

# Praktikum 2

Roland Kluge, 1780358

Effiziente Graphenalgorithmen — WS 2011/2012

## 1 Testdaten

Die Tests, welche dieser Auswertung zugrunde liegen, können mit dem Skript *test.sh* nachvollzogen werden. Die Daten, auf deren Basis argumentiert wird, befinden sich in der Datei *test\_results.txt* oder in aufbereiteter Form im OpenDocument-Spreadsheet *test\_results.ods*. Sie wurden mit der aktuellen Version der Implementierung mittels der Befehlssequenz `qmake spp_cpp.pro; make release` erstellt und über 10 Testläufe gemittelt. Die Varianten werden von nun an stets so benannt, wie es auch in der Ausgabe zu sehen ist.

Die öffentliche Testdatendatei *FLA.sp* wurde so verändert, dass ein weiterer Knoten eingefügt wurde, der keine Kanten besitzt:

```
v 1070376 82544500 30998800
```

Dadurch ändert sich natürlich die Kopfzeile:

```
g n 1070377 m 2712798
```

## 2 Übersicht

Um dies nicht in jedem weiteren Abschnitt zu erwähnen, sei hier angemerkt, dass es in bestimmten Grenzen natürlich stets auf die Implementierung ankommt, ob eine bestimmte Variante schneller ist als eine andere. Bei den weiteren Aussagen wird dies ignoriert und davon ausgegangen, dass die vorliegende Implementierung optimal ist.

Die Ergebnisse der 13 relevanten Testfälle ist in Tabelle 1 zu sehen. Testfall 13 wurde ausgelassen, da es sich hierbei nur um einen Grenzfall für einen unverbundenen Graphen mit 2 Knoten handelt.

Var. \ Test Nr.	1	2	3	4	5	6	7	8	9	10	11	13
Heap	10	80	0	40	50	60	50	80	40	50	80	270
Heap, Bi	20	70	10	50	30	60	50	60	50	70	60	160
Heap, A*	10	230	0	70	80	190	60	170	70	110	180	280
Dial	10	60	0	40	40	50	40	50	40	50	60	250
Dial, Bi	40	70	10	40	30	60	30	60	50	50	50	190
Dial, A*	510	220	110	140	140	190	130	180	130	160	190	650
schnellster [ms]	10	60	0	40	30	50	30	50	40	50	50	160
langsamster [ms]	510	230	110	140	140	190	130	180	130	160	190	650
Heap ./ Dial	=	Dial	=	=	=	Dial	Dial	Dial	=	=	Dial	Heap
Abstand	0,00%	116,67%	0,00%	0,00%	0,00%	120,00%	166,67%	120,00%	0,00%	0,00%	120,00%	118,75%

Abbildung 1: Übersicht über die Ergebnisse. Grün - beste Laufzeit, Dunkelrot - schlechteste Laufzeit, Pink - zweitschlechteste Laufzeit

### 3 Dial's Implementierung

Die Implementierung mit einer einfachen Bucket Queue hat sich als recht effizient erwiesen. In 8 von 13 Tests (entspricht 62%) war *Dial* am schnellsten, in 10 von 13 Fällen (entspricht 77%) war diese Variante unter den besten beiden Laufzeiten.

### 4 Vergleich zwischen Binärem Heap und Bucket Queue

Setzt man die Implementierungen mit binärem Heap mit der entsprechenden Implementierung mittels Bucket Queue in Beziehung, so erhält man folgendes Ergebnis, wenn man jeweils das beste Ergebnis der Gruppe betrachtet:

In 6 von 13 Fällen (entspricht 46%) war die Bucket Queue dem Heap eindeutig überlegen, in 6 von 13 Fällen (entspricht 46%) waren beide Gruppen gleich stark und in 1 von 13 Fällen (entspricht 8%) war der Heap eindeutig überlegen (Zeile „Heap./.Dial“ in der Tabelle).

Im ersten Fall (BQ vor Heap) war die beste Heap-Implementierung jeweils 16,7%, 20,0%, 66,7%, 20,0% und 10,0% langsamer als die beste Bucket Queue-Implementierung, im umgekehrten Fall erhält man eine Laufzeitdifferenz von 18,8% (Zeile „Abstand“ in der Tabelle).

### 5 Zielgerichtete Suche

Beim ersten Betrachten der Ergebnisse fällt auf, dass *Dial*,  $A^*$  in 11 von 13 Testläufen (85%) die größte Laufzeit aufweist. Betrachtet man die zielgerichtete Suche ( $\cdot, A^*$ ) im Allgemeinen, dann ist eine der beiden Implementierungen in allen Testläufen am langsamsten.

Bei der Umsetzung von *Dial*,  $A^*$  trat unter anderem das Problem auf, dass zunächst einmal die Anzahl der Buckets bestimmt werden musste, wozu konservativ die Distanzen von allen Knoten zum Ziel berechnet wurden. Dies erzeugt gerade bei trivialen Grenzfällen einen enormen Overhead (siehe *Test 1*, hier liegt das Verhältnis von bester zu schlechtester Laufzeit bei 1:51), der in  $\Theta(n)$  liegt.

Verglichen mit den anderen 4 Varianten ist ein weiteres Problem, dass die Distanzberechnung auf einer Kugel eine verhältnismäßig aufwändige Operation ist.

### 6 Zufälliges Testen

Zusätzlich zur Testsuite *test.sh* wurde noch eine Reihe zufälliger Tests auf dem Graphen *COL.sp* durchgeführt. Der Befehl dafür lautete:

```
./Dijkstra ../../prog02/COL.sp -s 0 -t 50 -r
```

Die Ergebnisse dieses Testlaufs finden sich in der Datei *random\_test.txt*, wobei die Ergebnisse am Ende der Datei Mittelwerte über die Laufzeit und die Anzahl der PQ-Operationen enthalten.

Man sieht, dass im Mittel *heap*, *bidirectional* mit rund 62ms am schnellsten und *dial*,  $A^*$  und *heap*,  $A^*$  mit 291ms bzw. 148ms am langsamsten waren.

### 7 Fazit

Nach dieser kurzen Betrachtung können diese Schlussfolgerungen gezogen werden:

1. Für die gegebenen Testfälle haben sich die „nicht-beschleunigten“ Varianten (*Heap*, *Dial*) interessanterweise als die überlegenen gezeigt.
2. Tendenziell scheint *Dial* in seinen Varianten *Heap* überlegen zu sein, wobei in 50% der Testfälle eine Übereinstimmung der Laufzeiten festgestellt wurde.

3. Trotz des vielversprechenden heuristischen Ansatzes der zielgerichteten Suche, erwies sie sich in diesem Test als die zumeist langsamste Variante, was zum einen auf die komplizierte mathematische Berechnung der Potentiale und zum anderen - im Falle von *Dial*,  $A^*$  - auf die große Zahl leerer Buckets zurückzuführen ist.
4. Die *zufälligen Tests* haben den negativen Eindruck bezüglich der zielgerichteten Suche bestätigt, wohingegen in diesen Tests die schnellste Variante nicht die Bucket Queue- sondern die Heap-Implementierung war.

Gerade die große Anzahl leerer Buckets könnte reduziert werden, indem man bspw. Radix-Heaps einsetzt oder in einem Bucket nicht nur eine Distanz, sondern bspw. 10 Distanzen zulässt, wodurch natürlich eine Priorisierung innerhalb des Buckets nötig würde.