Hochschule für Technik, Wirtschaft und Kultur Leipzig

Projekt-Dokumentation

String-Alignment Algorithmen

Modul Algorithm Engineering

Leipzig, 3. März 2019

Bearbeiter: Jan Oelschlegel
Betreuer: Prof. Dr. rer. nat. Karsten Weicker

Inhaltsverzeichnis

1. Projektthema

1.1. Problembeschreibung

Das Projekt beschäftigt sich mit dem Thema des String- oder Sequenzalignment. Dabei werden zwei oder mehrere Zeichenketten Zeichen für Zeichen miteinander verglichen, um eine Aussage zur Ähnlichkeit treffen zu können. Rein formell gesehen wird ein Ähnlichkeitsfaktor zwischen String y und String x und eine resultierende Zeichenkette, mit gleichen Buchstaben und Lücken, ermittelt. Innerhalb dieses Projektes soll nur der Ähnlichkeitsfaktor von DNA-Sequenzen, welche aus den Grundbasen Thymin (T), Guanin (G) , Adenin (A) und Cytosin (C) bestehen, betrachtet werden:

$$\Sigma = \{T, G, A, C\}, n \in \mathbb{N}$$
$$x, y = \Sigma^n$$
$$x \sim y$$

1.2. Anwendungsszenario

Das Sequenzalignment kommt vorallem in der Bioinformatik zum Einsatz. Zum einen wird das globale Alignment dazu eingesetzt, um eine evolutionäre Verwandtschaft zweier Organismen festzustellen. Dazu wird die gesamte Erbinformation mit einer Anderen verglichen. Im Resultat ist dann erkennbar, inwieweit die Organismen gleiche Passagen und Mutationen haben bzw. in welchem Grad sie miteinander verwandt sein könnten. Zum anderen wird das semi-globale Alignment eingesetzt, um Abschnitte innerhalb der unterschiedlichen Zeichenketten zu finden, in denen sie sich am meisten ähnlich sind. Dadurch kann bestimmt werden, ob bestimmte Proteine oder Ähnliches an dieser Stelle andocken könnten oder nicht.

Die Erbinformationen haben meist einen großen Umfang, die Zeichenketten könnrn eine Länge im Millionen wenn nicht sogar Milliarden-Bereich annehmen. In dieser Größenordnung ist es nicht mehr möglich nur mit dem menschlichen Auge die Aufgabe des Alignments durchzuführen. Deshalb wurden viele Algorithmen im Bereich dieser Problematik entwickelt, damit die Arbeit möglichst schnell und effektiv durch einen Computer erledigt werden kann.

2. Algorithmus

Im Projekt wird der Needleman-Wunsch Algorithmus verwendet, welcher auf dem Paradigma des dynamischen Programmierens basiert. Dabei wird das Problem in Teilprobleme zerlegt und die Zwischenergebnisse abgespeichert. Aus den Zwischenresultaten werden nach einer Berechnungsvorschrift fortlaufend neue Ergebnisse bis hin zum Endresultat berechnet. Der Needleman-Wunsch Algorithmus führt ein globales Alignment durch, indem der Ähnlichkeitsfaktor bzw. die Edit-Distanz berechnet wird. Dieser Wert gibt die minimale Anzahl an Umformungsoperationen (Ersetzen, Einfügen, Löschen) an, welche die eine Zeichenkette a in die zu vergleichende Zeichenkette b überführen würden. Die Zwischenergebnisse werden in einer zweidimensionalen Matrix M gespeichert. Die Edit-Instanz ist dann gleich dem Wert rechts unten in der Matrix. Von diesem Wert aus startet auch das Traceback, um das fertige Alignment zu erhalten. Dieser Teil des Algorithmus soll aber nicht Bestandteil dieser Arbeit sein. Ein weiterer wichtiger Begriff innerhalb des Alignments ist die Lücke (Gap). Sie steht für das Auslassen einer Position resultierenden Alignment. Der Vergleich zweier Positionen wird mithilfe einer Bewertungsfunktion w durchgeführt, wobei folgende Werte gelten:

$$w(x, y) = \begin{cases} 1 & \text{für } x = y \\ 0 & \text{sonst} \end{cases}$$

Zu Beginn wird die Matrix mithilfe der einheitlichen Gap-Kosten c initialisiert:

$$\begin{aligned} m &= \text{length}(a) \\ n &= \text{length}(b) \\ c &= -1 \\ M(0,0) &= 0 \\ M(i,0) &= M(i-1,0) + c, 1 \leq i \leq m \\ M(0,j) &= M(0,j-1) + c, 1 \leq j \leq n \end{aligned}$$

Danach werden alle Werte in der Matrix nach und nach aus den bereits vorhandenen Werten unter Verwendung der Bewertungsfunktion w und der Gap-Kosten berechnet:

$$M(i,j) = \max \left\{ \begin{array}{cc} M(i-1,j-1) + w\left(a_i, b_j\right) & \text{Match bzw. Mismatch} \\ M(i-1,j) + c & \text{Deletion} \\ M(i,j-1) + c & \text{Insertion} \end{array} \right\}, 1 \leq i \leq m, 1 \leq j \leq n$$

Die Edit-Distanz ist am Ende des Algorithmus aus der Matrix M ablesbar:

$$distance = M(m,n)$$

Für den Needleman-Wunsch-Algorithmus mit einheitlichen Gap-Kosten wurde eine Laufzeit von $O\left(n^2\right)$ festgestellt. Aufgrund der Speicherung in einer $m \times n$ - Matrix ist der Speicherbedarf hoch und liegt in $O(nm)$.

3. Laufzeitmessung

Dieser Abschnitt soll ein Überblick vermitteln, wie die Laufzeitmessung während des Projektes vonstatten ging. Diese ist notwendig, um die reale mit der vorhergesagten Laufzeit vergleichen zu können. Außerdem kann dadurch gezeigt werden, inwieweit sich die Laufzeit bei realen und generierten Anwendungsdaten unterscheidet. Das gesamte Projekt wurde in *Typescript*[1] (Version 3.3) implementiert und in einer *NodeJS*[2] (Version 10.15) Umgebung ausgeführt

3.1. Vorgehensweise

Die Laufzeit des Algorithmus wird mithilfe der Javascript-Funktion *performance.now()* gemessen, welche die vergangene Zeit seit dem Start des Ausführungskontextes in Millisekunden zurück gibt. Es wird nur die Ausführung des Algorithmus gemessen, von der Initialisierung der Matrix bis zum Ausgeben der Edit-Distanz. Da der JIT-Compiler[3] von *NodeJS* zur Laufzeit Optimierungen am Code vornimmt, werden die ersten fünf Messungen verworfen, damit nur die Zeiten des optimierten Codes in Betracht gezogen werden. Insgesamt werden pro Durchlauf 40 Messungen durchgeführt, abzüglich der „Aufwärmrunden“ ergibt das 35 Messungen. Javascript besitzt eine automatische Speicherverwaltung, sodass ein *Garbage Collector* zum Einsatz kommt. Dieser wird am Ende eines Durchlaufes getriggert, damit die *Garbage Collection* nicht zufällig in einer Messung durchgeführt wird und somit die Laufzeit beeinflusst. Für einen Durchlauf gibt es eine definierte Zeichenkette-Länge bzw. gibt es einen Pool an Längen. Dieser wird Durchlauf für Durchlauf abgearbeitet, sodass für eine Länge 35 Messwerte vorliegen. In diesem Projekt liegen die String-Längen im Bereich von 1.000 bis 15.000 in 1.000er Schritten. Insgesamt werden demnach 525 (15x35) Messungen erfasst.

Die gemessenen Zeiten werden zusammen mit der Länge der Zeichenkette zeilenweise in eine Datei geschrieben. Diese Daten werden mit dem Programm *Gnuplot*[4] ausgewertet und visualisiert.

3.2. Vergleich reale/generierte Daten

Es soll untersucht werden, ob sich die Laufzeiten bei realen DNA-Daten anders verhalten als bei generierten Daten. Beziehungsweise kann daraus abgeleitet werden, ob das gesamte

[1] https://www.typescriptlang.org/
[2] https://nodejs.org/en/
[3] Just-in-Time Compiler
[4] http://www.gnuplot.info/

Projekt mit Echtdaten durchgeführt werden muss oder die Daten zufällig in der richtigen Länge erzeugt werden können.

Als reale DNA-Daten werden die der *westlichen Honigbiene* und *dunklen Erdhummel* verwendet. Es ist zu erwähnen, dass nur ein Auszug des gesamten Genoms genommen wird. Von diesem Teil werden pro Durchlauf nur die ersten Zeichen entsprechend der definierten String-Länge für den Vergleich benutzt.

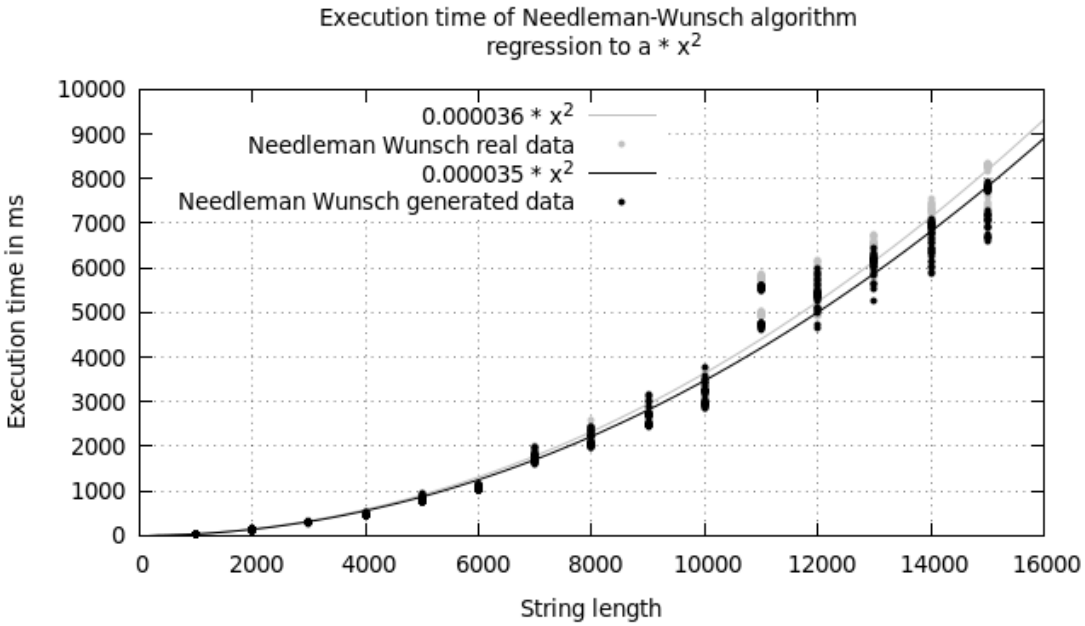

Abbildung 3.1.: Laufzeit-Vergleich reale/generierte Daten

Aus der Grafik geht hervor, dass es keinen bedeutenden Unterschied in der Laufzeit gibt, wenn man anstatt von realen Daten generierte verwendet. Der Needleman-Wunsch Algorithmus war bei den Echtdaten und 15.000 Zeichen um etwa 300ms langsamer. In diesem Projektumfang ist dies eine vernachlässigbare Abweichung. Der Grund dafür könnten andere Prozesse sein, die zufälligerweise während der Messung ausgeführt wurden, obwohl dies durch Beendigung unnötiger Programme eingedämmt wurde.

3.3. asymptotische Laufzeit

Aufbauend aus den Erkenntnissen von Abschnitt 3.2 erfolgt die Erstellung der Daten per Zufall. Dazu werden über eine Funktion zufällige Zeichenketten aus dem Alphabet $\Sigma = \{T, G, A, C\}$ in den definierten Längen generiert. Die Messdaten werden über einen Boxplot dargestellt, um Ausreißer erkenntlich zu machen und statische Grundwerte, wie dem Median, anzuzeigen:

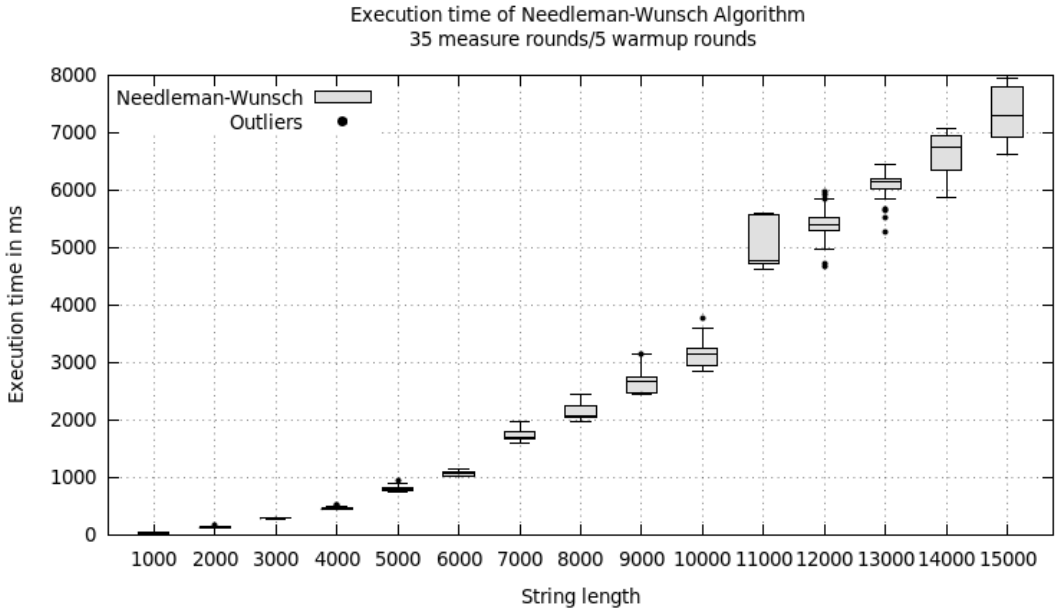

Abbildung 3.2.: Needleman-Wunsch Laufzeiten als Boxplot

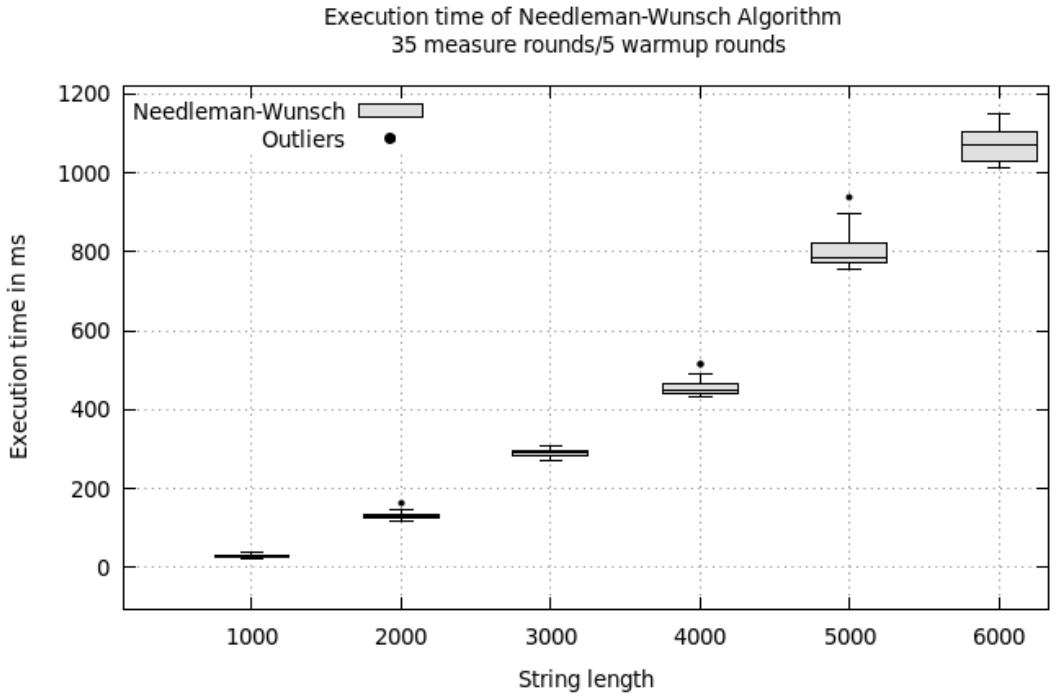

Abbildung 3.3.: Needleman-Wunsch Laufzeiten als Boxplot (String-Length 1000-6000)

Die vorgesagte, asymptotische Laufzeit für den Needleman-Wunsch-Algorithmus[5] liegt in $O(n^2)$, mit n als String-Länge beider Vergleichsobjekte. Dies soll anhand der Messdaten

[5]http://math.mit.edu/classes/18.417/Slides/alignment.pdf

überprüft werden. Dazu wird per Gnuplot eine lineare Regression zur Gleichung $f(x) = a \cdot x^2$ durchgeführt:

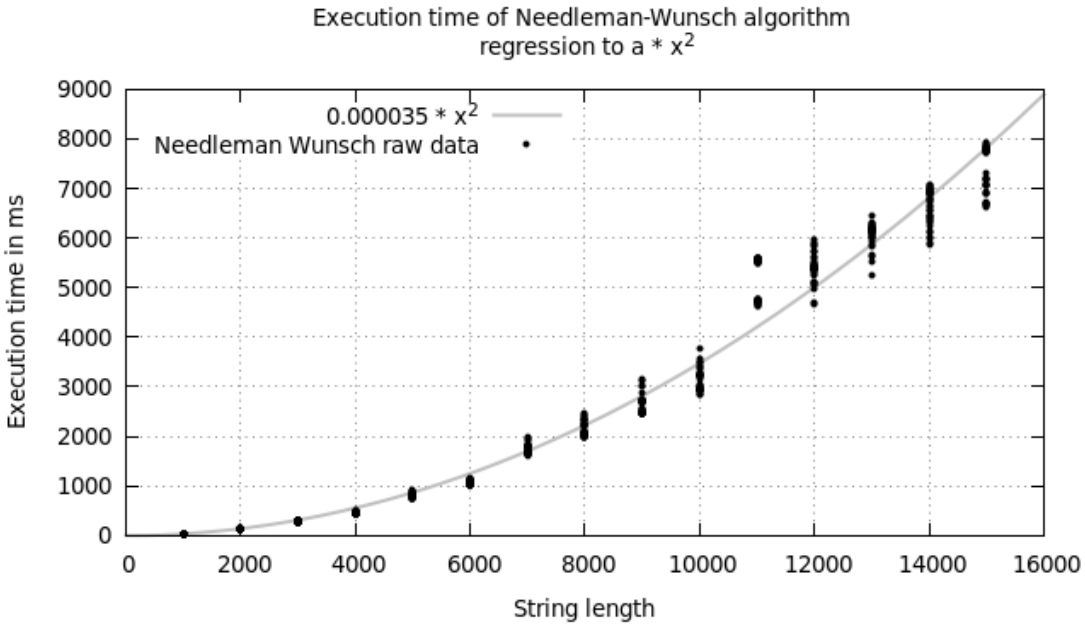

Abbildung 3.4.: lineare Regression der Needleman-Wunsch Laufzeiten

Aus dem Plot geht hervor, dass die vorhergesagte Laufzeit des Algorithmus stimmt, da das Landau-Symbol O verwendet wurde bzw. die Laufzeit-Funktion höchstens so schnell wächst wie $O(n^2)$. Diese Betrachtung zeigt auch, dass die Landau-Notation nur ein Richtwert in der Abschätzung von Algorithmen-Laufzeiten ist. In der Realität spielen viele Faktoren eine Rolle, zum Beispiel die verwendete Programmiersprache oder die Auslastung des ausführenden Systems, welche die Laufzeit beeinflussen können. Sie kann auch, wie dieses Projekt zeigt, wesentlich besser ausfallen, denn der Faktor a der Regressionsfunktion besitzt in der Projektmessung einen sehr kleinen Wert im Bereich von $1 \cdot 10^{-5}$.

Einen kleinen Ausblick auf die erwarteten Laufzeiten mit größeren Problemgrößen lässt sich durch folgenden Grafik darstellen, indem das Intervall der x-Achse etwas vergrößert wurde:

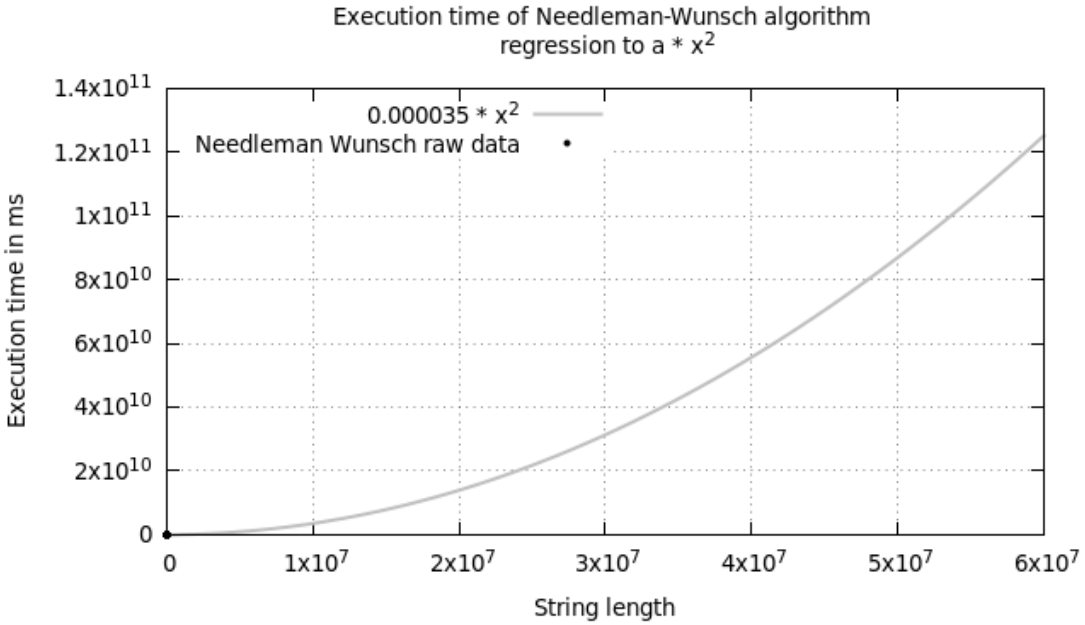

Abbildung 3.5.: lineare Regression der Needleman-Wunsch Laufzeiten mit erweiterter x-Achse

3.4. Profiling

Mithilfe von Profilern hat der Programmierer Einblick in die Ausführung des Codes innerhalb einer bestimmten Laufzeitumgebung. Durch dieses Programmierwerkzeug sind Parameter, wie zum Beispiel der Speicherbrauch, die Benutzung von CPU-Zeit der aufgerufenen Funktionen oder nebenläufige Prozesse, sichtbar. Der Entwickler kann dadurch Problembereiche im Quellcode aufdecken oder häufig aufgerufene Funktionen wenn möglich optimieren. Für *NodeJS* steht auch ein Profiler zur Verfügung. Dazu muss die Laufzeitumgebung mit dem Parameter *inspect* gestartet werden:

```
1 node --inspect main.js
```

Nach dem Start der Applikation stehen mithilfe des Chrome-Browsers als Debug-Schnittstelle verschiedene Profiling-Werkzeuge zur Verfügung. Das CPU-Profiling der Needleman-Wunsch Implementierung mit einer eingestellten String-Länge von 15.000 ergab folgende Auswertung:

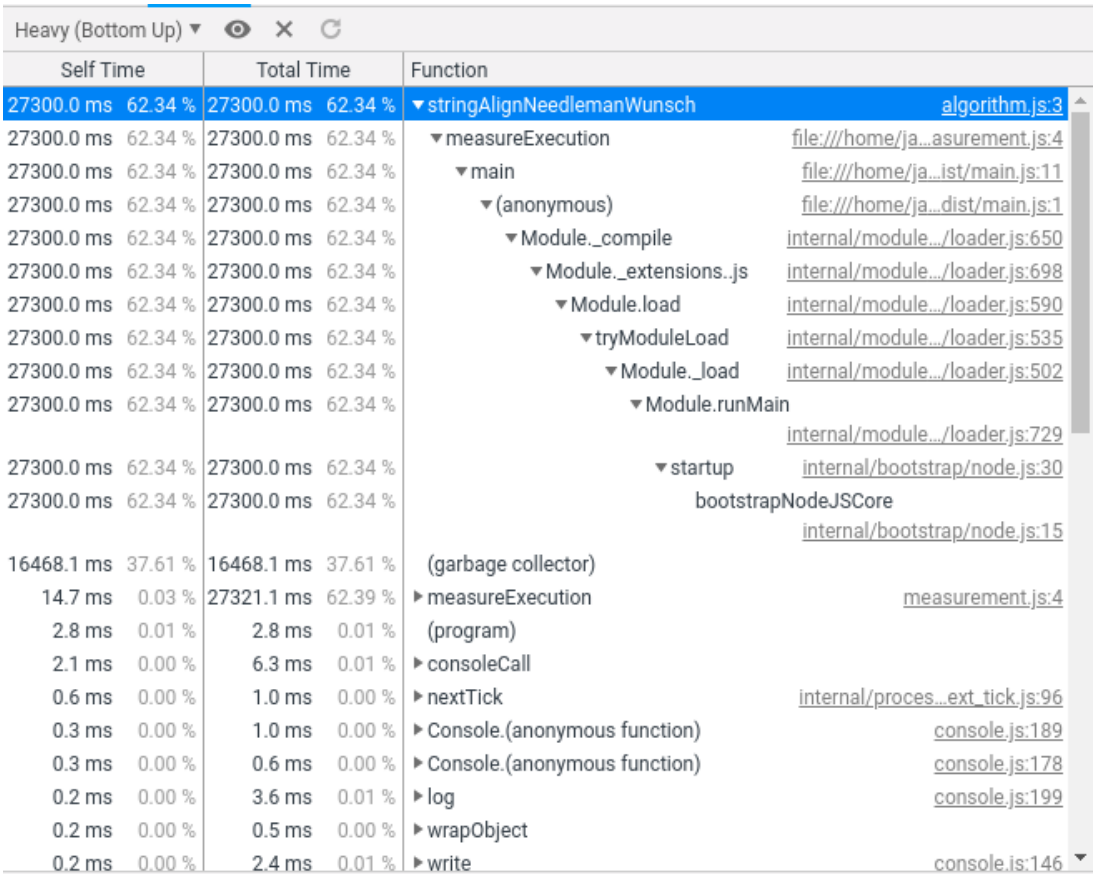

Abbildung 3.6.: CPU-Profiling der Projekt Implementierung

Daraus ist nicht ersichtlich, welche Code-Bereiche innerhalb der Implementierung eine lange Ausführungszeit besitzen. Eine alternative Messmethode ist der Einsatz der Javascript-Funktion *console.time()*, mit dessen Hilfe lässt sich die vergangene Zeit vom Beginn der Ausführung einer Zeile im Code bis zum Ende der Ausführung einer anderen Zeile messen. Mit dieser Funktion wurden grundlegende Bereiche gemessen:

Aktion	**Laufzeit**
Array-Erstellung	0.199ms
Befüllen mit Basiswerten	0.550ms
Gehe durch das Array und berechne alle Werte	7205.058ms
Errechne maximalen Wert aus Zellen	0.001ms
Gesamte Laufzeit	7206.136ms

Aus diesen Werten geht hervor, dass die meiste Zeit für die Berechnung der einzelnen Werte benötigt wird. Allgemein lässt sich behaupten, dass eine Optimierung des Needleman-Wunsch-Algorithmus eher schwierig ist, da dieser nicht sehr komplex ist und es

dadurch nur auf Grundoperationen (Array-Erstellung, arithmetische Operatoren, Schleifen) ankommt. Deshalb wurde der Speicherverbrauch genauer untersucht, um da eventuelle Optimierungspotentiale zu finden. Das Profiling des Heapspace ergab folgende Werte:

Constructor	Distance	Shallow Size ▼		Retained Size	
▸ (array) ×20453	2	690 430 272	100 %	690 636 592	100 %
▸ (compiled code) ×3530	3	1 074 552	0 %	2 486 176	0 %
▸ (string) ×6821	2	1 050 512	0 %	1 050 616	0 %
▸ (system) ×14118	–	790 936	0 %	1 384 088	0 %
▸ Array ×15178	3	485 696	0 %	689 962 560	100 %
▸ (closure) ×4255	2	256 744	0 %	1 553 172	0 %
▸ system / Context ×357	3	34 920	0 %	416 563	0 %
▸ (concatenated string) ×857	4	34 280	0 %	66 128	0 %
▸ Object ×576	2	33 416	0 %	879 952	0 %
▸ system / JSArrayBufferData ×17	5	9 135	0 %	9 135	0 %
▸ Error ×131	3	7 336	0 %	68 056	0 %
▸ NativeModule ×55	8	4 840	0 %	69 968	0 %
▸ TypeError ×56	5	3 136	0 %	29 568	0 %
▸ (symbol) ×77	3	2 464	0 %	3 688	0 %
▸ ContextifyScript ×57	4	1 848	0 %	5 168	0 %
▸ Node / ContextifyScript ×55	10	1 760	0 %	1 760	0 %
▸ (regexp) ×27	4	1 512	0 %	9 128	0 %
▸ RangeError ×26	5	1 456	0 %	13 728	0 %
▸ ArrayBuffer ×20	3	1 264	0 %	11 719	0 %
▸ TypedArray ×22	3	1 232	0 %	6 208	0 %
▸ ObjectMirror ×10	–	1 040	0 %	12 760	0 %

Abbildung 3.7.: Memory-Profiling der Projekt Implementierung

Es ist zu erkennen, dass der meiste Hauptspeicher für Arrays benötigt wird. Die Ursache ist der Ansatz des dynamischen Programmierens Zwischenergebnisse in einer Tabellen-Struktur zu halten. Durch sehr lange Zeichenketten wird das Resultat-Array groß, genauer gesagt $n \cdot m$ groß. Bei einer Zeichenlänge von 23.000 bricht die Ausführung des Codes mit einem „Allocation failed - JavaScript heap out of memory“ - Fehler ab, obwohl der Heapspace der NodeJS-Engine auf 6 GB erweitert wurde. Das grundlegende Problem des Needleman-Wunsch-Algorithmus ist also der Speicherverbrauch.

4. Optimierung

4.1. Ansatz

Im Abschnitt 3.4 wurde der Speicherverbrauch als Problem erkannt und soll innerhalb des Projektes optimiert werden. Die Implementierung des Algorithmus soll nur die Berechnung des Edit-Distanz zweier Zeichenketten und kein globales Alignment durch Backtracking durchführen. Dadurch ist es nicht nötig, alle Zwischenergebnisse bis zum Ende des Programms zu behalten. Aus der Beschreibung des Verfahrens im Abschnitt 2 geht hervor, dass zur Berechnung immer nur der linke, obere und schräg-linke Wert in der Tabelle benötigt wird. Da der Algorithmus zeilenweise vorgeht, werden demnach immer nur zwei Zeilen von Zwischenergebnissen benötigt, die aktuelle und die vorherige Zeile. Beide Zeilen werden jeweils durch ein Array repräsentiert, zum Beispiel *current[]* und *past[]*. Am Ende der Berechnung einer Zeile wird *past* mit *current* überschrieben und in *current* werden die aktuellen Ergebnisse Spalte für Spalte abgelegt. Dadurch rücken sinnbildlich immer zwei Zeilen um eins weiter. Zusammenfassend gesagt lässt sich also das Ergebnis-Array von $n \cdot m$ auf $2 \cdot m$ verkleinern, was sich auch positiv auf den Speicherverbrauch bemerkbar machen sollte. Eventuell ist sogar eine Verbesserung in der Laufzeit erreichbar.

4.2. Beobachtung

Durch den Verbesserungsansatz aus dem Abschnitt 4.1 ist die Berechnung der Edit-Distanz von Zeichenketten der Länge 50 Mio. möglich geworden. Die Laufzeit mit dieser Länge ist mit ca. $8,5 \cdot 10^7$ Sekunden sehr groß[6], weswegen die endgültige Edit-Distanz im Testdurchlauf nicht berechnet werden konnte. Das Programm lief aber 10 Minuten lang ohne Abstürze. Da während der Ausführung aufgrund des neuen Ansatzes der Speicherverbrauch nicht ansteigen sollte, ist davon auszugehen, dass der Algorithmus ohne Abbruch bzw. Fehler zum Ende gekommen wäre. Der Speicherverbrauch wurde also deutlich optimiert.

Eine andere Frage ist, ob sich auch die Laufzeit durch die Code-Optimierung verbessert hat. Dazu wurde wieder eine Testreihe durchgeführt mit Zeichenketten von 1.000 bis 15.000 Buchstaben mit einer Schrittweite von 1.000:

[6] siehe Abbildung 3.5

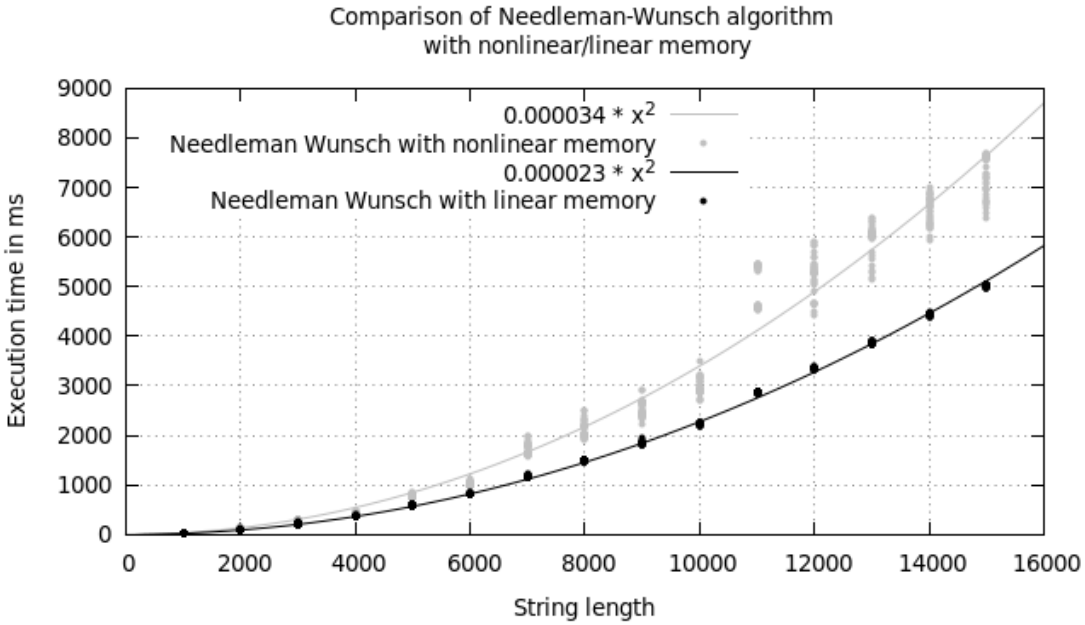

Abbildung 4.1.: Laufzeit-Vergleich ohne/mit Speicheroptimierung

Es ist erkennbar, dass anscheinend auch die Laufzeit durch die Optimierung verbessert wurde. Einen genauen Beweis kann ein Hypothesentest erbringen.

4.3. Hypothesen-Test

Der Hypothesen-Test wird mit einem Zweistichproben-t-Test für abhängige Stichproben durchgeführt. Bestandteil der Untersuchung sind der speicheroptimierte Algorithmus A_1 und der unoptimierte Algorithmus A_2. Es soll betrachtet werden, ob A_1 bei einer String-Länge von 1.000 und 15.000 schneller als A_2 ist. Daraus resultiert folgende Null-Hypothese:

$$\mu_1 < \mu_2$$
$$H_o : \tilde{\mu}_1 \geq \tilde{\mu}_2$$

Zunächst wird der Test mit der Problemgröße 1.000 durchgeführt[7]:

n	35
μ_d	5,06095
σ_d	3,81951
t	6,80998
α	0,001
$t(0,999,35)$	3,333

Aus den Werten geht hervor, dass die Nullhypothese abgelehnt werden muss, da:

$$t(0.999, 35) \approx 3.333 < 6,8$$

Der Hypothesentest zeigt, dass der speicher-optimierte Algorithmus A_1 schneller als der nicht-optimierte Algorithmus A_2 ist (mit einem Fehler von 0,1%).

Bei der Durchführung des Hypothesentests mit der String-Länge 15.000 ergaben sich folgende Werte[8]:

n	35
μ_d	2212,93821
σ_d	408,69380
t	13,76638
α	0,001
$t(0,999,35)$	3,333

Die Nullhypothese muss auch in diesem Fall abgelehnt werden, da:

$$t(0.999, 35) \approx 3.333 < 13,8$$

Der Hypothesentest zeigt wieder, dass der speicher-optimierte Algorithmus A_1 schneller als der nicht-optimierte Algorithmus A_2 ist (mit einem Fehler von 0,1%). Abschließend lässt sich also festhalten, dass die Optimierung nicht nur den Speicherverbrauch verringert hat, sondern auch die Laufzeit des Algorithmus. Die Vermutungen aus dem Abschnitt 4.2 wurden bestätigt.

[7] siehe Werte im Anhang A.1
[8] siehe Basiswerte im Anhang A.2

A. Anhang

A.1. Messwerte mit Stringlänge 1.000

A_1	A_2	Differenz d_i
22,8917679999722	29,5513249999494	-6,65955699997721
23,6395790000097	26,5960490000434	-2,95647000003373
24,7083529999945	29,9469399999944	-5,23858699999983
23,6405509999604	25,5453949999646	-1,90484400000423
25,0588330000173	25,5024969999795	-0,443663999962155
24,1096030000481	40,6144240000285	-16,5048209999804
25,236350000021	25,1474109999835	0,088939000037499
23,4745230000117	25,1836459999904	-1,70912299997872
23,2949420000077	31,6187490000157	-8,32380700000795
23,2052510000067	25,8027730000322	-2,59752200002549
22,9608500000322	32,0387280000141	-9,07787799998187
23,8606789999758	27,3897219999926	-3,52904300001683
24,5748620000086	29,3944670000346	-4,81960500002606
23,8174219999928	34,0919240000076	-10,2745020000148
23,9794049999909	27,4522570000263	-3,47285200003535
24,4938510000356	28,8177910000086	-4,32393999997294
24,1563440000173	30,6326520000002	-6,47630799998296
24,2146749999956	26,7012090000208	-2,48653400002513
23,4080759999924	31,6346239999984	-8,22654800000601
24,0309359999956	25,8981909999857	-1,86725499999011
23,318478000001	26,6049230000353	-3,28644500003429
27,2455780000309	31,2123750000028	-3,96679699997185
24,7822009999654	27,5201009999728	-2,73790000000736
24,3478590000304	31,7642660000129	-7,41640699998243
23,6883970000199	38,1214020000189	-14,433004999999
23,6732459999621	24,147071000014	-0,473825000051875
24,2650250000297	28,6820210000151	-4,41699599998537
26,8218120000092	26,3977750000195	0,424036999989767

25,9065469999914	32,6939589999965	-6,78741200000513
24,4842740000458	34,548414000019	-10,0641399999731
22,8548150000279	29,3823199999751	-6,52750499994727
24,0956320000114	25,2802890000166	-1,18465700000525
23,8302030000486	28,8475170000456	-5,01731399999699
23,6695860000327	26,7305460000061	-3,06095999997342
23,6925870000268	31,072668000008	-7,38008099998115

A.2. Messwerte mit Stringlänge 15.000

A_1	A_2	Differenz d_i
5038,25505000027	6880,03254600009	-1841,77749599982
5002,25651000021	7600,21829000022	-2597,96178000001
5028,17806099961	6726,98244599998	-1698,80438500037
4985,60948200012	7693,57829700038	-2707,96881500026
5045,86665900005	7636,03702099994	-2590,17036199989
5022,00408599991	7181,17869699979	-2159,17461099988
5028,55132699991	6389,8976970003	-1361,34637000039
4995,03144499986	6680,06992099993	-1685,03847600007
5018,5181979998	7642,48234899994	-2623,96415100014
4977,92923399992	7207,98024299974	-2230,05100899981
5033,66500199959	7647,70115200011	-2614,03615000052
5025,97221000027	7214,08166200016	-2188,10945199989
5024,12770000007	6498,90807900019	-1474,78037900012
5047,05245599989	7281,52186799981	-2234,46941199992
5015,82268600026	7636,28200900042	-2620,45932300016
5005,2283160002	6660,96978399996	-1655,74146799976
4983,54636099981	7560,03046600008	-2576,48410500027
5003,24110099999	7587,24094399996	-2583,99984299997
5002,80755200004	7097,9789450001	-2095,17139300006
5031,02484299988	7573,15609299997	-2542,13125000009
4999,32460600045	7087,7460769997	-2088,42147099925

5020,77416099981	7667,85300799972	-2647,07884699991
5048,45756000001	6960,70856499998	-1912,25100499997
4989,31516199978	7619,0335819996	-2629,71841999982
5019,05515799997	6980,47561399965	-1961,42045599967
5024,61529200012	7553,00858200016	-2528,39329000004
5028,68751299987	6789,34643599996	-1760,6589230001
5003,6992850001	7621,31834500004	-2617,61905999994
5025,93008799991	6605,28941800026	-1579,35933000036
5036,54110600008	7268,35989600001	-2231,81878999993
5035,27958300011	7699,03468699986	-2663,75510399975
4994,93236100022	6711,39943199977	-1716,46707099956
4993,27404200006	7405,58561299974	-2412,31157099968
4980,94557499979	7643,49861499993	-2662,55304000014
5000,76884600008	6960,1396420002	-1959,37079600012