



Hochschule für Technik
und Wirtschaft Berlin

University of Applied Sciences

Implementation eines verifizierbaren paarweisen Sequenzalinierers auf Basis eines gemischt-ganzzahligen Optimierungsproblems

Bachelorarbeit

Studiengang

Angewandte Informatik

Fachbereich 4

vorgelegt von

Fynn Freyer

Datum:

Berlin, 12.08.2024

Erstgutachter: Prof. Dr.-Ing. Piotr Wojciech Dabrowski

Zweitgutachter: M. Sc. Alexander Hinzer

Zusammenfassung

Die vorliegende Arbeit behandelt das paarweise optimale globale Sequenzalinierungsproblem im Allgemeinen und die folgenden Fragen im Speziellen:

1. Wie lässt sich Sequenzalinierung als lineares, gemischt-ganzzahliges Optimierungsproblem darstellen?
2. Mit welchen Methoden kann das aufgestellte Modell gelöst werden?
3. Wie können diese Methoden in einem Computerprogramm umgesetzt werden?
4. Wie kann die Korrektheit eines auf Basis des gewählten Ansatzes implementierten Softwaresystems verifiziert werden?

Die Beantwortung dieser Fragen zielt darauf ab, anhand eines praktischen Beispiels zu erforschen, wie qualitativ hochwertige Software gebaut werden kann. Insbesondere, inwiefern die Wahl einer nicht klassischen Formulierung bereits gelöster Probleme bei der Modellierung sinnvoll ist und welche Relevanz Korrektheit für das Software-Engineering hat bzw. ob es möglich ist, nachweislich fehlerfreie Programme zu schreiben und wie nützlich dieser Anspruch ist.

Dazu wurde zunächst ein Modell vorgestellt, welches Sequenzalinierung als mathematisches Optimierungsproblem formuliert. Daraufhin wurde ein Ansatz entwickelt, um dieses Modell zu lösen und die Lösung in der Programmiersprache Haskell zu implementieren. Zuletzt wurde die Korrektheit der Implementierung verifiziert.

Bei der Betrachtung möglicher Lösungsansätze weist die Arbeit einen Zusammenhang zwischen dem vorgestellten Modell und dem klassischen Algorithmus zur globalen Sequenzalinierung von Needleman und Wunsch nach. Da die Überführung des Optimierungsproblems in den Needleman-Wunsch-Algorithmus eine methodische Schwäche aufweist, lässt sich die Isomorphie beider Vorgehensarten jedoch nicht eindeutig nachweisen. In der Diskussion wurde daraufhin ein Ansatz entwickelt, um die identifizierte Komplikation zu beheben.

Dass das implementierte Programm, trotz nachgewiesener Korrektheit, signifikante Probleme aufweist, lässt darauf schließen, dass Korrektheitsbeweise zwar ein nützliches Werkzeug für Programmierer darstellen, aber ausführliche Tests und andere Praktiken der analytischen Qualitätssicherung nicht ersetzen können.

Korrektheit ist nur *eine* der notwendigen Voraussetzungen, um nutzbare und qualitativ hochwertige Software zu produzieren.

Der im Zusammenhang mit der Beweisführung aufgetretene Aufwand legt darüber hinaus den Schluss nahe, dass es vernünftig ist, Beweise auf klar abgrenzbare Systemkomponenten mit kritischer Funktionalität und klarer Spezifikation zu beschränken.

Danksagung

Gott sei Dank, dass es endlich vorbei ist.

Außerdem gilt mein Dank allen Dozenten, von denen ich interessante Dinge lernen durfte, allen Autoren die gute Bücher geschrieben haben, allen Kommilitonen die mir nicht auf den Nerv gegangen sind und allen Verwandten, Freunden und Bekannten die mich unterstützt haben.

Most people are fools, most authority is malignant, God does not exist, and everything is wrong.

– Ted Nelson

Inhaltsverzeichnis

1	Einleitung	1
1.1	Hintergrund	1
1.1.1	Sequenzalinierung als Optimierungsproblem	1
1.1.2	Korrektheit	1
1.2	Problemstellung	2
1.3	Aufbau	2
2	Material und Methoden	3
2.1	Theoretische Hintergründe	3
2.1.1	Literarische Programmierung	3
2.1.2	Optimierungsprobleme	3
2.1.3	Notation	4
2.1.4	Biologischer Hintergrund	4
2.1.5	Dynamische Programmierung	8
2.1.6	Funktionale Programmierung	10
2.1.7	Verifikation	11
2.2	Praktische Hintergründe	17
2.2.1	Pandoc	17
2.2.2	Haskell	17
2.2.3	Entangle	18
3	Durchführung und Ergebnisse	19
3.1	Sequenzalinierung als Optimierungsproblem	19
3.1.1	Templates	19
3.1.2	Problemformulierung	20
3.2	Lösung des Optimierungsproblems	27
3.2.1	Ausprobieren	27
3.2.2	Needleman-Wunsch	30
3.3	Implementation eines Sequenzalinierers	44
3.3.1	Naive Implementation	45
3.3.2	Needleman-Wunsch Implementation	55
3.3.3	Ausführung	71
3.3.4	Modularisierung	73
3.3.5	Paketierung	76
3.3.6	Test	78
3.4	Verifikation	80
3.4.1	Grundlegende Definitionen	80
3.4.2	Maximierung	83
3.4.3	Füllregeln	88
4	Diskussion	91
4.1	Zusammenfassung	91
4.1.1	Problemformulierung	91
4.1.2	Problemlösung	91
4.1.3	Implementation	92
4.1.4	Verifikation	93
4.2	Interpretation	93
4.3	Limitationen	94
4.3.1	Komplexität biologischer Fragestellungen	94
4.3.2	Spezifikation des Lösungsraums	94
4.3.3	Garantie fester Alignmentlängen	94
4.3.4	Performance	95
4.3.5	Grenzen der Beweisbarkeit	98
4.4	Ergänzungen	99
4.4.1	Affine Gapkosten	99
4.4.2	Erweiterung auf MSA	100

5	Fazit	103
5.1	Erkenntnisse	103
5.1.1	Sequenzalignierung als lineares, gemischt-ganzzahliges Optimierungsproblem . . .	103
5.1.2	Lösung des MILP-Problems	103
5.1.3	Umsetzung in ein Computerprogramm	103
5.1.4	Formale Verifikation der Implementation	104
5.1.5	Relevanz von Korrektheit im Software-Engineering	104
5.1.6	Vorteile einer nicht-klassische Formulierung	104
5.2	Ausblick	105
5.2.1	Feste Alignmentlängen	105
5.2.2	Performanceanalyse	105
5.2.3	Computergestützte Verifikation	105
5.2.4	Erweiterung des Modells	105
6	Quellen	106

Abbildungsverzeichnis

1	Eine konvexe und eine nicht-konvexe Menge.	8
2	Reihenfolge der Berechnungen und ausgefüllte NW-Matrix.	31
3	NW-Matrizen mit eingetragenen Distanzen zur Hauptdiagonalen.	36
4	Reihenfolge der Berechnungen und ausgefüllte Matrix für den angepassten NW-Algorithmus.	39
5	Ausführungszeiten von der Haskell-Implementation und Biopythons PairwiseAligner im Vergleich.	96
6	Die Speicherauslastung beim Alinieren der ersten 125 Symbole von pol. fa.	97
7	Übersicht über die Speicherauslastung pro Funktion.	98

1 Einleitung

Die vorliegende Arbeit untersucht das Problem der optimalen globalen Sequenzalinierung, Analogien zwischen verschiedenen Darstellungsformen dieses Problems und die Verifikation von Softwaresystemen.

1.1 Hintergrund

Sequenzalinierung wird in der Biologie genutzt, um die Ähnlichkeit zwischen verschiedenen DNA- und Proteinsequenzen zu bestimmen. Dadurch können z. B. konservierte Abschnitte identifiziert und Verwandtschaftsverhältnisse abgeschätzt werden.

1.1.1 Sequenzalinierung als Optimierungsproblem

Sequenzalinierung kann als Entscheidungsproblem interpretiert werden, bei dem Symbole einer Sequenz entweder einem Symbol der anderen Sequenz oder einer Lücke zugewiesen werden.

Es existieren viele Ansätze, um Entscheidungsprobleme zu lösen. Das Feld der mathematischen Optimierung stellt verschiedene Möglichkeiten und Verfahren zur Verfügung, um solche Probleme aufzustellen und optimale Lösungen für sie zu bestimmen.

Insbesondere ist das Problem der optimalen globalen Sequenzalinierung ein bereits seit 1970 durch den klassischen Algorithmus von Needleman und Wunsch gelöstes Problem. [1] Dieser verwendet Methoden der dynamischen Programmierung, um eine Rekursionsbeziehung zwischen Teilalignments der betrachteten Sequenzen zu formulieren, welche Optimalität garantiert.

Alternativ kann das Alinierungsproblem aber auch als Optimierungsproblem mit geschlossener Formel dargestellt werden, bei dem Sequenzsymbole einer Alignmentstabelle mit fester Größe zugewiesen werden. Möglicherweise kann das aus einer solchen Formulierung resultierende Vokabular genutzt werden, um eine, im Vergleich zu traditionellen Ansätzen, leichtere mathematische Analyse durchzuführen.

1.1.2 Korrektheit

Da Analyseergebnisse im medizinischen Bereich Therapieentscheidungen und damit indirekt den Gesundheitszustand von Patienten beeinflussen, ist die Richtigkeit der durchgeführten Analysen dort von besonderer Bedeutung.

Mithilfe von Sequenzalinierung kann bspw. das Vorhandensein bestimmter Resistenzmutationen bei einer HIV-Infektion festgestellt oder ausgeschlossen werden. Auf Grundlage dieser Informationen entscheiden Ärzte, mit welchen Medikamenten Patienten behandelt werden. Da diese Therapieentscheidungen direkten Einfluss auf den Therapieerfolg haben, ist es von großer Bedeutung, dass die Alinierung korrekt durchgeführt wurde.

Wenn wir in der Informatik von *Korrektheit* sprechen, meinen wir, dass ein Programm seine zugrundeliegende Spezifikation einhält und die Überprüfung der Korrektheit heißt *Verifikation*.

In der Praxis werden verschiedene Methoden genutzt, um Programme zu verifizieren. Eine typische Herangehensweise ist das Testen von Software.

[P]rogram testing can be a very effective way to show the presence of bugs, but is hopelessly inadequate for showing their absence. The only effective way to raise the confidence level of a program significantly is to give a convincing proof of its correctness.

– Edsger W. Dijkstra [2]

Dieses Zitat von Dijkstra zeigt ein fundamentales Problem mit diesem Ansatz auf.

Tests können nur die Abwesenheit von unerwünschtem Verhalten nachweisen. Aus dem Vorhandensein von Tests und dem erfolgreichen Durchlaufen dieser folgt aber nicht logisch zwingend, dass sich das Programm in allen Situationen richtig verhält.

Dijkstra erwähnt als Alternative, die dieses Problem umgeht, den Korrektheitsbeweis. Man spricht hier auch von *formaler* Verifikation.

Bei der formalen Verifikation wird ein mathematisches Modell des Programms aufgestellt und analysiert. Dabei soll gezeigt werden, dass durch die Ausführung eines Programms unter bestimmten Vorbedingungen die spezifizierten Nachbedingungen und Invarianten folgen müssen.

Aufgrund des Stellenwerts von Korrektheit im medizinischen Bereich im Allgemeinen und bei der Sequenzalinierung im Besonderen, kann formale Verifikation bei der Entwicklung von Software für diese Anwendungsbereiche eine wichtige Rolle spielen.

1.2 Problemstellung

Aus dem beschriebenen Hintergrund ergeben sich die folgenden Forschungsfragen:

1. Wie lässt sich Sequenzalinierung als lineares, gemischt-ganzzahliges Optimierungsproblem darstellen?
2. Mit welchen Methoden kann das aufgestellte Modell gelöst werden?
3. Wie können diese Methoden in einem Computerprogramm umgesetzt werden?
4. Wie kann die Korrektheit eines auf Basis der identifizierten Lösungsmethoden implementierten Softwaresystems verifiziert werden?
5. Welche Relevanz hat Korrektheit beim Bau qualitativ hochwertiger Software?
6. Bietet eine nicht klassische Formulierung des Alignmentproblems Vorteile?

1.3 Aufbau

Die Arbeit besteht aus einem Materialteil, einem Durchführungs- und Ergebnisteil, der Diskussion der Ergebnisse und einem Fazit.

Im Materialteil stellen wir die notwendigen Grundlagen vor, um die Arbeit im Durchführungsteil zu verstehen.

Der Hauptteil ist in vier Sektionen gegliedert.

1. Zunächst wird ein Modell für die Sequenzalinierung als lineares, gemischt-ganzzahliges Optimierungsproblem vorgestellt.
2. Darauf aufbauend wird ein Ansatz entwickelt, um das dargestellte Modell zu lösen.
3. Auf Grundlage des entwickelten Lösungsansatzes wird ein Softwaresystem implementiert.
4. Durch formale Verifikation wird die Korrektheit des Systems nachgewiesen.

An den Hauptteil anschließend wird die Bedeutung der erzielten Ergebnisse diskutiert und sowohl die Argumentation im Lösungsansatz als auch die Implementation auf Schwächen untersucht.

2 Material und Methoden

In diesem Kapitel werden die theoretischen und praktischen Hintergründe der Arbeit dargestellt.

2.1 Theoretische Hintergründe

Zu den wesentlichen theoretischen Hintergründen zählen:

2.1.1 Literarische Programmierung

Die Arbeit ist in Donald Knuths Stil des literarischen Programmierens (bzw. *literate Programming*) verfasst. [3] Dabei steht nicht der Code im Zentrum, sondern die prosaische Beschreibung der Intention und der narrativen Entwicklung des Vorgehens.

Dies hat den Grund, dass Code häufiger gelesen als geschrieben wird und eine verständliche Beschreibung daher ebenso wichtig ist, wie der Code selbst.

Bei der literarischen Programmierung gibt es die beiden Operationen “*tangle*” und “*weave*”, die auf den Projektquellen ausgeführt werden können. Beim **Tanglen**, wird aus dem Projekt Quelltext generiert, der ausgeführt oder kompiliert werden kann, während beim **Weaven** die menschenlesbare Projektdokumentation generiert wird.

Die wesentlichen Aspekte des Literate Programming kommen bei der Implementation des Aligners zur Anwendung, aber auch die restlichen Teile der Arbeit versuchen, die dargestellten Gedanken zu motivieren und mögliche Fragen vorwegzunehmen.

Extensive Beispiele, Informationen und sonstige Anmerkungen, die keine direkte Relevanz für die Forschungsfrage haben, aber z. B. dem besseren Veranschaulichung des Sachverhalts dienen, sind in farbig codierten Boxen untergebracht und können ggf. übersprungen werden.

2.1.2 Optimierungsprobleme

Bei Optimierungsproblemen wird versucht, eine optimale Lösung für einen Sachverhalt zu finden. Dazu wird eine Funktion $f : D \rightarrow \mathbb{R}$ aufgestellt, welche den “Wert” der zu betrachtenden Prozesse widerspiegelt und für diese einen Eingabewert $x^* \in D$ zu finden, der den maximalen Funktionswert ergibt, sodass $\forall x \in D : f(x^*) \geq f(x)$. Außerdem gibt es üblicherweise noch bestimmte Beschränkungen, denen die errechnete Lösung genügen muss. Bspw. können bei der Belegungsplanung für ein Krankenhaus Betten nicht doppelt belegt werden.

Da wir uns mit *linearen* Optimierungsproblemen beschäftigen und nur lineare Funktionen betrachten, können wir das Problem mittels Matrix-Multiplikation formulieren. Wir schreiben dafür $\max\{c^\top x : Ax \leq b, x \leq 0\}$. Dabei entsprechen die Vektoren $c, x \in \mathbb{R}^n$ den Koeffizienten c von bzw. den Eingaben x in f , und bilden als $c^\top x$ die zu optimierende Funktion f ab. Die Matrix $A \in \mathbb{R}^{n \times m}$ stellen “Kostenkoeffizienten” dar, mit denen die Eingaben x assoziiert sind und der Vektor $b \in \mathbb{R}^m$ entspricht einem zur Verfügung stehenden Budget.

Auf dieser Grundform aufbauend kann man andere Probleme ableiten, bei denen z. B. der Wertebereich bestimmter Variablen eingeschränkt wird. Das Vorgehen beim Aufstellen eines Optimierungsproblems ist üblicherweise dreigliedrig.

1. Die relevanten Variablen identifizieren,
2. Beschränkungen der Lösungsdomäne modellieren und
3. eine geeignete Zielfunktion formulieren.

2.1.3 Notation

Diese Sektion gibt Auskunft über die Bedeutung in der Arbeit genutzter Symbole und Notationen.

Wahrheitswerte werden mit 0, für "falsch" und 1, für "wahr" codiert und wir definieren $\mathbb{B} = \{0, 1\}$.

Indexmengen werden als J_n geschrieben, mit $n \in \mathbb{N}$ und $J_n = \{1, 2, \dots, n\}$, sodass $|J_n| = n$. Als Kurzschreibweise für die Menge $J_{|M|}$, die eine andere Menge M indiziert, schreiben wir J_M .

Prädikatabbildung für p wird mit Iverson-Klammern als $[p]$ notiert.

$$[p] = \begin{cases} 1, & p \text{ ist wahr} \\ 0, & p \text{ ist falsch} \end{cases}$$

Die Iverson-Notation wird nur in Summen-Termen genutzt und erlaubt die simple Darstellung von Index-Auswahlen.

2.1.3.1 Sequenzen Sei eine Sequenz s eine endliche Folge der Länge M von Symbolen über einem bestimmten Alphabet Σ .

$$(s_i)_{i \in J_M}, \text{ mit } M \in \mathbb{N} \quad s_i \in \Sigma \quad (2.1.1)$$

Sequenzlängen, also die Anzahl der Symbole einer Sequenz s , werden durch Betragsstriche notiert $|s|$.

Die **Sequenznummer** der m -ten Sequenz s^m in einer Familie von n Sequenzen s^1, \dots, s^n wird durch einen hochgestellten Index ausgedrückt. *Hochgestellte Indizes* bei anderen Variablen deuten *immer* auf einen *Zusammenhang mit einer bestimmten Sequenz* hin.

Die **Indizierung** von Sequenzen, also Wahl des i -ten Symbols in Sequenz s wird mittels tiefgestellter Indizes als s_i notiert. Dies ist nur definiert, für $i \in J_s$. *Tiefgestellte Indizes* bei anderen Variablen deuten *häufig*, aber nicht immer, auf einen *Zusammenhang mit einer bestimmten Sequenzposition* hin.

Beispiel 2.1

Sei bspw. $s^{\text{bsp}} = \text{GATTACA}$ eine Nukleotidsequenz über dem Alphabet $\Sigma = \{A, C, G, T\}$, bestehend aus den Nukleobasen der DNA.

Die Länge von s^{bsp} ist durch $|s^{\text{bsp}}| = 7$ gegeben.

Wenn wir referenzieren Position i in s^{bsp} als s_i^{bsp} . Also $s_3^{\text{bsp}} = s_4^{\text{bsp}} = T$ und $s_7^{\text{bsp}} = A$. Weiterhin sind s_0^{bsp} und s_8^{bsp} undefiniert, da die Indizes die Grenzen von s überschreiten.

2.1.3.2 Matrizen Eine Matrix $A = (a_{ij})$ kann mithilfe einer Befüllungsregel für ihre Elemente a_{ij} definiert werden. Bspw. $A = (a_{ij})$ mit $a_{ij} = 2(i + j)$.

2.1.4 Biologischer Hintergrund

Bevor das Problem aufgestellt werden kann, wird zunächst der biologische Hintergrund betrachtet.

Zweck der Sequenzanalyse ist es, die Verwandtschaftsbeziehung der betrachteten biologischen Sequenzen abzuschätzen, also welche evolutionären Events zwischen ihnen liegen bzw. durch welche Mutationen sie sich auseinanderentwickelt haben.

Zwei wichtige Klassen von Mutationen in DNA sind Einzelnukleotid-Polymorphismen (**SNPs** oder "Substitutionen") und Insertionen bzw. Deletion (**Indels**). Ein geeignetes Modell biologischer Verwandtschaft, muss diese Mutationen abbilden können.

Verwandtschaft zwischen Organismen lässt sich i.d.R. nicht direkt messen. Stattdessen kann deren aus biologischen Sequenzen bestehendes Erbgut verglichen werden. Dadurch entsteht ein Abbild des evolutionären Ist-Zustandes, welches wir nutzen um die Ähnlichkeit der genetischen Informationen, als indirektes Maß für Verwandtschaft, zu bestimmen.

Damit diese Ähnlichkeit bestimmt werden kann, müssen die Sequenzen gegeneinander ausgerichtet, bzw. aliniert werden. Dieser Vorgang ordnet die zusammengehörigen Symbole der Sequenzen einander zu.

Eine Analogie ist eine Tabelle, bei der verwandte Symbole in derselben Spalte eingetragen werden. Diese Tabelle entspricht einem sog. Alignment und die Ähnlichkeit zwischen den Sequenzen ergibt sich durch Gemeinsamkeiten und Unterschied der Spalten.

2.1.4.1 Arten von Alignments Wenn zwei Sequenzen aliniert werden, spricht man von paarweisem Sequenzalignment (**PSA**) und bei mehr als zwei Sequenzen von einem multiplen Sequenzalignment (**MSA**).

Weiterhin werden *globale* und *lokale* Alignments unterschieden. Ein globales Alignment zweier Sequenzen betrachtet die Gesamtheit der zu alinierenden Sequenzen, während bei einem lokalen Alignment ggf. nur Teilstücke beider Sequenzen gegeneinander ausgerichtet werden.

2.1.4.2 Substitutionen Bei Substitutionen wird ein einzelnes Symbol in einer Sequenz durch ein anderes ersetzt. Wenn sich die Sequenzsymbole in einem Alignment voneinander unterscheiden, spricht man von einem **Mismatch**. *Mismatches werden von uns als SNPs interpretiert.*

Um Ähnlichkeit zu berechnen, müssen die Kosten für die Substitution von einem Symbol durch ein anderes bestimmt sein.

Um solche Kosten zu formulieren, gibt es verschiedene gebräuchliche Ansätze.

2.1.4.2.1 Flache Substitutionskosten Die einfachste Möglichkeit bestraft jede Substitution mit einem festen Wert. ClustalW in Version 1.6¹ arbeitet mit einer festen Strafe für Mismatches. [4]

Für die Sequenzen s^1 und s^2 und Kosten für einen Match w_{match} , bzw. einen Mismatch durch w_{miss} , schreiben wir verkürzend w_{ij} für die Kosten, die durch eine Substitution zwischen s_i^1 und s_j^2 entstehen.

$$w_{ij} = \begin{cases} w_{\text{match}} & , s_i^1 = s_j^2 \\ w_{\text{miss}} & , \text{Andernfalls.} \end{cases} \quad (2.1.2)$$

2.1.4.2.2 Substitutionsmatrizen Ein differenzierterer Ansatz ist die Verwendung von Substitutionsmatrizen.

¹: In der [Dokumentation](#) von ClustalW 2.1 findet sich der folgende Text:

CLUSTALW(1.6). The previous system used by Clustal W, in which matches score 1.0 and mismatches score 0. All matches for IUB symbols also score 0.

Diese haben jeweils eine Spalte und Zeile für jedes mögliche Sequenzsymbol. Die Zellen enthalten die Substitutionskosten, die durch den Austausch von Symbolen entstehen. Anhand von Zeilen- und Spaltenindex wird einfach abgelesen, für welchen Austausch der entsprechende Wert steht.

Beispiel 2.2

Bspw. hätte eine Substitutionsmatrix für Nukleotidsequenzen die folgende Form:

$$W = \begin{array}{c|cccc} & A & C & G & T \\ \hline A & w_{AA} & w_{AC} & w_{AG} & w_{AT} \\ C & w_{CA} & w_{CC} & w_{CG} & w_{CT} \\ G & w_{GA} & w_{GC} & w_{GG} & w_{GT} \\ T & w_{TA} & w_{TC} & w_{TG} & w_{TT} \end{array}$$

Sei W eine Substitutionsmatrix, welche Werte für die Symbole X und Y enthält, dann steht w_{XY} für die Kosten, um ein X durch ein Y zu ersetzen.

Analog zu (2.1.2) können die Kosten einer Substitution zwischen s_i^1 und s_j^2 verkürzt als w_{ij} dargestellt werden.

$$w_{ij} = w_{s_i^1 s_j^2} \quad (2.1.3)$$

Clustal in Version 2 arbeitet mit Substitutionsmatrizen. [5] Gebräuchliche Matrizen für Substitutionen von Aminosäureresten bei Peptidsequenzen sind z.B. BLOSUM [6] und PAM [7].

Es existieren auch komplexe Substitutionsmodelle, wie TN93 [8] oder GTR [9]. Diese definieren bestimmte Eigenschaften für Substitutionsmatrizen, um die biologischen Prozesse im Hintergrund anzunähern. Auf Basis von real beobachteten Mutationen können mithilfe dieser Modelle geeignete Substitutionsmatrizen für spezifische Fragestellungen bestimmt werden.

2.1.4.3 Indels Der Begriff Indel ist ein Kunstwort, welches die Begriffe Insertion und Deletion verbindet. Dabei wurden, im Falle einer Insertion, in einer Sequenz bestimmte Nukleotide hinzugefügt, bzw., bei einer Deletion, entfernt. Alignments können dementsprechend Lücken enthalten. Wenn einer Position im Alignment kein Sequenzsymbol zugewiesen wurde, spricht man von einem **Gap**. *Gaps werden von uns als Indels interpretiert.*

Bei der Alinierung einer Query- und einer Referenzsequenz, deutet eine Lücke in der Query auf eine Deletion und eine Lücke in der Referenz auf eine Insertion hin.

Diese Lücken sind ein weiteres gebräuchliches Kriterium, um Ähnlichkeit zu bestimmen. Auch hier gibt es verschiedene gebräuchliche Modelle, um die Kosten zu modellieren.

2.1.4.3.1 Gaps sind Mismatches Im einfachsten Fall werden Gaps nicht gesondert behandelt, sondern als Mismatches gezählt. Die Bewertung erfolgt dann anhand der für Substitutionen verwendeten Methode.

2.1.4.3.2 Flache Gapkosten Wie bei den flachen Substitutionskosten werden Lücken in der Sequenz hier mit einem flachen Wert pro fehlendem Symbol bestraft. Allerdings existieren in dieser Variante un-

terschiedliche Werte für Substitutionen und Indels.

2.1.4.3.3 Affine Gapkosten Genauere Modelle für Gapkosten nutzen affine² Funktionen zur Kostenbestimmung. Dabei werden verschiedene Kosten für das Öffnen und das Verlängern eines Gaps angelegt.³

Information

Aufgrund biochemischer Zusammenhänge^a ist es relativ unwahrscheinlich, dass überhaupt eine Lücke entsteht.

Wenn dies jedoch der Fall ist, unterscheidet sich die Größe der Wahrscheinlichkeit für Länge 10 nicht stark von der Wahrscheinlichkeit für Länge 11.

^a† Drei aufeinanderfolgende Basen bilden ein Codon, welches eine Aminosäure für ein Protein codiert. Alle Indels mit Längenveränderung $d \not\equiv 0 \pmod{3}$ verschieben den Leserahmen für Codons und sind i. d. R. tödlich.

2.1.4.3.4 Konkave Gapkosten Noch genauere Modelle nutzen konkave Funktionen, um die Gapkosten zu berechnen. Dabei nehmen die Kosten, um eine Lücke zu erweitern, mit steigender Länge der Lücke ab.

Bei sehr kurzen Indels ist der Unterschied, ob eine Base mehr oder weniger entfernt wurde, wichtiger als bei sehr langen Indels.

²† Affine Funktionen werden in der Schulmathematik als "linear" bezeichnet. Affin ist z. B. die Funktion $y = mx + b$.

³† Kosten für Gaps der Länge l ließen sich dann bspw. als $g = w_{\text{extend}}l + w_{\text{open}}$ darstellen.

Information

Eine konkave Funktion hat einen nicht konvexen Epigraphen. Der Epigraph einer Funktion ist die Menge aller Punkte, die auf oder über dem Funktionsgraphen liegen.

Anschaulich gesprochen ist eine Menge dann konvex, wenn man von jedem Punkt in der Menge aus jeden anderen erreichen kann, ohne dabei die Menge zu verlassen.

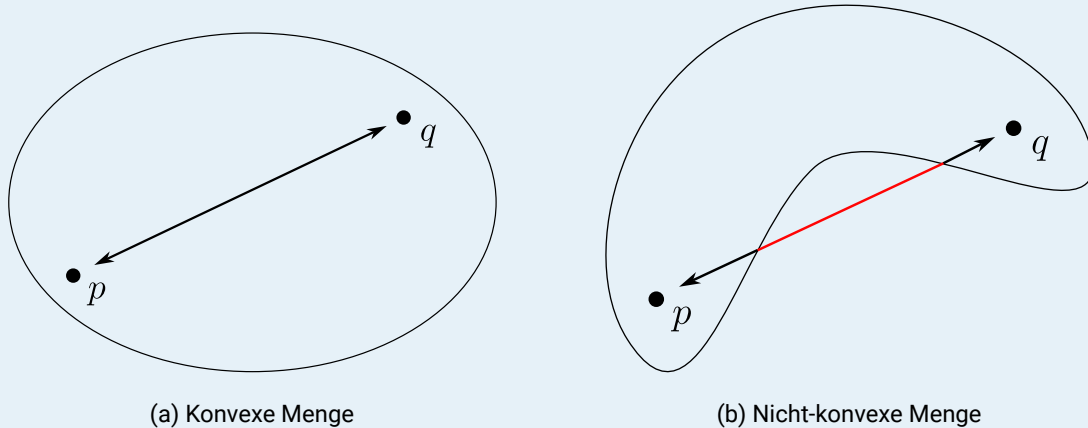


Abbildung 1: Eine konvexe und eine nicht-konvexe Menge.

Ein simples Beispiel für eine konkave Funktion ist $f(x) = -x^2$.

2.1.5 Dynamische Programmierung

Der Begriff der dynamischen Programmierung wurde von Richard Bellman geprägt. [10] Dieser zeigte, dass die Zielfunktion einer bestimmten Klasse von Optimierungsproblemen als Rekursionsformel dargestellt werden kann.

Diese Probleme haben **“optimale Substruktur”**. D.h. die optimale Lösung des Gesamtproblems ist auf die optimale Lösung kleinerer Teilprobleme zurückführbar.

Diese Eigenschaft nutzt man bei dynamischen Programmiermethoden, um Probleme rekursiv zu lösen. Ein bekanntes Beispiel ist die Suche nach dem kürzesten Pfad zwischen zwei Knoten in einem Graphen.

Um ein Problem mit optimaler Substruktur dynamisch zu lösen, wird es in diskrete Schritte aufgeteilt. Man definiert dann *Zustandsvariablen*, welche die aktuelle Situation nach bestimmten Schritten beschreiben, *Kontrollvariablen*, die beschreiben welche Entscheidung für den nächsten Schritt getroffen werden sollte und eine *Wertfunktion*, welche den optimalen Wert beschreibt, der, gegeben bestimmte Zustandsvariablen, erreicht werden kann. Die *Zielfunktion* stellt man dann als Rekursionsbeziehung zwischen der Wertfunktion für Schritt t und Schritt $t - 1$ dar.

2.1.5.1 Needleman-Wunsch Sequenzalinierung ist ein Problem mit optimaler Substruktur. Der Algorithmus für globale Sequenzalinierung von Needleman und Wunsch (**NW**),⁴ macht sich diese Tatsache zunutze um eine Rekursionsbeziehung zu formulieren, mithilfe derer das optimale Alignment produziert werden kann.

⁴ Ursprünglich entwickelt wurde der Algorithmus von Needleman und Wunsch in [1]. David Sankoff ergänzte das Vorgehen in [11] und Smith und Waterman erweiterten das Prinzip in [12] für das Finden lokaler Alignments.

Gegeben zwei Sequenzen $s^1 = (s_1^1, \dots, s_m^1)$ und $s^2 = (s_1^2, \dots, s_n^2)$, wird deren optimales Alignment in irgendeiner Art und Weise vom optimalen Alignment der Subsequenzen $(s_1^1, \dots, s_{m-1}^1)$ und $(s_1^2, \dots, s_{n-1}^2)$ abhängen.

Das leere Alignment dient als Basisfall, der Symbol für Symbol erweitert wird. Die Zustandsvariablen sind gegeben durch die Anzahl der Symbole aus s^1 bzw. s^2 , die bisher aliniert wurden.

Um dem Alignment die Sequenzsymbole s_i^1 und s_j^2 hinzufügen, existieren drei Möglichkeiten.

1. Man baut s_i^1 und s_j^2 als Match oder Mismatch ein.
2. Man baut s_i^1 und einen Gap anstatt s_j^2 ein.
3. Man baut s_j^2 und einen Gap anstatt s_i^1 ein.

Die optimale Wahl für diesen Schritt ergibt sich, durch den Vergleich der Werte aller drei Möglichkeiten. Da Symbole nicht mehrfach verwendet werden können, sind sie nach dem Einbau "konsumiert".

Für das Alignment zweier Sequenzen s^1 und s^2 mit Längen M und N definiert man nun rekursiv eine Matrix F , bei der Element f_{ij} den Wert des optimalen Alignments der Subsequenzen (s_1^1, \dots, s_i^1) und (s_1^2, \dots, s_j^2) darstellt. Folglich ist f_{MN} der Wert des optimalen globalen Alignments, da dieses beide Sequenzen auf voller Länge gegeneinander ausrichtet.

Für s^1 und s^2 sei $(f_{ij}) = F \in \mathbb{R}^{M+1 \times N+1}$. Gegeben seien Kosten für Matches w_{match} , Missmatches w_{miss} und Gaps w_{gap} , sei außerdem w_{ij} wie in (2.1.2)

F hat eine Zeile für jedes Symol in s^1 und eine Spalte für jedes Symbol in s^2 und zusätzlich eine nullte extra Zeile und nullte extra Spalte.

Die nullte Zeile und Spalte dienen als Rekursionsanker.

$$f_{i0} = w_{\text{gap}} \cdot i \quad f_{0j} = w_{\text{gap}} \cdot j \quad (2.1.4)$$

Für die restlichen Elemente gilt die folgende Rekursionsbeziehung:

$$f_{ij} = \max \begin{cases} f_{i-1,j-1} + w_{ij} & , \text{entspricht Match oder Mismatch} \\ f_{i-1,j} + w_{\text{gap}} & , \text{entspricht Gap in } s_j^2 \\ f_{i,j-1} + w_{\text{gap}} & , \text{entspricht Gap in } s_i^1 \end{cases} \quad (2.1.5)$$

Die Ursprungsrichtung hat dabei eine klare Interpretation im Kontext der Sequenzalinierung. Diagonale Schritte entsprechen Matches oder Missmatches, während horizontale und vertikale Schritte Gaps anzeigen.⁵ Daher ist es wichtig, dass man sich bei der Berechnung merkt, auf welchem Vorgängerwert f_{ij} beruht.

Anhand des so entstehenden Pfades kann aus der befüllten Matrix das optimale Alignment rekonstruiert werden. Dafür werden üblicherweise Pfeile eingezeichnet, welche auf die Vorgängerezelle⁶ zeigen.

Jeder Schritt durch die Matrix entspricht einer Position im Alignment. Aufgrund des Aufbaus der Matrix ist die Symbolanordnung gewahrt. Alle Symbole werden genau einmal genutzt und jede Position im Alignment bekommt höchstens eine Symbolzuweisung.

⁵ Zeilen- und Spaltenindizes sind mit den Sequenzsymbolen verknüpft. Bei diagonalen Bewegung erhöhen sich beide Sequenzindizes und es werden Symbole beider Sequenzen genutzt. Im Gegensatz dazu werden bei horizontaler bzw. vertikaler Bewegung nur Symbole einer Sequenz verbraucht.

⁶ Falls mehrere Vorgänger denselben Wert liefern sind beide Alignments gleichwertig und wir können entweder alle Möglichkeiten behalten oder uns für eine bestimmte entscheiden. Die Entscheidung für eine von mehreren Möglichkeiten sollte deterministisch sein, um reproduzierbare Ergebnisse zu garantieren.

2.1.6 Funktionale Programmierung

Die funktionale Programmierung baut auf dem von Alonzo Church entwickelten^[^lambda_calc] Lambda-Kalkül auf, welches, analog zur bekannteren Turing-Maschine, ein mathematisches Modell für den Begriff der Berechenbarkeit formuliert. [13] Alan Turing, dessen Doktorvater Church war, zeigte die Äquivalenz beider Modelle. [14]

In funktionalen Sprachen werden Funktionen deklariert, welche nicht nur miteinander verknüpft (komponiert) und aufeinander angewandt werden, sondern auch anderen Funktionen als Werte übergeben oder von diesen zurückgegeben werden.

Funktionen im Sinne der funktionalen Programmierung bilden Bäume von Ausdrücken. Dies steht im Kontrast zu den Funktionen in der klassischen imperativen Programmierung, die sequentiell Anweisungen ausführen, um den Programmzustand zu ändern.

2.1.6.1 Haskell In dieser Arbeit nutzen wir Haskell, um eine Lösung für das Sequenzalinierungsproblem zu implementieren. Haskell ist eine **nicht strikte** und **pur-funktionale** Sprache mit **call-by-name** Semantik.

2.1.6.1.1 Pur-funktionale Sprachen In "puren" funktionalen Sprachen sind Funktionen deterministisch, d. h. gleiche Eingaben produzieren immer gleiche Ausgaben und frei von Nebeneffekten, wie z. B. Änderungen an globalen Zuständen wie Variablen.

Funktionen in pur-funktionalen Sprachen verhalten sich also wie mathematische Funktionen, welche ihre Funktionsargumente deterministisch auf Funktionswerte abbilden.

Dies hat bestimmte praktische Konsequenzen für Haskell.

Alle Werte in Haskell sind Funktionen. Bspw. ist der Wert `1` eine Funktion, welche keine Argumente nimmt und auf sich selbst abbildet.

Werte in Haskell sind immutabel. Wenn ein Wert in einem bestimmten Kontext an einen Namen gebunden wurde, dann kann der Name in diesem Kontext nicht anderweitig belegt werden.

Sabry formalisiert den Begriff in [15] und definiert Sprachen als pur-funktional, wenn sie die folgenden Kriterien erfüllen:

- A language is purely functional if
- (i) it includes every simply typed λ -calculus term, and
- (ii) its call-by-name, call-by-need, and call-by-value implementations are equivalent (modulo divergence and errors).

2.1.6.1.2 Nicht-strikte Auswertung In einer nicht-strikten⁷ funktionalen Sprache werden Ausdrücke nur dann ausgewertet, wenn sie zur Berechnung eines Ergebnisses notwendig sind. Man spricht auch von *call-by-need* oder *lazy* Evaluierung.

Dies ermöglicht den Umgang mit undefinierten Ausdrücken. Der Wert einer rekursiven Funktion für eine Eingabe, bei der die Berechnung nicht terminiert, ist nicht definiert.

⁷ Das Gegenstück ist die strikte Auswertung, welche auch als *eager* Evaluierung bezeichnet wird.

Beispiel 2.3

Als praktisches Beispiel betrachten wir den Umgang mit unendlich langen Listen.

Der Ausdruck `map (+1) [1..]` (A) wendet die Nachfolgerfunktion^a auf alle natürlichen Zahlen an. Die resultierende Liste wäre unendlich lang und damit ist das Ergebnis dieser Berechnung nicht definiert.

Allerdings ist es möglich, mit `take 10 (map (+1) [1..])` (B) die ersten 10 Einträge dieser Liste zu berechnen. Da nur die Teile des Ausdrucks A ausgewertet werden, welche für die Berechnung von B notwendig sind, terminiert die Berechnung und der Ausdruck ist wohldefiniert.

^a Die Nachfolgerfunktion $S : \mathbb{N} \rightarrow \mathbb{N}$ bildet natürliche Zahlen auf ihren Nachfolger ab $S : n \mapsto n + 1$.

2.1.6.1.3 Call-by-name Semantik In Sprachen mit **call-by-name** werden Funktionsargumente bei Übergabe nicht ausgewertet, sondern deren Definitionen im Funktionskontext substituiert.

Wenn das Argument mehrfach verwendet wird, muss es potentiell bei jedem Vorkommen neu berechnet werden. Der GHC Compiler optimiert den Code zwar, um solche doppelten Berechnungen zu vermeiden, aber dies ist nicht durch Haskell's Sprachsemantik garantiert.

Beispiel 2.4

Betrachten wir die Funktion `sumTwice`, welche die Summe einer übergebenen Liste berechnet und diese mit sich selbst addiert.

```
sumTwice :: [Int] -> Int
sumTwice ns = nsum + nsum
  where
    nsum = sum ns
```

Nutzen wir nun den Ausdruck `map (+1) [1..20]` als Argument für `sumTwice`, dann wird folgendermaßen evaluiert:

1. Definition des Arguments einsetzen

```
sumTwice (map (+1) [1..20]) = nsum + nsum
  where
    nsum = sum (map (+1) [1..20])
```

2. Definition von `nsum` einsetzen

```
sumTwice (map (+1) [1..20])
  = (sum (map (+1) [1..20])) + (sum (map (+1) [1..20]))
```

2.1.7 Verifikation

Bei der Verifikation eines Programmes soll gezeigt werden, dass ein Programm seine Spezifikation⁸ erfüllt.

⁸ Eine Spezifikation ist die formale Beschreibung der gültigen Eingaben, des zu erwartenden Ergebnisses und den mit diesen zusammenhängenden Invarianten.

2.1.7.1 Tests Ein typischer Ansatz, um dies sicherzustellen, ist das Schreiben von Funktionstests. Dabei werden anhand der Spezifikation Äquivalenzklassen von Ein- und Ausgaben gebildet, die sich ähnlich verhalten. Die Gesamtheit der Äquivalenzklassen deckt nicht nur valide Eingaben und Ergebnisse, sondern auch die existierenden Randfälle und Fehlermodi ab.

Beispiel 2.5

Sei bspw. eine Funktion `double sqrt(double x)` definiert, welche einen `double` Wert als Eingabe nimmt und die Quadratwurzel dieses Werts als `double` zurückgibt.

Da der Rückgabewert `double` ist, sind die Ergebnisse beliebiger negativer Eingaben undefiniert und führen zu Fehlern. Also bilden alle negativen Eingaben eine Äquivalenzklasse für diesen bestimmten Fehlermodus.

Aus diesem Grund können wir das Verhalten der Funktion in solchen Fällen genauso gut durch Testen der Eingabe `-1`, wie `-2` absichern.

Das Problem mit diesem Ansatz ist, dass nicht mit Sicherheit gesagt werden kann, ob alle relevanten Äquivalenzklassen korrekt identifiziert wurden. Diese Klassen zu identifizieren hat viel mit Kreativität und Intuition zu tun und ist nicht leicht überprüfbar oder reproduzierbar.

Weiterhin könnte jemand, der alle möglichen Fehlermodi eines Programms kennt und auch weiß, wie man diese erkennt, einfach ein korrektes Programm schreiben.⁹

2.1.7.2 Verifikation durch Beweis Es ist nachvollziehbar, dass das Testen von Programmen alleine nicht ausreicht, um deren Korrektheit zu zeigen.

Ein alternatives Vorgehen für die Verifikation eines Programmes besteht darin, Aussagen über bestimmte Eigenschaften des Programmes mathematisch zu beweisen.

2.1.7.2.1 Begriffe Klären wir zunächst ein paar wichtige Begriffe.

Aussagen sind Sätze, denen man (prinzipiell) einen Wahrheitswert zuordnen kann. Ein Beispiel für eine Aussage ist "es hat heute geregnet", keine Aussage wiederum ist "dieser Satz ist falsch".

Aussageformen oder **Prädikate** sind Ausdrücke mit Variablen, die nach Einsetzen der Variablen zu Aussagen werden. Die Formel $3 \cdot x = 9$, welche durch Setzen von $x = 3$ zu einer wahren und sonst einer falschen Aussage wird ist ein Beispiel für eine Aussageform. Die Begriffe Aussage und Aussageform werden häufig synonym genutzt.

Junktoren sind logische Operatoren, welche verschiedene (Teil-)Aussagen zu neuen Aussagen verbinden. Das logische UND $p \wedge q$ stellt ein Beispiel für ein Junktor dar.

Schlussregeln erlauben es, Aussagen umzuformen und so andere Aussagen abzuleiten. Schlussregeln werden notiert, indem die Vorbedingungen oberhalb eines Striches und die äquivalenten Folgerungen unterhalb dessen notiert werden. Ein Beispiel für eine Schlussregel ist die Kontraposition.

$$\frac{p \rightarrow q}{\neg q \rightarrow \neg p}$$

^{9†} Man könnte fast sagen, "wer gute Tests schreiben kann, braucht sie nicht". Das ist natürlich polemisch, aber enthält zumindest ein Körnchen Wahrheit.

2.1.7.2.2 Beweistechniken Um Aussagen über Programme zu beweisen existieren, verschiedene gängige Beweistechniken. Im Folgenden betrachten wir eine Auswahl solcher Verfahren.

Hoare-Kalkül

CAR Hoare entwickelte in 1969 [16] ein Kalkül, um Aussagen *für imperative Programme* zu beweisen. Da unsere Implementation nicht in einer imperativen Sprache geschrieben wurde, werden wir uns hier nur am Rande damit beschäftigen.

Im Hoare-Kalkül werden **Hoare-Tripel** der Form $\{P\}C\{Q\}$ betrachtet. Bei P und Q handelt es sich um Zusicherungen und bei C um einen Befehl. Die Zusicherungen lassen sich als logische Aussagen betrachten und P betrifft einen Zustand vor Ausführung von C und Q einen anschließenden Zustand.

Wenn das Hoare-Tripel korrekt ist, dann folgt aus P nach Ausführung von C die Aussage Q . Hoare hat Axiome und Schlussregeln für imperative Sprachkonstrukte wie z. B. Zuweisungen, Schleifen etc. definiert, mit denen die Korrektheit von Hoare-Tripeln geprüft werden kann.

Abgleich

Haskell hat eine Syntax, die stark an klassisch mathematische Notation angelehnt ist. Dies erlaubt es in vielen Fällen, die mathematische und die programmatische Definition einer Funktion einfach abzugleichen.

Beispiel 2.6

Sei beispielsweise $\text{Col} : \mathbb{N} \rightarrow \mathbb{N}$ die Collatzfunktion.

$$\text{Col}(n) = \begin{cases} \frac{n}{2} & , n \text{ ist gerade} \\ 3n + 1 & , \text{Andernfalls.} \end{cases}$$

Sei weiterhin `col` die folgende Haskell-Implementation für `Col`.

```
col :: Int -> Int
col n
  | even n = div n 2
  | otherwise = 3 * n + 1
```

Wir sehen, dass die Bedingungen und die daraus folgenden Ergebnisse in den Funktionen identisch sind, woraus folgt, dass `Col` und `col` identisch sein müssen.^a

^a Strenggenommen nimmt die Funktion `col` Eingaben vom Typ `Int`, also \mathbb{Z} an. Sie wird für Eingaben in $\mathbb{Z} \setminus \mathbb{N}$ allerdings nicht terminieren, weswegen ihr Ergebnis in diesen Fällen auch nicht definiert ist. Weiterhin gilt für alle Werte $n \in \mathbb{N}$ dass $\text{Col}(n) = \text{col}(n)$.

Wenn wir zeigen können, dass die Funktionen unseres Programms identisch zu denen unseres Modells definiert sind, dann haben wir gezeigt, dass unser Programm das Modell akkurat implementiert.

Induktionsbeweise

Mithilfe der (mathematischen) Induktion können Beweise von Aussagen für alle Elemente einer abzählbar unendlichen Menge¹⁰ geführt werden.

¹⁰ Unendliche Mengen sind abzählbar, wenn eine Bijektion $f : M \rightarrow \mathbb{N}$ zwischen den Elementen der Menge und den natürlichen Zahlen existiert.

Dabei beweist man eine Aussage zunächst für einen Basisfall¹¹ und zeigt anschließend, dass aus der Annahme,¹² die Aussage gelte für einen beliebigen Wert, folgt, dass sie auch für den Nachfolger dieses Wertes gilt.¹³

Rekursion definiert Basisfälle als Rekursionsanker und führt Eingaben mithilfe von Rekursionsschritten auf diese Basisfälle zurück. Wir sehen also, dass es einen engen Zusammenhang zwischen Induktion und Rekursion gibt.

Aus diesem Grund sollte es uns nicht verwundern, dass sich die induktive Beweistechnik auf beliebige rekursiv definierte Strukturen, so wie z.B. Listen oder Bäume, erweitern lässt, um zu zeigen, dass eine Aussage für alle Elemente der Struktur gilt. Man spricht in diesen Fällen von **struktureller** oder **noetherscher**¹⁴ **Induktion**.

Sei M eine rekursiv definierte Menge, mit einer Menge von Schlussregeln zur Bildung von M , welche sowohl Basis-, als auch rekursive Fälle abdecken. Diese Schlussregeln besitzen die folgende Form:¹⁵

$$\begin{array}{c} \frac{Q(m)}{F(m) \in M} \\ \text{Bildungsregeln für Basisfälle} \end{array} \qquad \begin{array}{c} \frac{\bigwedge_{i=1}^n m_i \in M \quad R(m_1, \dots, m_n)}{G(m_1, \dots, m_n) \in M} \\ \text{Bildungsregeln für rekursive Fälle} \end{array}$$

Hierbei bezeichnet m bestimmte Elemente, welche im Basisfall beliebig sind und im rekursiven Fall aus M kommen, Q und R sind Prädikate mit Bedingungen die wir an diese m Stellen und F und G sind Abbildungen von den m auf die eigentlichen Elemente in M .¹⁶

¹¹ Der sog. **Induktionsanker**.

¹² Die sog. **Induktionshypothese**.

¹³ Der sog. **Induktionsschritt**.

¹⁴ So benannt nach Emmy Noether, der ersten Frau in Deutschland mit Dokortitel in der Mathematik, ersten weiblichen Professorin in Deutschland und Schülerin von David Hilbert. Noether spezifizierte u.a. den Begriff der wohlfundierten Relation, welcher die theoretische Basis der Technik liefert.

¹⁵ Strenggenommen sind auch Regeln für gemischt-rekursive Fälle mit n Elementen in M und k beliebigen anderen Elementen möglich. Der Umgang mit diesen erfordert keine besondere Vorsicht. Das Schema für derartige Bildungsregeln sähe in etwa so

aus:
$$\frac{\bigwedge_{i=1}^n m_i \in M \quad R'(m_1, \dots, m_n, m_{n+1}, \dots, m_{n+k})}{G'(m_1, \dots, m_n, m_{n+1}, \dots, m_{n+k}) \in M}$$

¹⁶ Man kann sich M als einen "Typen" und die Abbildungen F, G als "Konstruktoren" für M vorstellen.

Beispiel 2.7

Betrachten wir als Beispiel den Datentypen **Tree** für Binärbäume.

data **Tree** = **Branch** **Tree** **Tree** | **Leaf** **Int**

Der Typ **Tree** ist rekursiv definiert. Ein **Tree** ist entweder ein **Leaf**, mit einem Wert von Typ **Int**, oder ein **Branch**, an dem zwei weitere **Tree** Werte hängen. Wir können die beiden Typenkonstruktoren **Leaf** und **Branch** als Funktionen betrachten, welche Werte von Typ **Int** oder **Tree** auf einen Wert vom Typ **Tree** abbilden.

$$\begin{aligned}\text{Leaf} : \quad & \mathbb{Z} \rightarrow \text{Tree} \\ \text{Branch} : \quad & \text{Tree} \times \text{Tree} \rightarrow \text{Tree}\end{aligned}$$

Nun können wir mithilfe von Schlussregeln darstellen, aus welchen Werten die Menge **Tree**, die alle Werte des Typen **Tree** enthält, besteht.

$$\frac{n \in \mathbb{Z}}{\text{Leaf}(n) \in \text{Tree}} \qquad \frac{\begin{array}{l} a \in \text{Tree} \\ b \in \text{Tree} \end{array}}{\text{Branch}(a, b) \in \text{Tree}}$$

Wenn wir eine Aussage $P(e)$ für alle Elemente $e \in M$ einer derart definierten Menge beweisen wollen, dann modifizieren wir zunächst die gebildeten Schlussregeln, indem wir alle *Folgerungen* $e \in M$ durch $P(e)$ ersetzen¹⁷ und zu jeder *Voraussetzung* $e \in M$ zusätzlich $P(e)$ fordern.¹⁸ Wenn wir die Gültigkeit der modifizierten Schlussregeln beweisen¹⁹ können, dann haben wir $\forall e \in M : P(e)$ gezeigt.

¹⁷ Das Ersetzen entspricht dem Setzen des Induktionsankers.

¹⁸ Diese Forderung entspricht der Annahme der Induktionshypothese.

¹⁹ Der Beweis entspricht dem Induktionsschritt.

Beispiel 2.8

Betrachten wir wieder das Beispiel mit `Tree`.

Definieren wir nun die Funktionen `size` und `depth`, welche einerseits die Anzahl der Knoten und andererseits die Tiefe eines Baumes berechnen.

```
size :: Tree -> Int
size (Leaf _) = 1
size (Branch a b) = 1 + size a + size b

depth :: Tree -> Int
depth (Leaf _) = 1
depth (Branch a b) = 1 + max (size a) (size b)
```

Wir behaupten jetzt, dass ein Baum immer mindestens so viele Knoten wie Ebenen hat, also dass mit $P(t) = \text{size}(t) \geq \text{depth}(t)$ gilt, dass $\forall t \in \text{Tree} : P(t)$. Nachdem wir die initialen Schlussregeln umformen, bekommen wir:

$$\frac{n \in \mathbb{Z}}{\text{size}(\text{Leaf}(n)) \geq \text{depth}(\text{Leaf}(n))} \quad \frac{\begin{array}{l} a \in \text{Tree} \\ \text{size}(a) \geq \text{depth}(a) \\ b \in \text{Tree} \\ \text{size}(b) \geq \text{depth}(b) \end{array}}{\text{size}(\text{Branch}(a, b)) \geq \text{depth}(\text{Branch}(a, b))}$$

Wenn wir beide Schlussregeln beweisen können sind wir fertig.

Aus den Funktionsdefinitionen folgt $\text{size}(\text{Leaf}(n)) = 1 = \text{depth}(\text{Leaf}(n))$ und damit $\text{size}(\text{Leaf}(n)) \geq \text{depth}(\text{Leaf}(n))$, womit die erste Regel bewiesen ist.

Nehmen wir an, dass o.B.d.A. $\text{depth}(a) \geq \text{depth}(b)$, dann haben wir Folgendes:

$$\begin{array}{lcl} \text{size}(\text{Branch}(a, b)) & = & 1 + \text{size}(a) + \text{size}(b) \\ \text{depth}(\text{Branch}(a, b)) & = & 1 + \max\{\text{depth}(a), \text{depth}(b)\} \\ \\ 1 + \text{size}(a) + \text{size}(b) & \geq & 1 + \max\{\text{depth}(a), \text{depth}(b)\} \quad | \quad -1 \\ \text{size}(a) + \text{size}(b) & \geq & \max\{\text{depth}(a), \text{depth}(b)\} \quad | \quad \text{mit } \text{depth}(a) \geq \text{depth}(b) \\ \text{size}(a) + \text{size}(b) & \geq & \text{depth}(a) \end{array}$$

Mit der Induktionshypothese und $\text{size}(t) \geq 1$ für beliebige t , folgt, dass auch die zweite Schlussregel gilt und daher $\forall t \in \text{Tree} : \text{size}(t) \geq \text{depth}(t)$. ■

Haskell erlaubt auch die Definition parametrisierter Datentypen. Z. B. ein könnte mittels des folgenden Datentyps ein `Tree Int` oder `Tree String` dargestellt werden.

```
data Tree a = Branch (Tree a) (Tree a) | Leaf a
```

Das beschriebene Schema lässt sich analog auf solche polymorphen Typen erweitern.

2.2 Praktische Hintergründe

Folgende Informationen sind relevant für den praktischen Umgang mit den erzielten Arbeitsergebnissen, speziell die Generation von Quellcode und Dokumentation aus den Projektquellen.

2.2.1 Pandoc

Die Quellen für die Projektdokumentation²⁰ sind in Markdown verfasst und liegen im docs Ordner. Die Hauptdatei hat den Namen `bachelors_thesis.md`.

2.2.1.1 Konfiguration Um die Dokumentation zu bauen, wird das Programm [Pandoc](#) verwendet. Die Konfiguration befindet sich im `assets/pandoc` Ordner und alle Einstellungen sind in der `assets/pandoc/defaults.yaml` Datei zusammengefasst, welche Pandoc mit der `-d` Flag übergeben werden kann.

2.2.1.2 Filter Da die Konfiguration Filter zur Vorverarbeitung der Dokumentationsquellen anwendet, müssen diese vorher installiert werden. Es handelt sich dabei um Python-Programme, welche in der `pyproject.toml` bzw. der `requirements.txt` hinterlegt sind.

Wir gehen davon aus, dass wir auf einem UNIX-System mit einer Bourne-artigen Shell arbeiten und ein aktuelles²¹ Python 3 unter dem Namen `python` verfügbar ist. In diesem Fall kann mit den folgenden Befehlen eine virtuelle Umgebung mit den notwendigen Abhängigkeiten angelegt und betreten werden.

```
$ python -m venv venv
$ source venv/bin/activate
$ python -m pip install -r requirements.txt
```

2.2.1.3 Schriftarten Die Datei `assets/pandoc/yaml/fonts.yaml` spezifiziert die Nutzung bestimmter Fonts. Wenn diese nicht installiert sind, wird die Dokumentengeneration scheitern.

Wenn die genutzten Fonts verändert werden, sollte der genutzte monofont trotzdem die im Text verwendeten Glyphen unterstützen. Deswegen bietet sich dafür die Nutzung eines [NerdFonts](#) an.

2.2.1.4 Generation Wenn die notwendigen Filter und Schriftarten verfügbar sind, kann, aus dem Wurzelordner des Projekts heraus, die Dokumentation mit Pandoc als PDF-Datei generiert werden.

```
$ pandoc -d assets/pandoc/defaults.yaml docs/bachelors_thesis.md -o
↪ bachelors_thesis.pdf
```

Wobei `lang` durch den entsprechenden Sprachcode ersetzt werden muss.

2.2.2 Haskell

Um den Haskell-Code auszuführen, müssen die [array](#),²² [matrix](#) und [fasta](#) Pakete installiert werden.

Diese Abhängigkeiten sind bereits in der Projektkonfiguration hinterlegt, sodass sie beim Bauen der Software oder der Installation als Bibliothek automatisch aufgelöst und mitinstalliert werden.

Um das Projekt in Haskell's interaktivem REPL zu laden, ist ggf. eine manuelle Installation der Pakete vonnöten. Diese kann mit Cabal durchgeführt werden.

²⁰ Bzw. für die vorliegende Bachelorarbeit.

²¹ Mit Version ≥ 3.7 .

²² Für Haskell 2010 ist `array` zwar Teil der Standardbibliothek, aber es handelt sich um ein "verstecktes" Paket, weswegen wir es trotzdem explizit installieren.

```
$ cabal update && cabal install --lib array matrix fasta
```

2.2.3 Entangle

Als Tool für die Arbeit mit unseren literate Programming Quellen nutzen wir [Entangled](#).

Dies erlaubt es, mit dem `tangle` Befehl, Codeblöcke aus den Markdown- in die entsprechenden Haskell-Dateien zu schreiben und umgekehrt, mit dem `stitch` Befehl, Änderungen an den Haskell-Dateien zurückzuschreiben. Mit `watch` wendet Entangled die `tangle` und `stitch` Befehle automatisch an, sobald Änderungen an Projektdateien vorgenommen wurden.

Die Konfiguration für Entangled befindet sich in der `pyproject.toml` Datei.

3 Durchführung und Ergebnisse

Der Hauptteil der Arbeit besteht aus vier Sektionen. Im Folgenden werden wir:

1. das Problem der paarweisen Sequenzalinierung als gemischt ganzzahliges Optimierungsproblem formulieren,
2. besprechen, mit welchen Techniken das formulierte Modell gelöst werden kann,
3. ein Softwaresystem zur paarweisen Sequenzalinierung entwickeln, und
4. zeigen, dass dieses System korrekt implementiert ist.

Da die Sektionen aufeinander aufbauen, gehen Durchführung und Ergebnisse fließend ineinander über, weswegen wir auf die übliche Trennung von Durchführungs- und Ergebnisteil verzichten.

3.1 Sequenzalinierung als Optimierungsproblem

In dieser Sektion entwickeln wir, basierend auf [17], ein mathematisches Modell welches Sequenzalinierung als gemischt-ganzzahliges Optimierungsproblem (*Mixed Integer Linear Program* oder **MILP**) darstellt.

Da das Modell auf zuvor publizierter Arbeit basiert, wird auf ausführliche Beweise der Korrektheit verzichtet. Die Berechnung der Kosten für Mismatches und Gaps wurde vereinfacht und Notation benutzt, die an die Arbeit von Althaus et al. in [18] angelehnt ist.

3.1.1 Templates

Die zentrale Struktur, unseres Modells ist das "Template". Das Alignment einer Menge verschiedener Sequenzen $S = \{s^1, \dots, s^{|S|}\}$ wird durch eine Templatematrix T , bestehend aus $|S|$ Zeilen und K Spalten, angegeben.

Hierbei werden jeder Zeile m des Templates Symbole aus der entsprechenden Sequenz s^m , oder Gaps zugewiesen.

Beim Befüllen des Templates entsteht eine Anordnung, bei der die zueinander gehörigen Symbole verschiedener Sequenzen jeweils in derselben Spalte k stehen.

$$(t_k^m) = T \quad t_k^m = s_i^m, \text{ mit unbekanntem } i \quad (3.1.1)$$

Aufgrund des Zusammenhangs zwischen Zeile m und Sequenz s^m , stellen wir den Zeilenindex für Templatematrizen hoch und sprechen von t^m als dem "Alignment der Sequenz s^m ", oder, je nach Kontext, von t_k^m oder t_k als der "Position" k im Alignment.

Alignmentalphabet

Für Sequenzen über einem Alphabet Σ werden Lücken im Alignment durch ein spezielles Symbol²³ $c_{\text{gap}} \notin \Sigma$ gekennzeichnet. Daraus folgt, dass $\bar{\Sigma} = \Sigma \cup \{c_{\text{gap}}\}$ das Alphabet ist, über dem Alignments definiert sind, also $T \in \bar{\Sigma}^{|S| \times K}$.

Templatelängen entsprechen der Anzahl von Spalten des Templates.

Damit wir die Templatelänge K bestimmen können, muss eine maximale Anzahl erlaubter Gaps $g_{\text{max}} \in \mathbb{N}$ festgelegt sein. Dann ergibt sich K als Summe von g_{max} und der Länge der längsten Sequenz.

²³ Üblicherweise $c_{\text{gap}} = -$.

$$K = \max \{|s| \mid s \in S\} + g_{\max} \quad (3.1.2)$$

Beispiel 3.1

Seien bspw. zwei Sequenzen $s^1 = \text{AGTAC}$ und $s^2 = \text{ATGC}$ über dem Alphabet $\Sigma = \{A, C, G, T\}$ gegeben, das Gapsymbol als $c_{\text{gap}} = -$ definiert und höchstens ein Gap in der längsten Sequenz erlaubt, also $g_{\max} = 1$.

In diesem Fall wäre das Alignmentalphabet $\bar{\Sigma}$ durch die Menge $\{A, C, G, T, -\}$ gegeben und die Templatelänge wäre $K = g_{\max} + |s^1| = 6$.

Ein mögliches Alignment der Sequenzen wäre das folgende:

$$T = \begin{array}{c|c|c|c|c|c} A & G & T & A & C & - \\ \hline A & - & T & G & C & - \end{array}$$

3.1.2 Problemformulierung

Seien $S = \{s^1, s^2\}$ die betrachteten Sequenzen, mit $|s^1| = M, |s^2| = N$ wobei o.b.d.A. gelte, dass $M \leq N$, sei g_{\max} die erlaubte Anzahl an Gaps und sei T ein Template der entsprechenden Länge K . Wir möchten nun T mit Sequenzsymbolen und Gaps befüllen, sodass wir ein optimales globales Alignment erhalten.

Dabei müssen wir **jedes Sequenzsymbol**, in der **richtigen Reihenfolge** und an **genau einer** Position im Template zuweisen. Positionen, die nicht mit Sequenzsymbolen befüllt wurden, werden mit dem Gapsymbol versehen.

Entsprechend dem üblichen Vorgehen bei der Formulierungsproblemen, formulieren wir notwendige Variablen, Beschränkungen und eine Zielfunktion für das Alignmentproblem. Dabei versuchen wir anschaulich zusammenzufassen und zu interpretieren.

3.1.2.1 Variablen Um ein Template zu befüllen, muss bekannt sein welchen Stellen im Template die Symbole der zu alinierenden Sequenzen zugewiesen werden sollen. Dabei handelt es sich um eine Erweiterung des klassischen Zuordnungsproblems.²⁴

3.1.2.1.1 Zuweisungen Da es sinnlos ein Symbol "teilweise" zuzuweisen, sind Zuweisung also offensichtlich binär, d.h. die Zuweisungsvariablen liegen in \mathbb{B} .

Um zu kodieren, ob ein Sequenzsymbol s_i^m einer bestimmten Templateposition t_k^m zugewiesen wurde, müssen m, i und k einbezogen werden. Wir benötigen eine Variable für jede Kombination von Sequenz, Symbol- und Templateposition.

Sei $a_{ij}^m \in \mathbb{B}$ eine Zuweisungsvariable mit folgender Bedeutung:

$$a_{ik}^m = \begin{cases} 1, & s_i^m \text{ wird } T \text{ an Position } t_k^m \text{ zugewiesen} \\ 0, & \text{Andernfalls} \end{cases} \quad (3.1.3)$$

²⁴ Für eine allgemeine Diskussion des Zuordnungsproblems vgl. [19], pp. 5-6.

Alle Zuweisungsvariablen der Sequenz s^m , mit Länge M in ein Template der Länge K können zu einer "Zuweisungsmatrix" $\mathcal{A}^m \in \mathbb{B}^{M \times K}$ zusammengefasst werden.²⁵

$$\mathcal{A}^m = (a_{ik}^m) \quad (3.1.4)$$

Die Zuweisungsmatrix \mathcal{A}^m bezieht sich also eindeutig auf die Sequenz s^m .²⁶

Dabei haben Zeilen- und Spaltenindex eine klare Interpretation. Der Zeilenindex i entspricht der Position des Symbols in der Sequenz s^m und der Spaltenindex k der Spalte im Template t_k^m .

Beispiel 3.2

Die Befüllung des zuvor gegebenen Beispieltemplates $\begin{smallmatrix} \text{A} & \text{G} & \text{T} & \text{A} & \text{C} \\ \text{A} & - & \text{T} & \text{G} & \text{C} \end{smallmatrix}$ mit Sequenzsymbolen wird durch die folgenden Assignmentmatrizen dargestellt:

$$\mathcal{A}^1 = \begin{pmatrix} 1 & 0 & 0 & 0 & 0 & 0 \\ 0 & 1 & 0 & 0 & 0 & 0 \\ 0 & 0 & 1 & 0 & 0 & 0 \\ 0 & 0 & 0 & 1 & 0 & 0 \\ 0 & 0 & 0 & 0 & 1 & 0 \end{pmatrix} \quad \mathcal{A}^2 = \begin{pmatrix} 1 & 0 & 0 & 0 & 0 & 0 \\ 0 & 0 & 1 & 0 & 0 & 0 \\ 0 & 0 & 0 & 1 & 0 & 0 \\ 0 & 0 & 0 & 0 & 1 & 0 \end{pmatrix}$$

Wobei der Zusammenhang zwischen \mathcal{A}^1 , s^1 und t^1 folgendermaßen interpretiert werden kann:

\mathcal{A}^1	t_1^1	t_2^1	t_3^1	t_4^1	t_5^1	t_6^1
s_1^1	1	0	0	0	0	0
s_2^1	0	1	0	0	0	0
s_3^1	0	0	1	0	0	0
s_4^1	0	0	0	1	0	0
s_5^1	0	0	0	0	1	0

3.1.2.1.2 Gaps Ebenso wollen wir die Menge G aller Lücken im Alignment bestimmen. Die Lücken im Alignment der Sequenz s^m lassen sich in Zuweisungsmatrix $(a_{ik}^m) = \mathcal{A}^m \in \mathbb{B}^{N \times K}$ an den Nullspalten ablesen.

$$G = \left\{ i \mid \sum_{k=1}^K a_{ik}^m = 0 \right\}$$

Auf dieser Grundlage können wir eine Hilfsvariable g_k^m einführen, die aussagt, ob Sequenz s^m an Templateposition t_k^m eine Lücke zugewiesen wurde.

²⁵ Aufgrund der potentiell unterschiedlichen Länge der betrachteten Sequenzen ist es nicht möglich die Zuweisungsmatrizen weiter zu einem einzelnen Tensor zusammenzufassen.

²⁶ Was den hochgestellten Index erklärt.

$$g_k^m = \left[\sum_{i=1}^K a_{ik}^m = 0 \right] \quad (3.1.5)$$

3.1.2.1.3 Zusammenfassung Zu jeder Sequenz s^m , haben wir Zuordnungsvariablen in Form einer Zuweisungsmatrix $(a_{ik}^m) = \mathcal{A}^m$ definiert. Auf Basis der Zuweisungsmatrix bestimmen wir die Stellen g_k^m im Template, denen Lücken zugewiesen werden.

3.1.2.2 Beschränkungen Nun wollen wir die Bedingungen festlegen, welchen Alignments genügen müssen, damit wir sie als sinnvoll erachten. Bspw. ist es sinnlos, Symbole einer Sequenz mehrfach, oder in ungeordneter Reihenfolge, zuzuweisen.

Es ist klar, dass bestimmte Zuweisungen von Symbolen keinen Sinn ergeben. Dies können wir als Beschränkung der Zuweisungsmatrizen \mathcal{A} formulieren.

3.1.2.2.1 Nutzung der Symbole Wir möchten weder erlauben, dass ein Symbol mehrfach zugewiesen wird, noch dass es gar nicht zugewiesen wird. Jedes Symbol soll also genau einmal zugewiesen werden.

Wir erinnern uns an die Interpretation der Zuweisungsmatrix $(a_{ik}^m) = \mathcal{A}^m \in \mathbb{B}^{M \times K}$ und sehen, dass die **Summe von Zeile i** die **Anzahl aller Zuweisungen des Symbols s_i^m** beschreibt.

$$\forall i \in J_{s^m} : \sum_{k=1}^K a_{ik}^m = 1 \quad (3.1.6)$$

Da jedes Symbol genau einmal zugewiesen werden soll, muss jede Zeilensumme genau 1 ergeben.

3.1.2.2.2 Belegung des Templates Weiterhin dürfen wir jeder Stelle im Template höchstens ein Sequenzsymbol zuweisen. Da aber möglicherweise Lücken bestehen, ist es auch zulässig, einer bestimmten Stelle im Template kein Sequenzsymbol zuzuweisen.

Anhand der Interpretation der Zuweisungsmatrix $(a_{ik}^m) = \mathcal{A}^m \in \mathbb{B}^{M \times K}$ sehen wir, dass die **Summe von Spalte k** der **Anzahl aller dem Template an dieser Position zugewiesenen Sequenzsymbole** entspricht.

$$\forall k \in J_K : \sum_{i=1}^M a_{ik}^m \leq 1 \quad (3.1.7)$$

Da jeder Stelle in T höchstens ein Sequenzsymbol zugewiesen werden soll, folgt, dass alle Spaltensummen kleiner oder gleich 1 sein müssen.

3.1.2.2.3 Reihenfolge der Symbole Die Zuweisung der Sequenzsymbole zum Template muss auch an der Reihenfolge der Symbole in der Sequenz erhalten. D. h. ein Symbol s_i^m darf dem Template nicht nach dem Symbol s_j^m zugewiesen werden, wenn $i < j$, bzw. für zwei beliebige Indizes i, j von s^m muss gelten, dass wenn $i < j$, die Position von s_i^m in T , gegeben durch k , kleiner der Position von s_j^m in T , gegeben durch k' , ist.

$$\forall i, j \in J_N : i < j \implies k < k'$$

Wenn also $i < j$ und $a_{ik}^m = 1$, dann muss für alle $k' \leq k$ gelten, dass $a_{jk'}^m = 0$. Umgekehrt muss auch, wenn $a_{jk'}^m = 1$, gelten, dass $a_{ik}^m = 0$. Da höchstens einer der Terme den Wert 1 annehmen kann, muss auch die Summe beider Zuweisungen unter 1 bleiben.

$$\forall i, j \in J_M, k, k' \in J_K : i < j \wedge k' \leq k \implies a_{ik}^m + a_{jk'}^m \leq 1$$

Beispiel 3.3

Was würde eine falsche Reihenfolge in der Praxis bedeuten?

Betrachten wir wieder die Sequenzen $s^1 = \text{AGTAC}$ und $s^2 = \text{ATGC}$ aus dem vorigen Beispiel und das Alignment $T = \begin{smallmatrix} \text{A} & \text{G} & \text{T} & \text{A} & \text{C} & - \\ - & \text{G} & \text{T} & \text{A} & \text{C} & - \end{smallmatrix}$ indem wir die Zuordnung der Symbole $s_1^2 = \text{A}$, $s_2^2 = \text{T}$ und $s_3^2 = \text{G}$ in der falschen Reihenfolge vorgenommen haben.^a

Die Zuordnungen von s^2 zu T sind durch die folgende Zuweisungsmatrix gegeben.

$$\mathcal{A}^2 = \begin{pmatrix} 0 & 0 & 0 & 1 & 0 & 0 \\ 0 & 0 & 1 & 0 & 0 & 0 \\ 0 & 1 & 0 & 0 & 0 & 0 \\ 0 & 0 & 0 & 0 & 1 & 0 \end{pmatrix}$$

Wir sehen, dass die führenden Spalten nicht mehr geordnet sind, wodurch \mathcal{A}^2 nicht in reduzierter Zeilen-Stufen-Form vorliegt.

^{a†} Wir sehen, dass T offensichtlich ein besseres Alignment darstellt als die zuvor besprochene korrekt angeordnete Alternative $\begin{smallmatrix} \text{A} & \text{G} & \text{T} & \text{A} & \text{C} & - \\ \text{A} & - & \text{T} & \text{G} & \text{C} & - \end{smallmatrix}$. Diese Ordnung verletzt keines der zuvor formulierten Kriterien, weswegen es notwendig ist solche Neuankordnungen explizit auszuschließen.

Beziehung zu Nachfolger

Statt beliebige Sequenzindizes i und j zu vergleichen, reicht es, wenn wir die Beziehung zum direktem Nachfolger betrachten. Der führende Eintrag in Zeile i von \mathcal{A}^m muss auf der linken Seite des führenden Eintrags in Spalte $i + 1$ stehen.

Dies ist offensichtlich, lässt sich aber durch strukturelle Induktion über \mathcal{A}^m explizit zeigen.

Sei k_i der Index des führenden Eintrags²⁷ von Spalte i , sodass $a_{ik_i}^m = 1$ und $a_{jk_i}^m = 0$ für $j \neq i$, dann erhalten Templatezuweisungen die Ordnung der Sequenz, g.d.w. aus $i < j$ folgt, dass $k_i < k_j$.²⁸

$$\forall i, j \in J_M : i < j \implies k_i < k_j$$

Dies ist äquivalent zu der Aussage, dass $(k_i)_{i \in J_M}$ streng monoton steigt.

Als Induktionsanker sehen wir, dass leere Folgen und solche mit Länge 1 immer geordnet sind.

Nehmen wir jetzt als Induktionshypothese an, dass die Teilfolge bis Index m geordnet ist. Wenn wir einen Eintrag k_n hinzufügen, der größer als k_m ist, dann folgt aus der I.H. und der Transitivität von $<$, dass k_n auch größer als alle k_l mit $l < m$ ist und somit, dass auch die Folge bis n geordnet ist. ■

^{27†} Aus der Bedingung für Zeilensummen ergibt sich, dass ein solches k_i existieren muss.

^{28†} Aus $a_{ik_i}^m = 1$ i.V.m. der Bedingung für Spaltensummen ergibt sich $j \neq i \implies a_{jk_i}^m = 0$.

Damit bekommen wir die folgende Bedingung.

$$\forall i, i+1 \in J_M, k, k' \in J_K, k' \leq k : a_{ik}^m + a_{i+1,k'}^m \leq 1 \quad (3.1.8)$$

Anmerkung

Wir erinnern uns, dass für $(a_{ik}^m) = \mathcal{A}^m \in \mathbb{B}^{M \times K}$ die **Summe der Zeile i** der Zuweisungsmatrix der **Anzahl der Zuweisungen** des Sequenzsymbols s_i^m entspricht, und dass die **Spalte k** die **Position t_k^m im Template** codiert. Daher können wir die **Summe** der Elemente in Zeile i , **bis Spalte k als Anzahl an Zuweisungen von s_i^m vor der Position t_k^m** interpretieren.

Da wir voraussetzen, dass jedes Sequenzsymbol genau einmal zugewiesen wird kann die Anzahl aller Zuweisungen von s_{i+1}^m vor k höchstens 1 sein. Die Summe aller solcher Zuweisungen entspricht dem Term $\sum_{k'=1}^{k-1} a_{i+1,k'}^m$. Wie zuvor, sehen wir, dass auch die Summe dieses Terms und a_{ik}^m unter 1 bleiben muss.

$$\forall i, j \in J_M, k \in J_K : i < j \implies a_{ik}^m + \sum_{k'=1}^{k-1} a_{i+1,k'}^m \leq 1$$

Dies wäre eine noch stärkere Bedingung als die zuvor formulierte.

3.1.2.2.4 Zusammenfassung Wir haben verschiedene Beschränkungen für unsere Variablen formuliert.

- (3.1.6) Jedes Symbol der Sequenz s^m muss zugewiesen werden.
- (3.1.7) Jeder Stelle t_k im Template darf höchstens ein Symbol der Sequenz s^m zugewiesen werden.
- (3.1.8) Die Sequenzsymbole in s^m müssen in der richtigen Reihenfolge zugewiesen werden.

Diese Beschränkungen geben unseren Daten Struktur. Fassen wir nochmal kurz zusammen:

Die Assignmentmatrizen \mathcal{A}^1 und \mathcal{A}^2 haben pro Zeile genau einen und pro Spalte höchstens einen Eintrag ungleich Null. Darüber hinaus haben \mathcal{A}^1 und \mathcal{A}^2 reduzierte Zeilenstufenform.

Außerdem gibt es keine Beschränkungen, die sich auf mehrere Zuweisungsmatrizen gleichzeitig beziehen, d.h. sie sind sauber voneinander separierbar.

3.1.2.3 Zielfunktion Um ein *optimales* globales Alignment zu berechnen, muss ein Bewertungskriterium für die Güte von Alignments existieren.

3.1.2.3.1 Kostenmodell Zunächst wählen wir ein Kostenmodell, um anschließend eine darauf basierende Zielfunktion zu formulieren.

Im Rahmen dieser Arbeit verwenden wir eine flache Kostendarstellung, mit den drei Kostenfaktoren w_{match} , w_{miss} und w_{gap} , welche für Matches, Mismatches und Gaps respektive gelten.

Anmerkung

Man kann auf Basis von w_{match} , w_{miss} und w_{gap} ohne großen Aufwand eine Substitutionsmatrix W mit Einträgen für jedes Symbol $c_i \in \Sigma$ definieren. In W steht w_{match} auf der Hauptdiagonale und w_{miss} überall sonst.

$$W = \begin{array}{c|ccccc} & c_{\text{gap}} & c_1 & c_2 & \dots & c_n \\ \hline c_{\text{gap}} & 0 & w_{\text{gap}} & w_{\text{gap}} & \dots & w_{\text{gap}} \\ c_1 & w_{\text{gap}} & w_{\text{match}} & w_{\text{miss}} & \dots & w_{\text{miss}} \\ c_2 & w_{\text{gap}} & w_{\text{miss}} & w_{\text{match}} & \ddots & \vdots \\ \vdots & \vdots & \vdots & \ddots & \ddots & w_{\text{miss}} \\ c_n & w_{\text{gap}} & w_{\text{miss}} & \dots & w_{\text{miss}} & w_{\text{match}} \end{array}$$

Wir schreiben w_{ik} wie in (2.1.3)

Für Leser mit Interesse an komplexeren Kostenmodellen sei erneut auf die Arbeit von McAllister et al. [17] verwiesen, die sowohl affine, als auch konkave Gapkosten im Rahmen einer MILP-Formulierung modelliert.

3.1.2.3.2 Substitutionskosten Die totalen Substitutionskosten von T entsprechen den Substitutionskosten aller Symbole s_i^1 und s_j^2 , die derselben Position t_k zugewiesen wurden für alle Positionen.

Diese Zuweisungen sind durch die in (3.1.3) formulierten Variablen bzw. den zusammengefassten Matrizen \mathcal{A}^1 und \mathcal{A}^2 kodiert. Wenn sowohl s_i^1 , als auch s_j^2 der Position t_k zugewiesen wurden, dann gilt $a_{ik}^1 \cdot a_{jk}^2 = 1$.²⁹

Da die Multiplikation zweier Unbekannter zu einem nicht linearen Term führt, definieren wir eine Hilfsvariable ϕ_{ijk} .³⁰

$$\phi_{ijk} = a_{ik}^1 \cdot a_{jk}^2 \quad (3.1.9)$$

Dies ermöglicht es die Substitutionskosten als Summe über die Sequenz- und Templatepositionen darzustellen.

$$\sum_{i=1}^M \sum_{j=1}^N \sum_{k=1}^K [w_{ij} \cdot \phi_{ijk}] \quad (3.1.10)$$

Wobei w_{ij} wie in (2.1.2) definiert ist.

3.1.2.3.3 Gapkosten Gapkosten entstehen für alle Positionen, denen ein Gap bzw. kein Symbol zugewiesen wurde. Diese sind durch die in (3.1.5) formulierte und von \mathcal{A}^m abhängige Hilfsvariable g_k^m gegeben.

²⁹ Dies ist ein schönes Beispiel dafür, warum die Multiplikation die algebraische Interpretation des logischen UND ist.

³⁰ Wir könnten diese Hilfsvariable zu einem Tensor $(\phi_{ijk}) = \Phi \in \mathbb{B}^{M \times N \times K}$ zusammenfassen.

Information

Eine naive Formulierung könnte auf der Frage aufbauen, ob das Sequenzalignment t_k^1 oder t_k^2 einen Gap an Stelle k hat. Das logische ODER kann algebraisch als $+$ dargestellt werden, was uns die Auswahl von Gaps mit $g_k^1 + g_k^2$ ermöglicht. Diese Variante würde folgendermaßen aussehen:

$$\sum_{k=1}^K w_{\text{gap}} \cdot (g_k^1 + g_k^2)$$

Die Formulierung ist u.a. deswegen naiv, da wir nicht über dem Feld \mathbb{F}_2 arbeiten und folglich $1 + 1 = 2$ anstatt 1 ist.^a In diesem Ausdruck werden "beidseitige" Gaps also doppelt gezählt.

Durch die Wahl von g_{max} , kann regelmäßig ein ungenutztes Kontingent an Gaps verbleiben, welches zu solchen beidseitigen Gaps führen kann, die aber keinen Einfluss auf das Alignment haben sollten.

In Kombination mit bestimmten Gewichten kann dies zu sinnlosen Ergebnissen führen.

Beispiel 3.4

Machen wir ein Beispiel mit $w_{\text{match}} = 0$, $w_{\text{miss}} = 2$, $w_{\text{gap}} = 3$. Vergleichen wir nun $T_1 = \begin{smallmatrix} c & - \\ c & - \end{smallmatrix}$ und $T_2 = \begin{smallmatrix} - & c \\ - & c \end{smallmatrix}$. Wir sehen intuitiv, dass T_1 das bessere Alignment ist, jedoch ergibt sich mit den Kosten für den doppelten Gap der Wert 6. Die beiden einzelnen Gaps in T_2 verursachen dieselben Kosten, weswegen diese naive Formulierung keinen Unterschied zwischen T_1 und T_2 machen würde.

^a Auch ein einfaches Zählen von Doppelgaps wäre nicht perfekt, sollte bei sinnvoll gewählten Parametern für w_{gap} und w_{miss} allerdings nicht zu dem im Folgenden beschriebenen Fehlermodus führen.

Um das beschriebene Problem doppelter Gapkosten zu vermeiden, führen wir die Hilfsvariable γ_k ein.

$$\gamma_k = |g_k^1 - g_k^2| \quad (3.1.11)$$

Diese entspricht einem logischen XODER (exklusives ODER) und gibt an, ob genau eine Sequenz einen Gap in Position t_k besitzt.³¹

Die Summe der Gapkosten aller Positionen, denen genau ein Gap zugewiesen wurde ist durch folgenden Ausdruck gegeben:

$$\sum_{k=1}^K [w_{\text{gap}} \cdot \gamma_k] \quad (3.1.12)$$

3.1.2.3.4 Gesamtkosten Man kann eine Zielfunktion c' formulieren, deren Argumente die Zuweisungsvariablen a_{ik}^1 und a_{jk}^2 darstellen.

³¹ Diese können wir zu einem Vektor $\gamma \in \mathbb{B}^K$ zusammenfassen.

Da diese allerdings nicht linear ist, nutzen wir stattdessen die von den Zuweisungsvariablen abgeleiteten ϕ_{ijk} und γ_k , um eine äquivalente³² Funktion c zu formulieren.

Die Zielfunktion ist die Summe aller Substitutionskosten, die in (3.1.10) formuliert wurde und der Summe aller Gapkosten, die in (3.1.12) formuliert wurde.

$$\sum_{i=1}^M \sum_{j=1}^N \sum_{k=1}^K [w_{ij} \cdot \phi_{ijk}] + \sum_{k=1}^K [w_{\text{gap}} \cdot \gamma_k]$$

Dies kann man äquivalent umformulieren.

$$\sum_{k=1}^K \left[w_{\text{gap}} \cdot \gamma_k + \left[\sum_{i=1}^M \sum_{j=1}^N w_{ij} \cdot \phi_{ijk} \right] \right] \quad (3.1.13)$$

3.1.2.4 Gesamtdarstellung Wenn (3.1.13) maximal ist, ist per Definition der Wert des durch die Eingaben kodierten Alignments maximal.

Somit können wir unser Optimierungsproblem Z , basierend auf den Variablen, Beschränkungen und der Zielfunktion darstellen.

$$Z = \max \sum_{k=1}^K \left[w_{\text{gap}} \cdot \gamma_k \left[\sum_{i=1}^M \sum_{j=1}^N w_{ij} \cdot \phi_{ijk} \right] \right] \quad (3.1.14)$$

3.2 Lösung des Optimierungsproblems

In dieser Sektion formulieren wir einen Lösungsansatz für das Problem.

Um ein Problem $Z = \max\{c(x) \mid x \in X \subseteq \mathbb{Z}^n\}$ optimal zu lösen, müssen wir eine Eingabe $x^* \in X$ finden, bei der es nicht mehr möglich ist, den gefundenen Wert zu verbessern. Wir wollen zeigen, dass x^* auf den größten Funktionswert abbildet.

$$\forall x \in X : c(x) \leq c(x^*)$$

Um unser Problemmodell zu lösen, gibt es verschiedene Ansätze.

3.2.1 Ausprobieren

Die offensichtliche naive Strategie besteht darin alle validen Lösungen durchzuprobieren und sich die beste Eingabe zu merken.

Bei $|s^1| = M$ und $|s^2| = N$, gibt es $M! \cdot N!$ Permutationen alleine für die Sequenzsymbole. Mit Gaps wird der Lösungsbereich noch größer. Im Allgemeinen gibt es bei einem Template der Länge $K = \max\{M, N\} + g_{\text{max}}$ genau $(t!)^2$, mögliche Symbolzuordnungen.

Bei $t \approx 35$ gibt es mit $(35!)^2 \approx (10^{40})^2 = 10^{80}$ ungefähr so viele Lösungen wie Atome im sichtbaren Universum. Dieser Ansatz wird nicht funktionieren, aber kann man ihn verbessern?

Der Lösungsraum lässt sich mithilfe der zuvor formulierten Beschränkungen verkleinern. Diese erlauben es, ganze Klassen von Lösungen direkt aus der Betrachtung auszuschließen.

³² D.h. $c'(x') = c(x) \iff x' \implies x$.

Bspw. sind Lösungen, welche die Sequenzreihenfolge durcheinander bringen, nicht valide.

3.2.1.1 Intuition Es ist klar, dass \mathcal{G} eindeutig durch \mathcal{A}^1 und \mathcal{A}^2 festgelegt ist und umgekehrt. Daher reicht es, entweder die validen Belegungen der Zuweisungsmatrizen, oder der Gapmatrix zu betrachten.

Da die Reihenfolge der Sequenzsymbolzuweisungen vorgegeben ist, aber Gaps an beliebigen Stellen eingebaut werden können, ist die Anzahl der möglichen Belegungen einer Assignmentmatrix auch von der Anzahl der Gaps bestimmt. Die Anzahl an Gaps, die wir in das Alignment der Sequenz einbauen, entspricht der Differenz zwischen Sequenz- und Templatelänge.

Wir wissen, dass $\mathcal{A}^1 \in \mathbb{B}^{M \times K}$, $\mathcal{A}^2 \in \mathbb{B}^{N \times K}$, wobei die Zeilenzahl der Sequenz- und die Spaltenzahl der Templatelänge entspricht. Für \mathcal{A}^1 ergibt sich $\Delta_{g_1} = K - M$ und für \mathcal{A}^2 analog $\Delta_{g_2} = K - N$ als Anzahl von Gaps, bzw. Leerspalten.

Beispiel 3.5

Arbeiten wir weiter mit den Sequenzen aus unserem vorigen Beispiel, aber nehmen wir an, dass wir noch keine feste Belegung haben.

Wenn wir alle Zeilen, außer der letzten, belegt und bisher keine Leerspalten eingebaut haben, dann müssen alle verbleibenden Leerspalten in dieser Zeile Verwendung finden.

$$\begin{pmatrix} 1 & 0 & 0 & 0 & 0 & 0 \\ 0 & 1 & 0 & 0 & 0 & 0 \\ 0 & 0 & 1 & 0 & 0 & 0 \\ 0 & 0 & 0 & - & - & - \end{pmatrix}$$

Die Bindestriche symbolisieren valide Belegungen, von denen **genau eine** auf eins gesetzt werden muss. Die Anzahl an Möglichkeiten das Sequenzsymbol s_4^2 zu positionieren ist dann $(6 - 4) + 1 = 3$, bzw. $\Delta_g + 1$.

Wenn in den vorigen Spalten bereits g Leerspalten verwendet wurden, sinkt dieser Wert entsprechend. Angenommen wir bauen einen Gaps in t_3^2 und t_4^2 ein, dann ändern sich auch die möglichen Belegungen.

$$\begin{pmatrix} 1 & 0 & 0 & 0 & 0 & 0 \\ 0 & 1 & 0 & 0 & 0 & 0 \\ 0 & 0 & \mathbf{0} & \mathbf{0} & 1 & 0 \\ 0 & 0 & 0 & 0 & 0 & - \end{pmatrix}$$

Die Anzahl an Möglichkeiten sinkt auf $\Delta_g + 1 - g$. In diesem Fall haben wir $\Delta_g = g$ und damit $\Delta_g + 1 - g = 1$ nur eine mögliche Belegung.

Vorherige Gaps schränken spätere Belegungsmöglichkeiten ein. Mithilfe dieser Erkenntnis können wir alle validen Möglichkeiten durchnummerieren, ähnlich, wie wir es in einer Wahrheitstafel machen würden.

Dazu können wir mit der erweiterten Einheitsmatrix $[I_M \ 0]$ beginnen und wiederholt die Position in der letzten "beweglichen" Zeile nach rechts schieben, bis keine Bewegung mehr möglich ist. Anschließend rücken wir die Zeile darüber eins weiter und setzen alle nachfolgenden Zeilen auf die erste valide Position. Wir sind fertig, wenn keine Bewegung mehr möglich ist.

3.2.1.2 Verteilung von Gaps Da die Reihenfolge der Sequenzsymbole im Template unverändert bleibt, unterscheiden sich Alignments nur durch die Platzierung von Gaps. Wenn wir wissen, an welchen Stellen im Template Gaps stehen, können wir also das Alignment vollständig bestimmen.

Nehmen wir an, dass wir in eine Sequenz s , mit Länge M genau k Gaps einbauen wollen. D.h. wir wollen von M Positionen im Template k auswählen, wobei uns die Reihenfolge der Gaps nicht interessiert. Das entspricht allen ungeordneten Stichproben, bzw. Kombinationen, mit Länge k aus einer Menge mit M Elementen. Die Anzahl der k -Kombinationen aus dieser Menge ist durch den Binomialkoeffizienten $\binom{n}{k}$ gegeben.

3.2.1.2.1 Enumeration aller Kombinationen Wie nummerieren wir diese allerdings?

Wir können eine Kombination c auch als k -Tupel (c_1, \dots, c_k) betrachten, dessen Elemente aus J_n stammen und bei denen alle Elemente strikt kleiner als ihr Nachfolger sind.

$$C_k^n = \{(c_1, \dots, c_k) \mid c_i \in J_n, c_i < c_{i+1}, 1 \leq i < k\}$$

Anordnung der Kombinationen

Auf C_k^n können wir nun eine lexikografische Ordnung definieren.

Zwei Kombinationen c und c' sind genau dann gleich, wenn alle ihre Elemente gleich sind.

$$c = c' \iff \forall i \in J_k : c_i = c'_i$$

Weiterhin ist c genau dann kleiner als c' , wenn die erste Stelle, in der sie sich voneinander unterscheiden, kleiner ist.

$$c < c' \iff \exists i : c_i < c'_i \wedge \forall i' < i : c_{i'} = c'_{i'}$$

Die erste Kombination c^1 und letzte Kombination $c^{\binom{n}{k}}$ in dieser Folge, sind durch $c^1 = (1, 2, \dots, k)$, bzw. $c^{\binom{n}{k}} = (n - k, n - k + 1, \dots, n)$ gegeben.

Information

Nehmen wir an, dass es eine Sequenz c^0 mit $c^0 < c^1$ gibt, dann muss es gemäß Definition einen ersten Index i geben, in dem sich c^0 von c^1 unterscheidet. Da $c_i^1 = i$, folgt $c_i^0 < i$.

Wir wissen, dass im Allgemeinen $c_i < c_{i+1}$ und daher im Speziellen $c_1^0 < \dots < c_{i-2}^0 < c_{i-1}^0 < c_i^0 < i$. Da $i, c_j^0 \in \mathbb{N}$, ist die größtmögliche Zahl c_{i-1}^0 die $c_{i-1}^0 < i$ erfüllt durch $c_{i-1}^0 = i - 1$ gegeben. Daraus folgt $c_1^0 = 0$ als höchstmöglicher Startwert für c^0 , was im Widerspruch zu $c_i^0 \in J_n$ steht. ■

Der Beweis für $c^{\binom{n}{k}}$ funktioniert analog.

Nachfolger einer Kombination

Der Nachfolger $S(c)$ einer Kombination c wird dadurch erzeugt, dass wir die letzte Stelle i inkrementieren, die sich inkrementieren lässt, ohne dass wir C_k^n verlassen.

$$i = \max\{i' \mid c_{i'} + 1 \in J_n, (i' = n \vee c_{i'} < c_{i'+1})\} \quad S(c) = (c_1, \dots, c_i + 1, c_i + 2, \dots)$$

Eine Position c_i kann minimal und maximal diese Werte annehmen: $i \leq c_i \leq n - k + i$.

Wenn $c_i < n - k + i$, dann können wir an Stelle i inkrementieren, und falls $c_i = n - k + i$, dann interessiert uns die vorige Position c_{i-1} . Wenn $c_i = n - k + i$ und $i = 1$, dann gibt es keine nachfolgende Kombination.

Nachdem an Stelle i inkrementiert wurde, müssen die nachfolgenden Stellen zurückgesetzt werden. Wenn c' die auf c folgende Kombination ist, die an Stelle i inkrementiert wurde, mit $c'_i = c_i + 1$ ist, dann sind die folgenden Stellen gegeben durch $c'_{i+j} = c'_i + j$. Wir können zeigen, dass dies die kleinste Subkombination sein muss, indem wir dieselbe Logik wie im Beweis für die kleinste Kombination nutzen.

Anmerkung

Todo: Ausführen

Diese Kombinationen haben Ähnlichkeiten mit M -ären Zahlensystemen und Polynomen mit beschränkten Koeffizienten.

Mit Sequenzen der Länge M , bzw. M und Template Länge K sind alle möglichen Kombinationen von Gapverteilungen durch die Menge $C_{K-M}^K \times C_{K-N}^K$ gegeben.

$$|C_{K-M}^K \times C_{K-N}^K| = \binom{K}{K-M} \cdot \binom{K}{K-N} = \binom{K}{M} \cdot \binom{K}{N}$$

Das ist schon deutlich besser, da wir nun auch deutlich längere Sequenzen verarbeiten können. Bspw. kommen wir mit $|s^1| = |s^2| = 750$ und $g_{\max} = 20$ auf $\binom{750}{20}^2 \approx 10^{78}$ Kombinationen, also ganze zwei Größenordnungen weniger als die Anzahl der Atome im sichtbaren Universum.

3.2.2 Needleman-Wunsch

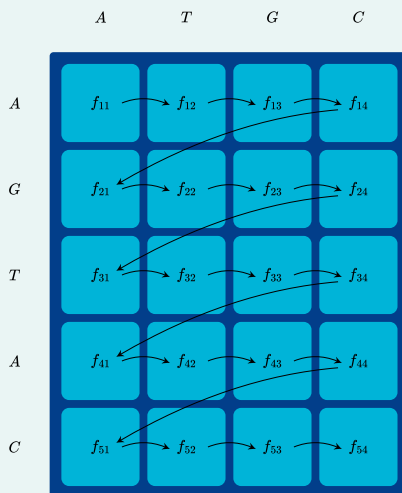
Es ist offensichtlich, dass der naive Ansatz nicht gut genug ist. Wie können wir diese riesige Menge an möglichen Lösungen sinnvoll händeln? Eine Möglichkeit ist es, Methoden der dynamischen Programmierung zu nutzen.

Im Kontext der Sequenzalinierung bietet sich da insbesondere der Algorithmus von Needleman-Wunsch (NW) an.

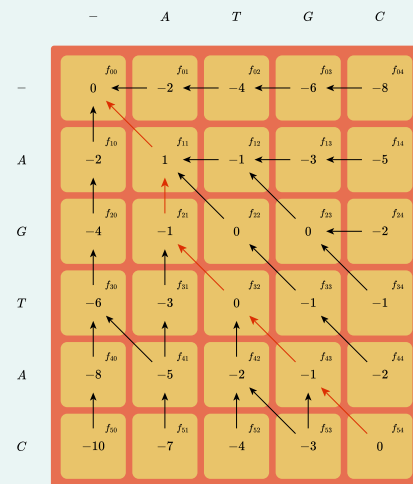
Beispiel 3.6

Als kurze Auffrischung betrachten wir NW anhand eines praktischen Beispiels.

Für die Sequenzen $s^1 = \text{AGTAC}$ und $s^2 = \text{ATGC}$ des Beispieltemplates mit $w_{\text{match}} = 1$, $w_{\text{miss}} = -1$ und $w_{\text{gap}} = -2$ werden zunächst die Rekursionsanker bestimmt, woraufhin die restliche Matrix Zeile für Zeile ausgefüllt werden kann. Die Reihenfolge der Berechnungen und eine entsprechende Matrix F , mit rot eingezeichnetem optimalen Pfad sähe z.B. so aus:



(a) Reihenfolge der Berechnungen



(b) Ausgefüllte NW-Matrix

Abbildung 2: Reihenfolge der Berechnungen und ausgefüllte NW-Matrix.

Wenn wir nach Befüllen der Matrix, von $f_{5,4}$ ausgehend, den optimalen Pfad zurückverfolgen ergibt sich das Alignment $\begin{smallmatrix} A & G & T & A & C \\ A & - & T & G & C \end{smallmatrix}$ mit Wert 0. Würden wir uns für das optimale Alignment der Teilsequenzen AGT und ATGC interessieren, könnten wir bei $f_{3,4}$ beginnen und erhielten das mit -1 gewertete Alignment $\begin{smallmatrix} A & G & T \\ A & T & G & C \end{smallmatrix}$. AGTAC und ATG können als $\begin{smallmatrix} A & G & T & A & C \\ A & - & T & G & - \end{smallmatrix}$ oder $\begin{smallmatrix} A & G & T & A & C \\ A & - & T & - & G \end{smallmatrix}$ gleichwertig mit einem Wert von -3 aligniert werden.

Allerdings möchten wir nicht einfach irgendwelche Matrizen befüllen, sondern unser MILP-Modell lösen. Können wir NW hinsichtlich Modells umformulieren?

Äquivalenz von NW und MILP

Wir behaupten, dass NW³³ eine Lösung für das initial formulierte MILP Problem findet. Dies wollen wir mithilfe der folgenden Schritte zeigen.

1. Reinterpretation der NW-Ergebnisse, Definition der Begriffe "Pfad" und "Schritt" und aufstellen der durch NW maximierten Zielfunktion mit optimalem Wert Z_P .
2. Anpassung des NW-Algorithmus hinsichtlich fester, aus g_{\max} resultierenden, Alignmentlängen.
3. Herleitung der Entscheidungsvariablen a_{ik}^m , bzw. der daraus resultierenden Variablen ϕ_{ijk} , g_k und γ_k , aus einem, durch NW gefundenen, Pfad P .

³³ Bzw. eine angepasste Variante von NW.

4. Beweis dass die für das MILP formulierten Beschränkungen auch für die neue Formulierung hinsichtlich NW gilt.
5. Herleitung der Äquivalenz von Funktion des Pfadwertes und Zielfunktion des MILP, woraus $Z_P = Z$ folgt.

3.2.2.1 Reinterpretation der Ergebnisse Versuchen wir nun, dies in die Sprache unseres initialen Optimierungsproblems zu übersetzen.

Sei F eine Matrix für das Alignment der Sequenzen s^1 und s^2 mit Längen M , bzw. N

$$(f_{ij}) = F \in \mathbb{R}^{M+1 \times N+1} \quad (3.2.1)$$

Und sei C_F die Menge der Zellkoordinaten von F .

$$C_F = \{0, 1, \dots, M\} \times \{0, 1, \dots, N\} \quad F \in \mathbb{R}^{M+1 \times N+1} \quad (3.2.2)$$

3.2.2.1.1 Pfade Ein **Pfad** $P = (p_0, \dots, p_K)$ durch F ist eine Folge benachbarter Zellen in F , dargestellt durch deren Koordinaten (i, j) .

$$\begin{aligned} P &= (p_i)_{i \in \{0\} \cup J_K} & P &\in C_F^{K+1} \\ &= (p_0, \dots, p_K) \end{aligned} \quad (3.2.3)$$

Komponenten

Wir bezeichnen die m -te Indexkomponente i^m von Pfadstück $p_k = (i^1, \dots, i^{|S|}) \in \mathbb{N}_0^{|S|}$ als p_k^m diese entspricht der Position in Sequenz s^m .³⁴

Start und Ende

Pfade beginnen immer im Ursprung $(0, 0)$ und enden mit den Koordinaten der Sequenzlängen (M, N) .

$$\forall p : p_0 = (0, 0) \wedge p_K = (M, N) \quad (3.2.4)$$

Im Allgemeinen mit Sequenzen $S = \{s^1, \dots, s^L\}$ gilt für den Ursprung p_0 , dass alle Komponenten den Wert null haben und für das letzte Pfadstück p_K , dass die Werte aller Komponenten den Sequenzlängen entsprechen.

$$\begin{aligned} p_0 &= \vec{0} & p_0 &\in \{0\}^L \\ p_K &= \bigtimes_{i=1}^L \{|s^i|\} & p_K &\in \mathbb{N}^L \end{aligned}$$

Unterschiede zwischen Nachbarn

Weiterhin gilt für beliebige aufeinanderfolgende $p_{k-1} = (p_{k-1}^1, \dots, p_{k-1}^n)$ und $p_k = (p_k^1, \dots, p_k^n)$ in P die Aussage $p_{k-1} \neq p_k$, dass also mindestens eine unterschiedliche Indexkomponente zwischen p_{k-1} und p_k existiert.

$$\forall p_{k-1}, p_k \in P \exists m \in J_L : p_{k-1}^m \neq p_k^m \quad (3.2.5)$$

³⁴ Auch hier weist der Superskript-Index von p_i^m auf den Zusammenhang mit dem Sequenzsymbol s_i^m hin.

und dass die Elemente im Nachfolger entweder denselben Wert haben, oder genau eins größer sind,

$$\forall p_{k-1}, p_k \in P, m \in J_L : p_{k-1}^m \vee p_k^m = p_{k-1}^m + 1 \quad (3.2.6)$$

woraus direkt $p_{k-1}^m \leq p_k^m$ für beliebige $p_{k-1}, p_k \in P$ folgt.

$$\forall p_{k-1}, p_k \in P : i_{k-1}^m \leq i_k^m \quad (3.2.7)$$

3.2.2.1.2 Schritte Ein Schritt q_k durch einen Pfad ist durch das Paar der zwei aufeinanderfolgenden Koordinaten (p_{k-1}, p_k) gegeben.

$$q_k = (p_{k-1}, p_k) \quad p_{k-1}, p_k \in P \quad (3.2.8)$$

Herkunft und Ziel

Wir schreiben für die Elemente von $q_k = (p_{k-1}, p_k)$ abkürzend $o_k = p_{k-1}$ für Herkunft und $d_k = p_k$ für Ziel und analog zur Notation der Pfadelemente bezeichnen wir deren Komponenten mit hochgestellten Indizes.

$$q_k = (p_{k-1}, p_k) = (o_k, d_k) = ((o_k^1, \dots, o_k^{|S|}), (d_k^1, \dots, d_k^{|S|})) \quad (3.2.9)$$

Schritte überlappen

Im Allgemeinen gilt für beliebige benachbarte Schritte $q_{k-1} = (p_{k-2}, p_{k-1}), q_k = (p_{k-1}, p_k)$ durch P , dass der nächste Schritt dort beginnt, wo der vorige aufgehört hat.

$$d_{k-1} = o_k \quad (3.2.10)$$

Schrittweiten und -richtungen

Aus den in (3.2.5) und (3.2.6) formulierten Regeln für benachbarte Elemente in Pfaden folgen analog Regeln für die Weite von Schritten. Nämlich gilt für beliebige $q_k = ((g, h), (i, j))$ entweder $i = g \wedge j = h + 1$ oder $i = g + 1 \wedge j = h$ oder $i = g + 1 \wedge j = h + 1$.

$$\forall q_k = ((g, h), (i, j)) : \dot{\vee} \begin{cases} i = g + 1 \wedge j = h + 1 & , \text{Diagonale} \\ i = g + 1 \wedge j = h & , \text{Vertikale} \\ i = g \wedge j = h + 1 & , \text{Horizontale} \end{cases} \quad (3.2.11)$$

In anderen Worten verlaufen Schritte entweder diagonal, vertikal oder horizontal. Dafür können wir für beliebige Schritte q_k von (g, h) nach (i, j) die folgenden Aussageformen definieren:

$$\begin{aligned} \text{diag}(q_k) &= (i = g + 1 \wedge j = h + 1) \\ \text{vert}(q_k) &= (i = g + 1 \wedge j = h) \\ \text{hori}(q_k) &= (i = g \wedge j = h + 1) \end{aligned} \quad (3.2.12)$$

Mit (3.2.11) sehen wir, dass aus Gleichheit und Ungleichheit von Indexkomponenten die in (3.2.11) definierten Prädikate folgen.

$$\begin{aligned}
\text{diag}(q_k) &\iff (i \neq g \wedge j \neq h) \\
\text{vert}(q_k) &\iff (i \neq g \wedge j = h) \\
\text{hori}(q_k) &\iff (i = g \wedge j \neq h)
\end{aligned} \tag{3.2.13}$$

Schrittmonotonie

Aus (3.2.11) folgt analog zu (3.2.7) dass Schritte nur benachbarte Zellen mit monoton höherem Index erreichen können.

$$\forall q_k : o_k^m \leq d_k^m \tag{3.2.14}$$

3.2.2.1.3 Gewichte Der Wert eines Schrittes q_k von (g, h) nach (i, j) ergibt sich aus dem Term, der in (2.1.5) auf den Wert des Vorgängerkandidaten f_{gh} dazuaddiert wird um den Wert f_{ij} zu bestimmen.

$$w(q_k) = \begin{cases} w_{ij}, & \text{diag}(q_k) \\ w_{\text{gap}}, & \text{Andernfalls} \end{cases} \tag{3.2.15}$$

Der Wert Z_P von Pfad P entspricht der Summe aller Schrittwerte $q_k = (p_{k-1}, p_k)$ in P , bzw. $\sum_{k=1}^K w(p_{k-1}, p_k)$. Da NW bei der Konstruktion von F den optimalen Pfad P durch F findet, ergibt sich für Z_P die folgende Formel.

$$Z_P = \max \sum_{k=1}^K w(q_k) \tag{3.2.16}$$

3.2.2.2 Anpassung für feste Alignmentlängen Bevor wir damit beginnen die Äquivalenz von MILP und NW zu zeigen, müssen wir über die unterschiedlichen Längen der Alignments bei der MILP-Methode mit Templates und NW sprechen.

Bei MILP wählen wir die Variable g_{\max} , aber bei NW nicht. Die Bedeutung von g_{\max} liegt darin, dass es die Anzahl der in das Alignment eingebauten Gaps bestimmt und dadurch die Alignmentlänge K festlegt.

Für s^1 mit Länge M und s^2 mit Länge N beträgt Templatelänge gem. (3.1.2) $K = \max\{M, N\} + g_{\max}$. Wenn, o.B.d.A. $M > N$, dann werden also in das Alignment t^1 von s^1 genau g_{\max} und in das Alignment t^2 von s^2 genau $g_{\max} + M - N$ Gapsymbole eingebaut.

Im Allgemeinen werden in das Alignment t^m einer Sequenz $s^m \in S$ genau g^m Gaps eingebaut. Sei $L = \max\{|s| : s \in S\}$ die Länge der längsten betrachteten Sequenz.

$$g^m = g_{\max} + (L - |s^m|) \tag{3.2.17}$$

Es kann passieren, dass die Templatelänge bei unserem MILP-Modell, im Vergleich zu den durch NW produzierten Alignments, eingeschränkt oder erweitert wird.

Anmerkung

Die Anzahl möglicher Schritte K durch F wird von unten mit $\max\{M, N\}$ durch die Chebyshev-^a und von oben mit $M + N$ durch die Manhattanndistanz^b eingegrenzt.

$$\max\{M, N\} \leq K \leq M + N$$

^a Wie ein König über das Schachfeld läuft.

^b Nur horizontale und vertikale Schritte sind gestattet.

Falls MILP ein Alignment der Länge K und NW ein Alignment der Länge K' produziert, dann gibt es drei Möglichkeiten, wie sich diese zueinander verhalten:

1. $K = K'$: g_{\max} ist ideal gewählt,
2. $K > K'$: g_{\max} ist größer als für ein Alignment mit NW erforderlich.
3. $K < K'$: g_{\max} ist kleiner als für ein Alignment mit NW erforderlich.

Fälle eins und zwei sind unproblematisch, da wir in diesen Fällen nichts machen, oder mit doppelten Gaps auffüllen können. Wir haben für Fall zwei festgestellt, dass in dieser Situation bei MILP doppelte Gaps produziert werden und haben aus diesem Grund die Zielfunktion so formuliert, dass Doppelgaps keine Rolle in der Wertung spielen.

Fall drei bedarf jedoch besonderer Behandlung, da hier der mögliche Ursprung für f_{ij} eingeschränkt wird.

3.2.2.2.1 Invalide Zellen ausschließen Ein Schritt mit gleichbleibendem Zeilenindex entspricht einem Gap in s^1 , während ein gleichbleibender Spaltenindex einen Gap in s^2 bedeutet. Da wir nur g^1 , bzw. g^2 , Gaps in das Alignment von s^1 , bzw. s^2 einbauen können beschränken wir die Anzahl solcher Schritte.

Information

Zusätzlich zu den Koordinaten (i, j) brauchen wir noch g^1 und g^2 , damit wir uns orientieren können und damit die Aussageformen, welche wir im Folgenden formulieren zu Aussagen mit eindeutigem Wahrheitswert werden.

Da diese aus den Rahmenbedingungen des Problems hervorgehen werden wir diese meist nicht gesondert als Argumente deklarieren.

Aufgrund von (3.2.14) wissen wir, dass die Indexkoordinaten durch Schritte monoton steigen. Horizontale Schritte erhöhen den Zeilenindex und vertikale den Spaltenindex. Wenn wir uns horizontal bewegen, bauen wir einen Gap in s^1 ein, von denen wir höchstens g^1 haben dürfen. Analoges gilt für vertikale Schritte mit s^2 und g^2 . Wir sehen, dass es eine Korrespondenz zwischen der Anzahl von eingebauten Gaps und dem Abstand zur Mittelachse gibt.

Zellen, welche einen gewissen Abstand zur Hauptdiagonale überschreiten kommen also prinzipiell nicht infrage. Solche f_{ij} brauchen wir nicht zu betrachten.

Um die Überschreitung eines bestimmten Abstands g von der Hauptdiagonale festzustellen, definieren wir die boolsche **Hilfsfunktion** $\text{dist}_{\text{diag}} : \mathbb{N}^3 \rightarrow \mathbb{B}$, **welche aussagt**, ob die Zelle f_{ij} , von der Hauptdiago-

nale aus, innerhalb von g Schritten erreichbar ist, bzw. äquivalent, **ob das Alignment**, welches mit f_{ij} korrespondiert, **höchstens g Gaps enthält**.

$$\text{dist}_{\text{diag}}(i, j, g) = |i - j| \leq g \quad (3.2.18)$$

Da wir bei f_{00} auf der Hauptdiagonale starten, kommen nur Zellen infrage, die höchstens g^1 Schritte rechts von, bzw. g^2 Schritte unter der Hauptdiagonalen liegen.³⁵

Also oberhalb der Mittelachse $|i - j| \leq g^1$ und unterhalb davon $|i - j| \leq g^2$. Wir befinden uns auf, oder oberhalb der Mittelachse, wenn $i \leq j$.

Beispiel 3.7

Betrachten wir die Situation anhand der Beispielsequenzen $s^1 = \text{AGTAC}$ und $s^2 = \text{ATGC}$, mit $g_{\max} = 1$. Durch die Wahl von $g_{\max} = 1$ ergeben sich die Gapzahlen $g^1 = 1$ und $g^2 = 2$.

In den folgenden Matrizen $(d_{ij}) = D$ ist der Abstand zur Mittelachse $d_{ij} = |i - j|$ in den Zellen vermerkt.

	-	A	T	G	C
-	f_{00} 0	f_{01} 1	f_{02} 2	f_{03} 3	f_{04} 4
A	f_{10} 1	f_{11} 0	f_{12} 1	f_{13} 2	f_{14} 3
G	f_{20} 2	f_{21} 1	f_{22} 0	f_{23} 1	f_{24} 2
T	f_{30} 3	f_{31} 2	f_{32} 1	f_{33} 0	f_{34} 1
A	f_{40} 4	f_{41} 3	f_{42} 2	f_{43} 1	f_{44} 0
C	f_{50} 5	f_{51} 4	f_{52} 3	f_{53} 2	f_{54} 1

(a) Die komplette Matrix

	-	A	T	G	C
-	f_{00} 0	f_{01} 1			
A	f_{10} 1	f_{11} 0	f_{12} 1		
G	f_{20} 2	f_{21} 1	f_{22} 0	f_{23} 1	
T		f_{31} 2	f_{32} 1	f_{33} 0	f_{34} 1
A			f_{42} 2	f_{43} 1	f_{44} 0
C				f_{53} 2	f_{54} 1

(b) Nur plausible Zellen

Abbildung 3: NW-Matrizen mit eingetragenen Distanzen zur Hauptdiagonalen.

Die Zellen deren Abstände zur Hauptdiagonale sie nicht direkt disqualifizieren, sind dunkel hinterlegt.

Definieren wir also auf Basis von (3.2.18) das Prädikat $\text{range}(i, j)$, welches prüft, ob Zelle f_{ij} einen plausiblen Abstand zur Mittelachse hat.

$$\text{range}(i, j) = \begin{cases} \text{dist}_{\text{diag}}(i, j, g^1) & , i \leq j \\ \text{dist}_{\text{diag}}(i, j, g^2) & , \text{Andernfalls} \end{cases} \quad (3.2.19)$$

³⁵† Genauso gut können wir anstatt "unter", "links von" und anstatt "rechts von", "über" sagen.

Vorsicht

Die mithilfe von $\text{range}(i, j)$ formulierte Bedingung für Vorgängerkandidaten ist notwendig aber nicht hinreichend um eine valide Alignmentlänge zu garantieren.

Mehr dazu befindet sich in der Diskussion.

3.2.2.2.2 Rekursionsanker Einträge f_{ij} können nur dann definiert sein, wenn zumindest der diagonale Vorgänger $f_{i-1,j-1}$ definiert ist. Im Falle der Rekursionsanker gibt es natürlich keinen diagonalen Vorgänger, aber wenn $\text{range}(i, j)$ nicht gilt, können sie auch keine Vorgänger sein.

Daher können wir den klassischen Rekursionsanker (2.1.4) auf Basis von (3.2.19) neu formulieren.

$$f_{i0} = \begin{cases} i \cdot w_{\text{gap}} & , \text{range}(i, 0) \\ \perp & , \text{Andernfalls} \end{cases} \quad f_{0j} = \begin{cases} j \cdot w_{\text{gap}} & , \text{range}(0, j) \\ \perp & , \text{Andernfalls} \end{cases} \quad (3.2.20)$$

3.2.2.2.3 Kandidatenwahl Um zu notieren, welche der potentiellen Vorgänger von f_{ij} eine gültige Anzahl an Gaps einbauen, bestimmen wir einfach aus s^1, s^2 und g_{\max} die Gapzahlen g^1, g^2 und schreiben

$$\begin{aligned} \text{I} &= \text{range}(i-1, j-1) & , \text{für den diagonalen Vorgänger } f_{i-1,j-1} \\ \text{II} &= \text{range}(i-1, j) & , \text{für den vertikalen Vorgänger } f_{i-1,j} \\ \text{III} &= \text{range}(i, j-1) & , \text{für den horizontalen Vorgänger } f_{i,j-1} \end{aligned} \quad (3.2.21)$$

Wenn I nicht gilt, dann ist f_{ij} nicht definiert, wenn II nicht gilt, kommt $f_{i-1,j}$ nicht als Vorgänger infrage, und wenn III nicht gilt, kommt $f_{i,j-1}$ nicht als Vorgänger infrage.

Entsprechend ergibt sich ein Algorithmus mit einer im Vergleich zu (2.1.4) angepassten Befüllungsregel, bei der wir den maximalen Wert nur für plausible Alignments betrachten.

$$f_{ij} = \begin{cases} \max\{f_{i-1,j-1} + w_{ij}, f_{i-1,j} + w_{\text{gap}}, f_{i,j-1} + w_{\text{gap}}\} & , \text{I} \wedge \text{II} \wedge \text{III} \\ \max\{f_{i-1,j-1} + w_{ij}, f_{i-1,j} + w_{\text{gap}}\} & , \text{I} \wedge \text{II} \\ \max\{f_{i-1,j-1} + w_{ij}, f_{i,j-1} + w_{\text{gap}}\} & , \text{I} \wedge \text{III} \\ \max\{f_{i-1,j-1} + w_{ij}\} & , \text{I} \\ \perp & , \text{Andernfalls} \end{cases} \quad (3.2.22)$$

Die leere Menge hat kein größtes Element, also ist $\max \emptyset$ undefiniert. Dementsprechend können wir das Ergebnis für den Sammelfall am Ende auch als $\max \emptyset$ schreiben.

Was wir also eigentlich machen, ist aufgrund der Regeln I, II und III die potentiellen Kandidaten auswählen, unter denen wir den besten finden wollen. Wenn I nicht gilt, dann ist f_{ij} überhaupt nicht definiert, wenn II nicht gilt, dürfen wir $f_{i-1,j}$ nicht betrachten, und wenn III nicht gilt, dürfen wir $f_{i,j-1}$ nicht betrachten.

Wir definieren eine Funktion $\text{candidates}(i, j)$ zur Wahl von Kandidatenzellen.

$$\text{candidates}(i, j) = \begin{cases} \{(i-1, j-1), (i-1, j), (i, j-1)\} & , I \wedge II \wedge III \\ \{(i-1, j-1), (i-1, j)\} & , I \wedge II \\ \{(i-1, j-1), (i, j-1)\} & , I \wedge III \\ \{(i-1, j-1)\} & , I \\ \emptyset & , \text{Andernfalls} \end{cases} \quad (3.2.23)$$

Sei C_{ij} die Indexmenge aller prinzipiell möglichen Vorgänger von f_{ij} , mit $C_{ij} = \{(i-1, j-1), (i-1, j), (i, j-1)\}$, dann können wir (3.2.23) umformulieren, indem wir candidates als Funktion beschreiben, welche genau die Kandidaten aus C_{ij} wählt, die das Prädikat range erfüllen.

$$\text{candidates}(i, j) = \{c = (g, h) \in C_{ij} \mid \text{range}(c)\} \quad (3.2.24)$$

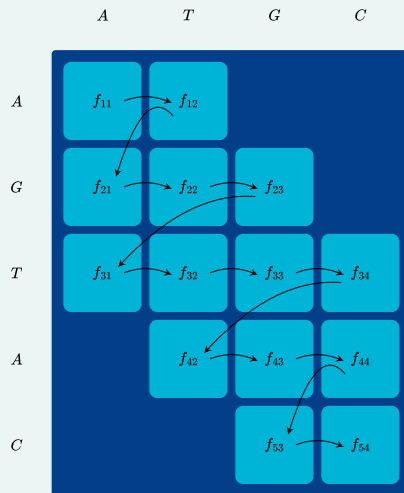
3.2.2.2.4 Rekursionsbeziehung In (3.2.15) haben wir definiert, wie wir Schritte werten können und mit (3.2.24) haben wir eine Auswahlfunktion für Kandidatenzellen.

Damit können wir (3.2.22) umformulieren und kommen zu einer neuen Definition unserer Rekursionsbeziehung.

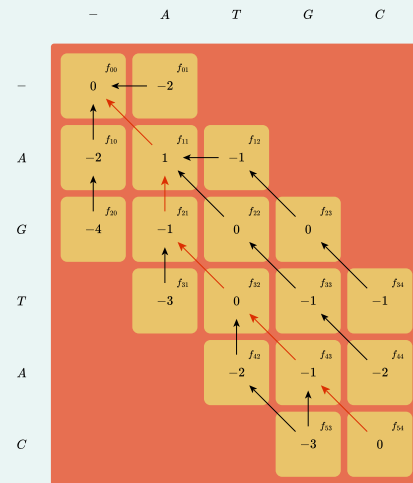
$$f_{ij} = \max\{f_{gh} + w((g, h), (i, j)) \mid (g, h) \in \text{candidates}(i, j)\} \quad (3.2.25)$$

Beispiel 3.8

Die Reihenfolge für Berechnungen und welche Zellen berechnet werden ändert sich entsprechend.



(a) Reihenfolge der Berechnungen beim angepassten Algorithmus



(b) Berechnete Zellen beim angepassten Algorithmus

Abbildung 4: Reihenfolge der Berechnungen und ausgefüllte Matrix für den angepassten NW-Algorithmus.

Wir sehen, dass bei NW mit beschränkter Gapzahl eine Bandmatrix entsteht.

Wenn wir fortan vom Needleman-Wunsch Algorithmus sprechen, meinen wir diese angepasste Version.

3.2.2.3 Herleitung der Variablen Können wir die Variablen des MILP-Modells mithilfe der neu interpretierten NW-Ergebnisse darstellen?

3.2.2.3.1 Zuweisungen Wenn Schritt $\text{diag}(q_k)$, dann $g \neq i \wedge h \neq j$ und s_i^1 und s_j^2 werden dem Alignment an Position k zugewiesen. Gemäß (3.1.3) der MILP-Formulierung wären $a_{ik}^1 = 1$ und $a_{jk}^2 = 1$, woraus mit (3.2.2.3.3) $\phi_{ijk} = 1$ folgt.

Verläuft q_k stattdessen vertikal, bzw. horizontal, also $g = i$ oder $h = j$, dann wird nur s_i^1 , bzw. s_j^2 , in das Alignment eingebaut, also $a_{jk}^2 = 0$, bzw. $a_{ik}^1 = 0$, woraus $\phi_{ijk} = 0$ folgt.

Wir beobachten, dass s_i^1 an Stelle k zugewiesen wird, wenn q_k den Zeilenindex von F verändert, also $g \neq i$ und dass dasselbe analog für s^2 und den Spaltenindex gilt.

$$a_{ik}^1 = \begin{cases} 1, & q_k = ((g, h), (i, j)) \in P \wedge g \neq i \\ 0, & \text{Andernfalls} \end{cases} \quad a_{jk}^2 = \begin{cases} 1, & q_k = ((g, h), (i, j)) \in P \wedge h \neq j \\ 0, & \text{Andernfalls} \end{cases} \quad (3.2.26)$$

Da a_{ik}^m in dieser für beliebige i und k definiert ist, können wir daraus direkt die Zuweisungsmatrizen $\mathcal{A}^1 \in \mathbb{B}^{M \times K}$ und $\mathcal{A}^2 \in \mathbb{B}^{N \times K}$ ableiten. Ebenso ergibt sich die Variable $\phi_{ijk} = a_{ik}^1 \cdot a_{jk}^2$ gemäß Definition (3.2.2.3.3)

Im Allgemeinen gilt Zuweisung a_{ik}^m in der NW-Formulierung, wenn q_k die m -te Indexkomponente ändert.

$$a_{ik}^m = \begin{cases} 1, & q_k \in P : o_k^m \neq d_k^m \wedge d_k^m = i \\ 0, & \text{Andernfalls} \end{cases} \quad (3.2.27)$$

3.2.2.3.2 Gaps Analog zu den Zuweisungsvariablen, ergeben sich für q_k die Gapvariablen g_k^1 und g_k^2 , mit $g_k^1 = 1$ und $g_k^2 = 0$ für horizontale, bzw. $g_k^1 = 0$ und $g_k^2 = 1$ für vertikale Schritte.³⁶

Im Allgemeinen entsteht ein Gap in der k -ten Stelle des Alignments von s^m wenn Schritt q_k nicht die m -te Koordinate verändert.

$$g_k^m = \begin{cases} 1, & o_k^m = d_k^m \\ 0, & \text{Andernfalls} \end{cases} \quad (3.2.28)$$

Für die Gaps einer Sequenz s^m mit $|s^m| = n$ haben wir in der MILP-Formulierung definiert, dass $g_k^m = [\sum_{i=1}^K a_{ik}^m = 0]$, also $g_k^m = 1$ g.d.w. die Summe der k -ten Spalte von \mathcal{A}^m gleich 0 ist und $g_k^m = 0$ sonst.

Aus (3.2.27) in Verbindung mit der Tatsache, dass der k -te Schritt q_k eindeutig durch seinen Index bestimmt ist, können wir die folgende Aussage ableiten:

$$o_k^m = d_k^m \implies \sum_{i=1}^{|s^m|} a_{ik}^m = 0$$

Wir sehen, dass aus der Voraussetzung für Gaps in NW die Voraussetzung für Gaps in MILP folgt.

3.2.2.3.3 Phi Mithilfe der Zuweisungsvariablen lässt sich ϕ_{ijk} direkt gem. (3.2.2.3.3) als $\phi_{ijk} = a_{ik}^1 \cdot a_{jk}^2$ definieren, aber wie lässt sich die Bedeutung von ϕ_{ijk} im Rahmen von NW interpretieren?

Mit (3.2.27) sehen wir, dass $\phi_{ijk} = 1 \cdot 1 = 1$ genau dann, wenn Schritt q_k beide Indexkomponenten verändert und in Zelle (i, j) endet. Aufgrund von (3.2.13) können wir Folgendes schreiben:

$$\phi_{ijk} = \begin{cases} 1, & \text{diag}(q_k) \wedge d_k = (i, j) \\ 0, & \text{Andernfalls} \end{cases} \quad (3.2.29)$$

D.h. die Symbole s_i^1 und s_j^2 werden an Position t_k zugewiesen wenn q_k diagonal verläuft und in (i, j) endet. Wir sehen, dass die MILP-Formulierung von ϕ_{ijk} konsistent zum NW-Modell ist.

3.2.2.3.4 Gamma Analog lässt sich durch (3.2.28) auch γ_k direkt gem. (3.2.2.3.4) als $\gamma_k = |g_k^1 - g_k^2|$ definieren. Es gibt aber auch bei γ_k Besonderheiten im Rahmen von NW.

Mit (3.2.5) sehen wir, dass sich zwischen Start und Ziel eines beliebigen Schrittes von q_k mindestens eine Koordinate unterscheiden muss. Daraus folgt, dass es bei NW im zweidimensionalen Fall keine doppelten Gaps geben kann.

³⁶ Für vertikale Schritte gilt $g = i$ und für horizontale $h = j$.

$$\forall k \in J_K : g_k^1 \neq 1 \vee g_k^2 \neq 1 \quad (3.2.30)$$

Dies ergibt intuitiv Sinn, da Schritte nicht gleichzeitig horizontal und vertikal verlaufen können.

Wenn wir nun $\gamma_k = |g_k^1 - g_k^2|$ entsprechend der MILP-Definition in (3.2.2.3.4) ableiten, dann sehen wir, dass aus einem Gap in Stelle k , egal in welcher Sequenz, automatisch $\gamma_k = 1$ folgt.

$$\gamma_k = \begin{cases} 1 & , g_k^1 = 1 \vee g_k^2 = 1 \\ 0 & , \text{Andernfalls} \end{cases} \quad \text{gleichbedeutend mit } \gamma_k \iff g_k^1 \vee g_k^2 \quad (3.2.31)$$

Mit (3.2.30) folgt auch, dass wir das logische ODER durch die Addition ersetzen können.

So ergibt sich eine vereinfachte Definition für den NW-Kontext.

$$\gamma_k = g_k^1 + g_k^2 \quad (3.2.32)$$

Dieser Zusammenhang gilt in der Form nicht im MILP-Modell, da es dort doppelte Gaps geben kann, sodass gleichzeitig $\phi_{ijk} = 0$ und $\gamma_k = 0$.

3.2.2.3.5 Unvereinbarkeit von Phi und Gamma Wenn wir (3.2.28) in (3.2.31) einsetzen, erhalten wir:

$$\gamma_k \iff o_k^1 = d_k^1 \vee o_k^2 = d_k^2$$

Mit (3.2.13) sehen wir nun, dass dies mit der Bedingung für (3.2.29) unvereinbar ist.

$$\forall q_k \text{ mit } d_k = (i, j) : \phi_{ijk} \iff \neg \gamma_k \quad (3.2.33)$$

3.2.2.4 Herleitung der Beschränkungen Gelten für die in (3.2.27) definierten a_{ik}^m die Beschränkungen des MILP-Modells?

Betrachten wir die Zuweisungen a_{ik}^m einer Sequenz s^m mit $|s^m| = M$, die sich aus dem k -ten Schritt q_k durch den Pfad P mit $K + 1$ Elementen und K Schritten ergeben. Dabei interessiert nur die m -te Koordinate der Elemente in P und es ist klar, dass es höchstens einen k -ten Schritt geben kann bzw. genau einen geben muss, wenn $k \leq K$.

3.2.2.4.1 Jedes Symbol genau einmal In unserem MILP-Modell gilt mit (3.1.6) die Beschränkung, dass jedes Symbol genau einmal zugewiesen werden muss.

$$\forall i \in J_n : \sum_{k=1}^K a_{ik}^m = 1$$

Sei $\delta_k^m = d_k^m - o_k^m$ die Schrittweite von q_k in der m -ten Koordinate, dann muss $d_k^m = o_k^m + \delta_k^m$. Mit (3.2.9) haben wir $d_k^m = p_k^m$ und mit (3.2.10) und (3.2.4) $d_0^m = 0 = p_0^m$.

Dies können wir nutzen um p_k^m als die Summe der ersten k Schrittweiten in der m -Koordinate darzustellen.

$$p_k^m = \sum_{i=1}^k \delta_i^m \quad (3.2.34)$$

Mit (3.2.4) gilt auch $p_K^m = M$ womit aus (3.2.34) $\sum_{k=1}^K \delta_k^m = M$ folgt. Mit (3.2.11) sehen wir weiterhin, dass $\delta_k^m \in \mathbb{B}$.

Es muss also genau M Schritte $q'_k \in P$ mit $\delta_k^m = 1$ und somit $o_k^m \neq d_k^m$ geben. Für jedes dieser q'_k folgt gem. (3.2.27) nach Wahl von $i = d_k^m$ die Aussage $a_{ik}^m = 1$. Gleichzeitig muss auch für jedes j mit $i \neq j$ gelten, dass $a_{jk}^m = 0$.³⁷

Die Annahme, dass zwei unterschiedliche dieser M Schritte q'_k und q'_l , dasselbe Ziel $d_k^m = d_l^m$ haben führt zum Widerspruch. Seien q'_k und q'_l aufeinanderfolgende Schritte, also $k < l$ mit $l = k + 1$, dann mit (3.2.9) $d_k^m = o_l^m$ und aufgrund der Voraussetzung $\delta_k^m = 1$ auch $d_l^m = o_l^m + 1$.

Aus der Annahme ergibt sich der Widerspruch $d_k^m = d_k^m + 1$. Daraus folgt, dass aufeinanderfolgende Schritte deren Schrittweite in der m -ten Koordinate ungleich null ist auch verschiedene Ziele in der m -ten Koordinate haben müssen.

Wir können für nicht direkte Nachfolger analog mit mehrfacher Anwendung dieser Regel vorgehen.

Wir haben gezeigt, dass es M verschiedene Schritte q'_k , mit $\delta_k^m = 1$ und paarweise unterschiedlichen Zielen d_k^m gibt, für die bei der Wahl $i = d_k^m$, die Aussage $a_{ik}^m = 1$ folgt. Da wir diese M unterschiedlichen d_k^m auf die M unterschiedliche Zeilen i der Matrix \mathcal{A}^m verteilen, folgt nach Taubenschlagprinzip, dass für jede Zeile genau ein solcher Schritt existiert.

$$\forall i \in J_n \exists! q'_k \in P : o_k^m \neq d_k^m \wedge d_k^m = i$$

Die NW-Definition (3.2.27) von a_{ik}^m besagt, dass für $q_k \in P$ gilt $a_{ik}^m = 1 \iff o_k^m \neq d_k^m \wedge d_k^m = i$ und $a_{ik}^m = 0$ sonst. Da wir $o_k^m \neq d_k^m \wedge d_k^m = i$ für q'_k bereits gezeigt haben folgt $\forall i \in J_n : \sum_{k=1}^K a_{ik}^m = 1$ und wir sind fertig. ■

3.2.2.4.2 Jede Position höchstens einmal In unserem MILP-Modell gilt mit (3.1.7) die Beschränkung, dass jeder Stelle im Template höchstens ein Sequenzsymbol zugewiesen werden darf.

$$\forall k \in J_K : \sum_{i=1}^K a_{ik}^m \leq 1$$

Aus der (3.2.27) ergibt sich, dass $d_k^m = i$ notwendige Voraussetzung für $a_{ik}^m > 0$ ist. Daher müssen Elemente in Zeilen j ungleich i in der k -ten Spalte von \mathcal{A}^m den Wert null haben, da trivial $i \neq j \wedge d_k^m = i \implies d_k^m \neq j$ gilt und $d_k^m \neq j$ die Aussage $a_{jk}^m = 1$ ausschließt. So folgt für beliebige $i, j, q_k \in P$ mit $i \neq j$ die Aussage $a_{ik}^m = 1 \implies a_{jk}^m = 0$.

Nehmen wir nun an, dass $\sum_{i=1}^K a_{ik}^m > 1$, dann haben wir $\exists i, j, i \neq j : a_{ik}^m = 1 \wedge a_{jk}^m = 1$, was im Widerspruch zum gezeigten $i \neq j \wedge a_{ik}^m = 1 \implies a_{jk}^m = 0$ steht. Dementsprechend kann die Spaltensumme 1 nicht überschreiten.

$$\forall q_k \in P : \sum_{i=1}^K a_{ik}^m \leq 1$$

³⁷ Dieses Zwischenergebnis nimmt den folgenden Beweis vorweg.

Pfad P hat K Schritte, also wird die Menge der Schritte in P durch J_K indiziert und $q_k \in P \implies k \in J_K$ und wir können äquivalent $\forall k \in J_K, i \neq j : \sum_{i=1}^K a_{ik}^m \leq 1$ schreiben, was (3.1.7) entspricht. ■

3.2.2.4.3 Reihenfolge der Sequenzsymbole In unserem MILP Modell gilt mit (3.1.8) die Beschränkung, dass die Reihenfolge der Sequenzsymbole erhalten bleiben muss.

$$\forall i, i+1 \in J_n, k, l \in J_K, k \leq l : a_{i+1,k}^m + a_{il}^m \leq 1$$

Die Aussage kann nur dann falsch sein, wenn gleichzeitig $a_{i+1,k}^m = 1$ und $a_{il}^m = 1$.

Wir haben in den vorigen Beweisen gezeigt, dass Alignmentpositionen nur einmal zugewiesen werden können, dass also für beliebige $i, j \in J_n, q_k \in P$ mit $i \neq j$ die Aussage $a_{ik}^m = 1 \implies a_{jk}^m = 0$ gilt. Falls also $k = l$, dann folgt unter der Annahme $a_{ik}^m = a_{il}^m = 1$ aus $i \neq i+1$ die Gleichung $a_{i+1,k}^m = 0$ und wir bekommen $0 + 1 \leq 1$, was offensichtlich unproblematisch ist.

Wir wissen weiterhin $\delta_k^m = 1$ ist notwendige Voraussetzung für $a_{ik}^m = 1$. Seien nun $q_{k-1}, q_k \in P$ zwei benachbarte Schritte und sei $\delta_k^m = 1$, dann $d_k^m = o_k^m + \delta_k^m = o_k^m + 1$. Mit $d_{k-1}^m = o_k^m$ ergibt sich $d_k^m = d_{k-1}^m + 1$ und somit $d_k^m > d_{k-1}^m$. Zusammengefasst haben wir gezeigt, dass

$$\forall q_{k-1}, q_k \in P : \delta_k^m = 1 \implies d_{k-1}^m < d_k^m$$

Mit $p^m \leq p_{k+1}^m$ folgt bei t -facher Anwendung trivial $p^m \leq p_{k+t}^m$. Also $k < l \wedge \delta_l^m = 1 \implies d_k^m < d_l^m$ für beliebige $q_k, q_l \in P$. Das bedeutet, dass ein beliebiger Schritt, der eine Änderung der m -ten Koordinate bewirkt, ein streng größeres Ziel hat, als beliebige vorherige Schritte.

Wählen wir $i+1 = d_k^m$ und $i = d_l^m$ und nehmen weiterhin an $\delta_k^m = 1$ und $\delta_l^m = 1$, damit wir $a_{i+1,k}^m = 1$ und $a_{il}^m = 1$ bekommen. Die Wahl $i+1$ steht aber im Widerspruch zu $k < l \wedge \delta_l^m = 1 \implies d_k^m < d_l^m$.

Da es, unter Annahme von $k \leq l$ nicht möglich ist, die Bedingung $a_{i+1,k}^m + a_{il}^m \leq 1$ zu verletzen, gilt die Beschränkung. ■

3.2.2.5 Herleitung der Zielfunktion Da wir wissen, dass für die im NW-Kontext formulierten a_{ik}^m und g_k dieselben Beschränkungen gelten wie im MILP Modell, muss das auch für die daraus resultierenden ϕ_{ijk} und γ_k gelten, da wir diese äquivalent zu den MILP Entsprechungen definiert haben.

3.2.2.5.1 Kosten Mit dieser Erkenntnis können wir den in (3.2.15) formulierten Schrittwert $w(q_k)$ alternativ hinsichtlich ϕ_{ijk} und γ_k formulieren. Bezeichnen wir diese alternative Formulierung vorerst mit $w'(q_k)$.

$$w'(q_k) = \phi_{ijk} \cdot w_{ij} + \gamma_k \cdot w_{\text{gap}}$$

Die Definition von ϕ_{ijk} in (3.2.29) setzt die Bedingung für $w(q_k) = w_{ij}$, in (3.2.15) voraus also folgt aus $\phi_{ijk} = 1$ die Aussage $w(q_k) = w_{ij}$ und unter Annahme von $d_k = (i, j)$ besteht Äquivalenz.

Weiterhin haben wir (3.2.2.3.4) und (3.2.28) Da $o_k^m \neq d_k^m$ notwendige Voraussetzung für $a_{ik}^m = 1$ ist, folgt aus $a_{ik}^m = 1$ für ein bestimmtes i , dass $g_k^m = 0$ und analog folgt aus $g_k^m = 1$, dass $a_{ik}^m = 0$, für beliebige i .

$$a_{ik}^m = 1 \iff g_k^m = 0 \quad \wedge \quad g_k^m = 1 \iff a_{ik}^m = 0$$

Mit (3.2.33) sehen wir, dass aus $\gamma_k = 1$ die Bedingung für $w(q_k) = w_{\text{gap}}$ folgt und $\gamma_k = 1$ g.d.w. $w(q_k) = w_{\text{gap}}$. Also folgt $w'(q_k) = w(q_k)$ für beliebige q_k .

$$w(q_k) = \phi_{ijk} \cdot w_{ij} + \gamma_k \cdot w_{\text{gap}} \quad (3.2.35)$$

Somit bekommen wir für Z_P den folgenden Ausdruck:

$$Z_P = \max \sum_{k=1}^K [\gamma_k \cdot w_{\text{gap}} + \phi_{ijk} \cdot w_{ij}] \quad (3.2.36)$$

Der Faktor ϕ_{ijk} hängt auch von i, j ab. Da $\phi_{ijk} = 0$ wenn $\text{dest}(q_k) \neq (i, j)$, können wir $\phi_{ijk} \cdot w_{ij}$ problemlos über die Sequenzlängen summieren, da Terme mit solchen i, j , die nicht dem Ziel von Schritt q_k entsprechen, immer den Wert null annehmen.

$$Z_P = \max \sum_{k=1}^K \left[\gamma_k \cdot w_{\text{gap}} + \sum_{i=1}^M \sum_{j=1}^N [\phi_{ijk} \cdot w_{ij}] \right] \quad (3.2.37)$$

Dies entspricht der MILP-Formulierung in (3.1.14) und wir sehen $Z = Z_P$.

3.3 Implementation eines Sequenzalinierers

In dieser Sektion widmen wir uns der Implementation einer Lösung für das Problem der paarweisen Sequenzalinierung, welche auf der Arbeit im vorigen Kapitel aufbaut.

Dafür definieren wir zuerst entsprechende Datentypen, um das Problem darzustellen und im Anschluss Funktionen, um es zu lösen.

Information

Dieses Kapitel beinhaltet viele Codeblöcke und beispielhafte Ein- und Ausgaben. Näheres zu den Hintergründen findet sich im Kapitel Methoden.

Codeblöcke

Die Codeblöcke sind im Text eingebunden und mit einer ID versehen. Die Block-ID steht am rechten Rand über dem Block und ist mit "BLOCK-ID:" gekennzeichnet. Beispielsweise hat der folgende Block die ID "example":

BLOCK-ID: «example»

```
foo :: String
foo = "bar"
```

Anhand der Block-ID kann ein Block später in anderen Codeblöcken referenziert werden. Block-IDs sind ggf. Pfade zu Dateien. In diesem Fall wird der Inhalt des Blocks in die referenzierte Datei geschrieben.

Der folgende Block referenziert den "example" Block und schreibt das Ergebnis nach `src/example.hs`:

BLOCK-ID: «src/example.hs»

```
module Example where
import Data.List
<<example>>
```

An der Position des Strings `<<example>>` werden die Inhalte des referenzierten Blocks eingefügt.

Für die Implementation nutzen wir die funktionale Sprache Haskell. Haskell nutzt Module, um Code zu strukturieren. Diese beginnen mit `module <name> where`, gefolgt von Imports `import Module (func1, func2, ...)`, welche am Anfang des Moduls durchgeführt werden müssen und anschließend den Funktionsdeklarationen.

3.3.1 Naive Implementation

Die einfachste Variante ein optimales Alignment zu finden ist es, alle Möglichkeiten auszuprobieren. Wir haben bereits im letzten Kapitel festgestellt, dass dies keine gute Idee ist, aber implementieren diese Variante als Fingerübung um uns mit Haskell's Syntax und dem typischen Vorgehen vertraut zu machen.

3.3.1.1 Datentypen Haskell arbeitet mit algebraischen Datentypen. Neue Datentypen können mit `data` oder `newtype` definiert werden und Typenaliasse mit `type`.

Information

Wir haben drei Möglichkeiten, um in Haskell neue Datentypen anzulegen. Dafür nutzen wir die Keywords **type**, **newtype** und **data**.

Mit **type** wird ein Typenalias angelegt. Z.b. kann man mit **type Point = (Float, Float)** Tupel von Gleitkommazahlen als **Point** referenzieren. Wenn jetzt eine Funktion **norm :: Point → Float** angelegt, dann nimmt diese ein solches Tupel.

Mit **newtype** wird ein neuer Typ angelegt, der Isomorph zu einem existierenden Typen ist. Würde man also **newtype Point = Point (Float, Float)** definieren, verhielte sich **Point** identisch zu Tupeln von Gleitkommazahlen, wäre aber selber kein solches Tupel. Legen wir jetzt wieder **norm :: Point → Float** an, dann nimmt diese nur eingaben vom Typ **Point**, nicht aber einfache Tupel von Zahlen.

Mit **data** wird ein neuer Datentyp angelegt. Dieser kann eine beliebige Struktur haben, die nicht auf anderen Typen basieren muss.

Wir brauchen Typen für:

- Das Alphabet Σ ,
- Sequenzen, als Folgen von Symbolen aus Σ ,
- Gaps, bzw. das Gapsymbol c_{gap} ,
- das Alignmentalphabet $\bar{\Sigma}$,
- alinierte Sequenzen, als Folgen über $\bar{\Sigma}$,
- Kosten für Matches, Missmatches und Gaps und
- Alignments, als geordnete Paare alinierter Sequenzen.

Zunächst definieren wir einen Datentypen für unser Alphabet Σ , bestehend aus den Nukleobasen $\{A, C, G, T\}$.

BLOCK-ID: «base»

```
-- | Datatype for nucleotides
data Base = A | C | G | T deriving (Eq, Show, Read)
```

Jetzt können wir die Buchstaben **A**, **C**, **G** und **T** als Konstruktoren für Basen verwenden.

Information

Typenklassen, wie `Eq`, `Show` und `Read`, ermöglichen die Nutzung bestimmter Operationen mit unseren Datentypen.

Die `Eq` Klasse ermöglicht es beispielsweise Elemente miteinander auf Gleichheit zu prüfen. Die `Show` und `Read` Klassen erlauben die Umwandlung zu und von Strings.

Mittels der `deriving` Statements leiten wir automatisch von bestimmten Typenklassen ab, aber nicht jede Typenklasse ist automatisch ableitbar. Um einen Datentypen manuell zu einer Typenklasse hinzuzufügen, nutzen wir das `instance` Keyword und implementieren die notwendigen Funktionen.

Typenklassen sind also ungefähr vergleichbar mit Interfaces in objektorientierten Sprachen, da sie nur definieren, was mit einem Datentypen gemacht werden kann, nicht aber unbedingt wie es gemacht wird.

Darüber hinaus wollen wir Sequenzen von Buchstaben nutzen können.

Block-ID: «naive-seq»

```
-- | Datatype for sequences
newtype Seq a = Seq [a] deriving (Eq)
```

Also legen wir mithilfe des `newtype` Keywords, den polymorphen `Seq` Konstruktor an. Mit diesem können wir Sequenzen aus Listen eines bestimmten Typs, z.B. von Basen, definieren.

Wenn wir `Seq` von `Show` und `Read` ableiten ließen, sähe die String-Darstellung der Sequenz "ACGT" so aus: `Seq [A,C,G,T]`. Wir wollen allerdings eine flache String-Darstellung. Dafür definieren wir eine Hilfsfunktion und fügen den Datentypen `Seq a` der Typenklasse `Show` manuell hinzu.

Weiterhin wollen wir Sequenzen wie Listen von Basen behandeln können, weswegen wir `Functor` und `Foldable` implementieren müssen, um über Elemente zu mappen.

Information

Das Mappen einer Funktion f über die Liste xs entspricht der Anwendung von f auf jedes Listenelement.

Die Definition von `map` gleicht dem Folgenden Code.

```
map :: (a -> b) -> [a] -> [b]
map f [] = []
map f (x:xs) = f x : map f xs
```

`map` nimmt eine Funktion $f :: a \rightarrow b$ und eine Liste von a Werten und produziert eine Liste von b Werten. Sie ist ein Sonderfall der allgemeineren `fmap` Funktion.

Die `fmap` Funktion hat die Signatur^a $\text{fmap} :: (\text{Functor } f) \Rightarrow (a \rightarrow b) \rightarrow f\ a \rightarrow f\ b$ und wendet eine Funktion auf einen Funktorkontext an.

Der Begriff Funktor stammt aus der Kategorientheorie und bezeichnet dort eine Abbildung $F : \mathcal{C} \rightarrow \mathcal{D}$, die Objekte und Morphismen^b in der Kategorie \mathcal{C} auf Objekte und Morphismen in der Kategorie \mathcal{D} abbildet und dabei die Identitätsmorphismen erhält.^c

Aus diesem Grund muss für `fmap` gelten, dass `fmap id == id`.

^a Da wir, der Anschaulichkeit wegen, Funktionssignaturen auch in `instance` Deklarationen nutzen möchten, müssen wir auch die `{-# LANGUAGE InstanceSigs #-}` Spracherweiterung aktivieren.

^b Strukturbewahrende Abbildungen innerhalb einer Kategorie.

^c Für jedes Objekt X in \mathcal{C} existiert ein Identitätsmorphismus $\text{id}_X : X \rightarrow X$, welcher X auf sich selber abbildet. Wenn F den Morphismus $f : X \rightarrow Y$ in \mathcal{C} , auf den Morphismus $F(f) : F(X) \rightarrow F(Y)$ abbildet, dann muss also gelten, dass $F(\text{id}_X) = \text{id}_{F(X)}$.

Block-ID: «naive-seq-classes»

```
-- | Helper for implementing flat readable types.
readFlatList :: Read a => String -> [a]
readFlatList str = read' str
  where
    wrap' :: String -> [String]
    wrap' str = [[toUpper ch] | ch <- str]

    read' :: Read a => String -> [a]
    read' = map (read :: Read a => String -> a).wrap'

-- | Make sequences display as flat strings.
instance Show a => Show (Seq a) where
  show (Seq symbols) = concatMap show symbols

-- | Enable reading sequences from flat strings.
instance Read a => Read (Seq a) where
  readsPrec _ chars = [(Seq $ readFlatList chars, "")]

-- | Make Seq a Functor, so we can apply functions to its contents.
instance Functor Seq where
  fmap = fmapDefault

-- | Make Seq Foldable, so we can use functions to summarize its contents.
instance Foldable Seq where
  foldMap = foldMapDefault
```

```
-- | Make Seq Traversable, so we can apply functions to its contents, while
  ↳ preserving the structure.
```

```
instance Traversable Seq where
  -- sequenceA :: Applicative f => Seq (f a) -> f (Seq a)
  traverse :: Applicative f => (a -> f b) -> Seq a -> f (Seq b)
  traverse g (Seq xs) = Seq <$> traverse g xs
```

Jetzt sieht die Darstellung von `Seq [A, C, G, T]` so aus: `ACGT` und wir können Strings von Basensymbolen einlesen und einfach Funktionen auf Sequenzen anwenden.

Um zwei Sequenzen zu alinieren, müssen wir die Möglichkeit haben Gaps einzubauen. Also brauchen wir einen Datentypen für das Alphabet $\bar{\Sigma} = \Sigma \cup \{c_{\text{gap}}\}$, für welchen es natürlich auch eine ansprechende Repräsentation geben sollte.

BLOCK-ID: «aln-char»

```
-- | Define an alignment alphabet with gaps, based on another alphabet type
data AlnChar a = Symbol a | Gap deriving (Eq)
```

```
-- gap chars that are allowed for reading
gapChar = '-'
gapSynonyms = "._."
gapChars = gapChar : gapSynonyms
```

```
-- | Make the alignment alphabet display properly.
```

```
instance Show a => Show (AlnChar a) where
  show (Symbol s) = show s
  show Gap = show gapChar
```

```
-- | Allow function application to the AlnChar contents.
```

```
instance Functor AlnChar where
  fmap :: (a -> b) -> AlnChar a -> AlnChar b
  fmap f (Symbol s) = Symbol (f s)
  fmap f Gap = Gap
```

Mit dem `AlnChar` Typkonstruktor können wir für ein beliebiges Alphabet Σ , bzw. `a`, ein entsprechendes Alignmentalphabet $\bar{\Sigma}$, bzw. `AlnChar a`, definieren, dessen Buchstaben entweder Symbole aus Σ (`Symbol a`), oder Gaps (`Gaps`) sind.

Zusätzlich zu $\bar{\Sigma}$ müssen wir auch Sequenzen über $\bar{\Sigma}$ darstellen und repräsentieren.

BLOCK-ID: «aln-seq»

```
-- | Type for aligned sequences, i.e., those potentially containing gaps.
```

```
newtype AlnSeq a = AlnSeq [AlnChar a] deriving (Eq)
```

```
-- | Make aligned sequences display properly.
```

```
instance Show a => Show (AlnSeq a) where
  show (AlnSeq symbols) = concatMap show symbols
```

```
-- | Enable reading aligned sequences from flat strings.
```

```
-- instance Read a => Read (AlnSeq a) where
--   readsPrec _ chars = [(AlnSeq $ readFlatList chars, "")]
```

```
-- | Allow function application to the AlnSeq contents.
```

```
instance Functor AlnSeq where
  fmap :: (a -> b) -> AlnSeq a -> AlnSeq b
  fmap f (AlnSeq seq) = AlnSeq $ map (fmap f) seq
```

```
-- | Allow summarizing AlnSeq contents.
instance Foldable AlnSeq where
  foldr :: (a -> b -> b) -> b -> AlnSeq a -> b
  foldr f z (AlnSeq (Gap:syms)) = foldr f z (AlnSeq syms)
  foldr f z (AlnSeq ((Symbol s):syms)) = f s $ foldr f z (AlnSeq syms)
```

Mithilfe von `AlnChar` definieren wir also den Typkonstruktor `AlnSeq` für alinierte Sequenzen von Symbolen in Σ , bzw. a . Dementsprechend repräsentiert `AlnSeq` Sequenzen über $\bar{\Sigma}$.

Nun überlegen wir uns, wie Alignments, bzw. Templates dargestellt werden sollten.

BLOCK-ID: «naive-aln»

```
-- | Define alignments as tuples of alignment sequences
newtype Aln a = Aln (AlnSeq a, AlnSeq a) deriving (Eq, Show)
```

Da wir nur paarweise Alignments betrachten, sind unsere Templates Matrizen der Form $2 \times k$. Der Einfachheit halber definieren wir Templates daher als Tupel alinierter Sequenzen.

Zuletzt definieren wir einen Typen für die Gewichte, die unsere Zielfunktion zum Bewerten der Alignments braucht und ein Alias, welches diese und g_{\max} bündelt.

Information

Beispiele

Ein- und Ausgaben werden im interaktiven Haskell REPL `ghci` werden folgenden Stil wiedergegeben:

```
ghci> func arg arg
<result>
```

Um diese selber auszuprobieren, kann die Arbeit geladen werden. Der Demo-Code für den naiven Ansatz befindet sich in `src/Align/Naive/Demo.hs` und der für die DP-Lösung in `src/Align/Demo.hs`. Um diesen zu laden muss zunächst in das `src/` Verzeichnis gewechselt werden.

Wenn es sich um Shell-Befehle außerhalb von `ghci` handelt, wird diese ein `$` vorangestellt. Bspw. wird mit dem folgenden Befehl der Demo-Code für den NW-Aligner in `ghci` geladen:

```
$ cd src/
$ ghci Align.hs
```

BLOCK-ID: «type-cost»

```
-- | Record type for costs.
data Cost = Cost {w_match :: Int, w_miss :: Int, w_gap :: Int} deriving (Eq, Show)
```


Beispiel 3.9

Bei `Cost` handelt es sich um einen sog. "Record Type", in dem verschiedene Werte gebündelt werden können. Dabei werden Felder mit bestimmten Datentypen deklariert, welche bei der Instanziierung gefüllt werden müssen. In diesem Falle gibt es die Felder `w_match`, `w_miss` und `w_gap`, welche Werte mit dem Typ `Int` halten und unseren Gewichten w_{match} , w_{miss} und w_{gap} entsprechen.

Die Übergabe der Argumente beim Definieren eines Wertes kann der Reihenfolge nach geschehen, oder die Argumente werden namentlich benannt.

```
ghci> Cost 9 12 (-3)
Cost {w_match = 9, w_miss = 12, w_gap = -3}
ghci> Cost {w_match = 1, w_gap = -2, w_miss = 3}
Cost {w_match = 1, w_miss = 3, w_gap = -2}
```

Um auf Feldwerte zuzugreifen, gibt es unterschiedliche Möglichkeiten.

Die Namen der Felder definieren automatisch Funktionen, welche für den Zugriff genutzt werden können.

```
ghci> let c = Cost 1 (-1) (-2)
ghci> w_gap c
-2
```

In Funktionsdefinitionen und `let` Bindungen kann auch sog. strukturelles Patternmatching genutzt werden. Dies funktioniert so wie die Instanziierung mit benannten Argumenten. Dabei werden die Feldwerte eines Funktionsargumentes direkt an Namen gebunden, unter denen sie in der Funktion verfügbar sein sollen.

```
total :: Cost -> Int
total cost@(Cost {w_match = match, w_miss = miss, w_gap = gap})
  = match + miss + gap
```

In diesem Beispiel binden wir `w_match`, `w_miss` und `w_gap` an die Namen `match`, `miss` und `gap` und bestimmen dann die Summe der Werte. Mit dem `@` geben wir dem gesamten Eintrag den Namen `cost`.

3.3.1.2 Logik Zunächst definieren wir Kombinationen als Listen ganzer Zahlen und eine Hilfsfunktion für die erste k -Kombination $(1, 2, \dots, k)$.

BLOCK-ID: «combination»

```
-- | Encodes an ordered list of indices.
type Combination = [Int]

-- | Compute the first combination of length k.
firstCombination :: Int -> Combination
firstCombination k = [1..k]
```

Bisher ist der Code trivial.

Versuchen wir nun eine Funktion $\text{succ} : C_k^n \rightarrow C_k^n$ zu definieren, welche für eine Kombination c_i den Nachfolger c_{i+1} findet.

BLOCK-ID: «successor»

```

-- | We chop off the last indices
-- fst gives number of chopped positions,
-- snd gives remaining indices
type TrimmedCombination = (Int, Combination)

-- | Compute the successor of a combination of indices for a list of length n.
successor :: Int -> Combination -> Maybe Combination
successor _ [] = Nothing
successor n comb = succ' comb
  where
    -- | Length of the combination.
    k :: Int
    k = length comb

    -- | Determine the first index, that can be incremented,
    -- and discard everything before.
    -- i: number of chopped indices, revComb: reversed combination
    incr :: Int -> Combination -> Maybe TrimmedCombination
    incr _ [] = Nothing
    incr i revComb@(x:xs)
      -- let i' := (k-i) be an index in a non-reversed combination
      -- then we have n-k+i' == n-k-(k-i) == n-i
      | x < (n-i) = Just (i, (x+1):xs)
      | x == (n-i) = incr (i+1) xs
      -- if x > (n-k+i') we're in illegal territory already
      | otherwise = Nothing

    -- | Fill the discarded bits of an incremented combination with the lowest
    -- possible subsequence.
    fill :: TrimmedCombination -> Combination
    fill (0, revComb) = let comb = reverse revComb in comb
    fill (i, x:xs) = fill (i-1, (x+1):x:xs)

    succ' :: Combination -> Maybe Combination
    succ' = (fmap fill).(incr 0).reverse

```

Das Vorgehen, um den Nachfolger einer Kombination c zu finden, funktioniert analog zur Beschreibung im vorigen Kapitel.

1. Zuerst finden wir mit `incr` den letzten Index i , mit $c_i < n - k + i$, inkrementieren diesen und schmeißen die höheren Stellen weg,
2. dann füllen wir in `fill` die entfernten Stellen mit der kleinsten Subkombination $(c_i + 1, c_i + 2, \dots, c_i + (k - i))$ wieder auf Länge k auf,
3. um anschließend beide Funktionen in `succ'` zu komponieren.

```

ghci> successor 4 [1, 4]
Just [2, 3]
ghci> successor 4 [3, 4]
Nothing

```

Der Wert `Nothing` signalisiert, dass es keinen validen Nachfolger gibt.

Information

Da C_k^n endlich ist, wissen wir, dass es eine letzte Kombination $c_{\binom{n}{k}} = (n - k, \dots, n)$, ohne Nachfolger gibt.

Um mit solchen Definitionslücken umzugehen, bietet Haskell die **Maybe** Monade an. **Maybe** a ist so definiert,^a dass es entweder einen Wert **Just** a oder **Nothing**, also keinen Wert, hat.

^a `data Maybe a = Just a | Nothing`

Mithilfe der Nachfolgefunktion `successor` können wir die Menge C_k^n aller $\binom{n}{k}$ möglichen Kombinationen berechnen.

Block-ID: «combinations»

```
-- | Compute all n choose k combinations.
combinations :: Int -> Int -> [Combination]
combinations n k
  -- TODO errors could be [] instead
  | n < k = error "k may not exceed n"
  | n < 0 = error "n may not be negative"
  | otherwise = cont start $ successor n start
  where
    start :: Combination
    start = firstCombination k

    cont :: Combination -> Maybe Combination -> [Combination]
    cont last Nothing = [last]
    cont last (Just next) = last : (cont next $ successor n next)
```

Wir starten mit der ersten Kombination der Länge k und fügen so lange den Nachfolger hinzu, wie dieser definiert ist.

```
ghci> combinations 4 3
[[1,2,3],[1,2,4],[1,3,4],[2,3,4]]
```

Da wir mit `combinations` alle validen Gapbelegungen erzeugen können, befüllen wir jetzt unsere Sequenzen mit diesen Gaps. Das bedeutet, dass wir von **Seq** in **AlnSeq** umwandeln. So können wir dann alle möglichen Alignments zweier Sequenzen generieren.

Block-ID: «alignments»

```
-- | Compute an aligned sequence from a sequence and a combination of gap positions.
alignSeq :: Seq a -> Combination -> AlnSeq a
alignSeq seq gaps = AlnSeq $ alignSeq' 1 seq gaps
  where
    alignSeq' :: Int -> Seq a -> Combination -> [AlnChar a]
    alignSeq' _ (Seq []) gaps = [Gap | g <- gaps]
    alignSeq' _ (Seq seq) [] = [Symbol sym | sym <- seq]
    alignSeq' pos seq@(Seq (b:bs)) comb@(gap:gaps)
      | gap == pos = Gap : alignSeq' (pos + 1) seq gaps
      | otherwise = (Symbol b) : alignSeq' (pos + 1) (Seq bs) comb

-- | Compute all possible alignments of two sequences with a given number of
  ↳ allowed gaps.
alignments :: Int -> Seq a -> Seq a -> [Aln a]
alignments g_max seq1 seq2 =
```

```
[Aln (alignSeq seq1 comb1, alignSeq seq2 comb2)
  | comb1 <- combinations k (k - m)
  , comb2 <- combinations k (k - n)]
where
  m = length seq1
  n = length seq2
  k = g_max + max m n
```

Um die Alignments der Sequenzen s^1 und s^2 , mit Längen M , bzw. N und Templatelänge $K = \max\{M, N\} + g_{\max}$ zu bestimmen, generieren wir die Menge $C_M^K \times C_N^K$ und nutzen alignSeq um die Sequenzen entsprechend der jeweiligen Kombination zu alinieren.

Nun können wir durch alle Möglichen Alignments iterieren. Was uns jetzt noch fehlt, ist eine Möglichkeit die Alignments zu bewerten.

In (3.1.13) haben wir die Zielfunktion folgendermaßen definiert:

$$\sum_{k=1}^K \left[w_{\text{gap}} \cdot \gamma_k \left[\sum_{i=1}^M \sum_{j=1}^N w_{i,j} \cdot \phi_{i,j,k} \right] \right]$$

BLOCK-ID: «naive-align-helpers»

```
score :: Eq a => Cost -> Aln a -> Int
score _ (Aln (AlnSeq [], AlnSeq [])) = 0
score cost (Aln (AlnSeq (x:xs), AlnSeq (y:ys))) =
  score' x y + score cost (Aln (AlnSeq xs, AlnSeq ys))
where
  score' :: Eq a => AlnChar a -> AlnChar a -> Int
  score' Gap Gap = 0
  score' Gap _ = w_gap cost
  score' _ Gap = w_gap cost
  score' x y
    | x == y = w_match cost
    | otherwise = w_miss cost
```

```
type ScoredAln a = (Int, Aln a)
```

```
maximize :: Eq a => Cost -> [Aln a] -> ScoredAln a
maximize _ [] = error "Nö"
maximize cost (aln:alns) = max (sc, aln) alns
where
  sc :: Int
  sc = score cost aln

  max :: Eq a => ScoredAln a -> [Aln a] -> ScoredAln a
  max scoredAln@(sc, best) [] = (sc, best)
  max scoredAln@(sc, best) (aln:alns)
    | sc <= nxtSc = max (nxtSc, aln) alns
    | otherwise = max scoredAln alns
  where
    nxtSc :: Int
    nxtSc = score cost aln
```

Jetzt können wir unsere naive Alignmentfunktion implementieren.

BLOCK-ID: «naive-align»

```
align :: Eq a => Seq a -> Seq a -> Cost -> Int -> ScoredAln a
align seq1 seq2 cost g_max = maximize cost $ alignments g_max seq1 seq2
```

Gegeben zwei Sequenzen `seq1` und `seq2`, Kosten und die Anzahl an zulässigen Gaps, berechnen wir das optimale Alignment, indem wir aus allen möglichen Alignments, gegeben durch `alignments g_max seq1 seq2`, dieses wählen, welches mit den gegebenen Kosten den maximalen Wert produziert.

Damit können wir das optimale Alignment für unsere Beispielsequenzen berechnen.

```
ghci> let cost = Cost 1 (-1) (-2)
ghci> let seq1 = read "agtac" :: Seq Base
ghci> let seq2 = read "atgc" :: Seq Base
ghci> align seq1 seq2 cost 1
(0,AIn (AGTAC-,A-TGC-))
```

3.3.2 Needleman-Wunsch Implementation

Eine bessere Variante unser Problem zu lösen ist die Alinierung mittels Needleman-Wunsch.

3.3.2.1 Datentypen Auch hier betrachten wir zunächst, welche Daten wir brauchen um das Problem darzustellen, bzw. zu lösen und definieren sinnvolle Datentypen.

Um unser Optimierungsproblem mit NW zu lösen, brauchen wir zumindest die folgenden Informationen:

- Anzahl erlaubter Gaps g_{\max} ,
- Kosten w_{match} , w_{miss} und w_{gap} ,
- Sequenzen s^1, s^2 ,

Wir können g_{\max} einfach als `Int` darstellen und für die Kosten haben wir bereits den `Cost` Record-Typen definiert. Wir definieren allerdings den `Seq` Typen neu, als schlichtes Alias für `String`. Dies erleichtert unsere Arbeit im weiteren Verlauf, erfordert aber auch, dass wir abhängige Typen wie z.B. `AlnChar` entsprechend anpassen.

Block-ID: «type-seq»

```
-- | We model sequences as plain strings, i.e., lists of chars.
type Seq = String

-- | Alignment characters consist of either symbols or gaps.
data AlnChar = Symbol Char | Gap deriving (Eq)

-- | Alignments are list of AlnChar tuples.
type Aln = [(AlnChar, AlnChar)]
```

Da die Laufzeit für die Wahl eines Listenelements mittels Index in der Klasse $\mathcal{O}(n)$ liegt, benutzen wir im Folgenden Arrays um Sequenzen zu speichern. Dadurch ist mit $\mathcal{O}(1)$ konstante Laufzeit für Zugriffe gewährleistet.

Um mit dem Typen `Array` zu arbeiten müssen wir diesen importieren. Wir importieren außerdem den `(!)` Operator, der Indexzugriff auf Arrayelemente erlaubt und die `listArray` und `elems` Funktionen, welche Arrays aus Listen erzeugen, bzw. in diese umwandeln.

Block-ID: «align-imports»

```
import Data.Array (Array, (!), listArray, elems)
```

Jetzt definieren wir den `SeqArray` Typen als Alias für `Array Int Char`, also mit `Int` indizierte Arrays von `Char` Werten, und eine Hilfsfunktion `mkArr` um Sequenzen in korrekt indizierte Arrays umzuwandeln.

Block-ID: «type-seq-arr»

```
-- | We use arrays for constant time access.
type SeqArr = Array Int Char

-- | Create sequence arrays from sequence strings.
-- Arrays allow constant time lookups.
-- Indexing is shifted by one, so we index from 2 through m+1.
mkArr :: Seq → SeqArr

mkArr xs = let m = length xs
            in listArray (2, m+1) xs
```

Aus den Sequenzen ergeben sich die Längen M, N und mit g_{\max} die erlaubten Gaps pro Sequenz g^1, g^2 .

Um all diese Informationen einfach zugreifbar zu haben, definieren wir einen entsprechenden Record-Typen `AlnInfo`, Funktionen um Längen, bzw. Gapzahlen zu bestimmen und einen Helfer um solche Records aus normalen Sequenzen und flachen Werten für Kosten und Gaps anzulegen.

Block-ID: «type-aln-info»

```
-- | Record with key data of the alignment problem.
data AlnInfo = AlnInfo
  { g_max    :: Int
  , weights  :: Cost
  , seqA     :: SeqArr
  , seqB     :: SeqArr
  } deriving (Eq, Show)

-- | Compute sequence lengths for an AlnInfo record.
seqLengths :: AlnInfo → (Int, Int)
seqLengths AlnInfo {seqA = s1, seqB = s2} = (length s1, length s2)

-- | Compute gap numbers for an AlnInfo record.
gapCounts :: AlnInfo → (Int, Int)
gapCounts info@(AlnInfo {g_max = g})
  = let (m, n) = seqLengths info
        l      = max m n
        in (g + (l - m), g + (l - n))

-- | Convenience constructor to create AlnInfo records from loose weights and lists.
mkInfo :: Int → Int → Int → Int → Seq → Seq → AlnInfo
mkInfo g_max w_match w_miss w_gap seqA seqB
  = AlnInfo g_max (Cost w_match w_miss w_gap) (mkArr seqA) (mkArr seqB)
```

Jetzt können wir die notwendigen Informationen, um das Alignmentproblem zu lösen als `AlnInfo` Einträge zusammenfassen und mit `seqLengths` und `gapCounts` die Sequenzlängen, bzw. Gapzahlen bestimmen.

```
ghci> let info = mkInfo 1 1 (-1) (-2) "AGTAC" "ATGC"
ghci> info
AlnInfo {g_max = 1, weights = Cost {w_match = 1, w_miss = -1, w_gap = -2}, seqA =
  ↪ array (2,6) [(2,'A'),(3,'G'),(4,'T'),(5,'A'),(6,'C')], seqB = array (2,5)
  ↪ [(2,'A'),(3,'T'),(4,'G'),(5,'C')]}
ghci> seqLengths info
(5,4)
ghci> gapCounts info
(1,2)
```

NW findet das optimale Alignment als maximalen Pfad durch eine Matrix, wobei der (Teil-) Pfad zu einer Zelle rekursiv aus den maximalen Schritten zu dieser Zelle aufgebaut wird. Pfade sind Folgen von Indizes und Schritte sind Paare von Indizes.

Da wir uns für viele Unterscheidungen dafür interessieren aus welcher Richtung wir kamen und nicht aus welcher spezifischen Zelle, definieren wir zusätzlich den Datentyp `StepDirection` und einen Helfer `fromStep` zur Umwandlung von `Step` in `StepDirection`.

BLOCK-ID: «type-mat-parts»

```
-- | Matrix indices.
type MatIdx = (Int, Int)

-- | Path through a matrix.
type Path = [MatIdx]

-- | Steps are `MatIdx` tuples of the form `(origin, destination)`.
type Step = (MatIdx, MatIdx)

-- | Datatype to denote step directions.
data StepDirection = Diagonal | Horizontal | Vertical deriving Eq

-- | Calculate a StepDirection from a Step.
fromStep :: Step -> StepDirection
fromStep (orig@(g, h), dest@(i, j))
  | i == g+1 && j == h+1 = Diagonal
  | i == g    && j == h+1 = Horizontal
  | i == g+1 && j == h    = Vertical
  | otherwise = error "illegal step"
```

Um einen Typen für unsere Matrix zu formulieren, müssen wir noch wissen, welche Art von Werten darin enthalten sind.

Information

Damit wir überhaupt mit Matrizen arbeiten können, importieren wir zunächst den `Matrix` Typen und einige Hilfsfunktionen aus dem, durch das `matrix` Paket bereitgestellten, `Data.Matrix` Modul.

BLOCK-ID: «align-imports» [2]

```
import Data.Matrix (Matrix, matrix, nrows, ncols, getElem, setElem)
```

Die `matrix` Funktion nimmt als Eingabe zwei Integer-Argumente, welche die Anzahl von Zeilen und Spalten darstellen und eine Füllfunktion `f :: Int -> Int -> a`, welcher die Zellindizes `i` und `j` übergeben werden und erstellt damit eine `Matrix a`. Die `nrows` und `ncols` Funktionen berechnen die Anzahl von Zeilen, bzw. Spalten einer Matrix und `getElem` und `setElem` werden genutzt um Werte von Zellen zu lesen, bzw. zu schreiben.

Offensichtlich müssen wir den Wert f_{ij} der Zelle speichern. Dieser ergibt sich aus den Kosten und kann als `Int` dargestellt werden. Zusätzlich dazu, wollen wir noch wissen, aufgrund welcher vorherigen Zellen der Wert zustande kam. Dazu nutzen wir den `StepDirection` Typen und da es potentiell mehr als einen Vorgänger geben kann, speichern wir eine Liste von Vorgängern. Wir können Werte in Zellen also als Tupel `(Int, [StepDirection])` darstellen.

Da wir am Anfang nicht definierte Einträge haben und auch beim anschließenden Befüllen nicht zwangsläufig alle Zellen berechnen, nutzen wir wieder die `Maybe` Monade und kennzeichnen undefinierte Zellen mit `Nothing`.

Wir arbeiten also über einer Matrix mit Zellwerten vom Typ `Maybe (Int, [StepDirection])`.

BLOCK-ID: «type-nw-matrix»

```
-- | What goes into the matrix? If a cell is defined,
-- we have just a value f_ij, and a list of one or more
-- precursors, otherwise nothing.
type CellValue = Maybe (Int, [StepDirection])

-- | A Needleman-Wunsch matrix.
type NWMatrix = Matrix CellValue

-- | Helper that changes the order of arguments for getElem.
getFrom :: NWMatrix -> (Int, Int) -> CellValue
getFrom mat (i, j) = getElem i j mat
```

Damit wir die Richtungen potentieller Kandidatenschritte leicht anhand der dazugehörigen Werte vergleichen können, wäre es nützlich einen Typen für Tupel der Form `(Maybe Int, StepDirection)` mit Wert und Richtung eines Kandidaten festzulegen.

BLOCK-ID: «type-scores»

```
type CellScore = Maybe Int
type ScoredStep = (CellScore, StepDirection)
```

Wie strukturieren wir nun unsere Ergebnisse?

Offensichtlich brauchen wir die errechneten Alignments³⁸ und deren Wert. Weiterhin ist es sinnvoll, die ursprünglichen Eingabe für das Problem zu speichern, also brauchen wir ein `AlnInfo` Feld.

Wir speichern auch die errechnete `NWMatrix`, obwohl wir sie nicht unbedingt brauchen, da die Daten so leichter inspiziert werden können. Um aber die Übersichtlichkeit zu wahren, wird sie später von der Stringrepräsentation ausgeschlossen.

BLOCK-ID: «type-aln-result»

```
data AlnResult = AlnResult
  { alnInfo  :: AlnInfo
  , nwMat    :: NWMatrix
  , optAlns  :: [Aln]
  , optScore :: Int
  } deriving Eq
```

Mit diesen Datentypen sind wir in der Lage Alignmentprobleme und deren Lösungen zu formulieren.

3.3.2.2 Logik Nachdem wir in der letzten Sektion die notwendigen Datentypen formuliert haben, können wir jetzt die eigentliche Programmlogik implementieren mithilfe derer wir Sequenzen alinieren. Betrachten wir zunächst eine grobe Übersicht der Schritte, die wir unternehmen müssen um unser Alignmentproblem zu lösen:

1. Wir initialisieren die Matrix F mit Rekursionsankern,
2. befüllen dann die restlichen Zellen von F und
3. berechnen aus dem befüllten F das optimale Alignment.

3.3.2.2.1 Initialisieren Bevor wir die Matrix F initialisieren, definieren wir einige Hilfsfunktionen um uns in Matrizen zu orientieren, darunter die `distdiag` Funktion und das davon abgeleitete Kriterium `range`.

BLOCK-ID: «range-hs»

³⁸ Da eine Zelle mehr als einen Vorgänger haben kann, kann es auch mehr als einen Pfad durch die Matrix geben.


```
-- | Helper function to compute whether an alignment introduces too many gaps.
-- True if distance |i-j| from main diagonal is lesser or equal to allowed gaps.
distDiag :: MatIdx → Int → Bool
distDiag (i, j) gaps = abs (i-j) <= gaps

-- The distance criterion for candidate cell consideration.
range :: (Int, Int) → MatIdx → Bool
range (g1, g2) cell@(i, j)
  | i <= j    = distDiag cell g1
  | otherwise = distDiag cell g2
```

Mithilfe dieser Funktionen können wir die Befüllungsregeln für f_{0j} und f_{i0} festzulegen.

Zum Initialisieren benötigen wir die Rekursionsanker $f_{i0} = w_{\text{gap}} \cdot i$ und $f_{i0} = w_{\text{gap}} \cdot i$.³⁹

BLOCK-ID: «init-mat»

```
d1 :: MatIdx → [StepDirection]
d1 (i, j)
  | i == 1 && j == 1 = []
  | i == 1           = [Horizontal]
  | j == 1           = [Vertical]
  | otherwise        = []

f1 :: Bool → Int → Int → CellScore
f1 valid gap i = if valid
  then Just $ (i-1) * gap
  else Nothing

initFillFunc :: Cost → (Int, Int) → MatIdx → CellValue
initFillFunc (Cost {w_gap = gap}) gaps cell@(i, j)
  | i == 1    = lift f_1j
  | j == 1    = lift f_i1
  | otherwise = Nothing
  where
    prv  = d1 cell           :: [StepDirection]
    valid = range gaps cell  :: Bool
    f_i1  = f1 valid gap i    :: CellScore
    f_1j  = f1 valid gap j    :: CellScore
    lift  = (flip toValue) prv :: (CellScore → CellValue)

initMat :: AlnInfo → NWMatrix
initMat info@(AlnInfo {weights = cost}) = matrix (m+1) (n+1) init
  where
    (m, n) = seqLengths info
    init    = initFillFunc cost (gapCounts info)
```

Mit `initFillFunc` haben wir den Rekursionsanker definiert, um F zu initialisieren und mit `initMat` können wir ein initialisiertes F berechnen.

```
ghci> initMat info
```

```
[
  Just (0,[]) Just (-2,[←])  Nothing  Nothing  Nothing
  Just (-2,[↑])  Nothing     Nothing  Nothing  Nothing
  Just (-4,[↑])  Nothing     Nothing  Nothing  Nothing
           Nothing  Nothing     Nothing  Nothing  Nothing
           Nothing  Nothing     Nothing  Nothing  Nothing
]
```

³⁹ Das `matrix` Paket indiziert Matrizen ab 1 und nicht ab 0, weswegen sich alle i, j entsprechend verschieben.

Nothing	Nothing	Nothing	Nothing	Nothing
---------	---------	---------	---------	---------

3.3.2.2.2 Berechnen Um F zu befüllen müssen wir u.a. in der Lage sein die Kandidatenschritte für eine bestimmte Zelle f_{ij} zu bestimmen. Dafür können wir mithilfe `range` testen, welche der potentiellen Vorgänger aus C_{ij} für die Zelle f_{ij} infrage kommen.

BLOCK-ID: «candidates-hs»

```
-- | Find potential candidate cells, from which f_ij may be derived.
candidates :: (Int, Int) -> MatIdx -> [MatIdx]
candidates gaps cell@(i, j) = filter valid [d, h, v]
  where
    valid = range gaps :: (MatIdx -> Bool)

    d = (i-1, j-1)
    v = (i-1, j )
    h = (i , j-1)
```

Für bekannte Gapzahlen können wir uns nun die indizes potentiell valider Vorgänger ausgeben lassen.

```
ghci> let gaps = (1, 2)
ghci> candidates gaps (4, 3)
[(3,2),(4,2),(3,3)]
ghci> candidates gaps (2, 3)
[(1,2),(2,2)]
ghci> candidates gaps (3, 7)
[]
```

Auch wenn wir jetzt mit `candidates` die Kandidaten für eine Zelle finden können, erlaubt uns dies alleine noch nicht Zellen in F zu befüllen.

Dafür müssen wir den besten Kandidaten bestimmen. Um den besten Kandidaten zu wählen, brauchen wir zunächst das Gewicht w_{ij} , bzw. w_{gap} für einen Schritt q_k .

BLOCK-ID: «weight»

```
-- | Calculate the weight of a substitution.
substWeight :: (Eq a) => Cost -> (a, a) -> Int
substWeight (Cost {w_match = match, w_miss = miss}) (s1, s2)
  | s1 == s2 = match
  | otherwise = miss

-- | Calculate the weight of a step.
stepWeight :: AlnInfo -> Step -> Int
stepWeight (AlnInfo {weights = cost, seqA = s1, seqB = s2}) ((g, h), (i, j))
  | subst = substWeight cost (s1 ! i, s2 ! j)
  | otherwise = w_gap cost
  where
    subst = i == g + 1
           && j == h + 1
```

Die Substitutionskosten sind ziemlich trivial.

```
ghci> let cost = weights info
ghci> substWeight cost ('a', 'a')
1
ghci> substWeight cost ('x', 'y')
-1
```

Die `stepWeight` Funktion erlaubt es uns allerdings bereits Schrittgewichte zu bestimmen.

```

ghci> let diagA = ((3, 2), (4, 3))
ghci> let diagB = ((2, 2), (3, 3))
ghci> let vert  = ((2, 2), (2, 3))
ghci> let hori  = ((2, 2), (3, 2))
ghci> stepWeight info diagA
1
ghci> stepWeight info diagB
-1
ghci> stepWeight info vert
-2
ghci> stepWeight info hori
-2

```

Anhand der Schrittgewichte können wir nun versuchen die besten Kandidaten zu finden und daraus die Zellwerte bestimmen.

Dafür definieren wir weitere Helfer zum Umgang mit unseren Daten.

BLOCK-ID: «max-helpers»

```

-- | Score a candidate step.
stepScore :: AInfo -> NWMatrix -> Step -> ScoredStep
stepScore info mat step@(candidate, cell) = (score, dir)
  where
    val  = getFrom mat candidate :: CellValue
    w_q  = stepWeight info step  :: Int
    score = fmap ((+w_q).fst) val  :: CellScore
    dir  = fromStep step          :: StepDirection

-- | Find the highest candidate score.
maxCellScore :: [ScoredStep] -> CellScore
maxCellScore steps = max' Nothing steps
  where
    max' :: CellScore -> [ScoredStep] -> CellScore
    max' accum [] = accum
    max' accum ((s,_):sc)
      | s > accum = max' s sc
      | otherwise = max' accum sc

-- | Collect candidate steps with a specific candidate score.
filterSteps :: Maybe Int -> [ScoredStep] -> [StepDirection]
filterSteps _ [] = []
filterSteps m ((s, d):sc)
  | s == m    = d : filterSteps m sc
  | otherwise =   filterSteps m sc

-- | Helper to create a cell value from a candidate score
-- and a list of steps, by "lifting" the steps into the Maybe.
toValue :: CellScore -> [StepDirection] -> CellValue
toValue Nothing _ = Nothing
toValue (Just v) ds = Just (v, ds)

```

1. stepScore bildet die **ScoredStep** Tupel für unsere Kandidaten,
2. maxCellScore bestimmt aus diesen Tupeln den höchsten Kandidatenwert,
3. filterSteps bildet Liste aller **StepDirections** aus den Kandidatentupeln, die den mit maxCellScore gefundenen Wert haben und
4. toValue baut uns aus diesen beiden Teilen einen Zellwert zusammen, den wir in eine **NWMatrix** Schreiben können.

Jetzt können wir diese Arbeitsschritte zu einer `maxValue` Funktion zusammensetzen.

BLOCK-ID: «max-val»

```
-- | Given weighted step directions, find the optimal cell value.
maxValue :: [ScoredStep] -> CellValue
maxValue steps = max `seq` dirs `seq` -- use seq to force strict evaluation
  toValue max dirs
  where
    max = maxCellScore steps
    dirs = filterSteps max steps
```

Jetzt, wo wir in der Lage sind, den optimalen Zellwert aus einer Liste von Kandidaten zu bestimmen, können wir auch damit beginnen die Matrix zu befüllen.

Dazu bilden wir, mithilfe der `stepScore` Funktion, die Liste von `(Maybe Int, StepDirection)` Tupeln aller Kandidaten und bestimmen mit `maxValDirs` den Zellwert.

BLOCK-ID: «fill-cell»

```
fillCell :: AInInfo -> MatIdx -> NWMatrix -> NWMatrix
fillCell info cell mat = best `seq` -- use `seq` to force strict evaluation
  setElem best cell mat
  where
    gaps    = gapCounts info           :: (Int, Int)
    idxs    = candidates gaps cell     :: [MatIdx]
    score   = stepScore info mat       :: (Step -> ScoredStep)
    toStep  = (\c -> (c, cell))        :: (MatIdx -> Step)
    steps   = map (score.toStep) idxs  :: [ScoredStep]
    best    = maxValue steps           :: CellValue
```

Um F zu befüllen, gehen wir nun von links nach rechts und von oben nach unten durch die Matrix und füllen die einzelnen Zellen.

Um den Index der nächsten legalen Zelle zu finden, definieren wir die Funktion `nextCell`. Da eine Rückgabe optional und sogar sinnlos ist, wenn wir z.B. in der letzten Zelle angekommen sind, nutzen wir auch hier wieder die `Maybe` Monade. Die Funktion sollte einen `MatIdx` nehmen und dafür ggf. einen Nachfolger berechnen, also `Maybe MatIdx` zurückgeben.

BLOCK-ID: «next-cell»

```
-- | Compute the index of the next cell to calculate.
nextCell :: AInInfo -> MatIdx -> Maybe MatIdx
nextCell info cell@(i, j)
  | incCol    = Just nxcol
  | incRow    = Just nxrow
  | otherwise = Nothing
  where
    (m, n)    = seqLengths info
    (g1, g2)  = (gapCounts info)
    valid     = range (g1, g2)

    nxcol     = (i, j+1)
    nxrow     = (i+1, max 2 (i+1 - g2)) -- max 2 (i+1 - g2) clamps the col idx
    incCol    = j < n+1 && valid nxcol
    incRow    = i < m+1 && valid nxrow
```

`nextCell` implementiert **Reihenfolge der Berechnungen**, die wir zuvor schon graphisch dargestellt hatten.

Wenn wir den Rand der Matrix erreichen, oder zu einer Zelle kommen, welche aufgrund zu vieler notwendiger Gaps illegal wäre, springen wir zur ersten legalen Zelle der nächsten Zeile. Falls wir uns in der letzten Zeile befinden, sind wir fertig und müssen keinen weiteren Index zurückgeben.

Anmerkung

Auf dieser Basis ließe sich leicht eine Funktion `iterMat` definieren, welche eine Liste der Matrixindizes in der Reihenfolge der Berechnung zurückgibt.

```
iterMat :: AlnInfo → MatIdx → [MatIdx]
iterMat info cell =
  case nextCell info cell of
    Just next → cell : iterMat info next
    Nothing   → [cell]
```

Wir bekommen als Reihenfolge ab f_{43} :

```
ghci> iterMat info (4, 3)
[(4,3),(4,4),(4,5),(5,3),(5,4),(5,5),(6,4),(6,5)]
```

Nun ließen sich leicht Funktionen für alle Zellindizes ausführen indem man über das Ergebnis von `iterMat` `mappt`, oder `faltet`, was konzeptionell^a ungefähr so aussehen könnte:

```
fill' info = foldr (fillCell info) mat idxs
  where
    mat = initMat info
    idxs = iterMat info (2, 2)
```

Wir werden diesen Weg allerdings nicht gehen, sondern simple Rekursion nutzen.

^a Der gezeigte Code produziert noch kein korrektes Ergebnis und soll nur zur Veranschaulichung der Idee dienen.

Jetzt befüllen wir die restlichen Einträge in F , indem wir bei einer Zelle, mit initialisierten Vorgängerkandidaten, starten, diese befüllen und solange es eine nächste valide Zelle gibt mit dieser weitermachen. Wenn es keine Zellen mehr gibt, die wir befüllen müssen, sind wir fertig.

BLOCK-ID: «fill-from»

```
-- | Given AlnInfo, a matrix index and an NWMatrix with the
-- previous indices filled, fill the rest of the matrix,
-- beginning at the given index.
fillFrom :: AlnInfo → MatIdx → NWMatrix → NWMatrix
fillFrom info cell mat = next `seq` m `seq` -- use `seq` to force strict evaluation
  case next of
    Nothing → m -- last cell → fill cell and return
    Just next → fillFrom info next m -- cells left → fill cell and continue
  where
    next = nextCell info cell
    m = fillCell info cell mat
```

Wir können nun eine `NWMatrix` mit `initMat` initialisieren und diese dann mit `fillFrom` befüllen.

Damit können wir F auf Basis eines `AlnInfo` Records berechnen, indem wir die initialisierte Matrix ab $(2, 2)$ ⁴⁰ befüllen.

BLOCK-ID: «fill»

⁴⁰ Wir erinnern uns, dass wir eine Indexverschiebung, aufgrund der eins-basierten Indizierung von Matrizen, haben.

```
fill :: AlnInfo -> NWMatrix
fill info = let mat = initMat info
             in mat `seq` fillFrom info (2, 2) mat
             where counts = gapCounts info
```

Mit der fill Funktion können wir nun die Matrix befüllen.

```
ghci> let mat = fill info
ghci> mat
```

Just (0,[])	Just (-2,[←])	Nothing	Nothing	Nothing
Just (-2,[↑])	Just (1,[↖])	Just (-1,[←])	Nothing	Nothing
Just (-4,[↑])	Just (-1,[↑])	Just (0,[↖])	Just (0,[↖])	Nothing
Nothing	Just (-3,[↑])	Just (0,[↖])	Just (-1,[↖])	Just (-1,[↖])
Nothing	Nothing	Just (-2,[↑])	Just (-1,[↖])	Just (-2,[↖])
Nothing	Nothing	Nothing	Just (-3,[↖,↑])	Just (0,[↖])

Das Ergebnis entspricht unseren Erwartungen.

3.3.2.2.3 Backtracken Jetzt müssen wir aus der Matrix die Liste der Pfade von $(1, 1)$ nach $(M + 1, N + 1)$ extrahieren. Dieser Prozess wird als “Backtracken” bezeichnet. Aus diesen Pfaden ergeben sich die optimalen Alignments unserer Sequenzen.

Dazu definieren wir zunächst geeignete Funktionen, um Indizes und `StepDirection` Werte in die, durch die Richtungen bezeichneten, Vorgängerindizes zu überführen.

BLOCK-ID: «origs»

```
-- | Calculate the origin of a StepDirection from a particular position.
getOrig :: MatIdx -> StepDirection -> MatIdx
getOrig dest@(i, j) Diagonal  = (i-1, j-1)
getOrig dest@(i, j) Horizontal = (i , j-1)
getOrig dest@(i, j) Vertical   = (i-1, j )
```

```
origs :: MatIdx -> CellValue -> [MatIdx]
origs cell@(i, j) elem =
  case elem of
    Nothing    -> []
    Just (_, dirs) -> map (getOrig cell) dirs
```

Jetzt können wir Backtracken. Dafür bilden wir die Pfade von $(M + 1, N + 1)$ zum Ursprung $(1, 1)$, wobei wir rekursiv vorgehen.

Wenn wir die Menge der Pfade vom Ursprung zu sich selbst betrachten, dann sehen wir, dass diese nur den Pfad der Länge 1, beinhaltet, der aus dem Ursprung selbst besteht. Die Pfade von einer beliebigen Zelle zum Ursprung ergeben sich, indem wir die betrachtete Zelle an die Pfade ihrer Vorgänger anfügen. Wegen des Aufbaus der Matrix müssen alle Pfade irgendwann im Ursprung enden.

Aufgrund der Performancecharakteristika⁴¹ von Listen in Haskell berechnen wir die Pfade von hinten nach vorne.

BLOCK-ID: «find-rev-paths»

⁴¹ Vorne Anfügen hat Laufzeit $\mathcal{O}(1)$, aber hinten Anfügen hat Laufzeit $\mathcal{O}(n)$.

```
-- | Helper for backtracking, that determines the
-- (reverse) matrix paths for a given index.
findRevPaths :: NWMatrix -> MatIdx -> [Path]
findRevPaths _ (1, 1) = [(1, 1)]
findRevPaths mat cell@(i, j) = (prepend.collect.continue) cell0rigs
  where
    elem :: CellValue
    elem = getFrom mat cell

    cell0rigs = origs cell elem
    continue = map (findRevPaths mat)
    collect = concat
    prepend = map (cell:)
```

Achtung

Die Implementation des Backtrackings ist durch die genutzte Rekursion zwar elegant gelöst, aber nicht sehr performant.

Mehr dazu findet sich im Diskussionsteil.

Versuchen wir bspw. den Pfeilen aus Zelle (4, 2), mit Wert **Just** (-3, [↑]), zum Ursprung zu folgen.

```
ghci> mat
```

Just (0, [])	Just (-2, [←])	Nothing	Nothing	Nothing
Just (-2, [↑])	Just (1, [↖])	Just (-1, [←])	Nothing	Nothing
Just (-4, [↑])	Just (-1, [↑])	Just (0, [↖])	Just (0, [↖])	Nothing
Nothing	Just (-3, [↑])	Just (0, [↖])	Just (-1, [↖])	Just (-1, [↖])
Nothing	Nothing	Just (-2, [↑])	Just (-1, [↖])	Just (-2, [↖])
Nothing	Nothing	Nothing	Just (-3, [↖, ↑])	Just (0, [↖])

```
ghci> let revpaths = findRevPaths mat (4, 2)
```

```
ghci> revpaths
```

```
[(4,2),(3,2),(2,2),(1,1)]
```

Wie erwartet finden wir ein Alignment, mit zwei Schritten nach oben gefolgt von einem Schritt in der Diagonalen.

Aus den Pfaden ergeben sich die eigentlichen Alignments. Um einen Pfad in ein Alignment umzuwandeln, müssen wir für jeden Index (i, j) im Pfad, die relevanten Symbole s_i^1 und s_j^2 bestimmen und dann anhand der Schrittrichtung bestimmen, wo wir die Symbole, bzw. Gaps einbauen.

BLOCK-ID: «convert-path»

```
-- | Compute the alignment of two sequences from a matrix path.
convertPath :: SeqArr -> SeqArr -> Path -> Aln
convertPath _ _ [] = []
convertPath _ _ [p] = []
convertPath s1 s2 (p@(g, h):p'@(i, j):ps) =
  case dir of
    Diagonal -> (sym1, sym2) : rest
    Horizontal -> (Gap, sym2) : rest
    Vertical -> (sym1, Gap) : rest
  where
```

```

dir  = fromStep (p, p')
sym1 = Symbol (s1 ! i)
sym2 = Symbol (s2 ! j)
rest = convertPath s1 s2 (p':ps)

```

Damit sind wir in der Lage die Pfade aus einer Matrix zu bestimmen und diese in Alignments umzuwandeln. Den zuvor bestimmten Pfad müssen wir natürlich vorher noch umdrehen.

```

ghci> let path = (reverse.head) revpaths
ghci> let AlnInfo {seqA = s1, seqB = s2} = info
ghci> convertPath s1 s2 path
[('A', 'A'), ('G', '-'), ('T', '-')]

```

Mithilfe dieser Bausteine können wir eine backtrack Funktion definieren, welche aus einer gefüllten NW-Matrix für zwei Sequenzen die optimalen Alignments bestimmt.

BLOCK-ID: «backtrack»

```

-- | Given a filled in NWMatrix for two sequences,
-- determine the optimal alignments.
backtrack :: NWMatrix -> SeqArr -> SeqArr -> [Aln]
backtrack mat s1 s2 = map (convertPath s1 s2) paths
  where
    revpaths = findRevPaths mat (nrows mat, ncols mat)
    paths = map reverse revpaths

```

Für die Beispielsequenzen bekommen wir die üblichen Ergebnisse. Versuchen wir also zusätzlich ein Beispiel mit mehreren gleichwertigen Alignments. Dafür nehmen wir die Sequenzen AGTG und AAGTCC, mit denselben Kosten und derselben Gapzahl.

```

ghci> backtrack mat s1 s2
[[('A', 'A'), ('G', '-'), ('T', 'T'), ('A', 'G'), ('C', 'C')]]

```

3.3.2.2.4 Kombinieren Wenn wir zuerst die Matrix F befüllen und dann aus dem befüllten F die Alignments bestimmen haben wir das Alignmentproblem lösen.

Dazu definieren wir eine entsprechende align Funktion.

BLOCK-ID: «align»

```

align :: AlnInfo -> AlnResult
align info@(AlnInfo {seqA = s1, seqB = s2}) =
  case maybeScore of
    Just (score, _) -> AlnResult info mat alns score
    Nothing -> error "global alignment is undefined"
  where
    mat = fill info
    alns = backtrack mat s1 s2
    maybeScore = getFrom mat (nrows mat, ncols mat)

```

Jetzt können wir mit align den gesamten Prozess laufen lassen und bekommen einen `AlnResult` Wert zurück.

```

ghci> let res = align info
ghci> res
AlnResult { alnInfo = AlnInfo {g_max = 1, weights = Cost {w_match = 1, w_miss = -1,
  ↳ w_gap = -2}, seqA = array (2,6) [(2,'A'),(3,'G'),(4,'T'),(5,'A'),(6,'C')], seqB
  ↳ = array (2,5) [(2,'A'),(3,'T'),(4,'G'),(5,'C')]], optAlns =
  ↳ [[('A', 'A'), ('G', '-'), ('T', 'T'), ('A', 'G'), ('C', 'C')]], optScore = 0 }

```

Das Problem mag zwar gelöst sein, aber die Übersichtlichkeit lässt noch zu wünschen übrig.

3.3.2.3 Repräsentation Wir können durch das Anhängen von `deriving Show` an Datentypdefinitionen durch Haskell automatisch Repräsentationsfunktionen generieren lassen. Die so generierte Darstellung ist allerdings nicht gut für den menschlichen Genuss geeignet.

Stattdessen wollen wir ein paar eigene Repräsentationsfunktionen zur textuellen Darstellung definieren. Z.b. `showResult`, um `AlnResult` Werte menschenlesbar zu formatieren.

```
ghci> putStrLn $ showResult res
Given a pairwise alignment problem with the following key data:
```

```
g_max   = 1
w_match = 1
w_miss  = -1
w_gap   = -2
seqA    = "AGTAC"
seqB    = "ATGC"
```

The problem is optimally solved by the following 1 global alignment(s), with score 0:

```
AGTAC
|-|. |
A-TGC
```

Wie machen wir das?

3.3.2.3.1 Alignment Info Beginnen wir mit den grundlegendsten Werten, nämlich dem `AlnInfo` Datentyp. Da wir am Ende auch die Eckdaten des gelösten Problems zusammenfassen wollen, ergibt eine ansprechendere Darstellung hier Sinn.

BLOCK-ID: «show-aln-info»

```
showInfo info@(AlnInfo {g_max = g_max', weights = cost, seqA = s1, seqB = s2})
= "  g_max   = " ++ show g_max'      ++ "\n"
++ "  w_match = " ++ (show.w_match) cost ++ "\n"
++ "  w_miss  = " ++ (show.w_miss) cost  ++ "\n"
++ "  w_gap   = " ++ (show.w_gap) cost   ++ "\n"
++ "  seqA    = " ++ showSeq s1          ++ "\n"
++ "  seqB    = " ++ showSeq s2          ++ "\n"
where
  showSeq = show.elems
```

Wir beschränken uns auf eine zeilenweise Ausgabe der Daten, wobei wir die einzelnen Gewichte für die Darstellung extrahieren.

```
ghci> putStrLn $ showInfo info
g_max   = 1
w_match = 1
w_miss  = -1
w_gap   = -2
seqA    = "AGTAC"
seqB    = "ATGC"
```

3.3.2.3.2 Schritte Für `StepDirection` Werte nutzen wir einfach Pfeilsymbole.

BLOCK-ID: «show-steps»

```
-- | Use arrow symbols to display step directions.
```

```
instance Show StepDirection where
```

```
  show Diagonal = "↖"
```

```
  show Horizontal = "←"
```

```
  show Vertical = "↑"
```

3.3.2.3.3 Alignments Für Alignments (`type Aln = [(AlnChar, AlnChar)]`) wollen wir eine ähnliche Darstellung wählen, wie die vom Biopython-Projekt [20] genutzte Darstellung paarweiser Sequenzalignments.

Dabei werden die Buchstaben des Alignments getrennt durch ein Symbol untereinander geschrieben. Welches Symbol verwendet wird hängt davon ab, ob es sich um Match, Mismatch oder Gap handelt.⁴²

Alignment Symbole

`AlnChar` Werte werden, im Falle von Symbolen einfach so dargestellt wie die Buchstaben selbst, bzw. bei Gaps als `-`.

BLOCK-ID: «show-aln-char»

```
toChar :: AlnChar -> Char
```

```
toChar Gap      = '-'
```

```
toChar (Symbol c) = c
```

```
instance Show (AlnChar) where
```

```
  show = show.toChar
```

```
  showList xs ys = showList (map toChar xs) ys
```

Mit `toChar` können wir `AlnChar` Werte wieder zu Buchstaben machen. Dies nutzen wir um unsere Anzeigelogik einfach von der `show` Funktion für `Char` abzuleiten.

Information

Die `showList` Funktion ermöglicht die besondere Formatierung von Listendarstellungen. Z.b. sind Strings als Listen von Chars definiert, also `type String = [Char]`, werden aber nicht als solche dargestellt.

Die Standardrepräsentation für Listen nutzt eckige Klammern. Daher sähe die Liste von Buchstaben die dem String "hello world!" entsprechen normalerweise so aus:

```
['h','e','l','l','o',' ','w','o','r','l','d','!']
```

Aber, da `Chars` Typenklassen-Instanz für `Show` die `showList` Funktion definiert, wird dieser String als "hello world!" dargestellt.

Formatierung

Jetzt wo wir `AlnChar` Werte darstellen können definieren wir Hilfsfunktionen um Match- Mismatch- und Gapsymbole zu generieren und mit Zeilenumbrüchen fertig zu werden.

BLOCK-ID: «show-aln»

⁴² Bei Match `|`, bei Mismatch `.` und bei Gap `-`.

```

-- | Produce the proper signifier for two aligned symbols.
-- I.e., '-' for gaps, '|' for matches, and '.' for mismatches.
tag :: (AlnChar, AlnChar) -> Char
tag (Gap, Gap) = '-'
tag ( _, Gap) = '-'
tag (Gap,  _) = '-'
tag (x, y)
  | x == y      = '|'
  | otherwise   = '.'

-- | Helper for calculating lines in the string representation of an Aln.
breakAlnLines :: Int -> [String] -> [String]
breakAlnLines _ [] = []
breakAlnLines width strings = partlines : breakAlnLines width rest
  where
    (parts, rests) = unzip $ map (splitAt width) strings
    rest           = filter (not.null) rests
    break          = concat.(intersperse "\n")
    end            = if null rest then "" else "\n\n"
    partlines      = break parts ++ end

```

Mit tag können wir die korrekten Symbole für die Alignmentpositionen bestimmen und breakAlnLines bricht diese sauber in Zeilen um.

Information

Die intersperse Funktion nimmt einen Wert und eine Liste von Werten desselben Typs und fügt den übergebenen Wert zwischen den Listenelementen ein. Da intersperse kein Teil des **Prelude**, dem automatisch importierten Teil der Standardbibliothek, ist, müssen wir diese noch importieren.

BLOCK-ID: «align-imports» [3]

```
import Data.List (intersperse)
```

Mit diesen Helfern können wir geeignete Repräsentationsfunktionen schreiben.

3.3.2.3.4 Alignmentdarstellung Nun können wir die tag und breakAln Funktionen zusammensetzen um eine Darstellung für Alignments zu generieren. Dabei möchten wir, dass Alignments nach 80 Zeichen⁴³ umbrechen.

BLOCK-ID: «show-aln-helpers»

```

-- | Pretty print an alignment. This does not take terminal width into account,
-- but simply wraps after 80 symbols.
showAln :: Aln -> String
showAln aln = (concat.breakAlnLines 80) [r1, syms, r2]
  where
    syms      = map tag aln
    (s1, s2) = unzip aln
    [r1, r2] = map (map toChar) [s1, s2]

-- | Pretty print a list of alignments. Wraps the same way as showAln does.

```

⁴³ Die Standardbreite für Terminals beträgt, auch heute noch, 80 Spalten, da die Lochkarten von IBM 12x80 Format hatten.

```
showAlns :: [Aln] -> String
showAlns = concat.(intersperse "\n\n").(map showAln)
```

Mit showAlns stellen wir Alignments nun in menschenlesbarer Form dar.

```
ghci> let aln = (head.optAlns) res
ghci> showAln aln
"AGTAC\n|-.|.\nA-TGC"
ghci> let longaln = ((take 120).cycle) aln
ghci> putStrLn $ showAln longaln
AGTACAGTACAGTACAGTACAGTACAGTACAGTACAGTACAGTACAGTACAGTACAGTACAGTACAGTACAGTAC
|-|.|-|.|-|.|-|.|-|.|-|.|-|.|-|.|-|.|-|.|-|.|-|.|-|.|-|.|-|.|-|.|-|.|-|.
A-TGCA-TGCA-TGCA-TGCA-TGCA-TGCA-TGCA-TGCA-TGCA-TGCA-TGCA-TGCA-TGCA-TGCA-TGC

AGTACAGTACAGTACAGTACAGTACAGTACAGTACAGTAC
|-|.|-|.|-|.|-|.|-|.|-|.|-|.|-|.|-|.|-|.
A-TGCA-TGCA-TGCA-TGCA-TGCA-TGCA-TGCA-TGC
```

Wir sehen, dass bei einem Alignment mit 120 Zeichen nach der 80. Spalte ein Umbruch eingebaut wird.

3.3.2.3.5 Alignment Ergebnisse Jetzt da wir Alignments sauber darstellen können beschäftigen wir uns mit der Repräsentation des Gesamtergebnisses. Dafür definieren wir die Funktion showResult, welche uns eine gut menschenlesbare Textdarstellung generieren soll.

Wir haben außerdem in `AlnResult` die errechnete `NWMatrix` gespeichert. Dies würde, besonders bei größeren Matrizen, zu einer wenig übersichtlichen Darstellung führen, weswegen wir das `nwMat` Feld aus der Standarddarstellung ausschließen.

Block-ID: «show-aln-result»

```
showResult :: AlnResult -> String
showResult (AlnResult {alnInfo = info, nwMat = mat, optAlns = alns, optScore =
  → score})
  = "Given a pairwise alignment problem with the following key data:\n\n"
  ++ showInfo info ++ "\n\n"
  ++ "The problem is optimally solved by the following "
  ++ (show.length) alns ++ " global alignment(s),\n"
  ++ "with score " ++ show score ++ ":\n\n"
  ++ showAlns alns

-- | Exclude the nwMat field from the show representation.
instance Show AlnResult where
  show (AlnResult {alnInfo = info, optAlns = alns, optScore = score})
    = "AlnResult { "
    ++ "  alnInfo = " ++ show info
    ++ ", optAlns = " ++ show alns
    ++ ", optScore = " ++ show score
    ++ " }"
```

Nun bekommen wir das Ergebnis in einer Form präsentiert, die einen schnellen Überblick ermöglicht.

```
ghci> putStrLn $ showResult res
Given a pairwise alignment problem with the following key data:
```

```
g_max    = 1
w_match  = 1
w_miss   = -1
w_gap    = -2
seqA     = "AGTAC"
seqB     = "ATGC"
```

The problem is optimally solved by the following 1 global alignment(s), with score 0:

```
AGTAC
|-|. |
A-TGC
```

3.3.3 Ausführung

Um die definierten Funktionen für Anwender nutzbar zu machen, bedarf es ein wenig mehr als nur Datentypen, Logik und Repräsentationsfunktionen. Irgendwie müssen Nutzer die Software starten und Eingaben vornehmen können. Der GHCi-REPL ist dafür nur bedingt geeignet.

Beim Start eines Haskell Programms, wird die Funktion `main` ausgeführt. Diese hat den Typ `IO ()`, was bedeutet, dass sie mit Nebeneffekten belastete Ein- und Ausgaben durchführt und keinen Wert zurückgibt. Wie `Maybe`, ist auch `IO` eine Monade.

Unsere `main` Funktion soll die Kommandozeilenargumente lesen und dann, wenn welche übergeben wurden, die Anwendungslogik mit diesen ausführen oder, wenn keine übergeben wurden, interaktiv danach fragen.

BLOCK-ID: «main»

```
main :: IO ()
main = do
    args <- getArgs
    let action = if (not.null) args
                then runArgs
                else runInteractive
    result <- action args
    putStrLn $ showResult result
```

3.3.3.1 CLI Argumente Nutzer sollen das Programm in einer Shell starten und die notwendigen Argumente beim Aufruf übergeben können.

Wir erwarten genau 5 Argumente. Das erste ist ein Pfad zu einer FASTA-Datei mit mindestens 2 Sequenzen, das zweite ist die Anzahl zulässiger Gaps g_{\max} und die folgenden sind die Kosten, in der Reihenfolge w_{match} , w_{miss} und w_{gap} .

BLOCK-ID: «run-args»

```
-- | Cast a String to an Int.
toInt :: String -> Int
toInt = read :: String -> Int

-- | Run the application from a list of command line arguments.
runArgs :: [String] -> IO AInResult
runArgs (fp:g_maxArg:w_matchArg:w_missArg:w_gapArg:restArgs) = do
    -- parse file and unpack contents
    fastas <- readFasta fp
    let ((h1, s1):(h2, s2):restFastas) = fastas

    -- cast and unpack int args
    let intArgs = map toInt [g_maxArg, w_matchArg, w_missArg, w_gapArg]
    let [g_max, match, miss, gap] = intArgs
```

```
-- build input data and align sequences
let info = mkInfo g_max match miss gap s1 s2
return $ align info
```

Die runArgs Funktion erwartet also die Argumente

1. Dateipfad zu FASTA,
2. g_{\max} ,
3. w_{match} ,
4. w_{miss} und
5. w_{gap}

und kann nun genutzt werden um Sequenzen zu alinieren.

3.3.3.2 Interaktiv Falls ein Nutzer zur Übergabe der Argumente nicht die Kommandozeile nutzen will, definieren wir eine Funktion um diese stattdessen interaktiv übergeben zu können.

BLOCK-ID: «run-interactive»

```
runInteractive :: [String] -> IO AlnResult
runInteractive _ = do
  info <- askInfo
  return $ align info
```

Dafür brauchen wir eine Funktion um aus den übergebenen Werten einen `AlnInfo` Record zu bilden.

BLOCK-ID: «ask-info»

```
askInfo :: IO AlnInfo
askInfo = do
  putStrLn "Provide a path to a FASTA file containing at least two sequences."
  fp <- getLine
  ((h1, s1):(h2, s2):restFastas) <- readFasta fp

  putStrLn "How many gaps are allowed in the alignment?"
  g_maxStr <- getLine
  let g_max = read g_maxStr :: Int

  putStrLn "Weight of matches, mismatches and gaps?"
  putStrLn "(Seperate input by commas.)"
  weightStr <- getLine
  let weights = tokenize weightStr
  let [w_match, w_miss, w_gap] = map (read :: String -> Int) weights

  return $ mkInfo g_max w_match w_miss w_gap s1 s2
```

Und einige Helfer um mit Nutzereingaben umzugehen.

BLOCK-ID: «ask-helpers»

```
-- | Helper to split strings on a specific character.
split :: Char -> String -> [String]
split c xs = reverse $ split' [] "" xs
  where
    split' :: [String] -> String -> String -> [String]
    split' accum curr [] = curr:accum
    split' accum curr [x] = accum
    split' accum curr (x:xs)
      | x == c    = split' (curr:accum) "" xs
      | otherwise = split' accum (x:curr) xs
```

```
-- | Helper to discard leading and trailing whitespace from a string.
strip :: String -> String
strip = rstrip.lstrip
  where
    lstrip [] = []
    lstrip (x:xs)
      | x == ' ' = lstrip xs
      | otherwise = (x:xs)

    rstrip = reverse.lstrip.reverse

tokenize :: String -> [String]
tokenize s = map strip $ split ',' s
```

3.3.3.3 Parsen Nutzer sollen Sequenzen i.F.v. Dateien übergeben können. Definieren wir also eine Funktion, die einen Pfad zu einer FASTA-Datei nimmt und deren Sequenzen zurückgibt.

BLOCK-ID: «read-fasta»

```
-- | Convert a FastaSequence into a (header, sequence) tuple.
tuplify :: FastaSequence -> (String, String)
tuplify FastaSequence{ fastaHeader = h, fastaSeq = s } = (h, s)

-- | Read a FASTA file located at the given path, and produce a list of (header,
  ↳ sequence) tuples.
readFasta :: String -> IO [(String, String)]
readFasta path = do
  fastaString <- readFile path
  let fastas = parseFasta fastaString

  return $ map tuplify fastas
```

Wir nutzen hier das 3rd-Party Modul `fasta`. Dieses muss installiert sein, damit der Code funktioniert.

3.3.4 Modularisierung

Ein zentraler Zweck des Software-Engineerings ist es, die Komplexität von Softwarelösungen zu managen. Wichtige Techniken dafür sind Modularisierung und Hierarchisierung.

Nun fassen wir die definierten Blöcke mit den Funktionen und Datentypen im `Align.Naive.Data` Modul zusammen.

Bei der Modularisierung versuchen wir solche Teile des Systems in Komponenten zu bündeln, die ähnliche Zwecke haben und bei der Hierarchisierung versuchen wir eine konsistente (Halb-) Ordnung von Modulen und Submodulen zu bilden.

Unsere Software soll Sequenzen alinieren, weswegen es Sinn ergibt, sie zu einem `Align` Modul zusammenzufassen. Wir haben eine naive und eine sinnvolle Lösung implementiert und Scaffolding-Code für Nutzereingaben geschrieben. Diese können in `Align.Data` bzw. `Align.Naive.Data` gebündelt werden. Das Hauptmodul kann dann zusätzlich `Align.Data` exportieren.

3.3.4.1 Hauptmodul Module in Haskell sind Deklarationen der Form `module Name (<exports>) where <code>`. Ein Modul kann andere Submodule, die es importiert hat, exportieren. So können Submodule definiert werden.

Wir definieren das Modul `Align` und brauchen die Submodule `Align.Data` und `Align.Naive.Data`. Bevor wir mit `Align.Naive.Data` arbeiten können, müssen wir jedoch das `Align.Naive` Modul definieren.

Weiterhin soll `Align` beim import standardmäßig den Code in `Align.Data` verfügbar machen, nicht aber den in `Align.Naive`, da dies zu Namenskonflikten führen würde. Um dies zu vermeiden, nutzen wir einen qualifizierten Import für `Align.Naive`.

BLOCK-ID: «src/Align.hs»

```
module Align
  ( module Align.Data
  , module Align.Naive ) where

import Align.Data
import qualified Align.Naive
```

3.3.4.2 Naives Modul Dieses importiert einfach nur `Align.Naive.Data` und exportiert es dann wieder. Der einzige Zweck dieses Umwegs ist es, ein konsistentes Namensschema für unsere Module zu ermöglichen.

BLOCK-ID: «src/Align/Naive.hs»

```
module Align.Naive
  ( module Align.Naive.Data ) where

import Align.Naive.Data
```

Nun definieren wir das `Align.Naive.Data` Modul. Dieses enthält die naiven Datentypen und Funktionen zur Berechnung von Alignments.

BLOCK-ID: «src/Align/Naive/Data.hs»

```
{-# LANGUAGE InstanceSigs #-}

module Align.Naive.Data where

import Align.Data (Cost(..))
import Data.Char (toUpper)
import Data.Traversable (Traversable, fmapDefault, foldMapDefault)

<<base>>
<<naive-seq>>
<<naive-seq-classes>>
<<aln-char>>
<<aln-seq>>
<<successor>>
<<combination>>
<<combinations>>
<<alignments>>
<<naive-aln>>
<<naive-align-helpers>>
<<naive-align>>
```

3.3.4.3 Needleman-Wunsch Modul Das Modul zum Berechnen von Alignments mithilfe von NW, hat den folgenden Aufbau:

BLOCK-ID: «src/Align/Data.hs»

```
module Align.Data where

<<align-imports>>
```



```
<<types>>
<<helpers>>
<<computation>>
<<representations>>
```

Zunächst fassen wir die Typendeklarationen zusammen.

Block-ID: «types»

```
-- DATA TYPES
```

```
<<type-cost>>
<<type-seq>>
<<type-seq-arr>>
<<type-aln-info>>
<<type-aln-result>>
<<type-mat-parts>>
<<type-nw-matrix>>
<<type-scores>>
```

Anschließend bündeln wir die Hilfsfunktionen, die wir später in den Berechnungen verwenden.

Block-ID: «helpers»

```
-- HELPER FUNCTIONS
```

```
-- helpers for matrix computation
```

```
<<range-hs>>
<<candidates-hs>>
<<next-cell>>
<<weight>>
<<max-helpers>>
<<max-val>>
```

```
<<init-mat>>
<<fill-cell>>
<<fill-from>>
```

```
-- helpers for backtracking
```

```
<<origs>>
<<find-rev-paths>>
<<convert-path>>
```

Nun werden die eigentlichen Berechnungen zusammengefasst.

Block-ID: «computation»

```
-- COMPUTATIONS
```

```
<<fill>>
<<backtrack>>
<<align>>
```

Auch die Repräsentationsfunktionen können wir gruppieren.

Block-ID: «representations»

```
<<show-aln-char>>
<<show-steps>>
<<show-aln-info>>
<<show-aln-result>>
<<show-aln-helpers>>
<<show-aln>>
```

3.3.4.4 Anwendungsmodule Das Folgende muss nicht groß kommentiert werden. Für die Anwendungslogik definieren wir ein `Main` Modul.

Block-ID: «app/Main.hs»

```
module Main where

import Align.Data (mkInfo, align, mkInfo, AlnInfo (..), showResult, AlnResult (..))
import Data.Fasta.String.Parse (parseFasta)
import Data.Fasta.String.Types (FastaSequence (..))
import System.Environment (getArgs)

<<read-fasta>>

<<run-args>>

<<ask-helpers>>
<<ask-info>>
<<run-interactive>>

<<main>>
```

3.3.5 Paketierung

Haskell-Projekte können mithilfe der *Common Architecture for Building Applications and Libraries* (**Cabal**) gebaut und paketiert werden. Dafür wird eine `project.cabal` Datei im Project-Root angelegt.

Information

Im Folgenden wird der Aufbau eines Cabal-Files im literarischen Programmierstil erklärt. Da Cabal allerdings keine Kommentare vor der Versionsangabe akzeptiert und Entangle^a Kommentare nutzt, um Codeblöcke auseinanderzuhalten, produziert dieses Beispiel keine validen Dateien.

Aus diesem Grund wurde dem Projekt mit der `seafovl.cabal` Datei ein valides, manuell geschriebenes, Cabal-File, mit identischem Inhalt, beigefügt.

^a Das Tool, welches das Generieren von Source-Dateien, aus Fließtext mit eingewobenen Codeblöcken, ermöglicht.

Der Aufbau eines Cabal-Files sieht folgendermaßen aus:

Block-ID: «seafovl.cabal»

```
cabal-version:      3.4

<<cabal-info>>

common common-options
  default-language: Haskell2010
  ghc-options: -Wall -O +RTS -ssderr

<<cabal-library>>
<<cabal-executable>>
```

Nach einer Präambel mit der Cabal-Version, folgen allgemeine Projektinformationen, Deklarationen von

geteilten Optionen und Build-Targets. Die Build-Targets können entweder ausführbare Dateien oder Libraries zum einbinden in anderen Projekten sein.

Die folgenden Eck-Daten beschreiben das Projekt:

BLOCK-ID: «cabal-info»

```

name:          seafovl
author:        Fynn Freyer
synopsis:      SEquence Aligner with Formally Verified Logic
description:   Seafovl is a formally verified sequence aligner.

license:       MIT
license-file:  LICENSE
copyright:     Copyright 2024 Fynn Freyer

-- Package version conforms to https://pvp.haskell.org
--      +-+----- breaking API changes
--      | | +----- non-breaking API additions
--      | | | +---- code changes with no API change
version: 0.1.0.0
maintainer:    fynn.freyer@student.htw-berlin.de

category:      Data
build-type:    Simple

extra-doc-files:  README.md,
                  CHANGELOG.md

```

Damit wir die Software bauen können, brauchen wir Build-Targets. Von diesen können wir auch mehrere definieren.

Zum Einen wollen wir anderen Programmierern ermöglichen den geschriebenen Code in ihre eigenen Projekte einzubinden. Dies entspricht einem library Build.

Dabei müssen wir angeben welche Module unsere Bibliothek exportiert und von welchen externen Paketen sie abhängt.

BLOCK-ID: «cabal-library»

```

library
  import:          common-options
  exposed-modules: Align,
                  Align.Data,
                  Align.Naive,
                  Align.Naive.Data,
  build-depends:   base ^>=4.17.1.0,
                  array ^>=0.5,
                  matrix ^>=0.3
  hs-source-dirs:  src

```

Zum Anderen sollen Nutzer die Software direkt ausführen können. Dies entspricht einem executable Build.

Dabei müssen wir zusätzlich zu den Abhängigkeiten, welche den zuvor definierten library Build enthalten, angeben welche Datei die main Funktion enthält und ob ggf. weitere Module als das mit der main eingebunden werden sollen.

BLOCK-ID: «cabal-executable»

```
executable seafovl
  import:      common-options
  main-is:     Main.hs
  build-depends: base ^>=4.17.1.0,
                fasta ^>=0.10,
                seafovl
  hs-source-dirs: app
```

Nun können wir die Anwendung mit `cabal run` kompilieren und starten.

3.3.6 Test

Können wir mit NW nun auch längere Sequenzen verarbeiten?

3.3.6.1 Testdaten Im `assets/` Ordner liegen die FASTA-Dateien `ins_prot.fa`, `ins.fa` und `pol.fa`.

Die `ins.fa` und `ins_prot.fa` Dateien enthalten die DNA- und Aminosäuresequenzen des Insulin-Proteins von Menschen⁴⁴ und Dromedaren.⁴⁵ In beiden Organismen haben die Sequenzen dieselbe Länge. Im Fall der DNA 331 Basen und bei den Proteinen 110 Aminosäuren.

Die `pol.fa` Datei enthält Sequenzen der POL Region für das humane und simiane Immundefizienz-Virus (**HIV** und **SIV**), mit Längen von 2739, bzw. 3180, Basen.

3.3.6.2 Insulin Wir setzen für beide Dateien Kosten mit $w_{\text{match}} = 1$, $w_{\text{miss}} = -1$ und $w_{\text{gap}} = -2$. Beim Alinieren der Aminosäuren erlauben wir 20 Gaps, bei der DNA 40.

Information

Die folgenden Ausgaben sind gekürzt, wobei Auslassungen mit `...` kenntlich gemacht wurden.

Weiterhin wurde eine Messung mit `/usr/bin/time -p cmd` vorgenommen. Diese soll nicht repräsentativ sein, sondern hat nur den Sinn dem Leser eine ungefähre Ahnung über die notwendige Ausführungszeit vermitteln. Der angegebene Wert entspricht dem Wert `real`.

```
$ cabal run -- seafovl assets/ins_prot.fa 20 1 -1 -2
```

Given a pairwise alignment problem with the following key data:

```
g_max    = 20
w_match  = 1
w_miss   = -1
w_gap    = -2
seqA     = "MALWMRLLPLLALLALWGPDPAAAFVNQHLCGSHL..."
seqB     = "MALWTRLLALLALLALGAPTPARAFANQHLCGSHL..."
```

The problem is optimally solved by the following 1 global alignment(s), with score
↪ 74:

⁴⁴ Die Accession-Number ist [AAA59172.1](#) für das Protein, bzw. [AH002844.2](#) mit Region `join(2424..2610,3397..3542)` für die DNA.

⁴⁵ Die Accession-Number ist [KAB1251309.1](#) für das Protein, bzw. [JWIN03000075](#), mit Region `join(6667075..6667261,6667771..6667916)` für die DNA.

```

MALWMRLPLLALLALWGPDPAAAFVNQHLCSHLVEALYLVCGERGFFYTPKTRREAEDLQVGQVELGGG
||||.||||.|||||.|||.|||.|||||.|||||.|||||.|||||.|||||.|||||.|||||.|||||.
MALWTRLLALLALLALGAPTPARAFANQHLCSHLVEALYLVCGERGFFYTPKARREVDTQVGGVELGGG

```

```

PGAGSLQPLALEGSLQKRGIVEQCCTSIQSLYQLENYCN
||||.||||.|||||.|||.|||.|||||.|||||.|||||.|||||.|||||.
PGAGGLQPLGPEGRPQKRGIVEQCCASVCSLYQLENYCN

```

Wenn wir die Aminosäuresequenzen alinieren, brauchen wir 0,19 Sekunden.

Machen wir mit der DNA weiter.

```
$ cabal run -- seafovl assets/ins.fa 40 1 -1 -2
```

Given a pairwise alignment problem with the following key data:

```

g_max   = 20
w_match = 1
w_miss  = -1
w_gap   = -2
seqA    = "ATGGCCCTGTGGATGCGCCTCTGCCCTGC..."
seqB    = "ATGGCCCTGTGGACACGCCTGCTGGCCCTGC..."

```

The problem is optimally solved by the following 4 global alignment(s), with score
↪ 225:

```

ATGGCCCTGTGGATGCGCCTCTGCCCTGCTGGCGCTGCTGGCCCTCTGGGGACCTGACCCAGCCGAGC
|||||.|||||.|||||.|||||.|||||.|||||.|||||.|||||.|||||.|||||.|||||.
ATGGCCCTGTGGACACGCCTGCTGGCCCTGCTGGCCCTGCTGGCCCTCGGGGCGCCACCCCGCCCGGGC

```

...

```

TGCTGTACCAGCATCTGCTCCCTCTACCAGCTGGAGAACTACTGCAACTAG
|||||.|||||.|||||.|||||.|||||.|||||.|||||.|||||.|||||.|||||.
TGCTGCGCCAGCGTCTGCTCGCTCTACCAGCTGGAGAACTACTGCAACTAG

```

...

Hier benötigen wir bereits 6,05 Sekunden.

3.3.6.3 HIV und SIV Auch hier setzen wir $w_{\text{match}} = 1$, $w_{\text{miss}} = -1$ und $w_{\text{gap}} = -2$, aber erlauben beim Alinieren 200 Gaps.

Vorsicht

Der folgende Aufruf wird nicht in einer angemessenen Zeit fertig werden.
Die Details dazu befinden sich im Diskussionsteil.

```
$ cabal run -- seafovl assets/pol.fa 200 1 -1 -2
```

...

3.4 Verifikation

Wir werden im Folgenden die zentralen Bestandteile des implementierten Softwaresystems einer mathematischen Prüfung unterziehen.

3.4.1 Grundlegende Definitionen

Rufen wir uns kurz die Definitionen grundlegender Datentypen und Funktionen in Erinnerung.

3.4.1.1 Datentypen Alle Berechnungen basieren in irgendeiner Art und Weise auf den zentralen Eckdaten des Alignmentproblems. Dies sind die zu betrachtenden Sequenzen und deren Längen, die Anzahl erlaubter Gaps und das Gewicht von Substitutionen und Gaps.

Der Typ für Gewichte ist `Cost` und hat Felder für w_{match} , w_{miss} und w_{gap} .

```
-- | Record type for costs.
data Cost = Cost {w_match :: Int, w_miss :: Int, w_gap :: Int} deriving (Eq, Show)
```

Die Felder halten Werte in \mathbb{Z} .

Wir fassen alle relevanten Eckdaten, inklusive der Gewichte zu `AlnInfo` Werten zusammen, bzw. bestimmen sie im Falle der Sequenzlängen mit der `seqLengths` Funktion.

```
-- | Record with key data of the alignment problem.
data AlnInfo = AlnInfo
  { g_max    :: Int
  , weights  :: Cost
  , seqA     :: SeqArr
  , seqB     :: SeqArr
  } deriving (Eq, Show)

-- | Compute sequence lengths for an AlnInfo record.
seqLengths :: AlnInfo -> (Int, Int)
seqLengths AlnInfo {seqA = s1, seqB = s2} = (length s1, length s2)
```

Der `AlnInfo` Datentyp hat Felder für g_{max} , die Gewichte und die Sequenzen s^1 und s^2 .

Als Hilfsfunktion um für die Sequenzen eines `AlnInfo` Wertes die Längen zu bestimmen, haben wir `seqLengths` definiert, welche das geordnete Paar $(|s^1|, |s^2|)$ produziert.

Die relevanten Datentypen im Zusammenhang mit Matrizen sind wie folgt definiert.

```
-- | Matrix indices.
type MatIdx = (Int, Int)
-- | Path through a matrix.
type Path = [MatIdx]
-- | Steps are `MatIdx` tuples of the form `(origin, destination)`.
type Step = (MatIdx, MatIdx)
-- | Datatype to denote step directions.
data StepDirection = Diagonal | Horizontal | Vertical deriving Eq
-- | If defined, tuple of cell value and list of precursors.
type CellValue = Maybe (Int, [StepDirection])
-- | A Needleman-Wunsch matrix.
type NWMatrix = Matrix CellValue
```

3.4.1.2 Grundlegende Funktionen Betrachten wir auch ein paar grundlegende Funktionen, aus welchen wir die Hauptfunktionen des Programms komponieren.

3.4.1.2.1 Gewichte Im Block «weight» haben wir Funktionen zum Umgang mit Gewichten definiert.

Substitutionskosten

Die Formel (2.1.2) gibt die Definition für w_{ij} wie folgt.

$$w_{ij} = \begin{cases} w_{\text{match}} & , s_i^1 = s_j^2 \\ w_{\text{miss}} & , \text{Andernfalls} \end{cases}$$

Wir haben für Substitutionskosten die Funktion `substWeight` definiert, deren Eingaben die zu verwendenden Gewichte und die beiden Symbole s_i^1 und s_j^2 sind.

```
-- | Calculate the weight of a substitution.
substWeight :: (Eq a) => Cost -> (a, a) -> Int
substWeight (Cost {w_match = match, w_miss = miss}) (s1, s2)
  | s1 == s2 = match
  | otherwise = miss
```

Wenn $s_i^1 = s_j^2$, dann w_{match} und sonst w_{miss} entspricht genau der Definition von w_{ij} .

Schrittweite

Nach (3.2.15) ergibt sich der Wert eines Schrittes wie folgt.

$$w(q_k) = \begin{cases} w_{ij}, & g \neq i \wedge h \neq j \\ w_{\text{gap}}, & \text{Andernfalls} \end{cases}$$

Um $w(q_k)$ darzustellen, haben wir die Funktion `stepWeight` formuliert.

```
-- | Calculate the weight of a step.
stepWeight :: AlnInfo -> Step -> Int
stepWeight (AlnInfo {weights = cost, seqA = s1, seqB = s2}) ((g, h), (i, j))
  | subst = substWeight cost (s1 ! i, s2 ! j)
  | otherwise = w_gap cost
  where
    subst = i == g + 1
          && j == h + 1
```

Aus (3.2.11) folgen $g \neq i \iff i = g + 1$ und $h \neq j \iff j = h + 1$. Daher sehen wir, dass die mit $i = g + 1 \wedge j = h + 1$ gegebene Aussage `subst` äquivalent zu der ersten Bedingung in (3.2.15) ist. Wir haben außerdem bereits gesehen, dass `substWeight` eine akkurate Entsprechung für das Gewicht w_{ij} einer Substitution ist.

Da wir, identisch zu $w(q_k)$, in allen sonstigen Fällen w_{gap} zurückgeben, muss auch `stepWeight` (3.2.15) entsprechen.

Schrittwertbestimmung

Wir arbeiten nicht direkt mit Zellwerten, sondern mit Werten von `ScoredStep`. Um diese zu erzeugen, haben wir im «max-helpers» Block die `stepScore` Funktion definiert.

```
-- | Score a candidate step.
stepScore :: AlnInfo -> NWMatrix -> Step -> ScoredStep
stepScore info mat step@(candidate, cell) = (score, dir)
  where
    val = getFrom mat candidate :: CellValue
    w_q = stepWeight info step :: Int
```

```
score = fmap ((+w_q).fst) val :: CellScore
dir   = fromStep step      :: StepDirection
```

Für den Kandidaten *candidate* mit Indizes (g, h) binden wir den Wert f_{gh} an den Namen *val*.

Als wir zuvor die Implementation von *stepWeight* besprochen, sahen wir, dass das Ergebnis von *stepWeight info* für den Schritt q_k dem Schrittgewicht $w(q_k)$ entspricht. Dieses binden wir an den Namen *w_q*. Außerdem wandeln wir den übergebenen Schritt in einen *StepDirection* Wert um und benennen diesen mit *dir*.

Der Funktionswert von *stepScore* für einen Schritt q , entspricht dem Tupel $(w(q), \text{dir})$, mit Gewicht und Richtung des Schritts.

3.4.1.2.2 Gaps Wir müssen auch die Funktionen betrachten, welche wir nutzen um mit Lücken im Alignment umzugehen.

Gapzahlen

Die Gapzahl g^x einer Sequenz s^x ist in (3.2.17) durch $g^x = g_{\max} + (L - |s^x|)$ gegeben, wobei L die Länge der längsten Sequenz bezeichnet. Um unsere g^x zu berechnen haben wir im «type-aln-info» Block *gapCounts* definiert.

```
-- | Compute gap numbers for an AlnInfo record.
gapCounts :: AlnInfo -> (Int, Int)
gapCounts info@(AlnInfo {g_max = g})
  = let (m, n) = seqLengths info
        l      = max m n
        in (g + (1 - m), g + (1 - n))
```

In der Definition von *gapCounts* binden wir die Sequenzlängen an die Namen *m* und *n* und bestimmen anschließend *l* als den größeren der beiden Werte. Danach berechnen wir g^1 und g^2 als $g^1 == g + (1 - m)$ bzw. $g^2 == g + (1 - n)$, wobei g unser g_{\max} bezeichnet. Die Definitionen sind offensichtlich identisch.

Distanz zur Hauptdiagonalen

Betrachten wir nun die Gleichung (3.2.18) mit $\text{dist}_{\text{diag}}(i, j, g) = |i - j| \leq g$. Ihr entspricht die im «range-hs» definierte *distDiag* Funktion.

```
-- | Helper function to compute whether an alignment introduces too many gaps.
-- True if distance |i-j| from main diagonal is lesser or equal to allowed gaps.
distDiag :: MatIdx -> Int -> Bool
distDiag (i, j) gaps = abs (i-j) <= gaps
```

Wir haben *distDiag (i, j) gaps = abs (i-j) <= gaps*, wobei *abs* den Absolutwert des Arguments berechnet. Dies entspricht genau der Definition von $\text{dist}_{\text{diag}}$.

Auswahlkriterium

Das in (3.2.19) definierte Prädikat *range* sagt, wenn es auf eine Zelle mit Koordinaten (i, j) angewandt wird, aus, ob der Abstand von f_{ij} zur Hauptdiagonalen zu groß ist um plausibel zu sein.

$$\text{range}(i, j) = \begin{cases} \text{dist}_{\text{diag}}(i, j, g^1) & , i \leq j \\ \text{dist}_{\text{diag}}(i, j, g^2) & , \text{Andernfalls} \end{cases}$$

Um dieses darzustellen, haben wir im «range-hs» Block die Funktion *range* definiert.


```
-- The distance criterion for candidate cell consideration.
range :: Int -> Int -> MatIdx -> Bool
range g1 g2 cell@(i, j)
  | i <= j    = distDiag cell g1
  | otherwise = distDiag cell g2
```

In `range` machen wir zunächst eine Fallunterscheidung zwischen $i \leq j$ und beliebigen sonstigen Fällen. Diese Bedingung entspricht dem ersten Pattern `i <= j` in `range`. Wenn $i \leq j$, dann nimmt `range` den Wert `distDiag cell g1`, also. $\text{dist}_{\text{diag}}(i, j, g^1)$ an. Das zweite Pattern deckt alle sonstigen Fälle ab und produziert den Wert $\text{dist}_{\text{diag}}(i, j, g^2)$.

Da `range` dieselben Fallunterscheidungen vornimmt, und dieselben Werte produziert, ist es offensichtlich äquivalent zu `range`.

3.4.1.3 Kandidatenwahl Wir haben in (3.2.24) die Regel $\text{candidates}(i, j) = \{c = (g, h) \in C_{ij} \mid \text{range}(c)\}$ zur Kandidatenwahl formuliert, wobei $C_{ij} = \{(i-1, j-1), (i-1, j), (i, j-1)\}$.

In «candidates-hs» haben wir eine entsprechende Funktion `candidates` formuliert.

```
-- | Find potential candidate cells, from which f_ij may be derived.
candidates :: (Int, Int) -> MatIdx -> [MatIdx]
candidates gaps cell@(i, j) = filter valid [d, h, v]
  where
    valid = range gaps :: (MatIdx -> Bool)

    d = (i-1, j-1)
    v = (i-1, j )
    h = (i , j-1)
```

Die Funktion `filter` nimmt ein Prädikat und eine Liste und gibt eine Liste aller Einträge zurück, die das Prädikat erfüllen.

Betrachten wir nun die im `where` Block befindlichen Definitionen, dann zeigt sich, dass das Prädikat `valid` durch partielle Anwendung des `gaps` Arguments (g^1 und g^1) auf die `range` Funktion entsteht. Wenn wir anschließend in `[d, v, h]` die Definitionen von `d`, `v` und `h` einsetzen, ergibt sich die Liste `[(i-1, j-1), (i-1, j), (i, j-1)]`, deren Elemente dieselben wie die in C_{ij} sind.

Wir sehen so, dass auch die Kandidatenwahl korrekt implementiert ist.

3.4.2 Maximierung

Um den größten Schrittkandidaten zu finden haben wir im «max-helpers» Block die `maxValue` Funktion definiert.

```
-- | Given weighted step directions, find the optimal cell value.
maxValue :: [ScoredStep] -> CellValue
maxValue steps = max `seq` dirs `seq` -- use seq to force strict evaluation
  toValue max dirs
  where
    max = maxCellScore steps
    dirs = filterSteps max steps
```

Die eigentliche Maximierung des Kandidatenwertes findet in der im selben Block definierten `maxCellScore` Funktion statt und `maxValue` dient nur dem Umgang mit den Schrittrichtungen und der Zusammenfassung des resultierenden Wertes. Da uns bei der Maximierung nur die `Int` Komponente des Zellwerts interessiert, werden wir auch hier nur die damit involvierte `maxCellScore` Funktion untersuchten.

```
-- | Find the highest candidate score.
maxCellScore :: [ScoredStep] → CellScore
maxCellScore steps = max' Nothing steps
  where
    max' :: CellScore → [ScoredStep] → CellScore
    max' accum [] = accum
    max' accum ((s,_):sc)
      | s > accum = max' s sc
      | otherwise = max' accum sc
```

Die Liste `steps` ist vom Typen `steps :: [ScoredStep]`, der Typ `ScoredStep` ist als `(CellScore, StepDirection)` definiert und der Typ `CellScore` entspricht `Maybe Int`.

Wir betrachten Listen als Folgen von Werten. Mit Länge $|steps| = n$ definieren wir:

$$steps = (step_i)_{i \in J_n} \quad (3.4.1)$$

Da nur die `CellScore` Komponente Relevanz für die Frage des maximalen Zellwertes hat, definieren wir auch eine Funktion $score : (f, d) \mapsto f$, welche auf diese Komponente abbildet und die Folge dieser Komponenten `scores`.

$$s_i = score(step_i) \quad (3.4.2)$$

Der größte Zellwert ist dann durch $\max\{s_i\}_{i \in J_n}$ gegeben.⁴⁶

Die Aussage, die es zu zeigen gilt, nämlich dass das Ergebnis von `maxVal steps` dem größten Zellwert in `steps` entspricht, können wir also folgendermaßen notieren:

$$maxValue(steps) = \max\{s_i\}_{i \in J_n} \quad (3.4.3)$$

`maxValue steps` ist definiert als `max' Nothing steps`, wobei `max'` eine im `where` Block definierte, rekursive Funktion vom Typ `(CellScore → [ScoredStep] → CellScore)` ist.

Um (3.4.3) zu zeigen, betrachten wir also zunächst `max'`.

```
max' :: CellScore → [ScoredStep] → CellScore
max' accum [] = accum
max' accum ((s,_):sc)
  | s > accum = max' s sc
  | otherwise = max' accum sc
```

Wir sehen, dass `max'` bei nichtleeren Listen $(x:xs)$ das erste Element der Liste entfernt und sich selber wieder mit dem Rest `xs` aufruft. Bei leeren Listen terminiert `max'`. Da `max'` auf `steps` angewandt wird ist also offensichtlich, dass es genau n Rekursionsschritte macht.

Wir definieren jetzt für beliebige Rekursionsschritte von `max' accum sc` eine Variable $accum_i$ als den Wert des `accum` Arguments nach Schritt $i \in J_n$. Sei außerdem $accum_0$ das initial übergebene Argument.

Wir erhalten mit $(accum_i)_{i \in J_n}$ die Folge der Funktionsargumente `accum` für rekursive Aufrufe von `max'`.

$$accum_i = \begin{cases} s_i & , s_i > accum_{i-1} \\ accum_{i-1} & , \text{Andernfalls} \end{cases} \quad (3.4.4)$$

⁴⁶ Hier bezeichnet $\{a_i\}_{i \in M}$ die Menge der Folgenglieder der Folge $(a_i)_{i \in M}$.

Information

An dieser Stelle sei auf die Anordnungsregeln von **Maybe** hingewiesen.

Für einen beliebigen Wert $x :: (\text{Ord } a) \Rightarrow a$ eines anordenbaren Typs a gilt:

$$\text{Nothing} < \text{Just } x \quad (3.4.5)$$

Weiterhin gilt für zwei beliebige Werte x und y eines anordenbaren Typs a :

$$\text{Just } x < \text{Just } y \iff x < y \quad (3.4.6)$$

Der Typ **Int** ist anordenbar, also gilt auch $\text{Nothing} < \text{Just } n$ für beliebige $n :: \text{Int}$ und $\text{Just } m < \text{Just } n$ für beliebige $m :: \text{Int}$ und $n :: \text{Int}$ mit $m < n$.

Die Definition von accum_i für $i < n$ entspricht dem Fall $\text{max}' \text{ accum } ((s, _):sc)$ in der Definition von max' und wir sehen, dass accum_n der Wert des accum nach dem letzten Rekursionsschritt ist. Aus $\text{max}' \text{ accum } [] = \text{accum}$ ergibt sich, dass accum_n dem Funktionswert von max' entspricht.

$$\text{max}'(\text{accum}_0, \text{steps}) = \text{accum}_n \quad (3.4.7)$$

Dementsprechend ist auch der Wert von maxVal durch accum_n gegeben und wir sehen, dass wir äquivalent zu (3.4.3) auch zeigen können, dass accum_n den größten Kandidatenwert bestimmt.

$$\text{accum}_n = \max\{\text{scores}_i\}_{i \in J_n} \quad (3.4.8)$$

Wir vermuten, dass accum nach jedem Rekursionsschritt $i \in J_n$ den größten bisher gesehenen Wert hält.

$$\forall i \in J_n : \text{accum}_i = \max\{\text{scores}_j\}_{j \in J_i} \quad (3.4.9)$$

Wenn wir (3.4.10) zeigen können, dann folgt daraus (3.4.8) und damit widerum (3.4.3)

Entsprechend ergibt sich als Induktionshypothese, dass accum den größten der bisher geprüften Werte enthält.⁴⁷

$$\text{accum}_i = \max\{\text{scores}_j\}_{j \in J_i} \quad (3.4.10)$$

3.4.2.1 Induktionsanker Um den Induktionsanker zu formulieren, betrachten wir solche Fälle, in denen die Eingabeliste steps leer ist und solche in denen sie Elemente hat.

3.4.2.1.1 Leere Liste Für leere Eingaben ergibt sich $\{\text{scores}_i\}_{i \in J_n} = \emptyset$. Das Maximum der leeren Menge ist undefiniert und wir repräsentieren \perp als **Nothing**.

⁴⁷ Diese Annahme können wir, wie wir später sehen, nur dann machen, wenn beim ersten Aufruf $\text{accum} == \text{Nothing}$ gilt, was per Definition von maxVal der Fall ist.

$$\max \emptyset = \perp \quad (3.4.11)$$

Es ist wichtig zu beachten, dass wir durch (3.4.11) auf den Startwert **Nothing** für `accum` festgelegt sind.

$$\text{accum}_0 = \max \emptyset = \perp$$

Dies ist durch die Definition von `maxVal` gegeben.

Mit `max' accum [] = accum` bekommen wir `max' Nothing [] = Nothing` und sehen, dass mit $\text{accum}_0 = \perp$ (3.4.10) für leere Eingaben gilt.

3.4.2.1.2 Nicht-leere Liste Sei $s_1 = (s, d)$, dann ergibt sich für nicht leere Eingaben $\{\text{scores}_1\} = \{s\}$.

$$\max\{\text{scores}_1\} = s \quad (3.4.12)$$

Bei einer einelementigen Menge von Tupeln ist der maximale Wert der ersten Komponenten immer der Wert der ersten Komponente des einen Mengeelements.

Es gibt genau zwei Möglichkeiten für den Wert von s .⁴⁸

$$\begin{aligned} s &= \text{Nothing} & , \text{I} \\ s &= \text{Just } f & , \text{II} \end{aligned}$$

Fall I

Für `steps == [(Nothing, d)]` ergibt sich `max' Nothing ((Nothing, d) : [])`. Die Bedingung `Nothing > Nothing` gilt nicht, weswegen wir in die zweite Bedingung, mit Resultat `max' accum sc`, durchfallen, welche immer wahr ist. Wenn wir nun die Werte einsetzen, ergibt sich `max' Nothing []`, was per definitionem dem Wert **Nothing** entspricht.

$$\text{accum}_1 = \perp = s$$

Damit gilt (3.4.12) in Fall I.

Fall II

Für `steps == [(Just f, d)]` bekommen wir `max' Nothing ((Just f, d) : [])`.

Mit (3.4.5) sehen wir, dass `Just f > Nothing` gilt, woraus sich `max' s sc` ergibt. Setzen wir ein, bekommen wir `max' (Just f) []` und damit definitionsgemäß `Just f`.

$$\text{accum}_1 = s$$

Wir sehen, dass (3.4.12) auch in Fall II gilt.

Da Fälle I und II halten, gilt (3.4.10) bei nichtleeren Eingaben nach dem ersten Rekursionsschritt.

⁴⁸ Wir könnten in anderer Schreibweise, aber äquivalent sagen, unterscheiden I : $s = \perp$ und II : $s = f$.

3.4.2.2 Induktionsschritt Zeigen wir nun, dass aus der Annahme für Schritt i direkt die Annahme für $i + 1$ folgt.

$$\text{accum}_i = \max\{\text{score}_j\}_{j \in J_i} \xRightarrow{\text{I.H.}} \text{accum}_{i+1} = \max\{\text{score}_j\}_{j \in J_{i+1}}$$

Wir haben $\text{accum}_i = \max\{s_j\}_{j \in J_i}$ und $\{s_j\}_{j \in J_{i+1}} = \{s_j\}_{j \in J_i} \cup \{s_{i+1}\}$. Es ist offensichtlich, dass für sich das Maximum einer Menge durch Hinzufügen eines kleineren Elements nicht ändert und dass ein größeres Element zum neuen Maximum wird.

$$\forall m \leq \max M : \max M = \max M \cup \{m\} \quad (3.4.13)$$

$$\forall m > \max M : m = \max M \cup \{m\} \quad (3.4.14)$$

Es gibt jeweils zwei Möglichkeiten für die Werte von accum_i und s_{i+1} . Entweder handelt es sich um definierte Werte, oder nicht.

Wir unterscheiden demnach vier Fälle:

$$\begin{array}{ll} \text{I :} & \text{accum}_i = \perp \quad \wedge \quad s_{i+1} = \perp \\ \text{II :} & \text{accum}_i = \perp \quad \wedge \quad s_{i+1} = \text{Just } f_{i+1} \\ \text{III :} & \text{accum}_i = \text{Just } f_{\max} \quad \wedge \quad s_{i+1} = \perp \\ \text{IV :} & \text{accum}_i = \text{Just } f_{\max} \quad \wedge \quad s_{i+1} = \text{Just } f_{i+1} \end{array}$$

Fall I

Für $\text{max}' \text{ Nothing } ((\text{Nothing}, d):sc)$ ergibt sich $\text{max}' \text{ Nothing } sc$.

$$\text{accum}_{i+1} = \text{accum}_i$$

Mit (3.4.13) sehen wir, dass $\text{accum}_{i+1} = \max\{s_j\}_{j \in J_{i+1}}$ und damit (3.4.10) für Fall I.

Fall II

Für $\text{max}' \text{ Nothing } ((\text{Just } f, d):sc)$ ergibt sich mit (3.4.5) $\text{max}' (\text{Just } f) sc$.

$$\text{accum}_{i+1} = s_{i+1}$$

Mit (3.4.14) sehen wir, dass $\text{accum}_{i+1} = \max\{s_j\}_{j \in J_{i+1}}$ und damit (3.4.10) für Fall II.

Fall III

Für $\text{max}' (\text{Just } f') ((\text{Nothing}, d):sc)$ ergibt sich mit (3.4.5) $\text{max}' (\text{Just } f') sc$.

$$\text{accum}_{i+1} = \text{accum}_i$$

Mit (3.4.13) sehen wir, dass $\text{accum}_{i+1} = \max\{s_j\}_{j \in J_{i+1}}$ und damit (3.4.10) für Fall III.

Fall IV

Für $\text{max}' (\text{Just } f') ((\text{Just } f, d):sc)$ gibt es die beiden Möglichkeiten $f \leq f'$ und $f > f'$.

Im Fall $f \leq f'$ ergibt sich aus der Definition von \max' mit (3.4.6) $\max' (\text{Just } f')$ sc.

$$\text{accum}_{i+1} = \text{accum}_i$$

Mit (3.4.13) sehen wir, dass $\text{accum}_{i+1} = \max\{s_j\}_{j \in J_{i+1}}$ und damit (3.4.10) für Fall IV mit $f \leq f'$.

Im Fall $f > f'$ ergibt sich aus der Definition von \max' mit (3.4.6) $\max' (\text{Just } f)$ sc.

$$\text{accum}_{i+1} = s_{i+1}$$

Mit (3.4.14) sehen wir, dass $\text{accum}_{i+1} = \max\{s_j\}_{j \in J_{i+1}}$ und damit (3.4.10) für Fall IV mit $f > f'$.

Damit ist (3.4.9) bewiesen, woraus (3.4.8) und (3.4.3) folgen. ■

3.4.3 Füllregeln

Prüfen wir jetzt, ob wir unsere Alignmentmatrix F mit den korrekten Werten befüllen.

F ist eine `NWMatrix` mit Werten vom Typ `Maybe (Int, [StepDirection])`. Definierte Zellen bekommen den Wert `Just (f_ij, directions)` und undefinierte Zellen `Nothing`. Die `[StepDirections]` Komponente interessiert uns nicht.

Die Alignmentmatrix wird mit der `fill` Funktion befüllt, welche in «fill» definiert wurde.

```
fill :: AInInfo -> NWMatrix
fill info = let mat = initMat info
             in mat `seq` fillFrom info (2, 2) mat
             where counts = gapCounts info
```

Diese nutzt zuerst `initMat`, um die Matrix zu initialisieren und dann `fillFrom` um die initialisierte Matrix zu befüllen.

`initMat` und `fillFrom` wiederum nutzen intern `initFillFunc`, bzw. `fillCell`. Daher betrachten wir im Folgenden hauptsächlich `initFillFunc` und `fillCell`.

3.4.3.1 Rekursionsanker Bei der Anpassung des Algorithmus auf feste Alignmentlängen haben wir festgestellt, dass Einträge f_{ij} nur dann definiert sein können, wenn zumindest der diagonale Vorgänger $f_{i-1,j-1}$ definiert ist. Daraufhin haben wir in (3.2.20) für die nullte Zeile und Spalte von F die folgenden Rekursionsanker formuliert.

$$f_{i0} = \begin{cases} i \cdot w_{\text{gap}} & , \text{range}(i, 0) \\ \perp & , \text{Andernfalls} \end{cases} \quad f_{0j} = \begin{cases} j \cdot w_{\text{gap}} & , \text{range}(0, j) \\ \perp & , \text{Andernfalls} \end{cases}$$

In `init-mat` haben wir als Rekursionsanker `initFillFunc` definiert.

```
initFillFunc :: Cost -> (Int, Int) -> MatIdx -> CellValue
initFillFunc (Cost {w_gap = gap}) gaps cell@(i, j)
  | i == 1    = lift f_1j
  | j == 1    = lift f_i1
  | otherwise = Nothing
where
  prv  = d1 cell           :: [StepDirection]
  valid = range gaps cell  :: Bool
  f_i1 = f1 valid gap i    :: CellScore
```

```
f_1j = f1 valid gap j      :: CellScore
lift = (flip toValue) prv :: (CellScore → CellValue)
```

Wir wollen zeigen, dass die f_{ij} die durch unsere Rekursionsanker definiert werden den Wert $(i-1) \cdot \text{gap}$, bzw. $(j-1) \cdot \text{gap}$, annehmen.⁴⁹

`initFillFunc` nimmt als Argumente w_{gap} und bindet dieses an den Namen `gap`, die Gapzahlen (g^1, g^2), welche an `gaps` gebunden werden und den Index (i, j) , bzw. (i, j) , der zu befüllenden Zelle. Sie berechnet $f_{1j} = f1 \text{ valid gap } j$ für $i == 1$ und $f_{i1} = f1 \text{ valid gap } i$ für $j == 1$ (wir können `lift` ignorieren, da es den Wert nicht beeinflusst).

Wenn wir nun die relevanten Definitionen einsetzen, ergibt sich der folgende Ausdruck.

```
initFillFunc (Cost {w_gap = gap}) gaps cell@(i, j)
| i == 1      = lift $ f1 (range gaps cell) gap j
| j == 1      = lift $ f1 (range gaps cell) gap i
| otherwise   = Nothing
-- ... rest omitted ...
```

Wir bilden also bei $i == 1$ auf $f1$ mit j ab und bei $j == 1$ auf $f1$ mit i und liften das Ergebnis anschließend.

Die Definition der $f1$ Funktion sieht so aus:

```
f1 :: Bool → Int → Int → CellScore
f1 valid gap i = if valid
  then Just $ (i-1) * gap
  else Nothing
```

Wir sehen, dass $f1$ für solche Einträge, die `valid` sind, `Just $ (i-1) * gap` zurückgibt. Der Wert `valid` wiederum ist durch das Prädikat `range` festgelegt. In sonstigen Fällen wird `Nothing` produziert. Dies erinnert bereits stark an unseren Anker.

Setzen wir nun auch die Definition für $f1$ ein, ergibt sich:

```
initFillFunc (Cost {w_gap = gap}) gaps cell@(i, j)
| i == 1      = lift $ if (range gaps cell) then (i-1) * gap else Nothing
| j == 1      = lift $ if (range gaps cell) then (j-1) * gap else Nothing
| otherwise   = Nothing
-- ... rest omitted ...
```

Wir sehen, dass das erste Pattern den Fall f_{1j} abdeckt und das zweite den f_{i1} Fall. Für f_{1j} wird, wenn `range(1, j)` gilt, der Wert $(i-1) \cdot w_{\text{gap}}$ produziert und sonst \perp und für f_{i1} m. m. ebenso.

Dies entspricht genau unserer Definition in (3.2.20)

3.4.3.2 Rekursionsbeziehung Wir haben in (3.2.25) die folgende angepasste Befüllungsregel für NW mit festen Alignmentlängen formuliert.

$$f_{ij} = \max\{f_{gh} + w((g, h), (i, j)) \mid (g, h) \in \text{candidates}(i, j)\}$$

Dafür haben wir im «fill-cell» Block die `fillCell` Funktion definiert.

⁴⁹ In unserem Code verschiebt sich die Indizierung der Matrix um 1, also wollen wir zeigen, dass für Einträge in `range` die Werte $f_{i1} = w_{\text{gap}} \cdot (i-1)$ bzw. $f_{1j} = w_{\text{gap}} \cdot (j-1)$ genutzt werden.

```

fillCell :: AlnInfo → MatIdx → NWMatrix → NWMatrix
fillCell info cell mat = best `seq` -- use `seq` to force strict evaluation
    setElem best cell mat
    where
        gaps    = gapCounts info           :: (Int, Int)
        idxs    = candidates gaps cell     :: [MatIdx]
        score    = stepScore info mat      :: (Step → ScoredStep)
        toStep  = (\c → (c, cell))         :: (MatIdx → Step)
        steps   = map (score.toStep) idxs  :: [ScoredStep]
        best    = maxValue steps           :: CellValue

```

Wir bekommen die durch candidates bestimmten Koordinaten der Kandidatenzellen und binden diese an den Namen `idxs`.

Dann definieren wir die Funktionen `toStep` und `score`. `toStep` nimmt einen Zellindex o und bildet ihn auf den Schritt (o, d) ab, wobei d den Index der zu befüllenden Zelle bezeichnet.

Die `score` Funktion entsteht durch die partielle Anwendung von `info` und `mat` auf die zuvor besprochene `stepScore` Funktion, wodurch diese den Typ $(\text{Step} \rightarrow \text{ScoredStep})$ annimmt, also einen Schritt q nimmt und ihn auf dessen Gewicht und Richtung $(w(q), \text{dir})$ abbildet.

Für die Elemente in `idxs` bestimmen wir nun mittels Komposition von `score` und `toStep` die jeweiligen `ScoredStep` Tupel, welche mit `maxValue` gewertet werden.

Wir haben gezeigt, dass `maxValue` die übergebene Liste von `ScoredStep` Werten auf den `CellValue` mit dem größten enthaltenen `CellScore` abbildet, welchen wir an den Namen `best` binden.

Mit `setElem best cell mat` füllen wir also die betrachtete Zelle mit dem besten Kandidatenwert, was genau der Anforderung entspricht.

4 Diskussion

In diesem Kapitel interpretieren wir die Ergebnisse und diskutieren etwaige Probleme mit diesen und die sich daraus ergebenden Limitationen. Zuletzt besprechen wir potentielle Ergänzungen des Modells, sowie Ansätze für interessante Folgefragen.

4.1 Zusammenfassung

Zunächst werden die Arbeitsergebnisse zusammengefasst. An dieser Stelle sei nochmal auf die verwendete Notation verwiesen.

4.1.1 Problemformulierung

Um das Problem zu formulieren, haben wir eingangs in (3.1.1) den Begriff des Alignments formalisiert. Ein Alignment bzw. ein Template ist eine Matrix, der zeilenweise Sequenzen (2.1.1) bzw. deren Symbole zugeordnet werden.

4.1.1.1 Variablen Alignments stellen das gewünschte Ergebnis dar. Wir wissen dabei noch nicht, wie diese belegt werden. Daher haben wir anschließend in (3.1.3) Zuweisungsvariablen formuliert, aus denen sich die Belegung ergibt. Da wir Bestimmung der Güte eines Alignments wissen müssen, an welchen Stellen Lücken sind, haben wir in (3.1.5) aus den Zuweisungsvariablen eine entsprechende Hilfsvariable abgeleitet.

4.1.1.2 Beschränkungen Bestimmte Belegungen von Alignments stellen bilden keine validen Lösungen. Dies haben wir in (3.1.6) (3.1.7) und (3.1.8) durch die Formulierung von drei Beschränkungen für Zuweisungsvariablen dargestellt.

4.1.1.3 Zielfunktion Lineare Optimierungsprobleme lassen sich im Allgemeinen leichter lösen als nicht-lineare. Da wir bei der Kostenbestimmung anhand der zuvor definierten Variablen mehrere Unbekannte miteinander multiplizieren, haben wir in (3.2.2.3.3) und (3.2.2.3.4) weitere Hilfsgrößen abgeleitet, um die Linearität zu bewahren.

In (3.1.13) ist dann auf Basis dieser Hilfsgrößen ein Bewertungskriterium für Alignmentgüte formuliert, welches bei Bestimmung des Optimums zu der Problemformulierung in (3.1.14) führt.

4.1.2 Problemlösung

Um das aufgestellte Problem zu lösen, haben wir daraufhin den naiven Ansatz des Ausprobierens betrachtet und zügig erkannt, dass dieser aufgrund kombinatorischer Explosion nicht zielführend ist. Da die optimale, globale, paarweise Sequenzalinierung ein bereits durch den klassischen Algorithmus von Needleman und Wunsch gelöstes Problem ist, lag anschließend die Überlegung nahe, einen Ansatz auf dessen Basis zu formulieren.

4.1.2.1 Reinterpretation von Needleman-Wunsch Um NW hinsichtlich des MILP-Modells umzuformulieren, wurde in (3.2.3) der Begriff des Pfades als Folge von K -Tupeln über der Indexmenge C_F einer von NW produzierten Alignmentmatrix F eingeführt.⁵⁰ Im Weiteren wurden Regeln bestimmt, denen Pfade genügen müssen.

⁵⁰ Alignmentmatrizen sind in (3.2.1) bzw. (2.1.4) und (2.1.5) beschrieben. Die Indexmengen von Alignmentmatrizen sind in (3.2.2) definiert.

Aus dem Pfadbegriff haben wir im Anschluss in (3.2.8) formal der Begriff des Schrittes, als geordnetes Paar aufeinanderfolgender Pfadelemente abgeleitet und festgestellt, dass Schritte festgelegten, danach beschriebenen Regeln gehorchen.

Auf Basis der in (2.1.5) beschriebenen kanonischen Rekursionsbeziehung für NW haben wir dann in (3.2.15) eine Gewichtsfunktion für Schritte abgeleitet.

NW berechnet den Wert des optimalen Teilalignments von s^1 bis i und s^2 bis j in Zelle f_{ij} aufgrund einer oder mehrerer Vorgängerzellen f_{gh} und produziert so Pfade durch F , die die optimalen Alignments von s^1 und s^2 darstellen. Der Pfadbegriff in NW entspricht daher dem Begriff des Alignments in MILP. Aus dieser Tatsache wurde in (3.2.16) eine Zielfunktion in geschlossener Form formuliert, die den Wert des von NW produzierten Alignments beschreibt.

4.1.2.2 Anpassung für feste Alignmentlängen Die geschlossene Formulierung der Zielfunktion in (3.2.16) nutzt die Variable K , die der Länge des berechneten Pfades entspricht. Daraus ergibt sich die Notwendigkeit, K zu festzusetzen, weil (3.2.16) sonst undefiniert wäre. Ein weiterer verwandter Grund, K festzusetzen ist, dass das MILP-Modell mit festen Alignmentlängen arbeitet und diese von NW emuliert werden müssen, um das aufgestellte Modell zu lösen.

Bei der Untersuchung des Problems wurde zunächst mit (3.2.17) die Anzahl der in eine bestimmte Sequenz einbaubaren Gaps beschrieben. Auf dieser Basis konnte mit (3.2.19) ein *notwendiges* Kriterium⁵¹ zur Einhaltung der Alignmentlänge identifiziert, durch das die ursprünglich aus (2.1.5) stammende und in (3.2.23) explizit gemachte Formulierung der Kandidatenwahl zu (3.2.24) angepasst wurde.

4.1.2.3 Herleitung der Variablen Mithilfe der formulierten Strukturen wurden mit (3.2.27) die Zuweisungsvariablen und mit (3.2.28) die Gapvariablen hergeleitet. In (3.2.29) und (3.2.32) wurden die Hilfsvariablen ϕ_{ijk} und γ_k in den NW-Kontext übertragen und dabei gezeigt, dass diese konsistent mit den ursprünglichen Definitionen in (3.2.2.3.3) und (3.2.2.3.4) sind.

4.1.2.4 Herleitung der Beschränkungen Für die hergeleiteten Zuweisungsvariablen wurde mittels der für Pfade und Schritte identifizierten Regeln bewiesen, dass die im MILP-Modell aufgestellten Beschränkungen auch für die aus NW hergeleiteten Variablen gelten.

4.1.2.5 Herleitung der Zielfunktion In das in (3.2.15) formulierte Schrittgewicht wurden die hergeleiteten Entscheidungsvariablen eingesetzt, wodurch sich (3.2.35) ergab. Durch Einsetzen der neuen Form in (3.2.16) entsteht (3.2.36) das sich äquivalent zu (3.2.37) übersetzen lässt.

Dies entspricht der ursprünglichen und in (3.1.14) beschriebenen MILP-Problemformulierung.

4.1.3 Implementation

Im Implementationsteil wurden die entwickelten Lösungsansätze im Stil der literarischen Programmierung in Programme umgewandelt. Dabei wurde zuerst der naive Ansatz implementiert, um die Leserschaft langsam an die genutzte Programmiersprache Haskell heranzuführen.

Die Implementierungen stellen zuerst die Datentypen vor, die notwendig sind, um das Problem zu beschreiben und entwickeln dann auf Basis der Erkenntnisse aus den vorigen Kapiteln Funktionen zur Problemlösung.

⁵¹ † Das gefundene Kriterium ist notwendig, nicht aber hinreichend um die Einhaltung zu garantieren. Mehr dazu findet sich in der entsprechenden Sektion zur Diskussion der Limitationen.

Das Vorgehen folgte einer narrativen Struktur. Bei der Implementation von NW wurden zuerst die Funktionen zur Initialisierung einer Alignmentmatrix F geschrieben und dann die zum Befüllen notwendige Rekursionsbeziehung implementiert. Anschließend wurden die Logik zum Generieren der Alignments durch Backtracking hinzugefügt und zuletzt alle Teile zu einer einzigen `align` Funktion komponiert.

Außerdem wurden Repräsentationsfunktionen, IO-Logik und Projekt-relevante Quellen zur Paketierung geschrieben, obgleich diese für die eigentliche Problemlösung keine Bedeutung besitzen.

4.1.4 Verifikation

Im Rahmen der Verifikation haben wir die zentralen Teile der Implementation betrachtet, um zu prüfen, ob das Programm die zuvor im Problemlösungsteil beschriebene Spezifikation erfüllt.

4.1.4.1 Grundlegende Funktionen Zuerst wurde die Korrektheit grundlegender Komponenten gezeigt, so dass spätere Beweise für komplexere Funktionen sich direkt darauf stützen können.

Dabei wurde nachgewiesen, dass die Kostenfunktionen `substWeight` und `stepWeight` den Definitionen in (2.1.2) und (3.2.15) entsprechen und dass die `gapCounts` und `range` Funktionen die zur Kandidatenwahl notwendige Bestimmung der Gapzahlen (3.2.17) und das Abstandskriterium (3.2.19) korrekt implementieren.

4.1.4.2 Kandidatenwahl Auf dieser Grundlage konnte zugleich die in (3.2.24) beschriebene und in `candidates` implementierte Auswahlfunktion für Schrittkandidaten verifiziert werden.

4.1.4.3 Maximierung des Zellwerts Durch Induktion über die Kandidatenliste, die der Funktion `maxValue` übergeben wird, konnte gezeigt werden, dass bei Terminierung der höchste Zellwert produziert wird und `maxValue` daher den durch Wahl eines Kandidatenschrittes entstehenden Zellwert maximiert.

4.1.4.4 Füllregeln Zuletzt wurden die in (3.2.20) definierten Rekursionsanker und die in (3.2.25) definierte Rekursionsbeziehung betrachtet. Es ließ sich nachweisen, dass `initFillFunc` bzw. deren Komponenten dem Rekursionsanker entsprechen, und dass `fillCell` die Rekursionsbeziehung darstellt.

4.2 Interpretation

Bei eingehender Betrachtung ergaben sich klare semantische Analogien zwischen dem beschriebenen MILP-Modell und dem NW-Algorithmus. Man kann zudem zeigen, dass sich durch bestimmte Anpassungen von NW die im MILP formulierte Problemstruktur bewahren lässt.

Da das MILP-Modell und NW denselben Zweck verfolgen, liegt die Vermutung, dass die Formulierungen sich isomorph zueinander verhalten, zwar nahe, aber es ist trotzdem überraschend, wie schön bestimmte Zusammenhänge ins Schloss fallen.

Bei der Implementation hat sich im Zusammenhang mit den im Folgenden besprochenen Problemen gezeigt, dass durch Korrektheitsbeweise nicht automatisch qualitativ hochwertige Software entsteht. Beweise können im besten Fall die Korrektheit funktionaler Anforderungen⁵² zeigen, sind aber nicht geeignet, um nicht-funktionale Anforderungen wie allgemeine Nutzbarkeit nachzuweisen.

Beweise können trotzdem sehr nützlich sein und es kann Sinn ergeben einen Korrektheitsbeweis zu führen um die Funktion einer Komponente zu verifizieren. Dies gilt insbesondere für kritische Systemteile,

⁵² Solche Anforderungen die Systemverhalten spezifizieren.

mit überschaubarer Spezifikation. Beweise ersetzen allerdings keine Tests und sind, wie auch das Testen, mit einem Aufwand verbunden, den nicht jeder zu zahlen bereit ist.

Neue Resultate?

Der Zusammenhang Isomorphie von NW und MILP

4.3 Limitationen

Es gibt einige Probleme mit dem im Rahmen der Arbeit aufgestellten Modell.

4.3.1 Komplexität biologischer Fragestellungen

Die biologische Realität ist komplizierter als dargestellt.

Einerseits existieren weitere Arten von Mutationen als Substitutionen und Indels und andererseits muss die gewählte Interpretation⁵³ nicht unbedingt die biologische Realität widerspiegeln.

Bspw. müssen Mismatches im Alignment nicht unbedingt aus der einfachen Substitution einzelner Basen herrühren, sondern z.B. aus der gleichzeitigen Substitution mehrerer Basen, oder wiederholter Substitution einer einzelnen Base. Genauso könnte die Existenz von Gaps im Alignment auf strukturellen Veränderungen beruhen, bei denen Teile einer Sequenz miteinander vertauscht wurden.

Sinn von Modellen ist es, eine vereinfachte, zweckorientierte Nachbildung des zu betrachtenden Systems und dessen Prozessen zu geben. Diese Ungenauigkeit liegt bei der Modellierung komplexer Sachverhalte also in der Natur der Sache.

4.3.2 Spezifikation des Lösungsraums

Üblicherweise werden Optimierungsprobleme in der Form $\max\{c^\top x : Ax \leq b, x \leq 0\}$ dargestellt, wobei der Vektor $x \in \mathbb{R}^n$ ein Punkt in einem n -dimensionalen Polytop ist, dessen Facetten durch die mit $Ax \leq b$ formulierten Beschränkungen gegeben sind.

Es wäre schöner, wenn wir die Beschränkungen umformulierten, um ein ideales⁵⁴ Polytop P zu finden, mit dem wir den Lösungsraum X darstellen können.

4.3.3 Garantie fester Alignmentlängen

Die Formulierung von NW für feste Alignmentlängen ist nicht ausreichend.

Unser Algorithmus wird in manchen Fällen mehr als g_{\max} Gaps einbauen. Der Grund dafür ist, dass das range Prädikat, welches den Abstand eines Kandidaten zur Hauptdiagonale prüft ein notwendiges, aber kein hinreichendes Kriterium ist.

```
ghci> let info = mkInfo 1 (-1) (-2) 0 "AGTAC" "ATGC"
ghci> putStrLn $ (showAln.head.optAlns.align) info
AG-T-A-C-
-----
--A-T-G-C
```

Dies passiert regelmäßig in Fällen, in denen w_{gap} kleiner ist als w_{match} und w_{miss} .⁵⁵

⁵³ Mismatches sind Substitutionen und Gaps sind Indels.

⁵⁴ Vgl. [19] Definition 1.1, pp. 12

⁵⁵ Man könnte diese Fälle als pathologisch bezeichnen, da sie nicht nur unintuitive Ergebnisse erzeugen, sondern auch biologisch keinen Sinn ergeben.

4.3.3.1 Kandidatenwahl mit erweitertem Abbruchkriterium Dieses Problem zu beheben ist nicht trivial, aber möglich.

Wir können beim Berechnen der Zellwerte bereits Eintragen, wie viele Gaps auf dem Weg zu dieser Zelle eingebaut wurden.

Als Rekursionsanker dienen $g_{i0} = i$ und $g_{0j} = j$.

Wenn für Zelle f_{ij} der potentielle Vorgänger f_{gh} den besten Wert liefert, dann entspricht die Anzahl der Gaps g_{ij} , welche wir auf dem Weg nach f_{ij} einbauen, der Anzahl g_{gh} von Gaps bis zum Vorgänger f_{gh} , im Falle von diagonalen Schritten und $g_{gh} + 1$ bei horizontalen oder vertikalen Schritten.

$$g_{ij} = \begin{cases} g_{gh} & , i = g + 1 \wedge j = h + 1 \\ g_{gh} + 1 & , \text{Andernfalls} \end{cases} \quad (4.3.1)$$

Da wir g_{ij} im gleichen Schritt wie f_{ij} berechnen können und dies pro Zelle zusätzlich nur konstante Rechenzeit und Speicher erfordert, sollte diese Korrektur weder Zeit- noch Speicherkomplexität beeinflussen.

Außerdem kann g_{ij} als zusätzlicher "Tie-Breaker" dienen, wenn die Werte f_{gh} verschiedener Kandidaten-schritte gleich sind.

Mit (4.3.1) können wir unsere Funktion zur Kandidatenwahl anpassen und prüfen, ob die Anzahl an Gaps, die durch einen bestimmten Kandidaten entstehen zu hoch wird. Damit ergibt sich die folgende korrigierte Funktion:

$$\text{candidates}(i, j) = \{c = (g, h) \in C_{ij} \mid \text{range}(c) \wedge g_{gh} \leq K\} \quad (4.3.2)$$

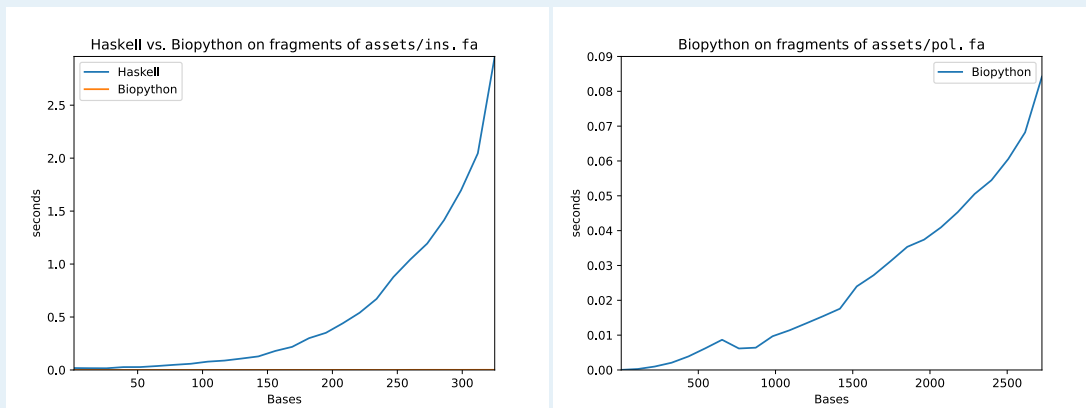
4.3.4 Performance

Beim Profilen des Programms stellte sich heraus, dass es nur ungenügende Performancecharakteristika hat. So wie sie ist, kann die Implementation nicht sinnvoll in der Praxis genutzt werden.

Der Code mit dem das Profiling erfolgte und zur Generierung der folgenden Graphiken befindet sich in dem Skript `assets/scripts/profile_aligner.py`. Weiterhin wurde das [eventlog2html](#) Paket verwendet um aus den `assets/profiling` hinterlegten Heap-Dumps die Informationen zur Speicherkomplexitätsanalyse zu extrahieren.

4.3.4.1 Zeitkomplexität Es hat sich gezeigt, dass die Berechnung des Alignments für die Sequenzen in `pol.fa` nicht in angemessener Zeit gelöst werden konnte.

Information



(a) Ausführungszeiten von Haskell und Biopython auf Teilstücken von assets/ins. fa

(b) Ausführungszeiten von Biopython auf Teilstücken von assets/pol. fa

Abbildung 5: Ausführungszeiten von der Haskell-Implementation und Biopythons Pairwise-Aligner im Vergleich.

Die Ausführungszeiten beider Algorithmen steigen klar exponentiell mit der Länge der verarbeiteten Sequenzen, allerdings ist klar zu sehen, dass das Steigungsverhalten der im Rahmen dieser Arbeit entwickelten NW-Implementation deutlich schlechter ist. Zur Berechnung des Alignments der pol. fa Sequenzen, an denen die Haskell-Implementation scheiterte, brauchte Biopython mit identischen Parametern ca. 0,08 Sekunden.

Die Zeitkomplexität von NW für zwei Sequenzen mit Längen m und n liegt üblicherweise in der Klasse $\mathcal{O}(m \cdot n)$. Diese Einordnung setzt allerdings voraus, dass Einfügeoperationen in Matrizen in konstanter Zeit stattfinden. Das `matrix` Paket generiert beim Setzen eines Elements allerdings eine neue Matrix, anstatt die bestehende zu mutieren, weswegen Aufrufe von `setElem` nicht in der Klasse $\mathcal{O}(1)$, sondern in $\mathcal{O}(m \cdot n)$ liegen. Da der Algorithmus $m \cdot n$ Zellen in F befüllen muss, steigt die zu Ausführungszeit mit $\mathcal{O}(m^2 \cdot n^2)$.

Des Werte nicht verändert, sondern kopiert, werden ergibt Sinn, da Haskell eine pure funktionale Sprache ist, in der alle Werte immutabel sind. Dies kann aber umgangen werden, indem statt einer normalen `Matrix` eine mutable `MMatrix` verwendet wird, deren Zellen innerhalb einer Monade in konstanter Zeit mutiert werden können.

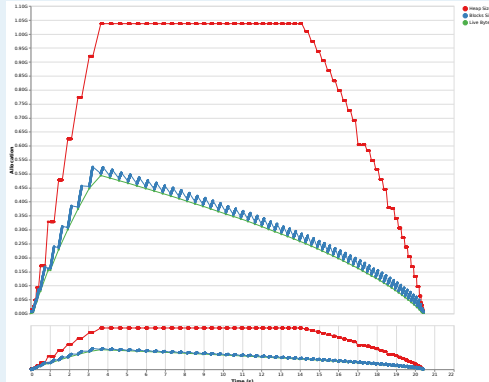
Da Monaden einfach nur Monoide in der Kategorie der Endofunktoren sind, ist dies konzeptionell natürlich trivial, aber leider reicht, zum Zeitpunkt an dem ich diese Worte schreibe, die Bearbeitungszeit dafür nicht mehr aus.

Ein weiteres gravierendes Problem liegt im ineffizienten Backtracking-Algorithmus, der die Ausführungszeiten noch weiter aufbläht. Weitere sich aus dem Backtracking ergebende Probleme betrachten wir in der nächsten Sektion.

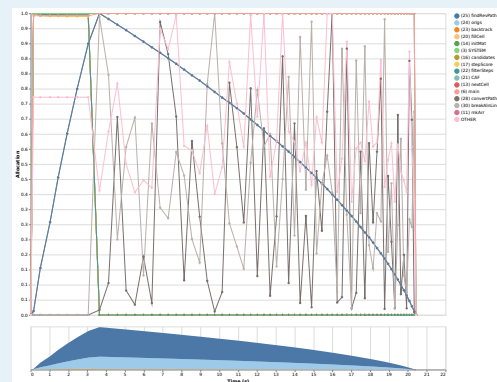
4.3.4.2 Speicherkomplexität Auch die Speichernutzung des Programms disqualifiziert es für den praktischen Gebrauch.

Information

Bei der Betrachtung der Speichernutzung über den Zeitverlauf fällt auf, dass die Nutzung zuerst explosionsartig zunimmt und dann nach und nach abgebaut wird.



(a) Die Größe des Heaps gibt uns eine Übersicht über die Gesamtauslastung des Speichers während der Ausführung.



(b) Der größte Verbrauch entsteht beim Backtracking in der findRevPaths Funktion.

Abbildung 6: Die Speicherauslastung beim Alinieren der ersten 125 Symbole von pol. fa.

Erinnern wir uns an die Definition von `findRevPaths :: NWMatrix -> MatIdx -> [Path]`.

```
-- ... excerpt abridged for clarity ...
findRevPaths mat cell = (prepend.collect.continue) cell0rigs
  where
    cell0rigs = origs cell elem
    continue  = map (findRevPaths mat)
    collect   = concat
    prepend   = map (cell:)
```

Wir erweitern zunächst den Pfad von den bisher gefundenen Vorgängern aus, fassen die so entstandenen Subpfade im nächsten Schritt zusammen und fügen die betrachtete Zelle am Schluss vorne an alle gesammelten Subpfade an.

D.h. es werden alle möglichen Pfade berechnet, bevor eine Rückgabe stattfindet.

Potentiell existieren bei jedem Schritt durch die Matrix drei Möglichkeiten den Pfad weiterzuführen. Mit insgesamt K Schritten,⁵⁶ ergeben sich schlimmsten Fall $\prod_{k=1}^K 3$ mögliche optimale Alignments produzieren.

Diese Anzahl von möglichen Alignments steigt also mit $K!$.

⁵⁶ Die wirkliche Komplexität hängt von der festen Dimension der Matrix und nicht der variablen Schrittzahl ab, aber die Darstellung mit Schritten ist deutlich anschaulicher.





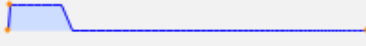
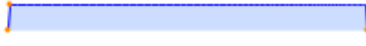
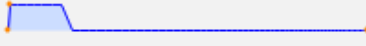

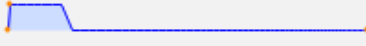
Profile	n	Label	Size (MiB s)	Stddev (MiB)
	31	(25) findRevPaths	6413.54	61.44
	30	(24) origs	2720.97	26.06
	29	(23) backtrack	113.43	1.08
	28	(20) fillCell	7.27	0.32
	27	(14) initMat	3.73	0.16
	26	(3) SYSTEM	2.07	0
	25	(16) candidates	1.97	0.08
	24	(17) stepScore	1.92	0.08
	23	(22) filterSteps	1.81	0.08

Abbildung 7: Übersicht über die Speicherauslastung pro Funktion.

4.3.5 Grenzen der Beweisbarkeit

Das gewählte Modell der Verifikation durch händische Beweise hat verschiedene Beschränkungen.

4.3.5.1 Implizite Annahmen Strenggenommen ist der Beweis der Korrektheit eines Programmes nicht so einfach, wie wir es beschrieben haben. Zusätzlich zum Beweis, dass der Code korrekt ist, müssten wir u.a. zeigen,

1. dass die Semantik der Sprache korrekt implementiert ist, dass also der Compiler/Interpreter die Sprachkonstrukte in die Maschinenanweisungen übersetzt, die wir erwarten, und
2. dass der Prozessor diese Anweisungen auch richtig verarbeitet.⁵⁷

In der Praxis gehen wir jedoch i.d.R. davon aus, dass die Betrachtung des Programmcodes ausreicht und die restlichen Teile des Systems ordnungsgemäß funktionieren.

4.3.5.2 Refaktorisierung von Programm und Beweis Mit der Größe und Komplexität eines Programmes steigt auch die Schwierigkeit des Beweises. Deswegen ist es sinnvoll Programm und Korrektheitsbeweis synchron zu schreiben.

Dies erfordert allerdings, dass Änderungen am Programm eine Anpassung der Beweisführung nach sich ziehen. Dadurch wird das Refaktorisieren des Programms deutlich erschwert.

⁵⁷ Diese Annahme galt z.B. nicht für die vom FDIV-Bug betroffenen Intel-Pentium-Prozessoren, was wahrscheinlich einer der Gründe ist, warum Intel in nachfolgenden Modellen begann verstärkt Methoden der formalen Verifikation zu nutzen. Vgl. [21].

4.3.5.3 Korrektheit von Beweisen

Genauso wie Programme können auch Beweise Fehler haben.

Bei langen händischen Berechnungen ist leicht Fehler zu machen und schwer zu prüfen, ob diese korrekt sind. Bei händischen Beweisen ist es genauso.

Haskell Curry⁵⁸ und William Howard haben entdeckt, dass Programme und Beweise isomorph sind. [22] Dieses Ergebnis wird als Curry-Howard-Korrespondenz bezeichnet. Durch diese Korrespondenz wissen wir, dass Beweise programmatisch darstellbar sind, was die Grundlage für Beweisassistenten wie [Coq](#) darstellt.

Auch ein programmatischer Beweis kann falsch sein, wenn das Modell nicht korrekt abgebildet wurde oder falsche Annahmen gemacht wurden. Es ist allerdings leichter zu prüfen, ob eine Summenfunktion korrekt implementiert ist, als manuell die Korrektheit der Summe von 1.000 Zahlen zu bestimmen. Ebenso ist es ggf. leichter sich von der Korrektheit eines programmatischen, als von der eines manuellen Beweises zu überzeugen.

4.4 Ergänzungen

Es gibt verschiedene Möglichkeiten das aufgestellte Modell sinnvoll zu erweitern.

4.4.1 Affine Gapkosten

Um affine Gapkosten mit unterschiedlichen Werten für Öffnen und Erweitern von Gaps zu modellieren, wie in [17], bräuchten wir extra Variablen an denen wir Start und Ende von Gaps ablesen können. Da an jeder Stelle des Templates potentiell ein Gap auftreten kann, wäre eine Variable für jedes Feld im Template notwendig.

Wir können sehen, dass auch diese Variablen als "Lückenmatrix" $\mathcal{G} \in \mathbb{B}^{|S| \times K}$, mit denselben Dimensionen wie T , darstellbar sind.

$$\mathcal{G} = (g_k^m) \quad g_k^m = \begin{cases} 1, & t_k^m := c_{\text{gap}} \\ 0, & \text{Andernfalls} \end{cases}$$

Auch hier stellen wir den Zeilenindex hoch, um den Zusammenhang mit s^m zu kennzeichnen.

Die Gapmatrix \mathcal{G} bezieht sich also auf das gesamte Alignment T .

Auch bei \mathcal{G} gibt es eine sinnvolle Interpretation der Indizes. Die Positionen entsprechen einfach denen des Templates, d.h. der Zeilenindex bestimmt auf welche Sequenz Bezug genommen wird und der Spaltenindex spiegelt die, potentiell mit Gaps aufgefüllte, Position im Alignment wider.

Beispiel 4.1

Die Befüllung des zuvor gegebenen Beispieltemplates $\begin{smallmatrix} A & G & T & A & C \\ A & - & T & G & C \end{smallmatrix}$ mit Gapsymbolen wird durch die folgende Gapmatrix dargestellt:

$$\mathcal{G} = \begin{pmatrix} 0 & 0 & 0 & 0 & 0 & 1 \\ 0 & 1 & 0 & 0 & 0 & 1 \end{pmatrix}$$

⁵⁸ Haskell Curry ist der Namenspatron der Programmiersprache Haskell.

Wenn wir zum Modellieren affiner Gapkosten von L Sequenzen eine Gapmatrix $(g_k^m) = \mathcal{G} \in \mathbb{B}^{L \times K}$ definiert haben, besteht ein klarer Zusammenhang zwischen Sequenzsymbolen und Gaps.

Wir wissen, dass an einer Stelle, an der ein Sequenzsymbol zugewiesen wurde, kein Gap sein kann, und dass umgekehrt an einer Stelle, an der kein Sequenzsymbol zugewiesen wurde, ein Gap sein muss.

$$\forall k \in J_K : g_k^m = 1 - \sum_{i=1}^{|s^m|} a_{ik}^m$$

Gaps g_k^m sind also genau an den Stellen t_k^m , wo keine Sequenzsymbole zugewiesen wurden.

4.4.2 Erweiterung auf MSA

Mit den für das MILP definierten Variablen können wir auch direkt eine Zielfunktion für das multiple Sequenzalignment aufstellen.

Bei MSA können wir Sequenzen in einer Menge S paarweise miteinander vergleichen und aufsummieren, um die Gesamtkosten zu bestimmen.

Wir betrachten zwei verschiedene Sequenzen s^m und s^n aus $S = \{s^1, \dots, s^L\}$ mit $m, n \in J_L, m \neq n$.

Die Definitionen für a_{ik}^m bzw. a_{jk}^n und g_k^m bzw. g_k^n bleiben unverändert und entsprechen (3.1.3) bzw. (3.1.5)

Wir schreiben w_{ij}^{mn} für das Gewicht einer Substitution zwischen s_i^m und s_j^n . Dieses kann entweder wie (2.1.2) flache Kosten benutzen, oder analog zu (2.1.3) anhand einer Substitutionsmatrix W als $w_{ij}^{mn} = W_{s_i^m, s_j^n}$ bestimmt werden.

$$w_{ij}^{mn} = \begin{cases} w_{\text{match}} & , s_i^m = s_j^n \\ w_{\text{miss}} & , \text{Andernfalls} \end{cases} \quad (4.4.1)$$

Wir definieren außerdem zwei Familien von Entscheidungsfunktionen $\phi_{ij}^{mn}(k) : \mathbb{N} \rightarrow \mathbb{B}$ und $\gamma^{mn}(k) : \mathbb{N} \rightarrow \mathbb{B}$.

$$\phi_{ij}^{mn}(k) = a_{ik}^m \cdot a_{jk}^n \quad \gamma^{mn}(k) = |g_k^m - g_k^n| \quad (4.4.2)$$

Die Funktion ϕ_{ij}^{mn} entspricht (3.2.2.3.3) aus der PSA-Formulierung und sagt aus, ob die Sequenzsymbole s_i^m und s_j^n in dieselbe Templatespalte t_k eingetragen wurden. Die Funktion γ^{mn} entspricht (3.2.2.3.4) aus der PSA-Formulierung sagt aus, ob es an der Stelle t_k einen Gap zwischen s^m und s^n gibt.

Wir schreiben verkürzt ϕ_{ijk}^{mn} , bzw. γ_k^{mn} .

Die Zielfunktion ist analog zu (3.1.13) gegeben durch:

$$\sum_{m=1}^L \sum_{n=1}^L \left[[m \neq n] \cdot \sum_{k=1}^K \left[w_{\text{gap}} \cdot \gamma_k^{mn} + \sum_{i=1}^{|s^m|} \sum_{j=1}^{|s^n|} [w_{ij}^{mn} \cdot \phi_{ijk}^{mn}] \right] \right] \quad (4.4.3)$$

Hier wird beachtet, dass nur die Kosten von unterschiedlichen Sequenzen betrachtet werden.

4.4.2.1 Needleman-Wunsch für multiple Sequenzalinierung Im Allgemeinen lässt sich NW für eine Menge $S = \{s^1, \dots, s^L\}$ von L Sequenzen formulieren.

Wir nutzen im Folgenden Multiindexnotation mit $\alpha = (\alpha^1, \dots, \alpha^L)$ und komponentenweiser Addition, Subtraktion und Multiplikation.

4.4.2.1.1 F als Tensor Sei $(f_\iota) = F$ ein Tensor mit Dimension $\times_{i=1}^L |s^i| + 1$. Die einzelnen Richtungen des Tensors korrespondieren mit den verschiedenen Sequenzen und die jeweiligen Indexkomponenten mit den Sequenzpositionen.

Analog zur ursprünglichen Interpretation, weisen Änderungen am Index in einer bestimmten Dimension auf den Einbau des entsprechenden Sequenzsymbols hin und ausbleibende Änderungen auf Gaps.

4.4.2.1.2 Rekursionsanker In Anlehnung an (2.1.4) dienen die an den Ursprung anliegenden Seiten des durch F dargestellten Hyperwürfels als Rekursionsanker.

Sei ι_0 ein Prädikat, welches aussagt, dass der Index ι eine Nullkomponente beinhaltet.

$$\iota_0 = \left[0 \neq \sum_{i=1}^L [\iota^i = 0] \right]$$

Der Rekursionsanker lässt sich dann folgendermaßen formulieren:

$$f_\iota = \iota_0 \cdot \sum_{i=1}^L \iota^i \cdot w_{\text{gap}}$$

4.4.2.1.3 Rekursionsbeziehung Um die nächste Zelle zu befüllen muss das Gewicht für einen Schritt von Vorgänger f_η zu Nachfolger f_ι bestimmt werden.

Schritte

Für die Vorgänger η von ι gilt, dass sich alle Komponenten höchstens um eins unterscheiden und dass es mindestens eine Komponente gibt die sich unterscheidet.

$$\forall i \in J_L : \eta_i \in \{\iota^i - 1, \iota^i\}$$

Daher muss $\iota - \eta \in \mathbb{B}^L$ und $\iota - \eta \neq \mathbf{0}$. Sei prev_ι ein Prädikat, welches aussagt, ob ein Index Vorgänger von ι sein kann.

$$\text{prev}_\iota(\eta) = \iota - \eta \in \mathbb{B}^L \wedge \iota - \eta \neq \mathbf{0}$$

Dann kann die Menge der Vorgänger von ι folgendermaßen dargestellt werden:

$$\text{precursors}_\iota = \{\eta \mid \eta \in \mathbb{N}_0^L \wedge \text{prev}_\iota(\eta)\}$$

Gapkosten

Um die Gapkosten eines Schrittes zu bestimmen, vergleichen wir die Komponenten von Vorgänger η und Ziel ι miteinander. Die Gapkosten entsprechen der Summe über die unveränderten Komponenten.

$$\sum_{i=1}^L [w_{\text{gap}} \cdot [\iota^i = \eta^i]]$$

Substitutionskosten

Sei w_{ij}^{mn} wie in (4.4.1)

Um die gesamten Substitutionskosten zu bestimmen, wird die Summe der Substitutionskosten für alle veränderten Indizes gebildet.

Wir bauen $s_{\iota^m}^m$ ein, wenn $\iota^m \neq \eta^m$.

$$\sum_{m=1}^L \sum_{n=1}^L [w_{\iota^m \iota^n}^{mn} \cdot [m \neq n \wedge \iota^m \neq \eta^m \wedge \iota^n \neq \eta^n]]$$

Gesamtkosten

Um die Kosten für einen Schritt durch die Matrix zu erhalten, müssen wir zu allen Vorgängern f_η von f_ι die Kosten von Substitutionen und Gaps addieren.

$$w(\iota, \eta) = \sum_{i=1}^L [w_{\text{gap}} \cdot [\iota^i = \eta^i]] + \sum_{m=1}^L \sum_{n=1}^L [w_{\iota^m \iota^n}^{mn} \cdot [m \neq n \wedge \iota^m \neq \eta^m \wedge \iota^n \neq \eta^n]]$$

Befüllungsregel

Damit bekommen wir die folgende Befüllungsregel:

$$f_\iota = \max \{f_\eta + w(\iota, \eta) \mid \eta \in \text{precursors}_\iota\}$$

5 Fazit

Die Arbeit stellt ein MILP-Modell für das paarweise optimale globale Sequenzalinierungsproblem vor und zeigt, dass ein eindeutiger Zusammenhang zwischen diesem Modell und dem klassischen Algorithmus zur globalen Sequenzalinierung von Needleman und Wunsch existiert. Auf Basis von NW ist eine Lösung für dieses Modell implementiert. Zu guter Letzt wird für deren zentrale Komponenten die Konformität mit der aus dem Ansatz resultierenden Spezifikation aufgezeigt.

5.1 Erkenntnisse

Die dargestellten Ergebnisse erlauben die Beantwortung der zu Beginn der Arbeit formulierten Forschungsfragen.

5.1.1 Sequenzalinierung als lineares, gemischt-ganzzahliges Optimierungsproblem

In Rahmen dieser Arbeit hat sich gezeigt, dass das existierende Modell von McAllister et al. in [17] eine solide Grundlage bildet, auf der eine Darstellung von Sequenzalinierung als MILP aufbauen kann. Zudem stellte sich heraus, dass eine, an die Arbeit von Althaus et al. angelehnte, Anpassung der Notation eine verständlichere Darstellung des Problems erlaubt. [18]

Mithilfe dieser angepassten Darstellung war es einfach die Formulierung der Entscheidungsvariablen bündig zu Matrizen zusammenzufassen, wodurch sich die Zusammenhänge im Modell besser herausarbeiten lassen.

5.1.2 Lösung des MILP-Problems

Aufgrund des ermittelten Zusammenhangs zwischen dem MILP-Modell und NW lässt sich vermuten, dass eine angepasste NW-Formulierung geeignet ist, um das vorgestellte Modell zu lösen. Infolge der erörterten methodischen Schwächen bei der Überführung des Modells kann diese Frage allerdings mit den Mitteln dieser Arbeit nicht mit hinreichender Sicherheit beantwortet werden.

5.1.3 Umsetzung in ein Computerprogramm

Die Programmiersprache Haskell wurde gewählt, da ihre syntaktische Nähe zu mathematischer Notation und die aus ihrer funktionalen Natur resultierenden Eigenschaften sie einer formalen Verifikation leicht zugänglich machen. Diese Wahl hat sich allerdings insoweit als problematisch erwiesen, als dieselben Eigenschaften das Schreiben von effizientem Code erschweren.

Die Freiheit von Nebeneffekten und der damit notwendig werdende Umgang mit Programmzuständen mithilfe von Monaden stellt für Einsteiger in die funktionale Programmierung eine hohe Hürde dar. Außerdem schränkt die nicht-strikte Auswertung die Nachvollziehbarkeit des Speichermanagements ein. Bei rekursiven Aufrufen ist für Neulinge häufig nicht direkt ersichtlich, in welchen Fällen der Garbage-Collector den Heap leert, um Speicher freizumachen und wann Werte unnötig im Speicher verweilen, weil die Auswertung erst stattfindet, sobald der Rekursionsanker erreicht wurde.

Die Performanceprobleme der erdachten Implementation offenbaren daher die Notwendigkeit kompetenter und mit dem funktionalen Paradigma vertrauter Programmierer. Die geringe Verbreitung solcher Sprachen macht es aber aufwändig, solche zu finden.

5.1.4 Formale Verifikation der Implementation

Aufgrund der Wahl der Programmiersprache reichten grundlegende Techniken, um die Korrektheit zentraler Komponenten des entwickelten Programms nachzuweisen. Anderweitige Methoden als der simple Abgleich von Definitionen und strukturelle Induktion waren nicht notwendig.

Die leichte Verifizierbarkeit funktionaler Sprachen macht diese trotz der beschriebenen Nachteile zu einer realistischen Alternative gegenüber klassisch-imperativen Sprachen, gerade bei kritischen Systemen mit klarer Spezifikation.

5.1.5 Relevanz von Korrektheit im Software-Engineering

Es liegt auf der Hand, dass Korrektheit ein notwendiges Kriterium ist, um gute Software zu produzieren. Die trotz Verifikation identifizierten Schwächen in der Implementation machen allerdings deutlich, dass Korrektheit alleine nicht ausreicht, um nutzbare und qualitativ hochwertige Software zu produzieren.

Aufgrund der fundamentalen Probleme des reinen Testens von Programmen stellen Beweise trotzdem ein nützliches Werkzeug für Softwareingenieure dar. Gerade bei kritischen Systemen bietet es sich an, bestimmte Kernkomponenten nicht nur zu testen, sondern diese zusätzlich einer formalen Verifikation zu unterziehen.

5.1.6 Vorteile einer nicht-klassische Formulierung

Da der gewählte Lösungsansatz nicht auf dem gebildeten Modell selbst, sondern auf der Überführung in NW basiert, trat die erhoffte Vereinfachung des Verifikationsprozesses nicht ein. Aus der Tatsache, dass dieser Vorteil nicht zustande kam, folgt allerdings nicht, dass die Nutzung alternativer Modelle zur Simplifikation der mathematischen Analyse im Allgemeinen unnütz ist.

Ein klarer Vorteil, der sich aus der Formulierung des MILP ergab, ist der dadurch entstehende Perspektivgewinn. Beide Modelle bieten verschiedene Blickwinkel, die einander ergänzen und helfen, das Alinierungsproblem besser zu verstehen.

MILP bildet eine solide mathematische Grundlage mit definierten Eingabewerten, klar formulierten Beschränkungen für diese Eingaben und einer einfach handhabbaren, geschlossenen Formel, um die Qualität eines Alignments zu bestimmen. Dabei ist es zuweilen schwer, gut verständliche Interpretationen für die im Rahmen von MILP formulierten Sachverhalte zu finden.

Die rekursive Natur von NW und die daraus resultierende Möglichkeit der kompakten und eleganten Formulierung machen es dagegen gut geeignet für intuitive Erklärungen. Zusammenhänge lassen sich einfach anhand nachvollziehbarer Beispiele demonstrieren. Andererseits sind die mathematischen Hintergründe bei NW nicht direkt ersichtlich und einem Laien kann es zunächst schwerfallen zu verstehen, wie NW das Alignmentproblem auf Teilschritte zurückführt.

Aufgrund des Zusammenhangs zwischen Matrix-Dimensionen und Sequenzen in NW kann die Forderung, dass die Anordnung der Sequenzsymbole bewahrt werden muss, durch die Beschränkung der Bewegungsrichtung auf diagonal, horizontal und vertikal dargestellt werden. Dies ist nicht nur leichter verständlich als die Darstellung mit der im MILP-Modell aufgestellten Forderung (3.1.2.2.3) sondern kann dank des anschaulichen Zusammenhangs auch deutlich leichter plausibilisiert werden. Andererseits lässt sich die Problemformulierung mit dem Vokabular, das das MILP-Modell bietet, deutlich leichter auf höhere Dimensionen verallgemeinern. NW nutzt eine verzweigte Rekursionsbeziehung, bei der pro zusätzlicher Sequenz ein rekursives Argument hinzukommt; dies erschwert die mathematische Analyse in höheren Dimensionen.

5.2 Ausblick

Auf der Grundlage der gewonnenen Erkenntnisse lassen sich Themen und Fragestellungen für die weitergehende Forschung identifizieren.

5.2.1 Feste Alignmentlängen

Die vorgestellte Argumentation für Isomorphie von MILP und NW erscheint schlüssig. Allerdings liefert die vorliegende Arbeit keinen angepassten Algorithmus, der die fixe Länge produzierter Alignments garantiert.

Es wäre interessant, den in der Diskussion der Limitationen vorgestellten Ansatz für ein erweitertes Abbruchkriterium und eine darauf basierende Wahlfunktion für Kandidatenschritte präzise auszuformulieren. Wenn sich aus der Anwendung der angepassten Kandidatenfunktion eine obere Schranke für Alignmentlängen ergibt, wäre die Isomorphie von MILP und NW gezeigt.

5.2.2 Performanceanalyse

Die identifizierten Performanceprobleme sprechen für die Notwendigkeit einer eingehenden Analyse und möglichen Neuimplementierung bestimmter Programmteile.

Interessante Kandidaten für die Identifikation der verantwortlichen Störquellen sind einerseits die unnötigen Kopien bei der Befüllung der Matrix, welche höchstwahrscheinlich negative Auswirkungen auf die Ausführungszeit haben und andererseits der Backtrackingalgorithmus, der sowohl Ausführungszeit als auch Speichermanagement negativ beeinflusst.

5.2.3 Computergestützte Verifikation

Zu Beginn der Arbeit wurde die Machbarkeit händischer Beweise deutlich über- und deren Fehleranfälligkeit deutlich unterschätzt.

Folgearbeiten könnten erforschen, inwieweit sich das dargestellte Modell in einem Beweisassistenten formulieren lässt, sowie ob und wie diese Formulierung für eine automatische Verifikation der Implementation genutzt werden kann.

5.2.4 Erweiterung des Modells

Auch die in der Diskussion erörterten Ansätze zur Erweiterung des vorgestellten Modells auf das Problem multipler Sequenzalignments oder zur Nutzung einer anderen Art von Kostenfunktion können eine fruchtbare Grundlage für Folgearbeiten darstellen.

6 Quellen

- [1] S. B. Needleman und C. D. Wunsch, „A general method applicable to the search for similarities in the amino acid sequence of two proteins“, *Journal of Molecular Biology*, Bd. 48, Nr. 3, S. 443–453, März 1970, doi: [10.1016/0022-2836\(70\)90057-4](https://doi.org/10.1016/0022-2836(70)90057-4).
- [2] E. W. Dijkstra, „The humble programmer“, *Commun. ACM*, Bd. 15, Nr. 10, S. 859–866, Okt. 1972, doi: [10.1145/355604.361591](https://doi.org/10.1145/355604.361591).
- [3] D. E. Knuth, „Literate Programming“, *The Computer Journal*, Bd. 27, Nr. 2, S. 97–111, Jan. 1984, doi: [10.1093/comjnl/27.2.97](https://doi.org/10.1093/comjnl/27.2.97).
- [4] J. D. Thompson, D. G. Higgins, und T. J. Gibson, „CLUSTAL W: improving the sensitivity of progressive multiple sequence alignment through sequence weighting, position-specific gap penalties and weight matrix choice“, *Nucleic Acids Research*, Bd. 22, Nr. 22, S. 4673–4680, 1994, doi: [10.1093/nar/22.22.4673](https://doi.org/10.1093/nar/22.22.4673).
- [5] M. A. Larkin u. a., „Clustal W and Clustal X version 2.0“, *Bioinformatics*, Bd. 23, Nr. 21, S. 2947–2948, Sep. 2007, doi: [10.1093/bioinformatics/btm404](https://doi.org/10.1093/bioinformatics/btm404).
- [6] S. Henikoff und J. G. Henikoff, „Amino acid substitution matrices from protein blocks.“, *Proceedings of the National Academy of Sciences*, Bd. 89, Nr. 22, S. 10915–10919, 1992, doi: [10.1073/pnas.89.22.10915](https://doi.org/10.1073/pnas.89.22.10915).
- [7] M. O. Dayhoff, R. M. Schwartz, und B. C. Orcutt, „A Model of Evolutionary Change in Proteins“, in *Atlas of protein sequence and structure*, 3. Aufl., Bd. 5, M. O. Dayhoff, Hrsg., Washington, DC: National Biomedical Research Foundation, 1978, S. 345–352.
- [8] K. Tamura und M. Nei, „Estimation of the number of nucleotide substitutions in the control region of mitochondrial DNA in humans and chimpanzees.“, *Molecular Biology and Evolution*, Bd. 10, Nr. 3, S. 512–526, Mai 1993, doi: [10.1093/oxfordjournals.molbev.a040023](https://doi.org/10.1093/oxfordjournals.molbev.a040023).
- [9] Z. Yang, „Estimating the pattern of nucleotide substitution“, *Journal of molecular evolution*, Bd. 39, S. 105–111, 1994.
- [10] R. Bellman, „On the Theory of Dynamic Programming“, *Proceedings of the National Academy of Sciences*, Bd. 38, Nr. 8, S. 716–719, Aug. 1952, doi: [10.1073/pnas.38.8.716](https://doi.org/10.1073/pnas.38.8.716).
- [11] D. Sankoff, „Matching sequences under deletion/insertion constraints“, *Proceedings of the National Academy of Sciences*, Bd. 69, Nr. 1, S. 4–6, 1972.
- [12] T. F. Smith und M. S. Waterman, „Identification of common molecular subsequences“, *Journal of Molecular Biology*, Bd. 147, Nr. 1, S. 195–197, März 1981, doi: [10.1016/0022-2836\(81\)90087-5](https://doi.org/10.1016/0022-2836(81)90087-5).
- [13] A. Church, „A Formulation of the Simple Theory of Types“, *The Journal of Symbolic Logic*, Bd. 5, Nr. 2, S. 56–68, 1940, doi: [10.2307/2266170](https://doi.org/10.2307/2266170).
- [14] A. M. Turing, „Computability and λ -definability“, *Journal of Symbolic Logic*, Bd. 2, Nr. 4, S. 153–163, 1937, doi: [10.2307/2268280](https://doi.org/10.2307/2268280).
- [15] A. Sabry, „What is a purely functional language?“, *Journal of Functional Programming*, Bd. 8, Nr. 1, S. 1–22, 1998, doi: [10.1017/S0956796897002943](https://doi.org/10.1017/S0956796897002943).
- [16] C. A. R. Hoare, „An axiomatic basis for computer programming“, *Commun. ACM*, Bd. 12, Nr. 10, S. 576–580, Okt. 1969, doi: [10.1145/363235.363259](https://doi.org/10.1145/363235.363259).
- [17] S. R. McAllister, R. Rajgaria, und C. A. Floudas, „A Template-Based Mixed-Integer Linear Programming Sequence Alignment Model“, in *Models and Algorithms for Global Optimization: Essays Dedicated to Antanas Žilinskas on the Occasion of His 60th Birthday*, A. Törn und J. Žilinskas, Hrsg., Boston, MA: Springer US, 2007, S. 343–360. doi: [10.1007/978-0-387-36721-7_21](https://doi.org/10.1007/978-0-387-36721-7_21).

- [18] E. Althaus, A. Caprara, H.-P. Lenhof, und K. Reinert, „A branch-and-cut algorithm for multiple sequence alignment“, *Mathematical Programming*, Bd. 105, Nr. 2–3, S. 387–425, Nov. 2005, doi: [10.1007/s10107-005-0659-3](https://doi.org/10.1007/s10107-005-0659-3).
- [19] L. A. Wolsey, *Integer Programming*, 2. Aufl. John Wiley & Sons, 2020.
- [20] P. J. A. Cock u. a., „Biopython: freely available Python tools for computational molecular biology and bioinformatics“, *Bioinformatics*, Bd. 25, Nr. 11, S. 1422–1423, März 2009, doi: [10.1093/bioinformatics/btp163](https://doi.org/10.1093/bioinformatics/btp163).
- [21] R. Kaivola u. a., „Replacing Testing with Formal Verification in Intel® Core™ i7 Processor Execution Engine Validation“, in *Computer Aided Verification*, A. Bouajjani und O. Maler, Hrsg., Berlin, Heidelberg: Springer Berlin Heidelberg, 2009, S. 414–429.
- [22] 1900.-1982. Curry Haskell B., J. R. Hindley, und J. P. Seldin, *To H.B. Curry: essays on combinatory logic, lambda calculus, and formalism*. London: Academic Press, 1980.

Hiermit versichere ich an Eides statt durch meine Unterschrift, dass

- ich die vorliegende wissenschaftliche Arbeit selbständig und ohne unerlaubte Hilfe angefertigt habe,
- ich andere als die angegebenen Quellen und Hilfsmittel nicht benutzt habe,
- ich die den benutzten Quellen wörtlich oder inhaltlich entnommenen Stellen als solche kenntlich gemacht habe und dass
- die Arbeit in gleicher oder ähnlicher Form noch keiner anderen Prüfbehörde vorgelegen hat.

Berlin, den 12.08.2024

Fynn Freyer
Fynn Freyer
