Kannten haben rechts von Knoten mit ausgehenden Kanten in der topologischen Sortierung sind. Diese Erweiterung verhindert das Knoten welche „ungenutzt“ sind geshiftet werden müssen was den Algorithmus schneller machen müsste.

Performancetests

Meinen Code habe ich mit einer Kombination aus Python und C++ getestet. Der C++ Test Code ist in fullinserttimingtestargs.cpp. Ich habe für alle Tests den Code mit -O2 kompiliert. Der C++ Code erzeugt einen zufälligen topologisch sortierbaren Graph. Dann wird der Graph in eine topologische Sortierung eingeführt und es werden in

regelmäßigen Abständen Zeitmessungen gemacht. Der Code nimmt folgende Parameter an:

<nodenum> <randseed> <insertsintervall> <repeats> <edgenum> <samplesintervall> <insertnodes>

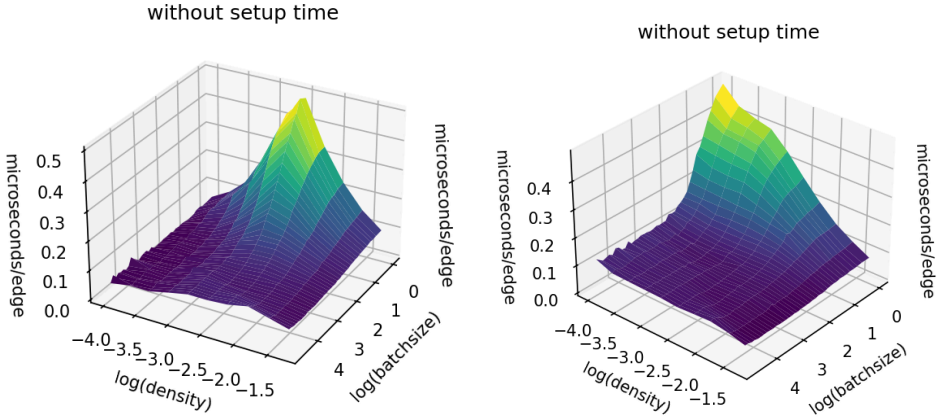
nodenum=anzahl Knoten
randseed=seed für wiederholbarkeit
insertsintervall=batchsize
repeats=wie oft der test wiederholt wird, zeiten werden gemittelt
edgenum=anzahl der Kanten
samplesintervall=jede sampleintervallte Kante wird eine Zeit gemessen
insertnodes=modus für inserts d.h. ob die nodes am anfang eingefügt werden oder ob nodehandles genutzt werden sollten

Zurückgegeben wird ein Plot, der die Anzahl eingefügte Kanten gegen Zeit in Millisekunden plottet.

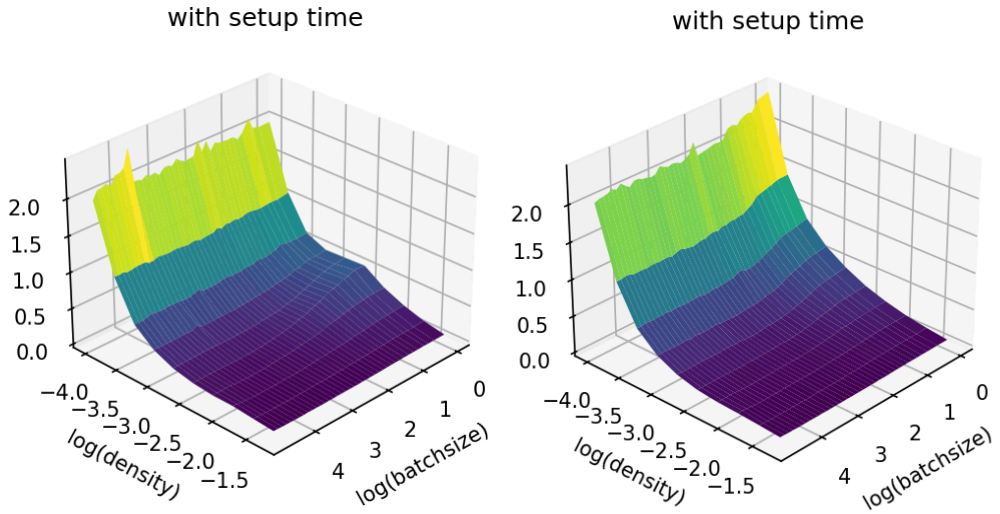
Dieser Plot kann mit den Python Programmen dargestellt werden.

Das Programm fullinserttimingtestargshandles2_3d.py misst die Millisekunden pro Kante in Abhängigkeit von Graph Dichte und Batch Size. Das heißt, für jedes Dichte-Batchsize-Paar erzeugen wir einen Graphen und fügen diesen in eine topologische Sortierung mit der Batchsize ein. Dabei messen wir, wie lange das Einfügen dauert. Geplottet wird dann diese Zeit geteilt durch Anzahl der Kanten und wir erhalten die durchschnittlichen Millisekunden pro Kante. Leider unterstützt mathplotlib keine logarithmischen Achsen in 3D d.h. die -4 bei Dichte ist eine Graph Dichte von 0.0001 und auch das plotten von mehreren Surfaces in 3D wird nicht gut unterstützt.

Das linke Diagramm ist mit meiner Erweiterung und das rechte ohne. (beim rechten Programm ist markNodeAsUsed() auskommentiert).

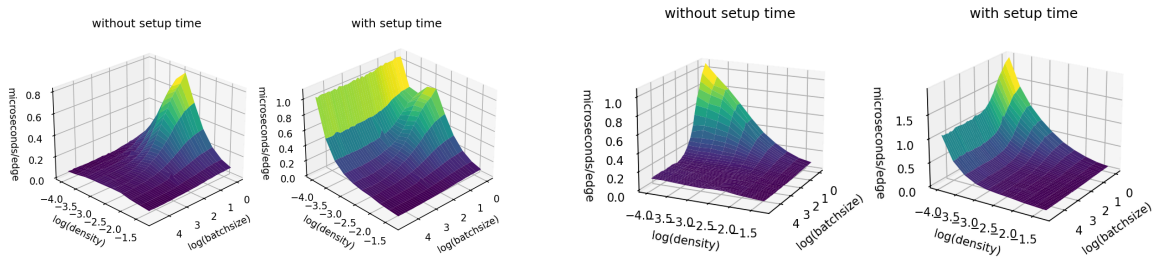


Hierbei muss beachtet werden, dass die Setup Zeit hierbei nicht berücksichtigt wurde. Bei den Tests wurden am Anfang alle Knoten hinzugefügt. Die Zeit, die für das Einfügen der Knoten aufgewendet wurde, wurde aus den Graphen ohne Setup-Zeit herausgerechnet. Mit Setup-Zeit sieht das ganze so aus:

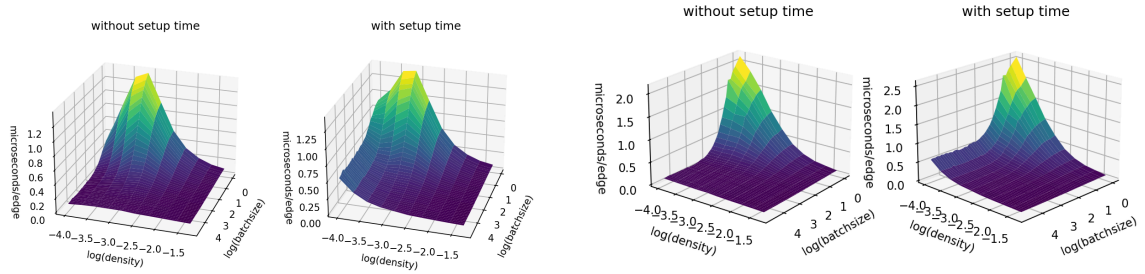


Auch hier ist das Ganze links schneller, allerdings ist der Unterschied nicht mehr sehr groß. Das liegt daran, dass die Zeit pro Kante gemessen wird. Wenn ich beispielsweise 2500 Knoten hinzufüge und nur 3 Kanten, ist die Zeit pro Kante sehr hoch (weil es nur so wenige Kanten sind). Wir plotten hier also (Zeit_Knoten + Zeit_Kanten) / Anzahl_Kanten = Zeit_Knoten / Anzahl_Kanten + Zeit_Kanten / Anzahl_Kanten. Da Zeit_Kanten / Anzahl_Kanten klein ist, überwiegt Zeit_Knoten / Anzahl_Kanten. Das bedeutet, wir plotten 1/x. Da die maximale Kantenanzahl quadratisch mit der Anzahl der Knoten wächst, lohnt sich das Ganze allerdings für Graphen mit mehr Knoten. Die linken Diagramme zeigen meine Erweiterung, während die rechten Diagramme den ursprünglichen Algorithmus zeigen.

5000 Knoten



10000 Knoten

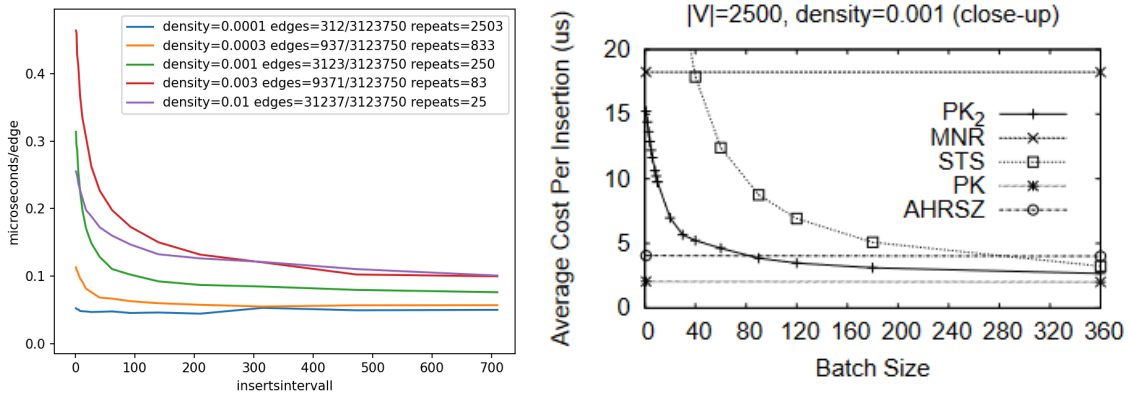


Die Einstellungen für diese Plots sind:

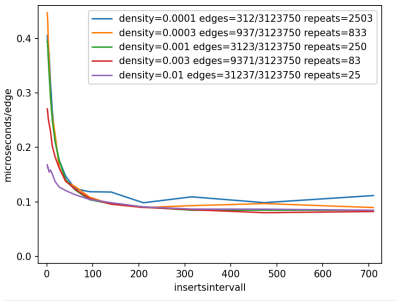
```
logdensity=True
logbatchsize=True
randseed = 42
nodenum = 2500

batchsizeminmaxstep=[1,50000,1.3]
densetyminmaxstep=[0.0001,0.1,2]
```

Mein Code erreicht seine langsamste Laufzeit pro Kante bei einer Dichte von etwa 0.00316, was bei 2500 Knoten ungefähr 4 Kanten pro Knoten entspricht. Das bedeutet, dass mein Algorithmus für Graphen mit geringer Dichte schneller ist als der ursprüngliche Algorithmus und bei hoher Dichte ungefähr gleich schnell. Dabei berechne ich die Dichte als (Anzahl_Kanten * 2) / ((Anzahl_Knoten - 1) * Anzahl_Knoten).

Ich habe auch noch dieses Graph gemacht (links), der im Wesentlichen das gleiche zeigt wie die 3d Graphen, aber dem Graph im Paper ähnlicher ist (rechts). (fullinserttimingtestargshandles2.py) (kein log und nicht 3D)



Hier nochmal ohne meine erweiterung:

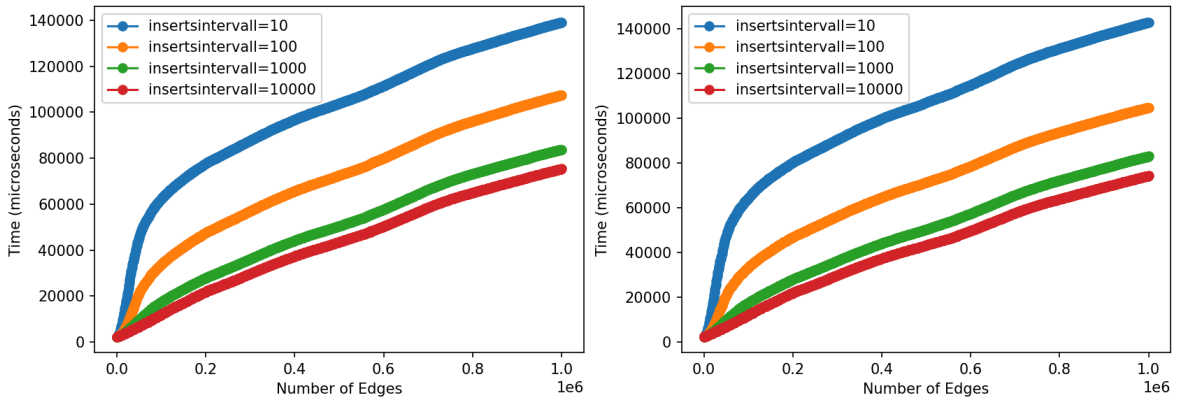


Auswirkung von hashing auf Performance

Bei meinen Tests habe ich mithilfe eines Profilers festgestellt, dass der Großteil der Zeit durch das Hashen in Sets und Dictionaries verbraucht wird, insbesondere bei dichten Graphen. Es gab zwei Stellen im Programm, an denen gehasht wird, beide in der Methode addEdge(Edge<T> edge). Ursprünglich dachte ich, dass Knoten nicht zweimal als Kinder eines Knoten existieren dürfen (d.h. doppelte Kanten), weshalb ich children als unordered_set implementierte. Allerdings hat sich herausgestellt, dass der Algorithmus damit keine Probleme hat. Dennoch könnte das Hinzufügen von Kanten mehrfach zu Leistungsproblemen führen, weshalb ich die alte Implementierung in topordwithhash.cpp belassen habe. Bei einem Graph mit 10000 Knoten und 1000000 Kanten war diese Implementierung ca. 6 mal langsamer.

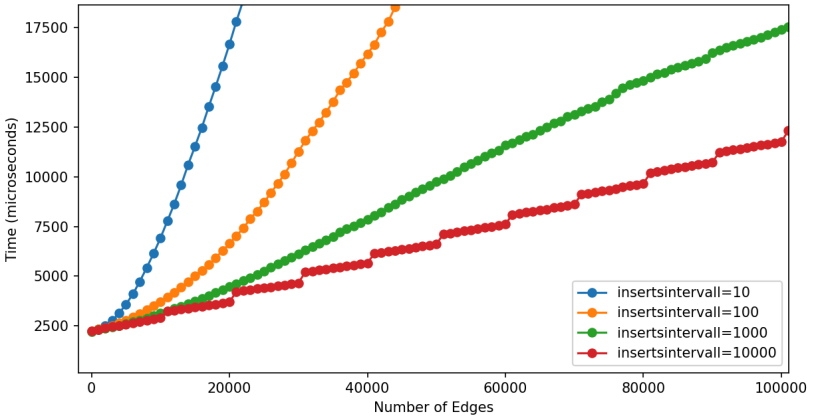
Die zweite Stelle, an der gehasht wird, ist das valueToNode-Dictionary. Um dieses zu umgehen, habe ich Code geschrieben, der es nicht benötigt (NodeHandles). Allerdings brachte das Optimieren dieser Stelle nicht viel, da die Laufzeit praktisch gleich blieb.

Die Diagramme zeigen Anzahl_Kanten/Zeit für eine topologische Sortierung. Das heißt, dass wir einen Graph in die topologische Sortierung einfügen und alle 1000 Kanten eine Zeitmessung machen. (fullinserttimingtestargshandles.py) (links mit Dictionary und rechts ohne)



Die Linien verlaufen hier relativ parallel, was dadurch kommt, dass der Graph relativ dicht ist und deshalb, wenn die Dichte hoch genug wird, schon fast sortiert ist. Dabei muss beachtet werden, dass der Graph mit jeder Iteration (d.h. Kanteneinfügung) dichter wird.

In diesem Ausschnitt sieht man auch nochmal (zumindest bei der roten Kurve), dass am Anfang ein relativ großer Teil der Zeit mit der Sortierung verbracht wird. Die Stufen in der roten Kurve entstehen durch den Aufruf von insertedges, während bei den geraden Abschnitten nur Kanten in die interne Warteliste geschrieben werden. Man sieht die Stufen bei den anderen nicht, da ich nur alle 1000 Kanten einen Messwert genommen habe. Außerdem sieht man hier, dass wir schon bei 0 edges bei ca 2500 microseconds sind. Das ist deshalb so, weil durch die Setup Time (d.h. durch das Einfügen der Knoten) diese Zeit verbraucht wurde.



WWorstC-CaseA-Analyse

In worstdesety.py wird analysiert, bei welcher Dichte das Einfügen am langsamsten ist.

Die x-Achse steht in beiden Diagrammen für die Knotenanzahl eines Graphen.

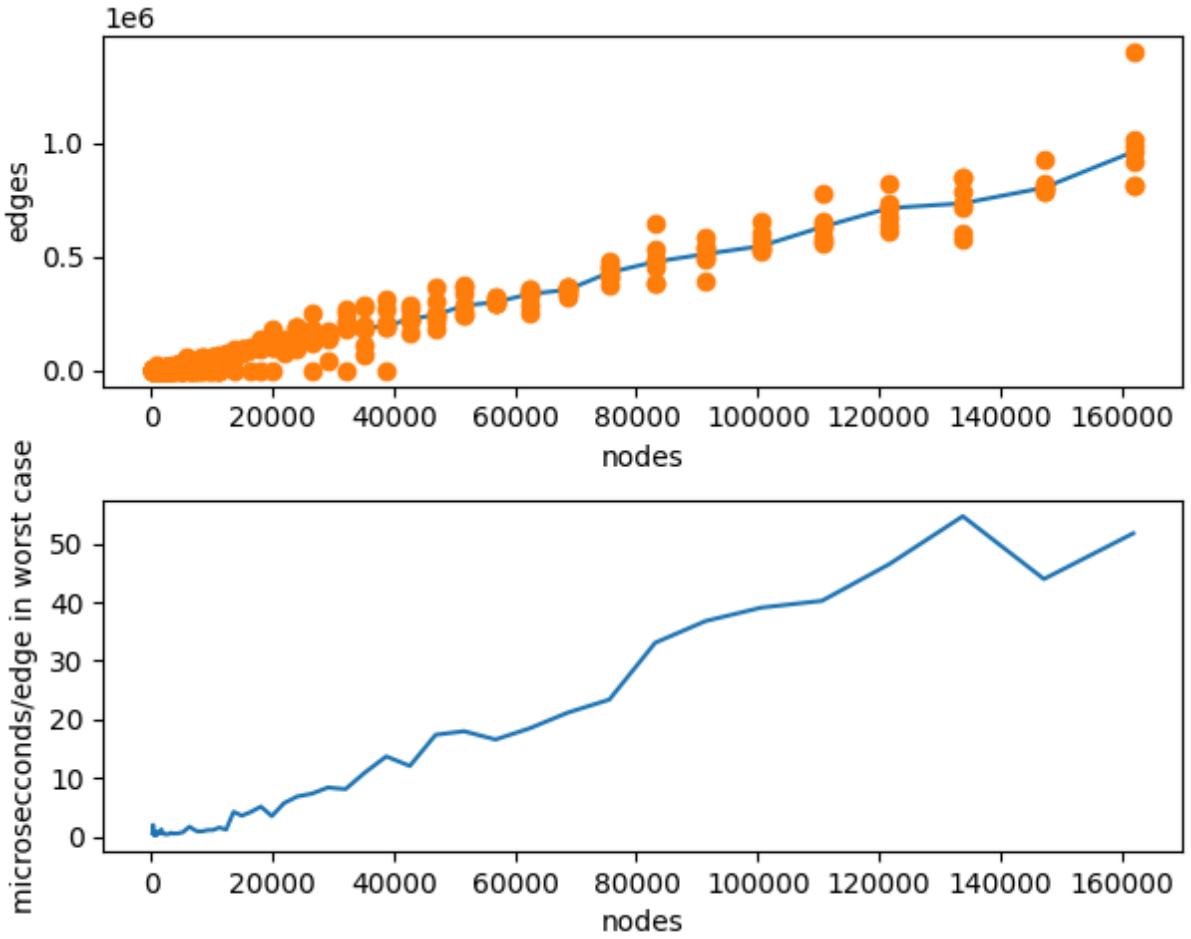
Für jeden x-Wert wurden mehrere Graphen gemacht.

Dabei sind die orangen Werte einzelne Messpunkte und die blaue Kurve in beiden Diagrammen der Median.

Der obere Graphen zeigt, bei welcher Kante microseconds/Kante maximal war. Das heißt vereinfacht, wir gucken, bei welcher Dichte die Microseconds/Kante am langsamsten war. Allerdings geben wir auf der y-Achse nicht die Dichte an, sondern wie viele Kanten hinzugefügt wurden. Man könnte das ganze in eine Dichte

umrechnen, allerdings sieht man so den linearen Zusammenhang besser. Bei ca. 6 Kanten/Knoten ist mein Algorithmus am langsamsten (der Plot hat eine Steigung von ca 6). Die Kurve im unteren Diagramm zeigt , wie viel Mikroseconds/Kanten wir brauchen. Wir plotten also den Worst Case in Abhängigkeit von der Knotenanzahl. Dabei muss beachtet werden, dass ich Batches von 100 Kanten gemacht habe und auch hier wieder die Zeit für das Einfügen der Knoten rausgenommen habe. Ich glaube, wenn ich andere Batch Sizes nehme, könnte sich der Worst Case verschieben. Außerdem habe ich nicht den ganzen Graphen eingefügt, da dies zu lange dauern würde. Stattdessen habe ich 30 mal so viele Kanten wie Knoten eingefügt. Es könnte also sein, dass der Algorithmus bei Graphen mit hohen Dichten wieder länger braucht (d.h. es könnte ein zweites lokales Maximum außerhalb der Messreichweite geben). Allerdings halte ich das für unwahrscheinlich. Weiterhin wurde dieser Test auf einem anderen Computer wie alle anderen Tests gemacht.

Insgesamt zeigt uns dieses Diagramm, dass im Worst Case die Zeit/Kante ungefähr linear abhängig von der Anzahl der Knoten ist.



Test Correctness

Die Tests funktionieren durch einen Wrapper um topologicalordering<int>. Dieser merkt sich welche Kanten/Knoten hinzugefügt wurden. Außerdem merkt er sich, welche Kanten nicht hinzugefügt werden konnten. Dieser Wrapper hat die Methode iscorrect welche hauptsächlich überprüft, ob die Kanten in der richtigen Reihenfolge sind und ob alles konsistent ist.

Die Tests werden mit relativ kleinen Graphen gemacht da iscorrect relativ langsam ist.

Es gibt 4 Tests:

Im randomtopgraphtest wird ein zufälliger, topologisch sortierbarer Graph erzeugt. Anschließend werden mit verschiedenen Batchgrößen die Kanten dem Wrapper hinzugefügt. Nach jedem Aufruf von insertEdges wird die Methode iscorrect aufgerufen. Falls diese Methode false zurückgibt, wird ein Fehler geworfen (hauptsächlich weil den Fehler so gut mit dem Debugger beobachten konnte). Am Ende wird außerdem überprüft, dass es keine falschen Kanten gibt, d.h. Kanten, die nicht hinzugefügt werden konnten. Dieser gesamte Prozess wird dann mehrfach mit verschiedenen topologisch sortierbaren Graphen wiederholt.

Im testcyclesonly wird ähnlich wie im vorherigen Test vorgegangen. Statt jedoch einen zufälligen topologisch sortierbaren Graphen zu erzeugen, wird hier ein Graph erzeugt, der aus m Zyklen besteht, die sich nicht "berühren". Zum Beispiel ein Graph mit den Kanten [(0,1),(1,2),(2,0),(3,4),(4,3)] für m=2. Am Ende wird überprüft, ob genau m falsche Kanten vorhanden sind.

Im testgraphwithcycles werden die ersten beiden Tests kombiniert. Es wird ein sortierbarer Graph erzeugt, welchem dann die Zyklen hinzugefügt werden, indem einfach die Kanten der beiden Graphen erst konkateniert und dann gemischt werden. Am Ende wird hier überprüft, ob es mindestens m falschen Kanten gibt.

Im testgraph2 werden vier sortierbare Graphen erzeugt, welche dann kombiniert werden, indem einfach die Kanten der beiden Graphen erst konkateniert und dann gemischt werden. Der resultierende Graph ist dabei nicht zwangsläufig topologisch sortierbar. Dieser Test testet unter anderem, was passiert, wenn Kanten mehrmals hinzugefügt werden.

Schlusswort

Insgesamt fand ich das Projekt interessant. Einerseits habe ich etwas über topologische Sortierung gelernt und andererseits habe ich viel über C++ gelernt. Ich

fand C++ eher unintuitiv, vor allem bei iteratoren. Allerdings ist das Endergebnis schnell. 😊

[Pearce & Kelly 2010] David J. Pearce & Paul H. J. Kelly: A Batch Algorithm for Maintaining a Topological Order, Proceedings 33rd Australasian Computer Science Conference (ACSC 2010), Brisbane, Australia, CRPIT Volume 102 - Computer Science 2010.