Lebenslinien

Inhalt

1 Definitionen:

Ein Graph G besteht aus Knoten V und Kanten E. Eine Kante E verbindet immer zwei Knoten V. Ein Graph kann auch mit $G = (V, E)$ dargestellt werden.

In einem ungerichteten Graph G muss für jede Kante E_1 zwischen x und y auch eine Kante E_2 zwischen y und x existieren ($x, y \in V$).

Die Nachbarn eines Knotens x ∈ V $N(x)$ sind alle Knoten in G, die mit einer Kante mit x verbunden sind. Sie werden auch als Adjazenzliste $Adj(x)$ bezeichnet. Dabei stellt $N^+(x)$ (rechte Nachbaren von x) eine Teilemenge von $N(x)$ dar, für die gilt

$$N^+(x) = \{\forall y \in N(x): \sigma^{-1}(x) < \sigma^{-1}(y)\}$$

σ stellt dabei eine Ordnung von G dar.

In einer Intervalldarstellung eines Graphen $G = (V, E)$ ist für jeden Knoten $v \in V$ ein Intervall $I(v)$ enthalten. Zwei Knoten x und y $(x, y \in V)$ dieses Graphen sind nur verbunden, wenn $I(x) \cap I(y) \neq \emptyset$.

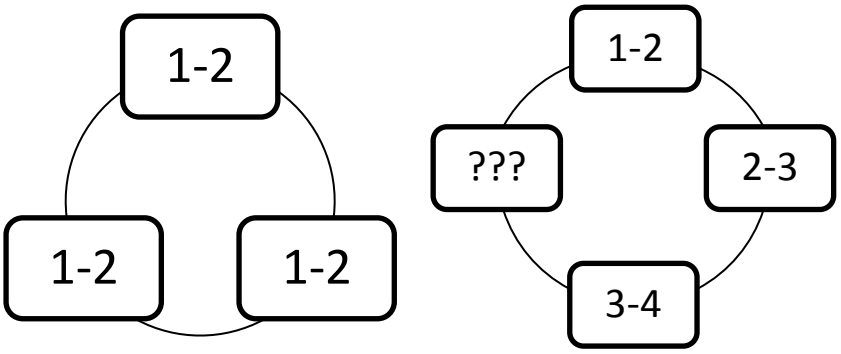
Ein LG ist ein ungerichteter Graph. Ein Knoten in diesem Graph repräsentiert eine Person und enthält deren Lebensspannen. Wenn sich die Lebensspanne von zwei Personen überschneiden, dann müssen die entsprechenden Knoten auch über eine Kante verbunden sein. In der Graphentheorie wird ein solcher Graph auch als Intervallgraph bezeichnet.

Ein PLG ist ein ungerichteter Graph, bei dem jeder Knoten eine Person darstellt. Im Gegenteil zum LG sind aber keine Lebensspannen eingetragen.

Ein PLG ist gültig, wenn für ihn eine Intervalldarstellung existiert.

Es gibt zwei Fälle in denen ein PLG kein gültiger LG ist:

- Ein PLG ist ungültig wenn er nicht chordal ist. Ein Graph ist chordal, wenn er keinen sehnenlosen Kreis mit mehr als 3 Knoten enthält. Eine Sehne innerhalb eines Kreises bezeichnet eine Kante die zwei zum Kreis gehörende Konten miteinander verbindet. Beispiel:



- Eine weitere Struktur die einen PLG ungültig macht ist ein "asteroidal triple". Für diesen Begriff gibt es keine gebräuchliche deutsche Übersetzung, er wird im Folgenden einfach als AT bezeichnet. Wenn ein Graph kein AT enthält, gilt er als AT-frei. Bei drei Knoten handelt es sich um ein AT, wenn zwischen allen Knotenpaaren $\{x, y\}$ des Triples ein Pfad zwischen x und y existiert, der weder den 3. Knoten enthält, noch die Nachbarschaft dessen durchquert. Folglich ist ein PLG gültig, wenn er AT-frei und chordal ist.

2 Lösungsidee:

2.1 Genereller Ansatz:

Es gibt zwei sinnvolle Möglichkeiten Intervallgraphen zu erkennen und ein Zertifikat zu erzeugen. Ein Zertifikat besteht in diesem Fall aus einer Intervalldarstellung als positive Bestätigung oder einem sehnenlosen Kreis als bzw. einem AT als negative Bestätigung.

Der erste Ansatz erstellt zunächst eine LexBFS Ordnung des Graphen. Wenn diese Ordnung eine Perfekte Eliminierungsordnung (auch perfect elemination order oder kurz PEO) darstellt, dann ist der Graph chordal. Sollte das nicht der Fall sein, so wird die ungültige Struktur identifiziert und als Zertifikat zurückgegeben. Andernfalls wird der Graph mit 6 LexBFS-Sweeps nummeriert. Dabei kommen neben normalem LexBFS noch die LexBFS Variationen LexBFS+ und LexBFS* zum Einsatz. Anschließend muss noch die Korrektheit der so generierten Nummerierung validiert und die Intervalldarstellung erzeugt werden. Wenn jedoch die Nummerierung nicht korrekt ist muss nach einem AT gesucht werden. Solange der Graph kein AT enthält, kann dieser Ansatz so implementiert werden, dass er in Linearzeit läuft. Um jedoch das AT zu finden muss jedes Tripel im Graphen überprüft werden, woraus sich eine Laufzeit $\sigma(|V|^3)$ ergibt. [3, 4]

Der andere Ansatz basiert anfangs auch auf einer Chordalitätsprüfung. Im nächsten Schritt wird jedoch ein MPQ-Tree für den Graphen erzeugt. MPQ-Trees werden inkrementell erzeugt. Wenn diese Erzeugung in einem Schritt fehlschlägt, dann kann der aktuell betrachtete Knoten dazu verwendet werden, ein AT in Linearzeit zu finden. Sonst wird aus dem MPQ-Tree eine Intervalldarstellung für G erzeugt. MPQ-Trees sind sehr komplizierte Datenstrukturen und der zur Erzeugung verwendete Algorithmus sehr komplex und folglich aufwendig zu implementieren. Dafür laufen alle Teileschritte in $\mathcal{O}(|V| + |E|)$. [1, 2, 3]

Letztendlich habe ich mich für die Implementierung mit MPQ-Trees entschieden. Dieser Ansatz hat nicht nur theoretisch eine bessere Laufzeit, sondern ist in den meisten Fällen auch in der Praxis schneller. Da die Gewinnung eines ATs in Linearzeit sehr kompliziert ist und dieser Fall nur selten auftritt, habe ich mich dazu entschlossen, einen einfacheren Ansatz zu verwenden. Dieser basiert auf dem $\mathcal{O}(|V|^3)$ Algorithmus, wurde von mir aber so abgewandelt, dass die Worst-Case Laufzeit bei $\mathcal{O}(|V|^2)$ liegt.

2.2 Verwendete Algorithmen:

2.2.1 LexBFS:

Bei LexBFS wird jedem Knoten $x \in V$ erst ein Label $\forall x \in V : L(x) = \emptyset$ hinzugefügt. Dieses Label enthält die Nummern aller bereits nummerierten Nachbarknoten in absteigender Reihenfolge sortiert. Es wird immer der Knoten als nächstes nummeriert, dessen Label nach lexikographischer Ordnung am größten ist. Beispiel: $\{1; 2\} < \{1; 3\} < \{2\}$. Die Nummerierung der Knoten erfolgt in absteigender Reihenfolge. Der erste Knoten bekommt die Gesamtanzahl n der Knoten zugeordnet, der zweite n-1 usw. Wenn mehrere Knoten das gleiche Label haben, kann beliebig zwischen diesen gewählt werden.

```
Eingabe: Ein Graph G = (V, E)
Ausgabe: Eine Sortierung σ von G

für jedes x in V
```

```
    L(x) = null

für i = Anzahl der Knoten n solange i > 0
    wähle Knoten x mit dem größten L(x) aus V
    σ(x) = i
    für jedes y in N(x)
        füge i zu L(y) hinzu
```

2.2.2 Chordalitätsprüfung:

Wie bereits unter 2.1 angeführt, wird im ersten Schritt der Graph auf Chordalität geprüft. Ein Graph G ist chordal, wenn er eine PEO besitzt. Eine Ordnung σ ist eine PEO, wenn gilt:

$$\forall z \in N^+(x) \wedge z \neq y : yz \in E$$

Wobei $x \in V$ und y das Element mit der kleinsten Nummerierung der rechten Nachbarn von x ist.

Das heißt eine Ordnung σ ist eine PEO wenn für jedes $x \in V$ $N^+(x)$ eine Clique bildet.

Ein trivialer Ansatz um festzustellen, ob σ eine PEO ist, wäre für jedes x zu testen, ob diese Bedingung zutrifft. Da diese Methode aber nicht in Linearzeit ausgeführt werden kann, verwendet man einen anderen Ansatz: Man berechnet für jeden Knoten, die Knoten mit denen er adjazent sein muss, damit seine rechten Nachbarn eine Clique bilden. Dazu muss y mit allen anderen rechten Nachbarn adjazent sein. Dann testet man ob in $N(x)$ eine zuvor berechnete Kante fehlt.

Dieses Vorgehen muss jetzt so abgewandelt werden, dass neben der simplen Rückgabe, ob G chordal ist, auch noch ein sehnenloser Kreis zurückgegeben wird, falls G nicht chordal ist. Dazu wird ausgenutzt, dass es in jedem nicht chordalen Graphen ein Triple $\{\mathrm{u}, \mathrm{v}, \mathrm{w}\}$ $(\mathrm{u}, \mathrm{v}, \mathrm{w} \in \mathrm{V})$ gibt, für das gilt:

$$\sigma^{-1}(u) < \sigma^{-1}(v) < \sigma^{-1}(w) \wedge uv \in E \wedge uw \in E \wedge vw \notin E$$

v und w werden bereits mit minimalen Abwandlungen von perfect und differ gefunden. Einen gemeinsamen Nachbarn u von v und w in Linearzeit zu finden ist trivial. Wenn diese Tripel lexikographisch maximal ist, dann liegen u, v und w auf einem sehnenlosen Kreis. Um die gewünschte lexikographische Maximalität von v und w zu erreichen, muss perfect erst alle Adjazenzlisten berechnen. Wenn man dann differ rückwärts über σ iteriert, findet man automatisch das lexikographisch höchste v und w. Das lexikographisch höchste u ist dann der gemeinsame Nachbar von v und w mit dem kleinsten Label.

Um den Rest des Kreises zu gewinnen, muss man nur noch den kürzesten Pfad zwischen v und w finden, der u sowie $\mathrm{N(u)}\backslash\{\mathrm{v}, \mathrm{w}\}$ vermeidet.

Dazu wird die LexBFS-Abwandlung LexBFSFindAPath verwendet. Auf diese wird unter 3.2.1 genauer eingegangen.

Die komplette Chordalitätsprüfung findet man in detaillierterer Ausführung sowie mit Laufzeitdetails und Beweisen in [3].

2.2.3 SortAdjLists [3]:

Um die lineare Laufzeit der Chordalitätsprüfung zu erreichen, müssen die Adjazenzlisten aller Personen nach ihrer Nummerierung geordnet sein. Da konventionelle Ordnungsverfahren nicht in Linearzeit implementiert werden können, ist ein spezieller Algorithmus nötig. Dieser sortiert die

Adjazenzlisten aller Knoten nach einer bestimmten Nummerierung und verlang als Eingabe eine Sortierung σ des Graphen

```
Eingabe: Ein Graph G = (V, E), eine Ordnung
Ausgabe: Ein Graph G = (V, E) mit sortierten Adjazenzlisten

Für jeden Knoten x in V
    Erstelle A(x)

Für jeden Knoten x in σ
    Für jeden Knoten y in N(x)
        Füge x zu A(y) hinzu

Für jeden Knoten x in V
    N(x) := A(x)
```

2.2.4 MPQ-Trees:

Ein MPQ-Tree repräsentiert alle möglichen Intervalldarstellungen eines Graphen, in dem er einen Baum aus den maximalen Cliquen bildet.

Dabei kommen PNodes, QNodes und Leaves zum Einsatz. Jede maximale Clique des dazugehörigen Graphen wird dabei von einem Pfad von der Root zu einem Leaf dargestellt. Jeder Knoten und jedes Leaf enthält dabei einen Verweis auf die exklusiv enthaltenen Personen.

Ein PNode fasst mindestens 3 maximale Cliquen mit Überschneidungen zusammen. Die gemeinsamen Personen sind dabei Inhalt der PNode und alle Knoten, durch die ein zu den betroffenen Cliquen gehörender Pfad verläuft, sind Kinder der PNode.

Eine QNode besteht aus mehreren Sections $\{S_1, \dots, S_i\}$ $(i \geq 2)$. Jede Section S_n $(1 \leq n \leq i)$ hat dabei einen Sohn. Für die genaueren Definitionen verweise ich hier auf [1].

Um mithilfe von MPQ-Trees zu testen ob ein Graph ein Intervallgraph ist, muss dem Graph inkrementell immer die Person als nächstes hinzugefügt werden, die die geringste LexBFS Nummerierung hat. Dabei kommen mehrere Templates zum Einsatz. Wenn das Hinzufügen bei einer Person x fehlschlägt, dann ist der bis zu diesem Schritt hinzugefügte Graph G ein Intervallgraph, $G+x$ jedoch nicht.

Es wird immer die Person mit der kleinsten LexBFS-Nummerierung als nächstes zum MPQ-Tree hinzugefügt. Das Hinzufügen einer Person geschieht dabei in zwei Phasen:

2.2.4.1 Labeling-Phase:

Dabei wird für jede Person x, die hinzugefügt wird, erst der MPQ-Tree gelabelt. Dabei bekommt jede Node bei der keine enthaltenen Personen zu x adjazent sind das Label 0, jede Node, deren enthaltene Personen teiladjazent zu x sind das Label 1, und jede Node, deren enthaltene Personen volladjazent zu x sind das Label unendlich.

2.2.4.2 Treebuilding-Phase:

Templates werden aufsteigend entlang eines Pfades P (inklusive Start- und Endnoten) angewandt. Um diesen Pfad zu ermitteln benötigt man zwei weitere Pfade: Den Pfad P_P der alle Knoten mit positiven Labels enthält und den Pfad P_R vom Root bis zu einem Leaf das in P_P enthalten ist. Der Pfad P beginnt in der höchsten nicht leeren PNode oder QNode die das Label 1 oder 0 hat und endet in der untersten Node in P_R, die das Label 1 oder unendlich hat.

Auf die einzelnen Templates möchte ich hier nicht genauer eingehen sondern verweise auf die dazugehörige Fachliteratur [1]. Dort befinden sich auch die Beweise und Laufzeitdetails. Im Anhang befindet sich ein Ausschnitt in dem die Templates erklärt werden.

2.2.5 Erzeugung der Intervalldarstellung:

Wenn das Erzeugen des MPQ-Trees erfolgreich ist, dann wird mit Tiefensuche jeder Pfad des MPQ-Trees begangen und dadurch eine Ordnung der maximalen Cliquen generiert. Aus dieser Ordnung wird dann die Intervalldarstellung erzeugt.

2.2.6 Finden eines ATs:

Ansonsten wird der Knoten, bei dem die Erzeugung des MPQ-Trees fehlgeschlagen ist zurückgegeben. Dieser Knoten muss Teil jedes ATs des Graphen sein.

Beweis: [2] Lemma 5.2

Dann kann man anhand des Ansatzes aus [4] ein AT finden. Dazu nutzt man eine bestimmte Eigenschaft von ATs aus:

Ein Tripel $\{u, v, w\}$ ist ein AT nur wenn $C_v(u) = C_v(w)$ und $C_u(v) = C_u(w)$ und $C_w(u) = C_w(v)$, wobei $C_v(w)$ der Zusammenhangskomponent von w in $\mathrm{G} - \mathrm{N}(\mathrm{x})$.

Um diese Bedingung zu überprüfen, wird eine Matrix erstellt, in der für jedes $\mathrm{x} \in \mathrm{V}$ die Label der Zusammenhangskomponenten aller anderen Knoten gespeichert werden. Wenn ein Knoten ein Nachbar von x ist, dann ist dieses Label 0. Dann wird nach v und w gesucht. [4]

Da in der Aufgabenstellung verlangt wird, dass die zurückgegebene Struktur kein gültiger PLG ist, müssen noch mittels LexBFSFindAPath die zum AT gehörenden Pfade ermittelt werden.

2.3 Erweiterung der Aufgabenstellung:

Die ursprüngliche Lösungsidee wurde vom mir um die Zielsetzung einer bestmöglichen Laufzeit erweitert. Dass ist mir bis auf das Finden von ATs gelungen und ich habe auch einen Ansatz, der ATs in Linearzeit findet gefunden [2]. Dieser wurde aber aufgrund der weiter oben angeführten Gründe nicht implementiert. Um dieses Ziel zu erreichen, musste ich jedoch einen kleinen Kompromiss bei der Aufgabenstellung eingehen. Dort wird gefordert, dass das Programm die kleinstmögliche ungültige Struktur zurückgibt, sodass bereits diese Struktur keinen LG darstellen kann. Diese Aufgabenstellung kann von meinem Programm nicht in allen Fällen vollkommen erfüllt werden. Wenn ein PLG zwei oder mehr ungültige Strukturen des gleichen Typs enthält, dann kann das Programm nicht ohne einen Laufzeitverlust sichergestellt werden, dass die gefundene Struktur die kleinstmögliche ist. Daher habe ich mich entschieden zugunsten der Laufzeit bewusst ein kleines Detail der Aufgabenstellung zu vernachlässigen. Dennoch möchte ich hier noch kurz anführen, wie dieses Detail eingehalten werden kann. Anstatt im Falle, dass der Graph nicht chordal ist nur ein mögliches verletzendes Triple zu ermitteln, ist es auch möglich sämtliche verletzenden Tripel zu ermitteln. Danach müsste für jedes Tripel der dazugehörige Kreis berechnet werden. Von allen so erzeugten Kreisen muss dann noch der Kürzeste ausgewählt werden. Bei den ATs ist ein ähnliches Vorgehen nötig. Zuerst müssen alle möglichen ATs gefunden werden, was trivialerweise die Best-Case-Laufzeit mit der Worst-Case-Laufzeit von $\mathcal{O}(|\mathrm{V}^2|)$ gleichsetzen würde. Dann müssten alle Pfade zwischen den einzelnen Knoten jedes möglichen ATs gefunden werden und aus diesen dann das Tripel mit der zusammengezählt kürzesten Pfadlänge ausgewählt werden.

3 Umsetzung:

3.1 Das Programm:

Das Programm wurde in C# umgesetzt, als Benutzeroberfläche kommt WPF zum Einsatz. Auf Features wie eine grafische Darstellung der Graphen habe ich bewusst verzichtet. Stattdessen verwaltet man im Programm die Adjazenzlisten der einzelnen Knoten. Diese Art der Eingabe ist zwar etwas umständlicher, erlaubt aber auch das Verwalten größerer Graphen. Die Ausgabe erfolgt in simpler Textform.

Außerdem kann das Programm Graphen einlesen und auch abspeichern. Alle in dieser Dokumentation verwendeten Beispiele werde ich als Datei bereitstellen.

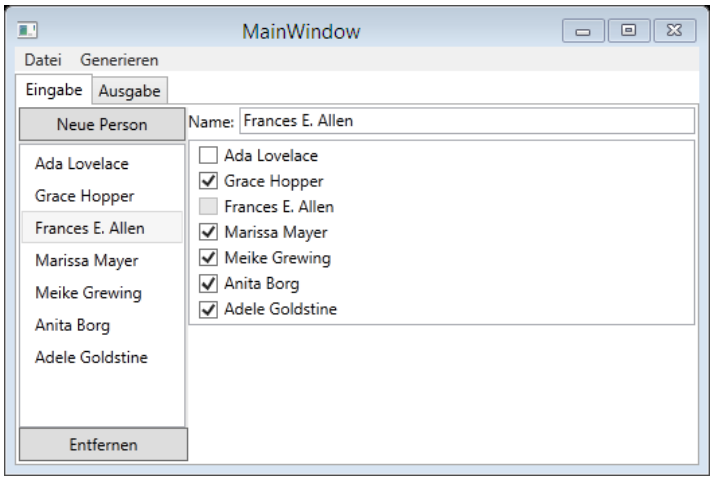


Abbildung 1: Beispieleingabe für den LG aus der Aufgabenstellung

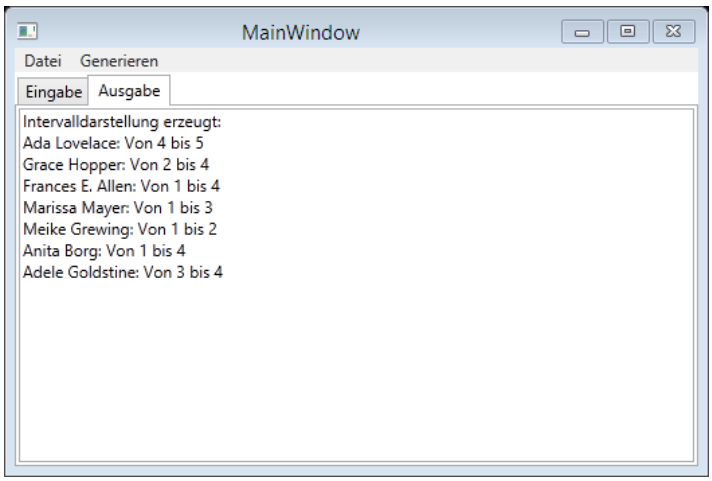


Abbildung 2: Ausgabe für den obrigen Graph

In der Datei `PLG.cs` befindet sich die Klasse PLG, die einen PLG repräsentiert. Diese Klasse hat eine Liste mit Referenzen zu den einzelnen Knoten eines PLGs, die von der Klasse Person repräsentiert werden. Die Klasse Person befindet sich in der Datei `Person.cs`. Dabei hat ein Objekt der Klasse Person neben einer Adjazenzliste noch weitere Variablen, wie z. B. ihren Namen und Variablen die die Verarbeitung im Programm erleichtern, wie z. B. ID. Ein MPQTree wird durch die Klasse `MPQTree` repräsentiert. Diese Klasse enthält auch noch Unterklassen, die in der Datenstruktur eines `MPQTrees` zum Einsatz kommen. Auf diese Datenstruktur möchte ich später noch genauer eingehen.

Wenn der Nutzer in der Benutzeroberfläche auf „Ausgabe“ wechselt, wird eine Methode in der GUI-Klasse `MainWindow` aufgerufen. Diese Methode erstellt im Hintergrund eine neue Instanz der Klasse `Plg`. Dann ruft sie die Methode `TestIntervalGraph` auf. In dieser Methode wird erst mit `TestChordal` getestet ob der gegebene `Plg` chordal ist. Wenn ja wird `false` und eine gefundener Kreis zurückgegeben. Sonst wird in `TestIntervalGraph` geprüft, ob sich alle Knoten des `Plg` zu einem `MPQTree` hinzufügen lassen. Wenn dies in einem Schritt fehlschlägt, wird `FindAT` aufgerufen, sonst wird eine Intervalldarstellung erzeugt in dem die `GenerateIntervalRepresentation` des `MPQTrees` aufgerufen wird. Abhängig vom der Rückgabe von `TestIntervalGraph` wird dann ein Ausgabetext erzeugt.

3.2 Implementierungsdetails:

3.2.1 LexBFS:

Um LexBFS in Linearzeit zu implementieren, bedarf es eines anderen Ansatzes als weiter oben dargelegt. Anstatt die Labels der einzelnen Knoten zu betrachten, weist man den Knoten verschiedene Sets zu, wobei ein Set jeweils nur Knoten mit dem gleichen Label enthält. Die Sets liegen dabei in nach Label geordneter Reihenfolge vor. Durch eine geschickte Implementierung ist es dabei nicht nötig die Sets zu ordnen, weil diese automatisch in geordneter Reihenfolge verbleiben. Es wird immer ein Knoten im höchsten Set nummeriert. Alle seine Nachbarn steigen dann ein Set auf.

Um eine lineare Laufzeit zu erreichen, muss auch auf die passende Wahl der Datenstrukturen geachtet werden. So wird jedes Set von einer doppelt verketteten List repräsentiert und die Liste aller Sets ist ebenfalls doppelt verkettet. Um einen schnellen Zugriff auf die einzelnen Personen und Sets zu garantieren, werden diese in Dictionaries zusammen mit einem Pointer auf ihre Position gespeichert. Für die doppelt verketteten Listen wird die .NET Klasse `LinkedList<T>` verwendet und für die Pointer die Klasse `LinkedListNode<T>`. Für die Korrektheit und Laufzeitbetrachtung sowie weitere Details verweise ich auf [3].

3.2.2 LexBFSFindAPath:

`LexBFSFindAPath` wird dazu eingesetzt, einen Pfad zu finden, der von w nach v verläuft und dabei $N(u) \cup u$ vermeidet. Um das zu erreichen, wird w als erstes nummeriert. $N(u) \cup u$ wird in ein Set hinter allen anderen Knoten gesetzt. Somit wird sichergestellt, dass diese Knoten als letztes nummeriert werden. Außerdem wird für jeden Knoten die Entfernung vom Startknoten gespeichert.

Wenn dann die Adjazenzlisten aller Knoten aufsteigend sortiert sind, kann in Linearzeit ein Pfad von w nach v gefunden werden. Dabzu muss solange von v ausgehend der erste Knoten des ersten Elements in den Nachbarn des aktuellen Knotens ausgewählt werden, bei dem die Entfernung zu w abnimmt, bis w gefunden ist.

3.2.3 MPQ-Trees:

Die Datenstrukturen der MPQTrees nutzen das Prinzip der Vererbung. So erben Leaf, PNode, QNode und Section alle von der abstarkten Klasse `BaseNode`. Das hat den Vorteil, dass alle Datenstrukturen eine gemeinsame Grundfunktionalität haben und alle dank Downcasting in einer Liste gespeichert werden können. Außerdem sind die Datenstrukturen selbstverwaltend. Das spart einem beispielsweise beim Setzten eines neuen Vaters für eine `BaseNode` die Überprüfung, ob weitere Aktionen aufgrund des Typs des Vaters nötig sind. Wenn der alte Vater zum Beispiel eine PNode ist, muss die aktuelle `BaseNode` aus den Kindern des alten Vaters entfernt werden.

Ein `MPQTree` hat ein `Dictionary`, das jede Person im `MPQTree` und die dazugehörige `BaseNode` enthält. Außerdem enthält ein `MPQTree` für jede Person einen Verweis auf die Position dieser Person innerhalb der enthaltenen Personen einer `BaseNode`. Die in der Fachliteratur vorgeschlagene Erweiterung, dass für jede Person zwei Verweise auf die Position vorhanden sind, für den Fall, dass sich die Person in einer QNode befindet habe ich weggelassen und auch einige Einschränkungen bezüglich des Typs eines Vaters habe ich vereinfacht, sodass die Datenstrukturen sich besser generalisieren lassen. Trotz der Vereinfachungen haben die Datenstrukturen immer noch die gleiche Aussage und ich konnte diese Vereinfachungen nur vornehmen, weil ich Features von C# genutzt habe, die zu der Zeit, als der Text verfasst wurde, noch nicht gebräuchlich waren, wie z. B. Casting oder andere objektorientierte Ansätze.

Das Labeling beim Hinzufügen einer neuen Person x zu einem MPQ-Tree wird wie folgt vorgenommen:

```
Für jede Person y in N(x)
    N := Node in der y enthalten ist
    Wenn N eine Section ist überprüfe ob N eine äußere Section ist
    Entferne y aus N und füge y A(N) hinzu
    Füge N zur Queue QU hinzu

Solange QU nicht leer ist
    Entferne N von QU
    flag(N) auf wahr setzten
    Vater V von N zu QU hinzufügen #FIFO
    Verweis von V auf N speichern
```

Anhand der so gewonnenen Informationen können jetzt die für die Tree-Building-Phase nötigen Nodes identifiziert werden.

4 Laufzeit und Korrektheit:

Die meisten Laufzeitbegründungen und Korrektheitsbeweise für die hier verwendeten Lösungsstrategien sind sehr komplex, deshalb werde ich auf diese nicht weiter eingehen. Nur die Teilschritte, die in der Fachliteratur nicht genauer behandelt werden und diejenigen die von mir laufzeitrelevant abgeändert wurden, werde ich hier abhandeln.

Dazu gehört die Generierung der Intervalldarstellung aus einem MPQ-Tree. Dazu wird zuerst eine Tiefensuche auf einen MPQ-Tree angewandt. Die so generierte Liste der maximalen Cliquen ist korrekt, weil die Tiefensuche jeden Pfad des MPQ-Trees von der Root bis zu einem Leaf von links nach rechts genau einmal besucht (trivial). Da die enthaltenen Personen auf jedem Pfad von der Root zum Leaf eines MPQ-Trees eine maximale Clique darstellen, muss so eine Liste mit allen maximalen Cliquen entstehen. Dadurch, dass ein korrekter MPQ-Tree alle möglichen Ordnungen der maximalen Cliquen repräsentiert, ist garantiert, dass die so erzeugte Liste linear geordnet ist. Die Laufzeit dieser Tiefensuche ist linear zur Anzahl der Knoten in G, da der komplette MPQ-Tree einmal durchlaufen wird und die Speicherkomplexität eines MPQ-Trees linear zur Anzahl der Knoten im dazugehörigen Graphen ist. [1]

Auch dass man aus einer linearen Ordnung der maximalen Cliquen eine Intervaldarstellung erzeugen kann, ist leicht zu beweisen. Dass die Schnittmenge der Intervalle alle Knoten innerhalb einer maximalen Clique keine leere Menge ist, ist trivial. Wenn ein Knoten in mehreren Cliquen enthalten

ist wird der Beweis etwas komplizierter. Dadurch, dass in einer linearen Ordnung eine Clique C_1, die sich mit einer anderen Clique C_2 überschneidet (also mindestens einen gemeinsamen Knoten hat) in der Ordnung neben dieser liegen muss, lässt sich ableiten, dass die gemeinsamen Knoten ein Intervall haben müssen, das sich sowohl mit allen Intervallen der ersten Clique C_1, als auch mit allen Intervallen der zweiten Clique C_2 schneidet.

Die Laufzeit der aus [4] abgewandelten Methode beträgt $\mathcal{O}(|V|^2)$, weil in [4] jedes einzelne Tripel des Graphen getestet wird (Laufzeit $\mathcal{O}(|V|^3)$) und im abgewandelten Ansatz durch die Vorgabe eines Knoten des Tripels auf eine Iteration durch V verzichtet werden kann. Dadurch beträgt die Laufzeit $\mathcal{O}(|V|^3 \div |V|) = \mathcal{O}(|V|^2)$

Um mit `LexBFSFindAPath` einen Pfad zu finden, wird maximal einmal die komplette Knotenmenge durchlaufen, sowie maximal jede Kante des Graphens einmal überprüft. Folglich ist die Laufzeit $\mathcal{O}(|V| + |E|)$.

5 Beispiele:

5.1 Beispiel 1:

Ein Graph, der dazugehörige MPQ-Tree und die Intervalldarstellung:

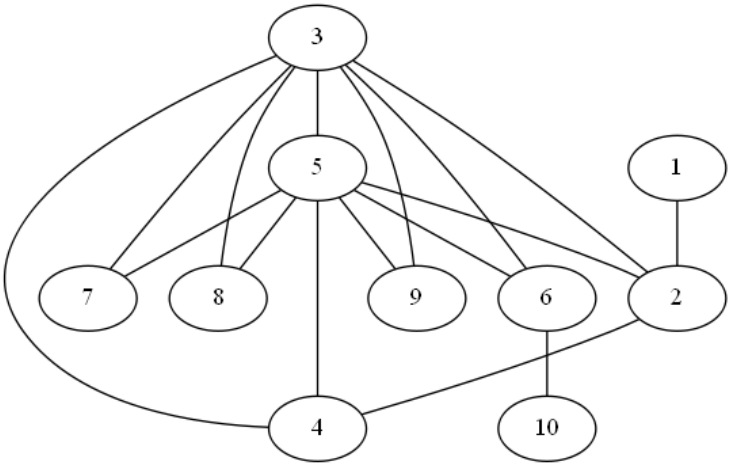


Abbildung 3: Ein Graph mit gültiger LexBFS Nummerierung

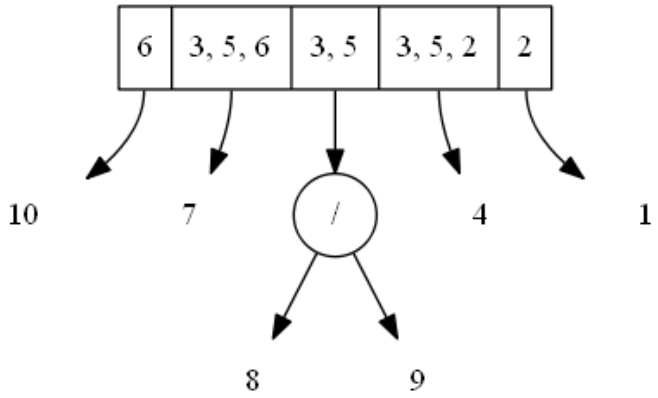


Abbildung 4: Der dazugehörige MPQ-Tree (eckig QNode, rund PNode, / entspricht ∅)

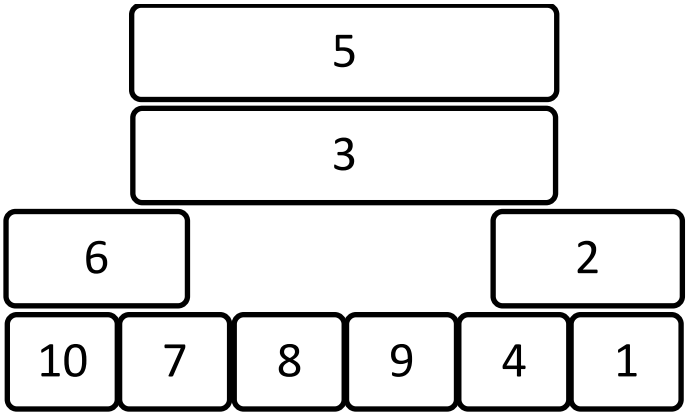


Abbildung 5: Die dazugehörige Intervalldarstellung

5.2 Beispiel 2:

Der Aufbau eines MPQ-Trees für einen Graphen mit AT:

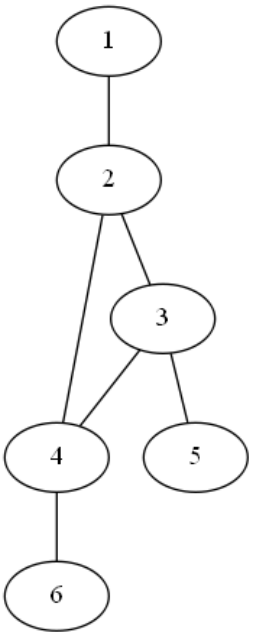
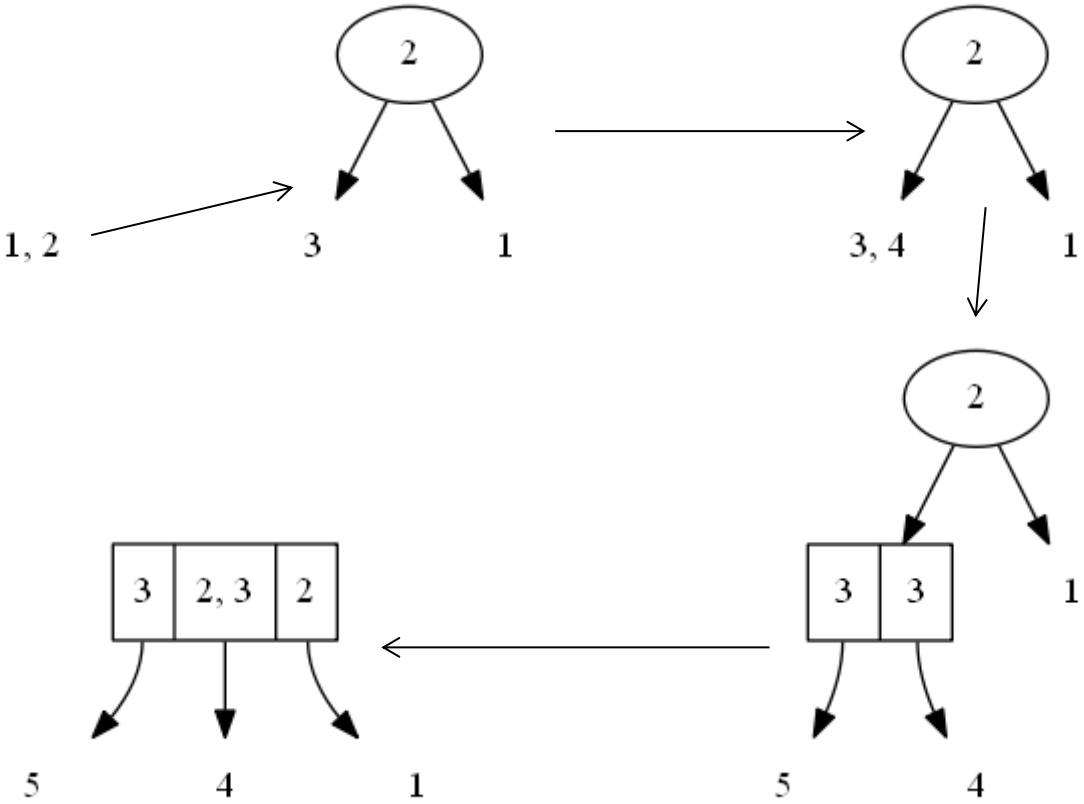


Abbildung 6: Ein Graph mit AT



Erstellung eines MPQ-Trees für den obrigen Graphen

	1	2	3	4	5	6
1	0	0	a	a	a	a
2	0	0	0	0	b	c
3	d	0	0	0	e	0
4	f	0	0	0	0	g
5	h	h	h	0	0	h
6	i	i	0	i	i	0

Abbildung 7: Matrix der Zusammenhangskomponenten mit Markierung des ATs

5.3 Beispiel 3:

Die Erkennung eines nicht chordalen Graphen:

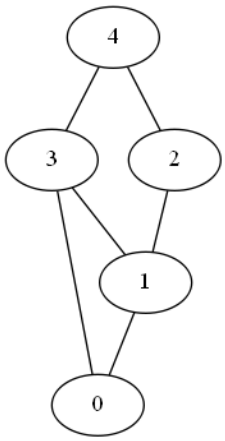


Abbildung 8: Ein nicht chordaler Graph mit eine korrekten LexBFS-Nummerierung

Liste der rechten Nachbarn:

$N^+(4) = \emptyset$

$N^+(3) = \{4\}$

$N^+(2) = \{4\}$

$N^+(1) = \{2; 3\}$

$N^+(0) = \{1; 3\}$

Vom Programm berechnete Adjazenzlisten und die echten Adjazenzlisten:

$A(1) = \{3\}$
$A(2) = \{3\}$

$A(4) = \emptyset$

$Adj(0) = \{1; 3\}$
$Adj(1) = \{0; 2; 3\}$
$Adj(2) = \{1; 4\}$
$Adj(3) = \{0; 1; 4\}$
$Adj(4) = \{2; 3\}$

A() steht dabei für die von perfect berechneten Listen und Adj() für die echten Adjazenzlisten. Die hervorgehobenen Elemente stehen für v und w. u ist das Element mit der kleinsten Nummerierung in $Adj(2) \cup Adj(3)$, nämlich 1.

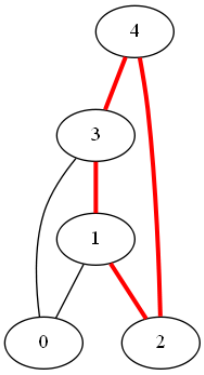


Abbildung 9: Der gefundene Kreis

6 Quellen:

[1] Korte, N., & Möhring, R. H. (1989). An incremental linear-time algorithm for recognizing interval graphs. SIAM Journal on Computing, 18(1), 68-81.

[2] Kratsch, D., McConnell, R. M., Mehlhorn, K., & Spinrad, J. P. (2006). Certifying algorithms for recognizing interval graphs and permutation graphs. SIAM Journal on Computing, 36(2), 326-353. (http://www.mpi-inf.mpg.de/~mehlhorn/ftp/KMMS.pdf)

[3] Biermann, M. (2007). Erkennen von Graphenklassen mittels lexikographischer Breitensuche (http://www.fernunihagen.de/MATHEMATIK/DMO/pubs/biermann.pdf)

[4] Köhler, E. (2000, January). Recognizing graphs without asteroidal triples. In Graph-Theoretic Concepts in Computer Science (pp. 255-266). Springer Berlin Heidelberg. (https://www.sciencedirect.com/science/article/pii/S157086670400019X)

7 Anhang A: Templates für MPQ-Trees [1]

In each step, the current node N (together with its associated sets and its label) is compared with a small number of patterns that trigger the appropriate modification of N. The combination of pattern recognition and modification is called a *template*. There are three groups of templates, depending on whether the current node N is a leaf, a P-node, or a Q-node. Each group has a template for up to three subcases, depending on whether or not $N = N_*$, $N_* \neq N^*$, etc. These groups of templates are displayed in Figs. 4–6. In all cases, $V_N = A \cup B$ denotes the partition of the vertex set V_N of the current node or section into the set A of vertices adjacent to u and the set B of vertices not adjacent to u. Furthermore, T_i and T_i' always denote subtrees of

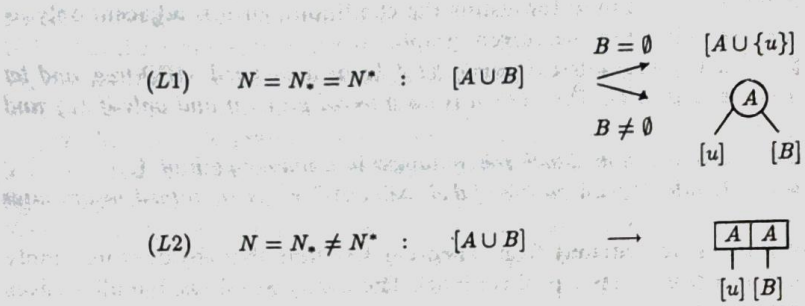
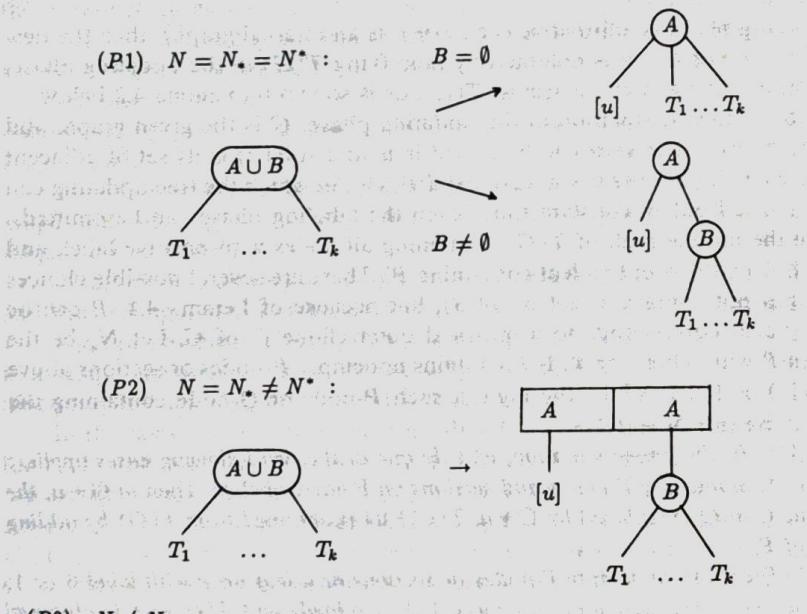


FIG. 4. *Templates for a leaf.*



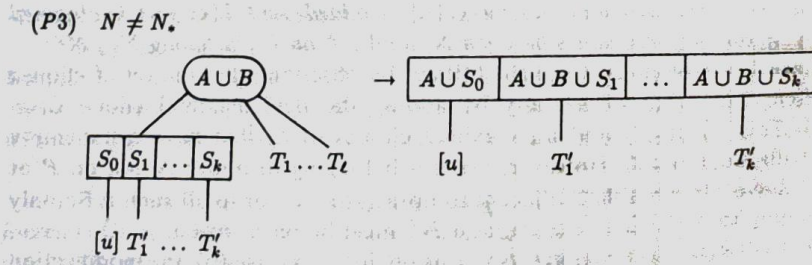


FIG. 5. *Templates for a P-node.*

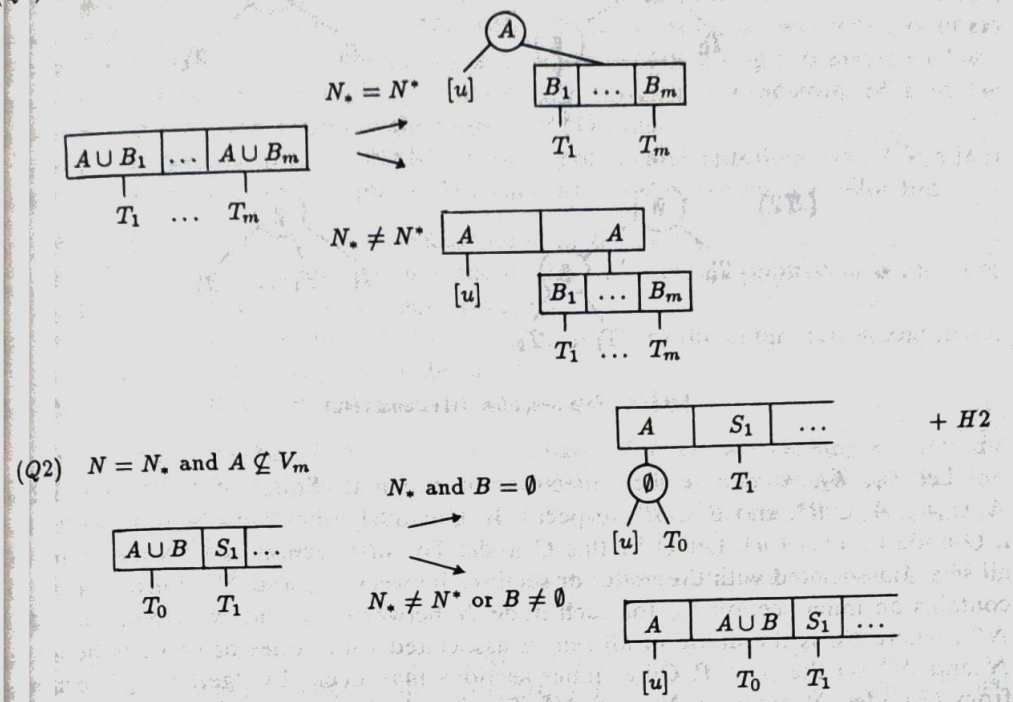


FIG 6. *Templates for a Q-node.*

$T(G)$ that contain only nodes with label 0. (This is an immediate consequence of Lemma 4.1.) $H1$ and $H2$ are help-templates that deal with special cases such as only one son, etc. They are displayed in Fig. 7. Note that template $(L1)$ is designed so as to also cover Case (a) in Lemma 4.2.

THEOREM 4.3. *The application of the templates along the path P from N_* to N^* correctly produces $T(G+u)$.*

Proof. It is easily seen that the templates cover all formally distinct cases. Also, all templates for the case $N_* = N^*$ and the help-templates $H1$ and $H2$ are easily shown to be correct.

Now consider the case $N_* \neq N^*$. Since N_* has the label 1 or ∞, the associated set A_* is nonempty. Similarly, the set B^* associated with N^* is nonempty. Let B^0 denote the set of vertices associated with nodes that are below N^* but not on P (which exist since N^* is not a leaf).

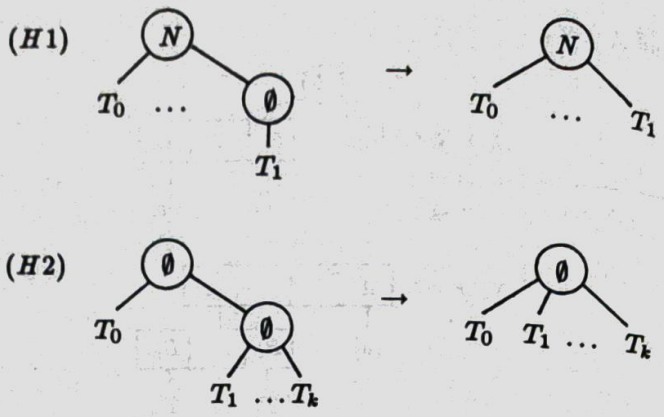


FIG. 7. *Help-templates* (H1) *and* (H2).

Let $\mathcal{C}_1$, $\mathcal{C}_2$, $\mathcal{C}_3$ denote the collections of maximal cliques of $G+u$ containing $A_* \cup \{u\}$, $A_* \cup B^*$, and $B^* \cup B^0$, respectively. Obviously, they must be represented by a Q-node in $T(G+u)$. Let $\bar{Q}$ be this Q-node. The first section of $\bar{Q}$ is the union of all sets A associated with the nodes or sections between N_* and N^*. Furthermore, $\bar{Q}$ contains an inner section S_N for each node N between N_* and N^* (including N_*, N^*), where S_N is the union of all sets A associated with nodes or sections between N and N^* on the path P. Other inner sections may occur by "gathering" sections from Q-nodes N between N_* and N^*. Finally, the last section has the associated vertex set $A^* \cup B^*$ of N^*.

The templates $L2$, $P2$, $Q1$ a) and $Q2$ a) (Figs. 4–6) construct the first part of this Q-node and return it as input for the next template. Inductively, the remaining templates on the path from N_* to N^* construct the complete Q-node. Although straightforward, the details are rather tedious, and therefore omitted. □

COROLLARY 4.4. *If $N_* \neq N^*$, then all nodes properly between N_* and N^* on the path P in $T(G)$ will become inner sections of a Q-node in $T(G+u)$.*

As an example, consider the interval graph in Fig. 3. Using the given LEXBFS, the following templates are applied (the notation is "added vertex: templates-added vertex," etc., the starting graph is the graph with vertex 1) 2: $L1$-3: $L1$-4: $L1$-5: $L1$-6: $L2$, $P3$, $H1$-7: $L1$-8: $Q2$-9: $P1$-10: $L2$, $P3$, $Q3$. This example also shows that while the Q-node $\bar{Q}$ of $T(G+u)$ is constructed, several sections may be equal or empty. This happens, e.g., when 10 is added and $P3$ is applied.

5. Implementation and complexity. In this section, we show that the above methods lead to an $O(|V|+|E|)$ algorithm for recognizing whether a given graph $G=(V, E)$ is an interval graph and for producing its associated MPQ-tree $T(G)$ if it is.

To achieve this complexity, we must use a good data structure for MPQ-trees. We assume that the children $N_1, \cdots, N_k$ of a P-node N are kept in a doubly linked circular list. Each N_i has a parent pointer to N, and N has a child pointer to one N_i. The sections $S_1, \cdots, S_m$ of a Q-node N have a pointer to their neighbor sections (so inner sections have two, while outer sections have only one). N has child pointers only to the outer sections S_1 and S_m, and only these have a parent pointer to N. Each section S_i has a pointer to its son. Each node has a parent pointer to its father (which is a P-node or a section of a Q-node). With each node or section, the associated set of vertices is given by a doubly linked circular list. Vertices contained in several sections

8 Anhang B: Quellcode

8.1 Person.cs

```
using System.Collections.Generic;
using System.ComponentModel;
using System.Diagnostics;
namespace PLG
{
    [DebuggerDisplay("ID = {ID}, Name = {Name}")]
    class Person : INotifyPropertyChanged
    {
        private string name;

        public readonly int ID;
        public int Birth, Death, Position = -1, Distance = -1; //Because of the arrays
being based on zero, -1 means not in the list
        public List<Person> Neighbors;
        public bool Flagged;

        public string Name
        {
            get { return name; }
            set
            {
                name = value;
                OnPropertyChanged("Name");
            }
        }

        public Person(int ID)
        {
            this.ID = ID;
            Neighbors = new List<Person>();
            Name = (ID + 1).ToString();
        }

        public event PropertyChangedEventHandler PropertyChanged;

        private void OnPropertyChanged(string name)
        {
            if (PropertyChanged != null)
                PropertyChanged(this, new PropertyChangedEventArgs(name));
        }
    }
}
```

8.2 PLG.cs

```
using System;
using System.Collections.Generic;
using System.Linq;

namespace PLG
{
    class Plg
    {
        public List<Person> Persons
        {
            get;
            private set;
        }
```

```
        public Plg(IList<Person> persons)
        {
            Persons = new List<Person>(persons);

            foreach (Person person in persons)
            {
                person.Flagged = false;
                person.Distance = -1;
                person.Position = -1;
            }
        }

        /// <summary>
        /// Perform an LexBFS ordering on G
        /// </summary>
        /// <param name="direction">Direction of the ordering
(forwards/backwards)</param>
        /// <returns>An order of G</returns>
        private Person[] LexBFS(Direction direction)
        {
            //Initialize data structures
            LinkedList<SetList<Person>> listOfSets = new
LinkedList<SetList<Person>>();
            Dictionary<Person, LinkedListNode<SetList<Person>>> setPointers = new
Dictionary<Person, LinkedListNode<SetList<Person>>>();
            Dictionary<Person, LinkedListNode<Person>> positionPointers = new
Dictionary<Person, LinkedListNode<Person>>();

            //Set that have to be unflagged after the iteration
            List<LinkedListNode<SetList<Person>>> setsToCheck = new
List<LinkedListNode<SetList<Person>>>();

            Person[] outputArray = new Person[this.Persons.Count];

            //Generate first set (represents all persons without label)
            SetList<Person> firstSet = new SetList<Person>();
            listOfSets.AddFirst(firstSet);

            //Get all persons ready
            foreach (Person person in Persons)
            {
                person.Position = -1;
                positionPointers[person] = firstSet.AddLast(person);
                setPointers[person] = listOfSets.First;
            }

            //Determine startindex depending on direction
            int i = (direction == Direction.Backwards ? this.Persons.Count - 1 : 0);

            //Label all persons
            while(direction == Direction.Backwards ? i >= 0 : i < this.Persons.Count)
            {
                //Select current person for labeling and remove it from its set
                Person currentPerson = listOfSets.Last.Value.First.Value;
                positionPointers.Remove(currentPerson);
                listOfSets.Last.Value.RemoveFirst();

                //Remove set if empty
                if (listOfSets.Last.Value.Count == 0)
                    listOfSets.Remove(listOfSets.Last);

                //Assign label
                currentPerson.Position = i;
```

```
                outputArray[i] = currentPerson;

                //Assign new sets to all neighbors
                foreach (Person neighbor in currentPerson.Neighbors)
                {
                    //Ignore labeled neighbors
                    if (neighbor.Position == -1)
                    {
                        //Get current set of neighbor
                        LinkedListNode<SetList<Person>> currentSet =
setPointers[neighbor];

                        //If currentSet has no replacement generate replacement set
generate one
                        if (!currentSet.Value.HasReplacement)
                        {
                            currentSet.Value.HasReplacement = true;
                            SetList<Person> newSet = new SetList<Person>();
                            listOfSets.AddAfter(currentSet, newSet);

                            setsToCheck.Add(currentSet);
                        }

                        //Put neighbor in a set one level higher
                        currentSet.Value.Remove(positionPointers[neighbor]);
                        setPointers[neighbor] = currentSet.Next;
                        positionPointers[neighbor] =
currentSet.Next.Value.AddLast(neighbor);
                    }
                }

                //Unflag all previously flagged set and delete them if empty
                foreach (LinkedListNode<SetList<Person>> setList in setsToCheck)
                {
                    if (setList.Value.Count == 0)
                        listOfSets.Remove(setList);

                    else
                        setList.Value.HasReplacement = false;
                }

                setsToCheck.Clear();

                i = i + (direction == Direction.Backwards ? -1 : 1);
            }
            return outputArray;
        }

        /// <summary>
        /// Test if this PLG is chordal
        /// </summary>
        /// <param name="output">If this PLG is not chordal, it returns an circle (in
the form of an array with more than 3 elements)</param>
        /// <returns>Returns true if the graph is chordal</returns>
        private bool TestChordal(out Person[] output)
        {
            //Generate a potential perfect elimination order (PEO) with LexBFS
            Person[] potentialPEO = LexBFS(Direction.Backwards);
            Person[] triple;

            //If potentialPEO is a PEO the graph is chordal
            if (TestPerfectEleminationOrder(potentialPEO, out triple))
            {
```

```
                //Set the PEO as output
                output = potentialPEO;

                return true;
            }

            else
            {
                //Calculate a path from u, v and w leading to an chordless cycle with
more then 3 vertices
                //Used as certificate because of being a potential smallest invalid
structure

                //Uses LexBfs to label the node for finding a path
                Person[] pathOrder = LexBFSFindAPath(triple);

                //If adjacent lists are sorted the first node with the lower distance
is always on the path between w and v
                SortAdjLists(pathOrder);

                //List representing the found cycle
                List<Person> cycle = new List<Person>();

                Person currentPerson = triple[1];

                while (currentPerson.Distance > 0)
                {
                    cycle.Add(currentPerson);

                    //Find first neighbor with a lower distance
                    foreach (Person neighbor in currentPerson.Neighbors)
                    {
                        if (neighbor.Distance < currentPerson.Distance)
                        {
                            currentPerson = neighbor;
                            break;
                        }
                    }
                }

                cycle.Add(triple[2]);
                cycle.Add(triple[0]);

                //Return the found cycle
                output = cycle.ToArray();

                return false;
            }
        }

        /// <summary>
        /// Test if a given order is a PEO
        /// </summary>
        /// <param name="order">The order to be tested</param>
        /// <returns>Trivial</returns>
        private bool TestPerfectEleminationOrder(Person[] order, out Person[] triple)
        {
            //Initialize data structures
            Dictionary<Person, List<Person>> calcuatedLists = new Dictionary<Person,
List<Person>>();
            triple = new Person[3];
            SortAdjLists(order);
```

```
            foreach (Person person in Persons)
                calcuatedLists.Add(person, new List<Person>());

            for (int i = 0; i < Persons.Count; i++)
            {
                Person currentPerson = order[i];

                //List with the right neighbors of currentPerson (all neighbors with a
higher label)
                List<Person> rightNeigbors = new List<Person>();
                //Adjacent lists are ordered -> All right neighbors are in the upper
part of the neighbors
                int currentPos = currentPerson.Neighbors.Count - 1;
                while (currentPos >= 0 && currentPerson.Neighbors[currentPos].Position
> currentPerson.Position)
                {
                    rightNeigbors.Add(currentPerson.Neighbors[currentPos]);
                    currentPos--;
                }

                if(rightNeigbors.Count > 1)
                {
                    //Find element with the smallest position (== latest labeled) in
the right neighbors
                    int positionOfLeftmostElement = 0;
                    for (int a = 0; a < rightNeigbors.Count; a++)
                    {
                        if (rightNeigbors[positionOfLeftmostElement].Position >
rightNeigbors[a].Position)
                            positionOfLeftmostElement = a;
                    }

                    //Add every right neighbor to calculated adjacent list of the
leftmost element while avoiding the element itself
                    for (int a = 0; a < rightNeigbors.Count; a++)
                        if (a != positionOfLeftmostElement)

calcuatedLists[rightNeigbors[positionOfLeftmostElement]].Add(rightNeigbors[a]);
                }
            }

            //Check all elements for differ (backwards)
            //Used for getting the highest violating triple
            for (int i = Persons.Count - 1; i >= 0; i--)
            {

System.Diagnostics.Debug.WriteLine(calcuatedLists[order[i]].Count.ToString());
                if (Differ(order[i].Neighbors, calcuatedLists[order[i]]) != -1)
                {
                    triple[1] = order[i]; //v is the person with the different
adjacent lists
                    triple[2] = Persons[Differ(order[i].Neighbors,
calcuatedLists[order[i]])]; //w is the odd element
                    break; //Nothing more to check
                }
            }

            if (triple[2] != null) //Not chordal
            {
                //Common elements of the neighbors of v and w
                //One of them is u
                List<Person> potentialUs =
triple[1].Neighbors.Intersect(triple[2].Neighbors).ToList();
```

```
                //Search for the element with the smallest position (u)
                int positionOfCurrentU = 0;
                for (int i = 0; i < potentialUs.Count; i++)
                {
                    if (potentialUs[i].Position <
potentialUs[positionOfCurrentU].Position)
                        positionOfCurrentU = i;
                }

                //u is the common neighbor of v and w with the lowest position
                triple[0] = potentialUs[positionOfCurrentU];

                return false;
            }

            return true; //Chordal
        }

        /// <summary>
        /// Tests if two adjacent lists differ
        /// </summary>
        /// <param name="originalAdjList">Original adjacent list</param>
        /// <param name="calculatedAdjList">Calculated (by Perfect()) adjacent
list</param>
        /// <returns>-1 if the list don't differ or the odd element</returns>
        private int Differ(List<Person> originalAdjList, List<Person>
calculatedAdjList)
        {
            //Flag all persons
            foreach (Person person in originalAdjList)
                person.Flagged = true;

            //Check for an element only contained in the calculated adjacent list
            foreach (Person person in calculatedAdjList)
                if (!person.Flagged)
                    return person.ID; //Return the odd element

            //Unflag all persons
            foreach (Person person in originalAdjList)
                person.Flagged = false;

            return -1; //The lists don't differ
        }

        /// <summary>
        /// Sort all adjacent list in linear time
        /// </summary>
        /// <param name="order">A order of the graph</param>
        private void SortAdjLists(Person[] order)
        {
            //Initialize the required data structures
            Dictionary<Person, List<Person>> listOfAdjLists = new Dictionary<Person,
List<Person>>();
            foreach (Person person in Persons)
                listOfAdjLists.Add(person, new List<Person>());

            //Put the ordered persons in the new adjacent lists of their neighbors
            foreach (Person person in order)
                foreach (Person neighbor in person.Neighbors)
                    listOfAdjLists[neighbor].Add(person);

            //Overwrite the old adjacent lists with the new ones
```

```
            foreach (Person person in Persons)
                person.Neighbors = listOfAdjLists[person];
        }

        /// <summary>
        /// Generates a LexFS ordering of G used to find a path
        /// Used for finding an circle if G is not chordal
        /// </summary>
        /// <param name="triple">A triple containing u, v and w</param>
        /// <returns>An ordering leading to a path from v to w avoiding u</returns>
        private Person[] LexBFSFindAPath(Person[] triple)
        {
            //Initialize data structures
            LinkedList<SetList<Person>> listOfSets = new
LinkedList<SetList<Person>>();
            Dictionary<Person, LinkedListNode<SetList<Person>>> setPointers = new
Dictionary<Person, LinkedListNode<SetList<Person>>>();
            Dictionary<Person, LinkedListNode<Person>> positionPointers = new
Dictionary<Person, LinkedListNode<Person>>();

            //Set that have to be unflagged after the iteration
            List<LinkedListNode<SetList<Person>>> setsToCheck = new
List<LinkedListNode<SetList<Person>>>();

            Person[] outputArray = new Person[this.Persons.Count];

            //Set for all node without N(u)\{v,w}; w is at the first position
            SetList<Person> firstSet = new SetList<Person>();
            listOfSets.AddFirst(firstSet);

            //Set for N(u)\{v,w}; after the first set -> gets labeled last
            SetList<Person> secondSet = new SetList<Person>();
            listOfSets.AddFirst(secondSet);

            //Adds w so w gets labeled first
            positionPointers[triple[2]] = firstSet.AddFirst(triple[2]);
            setPointers[triple[2]] = listOfSets.Last;

            //Adds v separately
            positionPointers[triple[1]] = firstSet.AddLast(triple[1]);
            setPointers[triple[1]] = listOfSets.Last;

            //Set origin distance
            triple[2].Distance = 0;

            //Add N(u)\{v,w}
            foreach (Person person in triple[0].Neighbors)
            {
                if (person != triple[1] || person != triple[2])
                {
                    positionPointers[person] = secondSet.AddFirst(person);
                    setPointers[person] = listOfSets.First;
                }
            }

            //Add u
            positionPointers[triple[0]] = secondSet.AddFirst(triple[0]);
            setPointers[triple[0]] = listOfSets.First;

            //Get all persons ready
            foreach (Person person in Persons)
            {
                person.Position = -1;
```

```
                //Don't add neighbors of u, u, v or w
                if (!setPointers.ContainsKey(person))
                {
                    positionPointers[person] = firstSet.AddLast(person);
                    setPointers[person] = listOfSets.Last;
                }
            }

            //Label all persons
            for (int i = 0; i < Persons.Count; i++)
            {
                //Select current person for labeling and remove it from its set
                Person currentPerson = listOfSets.Last.Value.First.Value;
                listOfSets.Last.Value.RemoveFirst();
                positionPointers.Remove(currentPerson);

                //Remove set if empty
                if (listOfSets.Last.Value.Count == 0)
                    listOfSets.Remove(listOfSets.Last);

                //Assign label
                currentPerson.Position = i;
                outputArray[i] = currentPerson;

                //Assign new sets to all neighbors
                foreach (Person neighbor in currentPerson.Neighbors)
                {
                    //Ignore labeled neighbors
                    if (neighbor.Position == -1)
                    {
                        //Get current set of neighbor
                        LinkedListNode<SetList<Person>> currentSet =
setPointers[neighbor];

                        //If currentSet has no replacement generate replacement set
generate one
                        if (!currentSet.Value.HasReplacement)
                        {
                            currentSet.Value.HasReplacement = true;
                            SetList<Person> newSet = new SetList<Person>();
                            listOfSets.AddAfter(currentSet, newSet);

                            setsToCheck.Add(currentSet);
                        }

                        //Put neighbor in a set one level higher
                        currentSet.Value.Remove(positionPointers[neighbor]);
                        setPointers[neighbor] = currentSet.Next;
                        positionPointers[neighbor] =
currentSet.Next.Value.AddLast(neighbor);

                        //Set distance of current neighbor
                        neighbor.Distance = currentPerson.Distance + 1;
                    }
                }

                //Unflag all previously flagged set and delete them if empty
                foreach (LinkedListNode<SetList<Person>> setList in setsToCheck)
                {
                    if (setList.Value.Count == 0)
                        listOfSets.Remove(setList);
```

```
                    else
                        setList.Value.HasReplacement = false;
                }

                setsToCheck.Clear();
            }

            return outputArray;
        }

        /// <summary>
        /// Test if the given Graph is an interval graph
        /// </summary>
        /// <param name="output">Returns a invalid structure</param>
        /// <returns>true if the graph is a interval graph otherwise false</returns>
        public bool TestIntervalGraph(out Person[] output)
        {
            if (!TestChordal(out output))
                return false;

            output = LexBFS(Direction.Forwards);

            MPQTree tree = new MPQTree();
            for (int i = 0; i < output.Length; i++)
            {
                if (!tree.AddVertice(output[i]))
                    return FindAT(out output, output[i]);
            }

            tree.GenerateIntervalRepresentation();
            return true;
        }

        /// <summary>
        /// Finds a AT in a MPQTree of a chordal graph
        /// </summary>
        /// <param name="output">The AT</param>
        /// <param name="x">The person at whose consideration the building of the
MPQTree failed</param>
        private bool FindAT(out Person[] output, Person x)
        {
            uint[,] connectedComponentMap = new uint[Persons.Count, Persons.Count];
            uint highestLabel = 0;

            //We must consider every component of the current graph separately
            List<List<Person>> PLGComponents = findConnectedComponents(Persons);

            foreach (List<Person> component in PLGComponents)
            {
                foreach (Person person in component)
                {
                    foreach (Person p in component)
                        p.Flagged = false;

                    //Keep track of the persons still not labeled
                    HashSet<Person> personSet = new HashSet<Person>(component);

                    //Person is in one connected component with himself
                    personSet.Remove(person);
                    connectedComponentMap[person.ID, person.ID] = 0;
                    person.Flagged = true;

                    //Neighbors are in the same connected component as person
```

```
                    foreach (Person neighbor in person.Neighbors)
                    {
                        neighbor.Flagged = true;
                        connectedComponentMap[person.ID, neighbor.ID] = 0;
                        personSet.Remove(neighbor);
                    }

                    //Perform BFS
                    Queue<Person> queue = new Queue<Person>();

                    //Go on until all persons are considered
                    while (personSet.Count > 0)
                    {
                        if (queue.Count == 0)
                        {
                            highestLabel++;
                            queue.Enqueue(personSet.First());
                        }

                        //Get the first person (FIFO)
                        Person currentPerson = queue.Dequeue();

                        currentPerson.Flagged = true;
                        personSet.Remove(currentPerson);
                        connectedComponentMap[person.ID, currentPerson.ID] =
highestLabel;

                        //Enqueue all unvisited neighbors
                        foreach (Person neighbor in currentPerson.Neighbors)
                            if (!neighbor.Flagged)
                                queue.Enqueue(neighbor);
                    }
                }
            }

            //Check all triples in the graph
            //Because x must be part of the AT we must only search for y and z
            foreach (Person y in Persons)
            {
                if (y == x)
                    continue;

                foreach (Person z in Persons)
                {
                    if (z == y || z == x)
                        continue;

                    if (connectedComponentMap[x.ID, y.ID] ==
connectedComponentMap[x.ID, z.ID] && connectedComponentMap[y.ID, x.ID] ==
connectedComponentMap[y.ID, z.ID] && connectedComponentMap[z.ID, x.ID] ==
connectedComponentMap[z.ID, y.ID])
                    {
                        //AT found, return it
                        output = new Person[3];
                        output[0] = x;
                        output[1] = y;
                        output[2] = z;

                        return false;
                    }
                }
            }
```

```
            throw new InvalidOperationException("Only graphs containing an AT may be
checked");
        }

        /// <summary>
        /// Find all the paths between the nodes of the at
        /// </summary>
        /// <param name="at">An at</param>
        /// <returns>A list of 3 paths</returns>
        public List<List<Person>> FindATPaths(Person[] at)
        {
            //All combinations necessary for finding all paths
            Person[][] tripelCombinations = new Person[3][];

            tripelCombinations[0] = new Person[3];
            tripelCombinations[0][0] = at[2];
            tripelCombinations[0][1] = at[0];
            tripelCombinations[0][2] = at[1];

            tripelCombinations[1] = new Person[3];
            tripelCombinations[1][0] = at[1];
            tripelCombinations[1][1] = at[2];
            tripelCombinations[1][2] = at[0];

            tripelCombinations[2] = new Person[3];
            tripelCombinations[2][0] = at[0];
            tripelCombinations[2][1] = at[1];
            tripelCombinations[2][2] = at[2];

            List<List<Person>> returnList = new List<List<Person>>(3);

            //Find the path for every pair of the triple
            for (int i = 0; i < 3; i++)
            {
                returnList.Add(new List<Person>());

                Person[] order = LexBFSFindAPath(tripelCombinations[i]);
                SortAdjLists(order);

                Person currentPerson = tripelCombinations[i][1];

                while (currentPerson.Distance > 0)
                {
                    returnList[i].Add(currentPerson);

                    //Find first neighbor with a lower distance
                    foreach (Person neighbor in currentPerson.Neighbors)
                    {
                        if (neighbor.Distance < currentPerson.Distance)
                        {
                            currentPerson = neighbor;
                            break;
                        }
                    }
                }

                returnList[i].Add(tripelCombinations[i][2]);
            }

            return returnList;
        }

        /// <summary>
```

```
        /// Finds the connected components of the current graph
        /// </summary>
        /// <param name="persons">A list of persons representing a graph</param>
        public static List<List<Person>> findConnectedComponents(IEnumerable<Person>
persons)
        {
            //List of anchors for the graph
            Dictionary<Person, Person> anchors = new Dictionary<Person,
Person>(persons.Count());

            //Assign all anchors to the persons themselves
            foreach (Person person in persons)
                anchors.Add(person, person);

            //Enumerates through all persons
            foreach (Person person in persons)
            {
                //And their neighbors
                foreach (Person neighbor in person.Neighbors)
                {
                    //Ignore persons that have a lower ID (they were already
processed)
                    if (neighbor.ID < person.ID)
                        continue;

                    //Find the anchors of the current person and its neighbor
                    Person person1 = findAnchor(person, ref anchors);
                    Person person2 = findAnchor(neighbor, ref anchors);

                    //Overwrite the anchor with the higher ID
                    if (person1.ID < person2.ID)
                        anchors[person2] = person1;

                    else if (person2.ID < person1.ID)
                        anchors[person1] = person2;
                }


            }

            List<List<Person>> returnList = new List<List<Person>>();
            foreach (Person person in anchors.Values.Distinct())
                returnList.Add((from keyValuePair in anchors where keyValuePair.Value
== person select keyValuePair.Key).ToList());

            return returnList;
        }

        /// <summary>
        /// Finds the anchor of the current node
        /// Uses pathcompression
        /// </summary>
        /// <param name="node">The person of which the anchor should be
searched</param>
        /// <returns>The requested anchor</returns>
        private static Person findAnchor(Person node, ref Dictionary<Person, Person>
anchors)
        {
            Person start = node;
            while (node != anchors[node])
                node = anchors[node];
            anchors[start] = node;
            return node;
```

```
        }
    }

    /// <summary>
    /// Extended LinkedList
    /// Used for LexBFS
    /// </summary>
    internal class SetList<T> : LinkedList<T>
    {
        public bool HasReplacement = false;
    }

    /// <summary>
    /// Enum of sorting directions
    /// Used for LexBFS
    /// </summary>
    internal enum Direction
    {
        Forwards,
        Backwards,
    }
}
```

8.3 MPQTree.cs

```
using System;
using System.Collections.Generic;
using System.Linq;
using System.Diagnostics;

namespace PLG
{
    class MPQTree
    {
        #region Nodes
        abstract class BaseNode
        {
            protected MPQTree tree;

            protected BaseNode father;
            public BaseNode Father
            {
                get { return father; }
                [DebuggerStepThrough]
                set
                {
                    //Prevent a QNode from being a father, else possibility for a
wrong implementation later on
                    if (!(value is QNode) || this is Section)
                    {
                        if (father != null)
                            father.RemoveSon(this);

                        father = value;

                        if (value == null)
                            tree.root = this;

                        else value.SetSon(this);

                        if (this is PNode)
                            tree.manipulatedPNodes.Add(this as PNode);
                    }
```

```
                else if (!(this is Section))
                    throw new FormatException("Father may not be a QNode");
            }
        }

        public LinkedList<Person> ContainedPersons;

        [DebuggerStepThrough]
        public BaseNode(MPQTree tree)
        {
            ContainedPersons = new LinkedList<Person>();
            this.tree = tree;
        }

        [DebuggerStepThrough]
        public virtual void AddContainedPerson(Person p)
        {
            tree.nodePointer[p] = this;
            tree.positionPointers[p] = ContainedPersons.AddLast(p);
        }

        [DebuggerStepThrough]
        public void AddContainedRange(IEnumerable<Person> persons)
        {
            foreach (Person p in persons)
                AddContainedPerson(p);
        }

        [DebuggerStepThrough]
        protected virtual void RemoveSon(BaseNode son)
        {

        }

        [DebuggerStepThrough]
        protected virtual void SetSon(BaseNode son)
        {

        }

        [DebuggerStepThrough]
        public void Vanish()
        {
            Father = this;
        }
    }

    class PNode : BaseNode
    {
        private bool ignoreSetSon = false;

        public LinkedList<BaseNode> Children;
        public BaseNode SomeSon;

        [DebuggerStepThrough]
        public PNode(MPQTree tree)
            : base(tree)
        {
            Children = new LinkedList<BaseNode>();
            tree.manipulatedPNodes.Add(this);
        }

        [DebuggerStepThrough]
```

```
        public void AddLastChild(BaseNode child)
        {
            if (child is Section)
                throw new FormatException("child mustn't be a section");

            ignoreSetSon = true;
            child.Father = this;
            ignoreSetSon = false;

            Children.AddLast(child);
        }

        [DebuggerStepThrough]
        public void AddFirstChild(BaseNode child)
        {
            if (child is Section)
                throw new FormatException("child mustn't be a section");

            ignoreSetSon = true;
            child.Father = this;
            ignoreSetSon = false;

            Children.AddFirst(child);
        }

        [DebuggerStepThrough]
        protected override void SetSon(BaseNode son)
        {
            if (!ignoreSetSon)
            {
                if (son is Section)
                    throw new FormatException("child mustn't be a section");

                Children.AddLast(son);
            }

            SomeSon = son;
        }

        [DebuggerStepThrough]
        protected override void RemoveSon(BaseNode son)
        {
            if (Children.Remove(son))
            {
                tree.manipulatedPNodes.Add(this);
            }

            else throw new InvalidOperationException("Son is not in Children");
        }
    }

    class QNode : BaseNode
    {
        public List<Section> Sections;
        public Section OuterSectionRight, OuterSectionLeft;

        [DebuggerStepThrough]
        public QNode(MPQTree tree)
            : base(tree)
        {
            Sections = new List<Section>();
        }
```

```
        [DebuggerStepThrough]
        public Section AddNewLastSection()
        {
            Section section = new Section(tree);
            if (Sections.Count != 0)
            {
                section.LeftNeighbor = OuterSectionRight;
                OuterSectionRight.RightNeighbor = section;
            }

            else
                OuterSectionLeft = section;

            OuterSectionRight = section;
            Sections.Add(section);

            section.Father = this;

            return section;
        }

        [DebuggerStepThrough]
        public Section AddNewFirstSection()
        {
            Section section = new Section(tree);
            if (Sections.Count != 0)
            {
                section.RightNeighbor = OuterSectionLeft;
                OuterSectionLeft.LeftNeighbor = section;
            }

            else
                OuterSectionRight = section;

            OuterSectionLeft = section;
            Sections.Add(section);

            section.Father = this;

            return section;
        }

        [DebuggerStepThrough]
        protected override void SetSon(BaseNode son)
        {
            if (!(son is Section))
                throw new FormatException("Only sections may be sons of QNodes");
        }

        [DebuggerStepThrough]
        protected override void RemoveSon(BaseNode son)
        {
            if (!(son is Section))
                throw new FormatException("Only sections may be sons of QNodes");

            if (son.Father != this)
                throw new InvalidOperationException("Son must be part of current
QNode");

            RemoveSection(son as Section);
        }

        [DebuggerStepThrough]
```

```
            public void RemoveSection(Section section)
            {
                Sections.Remove(section);

                if (section.RightNeighbor == null)
                    OuterSectionRight = section.LeftNeighbor;
                else
                    section.RightNeighbor.LeftNeighbor = section.LeftNeighbor;

                if (section.LeftNeighbor == null)
                    OuterSectionLeft = section.RightNeighbor;
                else
                    section.LeftNeighbor.RightNeighbor = section.RightNeighbor;

                Sections.Remove(section);
            }
        }

        class Section : BaseNode
        {
            public BaseNode Son;
            public Section LeftNeighbor, RightNeighbor;

            [DebuggerStepThrough]
            public Section(MPQTree tree)
                : base(tree)
            { }

            [DebuggerStepThrough]
            protected override void SetSon(BaseNode son)
            {
                if (son is Section)
                    throw new FormatException("child mustn't be a section");

                Son = son;
            }

            [DebuggerStepThrough]
            public override void AddContainedPerson(Person p)
            {
                //Only the outer section is important if the person has already been
added
                if (LeftNeighbor == null || RightNeighbor == null ||
!tree.positionPointers.ContainsKey(p))
                    base.AddContainedPerson(p);

                else
                    ContainedPersons.AddLast(p);
            }

            [DebuggerStepThrough]
            protected override void RemoveSon(BaseNode son)
            {
                if (son is Section)
                    throw new FormatException("child mustn't be a section");

                if (Son == son)
                    Son = null;
            }
        }

        class Leaf : BaseNode
        {
```

```
            [DebuggerStepThrough]
            public Leaf(MPQTree tree)
                : base(tree)
            { }
        }
        #endregion

        BaseNode root;
        Dictionary<Person, BaseNode> nodePointer;
        Dictionary<Person, LinkedListNode<Person>> positionPointers;
        HashSet<PNode> manipulatedPNodes;

        public MPQTree()
        {
            nodePointer = new Dictionary<Person, BaseNode>();
            positionPointers = new Dictionary<Person, LinkedListNode<Person>>();
            manipulatedPNodes = new HashSet<PNode>();
        }

        /// <summary>
        /// Add a person to the MPQ-Tree
        /// If this fails the graph is no interval graph
        /// </summary>
        /// <param name="person"></param>
        /// <returns>If false adding the person fails</returns>
        public bool AddVertice(Person person)
        {
            //Can be omitted if person has no neighbors or there are no nodes
            if (person.Neighbors.Count > 0 && root != null)
            {
                //nDown is the lowest node on P with label 1 or infinity
                //If there is a node with the label 0 or 1 let nUp be the highest such
node else nUp is nDown
                //nCurrent is the current node between nDown and nUp
                BaseNode nUp = null, nDown, nCurrent;

                //List of the persons in A for a certain node
                Dictionary<BaseNode, List<Person>> ALists = new Dictionary<BaseNode,
List<Person>>();

                //Represents a path P from the root to a positively labeled leaf
                Dictionary<BaseNode, BaseNode> rootedPath = new Dictionary<BaseNode,
BaseNode>();

                Dictionary<BaseNode, bool> flagMap = new Dictionary<BaseNode, bool>();

                foreach (BaseNode node in nodePointer.Values)
                {
                    flagMap[node] = false;
                }

                //All nodes that have to be check whether they are contained on a path
                Queue<BaseNode> QueueToCheck = new Queue<BaseNode>();

                //Check all neighbors
                foreach (Person neighbor in person.Neighbors)
                {
                    //Ignore persons who are not part of the tree
                    if (neighbor.Position < person.Position)
                    {
                        //The node the current neighbor belongs to
                        BaseNode associatedNode = nodePointer[neighbor];
```

```
                        if (associatedNode is Section)
                        {
                            Section associatedSection = associatedNode as Section;

                            //The neighbor must be contained in an outer section
                            if (associatedSection.RightNeighbor == null &&
associatedSection.LeftNeighbor == null)
                                return false;
                        }

                        //Creates a new A list for the current node
                        if (!ALists.ContainsKey(associatedNode))
                            ALists.Add(associatedNode, new List<Person>());

                        //Adds the current neighbor to the A list
                        ALists[associatedNode].Add(neighbor);
                        //Remove the current neighbor from the B list

associatedNode.ContainedPersons.Remove(positionPointers[neighbor]);

                        QueueToCheck.Enqueue(associatedNode);
                    }
                }

                //Current person has no neighbors, which are in the tree
                if (QueueToCheck.Count == 0)
                {
                    //We have already a PNode without contained persons as root -> no
need to create one
                    if (root is PNode && root.ContainedPersons.Count == 0)
                    {
                        Leaf leaf = new Leaf(this);
                        leaf.AddContainedPerson(person);
                        leaf.Father = root;
                    }

                    //Create a new PNode without contained persons
                    else
                    {
                        PNode newRoot = new PNode(this);
                        newRoot.AddFirstChild(root);
                        newRoot.Father = null;

                        Leaf leaf = new Leaf(this);
                        leaf.Father = newRoot;
                        leaf.AddContainedPerson(person);
                    }

                    return true;
                }

                BaseNode currentNode;

                //Flag all elements on QueueToCheck
                while (QueueToCheck.Count > 0)
                {
                    currentNode = QueueToCheck.Dequeue();

                    //Node not yet visited
                    flagMap[currentNode] = true;

                    if (currentNode.Father != null)
                    {
```

```
                        //Father has to be flagged too
                        QueueToCheck.Enqueue(currentNode.Father);

                        //All nodes must be on a path -> if the current node is not a
duplicate it may not be on the rooted path
                        if (!(currentNode is Section))
                        {
                            if (rootedPath.ContainsKey(currentNode.Father) &&
rootedPath[currentNode.Father] != currentNode)
                                return false;
                        }

                        //Things are different with section: There may be multiple
sections of one QNode, but only one of them may be a outer section
                        else if (rootedPath.ContainsKey(currentNode.Father) &&
currentNode != rootedPath[currentNode.Father])
                        {
                            Section currentSection = currentNode as Section;
                            Section otherContainedSection =
rootedPath[currentNode.Father] as Section;

                            if (currentSection.LeftNeighbor == null ||
currentSection.RightNeighbor == null)
                                if (!(otherContainedSection.LeftNeighbor == null ||
otherContainedSection.RightNeighbor == null))
                                    rootedPath[currentNode.Father] = currentNode;
                                else return false; //There is no path
                        }

                        //Keep a link from the father to its son
                        rootedPath[currentNode.Father] = currentNode;
                    }
                }

                //The path must start at the root
                currentNode = root;

                //Travers the rooted path downwards to find nUp
                while (rootedPath.ContainsKey(currentNode))
                {
                    //nUp not found yet
                    if (nUp == null)
                    {
                        //B mustn't be empty (-> label is not Infinity)
                        if (currentNode.ContainedPersons.Count > 0)
                        {
                            if (currentNode is Section)
                            {
                                //The path must go through a outer section
                                Section currentSection = currentNode as Section;
                                if (currentSection.LeftNeighbor != null &&
currentSection.RightNeighbor != null)
                                    return false;

                                //nUp must be a qNode containing the Section
fulfilling the condition
                                nUp = currentNode.Father as QNode;
                            }

                            else if (currentNode is PNode)
                            {
                                nUp = currentNode;
                            }
```

```
                        }
                    }

                    currentNode = rootedPath[currentNode];
                }

                //If currentNode is a Section get the associated QNode
                if (currentNode is Section)
                    currentNode = currentNode.Father;

                //nDown is the lowest node on the path P
                nDown = currentNode;

                //If there is no node matching the conditions for nUp, nUp is nDown
                if (nUp == null)
                    nUp = nDown;

                //nCurrent must be traversed bottom-up (-> is set to nDown at start)
                nCurrent = nDown;

                //The path between nDown and nUp
                Dictionary<BaseNode, BaseNode> progressionPath = new
Dictionary<BaseNode, BaseNode>();

                currentNode = nDown;

                //Build a path from nDown to nUp
                if (nUp == nDown)
                    progressionPath.Add(nDown, null);

                else
                {
                    while(currentNode != null)
                    {
                        progressionPath.Add(currentNode, currentNode.Father);
                        currentNode = currentNode.Father;
                    }
                }

                //Apply templates
                do
                {
                    if (!ALists.ContainsKey(nCurrent))
                    {
                        //AList are generated for Sections -> need to be transformed
to QNodes
                        if (nCurrent is QNode)
                        {
                            QNode currentQNode = nCurrent as QNode;

                            if (ALists.ContainsKey(currentQNode.OuterSectionLeft))
                            {
                                ALists[nCurrent] =
ALists[currentQNode.OuterSectionLeft];
                                ALists.Remove(currentQNode.OuterSectionLeft);
                            }

                            else if
(ALists.ContainsKey(currentQNode.OuterSectionRight))
                            {
                                ALists[nCurrent] =
ALists[currentQNode.OuterSectionRight];
                                ALists.Remove(currentQNode.OuterSectionRight);
```

```
            }

            else
                ALists[nCurrent] = new List<Person>();
        }

        else
            ALists[nCurrent] = new List<Person>();
    }

    //Templates L1 and L2
    if (nCurrent is Leaf)
    {
        //Template L1
        if (nUp == nDown)
        {
            //B is empty
            if (nCurrent.ContainedPersons.Count == 0)
            {
                nCurrent.AddContainedRange(ALists[nCurrent]);

                nCurrent.AddContainedPerson(person);
            }

            //B is not empty
            else
            {
                PNode pNode = new PNode(this);
                pNode.Father = nCurrent.Father;

                Leaf leaf = new Leaf(this);
                leaf.Father = pNode;

                leaf.AddContainedPerson(person);

                pNode.AddLastChild(nCurrent);

                pNode.AddContainedRange(ALists[nCurrent]);
            }
        }

        //Template L2
        else
        {
            QNode qNode = new QNode(this);
            qNode.Father = nCurrent.Father;

            Section section1 = qNode.AddNewLastSection();
            Section section2 = qNode.AddNewLastSection();

            section1.AddContainedRange(ALists[nCurrent]);
            section2.AddContainedRange(ALists[nCurrent]);

            Leaf leaf = new Leaf(this);
            leaf.Father = section1;
            leaf.AddContainedPerson(person);

            nCurrent.Father = section2;
        }
    }

    //Templates P1, P2 and P3
    else if (nCurrent is PNode)
```

```
                    {
                        //Template P1
                        if (nUp == nDown)
                        {
                            //B is empty
                            if (nCurrent.ContainedPersons.Count == 0)
                            {
                                Leaf leaf = new Leaf(this);
                                leaf.AddContainedPerson(person);
                                leaf.Father = nCurrent;

                                nCurrent.AddContainedRange(ALists[nCurrent]);
                            }

                            //B is not empty
                            else
                            {
                                PNode pNode = new PNode(this);
                                pNode.Father = nCurrent.Father;

                                Leaf leaf = new Leaf(this);
                                leaf.AddContainedPerson(person);

                                leaf.Father = pNode;
                                nCurrent.Father = pNode;
                            }
                        }

                        //Template P2
                        else if (nCurrent == nDown) //No need to check nDown != nUp
(previously checked)
                        {
                            QNode qNode = new QNode(this);
                            qNode.Father = nCurrent.Father;

                            Section section1 = qNode.AddNewLastSection();
                            Section section2 = qNode.AddNewLastSection();

                            Leaf leaf = new Leaf(this);
                            leaf.AddContainedPerson(person);
                            leaf.Father = section1;

                            nCurrent.Father = section2;

                            section1.AddContainedRange(ALists[nCurrent]);
                            section2.AddContainedRange(ALists[nCurrent]);
                        }

                        //Template P3
                        else if (nCurrent != nDown)
                        {
                            PNode currentPNode = nCurrent as PNode;
                            QNode currentQNode = currentPNode.SomeSon as QNode;

                            //No need to edit the father correctly; this will happen
in Q3
                            if (currentPNode.Father is Section)
                                foreach (Person p in
currentPNode.Father.ContainedPersons)
                                    currentPNode.ContainedPersons.AddLast(p);

                            if (currentQNode != null &&
currentQNode.OuterSectionLeft.Son.ContainedPersons.First.Value == person)
```

```
                            {
                                QNode newQNode = new QNode(this);
                                newQNode.Father = currentPNode.Father;

                                Section currentSection =
currentQNode.OuterSectionLeft.RightNeighbor;
                                while (currentSection != null)
                                {
                                    Section newSection = newQNode.AddNewLastSection();
                                    currentSection.Son.Father = newSection;

                                    IEnumerable<Person> newContainedPersons =
ALists[nCurrent].Union(nCurrent.ContainedPersons).Union(currentSection.ContainedPerson
s);
                                    newSection.AddContainedRange(newContainedPersons);

                                    currentSection = currentSection.RightNeighbor;
                                }

                                Section lastNewSection = newQNode.AddNewLastSection();

                                Section section1 = newQNode.AddNewFirstSection();
                                currentQNode.OuterSectionLeft.Son.Father = section1;


section1.AddContainedRange(ALists[nCurrent].Union(currentQNode.OuterSectionLeft.Contai
nedPersons));

                                PNode newPNode = new PNode(this);
                                newPNode.Father = lastNewSection;


lastNewSection.AddContainedRange(ALists[nCurrent].Union(nCurrent.ContainedPersons));

                                foreach (BaseNode node in
currentPNode.Children.ToArray())
                                    if (node != currentQNode)
                                        newPNode.AddLastChild(node);
                            }

                                currentPNode.Vanish();
                        }
                    }

                    //Templates Q1, Q2 and Q3
                    else if (nCurrent is QNode)
                    {
                        //Templates Q1 and Q2
                        if(nCurrent == nDown)
                        {
                            QNode currentQNode = currentNode as QNode;

                            //Template Q1

if(!ALists[currentQNode].Except(currentQNode.OuterSectionRight.ContainedPersons).Any()
)
                            {
                                if(nUp == nDown)
                                {
                                    PNode pNode = new PNode(this);
                                    pNode.Father = nCurrent.Father;

                                    Leaf leaf = new Leaf(this);
```

```
                                    pNode.AddFirstChild(leaf);
                                    leaf.AddContainedPerson(person);

                                    pNode.AddContainedRange(ALists[nCurrent]);

                                    Section currentSection =
currentQNode.OuterSectionLeft;
                                    while(currentSection != null)
                                    {
                                        //Removes a from all sections
                                        currentSection.ContainedPersons = new
LinkedList<Person>(currentSection.ContainedPersons.Except(ALists[nCurrent]));

                                        currentSection = currentSection.RightNeighbor;
                                    }

                                    pNode.AddLastChild(currentQNode);
                                }

                                else
                                {
                                    QNode newQNode = new QNode(this);
                                    newQNode.Father = nCurrent.Father;

                                    Section firstSection =
newQNode.AddNewLastSection();
                                    Section secondSection =
newQNode.AddNewLastSection();

                                    Leaf leaf = new Leaf(this);
                                    leaf.Father = firstSection;
                                    leaf.AddContainedPerson(person);

                                    firstSection.AddContainedRange(ALists[nCurrent]);
                                    secondSection.AddContainedRange(ALists[nCurrent]);

                                    Section currentSection =
currentQNode.OuterSectionLeft;
                                    while (currentSection != null)
                                    {
                                        //Removes a from all sections
                                        currentSection.ContainedPersons = new
LinkedList<Person>(currentSection.ContainedPersons.Except(ALists[nCurrent]));

                                        currentSection = currentSection.RightNeighbor;
                                    }

                                    currentQNode.Father = secondSection;
                                }
                            }

                            //Template Q2
                            else
                            {
                                //B is empty
                                if (nCurrent.ContainedPersons.Count == 0)
                                {
                                    PNode pNode = new PNode(this);


pNode.AddFirstChild(currentQNode.OuterSectionLeft.Son);
                                    pNode.Father = currentQNode.OuterSectionLeft;
```

```
currentQNode.OuterSectionLeft.AddContainedRange(ALists[nCurrent]);

                                    Leaf leaf = new Leaf(this);
                                    leaf.AddContainedPerson(person);
                                    pNode.AddFirstChild(leaf);
                                }

                                else if (nCurrent.ContainedPersons.Count == 0 || nUp
!= nDown)
                                {
                                    Section section =
currentQNode.AddNewFirstSection();
                                    section.AddContainedRange(ALists[nCurrent]);

                                    Leaf leaf = new Leaf(this);
                                    leaf.Father = section;
                                    leaf.AddContainedPerson(person);
                                }
                            }
                        }

                        //Template Q3
                        else
                        {
                            QNode currentQNode = nCurrent as QNode;
                            QNode currentLowerQNode = null;

                            Section parentSection;
                            parentSection = currentQNode.OuterSectionLeft as Section;

                            if (currentQNode.OuterSectionLeft.Son is QNode)
                            {
                                parentSection = currentQNode.OuterSectionLeft;
                                currentLowerQNode = parentSection.Son as QNode;
                            }

                            if(currentLowerQNode != null &&
currentLowerQNode.OuterSectionLeft.Son.ContainedPersons.First.Value == person)
                            {
                                currentQNode.RemoveSection(parentSection);

                                Section currentSection =
currentLowerQNode.OuterSectionRight;
                                while (currentSection.LeftNeighbor != null)
                                {
                                    Section newSection =
currentQNode.AddNewFirstSection();
                                    if (currentSection.Son != null)
                                        currentSection.Son.Father = newSection;

                                    IEnumerable<Person> newContainedPersons =
ALists[nCurrent].Union(nCurrent.ContainedPersons).Union(currentSection.ContainedPerson
s);
                                    newSection.AddContainedRange(newContainedPersons);

                                    currentSection = currentSection.LeftNeighbor;
                                }

                                Section firstNewSection;
                                firstNewSection = currentQNode.AddNewFirstSection();
```

```
                                currentLowerQNode.OuterSectionLeft.Son.Father =
firstNewSection;


firstNewSection.AddContainedRange(currentLowerQNode.OuterSectionLeft.ContainedPersons.
Union(ALists[nCurrent]));
                            }
                        }
                    }

                    //Remove the AList of nCurrent -> every AList contained in the end
has to be written back
                    ALists.Remove(nCurrent);

                    //Go one node up
                    nCurrent = progressionPath[nCurrent];

                    //If nCurrent now is a Section get the associated QNode
                    if (nCurrent is Section)
                    {
                        Section currentSection = nCurrent as Section;
                        nCurrent = currentSection.Father;
                    }
                }
                while (nCurrent != null);

                //Add the unused ALists
                foreach (BaseNode baseNode in ALists.Keys)
                    baseNode.AddContainedRange(ALists[baseNode]);

                //Check all manipulated PNodes
                //Helper templates might be necessary
                foreach (PNode pNode in manipulatedPNodes.ToArray())
                {
                    //Template H1
                    if (pNode.ContainedPersons.Count == 0 && pNode.Children.Count == 1
&& pNode.Father != pNode)
                    {
                        pNode.Children.First.Value.Father = pNode.Father;
                        pNode.Vanish();
                    }

                    //Template H2
                    else if (pNode.Father is PNode)
                    {
                        PNode fatherPNode = pNode.Father as PNode;
                        if (fatherPNode.ContainedPersons.Count == 0 &&
pNode.ContainedPersons.Count == 0)
                        {
                            foreach (BaseNode node in pNode.Children.ToArray())
                            {
                                fatherPNode.AddLastChild(node);
                            }

                            pNode.Vanish();
                        }
                    }
                }

                manipulatedPNodes.Clear();
            }

            else
```

```
            {
                if (person.Neighbors.Count == 0 && root != null)
                {
                    //We have already a PNode without contained persons as root -> no
need to create one
                    if (root is PNode && root.ContainedPersons.Count == 0)
                    {
                        Leaf leaf = new Leaf(this);
                        leaf.AddContainedPerson(person);
                        leaf.Father = root;
                    }

                    //Create a new PNode without contained persons
                    else
                    {
                        PNode newRoot = new PNode(this);
                        newRoot.AddFirstChild(root);
                        newRoot.Father = null;

                        Leaf leaf = new Leaf(this);
                        leaf.Father = newRoot;
                        leaf.AddContainedPerson(person);
                    }
                }

                //Create a new leaf
                else if (root == null)
                {
                    Leaf leaf = new Leaf(this);
                    leaf.Father = root;
                    leaf.AddContainedPerson(person);
                }
            }

            return true;
        }

        /// <summary>
        /// Generates the intervalrepresentation of the current MPQ-Tree
        /// </summary>
        public void GenerateIntervalRepresentation()
        {
            List<List<Person>> cliquesList = new List<List<Person>>();

            //Perform recursive backtracking on the current MPQ-Tree
            //Gets a list with all maximum cliques
            Visit(root, ref cliquesList);

            //Current position in the current interval representation
            int currentPosition = 1;

            //Enumerate through all cliques
            foreach (List<Person> section in cliquesList)
            {
                //Enumerate through all persons in the current clique
                foreach (Person person in section)
                {
                    //When person has no birth set the birth to the current position
in the interval representation
                    if (person.Birth == 0)
                        person.Birth = currentPosition;

                    if (person.Death == 0)
```

```
                        person.Death = currentPosition + 1;

                    //This person occurs more than once -> increase the end in the
interval representation
                    else
                        person.Death++;
                }

                currentPosition++;
            }
        }

        /// <summary>
        /// DFS used for traversing through any path of the MPQTree and get thus get
the maximal cliques of the associated graph
        /// </summary>
        /// <param name="cliquesList">A reference to the clique list</param>
        /// <returns>All indices included by the current subtree</returns>
        private List<int> Visit(BaseNode currentNode, ref List<List<Person>>
cliquesList)
        {
            List<int> returnList = new List<int>();

            //End of path
            if (currentNode is Leaf)
            {
                cliquesList.Add(new List<Person>());
                returnList.Add(cliquesList.Count - 1);
            }

            else if (currentNode is PNode)
            {
                PNode currentPNode = currentNode as PNode;

                //Enumerate trough all children and get their sections
                foreach (BaseNode child in currentPNode.Children)
                    returnList.AddRange(Visit(child, ref cliquesList));
            }

            else if (currentNode is QNode)
            {
                QNode currentQNode = currentNode as QNode;
                Section currentSection = currentQNode.OuterSectionLeft;

                //Enumerate through all sections from left to right
                while (currentSection != null)
                {
                    returnList.AddRange(Visit(currentSection, ref cliquesList));
                    currentSection = currentSection.RightNeighbor;
                }

                //No need to add contained persons (QNode mustn't have any)
                return returnList;
            }

            else if (currentNode is Section)
            {
                Section currentSection  = currentNode as Section;
                returnList.AddRange(Visit(currentSection.Son, ref cliquesList));
            }

            //Add the contained persons to all child cliques
            foreach (int i in returnList)
```

```
                cliquesList[i].AddRange(currentNode.ContainedPersons);

            return returnList;
        }
    }
}
```

8.4 MainWindow.cs (Auszug)

```
void backgroundWorker_DoWork(object sender, DoWorkEventArgs e)
        {
            Plg plg;
            lock (persons)
            {
                plg = new Plg(persons);
            }

            Person[] output;
            string outString;

            if (!plg.TestIntervalGraph(out output))
            {
                if (output.Length == 3)
                    outString = "AT gefunden: \r\n";
                else
                    outString = "Sehnenloser Kreis gefunden:\r\n";

                foreach (Person p in output)
                {
                    outString += p.Name;
                    outString += "\r\n";
                }

                if (output.Length == 3)
                {
                    var result = plg.FindATPaths(output);
                    foreach (var list in result)
                    {
                        outString += "Pfad:\r\nStart:\r\n";

                        foreach (Person p in list)
                            outString += p.Name + "\r\n";

                        outString += ":Ende\r\n";
                    }
                }
            }

            else
            {
                outString = "Intervalldarstellung erzeugt:\r\n";
                foreach (Person p in persons)
                    outString += String.Format("{0}: Von {1} bis {2}\r\n", p.Name,
p.Birth, p.Death);
            }

            e.Result = outString;
        }
```