Algorithmen für das Finden von Gen-Clustern

Viktor Krammer ⟨9826988⟩

Abstract. Beobachtungen haben gezeigt, dass funktional zusammengehörende Gene räumlich eng beieinander liegen. Diese Cluster gilt es nun in einer Serie von mehreren sequenzierten Genomen zu finden. Dabei werden die Sequenzen als Permutationen modelliert. Als gemeinsames Intervall bezeichnet man eine "Sequenz von Elementen", die in allen Permutationen in bliebiger Reihenfolge, jedoch in einem "Block" auftreten. In diesem Artikel werden vier Algorithmen für das Finden aller gemeinsamen Intervalle von n Permutationen vorgestellt. Dabei werden auch zirkuläre und partitionierte Gen-Sequenzen berücksichtigt. Alle Algorithmen sind linear in Bezug auf Zeit- und Speicheraufwand.

1 Einleitung

Dies ist eine Zusammenfassung des Artikels [HS01a] von Heber und Stoye, der sich mit dem Problem, Gen-Cluster in einer Serie von sequenzierten Genomen zu finden, befasst. Ausgehend von der starken Vermutung, dass sich funktional zusammengehörende Gene in diesen Sequenzen in Nachbarschaft befinden, sollen nun Algorithmen nach diesen Ketten suchen, die in allen zu untersuchenden Genomen vorkommen, sich jedoch nicht notwendigerweise in der gleichen Reihenfolge wiederholen.

Die zu untersuchenden Genome werden als Permutationen der sequenzierten Gene modelliert. Die Gene werden dabei einfach mit Natürlichen Zahlen bezeichnet. Das Problem ist nun, alle gemeinsame Intervalle, welche den Clustern entsprechen, zu finden.

Als Basis dient der Algorithmus RC von Uno und Yagiura [UY00], der alle gemeinsamen Intervalle $K \leq \binom{n}{2}$ zweier Permutationen von n Elementen findet. Dieser Algorithmus wird nun auf eine Familie von Permutationen (Kardinalität ≥ 2) verallgemeinert, der nach wie vor in optimaler Zeit $O(n + K)$ und mit optimalen Speicherverbrauch $O(n)$ operiert.

Da Genome aus einem oder mehreren Chromosomen bestehen und diese normalerweise auch die Grenze für Gen-Cluster darstellen, wird ein abgewandelter Algorithmus angegeben, der diese Tatsache berücksichtigt. Weiters treten in der Natur auch zyklische (kreisförmige) Genome auf, die eine Adaption des Algorithmus erfordern, um das Genom nicht an willkürlicher Stelle auseinanderzudividieren und so Gen-Cluster zu übersehen.

Dieser Artikel gibt im Abschnitt 2 eine exakte Definition von gemeinsamen Intervallen und den dazugehörigen Algorithmen an. Im Abschnitt 3 geht es um die Erweiterung auf Genome mit mehreren Choromsomen. Abschnitt 4 setzt mit der zyklischen Erweiterung fort. In Abschnitt 5 werden diese Erweiterungen kombiniert und im letzten Abschnitt 6 wird auf die Praxisrelevanz eingegangen.

2 Gemeinsame und irreduzible Intervalle

2.1 Grundlagen

Definition 1 *Eine Permutation π der Menge $N := \{1, 2, ..., n\}$ ist eine Umordnung der Elemente von N. Mit $\pi(i) = j$ bezeichnen wir, dass das i-te Element unter der Permutation π j ist. Für $1 \leq x \leq n$ definieren wir $[x, y] := \{x, x + 1, ..., y\}$ und nennen $\pi([x, y]) := \{\pi(i) | i \in [x, y]\}$ ein Intervall von π.*

Definition 2 *Sei $\Pi = (\pi_1, ..., \pi_k)$ eine Familie von k Permutationen über N, o.B.d.A. sei $\pi_1 = id_n := (1, ...n)$. Eine Teilmenge $c \subseteq N$ ist ein gemeinsames Intervall von Π, gdw. $1 \leq l_j < u_j \leq n$ für alle $1 \leq j \leq k$ existieren, sodass gilt*

$$c = \pi_1([l_1, u_1]) = \pi_2([l_2, u_2]) = ... = \pi_k([l_k, u_k]).$$

Im folgenden bezeichnen wir ein gemeinsames Intervall c entweder durch Angabe dessen Elemente, oder durch $\pi([l_i, u_j])$ für $j \in \{1, ..., n\}$ bzw. $[l_j, u_j]$ für $\pi_j = id_n$. Die Menge aller gemeinsamen Intervalle sei C_Π.

Beispiel Sei $N = 1, ..., 9$ und $\Pi = \{\pi_1, \pi_2, \pi_3\}$ mit $\pi_1 = id_9, \pi_2 = (3, 2, 1, 9, 7, 8, 6, 5, 4)$ und $\pi_3 = (4, 5, 6, 8, 7, 1, 2, 3, 9)$. Bezüglich π_1 ist

$$C_\Pi = \{[1, 2], [1, 3], [1, 9], [2, 3], [4, 5], [4, 6], [4, 8], [5, 6], [5, 8], [6, 8], [7, 8]\}.$$

□

2.2 Der Algorithmus RC [UY00]

Gegeben sei $\Pi = \{\pi_1, \pi_2\}$ mit $\pi_1 = id_n$. Mit Hilfe der drei Funktionen

$$\begin{aligned} l(x, y) &:= \min \pi_2([x, y]) \\ u(x, y) &:= \max \pi_2([x, y]) \\ f(x, y) &:= u(x, y) - l(x, y) - (y - x) \end{aligned}$$

kann leicht getestet werden, ob $\pi_2([x, y])$ ein gemeinsames Intervall von Π ist. Das ist der Fall, gdw. $f(x, y) = 0$, da $f(x, y)$ die Anzahl der Elemente des Intervalls $[l(x, y), u(x, y)] \backslash \pi_2([x, y])$ zählt. Ein einfacher Algorithmus könnte so aussehen, $f(x, y) = 0$ für alle Paare $1 \leq x < y \leq n$ zu testen.

Uno und Yagiura [UY00] haben für dieses Problem allerdings effizientere Algorithmen entworfen. Einer davon ist der Algorithmus RC, der Zeit dadruch einspart, dass $f(x, y) = 0$ nicht für alle Paare (x, y) getestet wird.

Definition 3 *Für ein fixes x wird ein oberes Intervallende $y > x$* ***unbrauchbar*** *(in [UY00]* ***wasteful****) genannt, falls $f(x', y) > 0$ für alle $x' \leq x$ gilt.*

Die Idee ist nun, die unbrauchbare y Kandidaten erst gar nicht zu testen. Als Datenstrukturen werden doppelt verkettete Listen für diese Kandidaten (Y) und zum effizienten Berechnen der Funktionen l und u verwendet. Zentral ist die äußerste Schleife, wo x von $n-1$ bis 1 läuft. Bei jedem Schleifendurchgang wird für jeden Kandidaten $y \in Y$ getestet, ob $[l(x, y), u(x, y)]$ ein gemeinsames Intervall ist. Danach werden unbrauchbare Intervallenden aus Y entfernt.

Algorithmus 1 (Reduce Candidate, RC)
Input: $\Pi = (\pi_1 = id_n, \pi_2)$
Output: C_Π

1: $Y := N$
2: **for** $x = n-1, ..., 1$ **do**
3: Ausgabe aller $y\ (> x) \in Y$, die $f(x, y) = 0$ erfüllen
4: $Y := Y \backslash W$ wobei $W \subseteq \{y \in N | y \geq x$ und $f(x', y) > 0$ für alle $x' < x\}$
5: **end for**

Unbrauchbare Kandidaten müssen lediglich vom Anfang der (sortierten) Y Liste gelöscht werden, wodurch sich eine insgesamte Laufzeit von $O(n + |C_\Pi|)$ ergibt. Für die genaue Verwaltung der Listen wird auf [UY00] verwiesen.

2.3 Irreduzible Intervalle

Bevor wir den Algorithmus RC auf gemeinsame Intervalle von $k \geq 2$ Permutationen verallgemeinern können, definieren wir eine erzeugende Teilmenge von C_Π bestehend aus den sogenannten irreduziblen Intervallen.

Definition 4 *Zwei gemeinsame Intervalle $c_1, c_2 \in C_\Pi$ haben eine nicht triviale Überlappung, wenn $c_1 \cap c_2 \neq \emptyset$ und keines der beiden Intervalle das andere enthält.*

Definition 5 *Eine Liste $p = (c_1, ..., c_{\ell(p)})$ von gemeinsamen Intervallen c_1 bis $c_\ell(p)$ wird Kette (der Länge $\ell(p)$) genannt, wenn jeweils zwei aufeinanderfolgende Intervalle eine nicht triviale Überlappung besitzen. Eine Kette mit nur einem Intervall wird trivale, alle anderen nicht triviale Kette genannt. Eine Kette, die an keinem der beiden Enden verlängert werden kann, ist eine maximale Kette. Es ist leicht zu sehen, dass $\tau(p) := \bigcup_{c' \in p} c'$, wobei p eine Kette ist, ein gemeinsames Intervall ist. Wir sagen p generiert $\tau(p)$.*

Definition 6 *Ein gemeinsames Intervall c heisst reduzibel, falls eine nicht triviale Kette existiert, die c generiert, andernfalls ist c irreduzibel.*

Diese Definition partitioniert $C_\Pi = I_\Pi \cup R_\Pi$ in eine Menge von irreduziblen und reduziblen Intervallen.

Beispiel Für das Beispiel von oben sind die irreduziblen Intervalle

$$I_\Pi = \{[1,2],[1,9],[2,3],[4,5],[5,6],[6,8],[7,8]\}$$

Die reduziblen Intervalle werden wie folgt genieriert:

$$\begin{aligned} [1,3] &= [1,2] \cup [2,3], \\ [4,6] &= [4,5] \cup [5,6], \\ [4,8] &= [4,5] \cup [5,6] \cup [6,8], \\ [5,8] &= [5,6] \cup [6,8]. \end{aligned}$$

□

2.4 Gemeinsame Intervalle von $k > 2$ Permutationen

Mit Hilfe der vorigen Definitionen und mit zwei Hilfssätzen aus [HS01b] kann das Problem dahingehend vereinfacht werden, nur noch alle irreduziblen gemeineinsamen Intervalle von k Permutationen zu finden und anschließend alle gemeinsamen Intervalle in $O(|C_\Pi|)$ Zeit zu erzeugen.

Algorithmus 2 (Suche aller gemeinsamen Intervalle)
Input: $\Pi = (\pi_1 = id_n, \pi_2, ..., \pi_k)$
Output: C_Π

1: $I_{\Pi_1} \leftarrow ([1,2],[2,3],...,[n-1,n])$
2: **for** $i = 2, ..., k$ **do**
3: $\quad I_{\Pi_i} \leftarrow \left\{\varphi_i(c) | c \in I_{\Pi_{i-1}}\right\}$ // siehe Algorithmus 3
4: **end for**
5: generiere C_Π aus $I_\Pi = I_{\Pi_k}$
6: Ausgabe von C_Π

Ausgehend von der Kette $I_{\Pi_1} = \{[j, j+1] | 1 \leq j \leq n-1\}$, berechnet der Algorithmus sukzessive I_{Π_i} aus $I_{\Pi_{i-1}}$ mit $\Pi_i := (\pi_1, ..., \pi_i)$. Dies geschieht durch die surjektive Funktion $\varphi_i : I_{\Pi_{i-1}} \rightarrow I_{\Pi_i}$, die jedem Intervall $c \in I_{\Pi_{i-1}}$ das kleinste gemeinsame Intervall $c' \in C_{\Pi_i}$ zuordnet.

Die Funktion φ kann effizient mittels des um eine Liste S erweiterten RC Algorithmus implementiert werden. S beinhaltet die Intervalle von $I_{\Pi_{i-1}}$, die noch nicht in I_{Π_i} abgebildet wurden. Ausgehend von π_1 und π_i und einem x sind die Kandidaten $y \in C_{(\pi_1, \pi_i)}$ in Y bekannt. Das Ziel von S ist, diese Kandidaten weiter einzuschränken und zwar auf jene Indizes y, die gleichzeitig

- $[x, y] \in C_{\Pi_{i-1}}$ (damit letztendlich auch $[x, y] \in C_{\Pi_i}$) und
- $[x, y]$ beinhaltet ein Intervall $c \in I_{\Pi_{i-1}}$, das in keinem kleineren Intervall von C_{Π_i} enthalten ist

erfüllen. Damit ist $[x, y] \in \varphi(I_{\Pi_{i-1}})$.

Algorithmus 3 (Erweiterter RC)
Input: $\Pi = (\pi_1 = id_n, \pi_i)$, $I_{\Pi_{i-1}}$
Output: I_{Π_i}

1: Y und S initialisieren
2: **for** $x = n - 1, ..., 1$ **do**
3: Y und S updaten
4: **while** ($[x', y] \leftarrow S$.first_active_interval(x) definiert und $f(x, y) = 0$) **do**
5: $[l(x, y), u(x, y)]$ ausgeben
6: $[x', y]$ in S deaktivieren
7: **end while**
8: **end for**

Die Schleife in Zeile 4 kann abgebrochen werden, wenn es in S keine Kandidaten mehr gibt oder $f(x, y) > 0$ gilt, da die Reihenfolge von S.first_active_interval zusichert, dass $f(x, y)$ monoton steigend ist. Für Einzelheiten, die die Datenstruktur S betreffen, sei der interessierte Leser auf [HS01b] verwiesen.

3 Erweiterung auf "multichromosome" Permutationen

Wie in der Einleitung angesprochen, kann ein Genom auch mehrere Chromosome besitzen. In diesem Abschnitt und dem folgenden wollen wir den Algorithmus 2 so erweitern, dass sich gemeinsame Intervalle nur über je ein Chromosom erstrecken. Sei nun wieder $N := 1, 2, ..., n$ die Menge von n Genen. Ein Chromosom c von N ist eine linear geordnete Teilmenge von N. Eine multichromosome Permutation π über N ist eine Menge von Chromosomen, die jedes Element von N genau einmal enthalten, d.h.

$$\pi = \{c_1, ..., c_\ell\} \quad \text{mit} \quad N = \bigcup_{1 \le i \le \ell} c_i.$$

Definition 7 *Sei $\Pi = (\pi_1, ..., \pi_k)$ eine Familie von k multichromosomen Permutationen über N, dann heißt $s \subseteq N$ ein gemeinsames Intervall von Π, gdw. für jede multichromosome Permutation $\pi_i, i = 1, ..., k$ ein Chromosom mit s als Intervall existiert.*

Beispiel Sei $N = \{1, ..., 6\}$ und $\Pi = (\pi_1, \pi_2, \pi_3)$ mit $\pi_1 = \{(1, 2, 3), (4, 5, 6)\}$, $\pi_2 = \{(1, 5, 6, 4), (3, 2)\}$ und $\pi_3 = \{(1, 6, 4, 5), (3), (2))\}$. Chromosome sind hier durch Klammern getrennt. Das einzige gemeinsame Intervall ist $\{4, 5, 6\}$. □

Der Algorithmus 2 wird wie folgt modifiziert: Da die Reihenfolge der Chromosome unwesentlich ist, werden diese in beliebiger Reihenfolge in eine (normale) Permutation zusammengefasst. Sei $\Pi' = (\pi'_1, \pi'_2, ..., \pi'_k)$ die so einstandene Familie mit o.B.d.A. $\pi'_1 = id_n$. Zeile 1 wird nun durch

$I_{\Pi'_1} = \{[j, j+1] | j, j+1$ befinden sich auf demselben Chromosom und $1 \leq j < n\}$

ersetzt und im Algorithmus 3 wird die while-Bedingung dahingehend ergänzt, dass sich x und y auf demselben Chromsom befinden müssen. Dieser zusätzliche Test kostet nur konstante Zeit und da es mit diesen Einschränkungen nicht mehr gemeinsame Intervalle geben kann, verändert sich der Aufwand nicht.

Theorem 1 *Seien k multichromosome Permutationen von $N = \{1, ..., n\}$ gegeben. Alle K gemeinsamen Intervalle können dann in $O(kn + K)$ Zeit mit $O(n)$ Speicheraufwand gefunden werden.* □

In einer ähnlichen Aufgabenstellung gilt es zu berücksichtigen, auf welchem DNA Strang sich die Gene befinden. Auch hier legen Beobachtungen nahe, dass funktional zusammengehörende Gene sich jeweils nur auf einem Strang befinden. Kodiert wird dieses Problem durch Permutationen mit Vorzeichen. Das Vorzeichen gibt an, auf welchem DNA Strang ein Gen liegt.

Beispiel Sei $N = \{1, ..., 6\}$ und $\Pi = (\pi_1, \pi_2, \pi_3)$ mit $\pi_1 = (+1, +2, +3, +4, +5, +6)$, $\pi_2 = (-3, -1, -2, +5, +4, +6)$ und $\pi_3 = (-4, +5, +6, -2, -3, -1)$.
Bezüglich π_1 ist [1,3] ein gemeinsames Intervall — [4,5] und [4,6] sind hingegen keine. □

Dieses Problem kann auf das von multichromosomen Permutationen zurückgeführt werden und zwar werden die vorzeichenbehafteten Permutationen einfach so partitioniert, dass bei jedem Vorzeichenwechsel eine neue Partition anfängt. Dies kann in linearer Zeit geschehen. Damit ändert sich nichts an der Größenordnung des Problems.

Theorem 2 *Seien k Permutationen mit Vorzeichen von $N = \{1, ..., n\}$ gegeben. Alle K gemeinsamen Intervalle können dann in $O(kn + K)$ Zeit mit $O(n)$ Speicheraufwand gefunden werden.* □

4 Erweiterung auf zirkuläre Permutationen

Da in der Natur auch zirkuläre DNA-Sequenzen vorkommen, muss der Alogirthmus 3 auch dahingehend erweitert werden, dass auch solche Cluster gefunden werden, die sonst durch die Linearisierung auseinanderdividiert wären.

Definition 8 *Sei* $\Pi = (\pi_1, ..., \pi_k)$ *eine Familie von* k *zirkulären Permutationen über* N *dann heißt* $c \subseteq N$ *ein gemeinsames Intervall von* Π*, gdw. die Elemente von* c *ohne Unterbrechung in jeder zirkulären Permutation auftreten.*

Beispiel Sei $N = \{1, ..., 6\}$ und $\Pi = (\pi_1, \pi_2, \pi_3)$ mit $\pi_1 = (1, 2, 3, 4, 5, 6)$, $\pi_2 = (2, 4, 5, 6, 1, 3)$ und $\pi_3 = (6, 4, 1, 3, 2, 5)$. Neben den trivialen ($N$, jedes Element $n \in N$ und $N \backslash n$) sind noch $\{1, 2, 3\}$, $\{1, 2, 3, 4\}$, $\{1, 4, 5, 6\}$, $\{2, 3\}$, $\{4, 5, 6\}$ und $\{5, 6\}$ gemeinsame Intervalle. □

Folgendes fällt sofort auf:

Lemma 1 *Sei* c *ein gemeinsames Intervall einer Familie* Π *von zirkulären Permutationen von* N*, dann ist das Komplement* $\bar{c} := N \backslash c$ *auch ein gemeinsames Intervall von* Π*.*

Beweis Dies folgt unmittelbar aus der Definition 8. □

Diese Tatsache rechtfertigt zunächst nur Intervalle der Länge $\leq \lfloor \frac{n}{2} \rfloor$ zu suchen und anschließend die restlichen durch Lemma 1 zu berechnen.

Algorithmus 4 (Gemeinsame Intervalle von zirkulären Permutationen finden)
Input: $\Pi = (\pi_1 = id_n, \pi_2, ..., \pi_k)$
Output: C_Π

1: $I^*_{\Pi_1} \leftarrow (\{1, 2\}, \{2, 3\}, ..., \{n-1, n\})$
2: **for** $i = 2, ..., k$ **do**
3: $\quad I^*_{\Pi^*_i} \leftarrow \left\{ \varphi^*_i(c) | c \in I^*_{\Pi_{i-1}} \right\}$
4: **end for**
5: generiere C^*_Π aus $I^*_\Pi = I^*_{\Pi_k}$
6: $\bar{C}^*_\Pi \leftarrow \{\bar{c} | c \in C^*_\Pi\}$
7: Ausgabe von $C^*_\Pi \cup \bar{C}^*_\Pi$

Die erweiterte Funktion φ^* berücksichtigt, dass es sich um zirkuläre Permutationen handelt und generiert irreduzible Intervalle der Größe $\leq \lfloor \frac{n}{2} \rfloor$. Dies wird dadurch erreicht, dass π_1 und π_i jeweils an zwei Stellen ($n \downarrow 1$ und $\lfloor \frac{n}{2} \rfloor \downarrow \lfloor \frac{n}{2} \rfloor + 1$) linearisiert und diese dann untereinander in gewohnter Art und Weise auf gemeinsame Intervalle untersucht werden. Da sich der Aufwand nur um einen konstanten Faktor vergrößert, gilt

Theorem 3 *Seien k zirkuläre Permutationen von $N = \{1, ..., n\}$ gegeben. Alle K gemeinsamen Intervalle können dann in $O(kn + K)$ Zeit mit $O(n)$ Speicheraufwand gefunden werden.* □

5 Kombination der Erweiterungen

Leider können die zuvor vorgestellten Erweiterungen in den Abschnitten 3 und 4 nicht so ohne weiteres kombiniert werden, da Lemma 1 dann nicht mehr gilt, wie das folgende Beispiel mit multichromosomen zirkulären Permutationen zeigt:

Beispiel Sei $N = \{1, ..., 8\}$ und $\Pi = (\pi_1, \pi_2)$ mit $\pi_1 = \{(1, 2, 3, 4), (5, 6, 7, 8)\}$ und $\pi_2 = \{(1, 3, 5, 6, 7), (2, 4, 8)\}$, wobei π_1 une π_2 zirkulär sind. Während $c = \{5, 6\}$ ein gemeinsames Intervall ist, ist dessen Komplement $N \backslash c = \{1, 2, 3, 4, 7, 8\}$ keines. □

Der im Artikel skizzierte Weg das Problem zu lösen, benötigt eine Vorverarbeitunsphase und passt den Algorithmus noch einmal an.

Im Vorverarbeitungsschritt werden die Chromosome künstlich unterteilt. Die Permutation π_1 wird nacheinander mit allen anderen Permutationen π_i verglichen, wobei untersucht wird, ob benachbarte Gen-Paare von π_1 auch in π_i auf einem einzigen Chromosom liegen, d.h. für jedes Chromosom $c = (\pi_1(l), \pi_1(l + 1), ..., \pi_1(r))$ werden die Paare $\{\pi_1(j), \pi_1(j + 1)\}$ $(\cup \{\pi_1(l), \pi_1(r)\}$ falls c zirkulär ist) mit $l \leq j \leq r - 1$ untersucht. Falls die Gene des zu untersuchenden Paares nicht auf einem Chromosom liegen, wird das Chromosom in π_1 genau zwischen diesen beiden Genen unterteilt. Falls ein zirkuläres Chromosom unterteilt wird, wird es dadruch linearisiert. Anschließend wird der Test in umgekehrter Richtung wiederholt, d.h. dass $\pi_i, 2 \leq i \leq k$ mit π_1 verglichen wird.

Danach folgt die eigentliche Verarbeitung die mit Fallunterscheidung arbeitet und im wesentlichen die beiden Erweiterungen aus Abschnitt 3 und 4 kombiniert. Gene, die nicht in zirkulären Chromosomen vorkommen, können mit dem Algorithmus für multichromsome Permutationen abgearbeitet werden. Im anderen Fall werden die Gene weiters partitioniert, sodass jedes zirkuläre Chromosom eine Partition darstellt. Danach werden diese Partitionen einzeln behandelt und innerhalb dieser, Gen für Gen einzeln betrachtet, es sei denn die ausgewählte Partition ist in allen Permutationen zirkulär. Dann kann Algorithmus 4 angewendet werden. Im gemischten Fall werden beide Algorithmen kombiniert. Für die genaue Vorgehensweise sei auf [HS01a] verwiesen.

Zusammenfassend begründet dies das abschließende Theorem:

Theorem 4 *Seien k gemischt multichromosome, zirkuläre bzw. lineare Permutationen über $N = \{1, ..., n\}$ gegeben. Alle K gemeinsamen Intervalle können dann in $O(kn + K)$ Zeit mit $O(n)$ Speicheraufwand gefunden werden.* □

6 Abschließende Bemerkung

Die Autoren des Originalartikels weisen zu Recht darauf hin, dass die hier vorgestellten Algorithmen zur Suche nach gemeinsamen Intervallen in der Praxis nur bedingt einsetzbar sind. Die Definition von gemeinsamen Intervallen, die den Gen-Clustern entsprechen sollen, ist für den realen Einsatz noch zu eng gefasst. Nicht selten fehlen in diesen Clustern nämlich einzelne Gene oder es kommen in der Sequenz zusätzliche vor. Solche Cluster können mit diesen Algorithmen nicht gefunden werden. Diese notwendige Ausweitung der Definitionen macht das Problem aber bei weitem schwerer.

7 Literatur

[HS01a] Steffen Heber, Jens Stoye, Algorithms for Finding Gene Clusters in O. Gascuel, B.M.E. Moret (Eds.), WABI 2001, LNCS 2149, SS. 252-263, Springer-Verlag Berlin Heidelberg, 2001

[HS01b] Steffen Heber, Jens Stoye, Finding all common intervals of k permutations in Proceedings of the 12th Annual Symposium on Combinatorial Pattern Matching, CPM 2001 Volume 2089 of Lecture Notes in Computer Science, SS. 207-219, Springer-Verlag New York, 2001

[UY00] Takeaki Uno, Mutsunori Yagiura, Fast Algorithms to Enumerate All Common Intervals of Two Permutations, Algorithmica Volume 26(2), SS. 290-309, Springer-Verlag New York, 2000

[SM97] João Setubal, João Meidanis, Introduction to computational molecular biology, Brooks/Cole Publishing Company, 1997