ETH
Eidgenössische Technische Hochschule Zürich
Swiss Federal Institute of Technology Zurich

D-ITET
Departement
für Elektrotechnik
und Informations-
technologie

Semesterarbeit
04/13/2004 -
07/16/2004

Autor
Thomas Frech

Professor
Prof.Dr.
Eckart Zitzler

Assistent
Stefan Bleuler

Biclustering von Genexpressionsdaten mit Ameisen Algorithmen

Titelfoto: mit freundlicher Genehmigung von A. San Juan
http://www.blueboard.com/leafcutters/images.htm

Inhaltsverzeichnis

1. Vorwort

Der Titel meiner Semesterarbeit ist irreführend. Auch wenn es in dieser Arbeit ursprünglich darum gehen sollte, zu zeigen, wie Genexpressionsdaten mit Ameisen Algorithmen analysiert werden können - zeigen musste ich schliesslich das Gegenteil. Und es ist sehr fraglich, ob Ameisen Algorithmen für diese Aufgabe überhaupt sinnvoll verwendet werden können. Neben prinzipiellen Gründen spricht auch die Tatsache, dass die deterministischen Algorithmen für die meisten Probleme so gut sind, dass es nichts mehr zu verbessern gibt, gegen den Einsatz von Ameisen Algorithmen.

Immerhin konnte ich den Algorithmus, der Amir Ben-Dor in einem Paper beschrieben hat, trotz anfänglichen Problemen erfolgreich implementieren und einen eigenen Algorithmus entwickeln, der viel schneller ist als derjenige von Ben-Dor. Zwar liefert mein eigener Algorithmus weniger gute Ergebnisse als derjenige von Ben-Dor, aber dafür kann er auch für Problemgrössen verwendet werden, die Ben-Dor's Algorithmus wegen der hohen Laufzeit unzugänglich sind.

Und gerade als ich - frustriert darüber, dass die Ameisen versagt haben - mit der Niederschrift dieses Berichtes begonnen habe, kam mir die Idee, wie Ben-Dor's Algorithmus und mein eigener zu vereinigen wären. Der daraus hervorgegangene Ben-Dor-Frech Algorithmus darf sich sehen lassen: er liefert bei fast gleichbleibender Laufzeit mindestens gleich gute, oft aber bessere Ergebnisse als der ursprüngliche Ben-Dor Algorithmus.

Und so beende ich diese Semesterarbeit mit einem lachenden und einem weinenden Auge.

Danken möchte ich Stefan Bleuler, der mir als Assistent stets hilfreich zur Seite stand, und Prof. Eckart Zitzler, in dessen Vorlesung ich gelernt habe, wie biologisch inspirierte Algorithmen erfolgreich zur Anwendung gebracht werden können - auch wenn es in dieser Arbeit nicht geklappt hat damit.

Zürich, 18. Juli 2004

Thomas Frech
tfrech@ee.ethz.ch

2. Einführung

2.1. Biclustering von Genexpressionsdaten

Die genetische Information, gespeichert in der DNA, ist in allen Zellen innerhalb eines Organismus identisch. Unterschiedliche Zellfunktionen kommen erst durch unterschiedliche Aktivitäten der Gene, die den Aufbau von Proteinen codieren, zu Stande: je aktiver ein Gen ist, desto mehr Protein wird gemäss dem Bauplan dieses Gens produziert. Dabei wird in einem Zwischenschritt, der "Genexpression" genannt wird, mRNA erzeugt. Durch die Messung der momentanen Konzentration der zu einem Gen gehörenden mRNA kann die momentane Aktivität dieses Gens bestimmt werden.

Mit Hilfe von Microarrays können die mRNA-Konzentrationen für alle Gene gleichzeitig gemessen werden. Da die meisten Prozesse einer Zelle über die Regulation der mRNA-Konzentration gesteuert werden, vermittelt eine solche Messung einen guten Eindruck über den Zustand einer Zelle. Werden solche Messungen unter verschiedenen Bedingungen vorgenommen (d.h. unter verschiedenen Umwelteinflüssen wie Lichtintensität oder Zufuhr von Chemikalien, während verschiedenen Wachstumsphasen oder Krankheitsstadien, etc.), erhält man eine Expressionsmatrix mit einer Zeile für jedes Gen und einer Spalte für jede Bedingung, unter der die Genexpressionen gemessen wurden. Der Matrixeintrag (a, b) steht also für den Expressionswert von Gen a unter der Bedingung b.

Um mehr über die Funktion von unbekannten Genen zu erfahren, werden Gruppen von ähnlich exprimierten Genen gesucht. Die Idee dabei ist, dass Gene, die eine ähnliche Funktion haben, sich auch ähnlich verhalten unter verschiedenen Bedingungen. Ist die Funktion eines Gens aus einer Gruppe bekannt, kann somit auf die Funktion der anderen Gene in dieser Gruppe geschlossen werden.

Um solche Gruppen von Genen zu finden, wurden verschiedene Clustering Algorithmen entwickelt, z.B. [6] und [5]. Diese Algorithmen ziehen aber immer alle vorhandenen Bedingungen in Betracht, um die Ähnlichkeit der Expressionsmuster verschiedener Gene zu bestimmen. Dies ist ein Nachteil, da sich gewisse Gene oft nur unter einem Teil der verschiedenen Bedingungen ähnlich verhalten: z.B. ähnliches Verhalten zu Beginn einer Wachstumsphase, danach divergierendes Verhalten. Solche Expressionsmuster werden von den Standardalgorithmen nicht erkannt.

Um diesen und weitere Nachteile der Standardalgorithmen zu beheben wurden so genannte Biclustering Algorithmen entwickelt. Dazu gehören unter anderen [1], [3] und [7]. Ein Bicluster ist definiert als eine Teilmenge G aller Gene, die über eine Teilmenge T der Bedingungen ähnlich exprimiert sind.

Das Ziel von Biclustering Algorithmen ist, in einer Expressionsmatrix möglichst grosse Bicluster zu finden, in denen sich die Gene möglichst ähnlich[1] verhalten. Einerseits, weil möglichst grosse Gengruppen gesucht werden, andererseits wegen der statistischen Signifikanz der Bicluster: Ein Bicluster ist statistisch umso signifikanter, je grösser er ist und je ähnlicher die Expressionsmuster der darin enthaltenen Gene sind. Dies, weil die Wahrscheinlichkeit, dass ein Bicluster in einer Matrix aus Zufallswerten vorhanden ist, mit zunehmender Bicluster-Grösse und zunehmender Ähnlichkeit der Expressionsmuster abnimmt. Je grösser und ähnlicher ein Bicluster also ist, desto kleiner ist die Wahrscheinlichkeit, dass er nur durch Zufall zu Stande kam und nichts mit der Zusammengehörigkeit von Genen zu tun hat.

Um den grossen Suchraum zu bewältigen (mehr dazu in Kapitel 3), verwenden Biclustering Algorithmen häufig eine "greedy" Strategie.

2.2. Das "Order Preserving Submatrix" Problem

Während einige Biclustering Algorithmen eine Ähnlichkeitsdefinition verwenden, die auf Korrelationsfunktionen beruht, besteht beim "Order Preserving Submatrix" Problem ein ganz anderer Ansatz: Es sollen Gene mit gleichen Trends gefunden werden. Dabei wird nur auf die Rangfolge der Expressionswerte geachtet, nicht auf den absoluten Wert.

Eine Order Preserving Submatrix (OPSM) ist definiert als Teilmenge G von Genen und Teilmenge T von Bedingungen, so dass alle Gene in G durch die Rangfolge ihrer Expressionswerte dieselbe linare Ordnung in T induzieren. Es besteht also eine Rangfolge der Bedingungen in T, die mit der der Rangfolge der Expressionswerte aller Gene in G übereinstimmt.

Das Ziel eines OPSM Algorithmus ist es, OPSM's zu finden, bei denen die Kardinalität von G und T möglichst gross ist.

Der Vorteil von OPSM Algorithmen ist, dass mit ihrer Hilfe Trends gefunden werden, an denen die Biologen interessiert sind. Im Gegensatz dazu finden Algorithmen, die Korrelationsfunktionen verwenden, oft Bicluster, bei denen sich die Gene zwar sehr ähnlich verhalten, andererseits aber die Expressionswerte sich nur wenig verändern und der Bicluster damit wenig Information enthält.

[1] Je nach Algorithmus wird eine unterschiedliche Definition von Ähnlichkeit verwendet, was dazu führt, dass verschiedene Algorithmen nur bedingt vergleichbar sind.

2.3. Ameisen Algorithmen

Ein Ameisen Algorithmus ist eine randomisierte Heuristik, die nach dem Vorbild des Verhaltens von Ameisen bei der Futtersuche funktioniert. Dieses Verhalten basiert auf positivem Feedback, da die Ameisen Wege nehmen, die bereits von vielen anderen Ameisen gewählt wurden, wobei jede Ameise eine Pheromonspur hinterlässt, die die Information über den benutzten Pfad weitergibt. Findet eine Ameise einen kurzen Weg vom Nest zu einer Futterstelle, so bewegt sie sich öfter zwischen dem Nest und der Futterstelle hin und her als eine gleich schnelle Ameise, die einen langen Weg nimmt. Dadurch kommt auf dem kurzen Weg mehr Pheromon zu liegen als auf dem langen, und somit steigt die Wahrscheinlichkeit, dass andere Ameisen ebenfalls den kurzen Weg wählen. Für das Travelling Salesman Problem (TSP) und weitere verwandte Probleme wurden Ameisen Algorithmen bereits erfolgreich angewandt. Eine detaillierte Beschreibung von Ameisen Algorithmen und ihren Anwendungen findet man in [2] und [4].

Da den Ameisen Algorithmen eine "greedy" Strategie zugrunde liegt, die randomisiert wird, und bestehende Biclustering Algorithmen meist "greedy" Strategien verwenden, sollten sie sich gut zur Kombination mit Ameisen Algorithmen eignen. Dies ist meines Wissens aber noch nie untersucht worden. Ein entsprechender Versuch soll in dieser Arbeit unternommen werden.

3. Mathematische Problemformulierung

3.1. Genexpressionsmatrix

Notation

$\mathbf{m}$	Anzahl Spalten (Bedingungen)
$\mathbf{n}$	Anzahl Zeilen (Gene)
$\mathbf{D_{row,col}}$	rangierter Matrixeintrag an Position (row,col)

Beschreibung

Wie in Abschnitt 2.1 beschrieben, werden die Genexpressionswerte in einer Matrix gespeichert. Da die Matrixeinträge aus Expressionsstärken darstellenden Fliesskommawerten bestehen, OPSM Algorithmen aber nur mit den Rangfolgen der Einträge arbeiten, besteht der erste Schritt von OPSM Algorithmen darin, jeden Matrixeintrag durch den zugehörigen Rang zu ersetzen. Prinzipiell könnte die Rangierung über die gesamte Matrix vorgenommen werden, so dass jeder Rang nur 1 Mal auftritt in der gesamten Matrix. Dies ist jedoch nicht notwendig, da OSPM Algorithmen nur den Rang von Einträgen innerhalb einer Zeile vergleichen müssen. Es genügt deshalb, die Rangierung zeilenweise vorzunehmen, also in jeder Zeile die Ränge von $1 - m$ zu vergeben. Rang 1 wird dem niedrigsten Expressionswert innerhalb einer Zeile zugeordnet, Rang m dem höchsten.

Beispiel

Die Expressionsmatrix

	0	**1**	**2**	**3**
0	5.6	7.2	1.3	2.8
1	3.2	1.1	9.3	4.3
2	7.0	8.5	7.0	2.1

wird in folgende rangierte Expressionsmatrix umgewandelt:

	0	**1**	**2**	**3**
0	3	4	1	2
1	2	1	4	3
2	2	4	3	1

Die Rangfolge von identischen Expressionswerten kann beliebig gewählt werden. Im vorhergehenden Beispiel könnten die Ränge 2 und 3 in der letzten Zeile vertauscht werden.

3.2. Order Preserving Submatrix (OPSM)

OPSM Definition

Eine Order Preserving Submatrix (OPSM) ist definiert als Teilmenge G von Genen und Teilmenge T von Bedingungen, so dass alle Gene in G durch die Rangfolge ihrer Expressionswerte dieselbe linare Ordnung in T induzieren. Es existiert also eine Rangfolge der Bedingungen in T, die mit der der Rangfolge der Expressionswerte aller Gene in G übereinstimmt.

Notation

s	Anzahl Spalten ($= \|T\|$)
k	Anzahl Zeilen ($= \|G\|$)
T	Menge der Spaltenindizes
π	Lineare Ordnung von T
G	Menge der Zeilenindizes

Die Menge T und die lineare Ordnung π werden in der Implementation durch ein einziges eindimensionales Integer-Array namens T repräsentiert. An Position 0 im Array steht der Index der Spalte mit dem niedrigsten Rang, an Position $m-1$ der Index der Spalte mit dem höchsten Rang. Die Menge G wird ebenfalls durch ein eindimensionales Integer-Array repräsentiert, wobei die Zeilen nach Indizes geordnet sind. An Position 0 im Array steht der niedrigste Zeilenindex, an Position $n-1$ der höchste Zeilenindex.

Die Notation wird damit folgendermassen vereinfacht:

s	Anzahl Spalten ($= \|T\|$)
k	Anzahl Zeilen ($= \|G\|$)
T	Array der Spaltenindizes, sortiert nach aufsteigendem Rang
G	Array der Zeilenindizes, sortiert nach aufsteigendem Zeilenindex

Im Folgenden wird die in der Implementation verwendete Notation, die T und π vereinigt, gebraucht, da sie einfacher und intuitiver ist. $T[i]$ sei der Spaltenindex an Position i, also der Spaltenindex mit Rang $i+1$. (Die Verschiebung kommt daher, dass die Arrayindizes von 0 - $(m-1)$ laufen, die Ränge aber von 1 - m.)

Model Definition

Array T (also die Menge T und die zugehörige lineare Ordnung π) wird auch als "Model" bezeichnet. Zeile r wird "kompatibel zu T" genannt, falls folgende Bedingung erfüllt ist:

$$D_{r,T[i]} < D_{r,T[j]} \quad \forall i,j \in [0, s-1],\ i < j \tag{3.1}$$

G enthält definitionsgemäss nur kompatible Zeilen.

Beispiel

Die rangierte Expressionsmatrix

	0	**1**	**2**	**3**	**4**
0	1	3	2	4	5
1	2	5	4	3	1
2	5	2	1	4	3
3	1	4	3	2	5
4	3	5	4	1	2
5	4	1	2	5	3

enthält folgende OPSM:

s	=	3
k	=	4
T	=	[0, 2, 1]
G	=	[0, 1, 3, 4]

Zeile r ist kompatibel, falls Gleichung 3.1 erfüllt ist für das gegebene Model:

$$D_{r,T[0]} < D_{r,T[1]} < D_{r,T[2]}$$

$$\Leftrightarrow D_{r,0} < D_{r,2} < D_{r,1}$$

Zeile 0 ist kompatibel, weil

$$D_{0,0} = 1 \ < \ D_{0,2} = 2 \ < \ D_{0,1} = 3$$

Zeile 1 ist kompatibel, weil

$$D_{1,0} = 2 \ < \ D_{1,2} = 4 \ < \ D_{1,1} = 5$$

Zeile 2 ist nicht kompatibel, weil

$$D_{2,0} = 5 \ \not< \ D_{2,2} = 1 \ < \ D_{2,1} = 2$$

Die Argumentation für die restlichen Zeilen funktioniert analog.

3.3. OPSM Problem

Für die Entwicklung von Algorithmen ist die Verwendung von realen Expressionsmatrizen aus zwei Gründen ungeeignet: Erstens können reale Expressionsmatrizen mehrere grosse, sich teilweise überschneidende OPSM's enthalten, was einen Algorithmus unter Umständen in die Irre leitet. Zweitens ist bei realen Expressionsmatrizen nicht bekannt, welche OPSM's sie enthalten - genau dieses Problem soll ja gelöst werden.

Es besteht daher Bedarf nach einem statistischen Modellproblem, anhand dessen Algorithmen entwickelt und beurteilt werden können. Für diese Arbeit wurde die Problemformulierung aus [1] - in leicht abgewandelter Form - übernommen:

Gegeben sei eine $n \times m$ Matrix mit gleichverteilten Zufallswerten. Darin werde genau 1 OPSM mit g Zeilen und t Spalten[1] implantiert[2], wobei die Auswahl der implantierten Zeilen und Spalten sowie die Rangfolge der ausgewählten Spalten gleichverteilt zufällig erfolgt.[3] Die Aufgabe eines OPSM Algorithmus ist es dann, die implantierte OPSM möglichst vollständig zu finden.

Da die gegebene Zufallsmatrix nicht nur die implantierte OPSM, sondern auch viele andere - meist kleinere - OPSM's enthält[4], stellt sich folgende Frage: Wenn ein Algorithmus mehrere OPSM's findet, welche soll dann als Ergebnis ausgegeben werden?

Da es zwei Parameter - Anzahl Zeilen (k) und Anzahl Spalten (s) - zu maximieren gilt, wird sofort klar, dass man zunächst eine Menge von Pareto-optimalen OPSM's erhält.

[1] Um die Anzahl der implantierten Zeilen und Spalten von den Grössen s und k, die die Anzahl Zeilen und Spalten einer OPSM repräsentieren, zu unterscheiden, weiche ich hier von Ben-Dor's Notation ab.

[2] Eine OPSM zu implantieren bedeutet, die ausgewählten Zeilen durch Umordnen der Matrixeinträge kompatibel zu machen zur Rangfolge der ausgewählten Spalten.

[3] Im Gegensatz dazu werden in [1] nur die t Spalten gleichverteilt zufällig auswählt, während jede der n Zeilen mit einer Wahrscheinlichkeit von $p = g/n$ implantiert wird, was zur Folge hat, dass die Zahl der implantierten Zeilen durch den Zufallsgenerator bestimmt wird.

[4] Wie leicht einzusehen ist, enthält z.B. schon eine winzige 3×2 Matrix gezwungenermassen immer eine 2×2 OPSM.

Ein OPSM Algorithmus kann nun entweder alle Pareto-optimalen OPSM's als Lösung ausgeben, oder er kann die beste OPSM auswählen - wobei zunächst zu definieren ist, was "beste" heissen soll.

Der einfachste Ansatz ist, die Fläche als Qualitätsmass für eine OPSM zu verwenden:

$$score(k, s) = area(k, s) = k \cdot s \tag{3.2}$$

Allerdings liefert dieses Mass keine Information über die statistische Signifikanz der OPSM. Und - noch schlimmer - statistisch wenig signifikante OPSM's werden oft besser bewertet als statistisch signifikante.

Amir Ben-Dor verwendet in [1] eine Approximation der statistischen Signifikanz zur Bewertung von OPSM's:

$$score(k, x) = U(n, m, k, s) = m \cdots (m - s + 1) \sum_{i=k}^{n} \binom{n}{i} \left(\frac{1}{s!}\right)^{i} \left(1 - \frac{1}{s!}\right)^{(n-i)} \tag{3.3}$$

Diese Approximationsformel liefert eine obere Schranke für die Wahrscheinlichkeit, dass eine OPSM der Grösse $k \times s$ in einer $n \times m$ Matrix aus gleichverteilten Zufallswerten auftritt. Je kleiner der Wert $U(n, m, k, s)$, desto signifikanter ist die OPSM.

Ein Algorithmus wird gemäss Amir Ben-Dor dann als erfolgreich angesehen, wenn er die implantierte OPSM oder eine statistisch signifikantere OPSM findet.

Die Verwendung der statistischen Signifikanz bereitet allerdings zwei Probleme:

- Die exakte Formel ist sehr komplex, weil die Wahrscheinlichkeit, dass eine Zeile r zu einem Model A kompatibel ist, nicht unabhängig ist von der Wahrscheinlichkeit, dass Zeile r zu einem sich mit A überschneidenden Model B kompatibel ist. Für praktische Zwecke ist man daher fast gezwungen, eine Approximation wie die oben Gezeigte zu verwenden, wobei uns Ben-Dor im Unklaren darüber lässt, wie treu diese Approximation ist.
- Die Wahrscheinlichkeit, dass eine OPSM in einer Zufallsmatrix auftritt, nimmt mit zunehmendem s und k rapide ab. Schon bei mässig grossen Expressionsmatrizen (einige hundert Zeilen, einige zehn Spalten), liegt der Funktionswert von $U(n, m, k, s)$ bei vielen OPSM's so nahe bei Null, dass er nicht mehr durch die in Programmiersprachen zurzeit gängigen 64-Bit-Fliesskommatypen (double in C++ und Java) dargestellt werden kann (siehe Abbildung 3.1). Damit leidet entweder die Genauigkeit (statistisch hoch signifikante OPSM's sind nicht unterscheidbar), oder der Rechenaufwand muss stark erhöht werden.

Ich habe diese Probleme umgangen, indem ich alle Algorithmen so implementiert habe, dass jeweils alle gefundenen OPSM's, die pareto-optimal sind, ausgegeben werden. Ein

Abbildung 3.1.: Einige Werte von $U(n, m, k, s)$, berechnet mit Matlab

n	m	k	s	$U(n, m, k, s)$
500	50	10	10	2.31e-29
500	50	25	10	3.94e-106
500	50	50	10	0
500	50	10	20	3.88e-132
500	50	10	30	0
1000	100	10	10	4.18e-23
1000	100	25	10	3.03e-95
1000	100	50	10	0
1000	100	10	20	4.73e-122
1000	100	10	30	0

Algorithmus wird dann als erfolgreich angesehen, wenn er die implantierte OPSM, eine in beiden Dimensionen gleich grosse OPSM, oder eine die implantierte OPSM dominierende OPSM findet. Es ist dem Anwender der Algorithmen überlassen, allenfalls eine Bewertungsfunktion zu bestimmen und anzuwenden.

Wenn ich trotzdem wissen wollte, ob eine von einem Algorithmus in einer Matrix M gefundene OPSM O statistisch signifikant ist (z.B. um OPSM's zu beurteilen, die in realen Expressionsmatrizen gefunden wurden), bestimmte ich die statistische Signifikanz experimentell, indem ich alle verfügbaren Algorithmen auf einer Zufallsmatrix der gleichen Grösse wie M, aber ohne implantierte OPSM, laufen liess. Waren die damit gefundenen Pareto-optimalen OPSM's, selbst wenn sie leicht vergrössert würden, nicht in der Lage, O zu dominieren, dann bin ich davon ausgegangen, dass O statistisch signifikant ist. Auch wenn man mit diesem Verfahren nur beweisen kann, dass eine OPSM statistisch nicht signifikant ist, nicht aber, dass eine OPSM statistisch signifikant ist, so lässt die Entfernung von O zur Pareto-Front der auf der Zufallsmatrix gefundenen OPSM's doch eine ungefähre Einschätzung der statistischen Signifikanz zu.

3.4. OPSM Problem Komplexität

Die Suche nach einer OPSM beschränkt sich im Wesentlichen auf die Suche nach einem Model. Zu prüfen, welche Zeilen zu einem gegebenen Model kompatibel sind, ist im Vergleich dazu keine aufwendige Angelegenheit. Es muss lediglich für jede Zeile geprüft werden, ob an irgendeiner Stelle eine Verletzung der durch das Model vorgegebenen Reihenfolge der Expressionswerte vorliegt, um eine Zeile als kompatibel oder nicht kompatibel zu qualifizieren.

Die Grösse eines Models zu einer Matrix mit m Spalten ist eine natürliche Zahl im Bereich $[1, m]$. Für jede Model-Grösse s existieren $m \cdot (m-1) \cdot ... \cdot (m-s+1) = m!/(m-s)!$ verschiedene Models. Die totale Anzahl Models ist daher

$$N(m) = \sum_{s=1}^{m} m \cdot (m-1) \cdot ... \cdot (m-s+1) = m! \cdot \sum_{s=1}^{m} \frac{1}{(m-s)!} \approx e \cdot m!$$

Es ist offensichtlich, dass es bei der Betrachtung von real grossen Matrizen (einige zehn bis einige hundert Spalten) viel zu viele Models gibt, als dass man sie der Reihe nach testen könnte, um diejenigen Models zu finden, die zu Pareto-optimalen OPSM's gehören. Und da das OPSM Problem gemäss [1] NP-vollständig ist, besteht die Aufgabe darin, Heuristiken zu entwickeln, die die Laufzeitkomplexität minimieren und dennoch gute Ergebnisse liefern.

4. Algorithmen

4.1. Übersicht

Während meiner Arbeit habe ich verschiedene Algorithmen zur Lösung des OPSM-Problems entwickelt und implementiert. In diesem Kapitel werden die Algorithmen beschrieben und in Kapitel 5 werden sie anhand von Tests auf grossen Datenmengen miteinander verglichen.

Die folgende Liste gibt eine kurze Übersicht über die implementierten Algorithmen:

Frech	meine eigene, einfache Heuristik
Frech AS	Frech Algorithmus erweitert mit Ant System
Frech ACS	Frech Algorithmus erweitert mit Ant Colony System
Frech Random	randomisierter Frech Algorithmus
Ben-Dor	Algorithmus von Amir Ben-Dor [1]
Ben-Dor Reloaded	gleich wie Ben-Dor Algorithmus, aber schneller
Ben-Dor ACS	Ben-Dor Algorithmus erweitert mit Ant Colony System
Ben-Dor Random	randomisierter Ben-Dor Algorithmus
Ben-Dor Plus	erste, einfache Erweiterung des Ben-Dor Algorithmus
Ben-Dor-Frech	ausgefeiltere Erweiterung des Ben-Dor Algorithmus
Ben-Dor-Frech Reloaded	gleich wie Ben-Dor-Frech Algorithmus, aber schneller

Jedem der aufgelisteten Algorithmen (ausser den Random- und Reloaded-Versionen) ist je ein Abschnitt gewidmet. Die Random- und Reloaded-Versionen werden im Abschnitt der jeweiligen Basisalgorithmen erläutert.

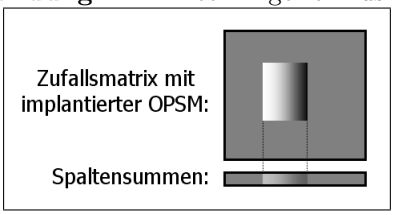
Im letzten Abschnitt befinden sich Informationen zu den Implementationsdetails.

4.2. Frech Algorithmus

4.2.1. Idee

In einer Zufallsmatrix bilden die Spaltenmittelwerte eine Normalverteilung um den mittleren Rang, wobei die durch die Spaltenmittelwerte implizierte Rangfolge der Spalten völlig zufällig ist. In einer OPSM ist die Varianz der Spaltenmittelwerte deutlich grösser als in einer Zufallsmatrix, weil alle Einträge der OPSM mit der definierten Rangfolge der Spalten übereinstimmen müssen. Daher kann eine in eine Zufallsmatrix implantierte OPSM den Spaltenmittelwerten der Zufallsmatrix die Rangfolge der implantierten Spalten aufprägen, sofern die OPSM nicht zu klein ist und im statistischen Rauschen untergeht. Da nur die Rangfolge der Spaltenmittelwerte ausschlaggebend ist, können die Spaltenmittelwerte durch die Spaltensummen ersetzt werden. Abbildung 4.1 veranschaulicht diese Idee.

Abbildung 4.1.: Frech Algorithmus Idee



Diese Idee greift allerdings nur dann optimal, wenn kein gegenläufiger Trend vorhanden ist. Ansonsten löschen sich Trend und Gegentrend teilweise aus. Im statistischen Modell von Amir Ben-Dor gibt es keine gegenläufigen Trends (obwohl der Ben-Dor Algorithmus damit keine Mühe haben sollte), in biologischen Messdaten leider schon. In Abschnitt 7.1 wird ein möglicher Ausweg aufgezeigt.

4.2.2. Begriffsdefinition: Kollision

Eine einzelne Verletzung der Bedingung

$$D_{r,T[i]} < D_{r,T[j]} \ , \ i < j$$

wird im Zusammenhang mit dem Frech Algorithmus als "Kollision" bezeichnet. Die Zeile r und die Spalten $T[i]$ und $T[j]$ werden als "Verursacher der Kollision" bezeichnet.

Die Bedeutung einer Kollision ist die, dass eine Zeile, die eine Kollision verursacht, nicht kompatibel ist zur Rangfolge der beiden in die Kollision involvierten Spalten.

4.2.3. Beschreibung

Der Frech Algorithmus arbeitet mit einer einer einzigen Submatrix, die iterativ verändert wird. In der Implementierung wird diese Submatrix durch eine Instanz der Klasse "Frech-Model" repräsentiert, wobei zu beachten ist, dass die Submatrix erst am Schluss eine OPSM gemäss Definition in Abschnitt 3.2 ist.

Zu Beginn besteht die Submatrix aus der gesamten Expressionsmatrix. Dann wird folgende Iteration ausgeführt:

1. Es werden alle Submatrixeinträge spaltenweise aufsummiert. Aus den Spaltensummen wird die mittlere Rangfolge extrahiert und als Model im Array T gespeichert.

2. Für jede Zeile und jedes Spaltenpaar $\{T[i1],\ T[i2]\}$ der Submatrix wird getestet, ob eine Kollision verursacht wird. Für jede Zeile und jede Spalte der Submatrix wird die Anzahl der mitverursachten Kollisionen aufsummiert.

3. Es wird ein "Schönheitswettbewerb" durchgeführt: Je weniger Kollisionen eine Zeile bzw. Spalte verursacht, desto wünschenswerter ist es, sie in der Submatrix zu behalten (weil eine Zeile bzw. Spalte, die zu einer grossen OPSM gehört, im Mittel weniger Kollisionen verursacht, als eine, die nicht zu einer grossen OPSM gehört). Es werden also diejenigen Zeilen bzw. Spalten entfernt, die am meisten Kollisionen verursachen. Pro Iterationsschritt werden entweder nur Zeilen oder nur Spalten entfernt. Um zu bestimmen, ob Zeilen oder Spalten entfernt werden sollen, wird der Parameter *ratio* benötigt, der das Ziel-Seitenverhältnis $k : s$ der OPSM festlegt. Besteht ein Spaltenüberschuss, werden Spalten entfernt, sonst Zeilen. Die Anzahl zu entfernender Zeilen bzw. Spalten richtet sich nach der Grösse der Submatrix und der Anzahl Iterationsschritte, die bereits ausgeführt wurden. Zu Beginn werden viele Zeilen bzw. Spalten auf einmal entfernt, am Schluss nur noch wenige (aber mindestens eine) pro Iterationsschritt.

4. Alle bereits entfernten Zeilen, die kompatibel sind zur Rangfolge der übriggebliebenen Spalten, werden wieder hinzugefügt.

Die Iterationsschritte von 1 bis 4 werden solange durchgeführt, bis die Submatrix eine OPSM ist. Die gefundene OPSM wird in einen Record abgelegt, der die Pareto-optimalen[1] OPSM's aufbewahrt für die Resultatausgabe.

Da das Seitenverhältnis der implantierten OPSM unbekannt ist, muss die Iteration für verschiedene Werte des Parameters *ratio* ausgeführt werden. Der resultierende Algorithmus nimmt daher 3 Parameter entgegen: Einen Start- und einen Endwert für *ratio* sowie einen Faktor, um den *ratio* erhöht wird von einer Iteration zur nächsten. Welcher Bereich *ratio* abdecken sollte, hängt davon ab, welche Seitenverhältnisse eine statistisch signifikante OPSM haben kann in der gegebenen Expressionsmatrix.

[1] Pareto-optimal bezüglich s und k, siehe auch Abschnitt 3.3

4.2.4. Parameter

Interne Parameter des Algorithmus

ratio	Ziel-Seitenverhältnis der gesuchten OPSM, $ratio := k : s$

Eingabe-Parameter des Algorithmus

lowest_ratio	Startwert für *ratio*
highest_ratio	Endwert für *ratio*
ratio_factor	Faktor, um den *ratio* erhöht wird nach einer Iteration

4.2.5. Programmstart

Allg. Kommandozeilenaufruf

```
java opsmextraction.frech.FrechMain <dataset filename>
   <lowest_ratio> <highest_ratio> <ratio_factor>
```

Beispiel

```
java opsmextraction.frech.FrechMain data/dataset.txt 0.25 8 2
```

führt den Frech Algorithmus auf der Expressionsmatrix in der Datei "data/dataset.txt" mit den *ratio*-Werten 0.25, 0.5, 1, 2, 4 und 8 aus.

4.2.6. Pseudocode

```
create empty pareto model record
set ratio = lowest_ratio

do
{
   create model which contains all columns and rows of dataset

   do
   {
      determine average column order of columns in T by
         columnwisely summing up the ranks

      determine collision count of all columns in T and rows in G

      if (k/s<ratio) // too many columns
      {
         if (s<m/4)
            set a = max(1,s/16)
         else if (s<m/2)
            set a = max(1,s/8)
         else
            set a = max(1,s/4)

         remove from T those a columns with the highest collision counts

         add all rows to G which were removed before but are compatible
            to the new T
      }
      else // too many rows
      {
         if (k<n/4)
            set a = max(1,k/16)
         else if (k<n/2)
            set a = max(1,k/8)
         else
            set a = max(1,k/4)

         remove those a rows from G with the highest collision counts
```

```
      }
   }
   while (G and T do not represent an OPSM)

   add all rows to G which are compatible to the final model
   add all columns to T which are compatible to the final model
   add the final model to the pareto model record

   set ratio = ratio * ratio_factor
}
while (ratio <= highest_ratio)

output content of pareto model record
```

4.2.7. Laufzeitkomplexität

Die Zahl der Werte, die für *ratio* verwendet werden - also die Anzahl Iterationen - wird als konstant angenommen. Dies, weil *ratio* nur ein Verhältnis von Seitenlängen darstellt, und somit Veränderungen der Expressionsmatrixgrösse keinen Einfluss auf die Zahl der zu verwendenden Werte für *ratio* haben, solange die Seitenverhältnisse in einem bestimmten Bereich bleiben.

Die aufwendigste Arbeit in einem Iterationsschritt ist das Zählen der Kollisionen. Im Iterationsschritt i muss für ${m_i}^2$ Spaltenpaare und n_i Zeilen getestet werden, ob eine Kollision verursacht wird, wobei m_i die Anzahl Spalten und n_i die Anzahl Zeilen darstellt, die in Iterationsschritt i bearbeitet werden müssen. Iterationsschritt i benötigt also den Aufwand $c \cdot n_i \cdot {m_i}^2$, wobei c eine unbekannte Konstante ist.

Fasst man je einen Iterationsschritt, bei dem Zeilen entfernt werden, und einen Iterationsschritt, bei dem Spalten entfernt werden, zu einem einzigen Iterationsschritt i zusammen, so werden pro Iterationsschritt jeweils $f \cdot n_i$ Zeilen und $f \cdot m_i$ Spalten entfernt, mit $f = 1/16$ (abgesehen von den ersten paar Iterationsschritten, bei denen f grösser ist). Damit gilt $n_{i+1} = (1 - f) \cdot n_i$ und $m_{i+1} = (1 - f) \cdot m_i$.

Der Aufwand einer Iteration mit z Iterationsschritten berechnet sich damit zu

$$c \cdot n_0 \cdot {m_0}^2 + c \cdot n_1 \cdot {m_1}^2 + ... + c \cdot n_{z-1} \cdot {m_{z-1}}^2$$

$$= c \cdot \sum_{i=0}^{z-1} n_i \cdot {m_i}^2$$

$$= c \cdot \sum_{i=0}^{z-1} (1-f)^i \cdot n \cdot ((1-f)^i \cdot m)^2$$

$$= c \cdot \sum_{i=0}^{z-1} (1-f)^{3i} \cdot n \cdot m^2$$

$$= c \cdot n \cdot m^2 \cdot \sum_{i=0}^{z-1} (1-f)^{3i}$$

Die Summe kann wie folgt umgeschrieben werden:

$$\sum_{i=0}^{z-1} (1-f)^{3i} = \sum_{i=0}^{z-1} g^i \quad \text{mit } g = (1-f)^3 < 1$$

Je grösser n und m werden, desto mehr Iterationsschritte müssen pro Iteration ausgeführt werden. Im Extremfall, also für $z \to \infty$, gilt

$$\sum_{i=0}^{\infty} g^i = \frac{1}{1-g} = \frac{1}{1-(1-f)^3} = const.$$

Da diese Summe konstant ist, leistet sie keinen Beitrag zur Laufzeitkomplexität. Diese beträgt somit $O(n \cdot m^2)$. Falls $O(n) = O(m)$ gilt, kann die Laufzeitkomplexität zu $O(n^3)$ vereinfacht werden.

Die Laufzeitmessungen in Abschnitt 5.3.2 bestätigen die berechnete Laufzeitkomplexität.

4.2.8. Bemerkungen

1. Ursprünglich habe ich den Algorithmus so implementiert, dass in jedem Iterationsschritt nur 1 Zeile bzw. Spalte entfernt wird. Ich habe aber festgestellt, dass die Verringerung der Laufzeitkomplexität von $O(n^4)$ zu $O(n^3)$ kaum Verschlechterungen bei der Ergebnisqualität bringt mit dem gewählten Muster zur Bestimmung der Anzahl zu entfernenden Zeilen bzw. Spalten. Nur in seltenen Fällen hat sich die Grösse der gefundenen OPSM leicht reduziert. Die Festlegung des Musters basiert jedoch nicht auf systematischen Tests mit grossen Datenmengen. Verfeinerungen sind in diesem Bereich möglich.

2. Es hat sich gezeigt, dass mit einem Wert für $ratio\_factor$ um 1.5 gute Ergebnisse erzielt werden. Kleinere Werte bringen kaum Verbesserungen. Auch hier könnten systematische Tests mit grossen Datenmengen genauere Erkenntnisse liefern.

4.2.9. Frech Random

Um in Abschnitt 5.6 zu zeigen, weshalb dem Frech AS Algorithmus aus Abschnitt 4.3 kein Erfolg beschieden ist, habe ich eine randomisierte Version des Frech Algorithmus implementiert. Der Frech Random Algorithmus wählt die a zu entfernenden Zeilen bzw. Spalten zufällig aus, wobei die Wahrscheinlichkeit für eine Zeile bzw. Spalte, entfernt zu werden, proportional ist zur Anzahl Kollisionen, die sie verursacht.

4.3. Frech AS Algorithmus

4.3.1. Idee

In [2] wird beschrieben, wie ein Ant System (AS) für die Lösung des Travelling Salesman Problem (TSP) verwendet wird. Die dem Frech AS Algorithmus zugrunde liegende Idee ist, das TSP-AS für den Frech Algorithmus zu adaptieren. Die Adaption beruht im Wesentlichen darauf, Pheromon auf Zeilen und Spalten zu legen - statt auf Kanten zwischen Knoten wie beim TSP - und das Distanzmass des TSP durch die Kollisionszahl zu ersetzen. Je mehr Pheromon auf einer Zeile bzw. Spalte liegt, desto geringer die Wahrscheinlichkeit, dass sie entfernt wird.

Die Implementation des Frech AS Algorithmus war als Vorarbeit für den komplexeren Frech ACS Algorithmus gedacht.

4.3.2. Adaption des TSP-AS

Analogietabelle

AS für TSP	**AS für Frech Algorithmus**
Kanten	Zeilen und Spalten
Knotenindizes i, j	Spaltenindizes i, Zeilenindizes j
Distanz d_{ij} (statisch)	Kollisionszahl cc_i, cc_j (dynamisch)
Sichtbarkeit $\eta_{ij} = 1/d_{ij}$ (statisch)	Kompatibilität $\eta_i = 1/cc_i$, $\eta_j = 1/cc_j$ (dynamisch)
Transitionswahrscheinlichkeit p_{ij} (Gl. 4.1)	Haltewahrscheinlichkeit p_i, p_j (Gl. 4.7 und 4.8)
Pheromonupdate $\Delta\tau_{ij}$	Pheromonupdate $\Delta\tau_i$, $\Delta\tau_j$

Herleitung der Haltewahrscheinlichkeit

Im Frech Algorithmus werden pro Iterationsschritt die a Zeilen oder Spalten mit der höchsten Kollisionszahl entfernt. In der AS-Version des Frech Algorithmus bleibt die Zahl a die selbe, nur die Auswahl der Zeilen bzw. Spalten, die entfernt werden, ist unterschiedlich. Jeder Zeile bzw. Spalte wird eine Haltewahrscheinlichkeit zugeordnet: die Wahrscheinlichkeit, dass die betreffende Zeile bzw. Spalte nicht entfernt wird. Ausgangspunkt für die Herleitung der Formel für die Haltewahrscheinlichkeit ist die Formel für die Transitionswahrscheinlichkeit beim TSP-AS:

$$p_{ij} = \frac{{\tau_{ij}}^{\alpha} \cdot {\eta_{ij}}^{\beta}}{\sum_{l \in J_i} {\tau_{il}}^{\alpha} \cdot {\eta_{il}}^{\beta}} \tag{4.1}$$

Es wird die Herleitung für Spalten gezeigt. Die Herleitung für Zeilen funktioniert analog, nur die Benennung der Indizes ändert.

Bedingungen:

1. pro Iterationsschritt sollen a Spalten entfernt werden $\Rightarrow \sum_{i=1}^{s}(1-p_i) = a$.
2. $p_i \sim {\tau_i}^{\alpha} \cdot {\eta_i}^{\beta} \Rightarrow p_i = f \cdot {\tau_i}^{\alpha} \cdot {\eta_i}^{\beta}$
3. es soll keine Spalte entfernt werden, die keine Kollisionen verursacht $\Rightarrow p_i = 1$ falls $cc_i = 0$

Bedingungen 2 und 3 implizieren

$$p_i = 1 \text{ falls } cc_i = 0,\ p_i = f \cdot {\tau_i}^{\alpha} \cdot {\eta_i}^{\beta} \text{ sonst} \tag{4.2}$$

$T' :=$ Menge der Spalten in T, die Kollisionen verursachen (d.h. Spalten in T mit $cc_i > 0$)

Bedingungen 1 und 3 implizieren

$$\sum_{l \in T'}(1-p_l) = a \tag{4.3}$$

Indem p_l durch 4.2, ersetzt wird, erhält man

$$\sum_{l \in T'}(1 - f \cdot {\tau_l}^{\alpha} \cdot {\eta_l}^{\beta}) = a \tag{4.4}$$

$$\Leftrightarrow |T'| - f \cdot \sum_{l \in T'} {\tau_l}^{\alpha} \cdot {\eta_l}^{\beta} = a \tag{4.5}$$

$$\Leftrightarrow f = \frac{|T'| - a}{\sum_{l \in T'} {\tau_l}^{\alpha} \cdot {\eta_l}^{\beta}} \tag{4.6}$$

Durch einsetzen von 4.6 in 4.2 erhält man die Haltewahrscheinlichkeit für Spalten:

$$p_i = 1 \text{ falls } cc_i = 0,\ p_i = \frac{(|T'| - a) \cdot {\tau_i}^{\alpha} \cdot {\eta_i}^{\beta}}{\sum_{l \in T'} {\tau_l}^{\alpha} \cdot {\eta_l}^{\beta}} \text{ sonst} \tag{4.7}$$

Die Haltewahrscheinlichkeit für Zeilen ist

$$p_j = 1 \text{ falls } cc_j = 0,\ p_j = \frac{(|G'| - a) \cdot {\tau_j}^{\alpha} \cdot {\eta_j}^{\beta}}{\sum_{l \in G'} {\tau_l}^{\alpha} \cdot {\eta_l}^{\beta}} \text{ sonst} \tag{4.8}$$

4.3.3. Parameter

Interne Parameter des Algorithmus

ratio	Ziel-Seitenverhältnis der gesuchten OPSM, $ratio := k : s$

Eingabe-Parameter des Algorithmus

lowest_ratio	Startwert für *ratio*
highest_ratio	Endwert für *ratio*
ratio_factor	Faktor, um den *ratio* erhöht wird nach einer Iteration
alpha	Einfluss des Pheromons
beta	Einfluss der Heuristik
rho	Pheromonzerfallskoeffizient
tau0	Startwert für τ
max_iteration	Anzahl Ameisen-Iterationen
max_ant	Anzahl Ameisen

4.3.4. Pseudocode

```
create empty pareto model record
set ratio = lowest_ratio

do
{
   set Q = 1/(k*s), with k and s found by Frech Algorithm

   for each column i in dataset
      set tau_col_i = tau0

   for each row j in dataset
      set tau_row_j = tau0

   for each iteration from 1 to max_iteration
   {
      for each ant from 1 to max_ant
      {
         create model which contains all columns and rows of dataset

         do
         {
            determine average column order of columns in T by
               columnwisely summing up the ranks

            determine collision counts cc_col_i of all columns in T
               and cc_row_j of all rows in G

            if (k/s<ratio) // too many columns
            {
               if (s<m/4)
                  set a = max(1,s/16)
               else if (s<m/2)
                  set a = max(1,s/8)
               else
                  set a = max(1,s/4)

               for each column i in T
                  set eta_col_i = 1/cc_col_i
```

```
   set s' = #columns in T with cc_col_i > 0

   set f = (s'-a) / sum(tau_col_i^alpha*eta_col_i^beta,
      all columns i in T with cc_col_i>0)

   for each column i in T
   {
      if (cc_col_i=0)
         set p_col_i = 1
      else
         set p_col_i = f*tau_col_i^alpha*eta_col_i^beta

      remove column i from T with probability 1-p_col_i
   }

   add all rows to G which were removed before but are
      compatible to the new T
}
else // too many rows
{
   if (k<n/4)
      set a = max(1,k/16)
   else if (k<n/2)
      set a = max(1,k/8)
   else
      set a = max(1,k/4)

   for each row j in G
      set eta_row_j = 1/cc_row_j

   set k' = #rows in G with cc_row_j > 0
   set f = (k'-a) / sum(tau_row_j^alpha*eta_row_j^beta,
      all rows j in G with cc_row_j>0)

   for each row j in G
   {
      if (cc_row_j=0)
         set p_row_j = 1
      else
         set p_row_j = f*tau_row_j^alpha*eta_row_j^beta
```

```
                remove row j from the G with probability 1-p_row_j
             }
          }
       }
       while (model does not represent an OPSM)

       add all rows to G which are compatible to the final model
       add all columns to T which are compatible to the final model
       add the final model to the pareto model record

       for each column i in final model
          set delta_tau_col_i = delta_tau_col_i + Q*k*s,
             with k and s of final model

       for each row j in final model
          set delta_tau_row_j = delta_tau_row_j + Q*k*s,
             with k and s of final model
    }

    for each column i in dataset
       set tau_col_i = (1-rho)*tau_col_i + delta_tau_col_i

    for each row j in dataset
       set tau_row_j = (1-rho)*tau_row_j + delta_tau_row_j
 }

 set ratio = ratio * ratio_factor
}
while (ratio <= highest_ratio)

output content of pareto model record
```

4.4. Frech ACS Algorithmus

4.4.1. Idee

In [2] wird beschrieben, wie ein Ant Colony System (ACS) für die Lösung des Travelling Salesman Problem (TSP) verwendet wird. ACS ist eine Erweiterung von AS, die hauptsächlich darauf abzielt, die Exploration zu begünstigen.

Die dem Frech ACS Algorithmus zugrunde liegende Idee ist, das TSP-ACS für den Frech Algorithmus zu adaptieren. Dazu habe ich zunächst den Frech AS Algorithmus entwickelt, auf dem der Frech ACS Algorithmus aufbauen sollte.

Unglücklicherweise funktioniert die Art und Weise, wie das Pheromon bei Frech AS verwendet wurde, bei Frech ACS nicht so gut. Dies, weil bei ACS eine lokale Pheromonupdate-Regel eingeführt wird, die verlangt, dass die Pheromonspuren der ausgewählten Zeilen bzw. Spalten abgeschwächt werden. Die Absicht der Regel besteht darin, das Pheromon des gewählten Wegs abzuschwächen, damit die anderen Wege bessere Chancen haben, auch zum Zug zu kommen, und somit möglichst viele Wege exploriert werden. Da aber beim Frech Algorithmus in einem Iterationsschritt nicht diejenigen Zeilen und Spalten ausgewählt werden, die am Schluss die OPSM bilden, sondern diejenigen, die entfernt werden, würde die Absicht dieser Regel ins Gegenteil verkehrt. Zeilen und Spalten, die entfernt werden, weisen im Mittel bereits geringe Pheromonmengen auf und würden damit noch schwächer, als sie schon sind - d.h. die Regel hätte einen verstärkenden statt einen abschwächenden Effekt.

Ich habe deshalb die Konvention für das Pheromon geändert: Je mehr Pheromon auf einer Zeile bzw. Spalte liegt, desto grösser die Wahrscheinlichkeit, dass sie entfernt wird.

4.4.2. Adaption des TSP-ACS

Analogietabelle

AS für TSP	**ACS für Frech Algorithmus**
Kanten	Zeilen und Spalten
Knotenindizes i, j	Spaltenindizes i, Zeilenindizes j
Distanz d_{ij} (statisch)	Kollisionszahl cc_i, cc_j (dynamisch)
Sichtbarkeit $\eta_{ij} = 1/d_{ij}$ (statisch)	Inkompatibilität $\eta_i = cc_i$, $\eta_j = cc_j$ (dynamisch)
Transitionswahrscheinlichkeit p_{ij} (Gl. 4.1)	Entfernwahrscheinlichkeit p_i, p_j (Gl. 4.9 und 4.10)
Pheromonupdate $\Delta\tau_{ij}$	Pheromonupdate $\Delta\tau_i$, $\Delta\tau_j$

Entfernwahrscheinlichkeit

Die Herleitung der Entfernwahrscheinlichkeit funktioniert analog zur Herleitung der Haltewahrscheinlichkeit beim Frech AS Algorithmus (siehe Abschnitt 4.3.2). Der Parameter α entfällt jedoch, und die Änderung der Pheromonkonvention muss berücksichtigt werden. Damit erhält man für die Entfernwahrscheinlichkeit für Spalten:

$$p_i = \frac{(|T'| - a) \cdot \tau_i \cdot {cc_i}^{\beta}}{\sum_{l \in T'} {\tau_l}^{\alpha} \cdot {cc_l}^{\beta}} \tag{4.9}$$

Die Entfernwahrscheinlichkeit für Zeilen ist

$$p_j = \frac{(|G'| - a) \cdot \tau_j \cdot {cc_j}^{\beta}}{\sum_{l \in G'} {\tau_l}^{\alpha} \cdot {cc_l}^{\beta}} \tag{4.10}$$

4.4.3. Parameter

Interne Parameter des Algorithmus

ratio	Ziel-Seitenverhältnis der gesuchten OPSM, $ratio := k : s$

Eingabe-Parameter des Algorithmus

lowest_ratio	Startwert für *ratio*
highest_ratio	Endwert für *ratio*
ratio_factor	Faktor, um den *ratio* erhöht wird nach einer Iteration
beta	Einfluss der Heuristik
rho	Pheromonzerfallskoeffizient
q0	1 - Explorationswahrscheinlichkeit
tau0	Startwert für τ
max_iteration	Anzahl Ameisen-Iterationen
max_ant	Anzahl Ameisen

4.4.4. Pseudocode

```
create empty pareto model record
set ratio = lowest_ratio

do
{
   set tau0 = (k*s)/(n*m), with k and s found by Frech Algorithm
   set best_score = 0

   for each column i in dataset
      set tau_col_i = tau0

   for each row j in dataset
      set tau_row_j = tau0

   for each iteration from 1 to max_iteration
   {
      for each ant from 1 to max_ant
      {
         create model which contains all columns and rows of dataset

         do
         {
            determine average column order of columns in T by
               columnwisely summing up the ranks

            determine collision counts cc_col_i of all columns
               in T and cc_row_j of all rows in G

            if (k/s < ratio) // too many columns
            {
               if (s<m/4)
                  set a = max(1,s/16)
               else if (s<m/2)
                  set a = max(1,s/8)
               else
                  set a = max(1,s/4)

               set s' = #columns with cc_col_i > 0
```

```
      set f = (s'-a) / sum(tau_col_i*cc_col_i^beta,
         for all i with cc_col_i>0)

      for each column i in T
      {
         if (cc_col_i=0)
            set p_col_i = 0
         else
            set p_col_i = f*tau_col_i*cc_col_i^beta

         set q = random number in range [0,1]

         if (q<q0)
            remove from T the a columns with the highest
               probability p_col_i
         else
            remove column i from T with probability p_col_i

         for every column i which was removed
            set tau_col_i = (1-rho)*tau_col_i + rho*tau0
      }

      add all rows which are compatible to the order of the
         remaining columns in T
   }
   else // too many rows
   {
      if (k<n/4)
         set a = max(1,k/16)
      else if (k<n/2)
         set a = max(1,k/8)
      else
         set a = max(1,k/4)

      set k' = #rows with cc_row_j > 0

      set f = (k'-a) / sum(tau_row_j*cc_row_j^beta,
         for all j with cc_row_j>0)

      for each row j in G
```

```
            {
               if (cc_row_j=0)
                  set p_row_j = 0
               else
                  set p_row_j = f*tau_row_j*cc_row_j^beta

               set q = random number in range [0,1]


               if (q<q0)
                  remove from G the a rows with the highest
                     probability p_row_j
               else
                  remove row j from G with probability p_row_i

               for every row j which was removed
                  set tau_row_j = (1-rho)*tau_row_j + rho*tau0
            }
         }
      }
      while (model does not represent an OPSM)

      add all rows which are compatible to the final model


      add all columns which are compatible to the final model
      add the final model to the pareto model record
   }

   if a model was found with s*k>best_score
   {
      set best_score = s*k

      for each column i not contained in model with best score
         set tau_col_i = (1-rho)*tau_col_i + rho*s*k

      for each row j not contained in model with best score
         set tau_row_j = (1-rho)*tau_row_j + rho*s*k
   }
}
```

```
   set ratio = ratio * ratio_factor
}
while (ratio <= highest ratio)

output content of pareto model record
```

4.5. Ben-Dor Algorithmus

4.5.1. Idee

Die Idee des Ben-Dor Algorithmus (beschrieben in [1]) besteht darin, gezielt nach einem Model mit einer bestimmten Grösse s zu suchen, indem ein leeres Array mit s Plätzen bereitgestellt wird und dann von beiden Seiten vom Rand her kommend die Spaltenindizes eingefügt werden. Es wird also zunächst die Spalte mit dem niedrigsten Rang und die Spalte mit dem höchsten Rang ausgewählt. Dann werden die Spalten dazwischen so eingefügt, dass es immer nur eine Lücke gibt - nämlich zwischen den Spalten mit niedrigen Rängen und denjenigen mit hohen Rängen. Dabei werden abwechslungsweise Spalten mit niedrigem Rang und Spalten mit hohem Rang eingefügt. Dies solange, bis das Model vollständig ist.

Um zu entscheiden, ob eine Spalte an einer bestimmten Position ins Model eingefügt werden soll, wird die erwartete Anzahl der Zeilen berechnet, die mit dem vervollständigten Model kompatibel sein werden. Die Spalte mit der grössten erwarteten Anzahl kompatibler Zeilen wird dann eingefügt.

Da dem Algorithmus die Grösse s der implantierten OPSM nicht bekannt ist, müssen alle möglichen Werte für s (d.h. $s = 2...m$) ausprobiert werden.

Die Performance des Algorithmus wird noch dadurch gesteigert, dass nicht nur 1 Model zusammengebaut wird, sonderen deren l. In jedem Iterationsschritt werden dann alle möglichen Erweiterungen der l Models aus dem letzten Iterationsschritt getestet und die l besten davon für den nächsten Iterationsschritt verwendet.

4.5.2. Parameter

l	Anzahl Models, die von Iterationsschritt zu Iterationsschritt weitergereicht werden

4.5.3. Programmstart

Allg. Kommandozeilenaufruf

```
java opsmextraction.bendor.BendorMain <dataset filename> <l>
```

Beispiel

```
java opsmextraction.bendor.BendorMain data/dataset.txt 10
```

führt den Ben-Dor Algorithmus mit $l = 10$ auf der Expressionsmatrix in der Datei "data/dataset.txt" aus.

4.5.4. Laufzeitkomplexität

Es müssen $O(m)$ Werte für die Model-Grösse s ausprobiert werden. Für jeden Wert von s müssen l Models mit s Spalten erstellt werden, wobei $O(s) = O(m)$. Um ein Model mit einer Spalte zu erweitern, muss für $O(m)$ Spalten berechnet werden, wie gross die erwartete Anzahl kompatibler Zeilen sein wird. Für diese Berechnung müssen alle n Zeilen in Betracht gezogen werden. Die totale Laufzeitkomplexität beträgt damit $O(n \cdot m^3 \cdot l)$. Unter der Annahme, dass $O(n) = O(m)$ gilt, vereinfacht sich die Laufzeitkomplexität zu $O(n^4 \cdot l)$.

4.5.5. Hinweise zum Ben-Dor Paper

Die folgenden Hinweise auf Unklarheiten und Druckfehler in [1] könnten für den Leser hilfreich sein:

- Es werden 6 Schreibweisen für θ verwendet (zumindest interpretiere ich diese Symbole aus dem Kontext heraus als θ): θ, f?, 0, 6, 6', 8
- Im Abschnitt 5.4 sind im Ausdruck $g_i^\theta = D[i, t_{s-b+1}] - D[i, t_a] - 1$ die Minuszeichen vergessen gegangen.

Ausserdem enthält das Paper zwei weitere inhaltliche Unklarheiten:

- A_i ist die Wahrscheinlichkeit, die Rangfolge $D_\theta(i)$ zu beobachten, unter der Bedingung, dass θ das korrekte[2] partielle Model darstellt und Zeile i zur implantierten OPSM gehört. Meiner Meinung nach ist die Wahrscheinlichkeit gleich 0, eine Rangfolge $D_\theta(i)$ zu beobachten, die inkompatibel ist zu θ, wenn θ korrekt und Zeile i implantiert ist. Der von Amir Ben-Dor verwendete Ausdruck $A_i = Prob[D_\theta(i)|X_i = 1] = binom(g_i^\theta, s - (a + b))/binom(m, s)$ wird jedoch nur dann 0, wenn entweder $g_i^\theta < 0$ oder $g_i^\theta < s - (a + b)$ gilt. $g_i^\theta \geq s - (a + b) \geq 0$ ist zwar eine notwendige, jedoch keine hinreichende Bedingung dafür, dass Zeile i zum partiellen Model θ kompatibel ist. Und wenn eine Zeile zu einem partiellen Model inkompatibel ist, kann sie auch zu keinem daraus abgeleiteten vollständigen Model kompatibel sein. In meiner Implementierung setze ich deshalb $A_i = 0$ für den Fall, dass Zeile i inkompatibel ist zum partiellen Model θ.

[2] "Korrekt" heisst hier, dass das partielle Model θ zum implantierten vollständigen Model τ erweitert werden kann.

- Es wird nirgends gesagt, was mit einer Spalte passieren soll, für die die Gleichung $\sum_{i=1}^{n} A_i \cdot p/(A_i \cdot p + B \cdot (1-p)) = n \cdot p$ keine Lösung im Bereich $[0,1]$ hat für p. In meiner Implementierung gehe ich davon aus, dass eine solche Spalte nichts taugt, und setze daher $p = 0$.

Meine Anfrage an Amir Ben-Dor zu diesen beiden inhaltlichen Unklarheiten blieb leider unbeantwortet. Aufgrund der zu Ben-Dor's Angaben ähnlichen Performance meiner Implementierung ist zumindest davon auszugehen, dass meine Überlegungen nicht ganz abwegig sein können.

4.5.6. Probleme

U(s,k)-Problem

Die von Amir Ben-Dor verwendete Funktion $U(s,k)$ zur Abschätzung der statistischen Signifikanz einer OPSM ist, wie in Abschnitt 3.3 genauer erläutert, problematisch für die Beurteilung von grossen OPSM's. Während sich Amir Ben-Dor auf Problemgrössen beschränkt, für die $U(s,k)$ darstellbar ist mit 64-Bit Fliesskommadatentypen, entledige ich mich dieser Einschränkung durch die Ausgabe aller Pareto-optimalen OPSM's.

Newton-Problem

Meine Implementierung verwendet das Newton-Verfahren, um die Gleichung für p zu lösen. Diese lautet

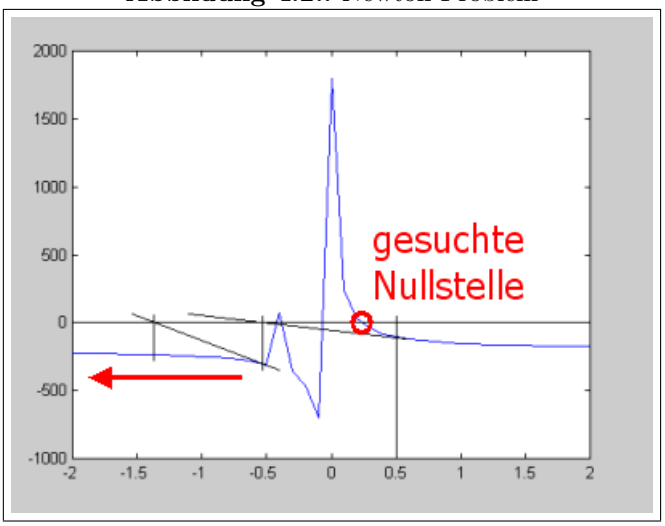
$$n - \sum_{i=1}^{n} \frac{A_i}{A_i + B_i \cdot (1-p)} = 0 \tag{4.11}$$

Diese Gleichung hat eine ungünstige Eigenschaft, die eine Anpassung des Newton-Verfahrens erfordert. Ohne diese Anpassung funktioniert der Ben-Dor Algorithmus nicht richtig, da dann ein Problem auftritt, das das Finden guter Models stark beinträchtigt. Figur 4.2 zeigt den Funktionsplot, aufgrund dessen ich das Problem entdeckt habe.

Gesucht ist eine Lösung für p im Bereich $[0,1]$. Ich wähle daher $p = 0.5$ als Startwert. In gewissen Fällen entschwindet die Iteration dann sofort in den negativen Bereich und findet nicht mehr zurück, obwohl eine Nullstelle im gesuchten Bereich existiert.

Ich habe das Newton-Verfahren deshalb dahingehend angepasst, dass ich, wenn p negativ würde nach einem Iterationsschritt, p stattdessen durch 2 teile. p wird so immer näher an die Nullstelle herangeführt, bis die Steigung der Kurve gross genug ist, dass die Iteration

Abbildung 4.2.: Newton-Problem



im positiven Bereich bleibt. Es werden aber maximal 20 Iterationsschritte ausgeführt. Entschwindet p im letzten davon, dann ist anzunehmen, dass keine Nullstelle im positiven Bereich existiert, und p wird auf Null gesetzt, was einer Disqualifikation der betreffenden Spalte gleichkommt. Gilt $p > 1$ am Ende der Iteration, wird p ebenfalls auf Null gesetzt.

4.5.7. Ben-Dor Reloaded

Die logische Funktion des Ben-Dor Reloaded Algorithmus ist exakt die gleiche wie diejenige des Ben-Dor Algorithmus. Der einzige Unterschied besteht in der Effizienz der Implementation: Abbildung 5.25 in Abschnitt 5.3.2 zeigt, dass Ben-Dor Reloaded bis zu dreimal schneller läuft als Ben-Dor.

4.5.8. Ben-Dor Random

Um in Abschnitt 5.6 zu zeigen, weshalb dem Ben-Dor ACS Algorithmus aus Abschnitt 4.6 kein Erfolg beschieden ist, habe ich eine randomisierte Version des Ben-Dor Algorithmus implementiert. Der Ben-Dor Random Algorithmus wählt die hinzuzufügenden Spalten zufällig aus, wobei die Wahrscheinlichkeit für eine Spalte, verwendet zu werden, proportional ist zur Wahrscheinlichkeit p.

4.6. Ben-Dor ACS Algorithmus

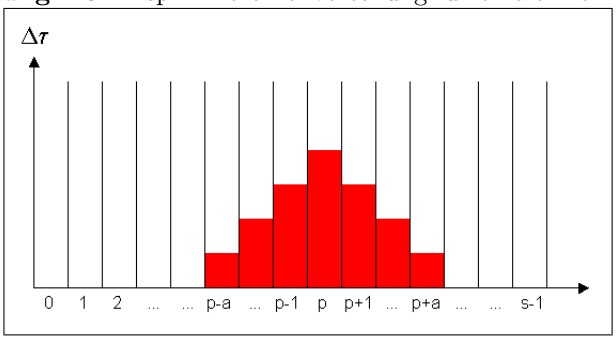
4.6.1. Idee

In [2] wird beschrieben, wie ein Ant Colony System (ACS) für die Lösung des Travelling Salesman Problem (TSP) verwendet wird. Die dem Ben-Dor ACS Algorithmus zugrunde liegende Idee ist, das TSP-AS für den Ben-Dor Algorithmus zu adaptieren. Die Adaption beruht darauf, das Pheromon auf die Spalten zu legen, wobei jeder Spalte nicht nur ein einziger Pheromonwert zugeordnet wird, sondern ein Pheromonwert pro mögliche Modelposition. Denn eine Spalte, die für niedrige Ränge im Model geeignet ist, ist nicht unbedingt auch für hohe Ränge geeignet. Je mehr Pheromon eine Spalte an der Position aufweist, an der das Model gerade erweitert wird, desto grösser die Wahrscheinlichkeit, dass sie verwendet wird.

4.6.2. Adaption des TSP-ACS

Abbildung 4.3 zeigt das Pheromonupdate für eine Spalte, die an Position p im Model verwendet wurde. Der Pheromonwert an Position p ist am höchsten, weil die Spalte für diese Position verwendet wurde. Aber auch für die benachbarten Positionen sollte sich die Spalte eignen, deshalb führe ich ein unscharfes Pheromonupdate durch. Die Breite des "Pheromonberges" ist proportional zu gesamten Modelgrösse.

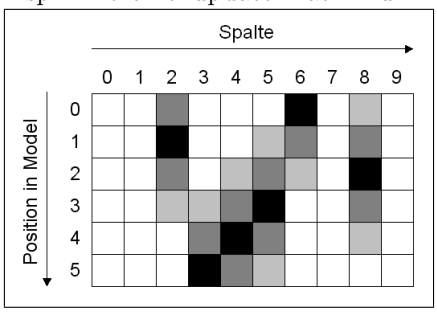
Abbildung 4.3.: Bsp.: Pheromonverteilung für eine einzelne Spalte



Das Pheromonupdate für ein gesamtes Model sieht so aus wie in Abbildung 4.4 gezeigt. Für jede Spalte wird ein "Pheromonberg" an die Position gesetzt, an der die Spalte im Model verwendet wurde.

Wenn in Iterationsschritt i die Modelposition p_i besetzt werden muss, so werden für alle Spalten die Pheromonwerte an Position p_i konsultiert.

Abbildung 4.4.: Bsp.: Pheromonupdate-Matrix für Model $[6, 2, 8, 5, 4, 3]$



Der Rest des Ben-Dor ACS Algorithmus funktioniert analog dem TSP-ACS. Mit Ausnahme einer Anpassung, die ich gemacht habe, weil zu selten ein besseres Model gefunden wurde als alle vorangehenden: Die Pheromonwerte werden mit jedem fertiggestellten Model aktualisiert.

4.6.3. Parameter

l	Anzahl Models, die von Iterationsschritt zu Iterationsschritt weitergereicht werden
beta	Einfluss der Heuristik
rho	Pheromonzerfallskoeffizient
q0	1 - Explorationswahrscheinlichkeit
tau0	Startwert für τ
max_iteration	Anzahl Ameisen-Iterationen
max_ant	Anzahl Ameisen

4.7. Ben-Dor Plus Algorithmus

4.7.1. Idee

Als ich Tests mit dem Ben-Dor Algorithmus durchführte, stellte ich fest, dass dieser beim Finden von grossen implantierten OPSM's scheitert. Eine genauere Analyse ergab, dass beim iterativen Einfügen von Spalten in das partielle Model einige implantierte Spalten übersprungen werden, wenn es viele implantierte Spalten gibt. Eine einmal übersprungene Spalte ist dann für den Rest der Iteration verloren. Die Menge der kompatiblen Zeilen wird jedoch meist nicht grösser durch das Überspringen von Spalten, da die Wahrscheinlichkeit, dass eine nichtimplantierte Zeile zufällig kompatibel ist zum lückenhaften Model, bei grossen Models sehr klein ist.

Die Idee des Ben-Dor Plus Algorithmus ist daher: Man nehme die vom Ben-Dor Algorithmus gefundenen Models und versuche bei jedem dieser Model, jede im Model noch nicht verwendete Spalte an jeder Zwischenposition im Model einzufügen, so dass die Menge der zum Model kompatiblen Zeilen sich nicht verändert. Gelingt dies, wird versucht, eine weitere Spalte einzufügen, usw., bis keine Spalten mehr eingefügt werden können.

Während der Ben-Dor Algorithmus also oft die gesamte Menge der implantierten Zeilen, jedoch nur eine Teilmenge der implantierten Spalten findet, kann Ben-Dor Plus die entstandenen Lücken füllen und somit die gesamte OPSM finden. Die benötigte Laufzeit, um zu einer fest gegebenen Menge von Zeilen Erweiterungen eines Models zu finden, die keine der gegebenen Zeilen inkompatibel werden lässt, ist gering im Vergleich zur Laufzeit des Ben-Dor Algorithmus (siehe Abschnitt 4.7.6). Und da der erweiterte Algorithmus im besten Fall bessere und im schlechtesten Fall gleich gute Ergebnisse liefert wie der ursprüngliche Algorithmus, hat die Erweiterung nur Vorteile.

4.7.2. Parameter

Die Erweiterung benötigt keine zusätzlichen Parameter. Ben-Dor Plus benötigt daher denselben Parameter wie Ben-Dor.

l	Anzahl Models, die von Iterationsschritt zu Iterationsschritt weitergereicht werden

4.7.3. Programmstart

Allg. Kommandozeilenaufruf

```
java opsmextraction.bendor_plus.BendorPlusMain <dataset filename> <l>
```

Beispiel

```
java opsmextraction.bendor_plus.BendorPlusMain data/dataset.txt 10
```

führt den Ben-Dor Plus Algorithmus mit $l = 10$ auf der Expressionsmatrix in der Datei "data/dataset.txt" aus.

4.7.4. Pseudocode

```
R := set of models found by Ben-Dor Algorithm (1 model for each model size s)

for each model T in R
{
   G := set of rows compatible to T
   s := #columns in T
   successful := true

   while (successful)
   {
      successful := false


      loop1: for each column c not used in T
      {
         for i = 0 to s
         {
            T' := copy of T
            insert column c into T' between index i-1 and i
            G' := set of rows compatible to T'

            if (G==G')
            {
               add T' to R
               successful := true
               break loop1
            }
         }
      }
   }
```

```
}

remove from R all non-pareto-optimal models
output R
```

4.7.5. Laufzeitkomplexität

Die Laufzeitkomplexität des Ben-Dor Algorithmus, der im ersten Teil des Algorithmus ausgeführt wird, beträgt $O(n \cdot m^3 \cdot l)$ (siehe Abschnitt 4.5.4).

Im zweiten Teil wird versucht, alle von Ben-Dor gefundenen OPSM's auszubauen. Da Ben-Dor für jede Model-Grösse s ausgeführt wird, ist mit $O(m)$ zu bearbeitenden OPSM's zu rechnen. Eine OPSM auszubauen heisst im schlimmsten Fall, in $O(m)$ Iterationen $O(m)$ Spalten mit $O(m)$ verschiedenen Rängen einzufügen versuchen. Dazu muss jeweils für $O(n)$ Zeilen geprüft werden, ob sie kompatibel bleiben. Die Laufzeitkomplexität des zweiten Teils beträgt damit $O(n \cdot m^4)$.

Die totale Laufzeitkomplexität ergibt sich aus der Summe der Laufzeitkomplexität der beiden Teile. Sie beträgt $O(n \cdot m^3 \cdot (l + m))$. Es kann für den allgemeinen Fall nicht gesagt werden, welcher Teil dominant ist bezüglich Laufzeitkomplexität. Tatsache ist jedoch, dass die pessimistischen Annahmen für die Berechnung der Laufzeitkomplexität des zweiten Teils in der Realität kaum eintreffen, bzw. die Zeitkonstante ist für den ersten Teil viel grösser ist als diejenige für den zweiten Teil. Die Beispiele in Abschnitt 4.7.6 illustrieren dies.

4.7.6. Beispiele zur Illustration

Erfolg

Als Beispiel für eine erfolgreiche Anwendung von Ben-Dor Plus soll eine 100×100[3] Matrix mit einer implantierten 20×50[4] OPSM dienen. Die folgende Tabelle demonstriert eindrücklich, wie gross der Gewinn der Erweiterung sein kann - und das ohne die Laufzeit merklich zu erhöhen. In der Rubrik "gefundene OPSM" habe ich jeweils diejenige pareto-optimale OPSM eingetragen, die der implantierten am nächsten kommt. Es ist in allen Fällen zugleich die statistisch Signifikanteste.

[3] $n \times m$
[4] $k \times s$

Abbildung 4.5.: Erfolg von Ben-Dor Plus

Algorithmus	Parameter l	gefundene OPSM	Laufzeit
Ben-Dor	1	20×25	20s
Ben-Dor	10	20×36	43s
Ben-Dor	100	20×39	234s
Ben-Dor Plus	1	20×50	20s

Die hier gezeigten OPSM's, die von Ben-Dor gefunden werden, enthalten jeweils alle implantierten Zeilen und ausschliesslich implantierte Spalten. Dies ist die Grundlage des Erfolgs von Ben-Dor Plus, der schon mit $l = 1$ die implantierte OPSM vollständig findet.

Die Dateien zu diesem Beispiel befinden sich auf der CD im Verzeichnis "OPSMExtraction/examples/bendor_plus/success/".

Misserfolg

Dass Ben-Dor Plus nicht immer erfolgreich sein kann, zeigt folgendes Beispiel mit einer 100×100 Matrix mit einer implantierten 8×20 OPSM. Dabei bezeichnet k_p (s_p) die Anzahl implantierter Zeilen (Spalten) und k_{np} (s_{np}) die Anzahl nichtimplantierte Zeilen (Spalten), die zur gefundenen OPSM gehören. Natürlich gilt immer $k = k_p + k_{np}$ und $s = s_p + s_{np}$.

Einige[5] pareto-optimale Models, gefunden von Ben-Dor mit $l = 1$:

Abbildung 4.6.: Ben-Dor mit $l = 1$

$k \times s$	k_p	k_{np}	s_p	s_{np}
2×12	2	0	8	4
6×9	6	0	8	1
21×7	20	1	7	0
23×5	20	3	5	0

Einige pareto-optimale Models, gefunden von Ben-Dor Plus mit $l = 1$:

Wie zu sehen ist, konnte nur das Model der 2×12 OPSM, die ohnehin nicht statistisch signifikant ist, leicht verbessert werden. Die implantierte OPSM hingegen wurde nicht gefunden. Das liegt daran, dass die Models der 2×12 OPSM und der 6×9 OPSM nichtimplantierte Spalten enthalten. Dadurch sind nicht alle implantierten Zeilen kompatibel zu diesen beiden Models, und sie wurden auch nicht durch nichtimplantierte Zeilen ersetzt. Die Models der 21×7 OPSM und der 23×5 OPSM enthalten zwar nur

[5] Diejenigen Models, die nichts zur Erhellung des Sachverhalts beitragen, habe ich weggelassen.

Abbildung 4.7.: Ben-Dor Plus mit $l = 1$

$k \times s$	k_p	k_{np}	s_p	s_{np}
2×16	2	0	8	8
6×9	6	0	8	1
21×7	20	1	7	0
23×5	20	3	5	0

implantierte Spalten, dafür aber nichtimplantierte Zeilen, die verhindern, dass die restlichen implantierten Spalten eingefügt werden können, ohne Zeilen zu verlieren. Daran scheitert Ben-Dor Plus.

Ben-Dor mit $l = 10$ hingegen findet die implantierte OPSM. Das "Plus" in Ben-Dor Plus ist in diesem Fall nutzlos. Immerhin ist auch hier die Laufzeit von Ben-Dor mit derjenigen von Ben-Dor Plus im Rahmen der Messgenauigkeit identisch, sie betrug in beiden Fällen 10 Sekunden. Geschadet hat die Erweiterung also nicht bezüglich Laufzeitverhalten.

Die Dateien zu diesem Beispiel befinden sich auf der CD im Verzeichnis "OPSMExtraction/examples/bendor_plus/failure/".

4.7.7. Bemerkung

Der Ben-Dor Plus Algorithmus ist ein Zwischenschritt auf dem Weg zum Ben-Dor-Frech Algorithmus, der im nächsten Abschnitt beschrieben wird.

4.8. Ben-Dor-Frech Algorithmus

4.8.1. Idee

Die Idee des Ben-Dor-Frech Algorithmus ist im Grunde die selbe wie beim Ben-Dor Plus Algorithmus: die vom Ben-Dor Algorithmus gelieferten Ergebnisse sollen verbessert werden. Während dies beim Ben-Dor Plus Algorithmus aber auf eine recht einfache Art und Weise geschieht, verwendet der Ben-Dor-Frech Algorithmus eine ausgefeiltere Methode.

Um in Fällen, in denen Ben-Dor Plus scheitert (siehe Abschnitt 4.7.6), Erfolg zu haben, ist Ben-Dor-Frech flexibler: es werden zunächst Zeilen und Spalten entfernt, um den Rest dann zu einer grösseren OPSM anwachsen zu lassen. Dazu wird die Idee des Frech Algorithmus verwendet. Der genaue Mechanismus wird in den folgenden Abschnitten erläutert.

4.8.2. Begriffsdefinitionen

Die folgenden Begriffe stehen immer in Bezug zu einem Model T, von dem angenommen wird, dass es viele implantierte Spalten enthält und viele der implantierten Zeilen dazu kompatibel sind.

schlechte Spalte

Eine Spalte im Model T, die nicht implantiert ist und damit verhindert, dass alle implantierten Zeilen kompatibel sind zu T, bezeichne ich als "schlechte Spalte".

gute Spalte

Eine implantierte Spalte, die sich nicht im Model T befindet, bezeichne ich als "gute Spalte".

schlechte Zeile

Eine nichtimplantierte Zeile, die kompatibel ist zum Model T und damit den Ben-Dor Plus Algorithmus scheitern lässt, bezeichne ich als "schlechte Zeile".

gute Zeile

Eine implantierte Zeile, die nicht kompatibel ist zum Model T, weil dieses nichtimplantierte Spalten enthält, bezeichne ich als "gute Zeile".

4.8.3. Beschreibung

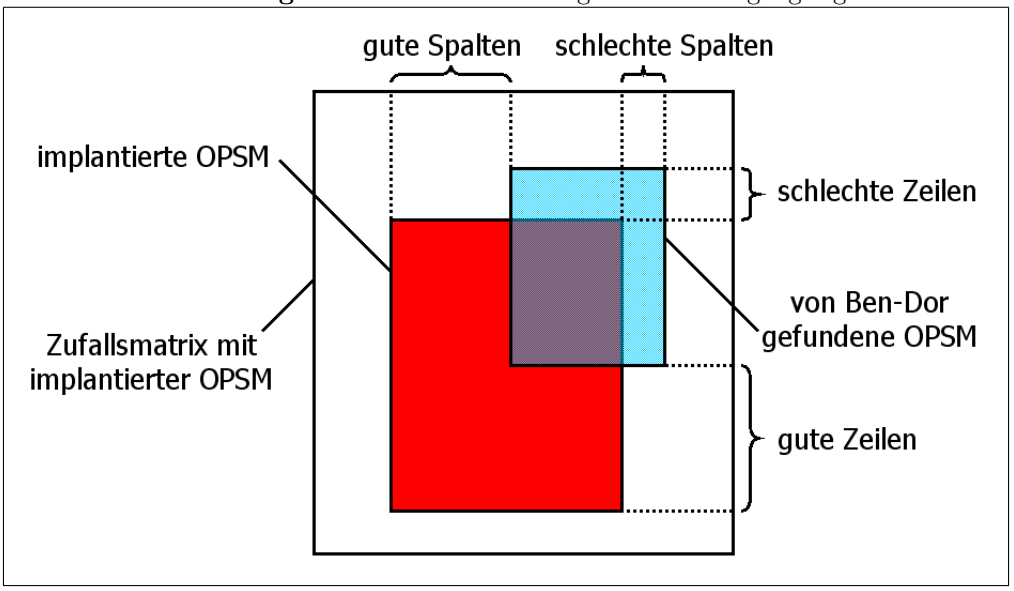
Ziel

Mit den in Abschnitt 4.8.2 gemachten Begriffsdefinitionen ist schnell erklärt, was das Ziel des Ben-Dor-Frech Algorithmus ist: aus einer vom Ben-Dor Algorithmus erhaltenen OPSM alle schlechte Zeilen und Spalten entfernen und dafür alle guten Zeilen und Spalten hinzufügen. Die ganze weitere Diskussion dreht sich nur um die Frage, wie erkannt werden kann, ob eine Zeile oder Spalte gut oder schlecht ist.

Ausgangslage

Die Ausgangslage präsentiert sich so: Es wird angenommen, dass Ben-Dor eine OPSM mit dem zugrunde liegenden Model T_{parent} und der dazu kompatiblen Zeilenmenge G_{parent} liefert, die sich teilweise mit der implantierten OPSM überschneidet, jedoch kleiner ist als diese. Abbildung 4.8 veranschaulicht dies.

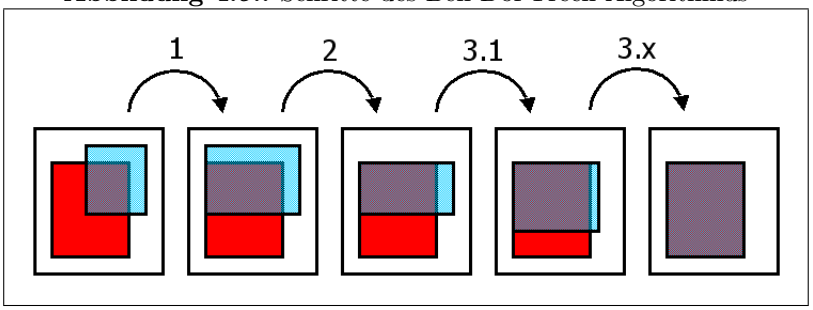
Abbildung 4.8.: Ben-Dor-Frech Algorithmus Ausgangslage



Auf der OPSM O_{parent} führt Ben-Dor-Frech 3 Schritte aus (siehe auch Abbildung 4.9):

1. alle guten Spalten hinzufügen
2. alle schlechten Zeilen entfernen
3. solange iterativ eine schlechte Spalte entfernen und kompatibel gewordene gute Zeilen hinzufügen, bis alle schlechten Spalten entfernt und alle guten Zeilen hinzugefügt wurden.

Abbildung 4.9.: Schritte des Ben-Dor-Frech Algorithmus



Schritt 1

Bereits im ersten Schritt kommt die Idee des Frech Algorithmus (siehe Abschnitt 4.2) zum Einsatz: Es wird die mittlere Rangfolge aller m Spalten der Expressionsmatrix bestimmt, wobei für die Bestimmung der Rangfolge nur die Zeilen in G_{parent} betrachtet werden. Unter der Annahme, dass nicht allzu viele Zeilen in G_{parent} schlecht sind[6], wird die Rangfolge des implantierten Models in der Rangfolge aller Spalten enthalten sein. Die mittlere Rangfolge aller Spalten ausser denjenigen in T_{parent} bildet das Model $T_{average}$. $T_{average}$ enthält also alle guten und keine schlechten Spalten.

Für jede Spalte in $T_{average}$ wird gezählt, wieviele Kollisionen mit den k_{parent} Zeilen in G_{parent} verursacht werden. Diejenigen Spalten, die weniger als $mrcc \cdot k_{parent}$ Kollisionen verursachen, werden als gute Spalten eingestuft. Alle guten Spalten werden in das Model T_{parent} eingefügt und bilden damit das Model T_{child_1}.

Für $mrcc$ (maximum relative collision count) wird der Wert 0.2 verwendet.

[6]Die Erfahrungen mit dem Frech Algorithmus lassen vermuten, dass ein Anteil von 10-20 Prozent gute Zeilen bereits ausreicht, um dem implantierten Model genügend Gewicht zu verleihen.

Schritt 2

Alle Zeilen, die nicht kompatibel sind zu T_{child_1}, werden aus G_{parent} entfernt. Übrig bleibt die zu T_{child_1} kompatible Zeilenmenge G_{child_1}.

Schritt 3

Im letzten Schritt werden iterativ solange einzelne Spalten aus dem Model entfernt, wie ein markanter Zuwachs der kompatiblen Zeilenmenge erreicht werden kann.[7] Als "markant" wird eine Vergrösserung der Zeilenmenge um mehr als den Faktor mgf betrachtet. Für die Entscheidung, welche Spalte in einem Iterationsschritt entfernt werden soll, wird wiederum die Idee des Frech Algorithmus verwendet: Es wird für jede Spalte in T_{child_i} gezählt, wieviele Kollsionen mit sämtlichen n Zeilen der Expressionsmatrix verursacht werden. Diejenige Spalte, die am meisten Kollisionen verursacht, wird aus T_{child_i} entfernt, womit das verkleinerte Model $T_{child_{i+1}}$ resultiert. Anschliessend wird G_{child_i} erweitert zu $G_{child_{i+1}}$, indem alle zu $T_{child_{i+1}}$ kompatiblen Zeilen hinzugefügt werden.

Für mgf (minimum grow factor) wird der Wert 1.2 verwendet.

Zusammenspiel von Ben-Dor und Frech

Prinzipiell könnte zunächst der Ben-Dor Algorithmus vollständig ausgeführt werden und danach alle erhaltenen Pareto-optimalen OPSM's bearbeitet werden. Dies birgt jedoch die Gefahr, dass diejenige OPSM, aus der die implantierte OPSM rekonstruiert werden könnte, von einer anderen OPSM, aus der die implantierte OPSM nicht rekonstruiert werden kann, dominiert wird und deshalb nicht in der Pareto-Menge enthalten ist.

Der Ben-Dor Algorithmus wird daher für jede Model-Grösse s separat laufen gelassen. Bei jedem Lauf fallen mehrere OPSM's an: die Original-OPSM des Ben-Dor Algorithmus, die nachbearbeitete Ben-Dor-Frech Version davon, sowie die Zwischenprodukte aus der Nachbearbeitung. Aus der Menge der OPSM's aller Läufe werden am Schluss die Pareto-optimalen ausgegeben.

4.8.4. Parameter

Die beiden Parameter der Erweiterung von Ben-Dor, $mrcc$ und mgf, sind nicht dazu gedacht, vom Anwender des Algorithmus variiert zu werden. Sie sollten auf Grund von

[7] Wobei ein Model mit nur noch 2 Spalten natürlich nicht mehr weiter verkleinert wird, weil daraus ein triviales Model resultieren würde, zu dem alle Zeilen kompatibel sind.

systematischen Messungen mit grossen Datenmengen optimal eingestellt werden. Dazu fehlte mir leider die Zeit.

Ben-Dor-Frech benötigt daher als einzigen Parameter denjenigen des zugrundeliegenden Ben-Dor Algorithmus.

l	Anzahl Models, die von Iterationsschritt zu Iterationsschritt weitergereicht werden

4.8.5. Programmstart

Allg. Kommandozeilenaufruf

```
java opsmextraction.frech.BendorFrechMain <dataset filename> <l>
```

Beispiel

```
java opsmextraction.frech.BendorFrechMain data/dataset.txt 10
```

führt den Ben-Dor-Frech Algorithmus mit $l = 10$ auf der Expressionsmatrix in der Datei "data/dataset.txt" aus.

4.8.6. Pseudocode

```
D := ranked dataset
R := set of models found by Ben-Dor Algorithm (1 model for each model size s)

for each model T_parent in R
{
   G_parent := set of rows compatible to T_parent
   k_parent := #rows in G_parent

   // step 1

   for each column c in D which is not in T_parent
      col_sum[c] := sum(D[r,c], r in G_parent)

   T_average := model induced by vector col_sum
```

```
   for each column c in T_average
      cc[c] := #collisions caused by c and rows in G_parent

   T_child[1] := T_parent extended with columns c in T_average
      with cc[c]<mrcc*k_parent

   add T_child[1] to R

   // step 2

   G_child[1] := set of rows compatible to T_child[1]
   k_child[1] := #rows in G_child[1]

   // step 3

   i := 0

   do
   {
      i := i + 1

      for each column c in T_child[i]
         cc[c] := #collisions caused by c and rows in D

      c_worst := column with maximum cc[c]
      T_child[i+1] := T_child[i] without column c_worst
      add T_child[i+1] to R
      G_child[i+1] := set of rows compatible to T_child[i+1]
      k_child[i+1] := #rows in G_child[i+1]
   }
   while (k_child[i+1]>mgf*k_child[i])
}

remove from R all non-pareto-optimal models
output R
```

4.8.7. Laufzeitkomplexität

Die Laufzeitkomplexität des Ben-Dor Algorithmus, der im ersten Teil des Algorithmus ausgeführt wird, beträgt $O(n \cdot m^3 \cdot l)$ (siehe Abschnitt 4.5.4).

Im zweiten Teil wird versucht, alle von Ben-Dor gefundenen OPSM's auszubauen. Da Ben-Dor für jede Model-Grösse s ausgeführt wird, ist mit $O(m)$ zu bearbeitenden OPSM's zu rechnen.

Die aufwendigste Arbeit in Schritt 1 ist das Zählen der Kollisionen. Die Summierung läuft über $O(n)$ Zeilen und $O(m^2)$ Spaltenpaare, woraus ein Aufwand von $O(n \cdot m^2)$ resultiert.

Schritt 2 erfordert ebenfalls einen Aufwand von $O(n \cdot m^2)$, da für $O(n)$ Zeilen und $O(m^2)$ Spaltenpaare die Kompatibilität getestet werden muss.

In Schritt 3 werden im schlimmsten Fall $O(m)$ Iterationen ausgeführt. In jeder Iteration werden Kollsionen gezählt, was einen Aufwand von $O(n \cdot m^2)$ erfordert. Der Gesamtaufwand für Schritt 3 beträgt damit $O(n \cdot m^3)$.

Bezüglich Laufzeitkomplexität dominiert Schritt 3 damit die anderen beiden Schritte. Der Gesamtaufwand für den zweiten Teil beträgt damit $O(n \cdot m^4)$.

Die totale Laufzeitkomplexität ist damit die selbe wie bei Ben-Dor Plus: $O(n \cdot m^3 \cdot (l + m))$. Und wie bei Ben-Dor Plus kann auch bei Ben-Dor-Frech festgestellt werden, dass die Laufzeit von Ben-Dor trotz unvorteilhaft erscheinender Laufzeitkomplexität nicht merklich vergrössert wird und die Erweiterung damit nur Vorteile bringt.

4.8.8. Beispiele zur Illustration

Als Beispiel für eine erfolgreiche Anwendung von Ben-Dor-Frech dient dasjenige aus Abschnitt 4.7.6, das zum Misserfolg von Ben-Dor Plus führte. Ausgehend vom Ergebnis von Ben-Dor mit $l = 1$ findet Ben-Dor-Frech die implantierte OPSM. Zwar findet Ben-Dor mit $l = 10$ die implantierte OPSM ebenfalls, benötigt aber mehr Laufzeit dazu.

Die Dateien zu diesem Beispiel befinden sich auf der CD im Verzeichnis "OPSMExtraction/examples/bendor_frech/success/".

4.8.9. Ben-Dor-Frech Reloaded

Der einzige Unterschied zwischen Ben-Dor-Frech und Ben-Dor-Frech Reloaded besteht darin, dass Ben-Dor-Frech den Ben-Dor Algorithmus und Ben-Dor-Frech Reloaded den Ben-Dor Reloaded Algorithmus verwendet. Der gewonnene Laufzeitvorteil ist ersichtlich in Abbildung 5.25 in Abschnitt 5.3.2.

4.9. Implementation

Alle Algorithmen wurden mit Hilfe von Borland JBuilder 9 in Java implementiert. Der Sourcecode befindet sich auf der beigelegten CD-ROM.

4.9.1. Paketstruktur und Klassen

Das Projekt "OPSMExtraction" enthält die folgenden Pakete:

opsmextraction.util	Allgemeine Klassen, die von allen OPSM Algorithmen verwendet werden können.
opsmextraction.gui	Klassen, die ein graphisches Userinteface zur Verfügung stellen. (z.Z. nur Pheromonvisualisierungen)
opsmextraction.frech	Frech Algorithmus
opsmextraction.frech_as	Frech AS Algorithmus
opsmextraction.frech_acs	Frech ACS Algorithmus
opsmextraction.frech_random	Frech Random Algorithmus
opsmextraction.bendor	Ben-Dor Algorithmus
opsmextraction.bendor_reloaded	Ben-Dor Reloaded Algorithmus
opsmextraction.bendor_acs	Ben-Dor ACS Algorithmus
opsmextraction.bendor_random	Ben-Dor Random Algorithmus
opsmextraction.bendor_plus	Ben-Dor Plus Algorithmus
opsmextraction.bendor_frech	Ben-Dor-Frech Algorithmus
opsmextraction.bendor_frech_reloaded	Ben-Dor-Frech Reloaded Algorithmus

Das Paket "opsmextraction.util" enthält folgende Klassen:

Dataset	Repräsentiert eine Genexpressionsmatrix. Enthält Funktionen, um Matrizen aus Dateien zu laden oder Zufallsmatrizen mit implantierter OPSM zu generieren und zu speichern.
Model	Enthält ein Model und die dazu kompatiblen Zeilen und repräsentiert damit eine OPSM. Dient als Superklasse für Algorithmus-spezifische Model-Klassen, in denen ein grosser Teil der Algorithmus-Funktionalität implementiert wird.
ParetoModelRecord	Enthält Funktionen, um Pareto-optimale OPSM's - repräsentiert durch Instanzen der Model-Klasse - zu speichern.
LinkListElement	Element einer bidirektionalen verketteten Liste aus Instanzen der Model-Klasse.
XMath	Stellt mathematische Funktionen zur Verfügung, die in "java.lang.Math" nicht vorhanden sind.
Main	Superklasse für Algorithmus-spezifische Main-Klassen, die für den Start eines Algorithmus aus der Kommandoeingabe benötigt werden. Enthält unter anderem Funktionalität zur Erstellung von Berichten über die Ergebnisse. Subklassen müssen nur wenige Methoden überschreiben.
XPrintStream	Enthält Funktionalität, um gleichzeitig auf den Bildschirm und in eine Datei zu schreiben.
Option	Optionsfelder, auf die nur ein Programmierer zugreifen muss während der Entwicklung von Algorithmen, wurden in diese Klasse ausgelagert, um die Liste der Kommandozeilenparameter kurz zu halten.

Ein Algorithmus-Paket enthält immer mindestens folgende 2 Klassen:

<**Name**>**Main**	Subklasse von "opsmextraction.util.Main" zum Start des Algorithmus aus der Kommandoeingabe.
<**Name**>**Algorithm**	Oberste Ebene der Algorithmus-Implementation.

Zusätzlich implementieren die meisten Algorithmen in der Model-Subklasse <Name>Model einen grossen Teil der Algorithmus-spezifischen Funktionalität. Einige Algorithmen enthalten noch weitere Klassen für Algorithmus-spezifische Funktionalität.

Die Model-Klassen der Basis-Algorithmen "Frech", "Ben-Dor" und "Ben-Dor Reloaded" werden von den darauf aufbauenden Algorithmen verwendet. Diese Abhängigkeit ist zu berücksichtigen, wenn ein Teil der Klassen in ein grösseres Projekt integriert wird.

4.9.2. Programmstart

Ein Algorithmus wird mit folgender Kommandozeilen-Eingabe gestartet:

```
java opsmextraction.<package>.<name>Main <parameter list>
```

Falls ungültige (oder keine) Parameter eingegeben werden, erscheint ein kurzer Hilfetext zu den Parametern.

Beispiele zu konkreten Kommandozeilen-Eingaben befinden sich in den Abschnitten über die Algorithmen.

4.9.3. Dateiformate

Datenset

Ein Datenset besteht aus einer eindimensionalen Liste von Einträgen, die durch Leerzeichen, Tabulatoren oder Zeilenumbrüche getrennt sind.

Der erste Eintrag muss "probeset" oder "randset" lauten. "probeset" kennzeichnet reale Daten, "randset" zufällig generierte Daten. Die Algorithmen werden durch den ersten Eintrag nicht beeinflusst.

Die nächsten m Einträge sind die Namen der Spalten. Sie dürfen keine gültigen double-Werte annehmen.

Anschliessend müssen für jede Zeile $n + 1$ Einträge vorhanden sein. Der erste davon ist der Zeilenname, der keinen gültigen double-Wert annehmen darf, danach folgen die Genexpressionswerte, die gültige double-Werte darstellen müssen.

Beispiele für gültige double-Werte: ”1234”, ”-12.34”, ”1.2e34”, ”-1.2E-34”.

Ein Datenset, das als Excel-Tabelle vorliegt und im Textformat gespeichert wird, entspricht dieser Norm, sofern die übrigen Konventionen für die Einträge eingehalten werden.

Model

Ein Model besteht aus einer eindimensionalen Liste von Einträgen, die durch Leerzeichen, Tabulatoren oder Zeilenumbrüche getrennt sind.

Der erste Eintrag ist s.

Der zweite Eintrag ist k.

Danach folgen die Indizes der s Spalten des Models.

Danach folgen die Indizes der k zum Model kompatiblen Zeilen.

Die von meiner Implementation erstellten Model-Dateien halten sich zudem an die Konvention, auf der ersten Zeile s anzugeben, auf der zweiten Zeile k, auf der dritten Zeile sämtliche Spalten, jeweils durch 1 Leerzeichen getrennt, und auf der vierten Zeile sämtliche Zeilen, jeweils durch 1 Leerzeichen getrennt. Abschliessend wird noch 1 Zeilenumbruch angefügt.

5. Ergebnisse

5.1. Übersicht

In diesem Kapitel vergleiche ich zunächst die Algorithmen "Frech", "Ben-Dor", "Ben-Dor Reloaded", "Ben-Dor-Frech" und "Ben-Dor-Frech Reloaded" in Bezug auf Ergebnis-Qualität und Laufzeitaufwand. Dazu habe ich Messungen auf grossen Datenmengen ausgeführt. Die anderen Algorithmen habe ich bei diesen Vergleichen aus folgenden Gründen weggelassen: "Ben-Dor Plus", weil er nur ein Zwischenschritt auf dem Weg zur Entwicklung von "Ben-Dor-Frech" darstellt, die Ameisen Algorithmen, weil sie nicht gut funktionieren und erst noch enorm viel Laufzeit benötigt hätten für diese Messungen (mehr als einen Monat auf meinem PC), "Frech Random" und "Ben-Dor Random", weil diese nur dazu da sind, Probleme des Ameisen-Ansatzes aufzuzeigen.

Danach präsentiere ich die Ergebnisse der Messungen, die ich unter den von Amir Ben-Dor beschriebenen Bedingungen durchgeführt habe, und vergleiche sie mit seinen Ergebnissen.

Dann zeige ich, wie sich die Algorithmen "Frech", "Ben-Dor" und "Ben-Dor-Frech" mit realen Expressionsmatrizen zurechtfinden, wobei diese Ergebnisse mit Vorsicht zu geniessen sind, da sie nur auf kleinen Datenmengen basieren.

Und zum Schluss zeige ich, warum die mit Ameisensystemen erweiterten Algorithmen nicht funktioniert haben.

5.2. Qualitätsvergleich

5.2.1. Messmethode

Um die Ergebnis-Qualität der Algorithmen zu vergleichen, habe ich für 3 Matrizengrössen[1], mit jeweils 36 Grössen für die implantierte OPSM, 20 Zufallsmatrizen generiert. Jeder Algorithmus, der in den folgenden Tabellen aufgeführt ist, wurde für jedes Parameter-Set auf jeder der insgesamt 2160 Matrizen 1 Mal[2] ausgeführt.

Definitionen:

- Aussage $A :=$ genau die implantierte OPSM wird gefunden
- Aussage $B :=$ implantierte OPSM, eine in beiden Dimensionen gleich grosse OPSM, oder eine die implantierte OPSM dominierende OPSM wird gefunden
- $p(A)$ bzw. $p(B)$ bezeichnet die Wahrscheinlichkeit, dass A bzw. B eintrifft.

Aus der Definition folgt die Beziehung $A \Rightarrow B$. Trifft B ein, A hingegen nicht, so bedeutet dies, dass die implantierte OPSM statistisch nicht signifikant ist, da zufälligerweise mindestens eine weitere gleich grosse oder sogar grössere OPSM vorhanden ist. Ein Algorithmus wird als erfolgreich angesehen, wenn B eintrifft.

Die Werte in den nachfolgenden Tabellen geben die Wahrscheinlichkeiten $p(A)$ und $p(B)$ an. Jeder Wert beruht auf 20 Messungen mit verschiedenen Matrizen, wobei jeder Algorithmus auf denselben 20 Matrizen ausgeführt wurde.

Aus Gründen der Zeitersparnis habe ich den Ben-Dor-Frech Algorithmus für $l = 10$ und $l = 100$ nur noch auf denjenigen Matrizen laufen lassen, bei denen nicht schon mit kleinerem l die Erfolgswahrscheinlichkeit $p(B) = 1$ betrug. Die $p(A)$-Tabellen weisen daher Lücken auf. In den $p(B)$-Tabellen habe ich die 1-Einträge aus den $p(B)$-Tabellen mit dem nächstkleineren l übernommen, da die Erfolgswahrscheinlichkeit mit zunehmendem l grösser wird.[3]

Die Algorithmen "Ben-Dor Reloaded" und "Ben-Dor-Frech Reloaded" sind hier nicht aufgeführt, da sie die selben Ergebnisse liefern wie "Ben-Dor" bzw. "Ben-Dor-Frech". Für die Messung der Ergebnis-Qualität habe ich nur deshalb nicht die schnelleren Reloaded-Versionen verwendet, weil ich diese zum Zeitpunkt der Qualitätsmessungen noch nicht implementiert hatte.

[1] Die Matrix- und OPSM-Grössen werden wie folgt angegeben: $n \times m$ bzw. $g \times t$.
[2] Da alle getesteten Algorithmen deterministisch sind, genügt dies.
[3] Es mag Matrizen geben, bei denen dies nicht der Fall ist. Statistisch gesehen bilden diese jedoch eine grosse Ausnahme.

Alle Matrizen und Programmprotokolle befinden sich auf der CD im Verzeichnis "OPSM-Extraction/tests/quality_tests/". Ich habe aber darauf verzichtet, die gefundenen Models abspeichern zu lassen, da wenig anzufangen wäre mit zehntausenden von Model-Dateien.

5.2.2. Ergebnisse auf 1000×50 Matrizen

Abbildung 5.1.: Frech mit $lowest\_ratio = 0.05$, $highest\_ratio = 100$, $ratio\_factor = 1.5$

p(A)

g / t	5	8	10	15	20	25
50	0	0.8	1	1	1	1
75	0	0.9	1	1	1	1
100	0	1	1	1	1	1
150	0	1	1	1	1	1
200	0	0.9	1	1	1	1
500	0	1	1	1	1	1

p(B)

g / t	5	8	10	15	20	25
50	0.1	0.8	1	1	1	1
75	0.4	1	1	1	1	1
100	0.65	1	1	1	1	1
150	1	1	1	1	1	1
200	1	1	1	1	1	1
500	1	1	1	1	1	1

Abbildung 5.2.: Frech mit $lowest\_ratio = 0.05$, $highest\_ratio = 100$, $ratio\_factor = 1.25$

p(A)

g / t	5	8	10	15	20	25
50	0	0.8	1	1	1	1
75	0	0.9	1	1	1	1
100	0	1	1	1	1	1
150	0	1	1	1	1	1
200	0	0.9	1	1	1	1
500	0.05	1	1	1	1	1

p(B)

g / t	5	8	10	15	20	25
50	0.05	0.8	1	1	1	1
75	0.4	1	1	1	1	1
100	0.75	1	1	1	1	1
150	1	1	1	1	1	1
200	1	1	1	1	1	1
500	1	1	1	1	1	1

Abbildung 5.3.: Ben-Dor mit $l = 1$

p(A)						
g / t	5	8	10	15	20	25
50	0	0.9	0.95	0.75	0.8	0.25
75	0	0.85	0.9	0.85	0.75	0.65
100	0	1	0.9	0.85	0.85	0.35
150	0	1	1	0.75	0.7	0.6
200	0	0.9	1	0.75	0.85	0.7
500	0.05	1	0.95	0.75	0.95	0.85

p(B)						
g / t	5	8	10	15	20	25
50	0.55	0.95	0.95	0.75	0.8	0.25
75	1	0.95	0.9	0.85	0.75	0.65
100	1	1	0.9	0.85	0.85	0.35
150	1	1	1	0.75	0.7	0.6
200	1	1	1	0.75	0.85	0.7
500	1	1	0.95	0.75	0.95	0.85

Abbildung 5.4.: Ben-Dor mit $l = 10$

p(A)						
g / t	5	8	10	15	20	25
50	0	0.95	1	1	1	0.65
75	0	0.9	1	1	1	1
100	0	1	1	1	1	0.7
150	0	1	1	1	1	0.9
200	0	0.9	1	1	1	1
500	0.05	1	1	1	1	1

p(B)						
g / t	5	8	10	15	20	25
50	0.85	1	1	1	1	0.65
75	1	1	1	1	1	1
100	1	1	1	1	1	0.7
150	1	1	1	1	1	0.9
200	1	1	1	1	1	1
500	1	1	1	1	1	1

Abbildung 5.5.: Ben-Dor-Frech mit $l = 1$

p(A)

g / t	5	8	10	15	20	25
50	0	0.95	1	1	1	1
75	0	0.9	1	1	1	1
100	0	1	1	1	1	1
150	0	1	1	1	1	1
200	0	0.9	1	1	1	1
500	0.05	1	1	1	1	1

p(B)

g / t	5	8	10	15	20	25
50	0.55	1	1	1	1	1
75	1	1	1	1	1	1
100	1	1	1	1	1	1
150	1	1	1	1	1	1
200	1	1	1	1	1	1
500	1	1	1	1	1	1

Abbildung 5.6.: Ben-Dor-Frech mit $l = 10$

p(A)

g / t	5	8	10	15	20	25
50	0					
75						
100						
150						
200						
500						

p(B)

g / t	5	8	10	15	20	25
50	0.9	1	1	1	1	1
75	1	1	1	1	1	1
100	1	1	1	1	1	1
150	1	1	1	1	1	1
200	1	1	1	1	1	1
500	1	1	1	1	1	1

Abbildung 5.7.: Ben-Dor-Frech mit $l = 100$

p(A)

g / t	5	8	10	15	20	25
50	0					
75						
100						
150						
200						
500						

p(B)

g / t	5	8	10	15	20	25
50	0.95	1	1	1	1	1
75	1	1	1	1	1	1
100	1	1	1	1	1	1
150	1	1	1	1	1	1
200	1	1	1	1	1	1
500	1	1	1	1	1	1

5.2.3. Ergebnisse auf 100×100 Matrizen

Abbildung 5.8.: Frech mit $lowest\_ratio = 0.05$, $highest\_ratio = 100$, $ratio\_factor = 1.5$

p(A)

g / t	5	8	10	15	20	50
5	0	0	0	0	0.1	0.95
8	0	0	0	0.4	0.7	1
10	0	0	0.05	0.75	1	1
15	0	0.35	0.75	1	1	1
20	0	0.75	1	1	1	1
50	0.75	1	1	1	1	1

p(B)

g / t	5	8	10	15	20	50
5	0.85	0	0	0	0.1	0.95
8	0.1	0	0	0.4	0.7	1
10	0	0	0.05	0.75	1	1
15	0	0.35	0.75	1	1	1
20	0.1	0.75	1	1	1	1
50	1	1	1	1	1	1

Abbildung 5.9.: Frech mit $lowest\_ratio = 0.05$, $highest\_ratio = 100$, $ratio\_factor = 1.25$

p(A)

g / t	5	8	10	15	20	50
5	0	0	0	0	0.15	0.95
8	0	0	0.1	0.35	0.75	1
10	0	0	0	0.8	1	1
15	0	0.5	0.8	1	1	1
20	0	0.75	1	1	1	1
50	0.75	1	1	1	1	1

p(B)

g / t	5	8	10	15	20	50
5	0.85	0	0	0	0.15	0.95
8	0.15	0	0.1	0.35	0.75	1
10	0	0	0	0.8	1	1
15	0.05	0.5	0.8	1	1	1
20	0.15	0.75	1	1	1	1
50	1	1	1	1	1	1

Abbildung 5.10.: Ben-Dor mit $l = 1$

p(A)

g / t	5	8	10	15	20	50
5	0	0	0	0	0	0
8	0	0	0	0.05	0.15	0
10	0	0	0	0.25	0.2	0
15	0	0.5	0.5	0.45	0.4	0
20	0.25	0.5	0.55	0.7	0.35	0
50	0.75	0.85	0.75	0.6	0.7	0

p(B)

g / t	5	8	10	15	20	50
5	1	0	0	0	0	0
8	0.7	0	0	0.05	0.15	0
10	0.1	0	0	0.25	0.2	0
15	0.1	0.5	0.5	0.45	0.4	0
20	0.5	0.5	0.55	0.7	0.35	0
50	1	0.85	0.75	0.6	0.7	0

Abbildung 5.11.: Ben-Dor mit $l = 10$

p(A)

g / t	5	8	10	15	20	50
5	0	0	0	0.2	0.35	0
8	0	0.2	0.55	0.9	0.95	0
10	0	0.45	0.85	1	0.95	0
15	0.05	0.95	1	1	1	0
20	0.45	1	1	1	1	0
50	0.75	1	1	1	1	0

p(B)

g / t	5	8	10	15	20	50
5	1	0	0	0.2	0.35	0
8	1	0.2	0.55	0.9	0.95	0
10	0.65	0.45	0.85	1	0.95	0
15	0.3	0.95	1	1	1	0
20	0.85	1	1	1	1	0
50	1	1	1	1	1	0

Abbildung 5.12.: Ben-Dor-Frech mit $l = 1$

p(A)

g / t	5	8	10	15	20	50
5	0	0	0	0	0.05	0.4
8	0	0	0.2	0.5	0.8	0.95
10	0	0.05	0.1	0.8	0.85	0.95
15	0	0.65	0.95	1	1	1
20	0.25	0.95	1	1	1	1
50	0.75	1	1	1	1	1

p(B)

g / t	5	8	10	15	20	50
5	1	0	0	0	0.05	0.4
8	0.75	0	0.2	0.5	0.8	0.95
10	0.2	0.05	0.1	0.8	0.85	0.95
15	0.1	0.65	0.95	1	1	1
20	0.55	0.95	1	1	1	1
50	1	1	1	1	1	1

Abbildung 5.13.: Ben-Dor-Frech mit $l = 10$

p(A)

g / t	5	8	10	15	20	50
5		0	0	0.3	0.6	1
8	0	0.25	0.75	0.9	1	1
10	0	0.55	0.85	1	1	1
15	0.05	1	1			
20	0.45	1				
50						

p(B)

g / t	5	8	10	15	20	50
5	1	0	0	0.3	0.6	1
8	1	0.25	0.75	0.9	1	1
10	0.65	0.55	0.85	1	1	1
15	0.3	1	1	1	1	1
20	0.85	1	1	1	1	1
50	1	1	1	1	1	1

Abbildung 5.14.: Ben-Dor-Frech mit $l = 100$

p(A)

g / t	5	8	10	15	20	50
5		0	0	0.85	1	
8		0.7	0.95	1		
10	0	0.95	0.95			
15	0.3					
20	0.55					
50						

p(B)

g / t	5	8	10	15	20	50
5	1	0	0	0.85	1	1
8	1	0.7	0.95	1	1	1
10	1	0.95	0.95	1	1	1
15	0.7	1	1	1	1	1
20	1	1	1	1	1	1
50	1	1	1	1	1	1

5.2.4. Ergebnisse auf 50×200 Matrizen

Abbildung 5.15.: Frech mit $lowest\_ratio = 0.05$, $highest\_ratio = 100$, $ratio\_factor = 1.5$

p(A)

g / t	10	15	20	30	40	100
5	0	0	0	0.1	0.75	1
8	0	0.4	0.45	1	1	1
10	0.1	0.7	1	1	1	1
15	0.8	1	1	1	1	1
20	1	1	1	1	1	1
25	1	1	1	1	1	1

p(B)

g / t	10	15	20	30	40	100
5	0	0	0	0.1	0.75	1
8	0	0.4	0.45	1	1	1
10	0.1	0.7	1	1	1	1
15	0.8	1	1	1	1	1
20	1	1	1	1	1	1
25	1	1	1	1	1	1

Abbildung 5.16.: Frech mit $lowest\_ratio = 0.05$, $highest\_ratio = 100$, $ratio\_factor = 1.5$

p(A)

g / t	10	15	20	30	40	100
5	0	0	0.05	0.3	0.9	1
8	0	0.3	0.7	0.95	1	1
10	0.2	0.65	1	1	1	1
15	0.9	1	1	1	1	1
20	1	1	1	1	1	1
25	1	1	1	1	1	1

p(B)

g / t	10	15	20	30	40	100
5	0	0	0.05	0.3	0.9	1
8	0	0.3	0.7	0.95	1	1
10	0.2	0.65	1	1	1	1
15	0.9	1	1	1	1	1
20	1	1	1	1	1	1
25	1	1	1	1	1	1

Abbildung 5.17.: Ben-Dor mit $l = 1$

p(A)						
g / t	10	15	20	30	40	100
5	0	0	0	0	0	0
8	0.1	0.1	0.15	0	0	0
10	0.2	0.3	0.2	0	0	0
15	0.4	0.4	0.3	0	0	0
20	0.65	0.4	0.3	0	0	0
25	0.65	0.7	0.5	0	0	0

p(B)						
g / t	10	15	20	30	40	100
5	0	0	0	0	0	0
8	0.1	0.1	0.15	0	0	0
10	0.2	0.3	0.2	0	0	0
15	0.4	0.4	0.3	0	0	0
20	0.65	0.4	0.3	0	0	0
25	0.65	0.7	0.5	0	0	0

Abbildung 5.18.: Ben-Dor mit $l = 10$

p(A)						
g / t	10	15	20	30	40	100
5	0	0.3	0.5	0	0	0
8	0.7	0.9	0.95	0	0	0
10	1	1	0.95	0	0	0
15	1	1	1	0	0	0
20	1	1	0.95	0	0	0
25	1	1	1	0	0	0

p(B)						
g / t	10	15	20	30	40	100
5	0	0.3	0.5	0	0	0
8	0.7	0.9	0.95	0	0	0
10	1	1	0.95	0	0	0
15	1	1	1	0	0	0
20	1	1	0.95	0	0	0
25	1	1	1	0	0	0

Abbildung 5.19.: Ben-Dor-Frech mit $l = 1$

p(A)						
g / t	10	15	20	30	40	100
5	0	0	0.05	0.15	0.4	0.95
8	0.25	0.55	0.75	0.75	0.85	1
10	0.65	0.9	0.95	1	1	1
15	0.95	1	1	1	1	1
20	1	1	1	1	1	1
25	1	1	1	1	1	1

p(B)						
g / t	10	15	20	30	40	100
5	0	0	0.05	0.15	0.4	0.95
8	0.25	0.55	0.75	0.75	0.85	1
10	0.65	0.9	0.95	1	1	1
15	0.95	1	1	1	1	1
20	1	1	1	1	1	1
25	1	1	1	1	1	1

Abbildung 5.20.: Ben-Dor-Frech mit $l = 10$

p(A)						
g / t	10	15	20	30	40	100
5	0	0.45	0.8	0.9	1	1
8	0.75	0.95	1	1	1	
10	1	1	1			
15	1					
20						
25						

p(B)						
g / t	10	15	20	30	40	100
5	0.05	0.45	0.8	0.9	1	1
8	0.75	0.95	1	1	1	1
10	1	1	1	1	1	1
15	1	1	1	1	1	1
20	1	1	1	1	1	1
25	1	1	1	1	1	1

Abbildung 5.21.: Ben-Dor-Frech mit $l = 100$

p(A)						
g / t	10	15	20	30	40	100
5	0.1	1	1	1		
8	1	1				
10						
15						
20						
25						

p(B)						
g / t	10	15	20	30	40	100
5	0.1	1	1	1	1	1
8	1	1	1	1	1	1
10	1	1	1	1	1	1
15	1	1	1	1	1	1
20	1	1	1	1	1	1
25	1	1	1	1	1	1

5.2.5. Statistische Signifikanz

Ein Vergleichstest mit reinen Zufallsmatrizen zeigt, wie statistisch signifikant die implantierten OPSM's in den getetesten Problemen sind. Die Dreiecke und Kreise in den folgenden Diagrammen zeigen OPSM's, die von Frech und Ben-Dor-Frech in reinen Zufallsmatrizen gefunden wurden. Die roten Punkte markieren die Grössen der implantierten OPSM's.

Die Diagramme zeigen, dass die vier kleinsten OPSM's, die in die 100 × 100-Matrizen implantiert wurden, statistisch sicher nicht signifikant sind. Dies passt zur Beobachtung, dass in den 100 × 100-Matrizen in diesen Problemen oft grössere als die implantierten OPSM's gefunden wurden.

Die implantierten OPSM's mit 5 Spalten in den 1000 × 50-Matrizen hingegen scheinen gemäss den Diagrammen statistisch einigermassen signifikant zu sein, trotzdem wurden sie nur selten gefunden. Das liegt daran, dass zwar viele der implantierten Spalten gefunden werden, aber auch einige fremde. Damit wird nicht genau die implantierte OPSM gefunden.

Abbildung 5.22.: Signifikanztest auf 1000 × 50-Matrizen

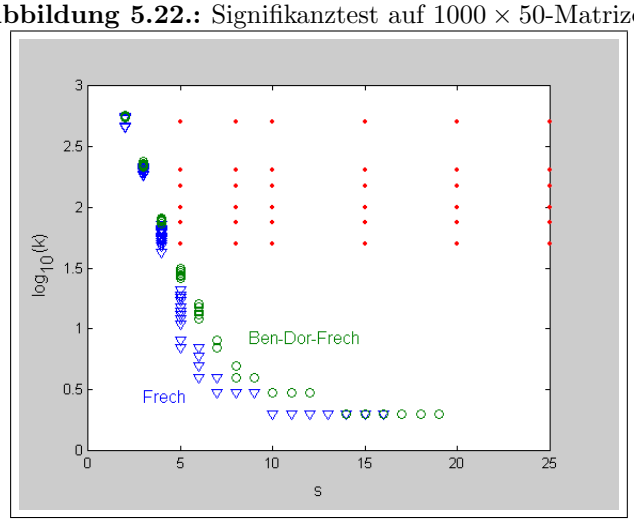


Abbildung 5.23.: Signifikanztest auf 100×100-Matrizen

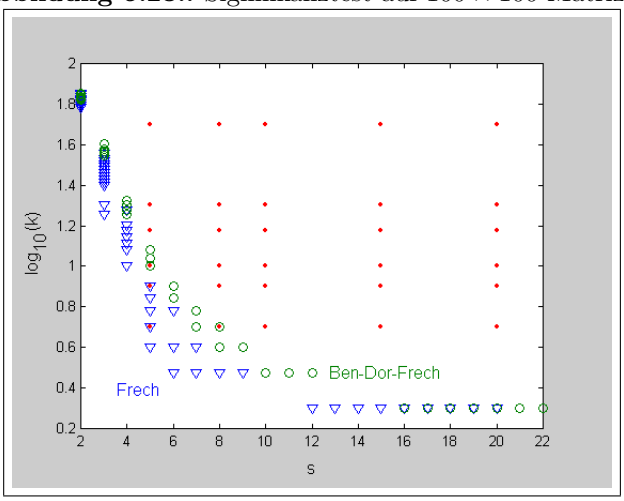
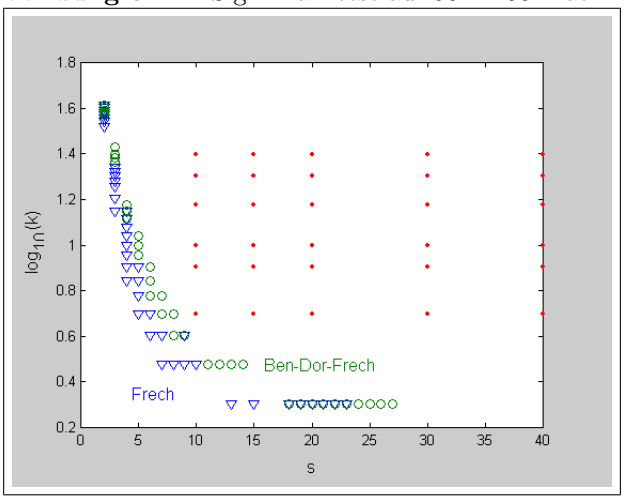


Abbildung 5.24.: Signifikanztest auf 50×200-Matrizen



5.2.6. Interpretation

Frech Algorithmus

Die Erfolgswahrscheinlichkeit des Frech Algorithmus scheint sich - einigermassen unabhängig von den Problemdimensionen - nach dem Flächenverhältnis von implantierter OPSM zu Expressionsmatrix zu richten.

In den Fällen mit implantierten OPSM's, die mehr als 5% der Expressionsmatrixfläche ausmachen, beträgt die Erfolgswahrscheinlichkeit $p(B) = 1$. Beträgt die Fläche der implantierten OPSM mehr als 2%, ist die Erfolgswahrscheinlichkeit $p(B)$ immer noch recht hoch. Erst wenn die implantierte OPSM kleiner ist als 1%, sinkt die Erfolgswahrscheinlichkeit auf 0 ab, sofern die implantierte OPSM statistisch signifikant ist.

Die Ergebnisse der Läufe mit $ratio\_factor = 1.5$ und $ratio\_factor = 1.25$ unterscheiden sich nur geringfügig. Zum Teil - bedingt durch Zufälligkeiten - liefert der Lauf mit dem kleineren $ratio\_factor$ sogar etwas schlechtere Ergebnisse. Dies zeigt, dass sich die Qualität der Ergebnisse durch eine Verfeinerung der *ratio*-Abstufungen nicht beliebig verbessern lassen, was die Annahme eines konstanten Wertes für die Anzahl der zu verwendenden *ratio*-Werte rechtfertigt.

Ben-Dor Algorithmus

Der Ben-Dor Algorithmus ist sehr stark, wenn es ums Finden von sehr kleinen OPSM's geht. Auch steigt die Erfolgswahrscheinlichkeit mit zunehmender Zeilenzahl der OPSM an. Aber sobald die OPSM mehr als 20-30 Spalten enthält, hat Ben-Dor grosse Mühe, die OPSM vollstänig zu finden. Wie in Abschnitt 4.7.1 erläutert, liegt das daran, dass Ben-Dor beim Aufbau des Models einige Spalten überspringt.

Die Erhöhung des Parameters l bringt die erwartete Leistungssteigerung, allerdings auf Kosten einer gleichermassen erhöhten Laufzeit.

Ben-Dor-Frech Algorithmus

Nimmt man das statistische Modell von Amir Ben-Dor als Mass, so lässt der Ben-Dor-Frech Algorithmus nicht mehr viele Wünsche offen. Mit $l = 100$ ist Ben-Dor-Frech nur dann nicht erfolgreich, wenn die implantierte OPSM statistisch nur knapp signifikant ist.

Vergleich

Dass Ben-Dor-Frech besser abschneidet als Ben-Dor überrascht nicht, denn das erklärt sich dadurch, dass Ben-Dor-Frech die Ergebnisse von Ben-Dor verwendet. Erstaunlich ist aber, dass Ben-Dor-Frech bereits mit $l = 1$ vergleichbare Ergebnisse liefert wie Ben-Dor mit $l = 10$. Der Versuch, die Schwäche des Ben-Dor Algorithmus zu beheben, ist gelungen. Er ist sogar so gut gelungen, dass auch der Frech Algorithmus bei keiner Problemstellung besser ist.

Im Vergleich mit Frech hat Ben-Dor-Frech noch einen weiteren Vorteil: Die Ergebnisqualität kann bei Ben-Dor-Frech durch Erhöhung des Wertes für l beliebig gesteigert

werden, sofern genügend Rechenleistung zur Verfügung steht. Bei Frech hingegen bringt es nichts, den *ratio*-Bereich beliebig fein zu unterteilen, weil das Scheitern von Frech bei kleinen OPSM's nicht durch unpassende *ratio*-Werte verursacht wird, sondern durch zu starkes Rauschen.

Fazit: Was die Qualität der Ergebnisse betrifft, ist Ben-Dor-Frech in allen Problemstellungen der beste Algorithmus.

5.3. Laufzeitvergleich

5.3.1. Messmethode

Für die Laufzeitmessungen habe ich spezielle Matrizen angefertigt mit den folgenden Eigenschaften:

- $m = n/10$
- $g = n/5$
- $t = m/5$

Die Seitenverhältnisse der Expressionsmatrix und der implementierten OPSM sind damit konstant, und das Grössenverhältnis von Expressionmatrix zu implementierter OPSM ist ebenfalls konstant. Damit soll erreicht werden, dass die Probleme möglichst gut skalieren und keine Störeinflüsse durch unterschiedliche Problemstrukturen erzeugt werden.[4]

Ich habe für alle Laufzeitmessungen meinen 2GHz-Pentium4 PC mit Windows XP verwendet, wobei ausser den Betriebssystemprozessen keine anderen Prozesse liefen.

Die Messgenauigkeit beträgt 1 Sekunde, die Standardabweichung ca. 1%. Alle Werte sind, sofern nicht anders gekennzeichnet, in Sekunden angegeben. Werte, die weniger als 2000s betragen, habe ich 3 Mal gemessen und gemittelt, die übrigen Werte habe ich 1 Mal gemessen. Die Werte in Klammern habe ich nicht gemessen, sondern mit Hilfe der gemessenen Werten und der Laufzeitkomplexität geschätzt, denn aus Zeitgründen musste ich darauf verzichten, Messungen durchzuführen, die länger als 2 Stunden dauern.

[4] Es hat sich gezeigt, dass die Laufzeit nicht nur von der Expressionsmatrixgrösse, sondern auch von der Grösse und dem Seitenverhältnis der implantierten OPSM abhängt.

5.3.2. Ergebnisse

Abbildung 5.25.: Laufzeitmessungen

n	250	500	1000	2000	3000	4000	6000
Frech mit $ratio = 10$	0	0	0	4	13	32	110
Frech mit 35 *ratio*-Werten	0	3	27	209	693	1607	5518
Ben-Dor mit $l = 10$	3	23	279	4130	(5.8h)	(18h)	(3.9d)
Ben-Dor Plus mit $l = 10$	3	23	276	4051	(5.7h)	(18h)	(3.8d)
Ben-Dor-Frech mit $l = 10$	3	23	281	4059	(5.7h)	(18h)	(3.8d)
Ben-Dor Reloaded mit $l = 10$	1	14	206	3178	(4.5h)	(14h)	(3.0d)
Ben-Dor-Frech Reloaded mit $l = 10$	1	13	206	3202	(4.5h)	(14h)	(3.0d)
Ben-Dor mit $l = 100$	21	138	1139	(5.1h)	(26h)	(3.4d)	(17d)
Ben-Dor Plus mit $l = 100$	20	135	1125	(5.0h)	(25h)	(3.3d)	(17d)
Ben-Dor-Frech mit $l = 100$	22	143	1167	(5.2h)	(26h)	(3.5d)	(18d)
Ben-Dor Reloaded mit $l = 100$	7	56	617	(2.7h)	(14h)	(1.8d)	(9.3d)
Ben-Dor-Frech Reloaded mit $l = 100$	7	55	619	(2.8h)	(14h)	(1.8d)	(9.3d)

Alle Dateien zu den Laufzeitmessungen befinden sich auf der CD im Verzeichnis "OPSMExtraction/tests/runtime_tests/".

5.3.3. Interpretation

Frech

Während der Frech Algorithmus keine Vorteile hat bezüglich Ergebnis-Qualität, offenbart sich bei der Laufzeitmessung seine Stärke. Bei den Problemgrössen, die für die Laufzeitmessung verwendet wurden, macht sich der Unterschied von Frech's $O(n^3)$ zu Ben-Dor's $O(n^4 \cdot l)$ bereits deutlich bemerkbar. Und diese Problemgrössen sind durchaus realistisch.

Ebenfalls interessant ist die Beobachtung, dass eine Erhöhung der Anzahl *ratio*-Werte eine überproportionale Laufzeiterhöhung bewirkt (50-fache Laufzeit für 35-fache Anzahl *ratio*-Werte). Die Laufzeit von Frech mit einem einzelnen *ratio*-Wert ist also nicht unabhängig vom gewählten Wert für *ratio*. Da die Anzahl der zu verwendenen *ratio*-Werte, wie in Abschnitt 4.2.7 erläutert, als konstant angenommen wird, hat diese Beobachtung aber keinen Einfluss auf die Laufzeitkomplexität.

Ben-Dor Familie

Die Implementation der Reloaded-Versionen hat sich gelohnt, die Laufzeit wird deutlich verringert - wobei der Effekt bei grösseren Werten für l stärker zum Tragen kommt.

Bei genauerem Hinsehen fällt auf, dass Ben-Dor zum Teil höhere Laufzeiten aufweist als die Erweiterungen Ben-Dor Plus und Ben-Dor-Frech, was eigentlich nicht sein kann. Das zeigt, dass diese drei Algorithmen im Rahmen der Messgenauigkeit gleich schnell sind.

Die Erhöhung des Wertes für l hat erwartungsgemäss eine deutliche Steigerung der Laufzeit zur Folge, wobei diese nicht so stark ausfällt, wie die Laufzeitkomplexität $O(n^4 \cdot l)$ erwarten lassen würde. Das liegt wohl daran, dass der lineare Laufzeitanstieg erst bei grösseren Werten von l eintritt, da im verwendeten Bereich Anfangseffekte noch eine Rolle spielen.

Vergleich

Klassiert man die Algorithmen nach Ergebnis-Qualität und Laufzeit, so gibt es 2 pareto-optimale Algorithmen: Den schnellen, aber qualitativ bescheideneren Frech Algorithmus und den langsamen, aber qualitativ guten Ben-Dor-Frech Algorithmus.

Je nach Problemgrösse, verfügbarer Zeit und Rechenkapazität ist der Frech-Algorithmus als einziger anwendbar.

5.4. Vergleich mit Ben-Dor's Ergebnissen

5.4.1. Messmethode

Die nachfolgenden Messungen habe ich nach dem gleichen Prinzip durchgeführt wie die Messungen in Abschnitt 5.2, jedoch mit den Problemgrössen, die Amir Ben-Dor in [1] verwendet hat: 1000×50-Expressionsmatrizen und Parameter $l = 100$, wie in Abschnitt 5.2 bereits verwendet, jedoch andere Grössen für die implantierten OPSM's.

Während Ben-Dor für seine Messungen 100 Matrizen verwendet hat pro Eintrag, habe ich mich aus Zeitgründen auf 20 Matrizen pro Eintrag beschränkt und dafür noch Vergleichsmessungen mit dem Frech Algorithmus und dem Ben-Dor-Frech Algorithmus durchgeführt.

5.4.2. Ergebnisse

Abbildung 5.26.: Ergebnisse von Amir Ben-Dor für Ben-Dor Algorithmus mit $l = 100$

p(A)

E[g] / t	3	4	5	7	10
25	0	0.01	0.21	0.72	0.92
50	0.17	0.7	0.94	1	1
75	0.69	0.98	1	1	1
100	0.92	1	1	1	1

p(B)

E[g] / t	3	4	5	7	10
25	1	0.94	0.43	0.72	0.92
50	1	0.82	0.94	1	1
75	1	0.98	1	1	1
100	1	1	1	1	1

Abbildung 5.27.: Meine Ergebnisse für Ben-Dor Algorithmus mit $l = 100$

p(A)

g / t	3	4	5	7	10
25	0	0	0	0.55	1
50	0	0	0	0.9	1
75	0	0	0	0.9	1
100	0	0	0	0.9	1

p(B)

g / t	3	4	5	7	10
25	1	1	1	0.65	1
50	1	1	0.95	1	1
75	1	1	1	1	1
100	1	1	1	1	1

Abbildung 5.28.: Ben-Dor-Frech mit $l = 100$

p(A)

g / t	3	4	5	7	10
25	0	0	0	0.65	1
50	0	0	0	0.9	1
75	0	0	0	0.9	1
100	0	0	0	0.9	1

p(B)

g / t	3	4	5	7	10
25	1	1	1	0.75	1
50	1	1	0.95	1	1
75	1	1	1	1	1
100	1	1	1	1	1

Abbildung 5.29.: Frech mit $lowest\_ratio = 1$, $highest\_ratio = 50$, $ratio\_factor = 1.25$

p(A)

g / t	3	4	5	7	10
25	0	0	0	0	0.2
50	0	0	0	0.45	1
75	0	0	0	0.85	1
100	0	0	0	0.9	1

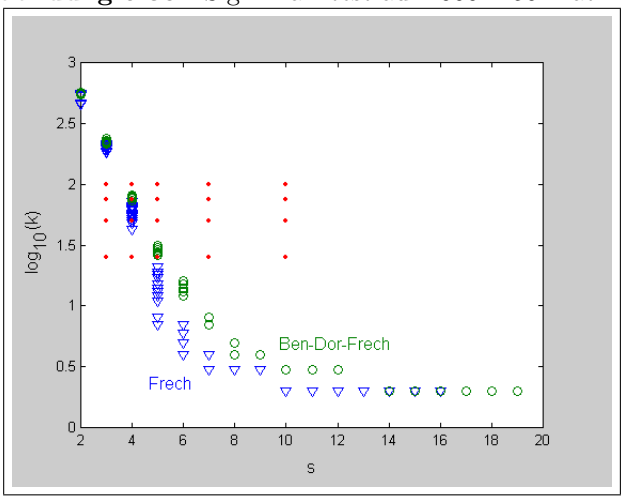
p(B)

g / t	3	4	5	7	10
25	1	1	0	0	0.2
50	1	1	0	0.5	1
75	1	0.15	0.2	0.95	1
100	1	0.4	0.75	1	1

5.4.3. Statistische Signifikanz

Ein Blick auf das Diagramm mit den in reinen Zufallsmatrizen gefundenen OPSM's zeigt, dass viele der von Ben-Dor verwendeten OPSM-Grössen statistisch nicht signifikant sind. Dies ist der Grund, warum ich beim Qualitätstest auf so kleine OPSM's verzichtet habe. (Erläuterung des Diagramms: siehe Abschnitt 5.2.5.)

Abbildung 5.30.: Signifikanztest auf 1000×50-Matrizen



5.4.4. Interpretation

Die Werte von Amir Ben-Dor weichen teilweise erheblich von meinen eigenen Werten ab. Ein Teil der Abweichung ist dadurch zu erklären, dass Ben-Dor ein etwas anderes statistisches Modell verwendet hat: er legt lediglich den Erwartungswert $E[g]$ der Anzahl implantierter Zeilen fest, nicht den exakten Wert g. Damit lassen sich die Abweichungen in der $p(B)$-Tabelle erklären: bei Ben-Dor's Messungen war g zufallsbedingt manchmal kleiner als in meinen Messungen und die implantierte OPSM daher schwieriger zu finden (weil sie statistisch oft trotzdem noch so relevant war, dass keine andere statistisch relevantere OPSM gefunden werden konnte).

Hingegen findet Ben-Dor in seinen Messungen für die Wahrscheinlichkeit $p(A)$, die implantierte OPSM zu finden, viel grössere Werte als ich in meinen Messungen. Die Abweichungen sind zu gross, als dass sie sich nur durch den kleinen Unterschied im statistischen Modell erklären liessen.

Wie sich herausstellte, beruht der Unterschied auf einem Missverständnis: Ben-Dor schreibt in [1]: "Table 2 reports the probabilities that the algorithm recovers correctly

the set of planted columns ...". Ich habe diese Aussage dahingehend interpretiert, dass Tabelle 2 die Wahrscheinlichkeiten angibt, dass genau die implantierte OPSM (nicht mehr und nicht weniger) gefunden wird. Wie Abbildung 5.31 zeigt, scheint Tabelle 2 jedoch anzugeben, wie gross die Wahrscheinlichkeit ist, dass die implantierte OPSM eine Teilmenge[5] der gefundenen OPSM ist. Ich nenne diese Wahrscheinlichkeit $p(A')$.

Abbildung 5.31.: Ben-Dor Algorithmus mit $l = 100$

p(A') gemäss Ben-Dor

E[g] / t	3	4	5	7	10
25	0	0.01	0.21	0.72	0.92
50	0.17	0.7	0.94	1	1
75	0.69	0.98	1	1	1
100	0.92	1	1	1	1

p(A') gemäss meinen Messungen

g / t	3	4	5	7	10
25	0.05	0	0.35	0.65	1
50	0	0.8	0.95	1	1
75	0.55	1	1	1	1
100	0.95	1	1	1	1

Damit sind die Ergebnisse vergleichbar. Ich gehe deshalb davon aus, dass die logische Funktion meiner Implementierung derjenigen von Ben-Dor entspricht.

In den Laufzeitmessungen besteht jedoch eine grosse Diskrepanz, deren Ursache ich nicht vollständig ermitteln konnte: Ben-Dor gibt für seinen in Matlab implementierten Algorithmus mit $l = 100$ eine Laufzeit von 23 Sekunden an auf einem 500MHz-PC, auf meinem 2GHz-PC dauert die Ausführung der Ben-Dor Reloaded Java-Implementation etwa 100 Sekunden.

Meine Korrektur der Formel für A_i (siehe Abschnitt 4.5.5) kann aus 2 Gründen nicht für die erhöhte Laufzeit verantwortlich sein. Erstens benötigt der Kompatibilitätstest in meiner Implementierung kaum zusätzliche Zeit. Zweitens lieferte die Implementierung mit der unkorrigierten Formel deutlich schlechtere Ergebnisse. Da sich Ben-Dor's Ergebnisse aber nicht wesentlich von meinen unterscheiden, ist davon auszugehen, dass er die selbe Formel verwendet hat wie ich.

Die Ursache für den Unterschied ist am ehesten in der Implementierung des numerischen Verfahrens zur Lösung der impliziten Gleichung für p zu suchen. Durch die Wahl eines besseren Startwertes für p ($p_0 = 0.05$ statt $p_0 = 0.5$) konnte ich die Laufzeit von 102 Sekunden auf 86 Sekunden reduzieren. Zusätzlich konnte ich durch eine Reduktion der Genauigkeit von $\epsilon = 0.0001$ nach $\epsilon = 0.01$ die Laufzeit auf 63 Sekunden reduzieren. Dabei leidet allerdings, wie ich feststellen musste, die Qualität der Ergebnisse. Deshalb habe ich zwar den besseren Startwert für p beibehalten[6], die Genauigkeitsreduktion jedoch

[5] Wobei ich eine OPSM O_1 als Teilmenge der OPSM O_2 qualifiziere, wenn die Spaltenmenge von O_1 eine Teilmenge der Spaltenmenge von O_2 ist, die Rangfolge der Spalten in O_1 nicht gegen die Rangfolge der Spalten in O_2 verstösst und die Zeilenmenge von O_1 eine Teilmenge der Zeilenmenge von O_2 ist.

[6] Für die Laufzeitmessungen in Abschnitt 5.3 kam diese Verbesserung jedoch zu spät, ich hatte sie bereits mit dem alten Wert $p_0 = 0.5$ durchgeführt.

rückgängig gemacht. Möglicherweise ist ein Teil der Abweichungen in der $p(B)$-Tabelle auch dadurch zu erklären, dass Ben-Dor dank einer reduzierten Genauigkeit Laufzeit gespart hat.

Meiner Meinung nach muss es aber noch weitere Gründe für den Laufzeitunterschied geben, denn ich glaube nicht, dass Java um einen Faktor 10 langsamer ist als Matlab, wenn keine Grafiken angezeigt werden müssen.

5.5. Ergebnisse auf realen Daten

5.5.1. Messmethode

Die reale Expressionsmatrix, die mir zur Verfügung stand, umfasst 22810 Zeilen und 153 Spalten und war damit zu gross, um von Ben-Dor oder Ben-Dor-Frech in vernünftiger Zeit analysiert zu werden. Ich habe sie deshalb in 1000×153 Submatrizen aufgeteilt. Auf den ersten 4 Submatrizen habe ich die Algorithmen Frech, Ben-Dor und Ben-Dor-Frech laufen lassen. Für den Frech Algorithmus habe ich die Parameter $lowest\_ratio = 0.1$, $highest\_ratio = 200$ und $ratio\_ractor = 1.25$ verwendet. Für Ben-Dor und Ben-Dor-Frech habe ich $l = 1000$ gesetzt. Die Laufzeit[7] für Frech betrug ca. 50 Sekunden, für Ben-Dor und Ben-Dor-Frech ca. 2 Stunden. Die von Frech gefundenen OPSM's sind blau eingezeichnet, die von Ben-Dor gefundenen grün und die von Ben-Dor-Frech gefundenen rot. Die Dreiecke markieren die von den drei Algorithmen in einer reinen Zufallsmatrix gemeinsam gefundenen OPSM's und geben damit einen Hinweis darauf, welche der in den realen Daten gefundenen OPSM's statistisch signifikant sind.

Alle Dateien zu diesen Messungen befinden sich auf der CD im Verzeichnis "OPSMExtraction/tests/realdata_tests/".

[7] Die in den Programmberichten angegebenen Laufzeiten sind nur bedingt aussagekräftig, da diese Tests als Hintergrundprozesse mit niedriger Priorität ausgeführt wurden. Bezüglich Laufzeit am aussagekräftigsten scheint mir Ben-Dor-Frech auf Teil 3 zu sein, weil dieser in einer Nacht als einziger lief und den Prozessor daher gut ausnutzen konnte.

5.5.2. Ergebnisse

Abbildung 5.32.: Teil 1

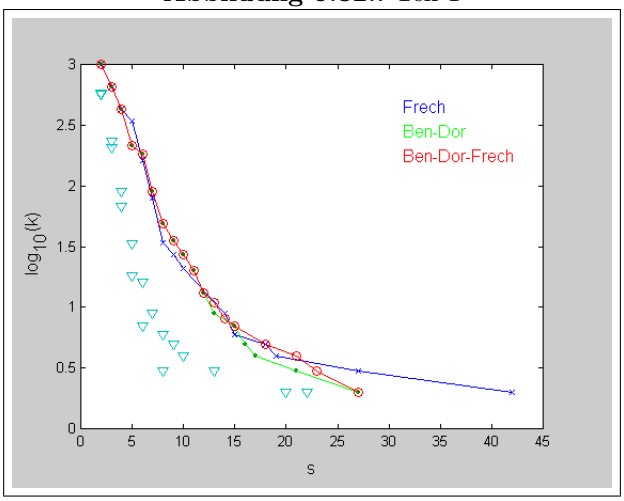


Abbildung 5.33.: Teil 2

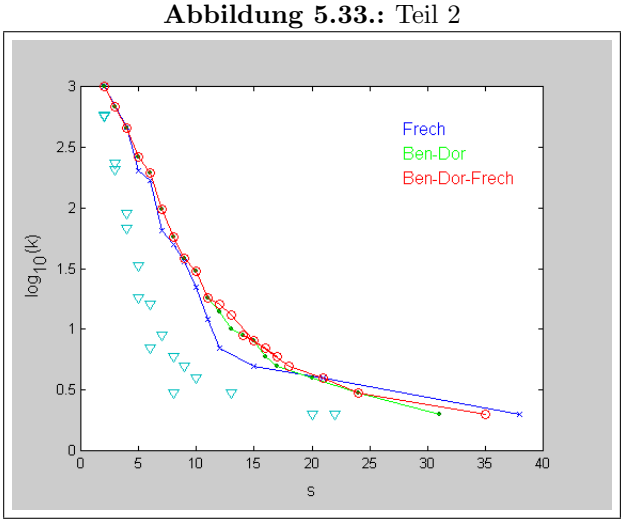


Abbildung 5.34.: Teil 3

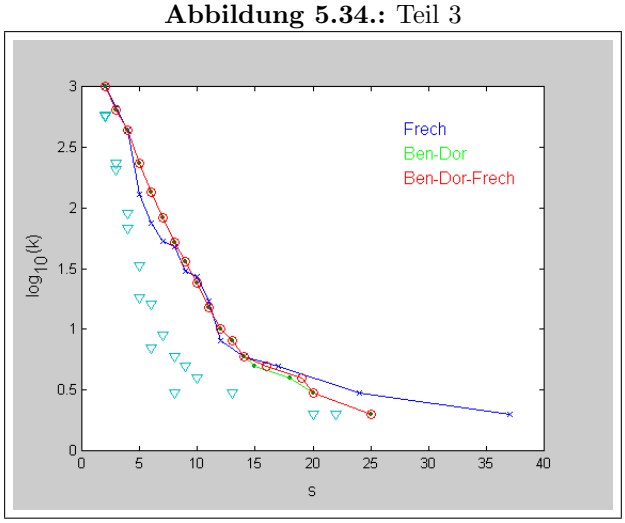
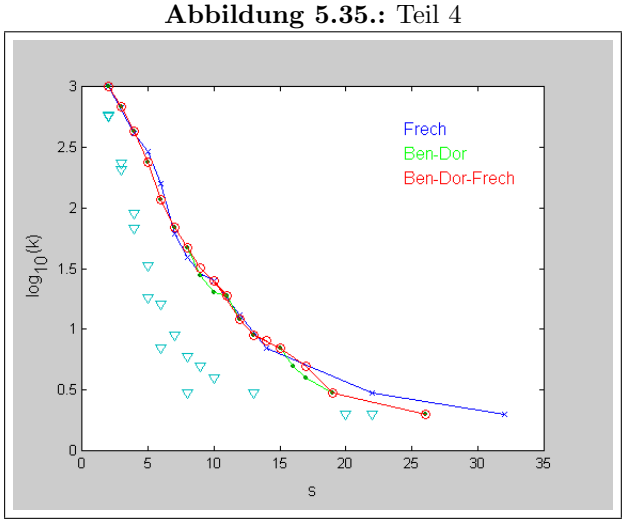


Abbildung 5.35.: Teil 4



5.5.3. Interpretation

Die Vergleichsmessung mit der Zufallsmatrix lässt vermuten, dass die OPSM's im mittleren Bereich im Diagramm allesamt statistisch signifikant sind.

Erstaunlicherweise liefert der Frech Algorithmus auf den realen Daten - von denen anzunehmen ist, dass sie auch Gegentrends enthalten, die ihm Schwierigkeiten bereiten - ausser in Teil 2 vergleichbare, zum Teil sogar bessere Resultate wie Ben-Dor und Ben-Dor-Frech. Wobei der Frech Algorithmus nur einen Bruchteil der Zeit von Ben-Dor benötigt. Zumindest in diesem Fall hat er sich als schnelle Heuristik bewährt.

Ebenfalls zu sehen ist, dass Ben-Dor-Frech die Ergebnisse von Ben-Dor in einigen Fällen noch etwas nachbessern kann, vor allem im statistisch signifikanten Bereich. Die Entwicklung von Ben-Dor-Frech scheint sich gelohnt zu haben.

Fazit: Die ersten Ergebnisse sind sehr zufriedenstellend, es sollten aber noch weitere Tests zur Verifizierung dieser Ergebnisse gemacht werden.

5.6. Ameisen Algorithmen

5.6.1. Einleitung

Wie sich gezeigt hat, sind die deterministischen Algorithmen, die in dieser Arbeit untersucht wurden, recht gut. In vielen der getesteten Fällen sind sie in der Lage, die implantierte OPSM in vernünftig kurzer Zeit zu finden. Ein Ameisen Algorithmus muss deshalb darauf abzielen, entweder die OPSM's schneller zu finden als die deterministischen Algorithmen, oder OPSM's zu finden, die von den deterministischen Algorithmen nicht gefunden werden können, wobei dann die Laufzeit eine untergeordnete Rolle spielt.

Nun ist es aber so, dass der Ben-Dor Algorithmus prinzipiell jede implantierte OPSM finden kann, wenn der Parameter l genügend erhöht wird, weil der Ben-Dor Algorithmus für $l > e \cdot m!$[8] eine erschöpfende Suche ausführt. Wegen der Endlichkeit des Suchraumes gibt es also keine OPSM, die nicht gefunden werden könnte.

Daher lautet die Minimalforderung an die Ameisen Algorithmen, dass sie in einigen Fällen in einer bestimmten Zeit eine statistisch signifikantere OPSM finden als der Ben-Dor Algorithmus in der gleichen Zeit.

Es ist für Ameisen Algorithmen ziemlich aussichtslos, mit den deterministischen Algorithmen konkurrieren zu wollen, wenn diese die implantierte OPSM in kurzer Zeit finden, da Ameisen Algorithmen die Laufzeit der zugrundeliegenden Heuristik vervielfachen. Erst im Bereich, in dem beim Ben-Dor Algorithmus eine Erhöhung von l um einen grossen Faktor nur noch eine geringe Verbesserung bewirkt, kann ein Ameisen Algorithmus konkurrieren.

Die Schwäche des Ben-Dor Algorithmus im Fall von implantierten OPSM's mit vielen Spalten habe ich bereits behoben, indem ich den Frech Algorithmus implementierte. Gemeinsam finden diese beiden Algorithmen alle mässig grossen bis sehr grossen OPSM's in vernünftiger Zeit. Ich habe daher versucht, mit den Ameisen Algorithmen kleine, statistisch jedoch noch signifikante OPSM's zu finden.

5.6.2. Ergebnisse

Leider konnte ich mit den AS bzw. ACS-Versionen des Frech Algorithmus und des Ben-Dor Algorithmus nie eine substantielle Verbesserung der Ergebnisse der zugrundeliegenden deterministischen Algorithmen erzielen.

Im Vergleich zu Frech lieferten Frech AS und Frech ACS mit 1000 Ameisendurchläufen zwar oft leicht verbesserte OPSM's (meist ein paar wenige Zeilen und Spalten mehr),

[8] siehe Abschnitt 3.4

aber diese Verbesserungen konnten weitgehend auch vom Frech Random Algorithmus (einfache Randomisierung, siehe Abschnitt 4.2.9) erzielt werden, wenn man ihn 1000 Mal laufen liess und die besten Ergebnisse sammelte. Die kleine Verbesserung, die AS bzw. ACS gebracht hat, ist also in erster Linie auf die Randomisierung zurückzuführen, das Pheromon hingegen schien fast keine Wirkung zu haben. Und die Verbesserung ist deutlich zu klein, um eine 1000-fach erhöhte Laufzeit zu rechtfertigen. Frech AS und Frech ACS können bei Weitem nicht mit Ben-Dor konkurrieren.

Ein Beispiel zur Performance von Frech AS (vergleichbar mit Frech ACS): In einer $n \times m = 200 \times 50$ Matrix mit einer implantierten $g \times t = 10 \times 10$ OPSM findet Frech nach weniger als 1 Sekunde eine 10×5 und eine 5×6 OPSM, Frech AS nach 139 Sekunden eine 6×8 und eine 11×7 OPSM und Ben-Dor nach nur 3 Sekunden mit $l = 1$ die implantierte OPSM. Dabei ist noch anzumerken, dass der Frech AS Algorithmus nur mit einem einzigen *ratio*-Wert laufen gelassen wurde und die Laufzeit sich also etwa verdreissigfachen würde, wenn die Information über das Seitenverhältnis der zu suchenden OPSM nicht genutzt werden könnte, so wie es in realen Anwendungen der Fall ist.

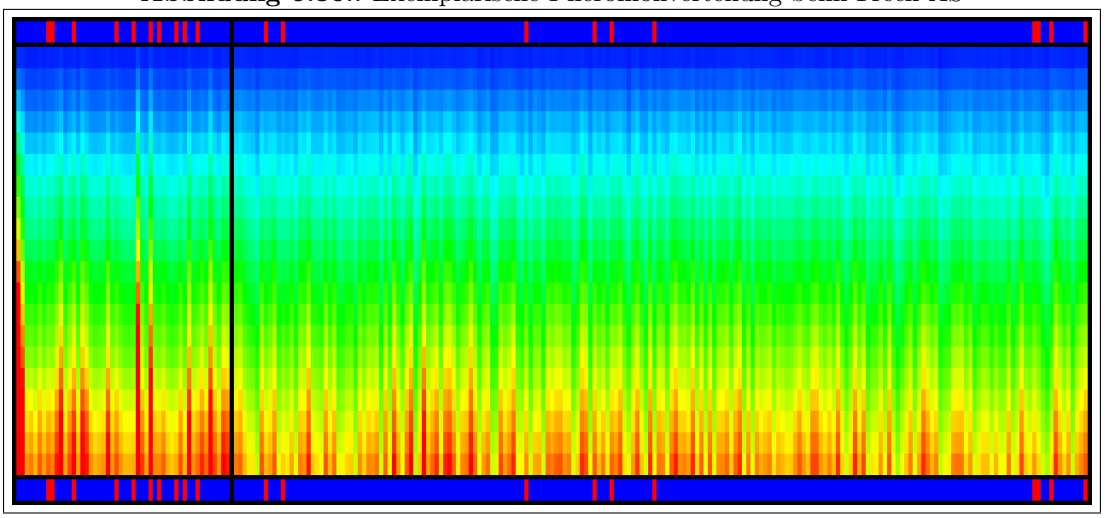
Abbildung 5.36 zeigt exemplarisch, wie wenig die Pheromonverteilung mit den implantierten Zeilen und Spalten zu tun hat. Der linke Bereich stellt das Spalten-Pheromon dar (1 Spalte pro Diagrammspalte), der rechte Bereich das Zeilen-Pheromon (1 Zeile pro Diagrammspalte). Die oberste und die unterste Diagrammzeile zeigt mit roten Balken die implantierten Zeilen und Spalten an. Im Mittelteil wird in jeder Diagrammzeile die Pheromonverteilung nach einer Ameisen-Iteration (entspricht dem Durchlauf von 50 Ameisen) angezeigt. Rot bedeutet, dass viel Pheromon auf einer Zeile bzw. Spalte liegt, blau bedeutet wenig Pheromon.

Während der Frech Algorithmus relativ schnell ist und daher eine 1000-fach erhöhte Laufzeit für die Experimente mit verschiedenen Parametern in Frech AS und Frech ACS in Kauf genommen werden konnte, stellte mich der Ben-Dor ACS Algorithmus diesbezüglich vor ein Problem. Ich reduzierte daher die Anzahl Ameisendurchläufe auf 100. Trotzdem konnte ich wegen Zeitmangel nicht viele Tests durchführen, und es ist mir auch nicht gelungen, mit dem Ben-Dor ACS Algorithmus bessere Ergebnisse zu erzielen als mit Ben-Dor. Im besten Fall wurden Ben-Dor's Ergebnisse nicht verschlechtert.

5.6.3. Gründe für das Scheitern

Ich hatte zuwenig Zeit, um die Frage, ob die Algorithmen "Frech" und "Ben-Dor" sinnvoll mit Ameisen Systemen erweitert werden können, abschliessend zu klären. Die folgenden Ausführungen zeigen jedoch, dass die Anwendung von Ameisen Algorithmen auf das OPSM-Problem mit erheblichen Schwierigkeiten verbunden ist.

Abbildung 5.36.: Exemplarische Pheromonverteilung beim Frech AS



Bedingungen für erfolgreiche Anwendung von AS bzw. ACS

Unter anderem müssen folgende Bedingungen erfüllt sein, um Ameisen Algorithmen erfolgreich anzuwenden (auch nachzulesen in [2], S.67f):

1. Das Problem muss Lokalität aufweisen.
2. Es muss eine vernünftige Heuristik zur Lösung des Problems existieren.
3. Die Wahrscheinlichkeit, dass eine zufällige Abweichung von der Heuristik eine Verbesserung bewirkt, darf nicht zu klein sein.

Wie im Folgenden gezeigt wird, ist für den Frech Algorithmus keine dieser drei Bedingungen erfüllt, für den Ben-Dor Algorithmus ist nur Bedingung 2 erfüllt.

Problem 1: Fehlende Lokalität

Im Gegensatz zum TSP[9] kennt das OPSM-Problem keine Lokalität. Abbildungen 5.37 und 5.38 veranschaulichen dies. Die roten Punkte in Abbildung 5.38 markieren die Zeilen und Spalten einer OPSM.

Beim TSP bewirkt eine kleine Änderung des Weges nur eine kleine Änderung der Weglänge. Und wenn in einem Teilbereich A eine Verbesserung erzielt wird, macht dies

[9] Travelling Salesman Problem

Abbildung 5.37.: Lokalität beim TSP

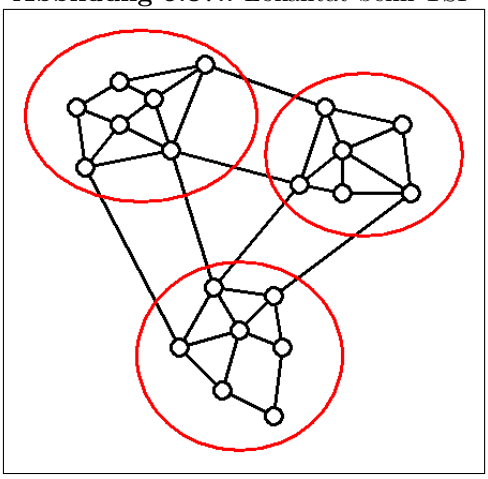
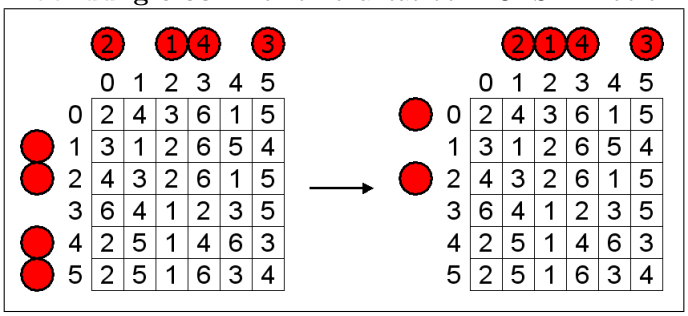


Abbildung 5.38.: Keine Lokalität beim OPSM-Problem



Verbesserungen in einem Teilbereich B nicht zunichte. Daher ist es möglich, den Gesamtweg zu verkürzen, indem viele kleine lokale Verbesserungen vorgenommen werden.

Im OPSM-Problem ist dies nicht der Fall. Wird im Model eine einzige Position verändert so kann sich die Menge der kompatiblen Zeilen stark ändern. Im gezeigten Beispiel verursacht die kleine Änderung des Models von $T_1 = [2, 0, 5, 3]$ nach $T_2 = [2, 1, 5, 3]$ die grosse Änderung der kompatiblen Zeilenmenge von $G_1 = \{1, 2, 4, 5\}$ nach $G_2 = \{0, 2\}$. In Frech AS bzw. Frech ACS entsteht dadurch das Problem, dass das Pheromon, das nach dem Finden des Models T_1 auf die dazu kompatiblen Zeilen gelegt wurde, bei einem anderen Model wie T_2, das sich zudem nur geringfügig von T_1 unterscheidet, die falschen Zeilen favorisiert. Mit dieser Begebenheit hat zwar Ben-Dor ACS keine Probleme, da bei diesem Algorithmus das Pheromon nur auf die Spalten zu liegen kommt, doch eine lokale Optimierung ist bei Ben-Dor aus einem anderen Grund nicht möglich: werden in zwei Durchgängen zwei gute Models gefunden mit unterschiedlichen Spalten, so bewirkt das Pheromon, dass die Spalten aus den beiden Models in späteren Durchgängen kombiniert

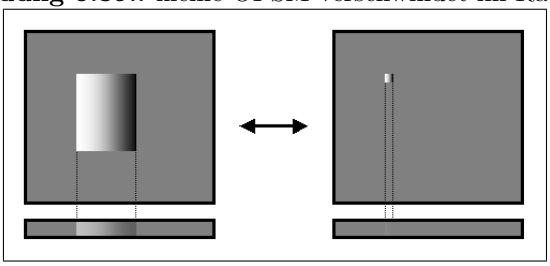
werden. Doch die Kombination von Spalten aus zwei guten Models ergibt meist ein schlechtes Model, weil viele Zeilen, die zum ersten Model kompatibel sind, zum zweiten Model nicht kompatibel sind und umgekehrt.

Problem 2: Keine gute Heuristik

Wenn der Frech Algorithmus die implantierte OPSM nicht findet, liegt das daran, dass diese im Rauschen untergeht (siehe Abbildung 5.39). In diesem Fall stellt der Frech Algorithmus jedoch eine zweifelhafte Heuristik dar, da das Rauschen die Bewertung der Zeilen und Spalten erheblich stört und zu vielen Fehlentscheidungen führt.

Dies ist ein spezifisches Problem des Frech Algorithmus, beim Ben-Dor Algorithmus besteht dieses Problem nicht.

Abbildung 5.39.: kleine OPSM verschwindet im Rauschen



Problem 3: Wenig Chancen für Randomisierung

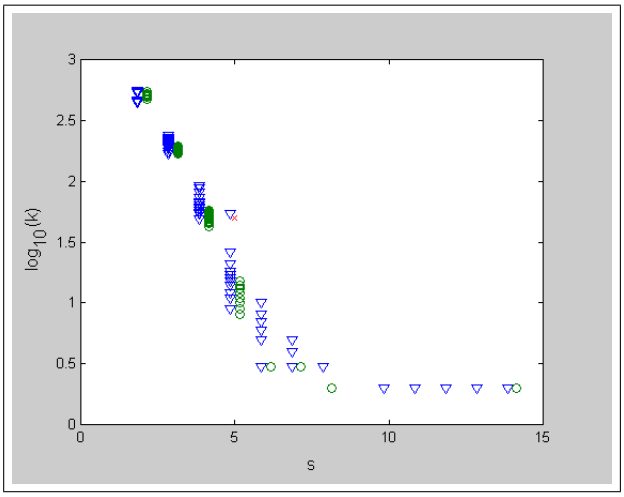
Um festzustellen, wie gross die Wahrscheinlichkeit ist, dass eine Randomisierung zum Erfolg führt, habe ich die randomisierten Algorithmen "Frech Random" und "Ben-Dor Random" implementiert und mit den deterministischen Algorithmen "Frech" und "Ben-Dor" verglichen. Für den Vergleich habe ich die Matrizen aus dem Qualitätstest verwendet, wobei ich Problemgrössen ausgewählt habe, die von den deterministischen Algorithmen nur mit mässiger bis kleiner Wahrscheinlichkeit erfolgreich bewältigt werden konnten, so dass eine Randomisierung Sinn machen könnte. Für den Vergleich von Frech mit Frech Random habe ich für jede der 3 Matrixgrössen 2 geeignete OPSM-Grössen ausgewählt. Bei Ben-Dor war die Auswahl etwas kleiner, weshalb ich mich für den Vergleich von Ben-Dor mit Ben-Dor Random auf 3 Problemdimensionen beschränkt habe. Für jeden Vergleich habe ich beide Algorithmen je einmal auf jeder der 20 Matrizen ausgeführt.

Wie die folgenden Abbildungen zeigen, ist das Ergebnis ernüchternd. Zwar wurden durch die Randomisierung in einigen Fällen OPSM's mit zusätzlichen Zeilen gefunden, dies jedoch fast ausschliesslich bei den statistisch nicht signifikanten OPSM's mit 2 oder 3

Spalten, die auch mit den zusätzlichen Zeilen nicht statistisch signifikant wurden.[10] Im Mittel wurde sowohl die Anzahl Zeilen wie auch die Anzahl Spalten deutlich reduziert. Die blauen Dreiecke stellen die Ergebnisse der deterministischen Algorithmen dar, die grünen Kreise die Ergebnisse der randomisierten Algorithmen. Damit sich die Dreiecke und Kreise nicht wirr überlappen, sondern die Häufungsbereiche der Ergebnisse der beiden Algorithmen erkennbar werden, habe ich die Dreiecke jeweils ein bisschen nach links, die Kreise ein bisschen nach rechts verschoben. Das rote Kreuz markiert die implantierte OPSM.

Alle Dateien zu den Vergleichsmessungen befinden sich auf der CD im Verzeichnis "OPSMExtraction/tests/randomization_tests/".

Abbildung 5.40.: Frech versus Frech Random auf 1000×50 Matrix mit 50×5 OPSM



[10] Bsp: In einer 1000x50 Matrix gibt es zwangsläufig immer mehrere 500x2 OPSM's, 167x3 OPSM's, 42x4 OPSM's etc., und durch Zufälligkeiten werden diese zwangsläufig vorhandenen OPSM's sogar noch grösser.

Abbildung 5.41.: Frech versus Frech Random auf 1000×50 Matrix mit 100×5 OPSM

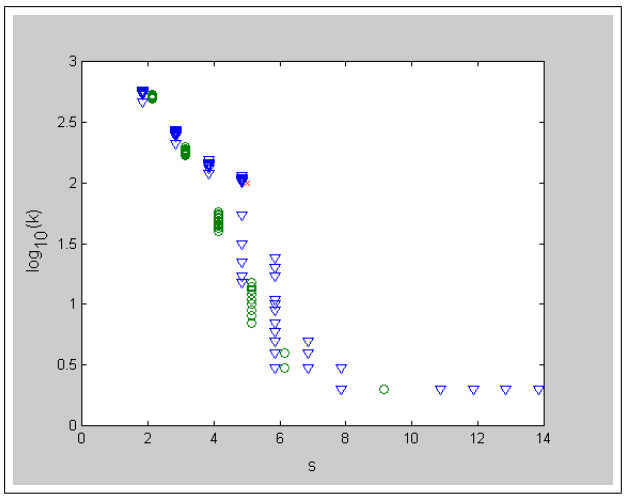


Abbildung 5.42.: Frech versus Frech Random auf 100×100 Matrix mit 8×10 OPSM

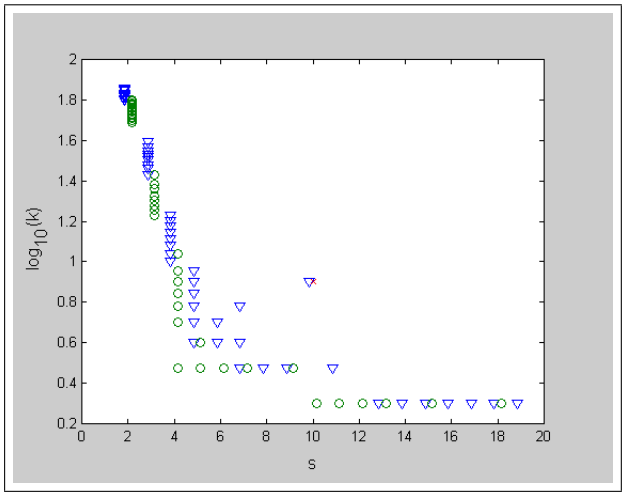


Abbildung 5.43.: Frech versus Frech Random auf 100×100 Matrix mit 15×8 OPSM

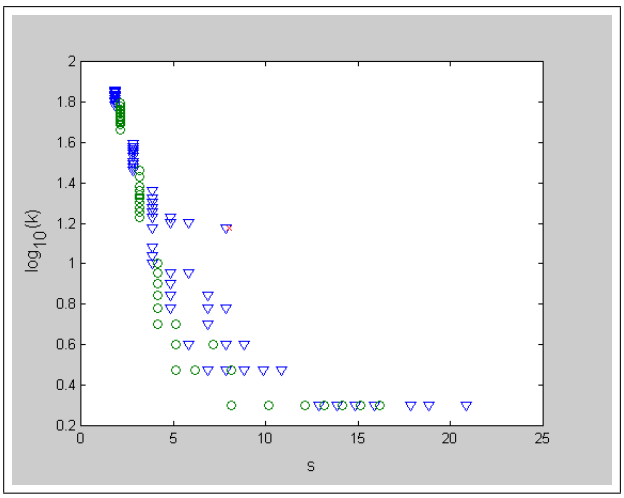


Abbildung 5.44.: Frech versus Frech Random auf 50×200 Matrix mit 5×20 OPSM

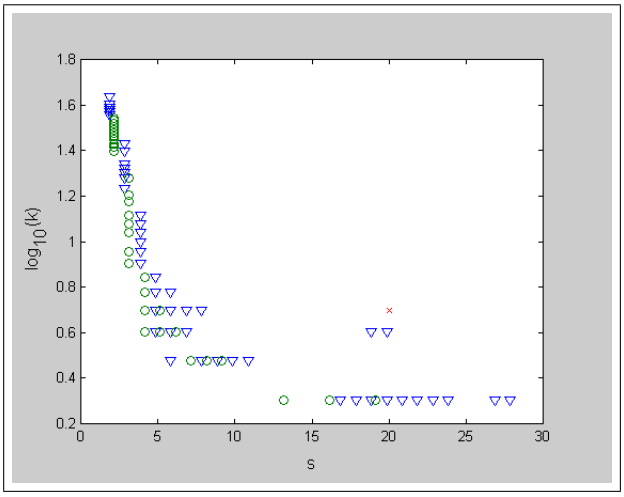


Abbildung 5.45.: Frech versus Frech Random auf 50×200 Matrix mit 8×15 OPSM

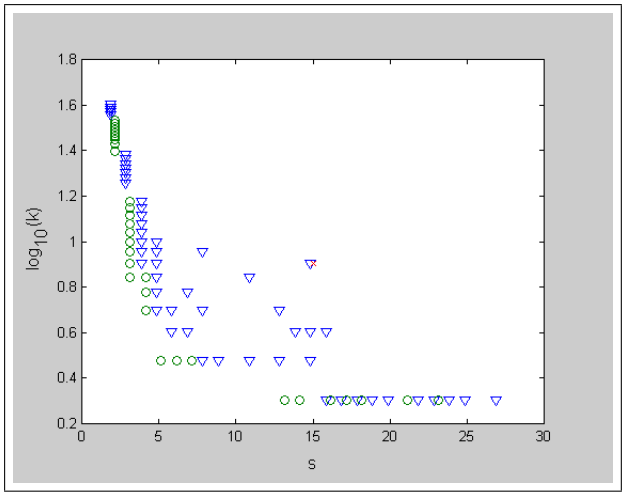


Abbildung 5.46.: Ben-Dor versus Ben-Dor Random auf 100×100 Matrix mit 5×10 OPSM

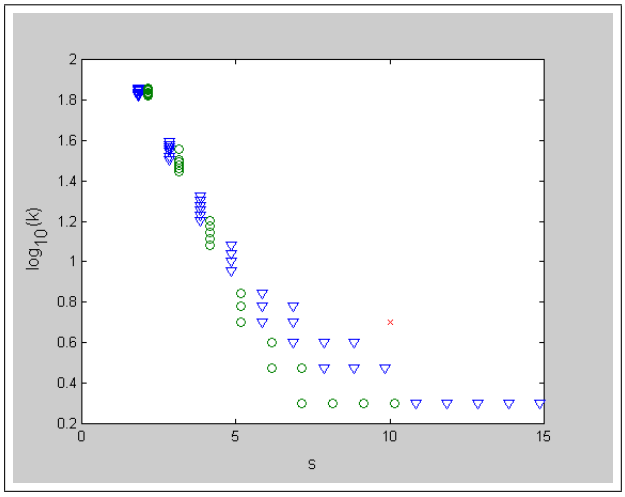


Abbildung 5.47.: Ben-Dor versus Ben-Dor Random auf 100×100 Matrix mit 8×8 OPSM

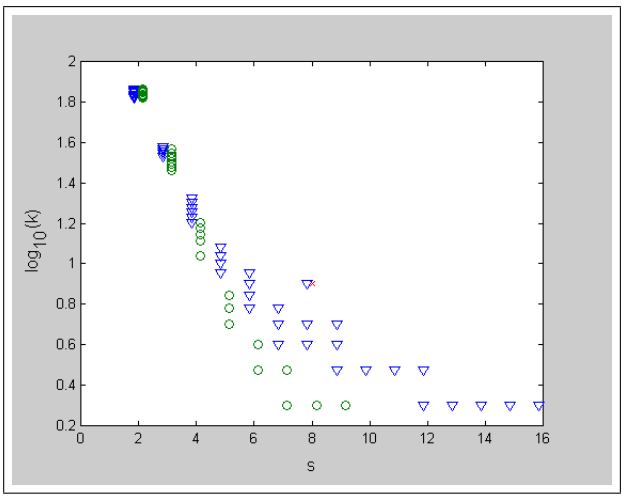
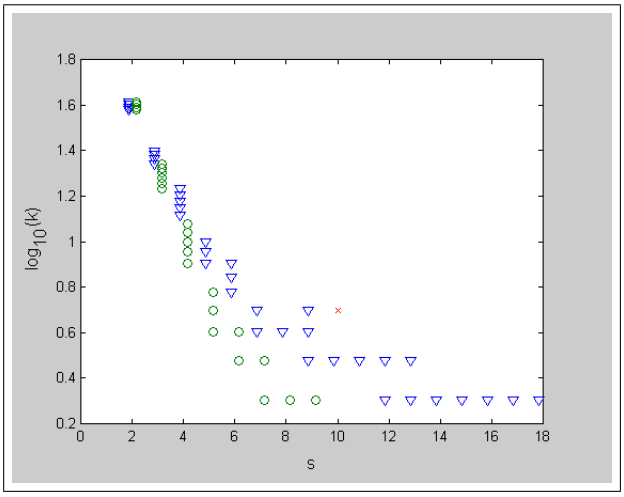


Abbildung 5.48.: Ben-Dor versus Ben-Dor Random auf 50×200 Matrix mit 5×10 OPSM



Wenn die Randomisierung stark abgeschwächt wurde (Frech Random: Wahrscheinlichkeit, eine Zeile oder Spalte zu entfernen proportional zu Kollisionszahl potenziert mit 5; Ben-Dor Random: Wahrscheinlichkeit, eine Spalte ins Model einzufügen, proportional zu p^5), war die Verschlechterung der deterministischen Ergebnissen zwar nicht mehr so ausgeprägt, doch es konnten wie bei der starken Randomisierung keine nennenswerten Verbesserungen im signifikanten Bereich erzielt werden, wie die folgenden Abbildungen zeigen:

Abbildung 5.49.: Frech versus mod. Frech Random auf 1000×50 Matrix mit 50×5 OPSM

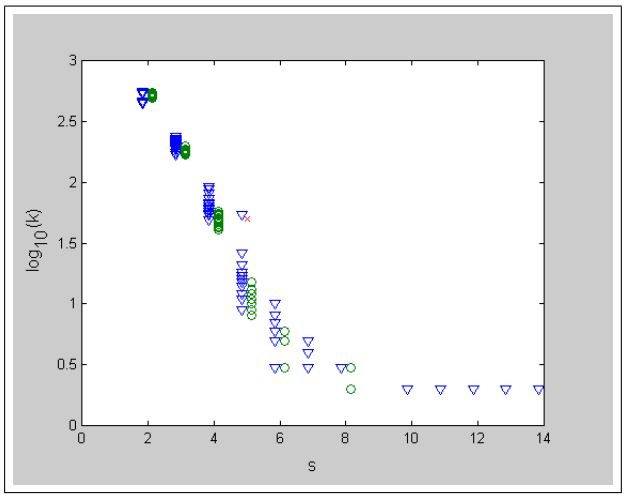


Abbildung 5.50.: Frech versus mod. Frech Random auf 1000×50 Matrix mit 100×5 OPSM

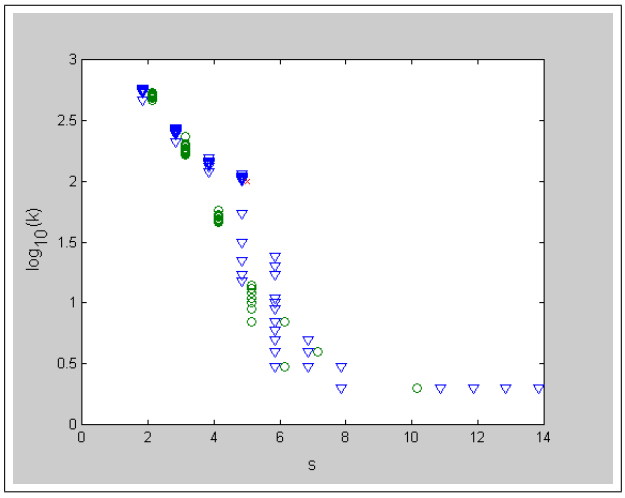


Abbildung 5.51.: Ben-Dor versus mod. Ben-Dor Random auf 100×100 Matrix mit 5×10 OPSM

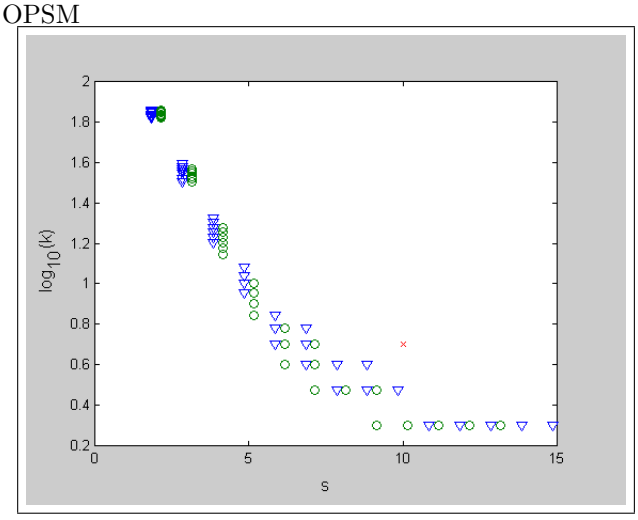


Abbildung 5.52.: Ben-Dor versus mod. Ben-Dor Random auf 100×100 Matrix mit 8×8 OPSM

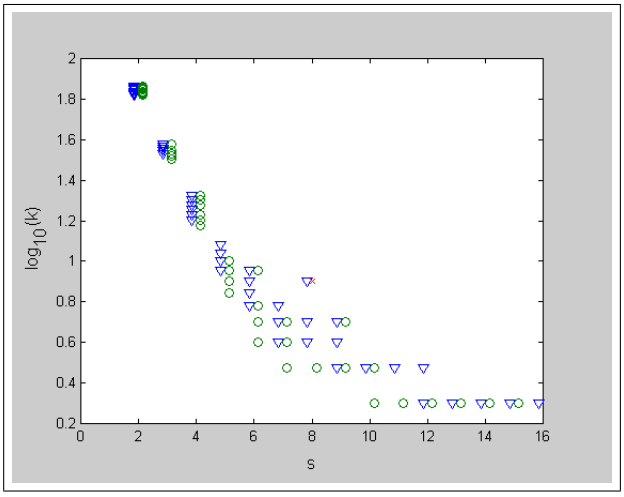
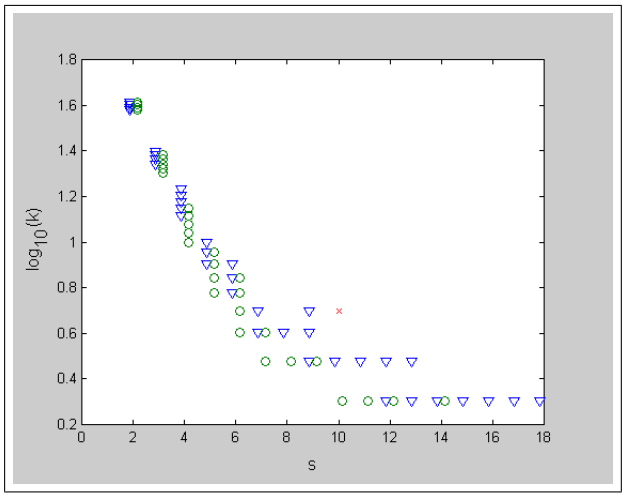


Abbildung 5.53.: Ben-Dor versus mod. Ben-Dor Random auf 50×200 Matrix mit 5×10 OPSM



Selbst wenn die Randomisierung in seltenen Fällen erfolgreich ist - die Erfolgswahrscheinlichkeit ist zu klein, als dass mit Hilfe von Pheromon die Verbesserungen verstärkt werden könnten.

5.6.4. Fazit

Die fehlende Lokalität und die geringe Erfolgswahrscheinlichkeit einer Randomisierung lassen eine sinnvolle Anwendung von Ameisen Algorithmen auf das OPSM-Problem kaum zu. Aus denselben Gründen sind auch andere Randomisierungsverfahren und lokale Suchverfahren für das OPSM-Problem wenig geeignet.

6. Schlussfolgerungen

Zusammenfassend können folgende Schlussfolgerungen aus meiner Arbeit gezogen werden:

6.1. Frech schneller, Ben-Dor besser

Der Frech Algorithmus hat sich als schnelle Heuristik bewährt, die zur ersten, schnellen Analyse von Daten eingesetzt werden kann um einen Überblick zu gewinnen. Der Ben-Dor Algorithmus liefert danach mit viel Rechenaufwand etwas bessere Ergebnisse, wobei die Qualität der Ergebnisse umso besser wird, je mehr Laufzeit investiert wird.

6.2. Gelungene Kombination Ben-Dor-Frech

Die Kombination von Ben-Dor und Frech zu Ben-Dor-Frech ist gelungen und scheint sich auch in der Anwendung auf reale Daten auszuzahlen. Dies, weil mit sehr wenig zusätzlicher Rechenleistung die Ergebnisse von Ben-Dor in vielen Fällen verbessert werden können.

6.3. Keine Ameisen für OPSM-Problem

Der sinnvolle Einsatz von Ameisen Algorithmen auf das OPSM-Problem scheint mangels Lokalität und mangels Erfolg von Randomisierungen nicht möglich zu sein.

7. Ideen

Hinweis:

Wenn ich in diesem Kapitel von "Algorithmen" spreche, meine ich in erster Linie die Pareto-optimalen Algorithmen, die in dieser Arbeit präsentiert wurden, also "Frech" und "Ben-Dor-Frech Reloaded".

7.1. Gegenläufige Trends berücksichtigen

Es liegt in der Natur von biologischen Prozessen, dass unter sich verändernden Bedingungen die Aktivität bei einigen Genen zu- und bei anderen Genen abnimmt. Es wäre daher sinnvoll, "Kompatibilität" so umzudefinieren, dass nicht nur Gene, die dem Trend folgen, als kompatibel gelten, sondern auch Gene, die dem genau gegenläufigen Trend folgen. Die Algorithmen wären dementsprechend anzupassen, wobei der Aufwand dafür nicht allzu gross wäre. In der Ausgabe der Algorithmen müsste dann ersichtlich sein, welche Gene dem Trend und welche dem gegenläufigen Trend folgen.

Beim Frech Algorithmus, dem gegenläufige Trends Schwierigkeiten bereiten, wäre zudem folgendes Vorgehen denkbar: Die mittlere Rangfolge wird wie gehabt durch Aufsummierung der Ränge ermittelt. Das Zählen der Kollisionen wird jedoch 2 Mal durchgeführt: einmal dem Trend folgend und einmal dem gegenläufigen Trend folgend. Falls eine Zeile beim gegenläufigen Trend weniger Kollisionen verursacht, werden alle Ränge in dieser Zeile umgedreht, d.h. der grösste Rang wird zum kleinsten, der zweitgrösste zum zweitkleinsten etc. - und verursacht eine umgedrehte Zeile in einem späteren Iterationsschritt beim Trend weniger Kollisionen, wird sie wieder zurückgedreht. Dann wird erneut die mittlere Rangfolge ermittelt, dann wieder die Kollisionen gezählt, etc., bis sich die Situation stabilisiert, also keine (oder fast keine) Zeilen mehr umgedreht werden müssen. Dann erst werden Zeilen oder Spalten entfernt. So kann der Keim, der vorhanden ist in Matrizen mit unterschiedlich grossen Mengen von Trend- und Gegentrend-Zeilen, zum Spriessen gebracht werden.

7.2. Mehrere OPSM's implantieren

In realen Matrizen gibt es nicht nur 1 statistisch signifikante OPSM. Das statistische Modell von Amir Ben-Dor ist deshalb zu erweitern. Es ist zu untersuchen, wie die Algorithmen sich verhalten, wenn mehrere OPSM's implantiert sind, die sich auch überschneiden können. Gegebenenfalls sind die Algorithmen für diesen Fall zu optimieren. Die Algorithmen müssten auch dahingehend angepasst werden, dass nicht nur die Pareto-optimalen OPSM's, sondern alle OPSM's, die eine bestimmte Grösse erreichen, ausgegeben werden.

7.3. OPSM-Kriterium entschärfen

Messungen enthalten immer Messfehler, und gerade bei biologischen Messungen sind diese oft sehr gross. Das scharfe OPSM-Kriterium (kompatible Gene müssen dem Trend exakt folgen) ist daher zu hart für reale Anwendungen. Der Benützer der Algorithmen sollte eine Schranke festlegen können, wieviele Abweichungen vom Trend erlaubt sind pro Gen, damit es noch als kompatibel angesehen wird. Sinnvoll wäre es, auch die Höhe der Abweichungen zu berücksichtigen, d.h. ein kompatibles Gen dürfte viele kleine aber nur wenige grosse Abweichungen aufweisen. Um die Höhe einer Abweichung zu bestimmen, sollten die unrangierten Expressionswerte verwendet werden.

Der Frech Algorithmus hat mit der Entschärfung des OPSM-Kriteriums keine Mühe, lediglich die Methode zur Bestimmung der Kompatibilität müsste umgeschrieben werden. Beim Ben-Dor-Frech Algorithmus ist der Aufwand grösser, da die statistischen Formeln angepasst werden müssen.

7.4. Unrangierte Expressionsmatrix verwenden

Der Ben-Dor Algorithmus ist auf eine Rangierung der Expressionswerte angewiesen wegen der statistischen Berechnungen. Der Frech Algorithmus hingegen könnte auch auf unrangierte Matrizen angewendet werden. Es sollte untersucht werden, ob und in welchen Fällen dies einen positiven Effekt hat bezüglich Ergebnis-Qualität. Ich vermute, dass eine kleine OPSM mit einem stark ausgeprägten Trend in der unrangierten Matrix besser erkannt werden könnte.

A. Inhalt der CD-ROM

- `documentation`: Präsentationsfolien und Bericht
- `documentation/LaTeX`: LaTeX Quellcode des Berichts
- `OPSMExtraction`: Java Projektverzeichnis und Wurzelverzeichnis für Algorithmen
- `OPSMExtraction/src`: Java Quellcode
- `OPSMExtraction/opsmextraction`: kompilierte Java Klassen
- `OPSMExtraction/tests`: Dateien zu den Tests
- `OPSMExtraction/examples`: Dateien zu den Beispielen
- `matlab`: Matlab-Dateien

Abbildungsverzeichnis

Literaturverzeichnis

[1] Amir Ben-Dor, Benny Chor, Richard Karp, and Zohar Yakhini. Discovering local structure in gene expression data: The order-preserving submatrix problem. In *International Conference on Computational Biology*, pages 49–57. ACM Press, 2002.

[2] E. Bonabeau, M. Dorigo, and G. Theraulaz. *Swarm Intelligence. From Natural to Artificial Systems*. Oxford University Press, 1999.

[3] Y. Cheng and G. M. Church. Biclustering of gene expression data. In *ISMB 2000*, pages 93–103, 2000. http://cheng.ecescs.uc.edu/biclustering.

[4] Marco Dorigo, Gianni Di Caro, and Luca M. Gambardella. Ant algorithms for discrete optimization. 1999.

[5] M. B. Eisen, P. T. Spellman, P. O. Brown, and D. Botstein. Cluster analysis and display of genome-wide expression patterns. *PNAS*, 95:14863–14868, December 1998.

[6] P. Tamayo et al. Interpreting patterns of gene expression with self-organizing maps: Methods and applicataion to hematopoietic differentiation. *PNAS*, 96:2907 – 2912, March 1999.

[7] A. Tanay, R. Sharan, and R. Shamir. Discovering statistically significant biclusters in gene expression data. *Bioinformatics*, 18(Suppl. 1):S136–S144, 2002.