PC- & PQ-trees

Ausarbeitung für das Seminar

Algorithm Engineering

Sommersemester 2009

von

Thomas Siwczyk

Technische Universität Dortmund
Fakultät für Informatik
Lehrstuhl XI

Datum: 8. Juli 2009

Betreuung:
Prof. Petra Mutzel
Dr. Markus Chimani
Bernd Zey

Inhaltsverzeichnis

1 Einleitung

Diese Arbeit, welche im Rahmen des Seminars *Algorithm Engineering* entstand, behandelt die Datenstrukturen *PQ-tree* und *PC-tree*, welche jeweils zur Repräsentation von Teilmengen aller möglichen Permutationen eines Alphabetes verwendet werden. Die jeweilige dargestellte Teilmenge muss dabei eine Eigenschaft erfüllen, was im Falle der *PQ-tree* die *Consecutive Ones Property* und im Falle der *PC-tree* die *Circular Ones Property* ist.

Die Besonderheit im Zusammenhang mit dem Algorithm Engineering liegt hierbei darin, dass beide Datenstrukturen bei ihrer Konstruktion komplett unterschiedliche Ansätze wählen. Dabei wird es sich herausstellen, dass die Vorgehensweise der *PC-trees* weitaus effizienter ist, als die der *PQ-Bäume*. Dieser Effizienzvorteil soll anschließend für die andere Datenstruktur genutzt werden, was durch eine Transformation bewerkstelligt wird.

Beginnen wird diese Arbeit mit den *PQ-trees*, welche 1976 von Booth und Lueker in [BL76] eingeführt worden sind. Dem folgt eine Beschreibung der *PC-trees*, welche erst viel später von Wen-Lian Hsu in [SH99] präsentiert wurden.

2 Der PQ-tree

Im folgenden soll die Datenstruktur *PQ-tree*, wie auch die *Consecutive Ones Property* , definiert und an einem Beispiel verdeutlicht werden. Anschließend wird der Aufbau eines solchen Baumes beschrieben und die Laufzeit kurz analysiert werden. Die Notation folgt dabei [Heu06].

2.1 Definition eines PQ-tree

Ein *PQ-tree* ist ein gewurzelter Baum, bei dem zwischen P-, Q-Knoten und Blättern unterschieden wird. Ein P-Knoten ist dabei ein Knoten mit mindestens zwei Kindern, wobei seine Kinder in beliebiger Reihenfolge angeordnet werden können. Die Q-Knoten hingegen besitzen mindestens drei Kinder, deren Reihenfolge nur umgedreht werden darf. Zudem gibt es für jedes $a \in \Sigma$ des Eingabealphabetes exakt ein Blatt. Die Darstellung der unterschiedlichen Knoten in allen folgenden Beispielen ist unter Abbildung 1 zu sehen. Formaler lässt sich ein *PQ-tree* wie folgt induktiv definieren:

Abbildung 1: Darstellung von P- und Q-Knoten

Definition 1. *Gegeben sei ein endliches Alphabet* Σ*, dann gilt für einen* PQ-tree *folgendes:*

- *Jedes Blatt bestehend aus einem Zeichen* $a \in \Sigma$ *ist ein* PQ-tree
- *Ein P-Knoten mit seinen Kindern* $T_1, \ldots, T_k$ *(*$k \geq 2$*) ist ein* PQ-tree*, wenn* $T_1 \ldots T_k$ *ebenfalls* PQ-trees *sind.*
- *Ein Q-Knoten mit seinen Kindern* $T_1, \ldots, T_k$ *(*$k \geq 3$*) ist ein* PQ-tree*, wenn* $T_1 \ldots T_k$ *ebenfalls* PQ-trees *sind.*

Zudem werden noch folgende Definitionen im folgen benötigt:

Definition 2. *Für einen* PQ-tree T *über dem Alphabet* Σ *sei die Frontier* $F(T)$ *die Permutation, welche man durch das Ablesen der Blätter von links nach rechts erhält.*

Definition 3. *Zwei* PQ-trees T *und* T' *heißen äquivalent zueinander (kurz* $T \cong T'$*), falls es durch das Vertauschen der Kinder der P-Knoten bzw. durch das Umdrehen der Reihenfolgen der Q-Knoten möglich ist beide Bäume ineinander zu überführen.*

Diese beiden Definitionen formen zusammen den Begriff der konsistenten Frontiers eines *PQ-Baumes*.

Definition 4. *Für einen* PQ-tree T *ist* $cons(T)$ *definiert als die Menge aller Frontiers, welche aus allen zu* T *äquivalenten Bäumen gelesen werden können. Folglich gilt* $cons(T) := \{F(T') \mid T \cong T'\}$.

2.2 Die Consecutive Ones Property

Ein Problem für das *PQ-trees* eingesetzt werden ist das Nachweisen der *Consecutive Ones Property* (*COP*). Diese Eigenschaft einer $(0, 1)$-Matrix besagt, dass es für eine solche Matrix eine Permutation der Spalten gibt, so das in jeder Zeile der Matrix die Einsen jeweils hintereinander stehen. Beispielsweise erfüllt die folgende Matrix die COP:

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

Es ist möglich diese Eigenschaft noch auf eine andere Art zu beschreiben. Dazu betrachtet man ein Universum U und eine Menge C, welche Teilmengen von U

enthält. Ziel ist es nun eine Permutation aller Elemente aus U zu finden, so dass für jedes $S \in C$ die Elemente aus S jeweils aufeinanderfolgend in der Permutation stehen.

Beispiel 1. *Gegeben sei* $U = \{A, B, C, D, E, F, G, H\}$ *und* $C = \{\{A, C\}, \{A, F\}, \{D, E, G\}, \{G, F\}\}$. *Eine Permutation welche die COP erfüllt wäre beispielsweise* $\pi = BDEGFACH$, *da jede Menge aus* C *zusammenhängend in* π *vorkommt.*

Formalisiert lässt sich dies wie folgt beschreiben:

Definition 5. *Für ein endliches Alphabet* Σ *und eine Menge von Formeln* $\mathcal{F} = \{F_1, \ldots, F_k\} \subseteq 2^{\Sigma}$, *die man als Restriktionen nennt, wobei jede Menge in* $\mathcal{F}$ *eine Teilmenge von* Σ *ist, ist* $\Pi(\Sigma, \mathcal{F})$ *die Menge der Permutationen über* Σ, *so das in jeder Permutation die Elemente jedes* $F \in \mathcal{F}$ *aufeinanderfolgend vorkommen.*

Da ein *PQ-tree* alle solchen Permutationen aufzeigen soll, kann man sagen, dass für einen erfolgreich konstruierten Baum T

$$cons(T) = \Pi(\Sigma, \mathcal{F})$$

gelten muss.

2.3 Beispiel für einen PQ-tree

Nun soll ein *PQ-tree* anhand eines Beispiels präsentiert werden. Der unter Abbil-

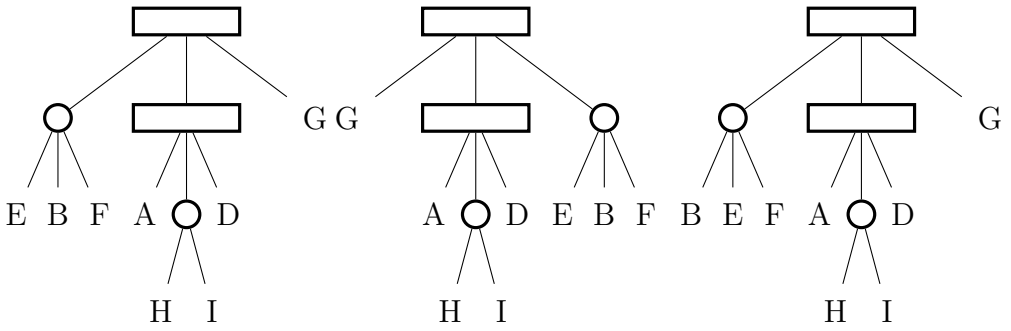

Abbildung 2: (a) Beispiel für einen *PQ-tree*, (b) selber Baum mit vertauschter Reihenfolge am Q-Knoten, (c) Beispiel mit vertauschten Kindern eines P-Knotens

dung 2(a) abgebildete Baum T hat das Frontier $EBFAHIDG$. Bei ihm wäre es nun z. B. möglich die Reihenfolge des als Wurzel fungierenden Q-Knotens umzudrehen, wodurch der äquivalente Baum einen Frontier von $GAHIDEBF$ haben würde (2(b)). Zudem könnte man die Reihenfolge des linken P-Knoten des Ursprungsbaumes frei umordnen, z. B. zu BEF. Nach einer solchen Operation, wäre

der neue Frontier $BEFAHIDG$ (2(c)). Würde man nun alle möglichen Vertauschungen durchführen, so würde man die Menge $\{EBFAHIDG, GAHIDEBF, BEFAHIDG, FEBAHIDG, \dots\}$ erhalten, welche gleichzeitig die Menge aller Frontiers, aller äquivalenter Bäume darstellt. Diese Menge ist bekannterweise $cons(T)$.

2.4 Aufbau eines PQ-trees

Die Konstruktion eines *PQ-trees* erfolgt iterativ. Dabei erzeugt man nacheinander eine Folge von *PQ-Bäumen* $T_0, \dots, T_k$ für die Restriktionen $F_1, \dots, F_k$, so dass jeweils gilt:

$$cons(T_i) = \Pi(\Sigma, \{F_1, \dots, F_i\})$$

Der zuerst erzeugte Baum T_0 repräsentiert dabei alle Permutationen, welche sich über dem Eingabealphabet Σ bilden lassen. Dies bedeutet also, dass der entsprechende *PQ-tree* aus einem P-Knoten besteht, der für jedes $a \in \Sigma$ ein Kind besitzt, welches eindeutig mit a markiert ist.

Das iterative Vorgehen beginnt zunächst mit dem Baum T_0, anschließend werden nacheinander die Restriktionen auf den Baum angewandt, so dass dieser keine der auf ihn angewandten Restriktionen verletzt. Dieser Vorgang wird durch die Prozedur `reduce` realisiert, wobei immer $T_i = \texttt{reduce}(T_{i-1}, F_i)$ gelten soll. Sie beginnt dabei bei den Blättern des aktuellen Baumes. Blätter die mit einem Zeichen beschriftet sind, welches in F_i vorkommt, heißen *voll*, ansonsten *leer*. Entsprechende Namensgebung setzt sich für die Knoten des Baumes fort. Ein Knoten wird als *voll* bezeichnet, falls alle seine Nachfolger ebenfalls als *voll* gelten. Entsprechend wird ein Knoten als *leer* bezeichnet, falls alle seine Nachfolger auch *leer* sind. Knoten die beide Arten von Knoten in dem von ihnen gebildeten Teilbaum enthalten werden *partiell* genannt.

Erstes Ziel der Prozedur ist es nun, einen Knoten $r(T_{i-1}, F_i)$ zu finden, der die niedrigste Wurzel eines Teilbaums T_i ist, der alle markierten Blätter enthält. Der Teilbaum $T_r(T_{i-1}, F_i)$, der aus diesem Knoten und seinen Nachfahren gebildet wird, heißt *reduzierter Teilbaum*.

Die Idee beim Anwenden der Prozedur besteht nun darin, dass man, ausgehend von den Blättern des reduzierten Teilbaums, sich bottom-up hinauf bis zur Wurzel arbeitet und bei jedem Schritt auf eine vorher definierte Schablone zurückgreift, wobei man jedes mal davon ausgeht, dass alle Nachfahren des gerade betrachteten Knotens bereits mittels dieser Methode *abgearbeitet* wurden.

Diese Schablonen sollen im nächsten Abschnitt genauer beschrieben werden. Dabei wird die aus Abbildung 1 bekannte Darstellung der Knoten verwendet. Zudem werden für Teilbäume Dreiecke als Schematisierung verwendet. Schwarze Knoten und Teilbäume gelten dabei als voll, graue als partiell und weiße als leer. Da die Reihenfolge der Kinder eines P-Knotens beliebig ist, soll für sie gelten, dass alle leeren Nachfahren gemeinsam links und alle vollen Nachfahren

zusammen rechts in der Skizze stehen.

2.4.1 Schablonen

An dieser Stelle folgen nun die bereits erwähnten Schablonen, die bei der bottom-up Abarbeitung des Baumes Anwendung finden. Hierbei wird der Knoten der aktuell untersucht wird als *betrachteter Knoten* bezeichnet. Als erstes werden nun die Regeln für den Fall vorgestellt, dass der betrachtete Knoten ein P-Knoten ist.

Schablone P_0

Die Schablone P_0, welche unter Abbildung 3 skizziert wird, zeigt an, dass man ausgehend von einem unmarkierten P-Knoten, der nur leere Nachfahren hat, nichts am Baum verändern muss und einfach den selbigen weiter aufsteigen kann.

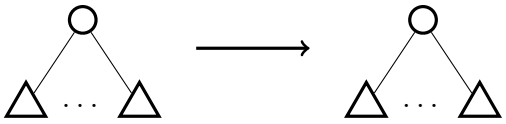

Abbildung 3: Schablone P_0

Schablone P_1

Die zweite Schablone P_1 (Abbildung 4) beschreibt den Fall, in dem beide Teilbäume unter einem P-Knoten voll sind, der betrachtete Knoten selbst es aber nicht ist. In diesem Fall muss dieser ebenfalls als voll markiert werden, bevor man den Baum weiter hinaufsteigt.

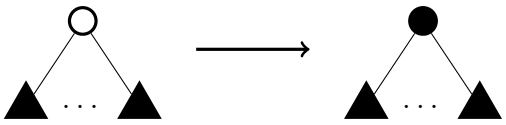

Abbildung 4: Schablone P_1

Schablone P_2

Als nächstes soll die Schablone P_2 betrachtet werden, welche unter Abbildung 5 skizziert dargestellt wird. Bei dieser Schablone wird angenommen, dass der betrachtete P-Knoten die Wurzel des reduzierten Teilbaums $T_r(T, F)$ ist und dieser jeweils nur leere oder volle, aber keine partiellen Teilbäume als Kinder hat. Um alle Restriktionen einzuhalten ist es nun von Nöten, die markierten

Teilbäume so miteinander zu verknüpfen, dass sie immer konsekutiv sind. Dazu ist es nötig den vorliegenden Teilbaum so zu verändern, dass man nur noch die vollen Bäume untereinander vertauschen kann, bzw. diese zusammen zwischen die leeren Teilbäume mischt. Dies geschieht durch einen neuen vollen P-Knoten, welcher nun Elter der vollen Teilbäume wird und selbst ein Kind des betrachteten Knotens ist. An dieser Stelle ist zudem zu beachten, dass man diese Schablone nur anwenden darf, falls es mehr als einen vollen Teilbaum gibt, der an der Wurzel hängt, da man ansonsten die Definition eines *PQ-trees* verletzten würde. Tritt dieser Sonderfall auf, so belässt man die Datenstruktur wie sie war.

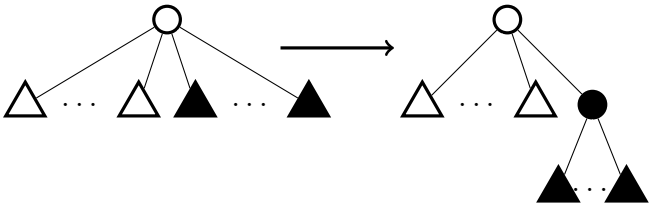

Abbildung 5: Schablone P_2

Schablone P_3

Die unter Abbildung 6 gezeigte Schablone P_3 entspricht von der Ausgangslage her der Schablone P_2, jedoch geht man hier, im Gegensatz zur vorherigen Regel, davon aus, dass der betrachtete Knoten nicht die Wurzel des reduzierten Teilbaums ist. Bei der Umwandlung wird nun ein neuer Q-Knoten geschaffen, welcher als partiell markiert wird. Nun folgt eine Aufteilung der Teilbäume, welche zuvor an dem betrachteten Knoten hingen. Zu einem wird dieser mit allen seinen leeren Teilbäumen ein Kind des neugeschaffenen Q-Knotens, zum anderen wird ein neuer P-Knoten geschaffen, welcher als voll markiert wird und alle vollen Teilbäume des betrachteten Knotens als Kinder bekommt. An dieser Stelle muss erneut darauf geachtet werden, dass man die Definition eines *PQ-trees* nicht verletzt. So wird kein neuer P-Knoten geschaffen, falls es nur einen vollen bzw. leeren Teilbaum gibt. Anders verhält es sich mit dem neu geschaffenen Q-Knoten, welcher laut Definition nicht legitim wäre, da ein solcher Knoten immer mindestens drei Kinder hat. An dieser Stelle soll dies jedoch erlaubt sein, da man im weiteren Verlauf der Prozedur diesen Fehler wieder beseitigt.

Schablone P_4

Die fünfte Schablone P_4, welche unter Abbildung 7 illustriert wird, enthält nun den Fall, dass der betrachtete Knoten die Wurzel des reduzierten Teilbaums ist und neben den bisher betrachteten leeren und vollen Teilbäumen diesmal auch ein partieller vorkommt. Da auch im weiteren Teil der Konstruktion nur partielle

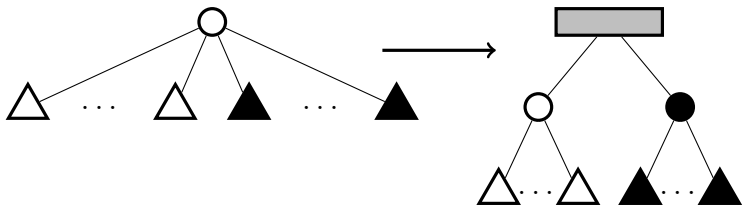

Abbildung 6: Schablone P_3

Q-Knoten erstellt werden, ist dieser partielle Knoten auf jeden Fall ein Q-Knoten. Der Umwandlungsschritt an dieser Stelle besteht nun darin, dass man alle vollen Teilbäume der Wurzel des reduzierten Teilbaums unter einen neuen P-Knoten hängt und diesen Knoten samt seiner Kinder an die entsprechende Stelle des partiellen Knotens anfügt. Hierbei ist erneut zu beachten, dass das Erstellen eines neuen Knotens nur erforderlich ist, falls es mindestens zwei volle Kinder der Wurzel gibt. Zudem muss darauf geachtet werden, dass die Reihenfolge der Kinder gewahrt bleibt, da es nicht möglich ist die Kinder des Q-Knotens beliebig zu vertauschen.

An dieser Stelle soll außerdem erwähnt werden, dass mithilfe dieser Schablone die Probleme aus P_3 bzgl. der zu geringen Kinderanzahl eines Q-Knotens gelöst werden können. Hatte ein solcher Knoten bis zu dem Anwenden dieser Schablone nicht die geforderte Kindesanzahl, so bekommt er durch diesen Schritt ein zusätzliches Kind, wodurch er nun die geforderte Mindestanzahl erreichen würde.

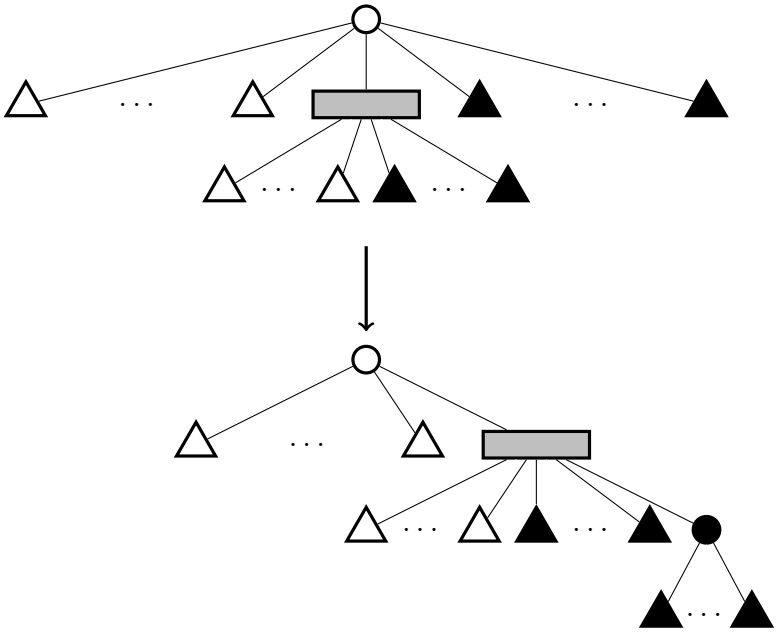

Abbildung 7: Schablone P_4

Schablone P_5

Die Schablone P_5 (Abbildung 8) behandelt nun den zu P_4 analogen Fall in dem der betrachtete Knoten nicht die Wurzel des reduzierten Teilbaums ist. Um die aktuelle Restriktion im Baum zu verankern ist es nun nötig die jeweiligen vollen und leeren Kinder des aktuellen Knotens unter einem neuen P-Knoten gleichen Typs zu vereinen. Dies ist erneut nur nötig, falls es mehr als einen Teilbaum einer Art gibt. Unter Einhaltung der Reihenfolge ist es nun möglich diese beiden Teilbäume zu Kindern des partiellen Q-Knotens zu machen, welcher nun an die Stelle des vorherigen rückt und diesen ersetzt.

Erneut lässt sich feststellen, dass ein durch die Schablone P_3 entstandener Q-Knoten durch diese Regel mindestens zwei neue Kinder bekommt, wodurch er nun der Definition eines solchen Knotens entspricht.

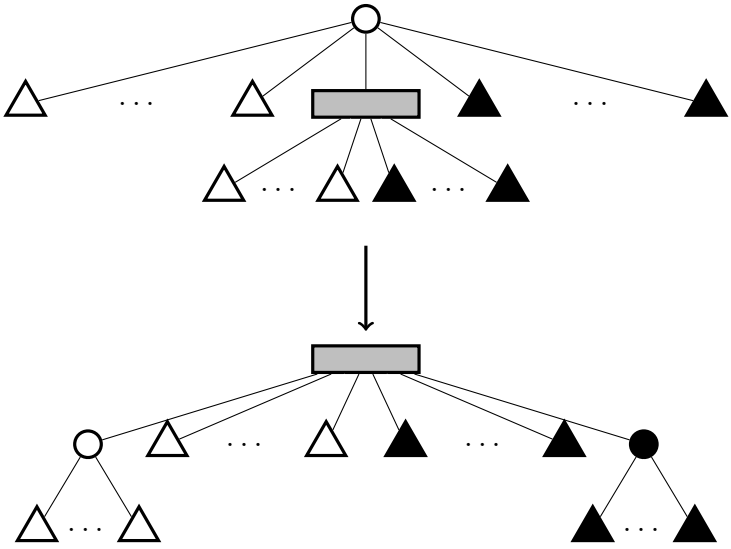

Abbildung 8: Schablone P_5

Schablone P_6

Bei der letzten Regel für die Umformung eines P-Knotens geht man erneut davon aus, dass der betrachtete Knoten die Wurzel des reduzierten Teilbaums sein muss, da es andernfalls nicht möglich wäre einen korrekten *PQ-tree* zu konstruieren. Der Grund hierfür liegt in der Struktur des betrachteten Falls, welche in Abbildung 9 aufgezeigt wird. Wäre der betrachtete Knoten nicht die Wurzel des reduzierten Teilbaums, so wäre es aufgrund der Q-Knoten nicht möglich die vollen Blätter konsekutiv anzuordnen. Die Schablone P_6 zeigt nun die Möglichkeit zwei partielle Q-Knoten miteinander zu verschmelzen. Hierfür wird ein Q-Knoten verwendet, der links bzw. rechts die Kinder der ursprünglichen Q-Knoten angehangen bekommt, wobei die Reihenfolge unbedingt eingehalten werden muss. Die entspre-

chenden markierten Teilbäume des betrachteten Knotens werden, falls nötig, zu Kindern eines als voll markierten P-Knotens. Dieser Knoten wiederum wird nun selbst Kind des neuen Q-Knotens, wobei erneut darauf zu achten ist, dass die Reihenfolge der Knoten korrekt bleibt.

Erneut sieht man leicht, dass an dieser Stelle Q-Knoten, die aus P_3 entstanden sind, hier durch genügend neue Kinder *aufgestockt* werden, so dass sie ab diesem Zeitpunkt der Definition entsprechen.

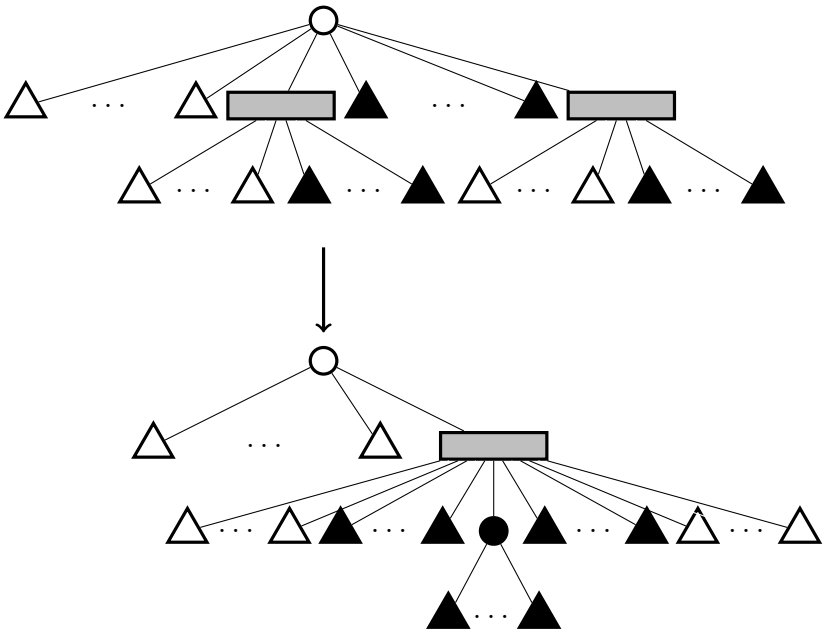

Abbildung 9: Schablone P_6

Dies war die letzte Schablone für P-Knoten. Nun folgen fünf weitere Regeln, die für Q-Knoten verwendet werden können.

Schablone Q_0

Die erste Schablone Q_0 für Q-Knoten (Abbildung 10) funktioniert analog zur Regel P_0. Sind alle Kinder des betrachteten Knotens leer, so ist dieser auch leer und man kann den Baum weiter hinaufsteigen, ohne etwas ändern zu müssen.

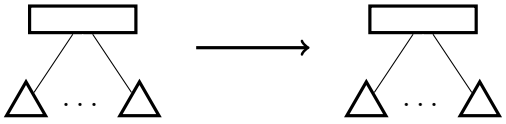

Abbildung 10: Schablone Q_0

Schablone Q_1

Auch die Schablone Q_1, welche unter Abbildung 11 skizziert wurde, lässt sich analog zu den Regeln für die P-Knoten anwenden. Betrachtet man einen Knoten, welcher nur volle Kinder besitzt, so gilt dieser Knoten laut Definition ebenfalls als voll. Somit muss dieser Knoten markiert werden und die bottom-up Bearbeitung des Baumes kann fortgesetzt werden.

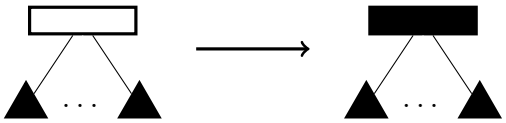

Abbildung 11: Schablone Q_1

Schablone Q_2

Hängen an einem Q-Knoten, wie in Abbildung 12, sowohl volle als auch leere Teilbäume, so genügt es für diesen Teilbaum den betrachteten Knoten als partiell zu Kennzeichnen und den Baum weiter hinaufzusteigen.

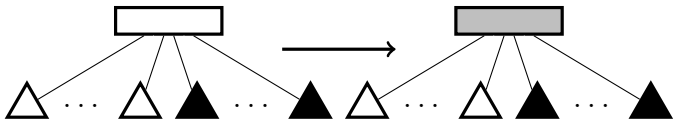

Abbildung 12: Schablone Q_2

Schablone Q_3

Die unter Abbildung 13 illustrierte Schablone Q_3 wird dann benutzt, wenn ein partieller Q-Knoten Kind des betrachteten Knotens ist und dieser Knoten zudem volle als auch leere Teilbäume als Nachfahren hat. Die Idee bei dieser Schablone liegt nun darin, dass man die beiden vorliegenden Q-Knoten zu einem neuen partiellen Q-Knoten verschmilzt, wobei man die Reihenfolge der einzelnen Kinder berücksichtigen muss.

Schablone Q_4

Die letzte Regel Q_4 (Abbildung 14) bezieht sich auf einen betrachteten Knoten welcher gleichzeitig die Wurzel des reduzierten Teilbaums ist. Dieser Knoten enthält in diesem Fall zwei partielle Q-Knoten. Diese Schablone besagt nun, dass man diese beiden Knoten entfernen kann und ihre Kinder unter Beachtung der Reihenfolge an den betrachteten Knoten hängen darf, welcher nun als partiell

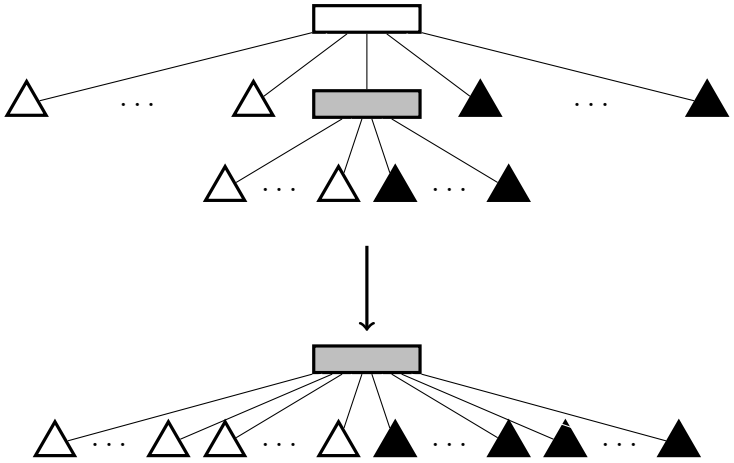

Abbildung 13: Schablone Q_3

gilt. Entsprechende aus P_3 stammende Q-Knoten würde an dieser Stelle zu einem, der Definition nach, gültigen Q-Knoten verschmolzen. Abschließend sei an dieser Stelle angemerkt, dass ein Anwenden dieser Schablone keine weiteren Schablonenanwendungen mehr erlaubt.

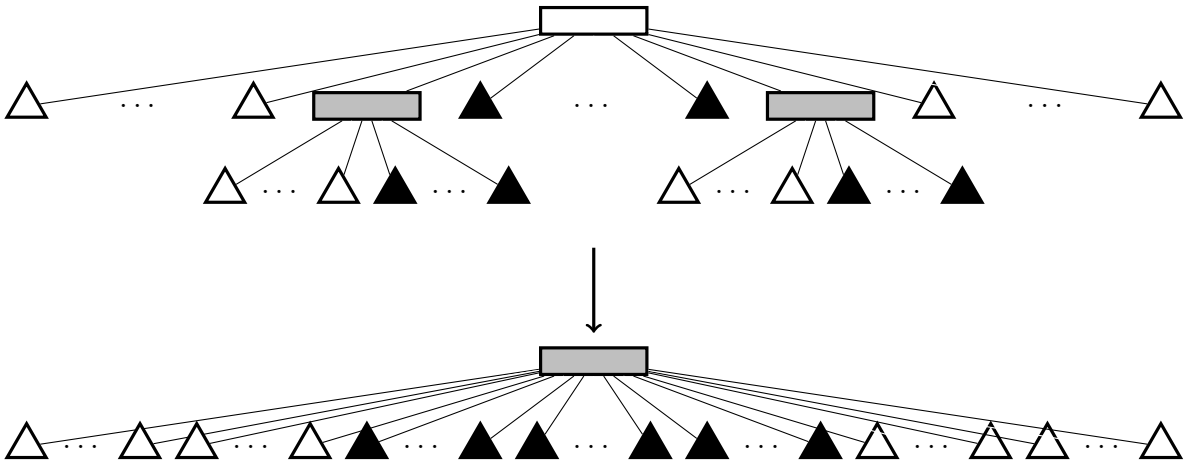

Abbildung 14: Schablone Q_4

An dieser Stelle sei explizit angemerkt, dass eine solche Konstruktion nur dann möglich ist, falls die Eingabe die *COP* erfüllt. Unter Abbildung 15 kann man die Konstruktion eines *PQ-trees* für die Restriktionen

$$\{\{A, B, C\}, \{B, E\}, \{A, C, D\}\}$$

sehen.

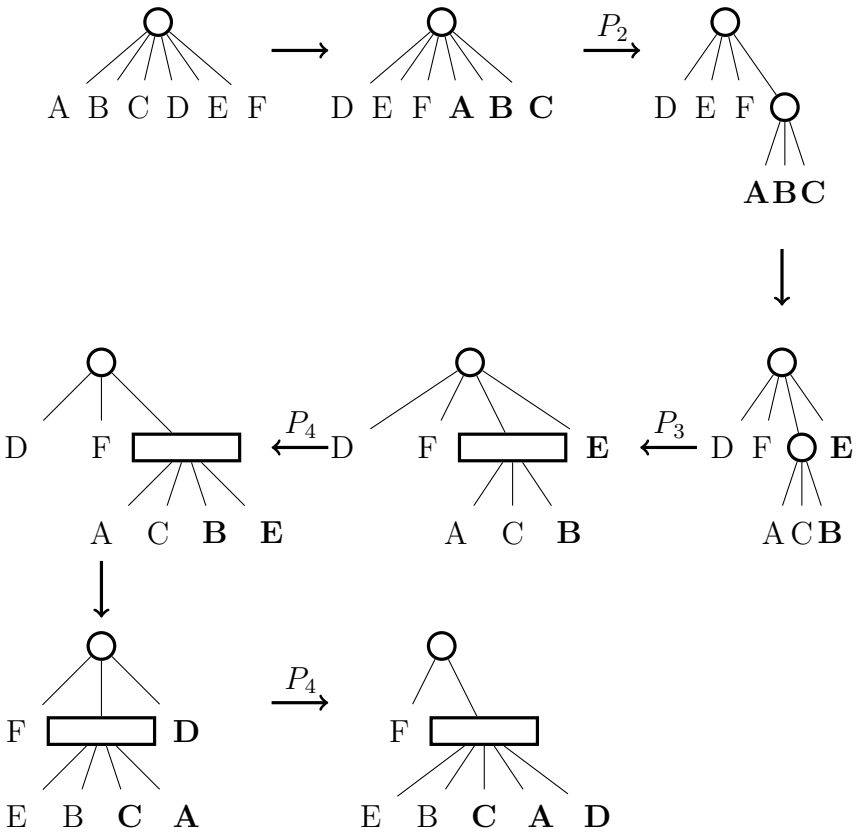

Abbildung 15: Beispiel für den Aufbau eines *PQ-trees*

2.5 Abschließender Kommentar

Anhand der Konstruktion des *PQ-trees* konnte man erkennen, dass die Prozedur zum Erstellen des selbigen äußerst kompliziert verläuft. Ebenso ist ihre Analyse äußerst komplex und würde den Rahmen dieser Arbeit sprengen, weshalb an dieser Stelle die Ergebnisse dieser kurz vorgestellt werden sollen. Für eine genau Laufzeitanalyse sei also auf die Literatur verwiesen.

Bei einer effizienten Implementierung der Datenstruktur, bzw. der `reduce` Operation, besteht die Möglichkeit nicht nur innerhalb des reduzierten Teilbaums zu arbeiten, und somit die Anzahl der betrachteten Knoten zu verringern, sondern es ist auch möglich die Menge der betrachteten Knoten weiter zu verkleinern. Hierzu betrachtetet man den sog. *relevanten reduzierten Teilbaum* $T_{rr}(T, F)$, welcher ein Teilgraph des reduzierten Teilbaums $T_r(T, F)$ ist. Dieser Baum $T_{rr}(T, F)$ ist dabei der minimale Teilgraph von T, der alle markierten Blätter aus F enthält. Man kann zeigen, dass fast die komplette Arbeit von `reduce` sich auf diesen Teilgraphen bezieht, was zu einer Laufzeitabschätzung von

$$\sum_{i=1}^{n} O\left(|T_{rr}\left(T_{i-1}, F_i\right)|\right)$$

für eine Menge von Restriktionen $\mathcal{F} = \{F_1, \ldots, F_n\}$ führt. Des Weiteren lässt sich zeigen, dass man die Kosten für das Anwenden der Schablonen so auf die Knoten des relevanten reduzierten Teilbaums verteilen kann, dass diese für jeden Knoten dieses Graphen konstant sind. Hierbei sei angemerkt, dass die Anzahl der inneren Knoten von T_{rr} gleich der Zahl angewendeter Schablonen ist. Diese Zeit lässt sich für den Fall einer erfolgreichen Konstruktion mit

$$O\left(\sum_{F \in \mathcal{F}} |F|\right)$$

abschätzen. Ist eine Konstruktion dagegen nicht erfolgreich, so benötigt man noch $O(|\Sigma|)$ Schritte um dies nachzuweisen, was eine worst-case Laufzeit von

$$O\left(|\Sigma| + \sum_{F \in \mathcal{F}} |F|\right)$$

bedeutet.

Abschließend lässt sich feststellen, dass das Lösen des *Consecutive Ones Property* mit Hilfe eines *PQ-trees* sehr komplex ist. Das Zurückgreifen auf Schablonen bei der Generierung eines solchen Baumes ist ein enorm komplexes Unterfangen und ist nicht nur bei der Analyse ein großes Problem, sondern erschwert die Implementierung dieser Datenstruktur ungemein. Der nun folgende Teil dieser Arbeit wird sich mit der Frage beschäftigen, ob es möglich ist eine Datenstruktur zu entwickeln, die diese Aufgaben einfacher bewältigt und dabei sogar effizienter arbeitet.

3 Der PC-tree

Der nun folgende Teil dieser Arbeit beschäftigt sich mit *PC-trees*, einer Datenstruktur, welche z. B. alle Permutationen einer Matrix darstellen soll, die die sog. *Circular Ones Property* ($CROP$) erfüllen. Die Besonderheit dieser Bäume liegt darin, dass sie in die vergleichsweise weniger bekannte Kategorie der *ungewurzelten Bäume* fallen. Zunächst soll diese Datenstruktur definiert und die Parallelen zum *PQ-tree* aufgezeigt werden. Anschließend folgt ihre Konstruktion zusammen mit einer kurzen Analyse und einigen Ideen zur Implementierung.

3.1 Definition eines PC-trees

Die Definition eines *PC-trees* unterscheidet sich kaum von der eines *PQ-Baumes*. Es gibt jedoch zwei signifikante Unterschied, so handelt es sich bei einem *PC-tree* um einen ungewurzelten Baum, der aus P- und C-Knoten und Blättern besteht. Die Besonderheit eines ungewurzelten Baumes besteht darin, dass es zwischen zwei verbundenen Knoten keine feste Rangfolge gibt, was das Erstellen

eines solchen Baumes vereinfachen wird. Die P-Knoten und die Blätter haben dabei die selbe Bedeutung wie bei einem *PQ-tree*, C-Knoten unterscheiden sich allerdings von den Q-Knoten. Im Gegensatz zu den Q-Knoten darf man ihre Kinder nicht nur in ihrer Reihenfolge umdrehen, zusätzlich darf man die Kinder zyklisch verschieben. In [HM] werden diese Möglichkeiten mit denen einer Münze verglichen, welche sowohl beliebig im Kreis gedreht, als auch umgedreht (*geflippt*) werden darf. Weiterhin sei erwähnt, dass $F(T)$ und $cons(T)$ die selbe Bedeutung für einen *PC-tree* haben wie bei einem *PQ-tree* .

Abbildung 16 zeigt einen *PC-tree* wobei durchgestrichene Kreise C-Knoten repräsentieren. Die Frontier dieses Baumes wäre $EFAHIDG$. Würde man nun die Möglichkeit nutzen und die Kinder der Wurzel rotieren lassen, so wäre beispielsweise ein Frontier $AHIDGEF$ möglich.

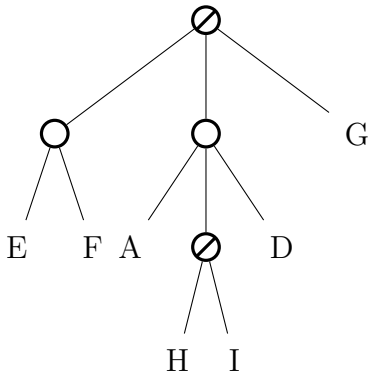

Abbildung 16: Beispiel für einen *PC-tree*

3.2 CROP und COP mit PC-trees

Die Eigenschaft einer Matrix, die durch einen *PC-tree* repräsentiert wird, nennt man *Circular Ones Property* (*CROP*). Sie besagt, dass durch ein Vertauschen der Spalten einer $(0, 1)$-Matrize die 0- oder die 1-Werte jeder Spalte konsekutiv vorkommen. Man kann dies auch durch die *COP* ausdrücken, so hat eine Matrix die *CROP*, falls die Matrix selbst die *COP* erfüllt oder dieses für die komplementäre Matrix gilt, in der Nullen und Einsen vertauscht werden.

Es gibt eine Möglichkeit einen *PC-tree* in einen *PQ-tree* zu überführen und so die *COP* zu zeigen. Hierzu fügt man der betrachteten Matrix eine neue Spalte x hinzu, welche dem Nullvektor entspricht. Anschließend berechnet man für diese Matrix einen PC-Baum. Aus dieser Datenstruktur generiert man nun einen PQ-Baum, indem man die C-Knoten durch Q-Knoten ersetzt und den Knoten zu dem das Blatt x adjazent ist als Wurzel des neuen *PQ-trees* ansieht. Abbildung 17 zeigt ein Beispiel für eine solche Umwandlung. Für den Korrektheitsbeweis soll an dieser Stelle auf [HM03] verwiesen werden.

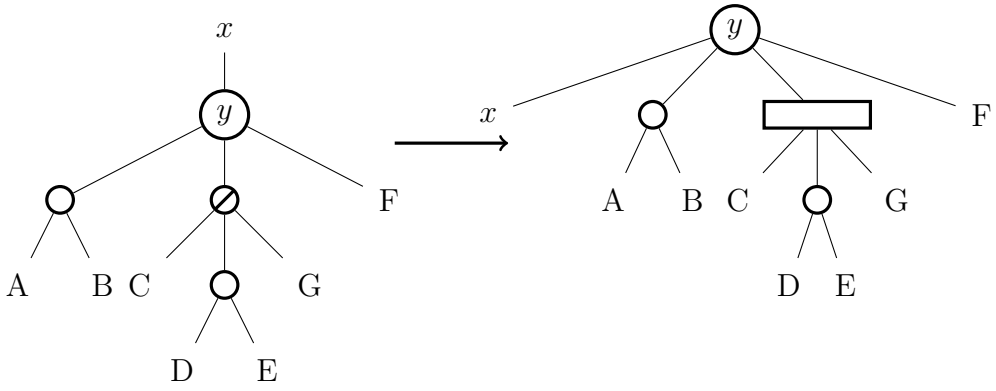

Abbildung 17: Beispiel für die Umwandlung eines *PC-trees* zu einem *PQ-tree*, wobei y die Wurzel dieses Baums ist

3.3 Konstruktion eines PC-trees

Nachdem die Bedeutungen der einzelnen Knotenarten eines *PC-trees* erläutert wurden, soll dieser nun aus einer $(0, 1)$-Matrix konstruiert werden. Dazu soll zuerst der Begriff des *circular module* definiert werden. Ein Menge ist ein *circular module*, wenn sie entweder ein *edge module* oder ein *P module* ist. Beides beschreibt dabei Blattmengen, die die Blätter jeweils eines Teilbaumes enthalten. Im Falle eines *edge modules* entstand dieser Teilbaum durch das Entfernen einer Kante und im Falle eines *P modules* durch das Enfernen eines P-Knoten.

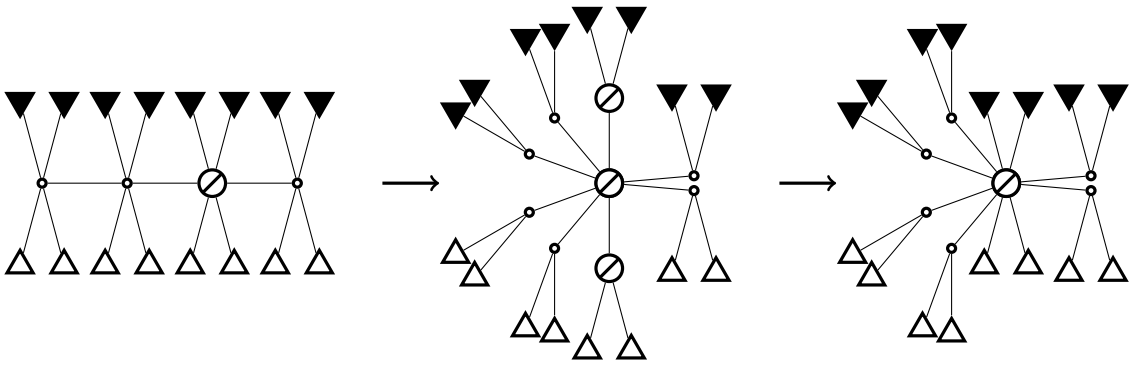

Abbildung 18: Skizze der Einarbeitung einer Restriktion in einen *PC-tree*

Die nun folgende Vorgehensweise zur Generierung des Baumes, die unter 18 skizziert wird, ähnelt der bereits für *PQ-trees* vorgestellten Methode. Man beginnt erneut mit einem P-Knoten, der alle Elemente des Eingabealphabets als Kinder zugewiesen bekommt. In einem zweiten Schritt wird anschließend die erste Zeile der Eingabematrix in den Baum eingearbeitet. Dabei entsteht anschließend ein Baum mit zwei P-Knoten, wobei ein Knoten alle vollen und ein Knoten alle leeren Blätter als Kinder zugewiesen bekommt (Definition für *voll* und *leer* sie-

he Abschnitt 2.4). Nun folgt, wie bei den *PQ-trees* zuvor auch, ein schrittweiser Aufbau des Baumes, wobei in jedem Schritt jeweils eine weitere Zeile der Matrix in die Datenstruktur eingebaut wird. Hierzu werden zunächst die entsprechenden Blätter, die den 1-Werten der aktuell betrachteten Zeile der Eingabematrix entsprechen, als voll und die 0-Werte als leer markiert. Durch diesen Schritt kann es passieren, dass es circular modules gibt, die sowohl Einsen als auch Nullen enthalten. In der nachfolgenden Betrachtung geht man davon aus, dass ein solcher Fehler erst durch das entsprechende Markieren generiert wurde. Wurde ein edge module auf dieser Weise ungültig, d. h. dass es selbst und der komplementäre Teilbaum sowohl volle als auch leere Blätter enthält, so muss dies wieder *repariert* werden. Das Entfernen einer solchen Kante würde also den *PC-tree* in zwei Teile trennen, wobei beide sowohl volle als auch leere Blätter enthalten. Eine solche Kante nennt man *terminal edge*. Hat eine Eingabematrix die *CROP*, so kann man sagen, dass alle diese terminal edges zusammen einen Pfad, den sog. *terminal path*, bilden (Beweis siehe [HM03]), dessen Endknoten *terminal nodes* genannt werden. An dieser Stelle sei zudem angemerkt, dass es auch einen terminal path der Länge null geben kann. Folgendes Lemma aus [HM03] hilft nun bei der Einarbeitung der neuen Zeile:

Lemma 1. *Jeder Knoten, der aufgrund eines Hinzufügens einer Zeile angepasst werden muss, liegt auf dem terminal path.*

Somit reicht es völlig aus einen terminal path zu finden und die Knoten auf folgende Art und Weise anzupassen:

1. Ordne die jeweiligen Kinder eines Knotens so, dass alle vollen und alle leeren Blätter jeweils *zusammen* auf einer Seite des Pfades liegen

2. Spalte den Knoten in zwei auf, wobei die Kinder des einen nur volle und die des anderen nur leere Teilbäume sind

3. Ersetze alle Knoten auf dem terminal path durch einen C-Knoten x unter Einhaltung der Reihenfolge

4. Verschmelze alle zu x adjazenten C-Knoten und Knoten mit nur zwei Nachbarn mit x

Für die Korrektheit dieses Vorgehens sei erneut auf die Literatur verwiesen.

3.4 Effiziente Implementierung und Analyse

Nun soll die im vorherigen Abschnitt beschriebene Idee konkretisiert werden. Dazu soll eine Möglichkeit der Implementierung aufgezeigt werden, welche gleichzeitig dazu genutzt wird um eine Laufzeitanalyse durchzuführen.

3.4.1 Implementierung der Datenstruktur PC-tree

Zunächst soll die Repräsentation der Datenstruktur vorgestellt werden. Da ein *PC-tree* ein Baum ist, hat dieser bei n Knoten $n-1$ Kanten. Jede dieser $n-1$ Kanten wird dabei exakt einem der adjazenten Knoten zugeordnet. Dies kann durch eine gewurzelte Baumstruktur realisiert werden, wobei ein beliebiger Knoten als Wurzel fungiert und so von diesem ausgehend eine Eltern-Kind Beziehung erstellt wird. Die jeweilige Kante zwischen den beiden Knoten wird dabei immer mit dem Elter assoziiert. Da es sich bei *PC-trees* um ungewurzelte Bäume handelt hat dies nur im Bezug auf die Effizienz eine Auswirkung, so kann man aufgrund dieser Struktur eine *low-level* Operation, wie z. B. die Überprüfung auf Adjazenz zweier Knoten, in konstanter Zeit ermöglichen.

Die einzelnen Elemente der Datenstruktur lassen sich wie folgt implementieren:

P-Knoten Ein P-Knoten enthält seine Elternkante.

Kante Eine Kante, welche in diesem Fall ungerichtet ist, wird jeweils durch zwei gerichtete Kanten, die *Bögen* genannt werden repräsentiert, wobei das Objekt der Kante diese beiden enthält.

Bogen Jeder Bogen (x, y) enthält je einen Zeiger auf die Nachbarn von y in der zyklischen Ordnung, einen Zeiger auf den Bogen (y, x), ein Bit, welches angibt ob y der Elter von x ist und im Falle dass y ein P-Knoten ist einen Zeiger auf y.

C-Knoten C-Knoten werden nicht explizit dargestellt, stattdessen wird die durch sie gegebene zyklische Ordnung durch eine doppelt-verkettete zyklische Liste ihrer inzidenten Kanten implizit repräsentiert. Da es laut Konstruktion nie zwei benachbarte C-Knoten geben kann zeigen diese Kanten eindeutig, dass es sich hierbei um einen solchen Knoten handelt. Hierbei ist festzustellen, dass falls man den Elter eines solchen Knotens sucht, man jeden adjazenten Knoten überprüfen muss, was nicht in $O(1)$ möglich ist.

Im Zuge der Konstruktion des Baumes kann es vorkommen, dass man zwei C-Knoten, unter Einhaltung der zyklischen Reihenfolge, verschmelzen muss. Dies geht, bei gegebenen Zeigern, in Zeit $O(1)$. Ebenso ist es in konstanter Zeit möglich Kanten aus der zyklischen Adjazenzliste eines Knotens zu entfernen bzw. hinzuzufügen.

3.4.2 Bestimmung des terminal path

Beim Finden des terminal paths greift man auf die jeweiligen Elterkanten zurück. Man geht dabei davon aus, dass die Blätter, abhängig von der aktuell betrachteten Zeile, als voll, bzw. als leer, markiert worden sind. Ausgehend von den Blättern führt man nun eine Markierung der übrigen Knoten durch. Dabei erhält

jeder Knoten einen Zähler, der die Anzahl der vollen Nachbarn mitprotokolliert. Immer wenn dieser Zähler für einen Knoten auf 1 gesetzt wird, wird der entsprechende Knoten als partiell markiert. Erreicht der Zähler eine Anzahl, die um eins kleiner ist als sein Grad, so gilt dieser Knoten als voll und der Zähler seines nicht-vollen Nachbarn wird um eins erhöht. Da jeder innere Knoten des Baumes mindestens Grad drei hat, kann man die Zahl innerer voller Knoten durch die Zahl der vollen Blätter k nach oben hin abschätzen. Zudem hat jeder volle Knoten mindestens einen partiellen Nachbarn wodurch man sagen kann, dass diese beiden Markierungsoperationen jeweils $O(k)$ Rechenschritte benötigen.

Beim finden der terminal edges nutzt man nun aus, dass diese immer zwischen zwei partiellen Knoten liegen müssen. Jedoch gibt es auch den Sonderfall eines terminal paths der Länge 0. Dieser lässt sich aber leicht finden, da dieser aus dem einzigen partiellen Knoten bestehen muss. Ist dieser Sonderfall nicht gegeben so versucht man zuerst einen sog. *Scheitel* zu finden. Damit ist ein Knoten gemeint, der der erste gemeinsame Vorfahre aller partiellen Knoten ist. Einen solchen Knoten kann man dadurch finden, dass man, ausgehend von allen partiellen Knoten, den Baum parallel hinaufsteigt und die jeweilige Elterkante besonders markiert. Trifft man dabei auf eine bereits markierte Kante so kann man aufhören diesen Pfad weiter zu verfolgen. Ziel ist es den letzten Knoten zu finden, in dem mehrere Pfade zusammentreffen. Dabei ist drauf zu achten, dass ein Pfad über den Scheitel *hinausgewachsen* ist. Entfernt man diese *überflüssigen* Markierungen so ist jede Kante die markierte wurde eine terminal edge. Somit wären die Kanten des terminal paths gefunden. An dieser Stelle sei angemerkt, dass aufgrund der Datenstruktur das *Aufsteigen* nicht trivial möglich ist. Zwar hat jeder P-Knoten einen Zeiger zu seinem Elter, wodurch dieser Schritt in $O(1)$ möglich ist, jedoch ist es weitaus schwerer den Elter eines C-Knoten in konstanter Zeit zu finden.

Abschließend kann man festhalten, dass das Finden des terminal paths in Zeit $O(p' + k)$ möglich ist, wobei p' die Anzahl der Knoten ist, die man beim Suchen des Scheitels *besucht* hat. Da diese Zahl höchstens so groß sein kann, wie die Länge des Pfades selbst, kann man sagen, dass $O(p' + k) = O(p + k)$ ist.

3.4.3 Hinzufügen einer neuen Restriktion

Fügt man eine neue Restriktion einem, bis dahin, korrekten *PC-tree* hinzu, so entspricht die Anzahl der vollen Nachbarn der Elemente auf dem terminal path, der Anzahl der Einsen in der entsprechenden Zeile der Eingabematrix. Diese Zahl sei k. Bevor man einen einzelnen Knoten auf dem Pfad aufspaltet, speichert man seine Nachbarn die auf dem selbigen liegen ab und löscht die entsprechenden Kanten in $O(1)$. Die Aufspaltung wird dadurch realisiert, dass man den aufzuspaltenden Knoten beibehält und an ihn nur die vollen Knoten hängt, während man einen zweiten Knoten erstellt, welcher die verbleibenden Nachbarn zugeordnet bekommt. Bezeichnet man die Länge des terminal paths mit p, so kann man die Laufzeit dieser Operation mit $O(p + k)$ abschätzen, da die Anzahl der

vollen Knoten durch k beschränkt ist. Zuletzt müssen alle Knoten entlang des betrachteten Pfades mit einem neuen C-Knoten verbunden werden, was in $O(p)$ möglich ist. Die anschließende Reparatur der Eltern-Kind Beziehung ist ebenfalls in $O(p)$ möglich. Hierbei wird der Scheitel zum Elter des neuen C-Knotens und alle aufgespaltenen Knoten werden wiederum zu dessen Kindern.

Als Ergebnis kann man also festhalten, dass das Hinzufügen einer Restriktion in Zeit $O(p + k)$ möglich ist.

3.4.4 Gesamtlaufzeit

Es soll nun mithilfe einer amortisierten Analyse gezeigt werden, dass die komplette Konstruktion eines *PC-trees* in linearer Zeit möglich ist, bezogen auf die Anzahl der Einsen in der Matrix. Zur Analyse wird dabei die Kontenmethode verwendet, wobei wir folgende Potentialfunktion definieren:

$$\phi(M, i) = u_i + |C_i| + \sum_{x \in P_i} (deg(x) - 1)$$

u_i steht dabei für die Anzahl der Eins-Einträge in der i-ten Zeile der Matrix, $deg(x)$ für den Grad eines Knotens x, C_i für die Menge der C- und P_i für die Menge der P-Knoten. Jede Operation, die den Wert von ϕ verringert, bucht dafür einen entsprechenden Wert vom Konto ab, welches den Startwert $\phi(M, 0) = \Theta(m)$ bekommt, wobei m die Anzahl der Einsen in der Matrix M ist. Nun gilt es zu zeigen, dass es möglich ist den kompletten Baum aufzubauen, ohne dass der Kontostand jemals negativ wird. Wenn dies gelingt, so kann man von einer linearen Laufzeit ausgehen. Aus Abschnitt 3.4.2 und 3.4.3 ist bekannt, dass man für das Hinzufügen einer Zeile jeweils $O(p + k)$ Operationen benötigt. Da die k nötigen Operationen jeweils für die 1-Werte einer Zeile stehen, können diese direkt bezahlt werden. Jeder C-Knoten auf dem terminal path wird der Einarbeitung einer Matrixzeile zuerst gespalten, dann aber wieder kontrahiert, wobei kein Grad eines P-Knoten verändert wird. Liegt ein P-Knoten auf dem terminal path, aber nicht an dessen Ende, und wir er aufgespalten so verändert sich die Summe der beiden Grade nicht, jedoch entstehen dadurch zwei Knoten, was aufgrund der Formel eine Einheit zur Verfügung stellt. Enthielt ein P-Knoten vorher nur Nachbarn eines Typs, so wird er nicht aufgespalten und durch die Umformung seiner Nachbarschaft verringert sich sein Grad um 1, was erneut einen freigewordenen Credit bedeutet. Die einzigen Knoten für die zusätzliche Kosten nötig sind, sind die terminal nodes, die jeweils einen terminal path begrenzen. Deren Anzahl ist jedoch durch $O(1)$ begrenzt, so dass dies kein Problem darstellt. Die Kontraktion zweier Knoten von Grad zwei lässt auch eine Einheit freiwerden wodurch diese Operation sich selbst bezahlt.

Somit sieht man, dass es möglich ist die $O(p + k)$ Kosten für jede Restriktion zu bezahlen. Damit benötigt die gesamte Konstruktion des Baumes amortisiert $O(m)$ Zeit.

4 Fazit

Anhand der Betrachtung der beiden hier vorgestellten Datenstrukturen kann man erkennen, dass es von Vorteil sein kann, Probleme teilweise über einen *Umweg* anzugehen.

Während die *PQ-trees* noch eine starre gewurzelte Baumstruktur verwenden, nutzten *PC-trees* den Vorteil eines ungewurzelten Baumes. Dies ermöglicht eine Methode, welche anstatt auf zwölf unterschiedliche Schablonen zurückzugreifen, die komplette Konstruktion des Baumes auf einen Fall reduziert. Durch die vorgestellte Transformation eines *PC-trees* in einen *PQ-tree* ist es möglich den so gewonnenen Laufzeitvorteil für beide Datenstrukturen zu nutzen. Bemerkenswert ist dabei, dass die neue Methode das Verfahren zum Erzeugen des entsprechenden Baumes stark vereinfacht, was angesichts der praktischen Anwendung der Methode eine nicht ganz unbedeutende Rolle spielt.

5 Literatur

Literatur

[BL76] Booth, Kellogg S. ; Lueker, George S.: Testing for the Consecutive Ones Property, Interval Graphs, and Graph Planarity Using PQ-Tree Algorithms. In: *J. Comput. Syst. Sci.* 13 (1976), Nr. 3, S. 335–379

[Heu06] Heun, Volker: *Skriptum zur Vorlesung Algorithmische Bioinformatik: Bäume und Graphen.* `http://www2.bio.ifi.lmu.de/~heun/lecturenotes/cb3ter.pdf`, 2006

[HM] *Kapitel* 1. In: Hsu, Wen-Lian ; McConnell, R.: *Handbook of Data Structures and Applications*

[HM03] Hsu, Wen-Lian ; Mcconnell, Ross M.: PC trees and circular-ones arrangements. In: *Theoretical Computer Science* 296 (2003)

[Hsu01] Hsu, Wen-Lian: PC-Trees vs. PQ-Trees. In: *COCOON '01: Proceedings of the 7th Annual International Conference on Computing and Combinatorics.* London, UK : Springer-Verlag, 2001. – ISBN 3–540–42494–6, S. 207–217

[SH99] Shih, Wei-Kuan ; Hsu, Wen-Lian: A new planarity test. In: *Theor. Comput. Sci.* 223 (1999), Nr. 1-2, S. 179–191. – ISSN 0304–3975