Optimierung von MST-Algorithmen

R. Azimi, F. Gläser, A. Stüben

3. September 2015

Ziele

Ziele

- Implementierung der folgenden Algorithmen:
  - Kruskal
    - Standard-Kruskal
    - Filter-Kruskal (Plus)
    - qKruskal
    - iQs
  - Prim
    - Binaryheap
    - Quickheap

Ziele

- Entwicklung einer Testumgebung, welche die Laufzeiten der Algorithmen testet.
  - Einlesen von Graphen.
  - Generieren von zufälligen Graphen.

Generierung zufälliger Graphen

Schwere und leichte Graphen

- Unterscheidung in zwei Arten von Graphen:
  1. Leichte Graphen mit einer Dichte $< 50\ \%$
  2. Dichte Graphen mit einer Dichte $>= 50\ \%$
- Erstellen der jeweiligen Graphen auf verschiedene Weise.
- Die Dichte errechnet sich aus der Anzahl der gegebenen Kanten durch die Zahl der maximal möglichen Kanten:

$$\frac{m \cdot 2}{n \cdot (n-1)}$$

n:= Anzahl der Knoten
m:= Anzahl der Kanten

Generierung eines leichten Graphen

- Erstellen eines aufspannenden Baumes mit zufälligen, gleichverteilten Kantengewichten zwischen 0 und 1.
  - Garantiert einen zusammenhängenden Graphen.
- Wiederholtes Wählen von zufälligen End- und Anfangsknoten und Gewicht. Einfügen dieser neue Kante in vorhandenen Kantenmenge, wenn nicht vorhanden.
- Da leichter Graph: Wahrscheinlichkeit, dass versucht wird eine schon vorhandene Kante einzufügen: $<= 50\%$.

Generierung eines schweren Graphen

- Erstellen eines vollen Graphen mit allen möglichen Kanten.
- Wiederholtes Löschen von zufälligen Kanten, wenn vorhanden.
- Da schwerer Graph: Wahrscheinlichkeit, dass versucht wird eine schon gelöschte Kante zu löschen: $<= 50\%$.
- Nachträglich wird getestet, ob resultierender Graph zusammenhängend ist.

Laufzeittest der Algorithmen

Laufzeittest der Algorithmen

- Abhängig durch Eingabeparameter:
  - Welche Algorithmen werden getestet.
  - Lade Graphen aus Datei.
  - Generiere zufällige(n) Graphen.
- Beispielsweise:
  - -a nK fK -t testresults.txt -o 2 100 300 100 0.1 0.3 0.01 0
  - -a all -t testresults.txt -e -o 2 100 300 100 100 1000 100 0
  - -a all -t testresults.txt -p graph.txt

Wiederholung: Implementierungen von Kruskal

Kruskal

1: Kruskal($G = (V, E), w$)
2: E_{sort} =Sortiere Kanten in E nach Gewichten.
3: $T = \emptyset$
4: **for each** $(e = (u, v) \in E_{sort})$
5: **if** $((V, T \cup e)$ ist kreisfrei)
6: Setze $T = T \cup e$;
7: **return** T

QuickKruskal

1: QuickKruskal(E, T, P)
2: **if** $m \leq kruskalThreshold(n, |E|, |T|)$
3: $Kruskal(E, T, P)$
4: **else**
5: Wähle ein Pivot $p \in E$
6: $E_{\leq} := \{e \in E : w(e) \leq p\}$
7: $E_{>} := \{e \in E : w(e) > p\}$
8: QuickKruskal($E_{\leq}, T, P$)
9: QuickKruskal($E_{>}, T, P$)

FilterKruskal

1: FilterKruskal(E, T, P)
2: **if** $m \leq$ kruskalThreshold$(n, |E|, |T|)$ **then**
3: Kruskal(E, T, P)
4: **else**
5: Wähle ein Pivot $p \in E$
6: $E_{\leq} := \{e \in E : w(e) \leq p\}$
7: $E_{>} := \{e \in E : w(e) > p\}$
8: FilterKruskal$(E_{\leq}, T, P)$
9: $E_{>} =$ **Filter**$(E_{>}, P)$
10: FilterKruskal$(E_{>}, T, P)$

1: Filter(E, P)
return $\{\{u, v\} \in E : u, v$ sind in versch. Komponenten von P $\}$

Incremental Quicksort

```
1: IQS(Set A, Index idx, Stack S)
2:     if idx = S.top() then S.pop(), return A[idx]
3:     pidx ← random[idx, S.top() − 1]
4:     pidx′ ←  partition(A, A[pidx], idx, S.top() − 1)
5:     S.push(pidx′)
6:     return IQS(A, idx, S)
```

Ergebnisse: Kruskal

Kruskal: Struktur

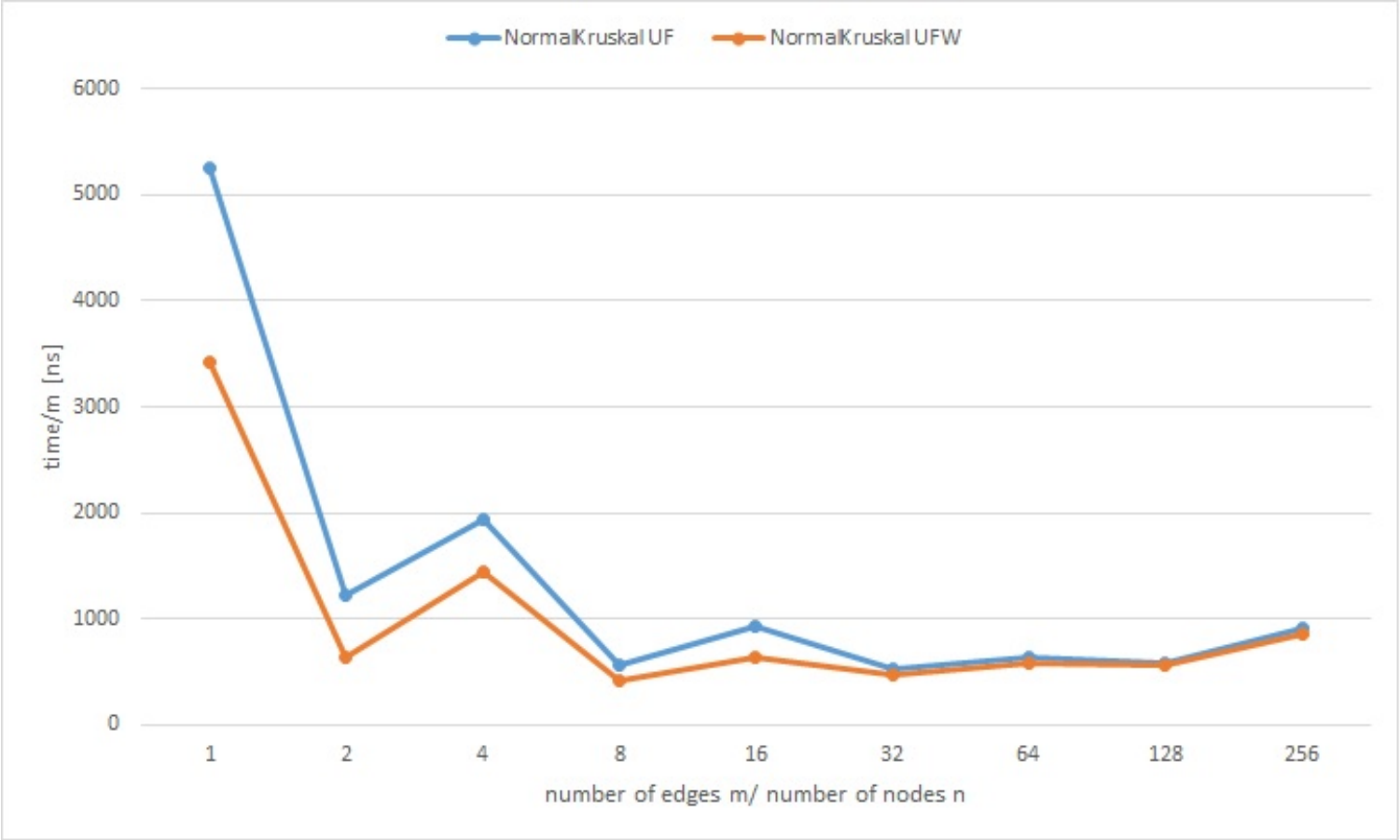
1. Laufzeit- & Speicherplatzoptimierung
2. Zusammenfassung: Vergleichsergebnisse
3. Vergleich von *Filter-*, *Filter-Plus-* und *qKruskal*
4. Vergleich zum Paper

Vergleiche verschiedener Implementierungen

Kruskal-Optimierung	
Kantenspeicherung	Listen vs. Arrays
Union-Find-Datenstruktur	Ohne vs. mit Pfadkompression
Sortierung	JavaSort vs. BH vs. IQS vs. DualPivotQS
Filter-Kruskal-Optimierung	
Programmierung	Rekursion vs. Iteration
Partitionsmethode	Lomuto vs. Hoare
Pivotwahl	Uniform zufällig vs. Median of Three

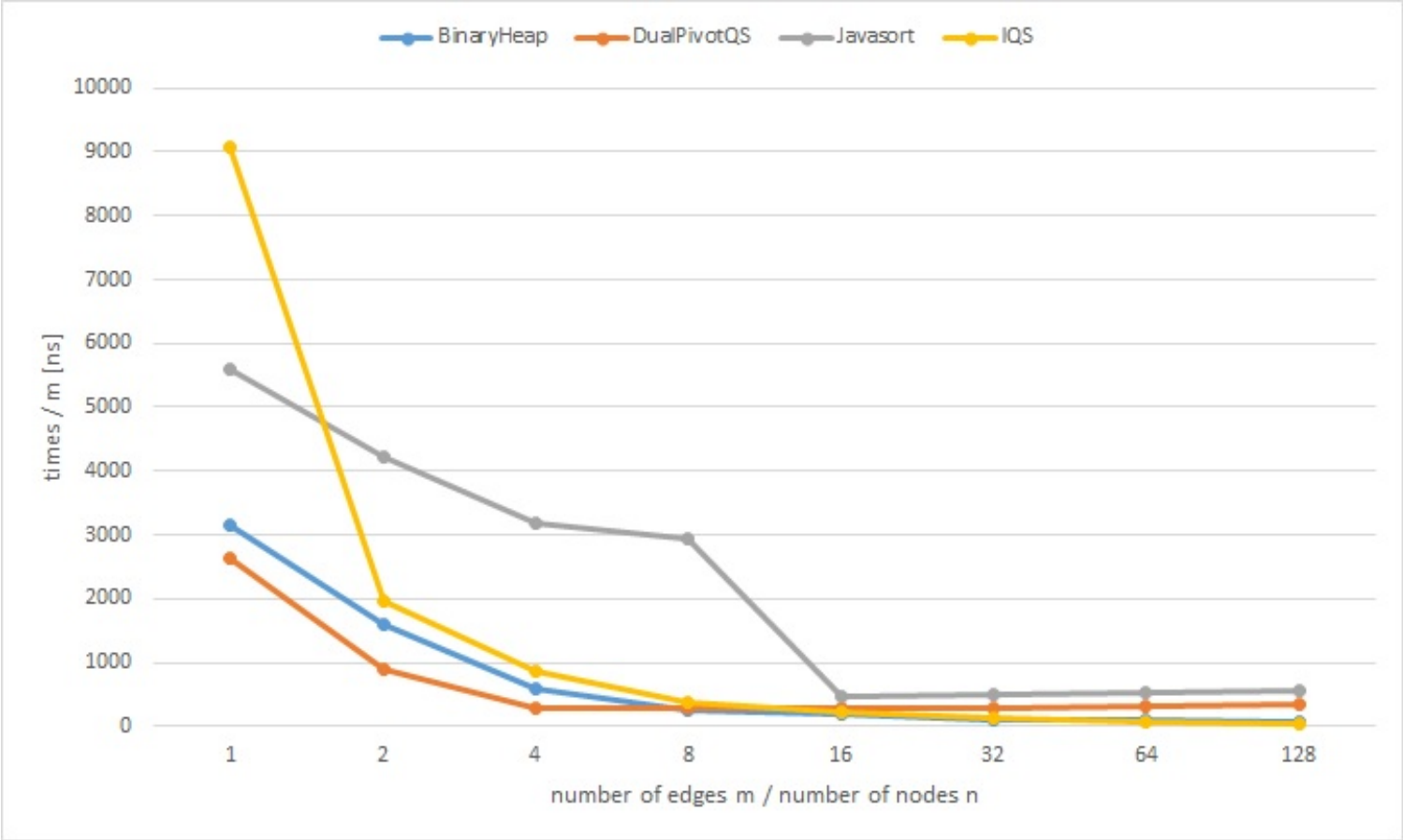
Union-Find-Datenstrukturen-Vergleich

UF ohne Pfadkompression vs. mit Pfadkompression



Vergleich Sortierungsmethoden

JavaSort vs. *DualPivotQuickSort* vs. *BinaryHeap* vs. *IQS*

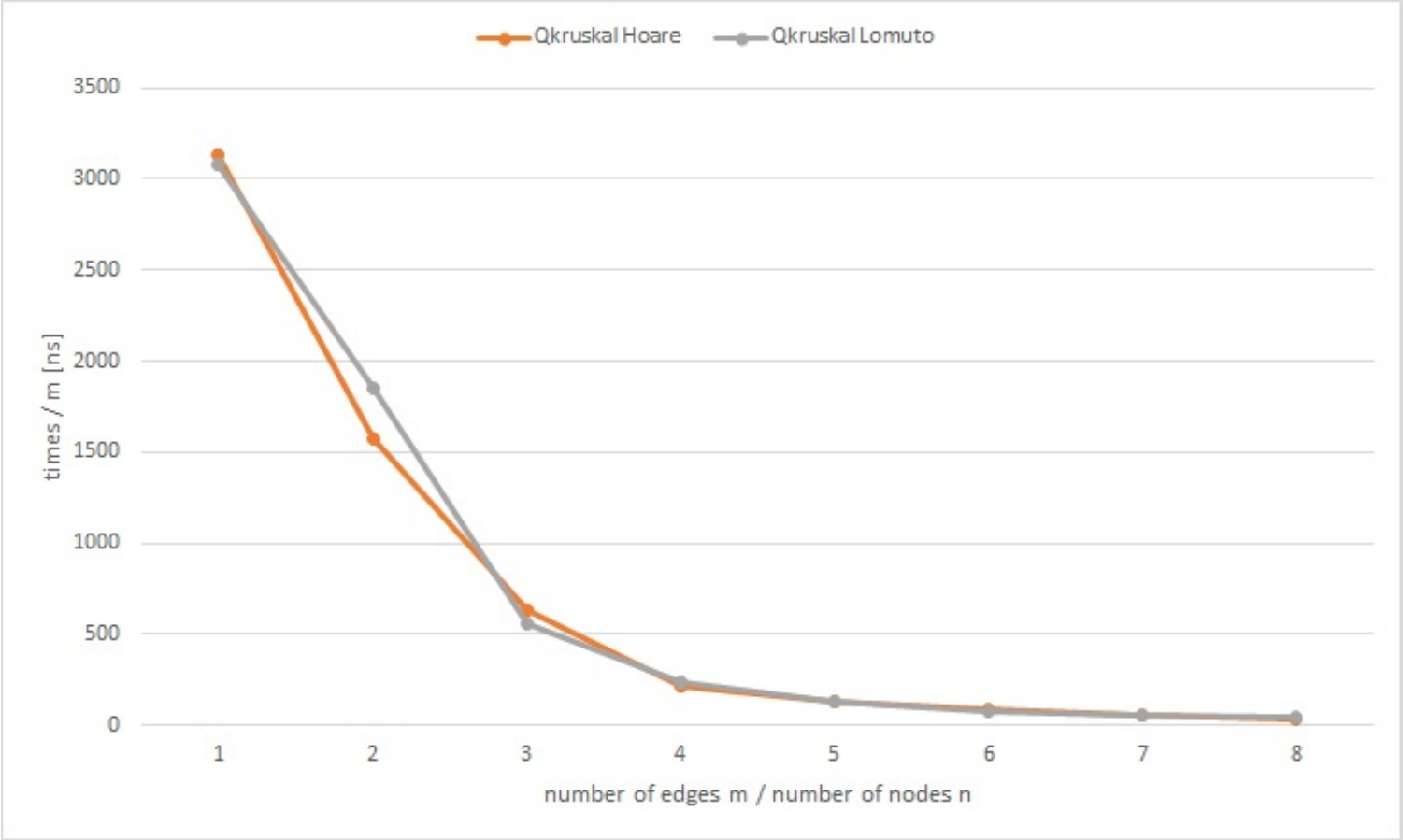


Rekursion auf Iteration

- Rekursion verursacht einen StackOverFlow bei großen Instanzen
- Iterative Implementierung:
  Mittels Speicherung der Indizes für die Partitionierung auf einem Stack im Heap

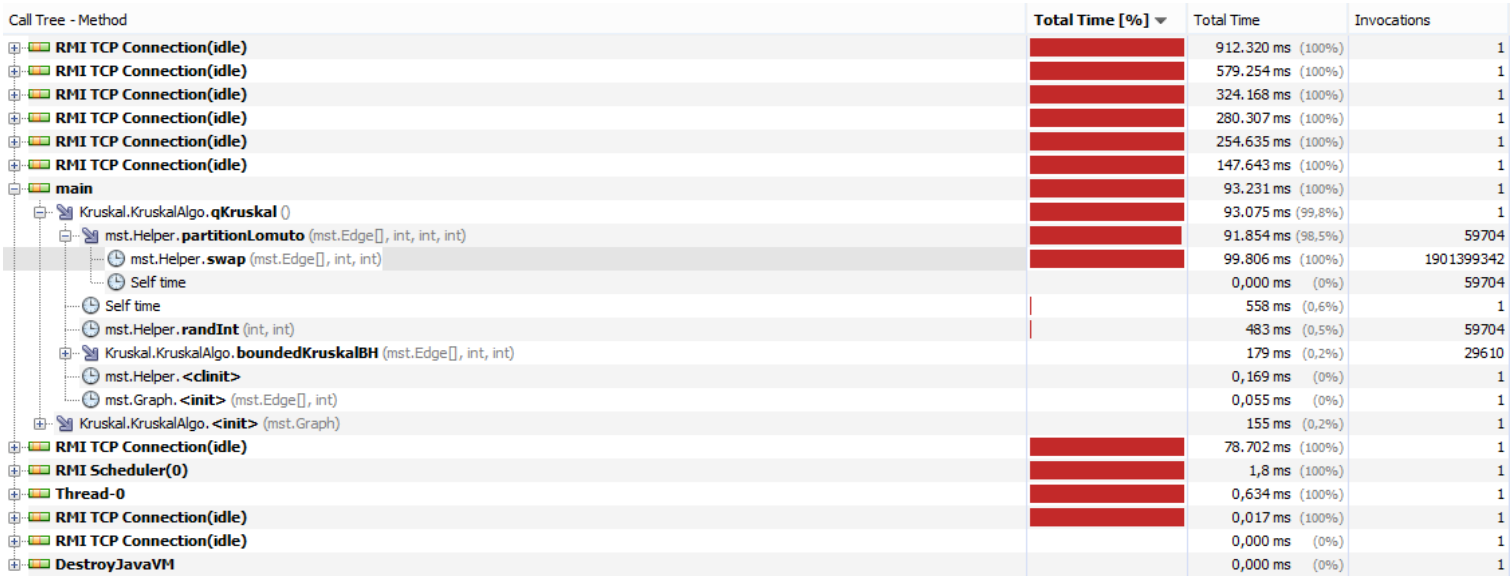
Vergleich Partitionsmethoden

Hoare-Partition vs. *Lomuto-Partition*



Lomotu-Partition im Profiler

Anzahl des Swap-Aufrufe in Lomuto: Sehr hoch.



Call Tree - Method	Total Time [%]	Total Time	Invocations
RMI TCP Connection(idle)		912.320 ms (100%)	1
RMI TCP Connection(idle)		579.254 ms (100%)	1
RMI TCP Connection(idle)		324.168 ms (100%)	1
RMI TCP Connection(idle)		280.307 ms (100%)	1
RMI TCP Connection(idle)		254.635 ms (100%)	1
RMI TCP Connection(idle)		147.643 ms (100%)	1
main		93.231 ms (100%)	1
Kruskal.KruskalAlgo.**qKruskal** ()		93.075 ms (99,8%)	1
mst.Helper.**partitionLomuto** (mst.Edge[], int, int, int)		91.854 ms (98,5%)	59704
mst.Helper.**swap** (mst.Edge[], int, int)		99.806 ms (100%)	1901399342
Self time		0,000 ms (0%)	59704
Self time		558 ms (0,6%)	1
mst.Helper.**randInt** (int, int)		483 ms (0,5%)	59704
Kruskal.KruskalAlgo.**boundedKruskalBH** (mst.Edge[], int, int)		179 ms (0,2%)	29610
mst.Helper.**<clinit>**		0,169 ms (0%)	1
mst.Graph.**<init>** (mst.Edge[], int)		0,055 ms (0%)	1
Kruskal.KruskalAlgo.**<init>** (mst.Graph)		155 ms (0,2%)	1
RMI TCP Connection(idle)		78.702 ms (100%)	1
RMI Scheduler(0)		1,8 ms (100%)	1
Thread-0		0,634 ms (100%)	1
RMI TCP Connection(idle)		0,017 ms (100%)	1
RMI TCP Connection(idle)		0,000 ms (0%)	1
DestroyJavaVM		0,000 ms (0%)	1

Hoare-Partition im Profiler

Anzahl des Swap-Aufrufe in Hoare: Stark gesungen relativ zu Lomuto.

Call Tree - Method	Total Time [%]	Total Time	Invocations
main		1.273 ms (100%)	1
Kruskal.KruskalAlgo.**qKruskal** ()		1.145 ms (89,9%)	1
mst.Helper.**partitionHoare** (mst.Edge[], int, int, int)		1.167 ms (91,7%)	23
Self time		1.179 ms (92,6%)	23
mst.Helper.**swap** (mst.Edge[], int, int)		0,000 ms (0%)	4595691
Self time		3,35 ms (0,3%)	1
mst.Helper.**randInt** (int, int)		0,250 ms (0%)	23
mst.Helper.**<clinit>**		0,167 ms (0%)	1
mst.Graph.**<init>** (mst.Edge[], int)		0,039 ms (0%)	1
Kruskal.KruskalAlgo.**boundedKruskalBH** (mst.Edge[], int, int)		0,000 ms (0%)	13
Kruskal.KruskalAlgo.**<init>** (mst.Graph)		128 ms (10,1%)	1
Thread-0		0,879 ms (100%)	1
RMI TCP Connection(idle)		0,068 ms (100%)	1
DestroyJavaVM		0,000 ms (0%)	1

Es gilt sogar:

Theorem

Die Anzahl der Swapaufrufe in Hoare sind im Erwartungswert um ein Faktor von drei weniger als die in Lomuto.

Zusammenfassung: Vergleichsergebnisse

Kruskal-Optimierung	
Kantenspeicherung	Listen vs. Arrays
Union-Find-Datenstruktur	Ohne vs. mit Pfadkompression
Sortierung	JavaSort vs. BH vs. IQS vs. DualPivotQS
Filter-Kruskal-Optimierung	
Programmierung	Rekursion vs. Iteration
Partitionsmethode	Lomuto vs. Hoare
Pivotwahl	Uniform zufällig vs. Median of Three

QKruskal versus Standard-Kruskal

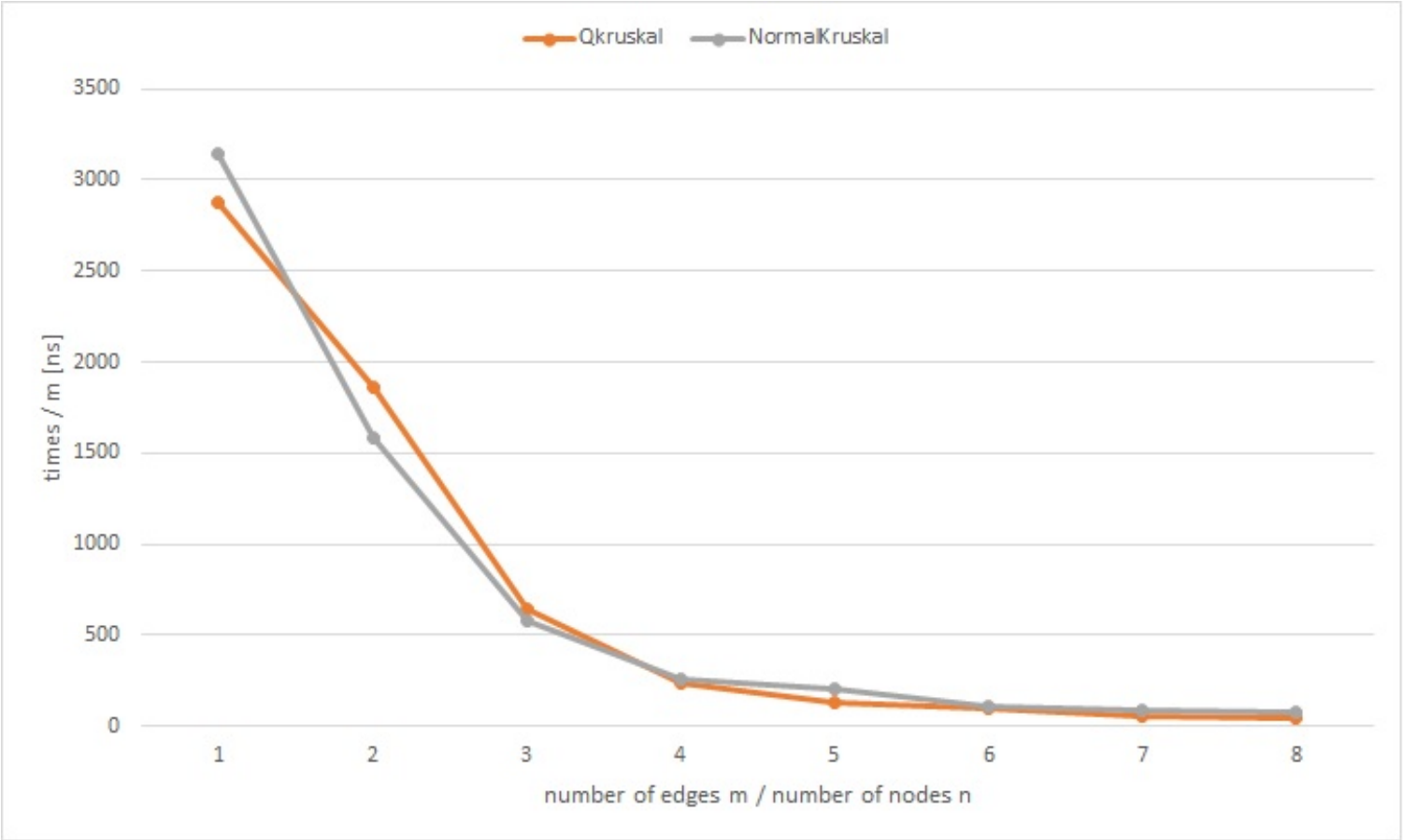


Abbildung: QKruskal und normaler Kruskal

Finaler Vergleich: Kruskal-Varianten

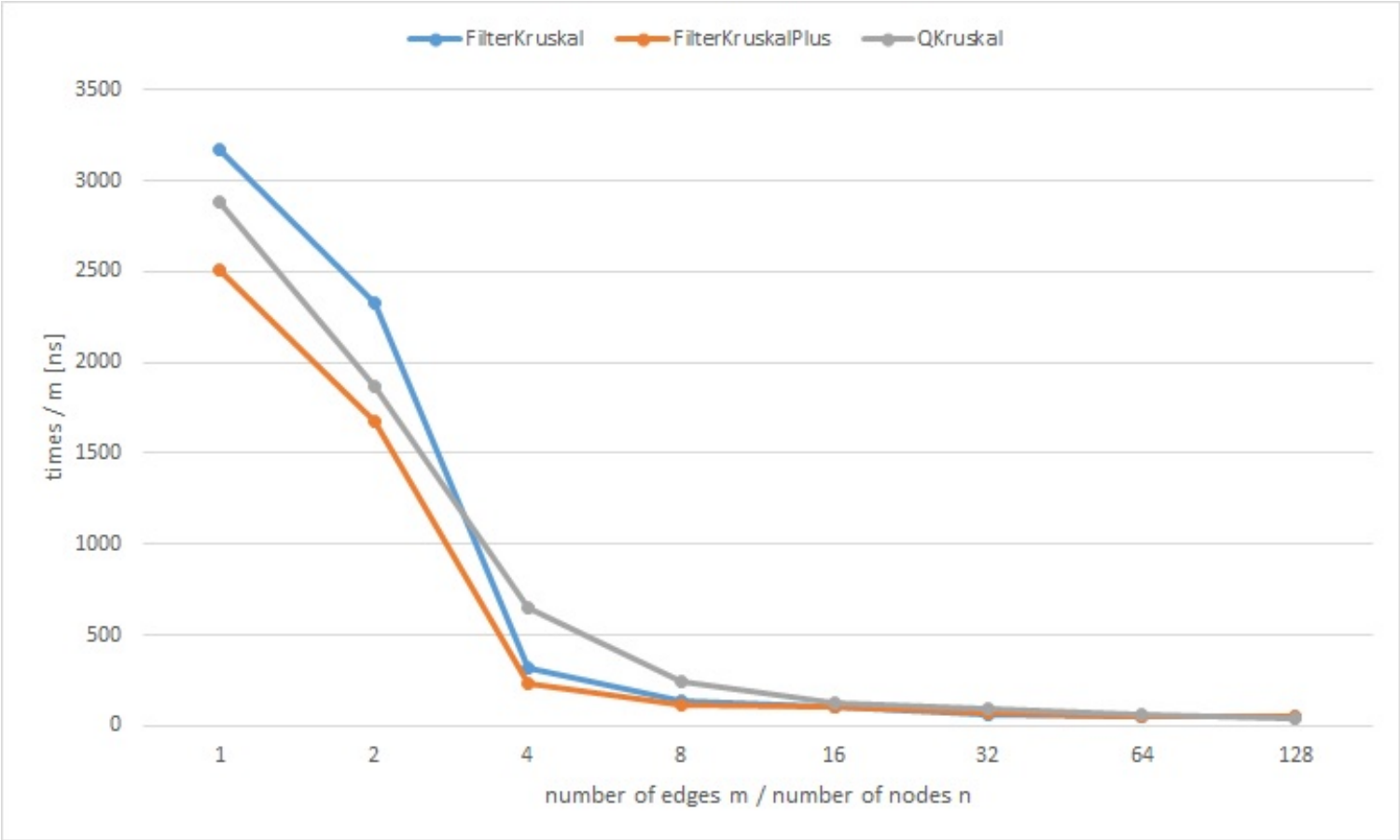


Abbildung: QKruskal, Filter-Kruskal und Filter-Kruskal-Plus

Vergleich zu Ergebnissen im Paper

- Geringfügige Unterschiede zwischen Kruskal, qKruskal, Filter-Kruskal und Filter-Kruskal-Plus
- Filter-Kruskal-Plus schneller als Filter-Kruskal

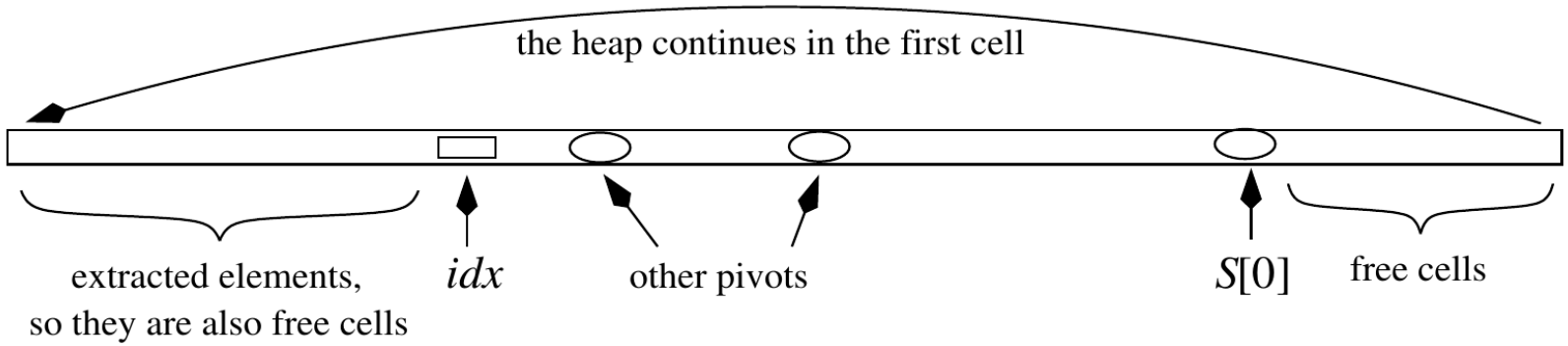
Wiederholung: Implementierungen von Prim

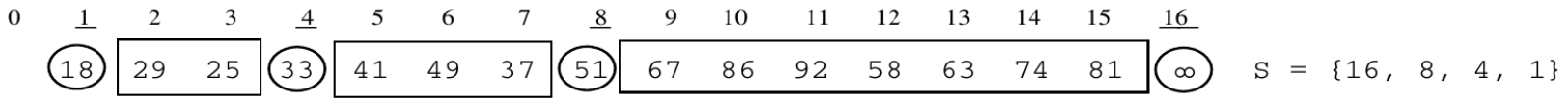
Prim

1: Prim($G = (V, E), w$)
2: Wähle einen Knoten $v_0 \in V$
3: Setze $V_T = \{v_0\}$ und $T = \emptyset$
4: **for** $(i = 1, \ldots |V| - 1)$
5: Bestimme eine gewichtsminimale Kante $e_i = \{u_i, v_i\}$ mit
6: $u_i \in V_T$ und $v_i \notin V_T$.
7: Setze $V_T = V_T \cup \{v_i\}$ und $T = T \cup \{e_i\}$.
8: **return** T

Wiederholung: Quickheaps

Quickheaps


the heap continues in the first cell
extracted elements,
so they are also free cells
idx
other pivots
S[0]
free cells


0 1 2 3 4 5 6 7 8 9 10 11 12 13 14 15 16
18 29 25 33 41 49 37 51 67 86 92 58 63 74 81 ∞
S = {16, 8, 4, 1}

Ergebnisse: Prim

aktueller Stand bei Prim

- Prim mit Quickheaps vor allem bei großen Instanzen schneller als Prim mit Binaryheaps
- Laufzeit hängt stark von der Dictionary Datenstruktur von Quickheap und Binaryheap ab
- Da bei Prim nur $|V|$ viele Elemente betrachtet werden, wird ein normales Array als Dictionary verwendet
- Bei IncrementalQuicksort ist iteratives Partitionieren und eine geschickte Wahl der Pivotelemente erforderlich

Fazit

Ausblick: "Was noch zu tun ist."

- Optimierung der Dictionary Datenstruktur für schnellen Zugriff auf die Elemente des Quickheaps
- Refactoring.
- Optimierung der Algorithmen.
- Testen und Präsentation der Ergebnisse im Arbeitsbericht.

Vielen Dank für Ihre Aufmerksamkeit!

Literaturverzeichnis

Gonzalo Navarro and Rodrigo Paredes.
On sorting, heaps, and minimum spanning trees.
Algorithmica, 57(4):585–620, 2010.

Vitaly Osipov, Peter Sanders, and Johannes Singler.
The filter-kruskal minimum spanning tree algorithm.
In *10th Workshop on Algorithm Engineering and Experiments (ALENEX)*, pages 52–61, 2009.