FDS2 - Übung 10

SS 2025

Tim Peko

Inhaltsverzeichnis

1. Beispiel 1: Mustersuchalgorithmen

1.1. Lösungsansatz

Die drei zu implementierenden Algorithmen sind:

- `brute_search`: Die naive, brute-force Suche.
- `bm_search`: Der Boyer-Moore-Algorithmus, der durch eine Heuristik (Bad-Character-Rule) in der Praxis oft sehr schnell ist.
- `kmp_search`: Der Knuth-Morris-Pratt-Algorithmus, der durch eine Vorverarbeitung des Musters (LPS-Array) unnötige Vergleiche vermeidet.

Jeder dieser Algorithmen wird als statische `search`-Methode in einer eigenen Klasse gekapselt. Diese Methode akzeptiert den zu durchsuchenden Text, das Muster, eine Startposition und einen `data_collector` zur Instrumentierung.

1.1.1. Instrumentierung mit `data_collector`

Ein zentraler Bestandteil der Aufgabe ist die Analyse der Algorithmen. Dazu wird die Klasse `data_collector` implementiert. Ihre Aufgaben sind:

- Zählen der gesamten Zeichenvergleiche während einer Suche.
- Aufzeichnen, wie oft jedes Zeichen im Text erfolgreich bzw. erfolglos verglichen wurde.
- Exportieren der gesammelten Daten in eine CSV-Datei für die externe Analyse und Visualisierung.

Die `search`-Methoden rufen Methoden des `data_collector` auf, um die Vergleiche zu protokollieren.

1.1.2. Projektstruktur

Die Lösung ist wie folgt strukturiert:

- `pattern_search.h/.cpp`: Header- und Implementierungsdateien für die Suchalgorithmen.
- `data_collector.h/.cpp`: Schnittstelle und Implementierung des Datensammlers.
- `example01.cpp`: Das Hauptprogramm, das die Experimente durchführt, die Testdaten generiert und die Algorithmen aufruft.

1.2. Analyseaufgaben

Die Analyse konzentriert sich auf das Laufzeitverhalten der Algorithmen in Abhängigkeit von der Beschaffenheit des Alphabets. Folgende Alphabete werden untersucht:

- **Binäres Alphabet**: `{0, 1}`
- **Alphabet der Basenpaare**: `{A, C, G, T}`
- **Alphabet der Aminosäuren**: 23 Symbole
- **ASCII-Zeichensatz**: 256 Symbole

Mithilfe eines Zufallszahlengenerators werden Texte und Muster verschiedener Längen erzeugt, um die in der Vorlesung besprochenen theoretischen Eigenschaften der Algorithmen (z.B. Best-Case, Worst-Case) experimentell zu überprüfen.

1.3. Experimentelle Ergebnisse

In diesem Abschnitt werden die Ergebnisse der durchgeführten Analysen dokumentiert. Die Rohdaten aus den CSV-Dateien werden mithilfe von Python mit Matplotlib visualisiert und hier dargestellt.

1.3.1. Vergleich der Vergleichsoperationen

Hier wird eine Grafik eingefügt, die die Anzahl der Zeichenvergleiche für jeden Algorithmus über verschiedene Text- und Musterlängen für ein bestimmtes Alphabet darstellt.

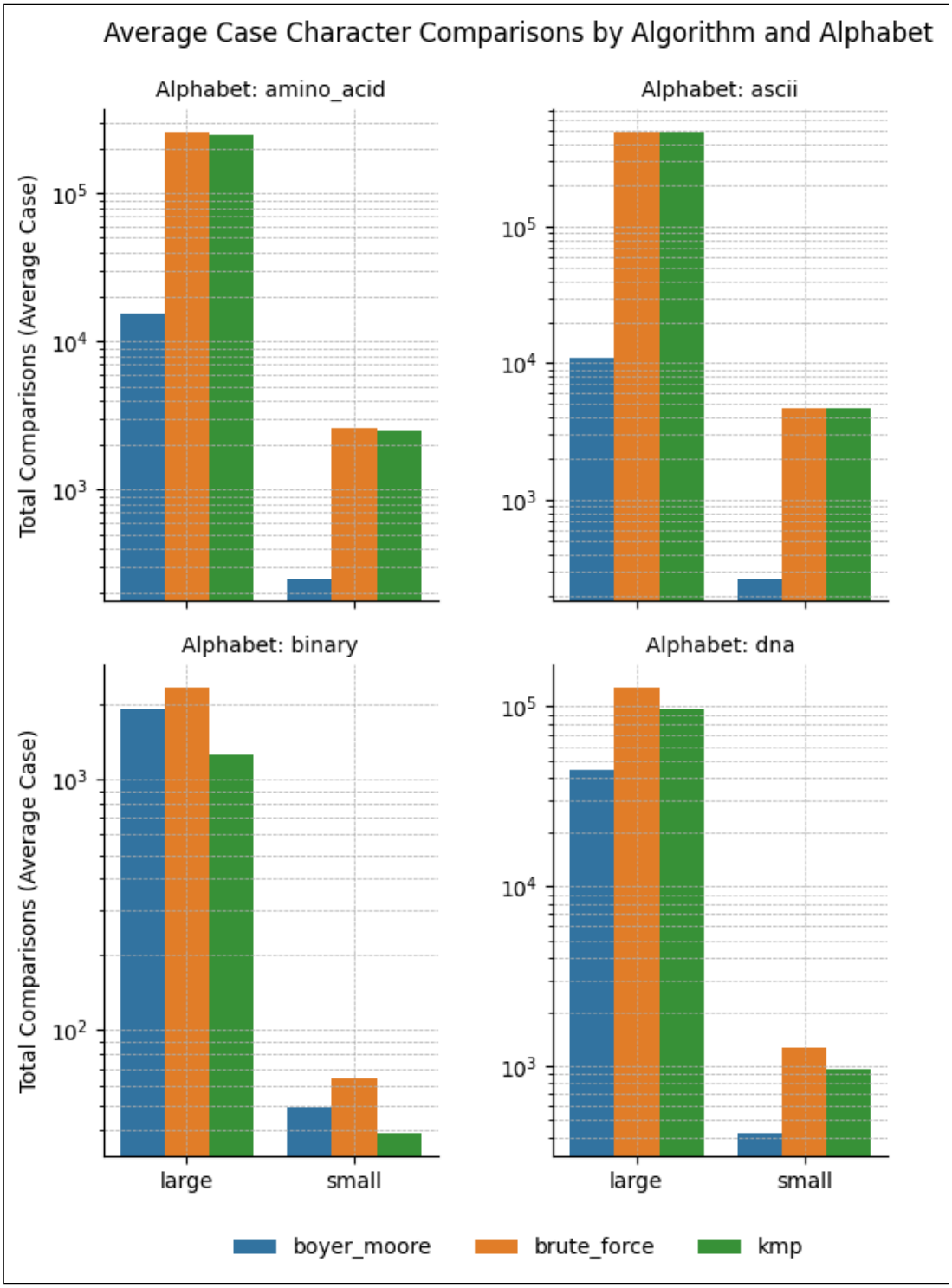

Abbildung 1: Anzahl der Vergleiche für alle Algorithmen und Alphabete

Analyse: Die Ergebnisse bestätigen die theoretischen Erwartungen. Der Brute-Force-Algorithmus zeigt durchgehend die höchste Anzahl an Vergleichen, insbesondere bei größeren Texten und kleineren Alphabeten. Boyer-Moore ist beinahe ausnahmslos signifikant effizienter als KMP. Der Unterschied wird mit steigender Alphabetgröße immer eindeutiger. KMP hat lediglich bei binären Alphabeten einen kleinen Vorteil.

1.3.2. Einfluss der Alphabetgröße

Hier wird eine Grafik eingefügt, die zeigt, wie sich die Größe des Alphabets auf die Effizienz der Algorithmen im Worst-Case-Szenario auswirkt.

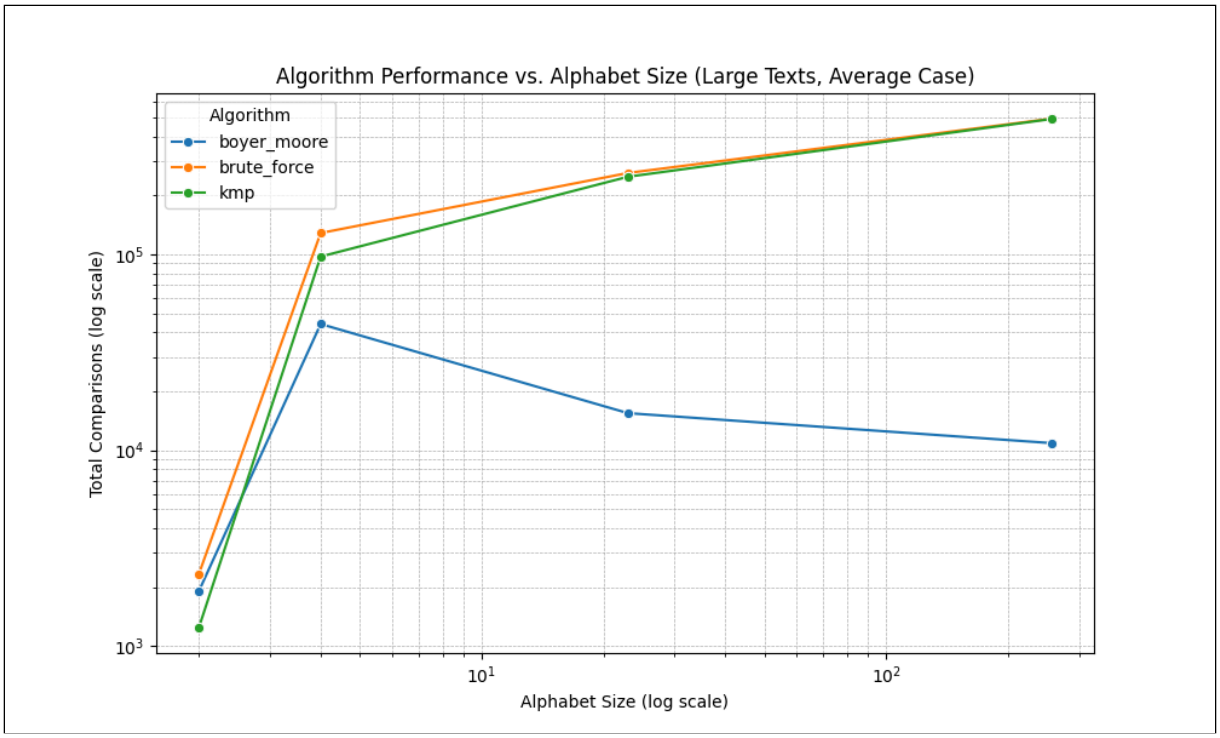

Abbildung 2: Effizienz in Abhängigkeit von der Alphabetgröße (Worst-Case, große Texte)

Analyse: Diese Grafik bestätigt, dass der Boyer-Moore-Algorithmus stark von einer größeren Alphabetgröße profitiert. Die Sprünge mittels der Bad-Character-Heuristik werden größer, was die Anzahl der Vergleiche drastisch reduziert. KMP zeigt eine relativ stabile Leistung, die weniger von der Alphabetgröße abhängt. Es scheint jedoch parallel zu Brute-Force zu verlaufen.

1.3.3. Heatmap der Zeichenvergleiche

Hier werden Heatmaps eingefügt, die visualisieren, welche Teile des Textes im Worst-Case-Szenario am häufigsten verglichen wurden. Die Visualisierung erfolgt mittels einer Hilbert-Kurve, um die Lokalität der Zugriffe in einer 2D-Darstellung zu erhalten.

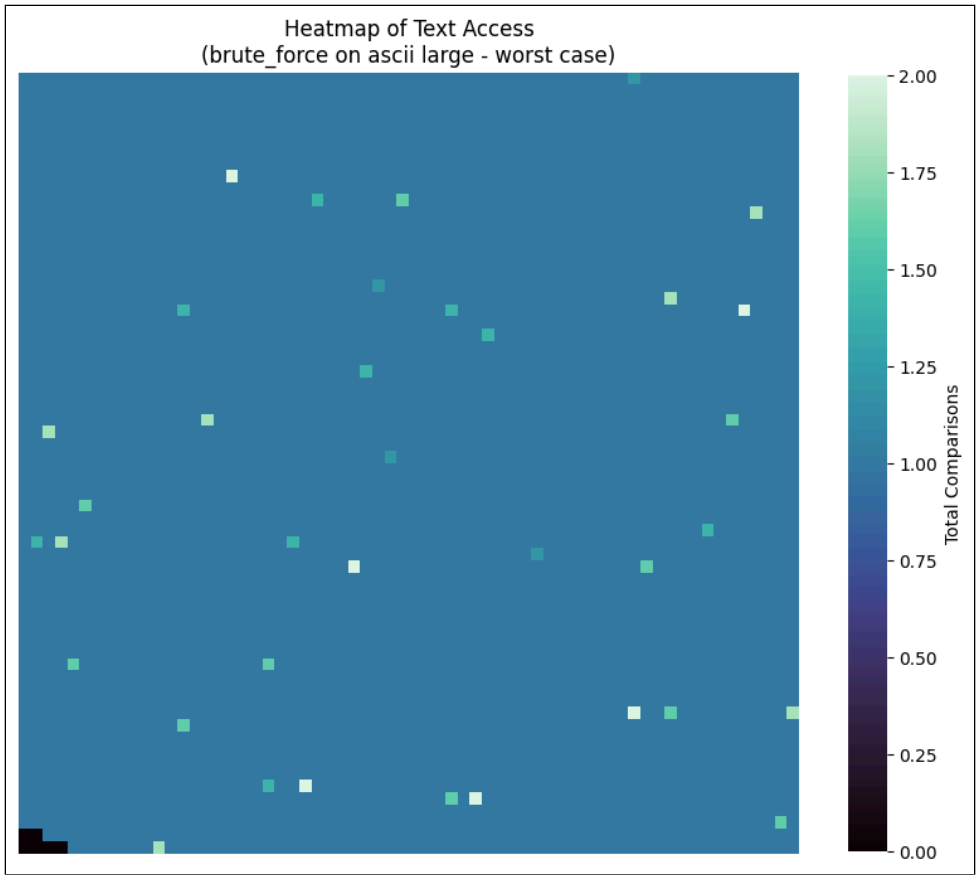

Abbildung 3: Heatmap der Textzugriffe für Brute-Force (ASCII, large, worst-case)

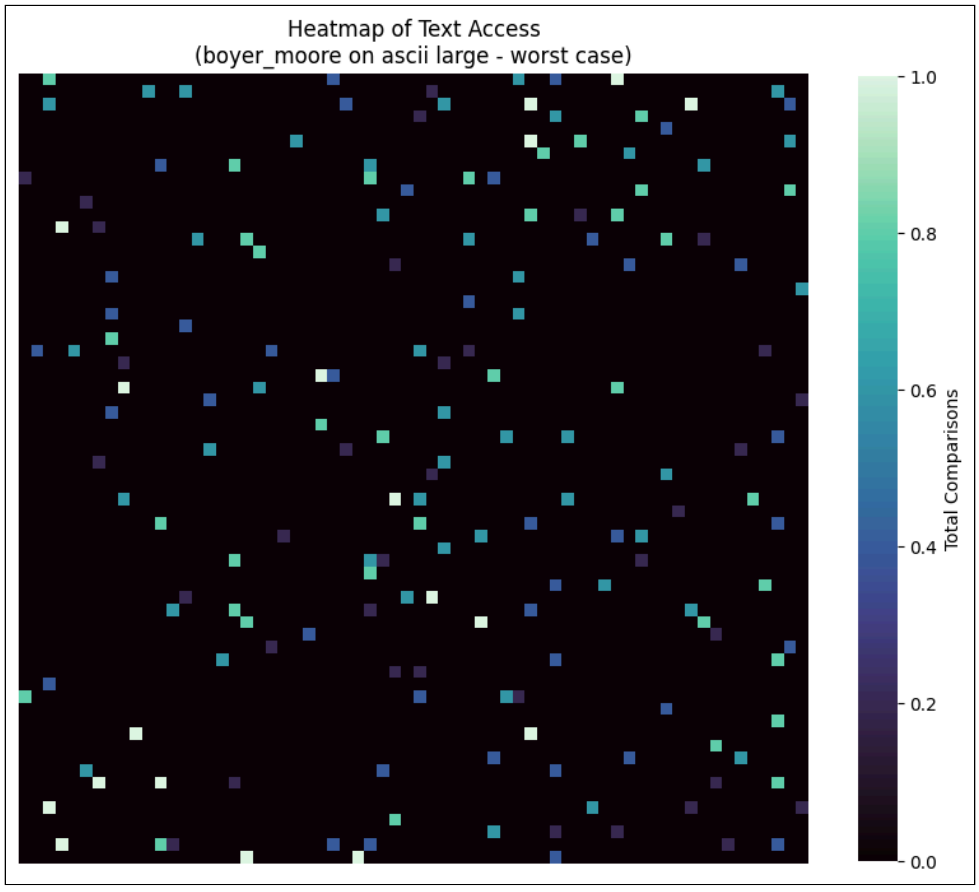

Abbildung 4: Heatmap der Textzugriffe für Boyer-Moore (ASCII, large, worst-case)

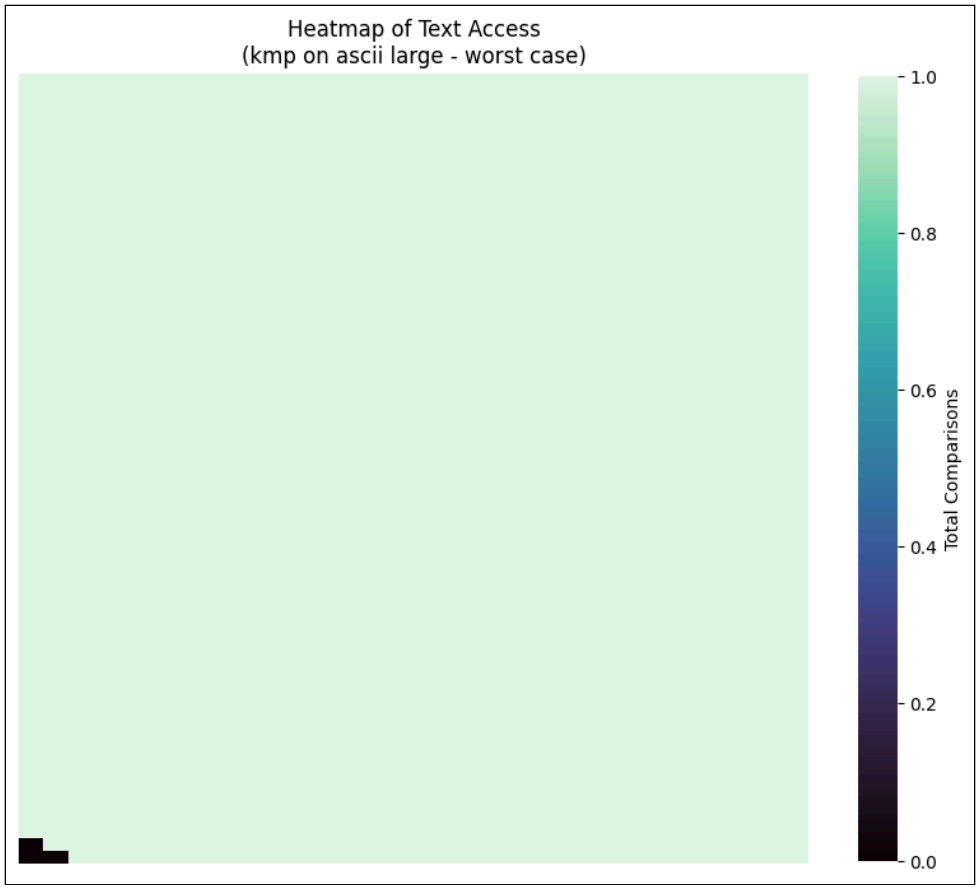

Abbildung 5: Heatmap der Textzugriffe für KMP (ASCII, large, worst-case)

Analyse: Der Vergleich der Heatmaps für den großen ASCII-Testfall im Worst-Case ist aufschlussreich. **Brute-Force** zeigt ein extrem dichtes durchgehendes Muster, was den ineffizienten, zeichenweisen Scan widerspiegelt. **Boyer-Moore** zeigt ein deutlich verbessertes Muster mit großen Lücken, was auf die weiten Sprünge der Bad-Character-Heuristik zurückzuführen ist. **KMP** weist ebenfalls ein gleichmäßiges komplett durchlaufendes Muster auf.

Der Vergleich der Heatmaps im Average-Case gibt einen guten Eindruck, wie die Algorithmen in der Praxis arbeiten.

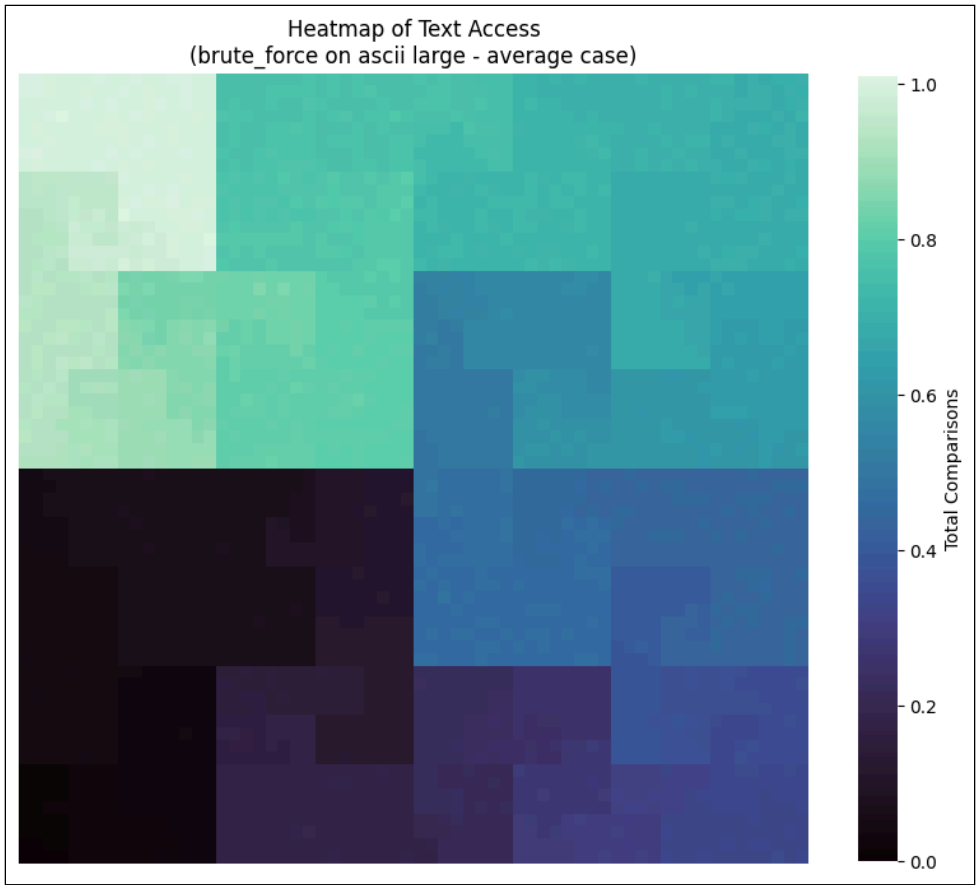

Abbildung 6: Heatmap der Textzugriffe für Brute-Force (ASCII, large, average-case)

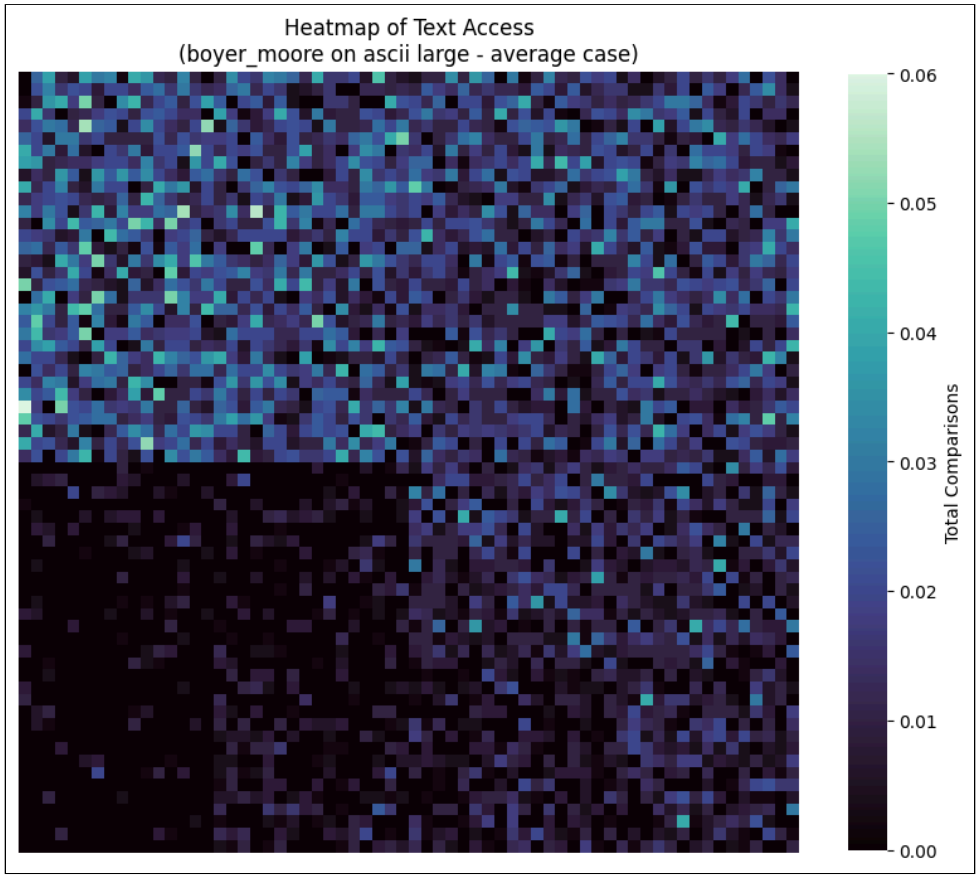

Abbildung 7: Heatmap der Textzugriffe für Boyer-Moore (ASCII, large, average-case)

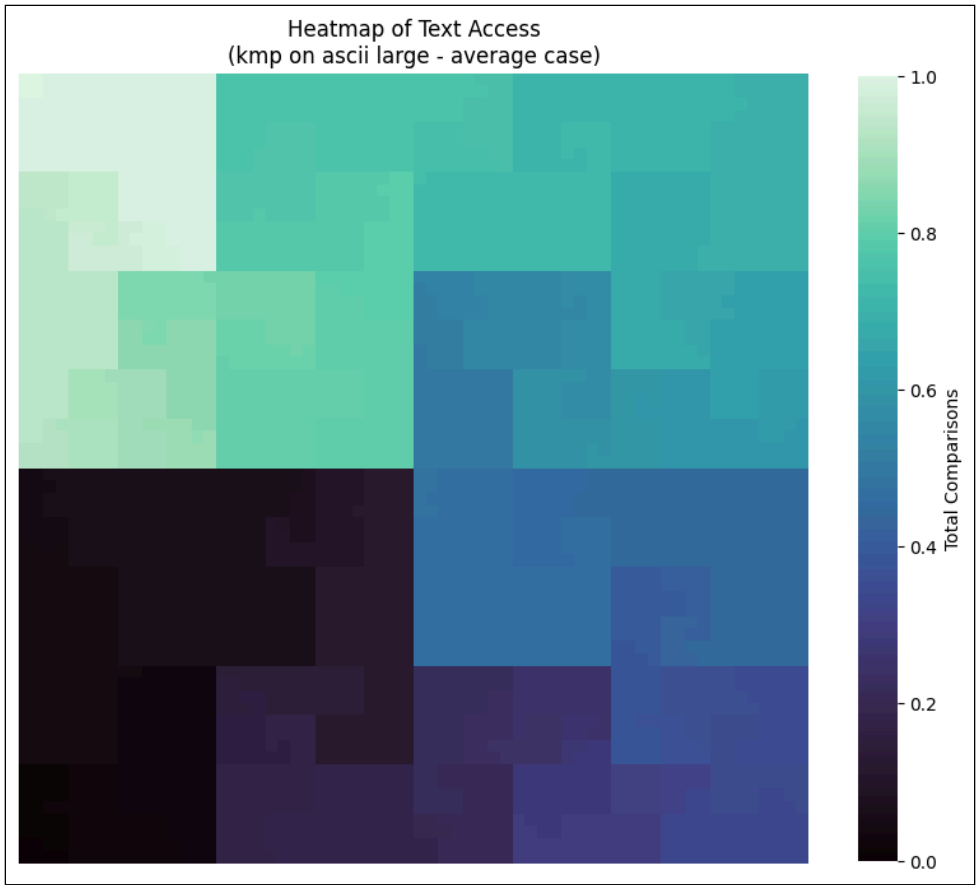

Abbildung 8: Heatmap der Textzugriffe für KMP (ASCII, large, average-case)

Man erkennt hier schön den abnehmenden Trend zu Ende des Textes. Da wir hier den Durschnittsfall analysieren und dieser per Zufall erzeugt wird, wird die Wahrscheinlichkeit, das Suchmuster noch nicht gefunden zu haben, gegen Ende des Textes immer geringer.

Der Vergleich im Average-Case unter einem binären Alphabet zeigt den Vorteil von **KMP** gegenüber den anderen Algorithmen.

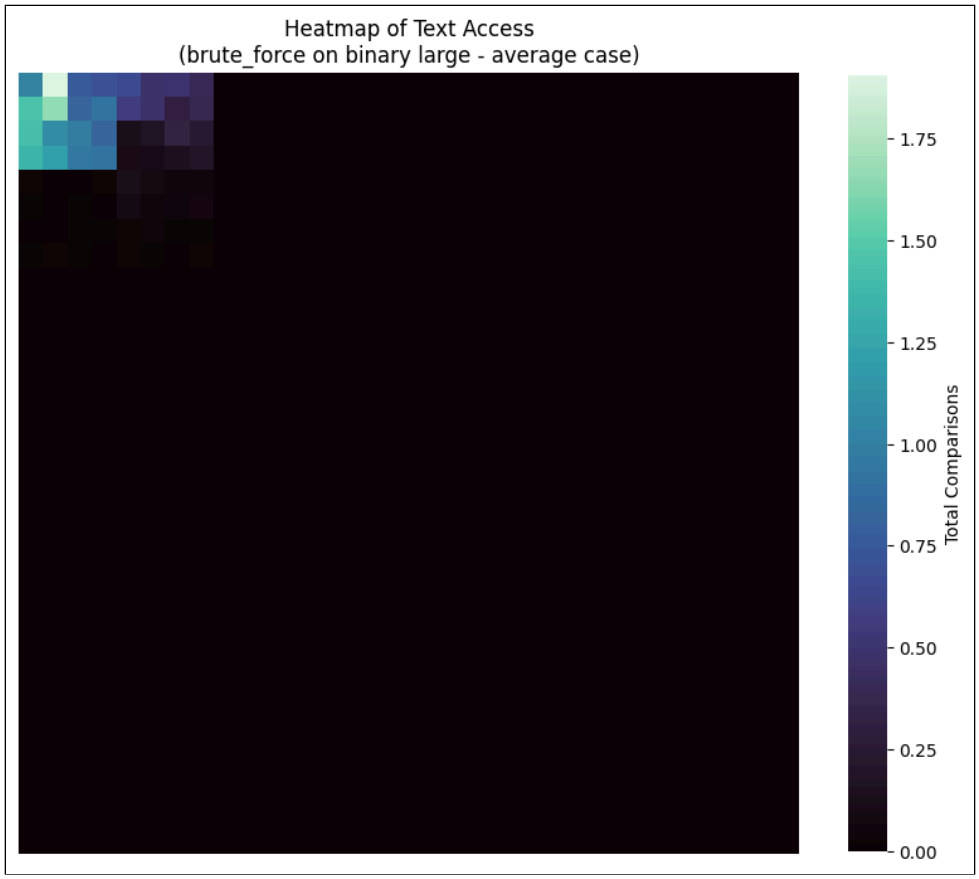

Abbildung 9: Heatmap der Textzugriffe für Brute-Force (binary, large, average-case)

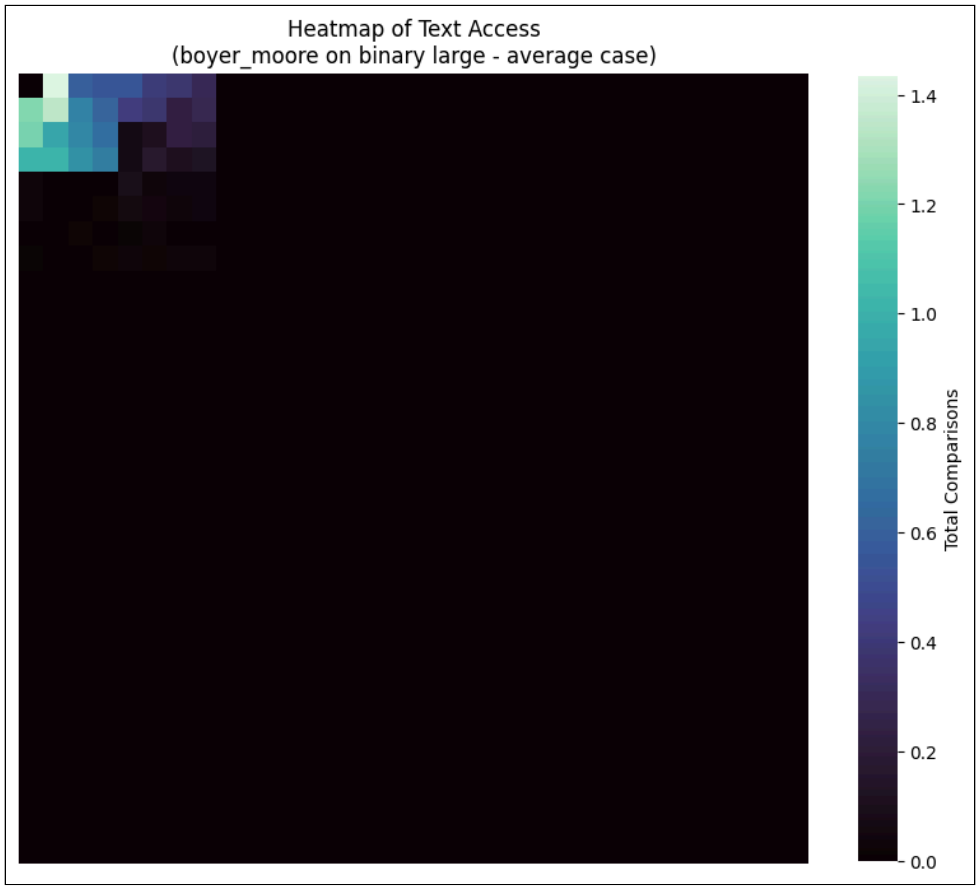

Abbildung 10: Heatmap der Textzugriffe für Boyer-Moore (binary, large, average-case)

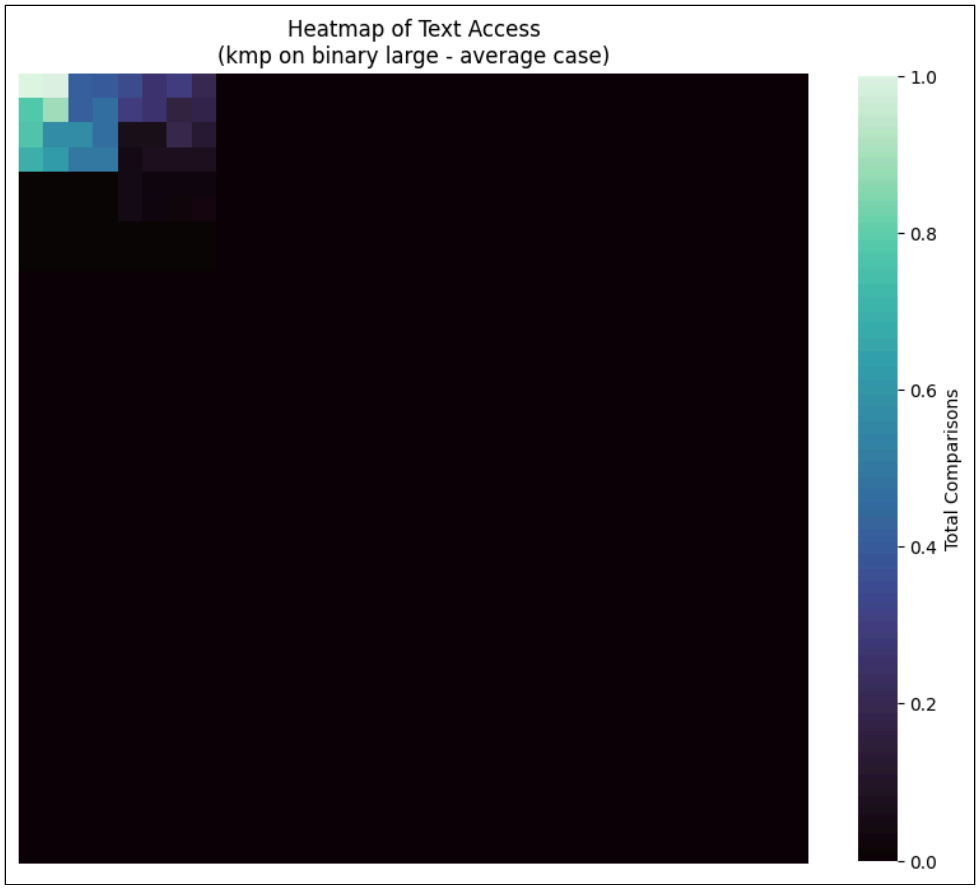

Abbildung 11: Heatmap der Textzugriffe für KMP (binary, large, average-case)

Aufwand in h: 8