Universität Passau
Fakultät für Informatik und Mathematik

Bachelor Arbeit

Optimierung von BDD-basierten Modelchecking durch effiziente Variablenordnung

Autor:
Andreas Buchecker

Betreuer:
Dr. Ing. Sven Apel
Alexander von Rhein

12. Dezember 2012

Zusammenfassung

Binäre Entscheidungsbäume (BDD) können boolsche Funktionen kompakt darstellen und werden daher gern für Softwareverifikation genutzt. Allerdings ist die Größe eines BDD stark von seiner Variablenordnung abhängig. In dieser Bachelorarbeit untersuchen wir welche Auswirkungen die Variablenordnung auf die Laufzeit von Verifikation haben kann. Dazu verwenden wir die Java Pathfinder-Erweiterung JPF-BDD. Da dass finden der optimalen Variablenordnung für ein BDD ein NP-Hartes Problem ist entwickeln wir einen heuristischen Algorithmus der eine gute Variablenordnung berechnen soll. Um diese Heuristik zu entwickeln verwenden wir den Kommunikationsgraphen, der die Beziehung zwischen boolschen Variablen im zu verifizierenden System darstellt. Außerdem implementieren wir diesen Algorithmus und testen diesen mit einer Testreihe. Damit zeigen wir, dass der Algorithmus in vielen Fällen eine gute Variablenordnung berechnet.

Inhaltsverzeichnis

1 Motivation **2**

2 JPF-BDD **3**

3 Kommunikationsgraph **4**

4 Experimente zur Optimierung der Variablenordnung **6**
4.1 ExperimentA . . . 9
4.2 ExperimentB . . . 12
4.3 ExperimentC . . . 15
4.4 ExperimentD . . . 20
4.5 ExperimentE . . . 23
4.6 Experiment F, G, H . . . 27
4.7 Algorithmus für die Variablenordnung . . . 30

5 Testreihe zur Prüfung der Optimierung **31**
5.1 Auswertung des Testreihe . . . 32

6 Grenzen des Algorithmus **35**

7 Ausblick **36**

8 Fazit **37**

A Anhang **39**

1 Motivation

Da Softwaresysteme immer größer werden und ein Fehlschlag in einem Softwareprojekt viel Geld kostet, gewinnt Softwareverifikation immer mehr an Bedeutung. Bei Softwareverifikation wird getestet ob ein System eine bestimmte Spezifikation erfüllt. Dies geschieht häufig durch erforschen des Zustandsraum vom System. Da die Softwaresysteme aber immer größer werden steigt die Anzahl der möglichen Zustände eines System sehr schnell. Besonders bei parallelen Systemen steigt die Anzahl der Zustände exponentiell an. Dieses Problem wird als Zustandsexplosion bezeichnet.. Ein Ansatz mit dieser Zustandsexplosion umzugehen, ist eine symbolische Repräsentation des Zustandraums vom zu verifizierenden System. Eine häufig verwendete Datenstruktur dafür sind binäre Entscheidungsbäume (BDD).

Ein BDD ist eine graphenbasierte Struktur, die verwendet wird um logische Funktionen zu repräsentieren. BDD's können auch dazu verwendet werden, um die Transitionsrelation von binär kodierten sequentiellen Maschinen zu repräsentieren [7]. BDD werden, wegen ihrer kompakten Größe gerne in Soft-und Hardwareverfikation eingesetzt. Bryant [4] hat gezeigt, dass die Größe eines BDD stark von der gewählten Variablenordnung abhängen. Diese Unterschiede können sogar zwischen linearem und exponentiellem Wachstum, in Abhängigkeit der Anzahl der Variablen variieren.

Ein Tool, das BDD's bei Softwareverifikation verwendet ist die Erweiterung JPF-BDD für JavaPathfinder[10]. Die Frage die wir in dieser Arbeit untersuchen ist, ob die Variablenordnung signifikante Auswirkungen auf die Laufzeit von JPF-BDD haben kann. Dessweiteren wollen wir auch eine optimale Variablenordnung angeben, wenn dadurch die Laufzeit reduziert werden kann.

Die Berechnung, der optimalen Variablenordnung für ein BDD, ist allerdings ein NP-Hartes Problem [3]. Darum ist das zweite Ziel dieser Arbeit einen heuristischen Algorithmus zu entwickeln, der uns eine gute Variablenordnung liefert. Es gibt bereits einige Arbeiten die versuchen gute Variablenordnungen zu finden. Jeong [5] gibt einen effizienten Algorithmus zum berechnen einer Ordnung an, der auf der algebraische Struktur der Folgezustandsfunktion basiert. In einem weiteren Ansatz erweitert Rudell [6] eine BDD-Bibliothek um eine automatische Ordnung der Variablenordnung. Diese Ordnung versucht die Anzahl der BDD-Knoten zu verringern, indem kleine Mengen von angrenzenden Variablen umgeordnet werden.

Den Ansatz den wir in dieser Arbeit verwenden ist der Kommunikationsgraph. Adnan Aziz [1] verwendet den Kommunkationsgraph für endliche Zustandsautomaten . Wir verwenden die Idee des Kommunkationsgraphen um den Quellcode eines Programms, dass verifiziert werden soll, zu untersuchen. Dabei soll der Kommunikationsgraph die Beziehungen der Variablen darstellen, die mit BDD behandelt werden. Aus diesen Beziehungen versuchen wir abzuleiten wie eine gute Variablenordnung für dieses BDD aussieht.

2 JPF-BDD

JPF-BDD ist eine Erweiterung von JavaPathfinder [10]. JavaPathfinder [9] ist eine Virtuelle Maschine für Java die zu einem Modellchecker für Java Programme konfiguriert werden kann. Dabei prüft JavaPathfinder den gesamten Zustandsraum des zu verifizierenden Systems, indem es das Programm ausführt und einen Entscheidungspunkt erstellt wenn mehrere Ausführungspfade möglich sind.

JPF-BDD teilt die Zustände des Zustandsraum in 2 Teile auf [10]. Der erste Teil beinhaltet die Kernvariablen von JPF (Programmzähler, usw.) und die Variablen, die nicht mit der speziellen JPF-BDD Anmerkung gekennzeichnet sind. Dieser Teil wird von JPF behandelt.

Der zweite Teile beinhaltet eine Menge von boolschen Variablen, die der Benutzer mit einer spezielle Anmerkung im Programm, das verifiziert wird, kennzeichnet. Die Werte dieser Variablen werden in BDD-Variablen gespeichert. Dadurch erreicht JPF-BDD eine Reihe von Optimierungen in Vergleich zu JPF.

- Die Werte der verwalteten Variablen werden nur gewählt, falls und erst dann wenn sie verwendet werden.
- Es wird eine komprimierte Darstellung der Werte der verwalteten Variablen erstellt. Dadurch können diese schnell ausgelesen werden.
- Zustände, die sich nur in den Werten der von JPF-BDD verwalteten Variablen unterscheiden werden zusammengefasst.

Eine für unsere Experimente wichtige Eigenschaft von JPF-BDD ist, dass JPF-BDD den boolschen Variablen, die es verwaltet erst einen Wert zuweist, wenn sie benutzt werden. Dabei ist vorallem wichtig, dass JPF-BDD bei einer boolschen Variable beide Belegungen betrachtet, wenn ihr im Programm kein Wert zugewiesen wird. Die Standardversion von JPF würde an dieser Stelle, von der normalen Java Semantik ausgehend, der Variable den Wert false zuteilen. Warum diese Eigenschaft für unsere Experimente wichtig ist, erklären wir wenn wir die Experimente vorstellen.

Um verschiedene Variablenordnungen mit JPF-BDD untersuchen zu können, haben wir JPF-BDD um die Funktion erweitert, dass der Benutzer eine Variablenordnung angeben kann. Der Benutzer gibt dazu den Pfad einer Datei an, in der die Variablenordnung steht. JPF-BDD liest dann, vor Beginn der Verifikation, die Reihenfolge der Variablen aus dieser Datei aus. Um die Funktion zu nutzen, müssen in der JPF-Datei 2 Optionen angegeben werden.

jpf-bdd.enableVarOrder wird auf true gesetzt werden, damit JPF-BDD die Variablenordnung aus der Datei lädt.

jpf-bdd.varOrderPath gibt den Pfad der Datei mit den Variablenordnungen an. Das Format der Datei ist wie folgt festgelegt. In jeder Zeile steht der vollständige Name einer Variable. Dabei steht in der ersten Zeile die Variable die in der Ordnung an erster Stelle stehen soll usw.. Es dürfen keine Leerzeilen zwischen 2 Variablen sein.

Da wir den vollständigen Namen der Variable (Packagename. Klassenname. Variablenname) angeben müssen, können wir mit dieser Methode nur die

Ordnung von statischen boolschen Variablen angeben. Lokale Variablen können mehrfach instanziert werden. Darum sind nicht alle möglichen Namen für diese Variablen bekannt und können auch nicht vorher in einer Datei angegeben werden. Diese Einschränkung hat Auswirkungen auf die Anwendbarkeit in der Praxis. In unseren Experimenten verwenden wir allerdings keine lokalen Variablen.. Außerdem können wir aus dieser Methode den Vorteil ziehen, dass wir während der Verifikation das BDD nicht umordnen müssen. Dies könnte die Ergebnisse der Experimente verfälschen.

3 Kommunikationsgraph

Für unsere heuristische Analyse werden wir die Beziehung, der mit JPF-BDD verwalteten Variablen darstellen. Dazu verwenden wir Kommunikationgraphen. Der Kommunikationsgraph wird von Adnan Azit [1] für einen endlichen Zustandsautomaten definiert und verwendet. Da wir aber den Quellcode des Programms untersuchen wollen, definieren wir den Kommunikationsgraphen wie folgt.

Definition:Für jede Variable im BDD existiert ein Knoten im Kommunikationsgraphen, der diese Variable repräsentiert. Zwischen zwei Knoten A und B wird eine ungerichtete Kante hinzugefügt, wenn die von A und B repräsentierten Variablen im Programmcode in der selben logischen Formel stehen. Diese logischen Formeln können die Bedingungen von if- bzw. while-Schleifen sein, sowie Zuweisungen.

In den folgenden Experimenten verwenden wir zwei Begriffe die wir hier ebenfalls definieren.

Unabhängiger Teilgraph: Ein unabhängiger Teilgraph ist eine Teilmenge von Knoten des gesamten Graphen mit folgenden Eigenschaften.
Jeder Knoten der Teilmenge hat eine Kante zu mindestens einem anderen Knoten der Teilmenge.
Kein Knoten hat eine Kante zu einem Knoten außerhalb der Menge.
Ein Beispiel für einen unabhängigen Teilgraphen zeigt Abbildung 1.

Innerer Teilgraph: Eine Innerer Teilgraph ist eine Teilmenge eines unabhängigen Teilgraphen. Innere Teilgraphen, sind die unabhängigen Teilgraphen die entstehen, wenn wir einen Knoten und alle seine Kanten aus dem Graphen entfernen. Wir sprechen also von den inneren Teilgraphen eines bestimmten Knoten. Ein Beispiel zeigt Abbildung 2 für den Knoten 1. Abbildung 3 zeigt den Graphen, nachdem Knoten 1 entfernt wurde.

Für die Berechnung des Kommunikationsgraph haben wir ein eigenes Analyseprogramm implementiert. Bevor das Analyseprogramm verwendet werden kann, muss das Java Optimierungsframework Soot auf die zu verifizierenden Programme angewendet werden. Wir verwenden Soot um ein AST für jede Klasse des Programms in Form einer Xml-Datei zu erstellen. Dazu verwenden wir die Standardkonfiguration von Soot mit der Ausgabeoption Xml. Für mehr Informationen zu Soot verweisen wir auf [8].

Die erstellten Xml-Dateien dienen dem Analyseprogramm als Eingabe. Dazu

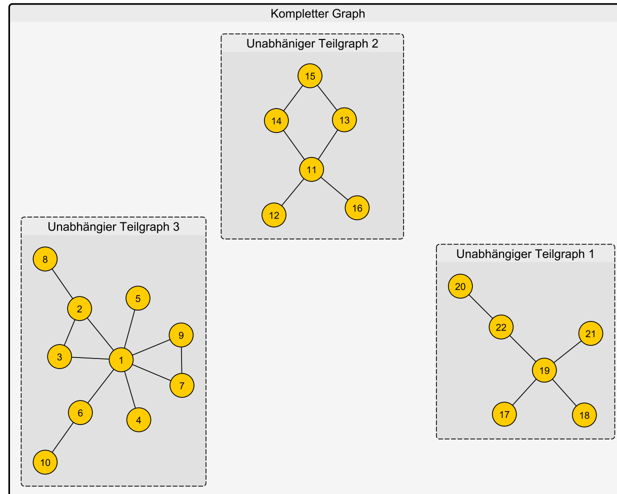

Abbildung 1: Beispiel für unabhängige Teilgraphen

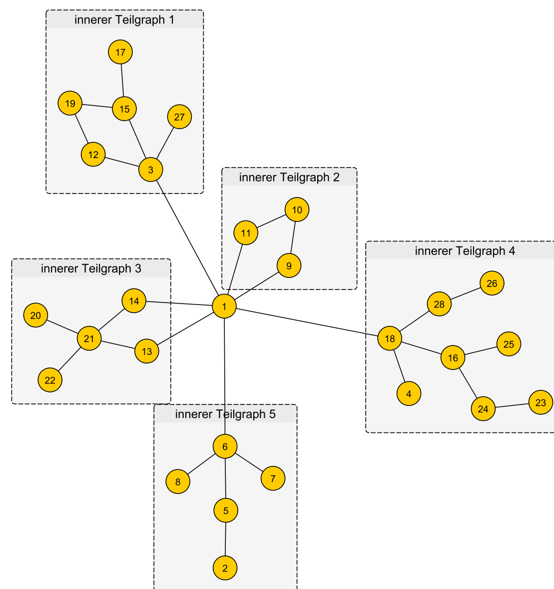

Abbildung 2: innere Teilgraphen von Knoten 1

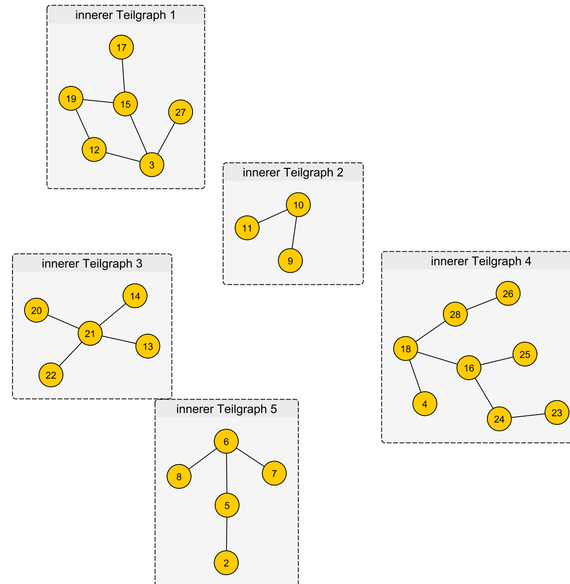

Abbildung 3: innere Teilgraphen nach entfernen von Knoten 1

kann man entweder die Pfade der Xml-Dateien einzeln angeben, oder den Pfad eines Ordners angeben, indem alle Xml-Dateien liegen. Dabei ist zu beachten, dass es in diesem Ordner keine Unterordnerstruktur geben darf. Aus den Xml-Dateien berechnet das Programm nun den Kommunikationsgraphen. Nach der Berechnung des Kommunikations, berechnet das Analyseprogramm aus diesem die optimierte Variablenordnung und gibt sie in einer Datei aus. Diese Datei kann dann direkt in JPF-Datei angegeben werden. Um die optimierte Ordnung zu berechnen verwendet das Analyseprogramm den Algorithmus den wir in dieser Arbeit entwickelt haben.

4 Experimente zur Optimierung der Variablenordnung

Mit unseren Experimenten wollen wir zum einen die Frage beantworten, ob die Variablenordnung einen bedeutenden Einfluss auf die Laufzeit haben kann. Außerdem wollen wir mit unseren Experimenten auch unseren Startalgorithmus optimieren, mit dem wir aus dem Kommunikationsgraph eine optimierte Variablenordnung berechnen.

Unsere ersten 2 Hypothesen erhalten wir aus den Arbeiten [2] und [1].

Hypothese 1. Knoten mit einem hohen Grad im Kommunkationsgraphen sollen in der Variablenordnung am Anfang stehen stehen.

Hypothese 2. Knoten die im Kommunikationsgraphen benachbart sind sol-

len auch in der Variablenordnung möglichst nahe beieinander stehen.

Aus diesen 2 Hypothesen erstellen wir unseren Startalgorithmus:

```
gegeben:  Kommunikationsgraph Graph K, Variablenordnung V
funktion BerechneVariablenordnung() start
    Liste<Knoten> k = SortiereKnotenNachGradAufsteigend(K);
    SortNachAbstandAufsteigend(k);
    for(int i = 0; i < k.size();i++){
        V.add(k.get(i));
    }
    Gibt V zurück;
ende

funktion SortNachAbstandAufsteigend (Liste<Knoten>k)
    /* Ordnet die Knoten, die den selben Grad haben,
       nach dem Abstand zum ersten Knoten der Liste aufsteigend
    *\
```

Für jedes Experiment erstellen wir einen Kommunikationsgraphen, mit dem wir versuchen bestimmte Hypothesen zu überprüfen, die wir zuvor aufgestellt haben. Aus dem Kommunikationsgraphen erstellen wir dann ein Programm, bei dem wir die Laufzeit von JPF-BDD mit verschiedene Variablenordnungen messen. Mit den Ergebnissen des Experiments verfeinern wir dann unseren Algorithmus.

Das Programm besitzt eine Main-Methode. Für jeden Knoten im Kommunikationsgraphen gibt es im Programm eine statische boolsche Variable. Diesen Variablen werden keine Werte zugewiesen. Für jede Kante des Kommunikationsgraph fügen wir eine if-Anweisung hinzu. In der Bedingung der if-Anweisung werden die 2 Variablen, die im Graphen durch die Kante verbunden sind mit einem logischen UND verknüpft.

In den ersten Experimenten haben wir zusätzlich überprüft ob es einen Unterschied im Verhalten gibt, wenn wir in den Bedingungen der if-Schleifen ein logisches ODER anstelle eines logischen UND verwenden.

Das Programm besitzt außerdem auch eine Zählvariable. Diese Zählvariable wird in jeder if-Schleife um 1 hochgezählt, wenn die entsprechende Bedingung der if-Schleife erfüllt ist. Am Ende der Main-Methode wird diese Zählvariable ausgegeben.

Wie bereits in Kapitel 2 beschrieben, betrachtet JPF-BDD beide Belegungen für eine boolsche Variable, wenn ihr kein Wert zugeteilt wird. Dadurch entdeckt JPF-BDD bei der Verifikation alle möglichen Werte der Zählvariable und gibt diese aus. Dadurch können wir mit diesem einfachen Programm relativ große Laufzeiten erreichen, die wir besser messen und miteinander vergleichen können. Trotzdem mussten wir für unsere Experimente Programme mit 40 oder mehr boolschen Variablen verwenden um ausreichend große Laufzeiten zu erreichen.

Das Programm zu 4 sieht also folgendermaßen aus:

```
static boolean a1;
static boolean a2;
static boolean a3;
static boolean a4;
static boolean a5;
public static void main(String[] args) {
    int i = 0;
    if(a1 & a2){
        i++;
    }
    if(a1 & a3){
        i++;
    }
    if(a1 & a4){
        i++;
    }
    if(a4 & a5){
        i++;
    }
    System.out.println(i);
}
```

Die Experimente sowie die Testreihe in Kapitel 5 werden auf folgendem System ausgeführt:

```
Prozessor: AMD Phenom 2 X6 1090T, 6x3.2 GHz
Arbeitsspeicher: 2x 4 GB DDR3- 1333 MHz Kingston VR
Mainboard: ASUS M4A87TD/USB3 - AMD 870 - AM3 - ATX
Betriebssystem: Windows 7 Professional 64 bit
Java: JRE JavaSE-1.6
Maximaler Speicher: 2048 MB
```

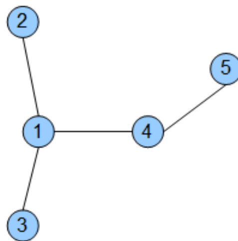

Abbildung 4:

4.1 Experiment A

Algorithmus

```
gegeben:  Kommunikationsgraph Graph K, Variablenordnung V
funktion BerechneVariablenordnung() start
    Liste<Knoten> k = SortiereKnotenNachGradAufsteigend(K);
    SortNachAbstandAufsteigend(k);
    for(int i = 0; i < k.size();i++){
        V.add(k.get(i));
    }
    Gibt V zurück;
ende

funktion SortNachAbstandAufsteigend (Liste<Knoten>k)
    /* Ordnet die Knoten, die den selben Grad haben,
       nach dem Abstand zum ersten Knoten der Liste aufsteigend
    *\
```

Hypothesen

Hypothese 1. Knoten mit einem hohen Grad im Kommunkationsgraphen sollen in der Variablenordnung am Anfang stehen stehen.

Hypothese 2. Knoten, die im Kommunikationsgraphen benachbart sind sollen auch in der Variablenordnung möglichst nahe beieinander stehen.

In diesem Experiment wollen wir unsere zwei anfänglichen Hypothesen überprüfen. Dazu betrachten wir diesen Kommunikationsgraphen 5.
Der Graph hat insgesamt 40 Knoten. Davon einen zentralen Knoten a1, der zu jedem anderen Knoten eine Verbindung hat. Die anderen Knoten dagegen haben nur eine Verbindung zu a1 und jeweils 2 weiteren Knoten. Der Grad des zentralen Knoten ist 39, während der Grad aller übrigen Knoten 3 beträgt.

Ordnung A Dies ist die Ordnung, wie sie von JPF-BDD aus dem Programm eingelesen wird, wenn wir keine eigene Ordnung angeben. Sie soll uns zum Vergleich mit anderen Ordnungen dienen.

Ordnung B Hier haben wir die Variablen a2 und a10 vertauscht. Die Knoten a3 und a2 sind im Kommunkationsgraphen eigentlich benachtbart. Dasselbe gilt für die Knoten a9, a10 und a11. Durch das Vertauschen der zwei Knoten erhöht sich der Abstand zu ihren Nachbarknoten in der Variablenordnung. Darum erwarten wir eine Laufzeiterhöhung bei dieser Ordnung im Vergleich zu Ordnung A.

Ordnung C Wie in Ordnung B haben wir in dieser Ordnung 2 Variablen vertauscht. Diesesmal die Variablen a2 und a20. Dadurch erhöht sich der Abstand zwischen benachbarten Knoten stärker, als es in Ordnung B der Fall ist. Deshalb erwarten wir eine größere Laufzeiterhöhung als bei Ordnung B.

Ordnung D Bei Ordnung D haben wir wie in Ordnung C, die Variablen a2 und a20 vertauscht. Zusätzlich haben wir auch die Variablen a3 und a21 miteinander vertauscht. Zwar stehen dadurch die Paare a2 und a3 sowie a20

und a21 wieder nebeneinander in der Variablenordnung, trotzdem erwarten wir eine weitere Laufzeiterhöhung im Vergleich zu Ordnung C. Der Grund dafür ist dass beim vertauschen von 4 Variablen mehr Variablen von einer Erhöhung des Abstands betroffen sind.

Ordnung E Mit Ordnung E wollen wir Hypothese 1 bestätigen. Darum setzen wir die Variable a1, die den größten Grad im Kommunikationsgraphen besitzt, in die Mitte der Variablenordnung zwischen a19 und a20. Wir vermuten, dass sich dadurch die Laufzeit erhöht.

Ordnung F Auch mit Ordnung F wollen wir Hypothese 1 bestätigen. Darum setzen wir die Variable a1 nun an die letzte Stelle der Variablenordnung. Wir vermuten, dass sich dadurch die Laufzeit stärker erhöht als bei Ordnung E.

Ordnung G Bei Ordnung G haben wir die Reihenfolge aller Variablen bis auf a1 umgedreht. Da sich hierbei der Abstand zwischen den einzelnen Variablen nicht ändert, erwarten wir auch keinen großen Unterschied zu Ordnung A in der Laufzeit.

Ordnung H In dieser Ordnung setzten wir a1 aus Ordnung G an die letzte Stelle der Variablenordnung. Wie auch bei Ordnung F wollen wir damit Hypothese 1 bestätigen und erwarten einen Anstieg der Laufzeit im Vergleich zu Ordnung G.

Ordnung I In Ordnung I haben wir die Variablen des Kreis um a1 nach dem Abstand geordnet. Begonnen haben wir mit a1 und dann a2. Danach kommen die Variablen die den Abstand 1 zu a2 besitzen, dann die Variablen mit Abstand 2 usw. Damit versuchen wir den durchschnittlichen Abstand zwischen den Variablen zu senken und so eine Verbesserung der Laufzeit zu erhalten.

Ergebnisse Laufzeiten des Programms mit den verschiedenen Ordnungen können wir in dieser Tabelle 6 betrachten.

Wie erwartet können wir für Ordnung B einen kleinen Anstieg der Laufzeit verzeichnen. Ebenfalls wie erwartet erhöht sich die Laufzeit bei Ordnung C sogar noch stärker, auf über das zweifache der ursprünglichen Zeit von Ordnung A. Bei Ordnung D verdreifacht sich die Laufzeit im Vergleich zu Ordnung C fast nocheinmal. Mit den Ergebnissen dieser 3 Ordnungen können wir unsere Hypothese 2 bekräftigen. Auch das Ergebniss von Ordnung I bekräftigt Hypothese 2. Bei dieser Ordnung messen wir eine Halbierung der Laufzeit.

Weder bei Ordnung E noch bei Ordnung F können wir ,entgegen unseren Erwartungen, eine Veränderung der Laufzeit messen. Wir denken es liegt daran, dass a1 der einzige Knoten mit einem höheren Knoten ist, während alle anderen Knoten den selben Grad haben. Dadurch könnte es eine weniger große Rolle spielen an welcher Stelle sich a1 in der Ordnung befindet. Außerdem ändert sich der durchschnittliche Abstand von a1 zu den anderen Variablen innerhalb der Ordnung nicht, wenn man a1 verschiebt. Dies liegt an der besonderen Form des Kommunikationsgraphen, in dem a1 zu allen Knoten einen Abstand von 1 besitzt.

Bei Ordnung G und H verringert sich die Laufzeit unerwartet auf ungefähr ein Viertel der ursprünglichen Zeit. Zwischen Ordnung G und H ändert sich, wie bereits bei Ordnung E und F beobachtet, die Laufzeit nicht, nachdem a1 an die letzte Stelle verschoben wurde. Da sich auch der Abstand zwischen den

Variablen nicht geändert hat, schließen wir daraus, dass die Laufzeitänderung mit der Reihenfolge der if-Schleifen aus dem Programm zusammenhängt. Da wir diese Reihenfolge im Allgemeinen nicht vorhersehen können, werden wir diesen Umstand nicht weiter untersuchen.

Fassen wir die Ergebnisse von Experiment A zusammen. Wir konnten Hypothese 2 bestätigen. Hypothese 1 dagegen bleibt noch unbestätigt und wird in weiteren Experimenten untersucht.

Da wir in diesem Experiment keine neuen Hypothesen aufgestellt und bestätigt haben verändert sich der Algorithmus nicht.

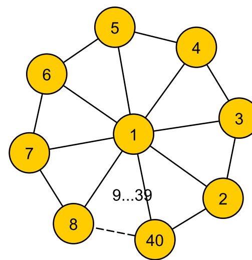

Abbildung 5: Kommunikationsgraph zu ExperimentA

Bezeichnung	Variablenordnung	Laufzeit bei UND-Verknüpfung	Laufzeit bei ODER-Verknüpfung
A	1>2>3>...>40	21s	19s
B	1>10>3>...>9>2>11>...>40	30s	27s
C	1>20>3>...>19>2>21>...>40	1:03m	50s
D	1>20>21>4>...>19>2>3>22>...>40	3:01m	2:37m
E	2>3>...>19>1>20>21>...>40	21s	19s
F	2>3>4>....>40>1	21s	20s
G	1>40>39>...>2	5s	4s
H	40>39>...>2>1	5s	5s
I	1>2>40>3>39>4>38>...	9s	8s

Abbildung 6: Laufzeiten zu ExperimentA

4.2 ExperimentB

Algorithmus

```
gegeben:  Kommunikationsgraph Graph K, Variablenordnung V
funktion BerechneVariablenordnung() start
    Liste<Knoten> k = SortiereKnotenNachGradAufsteigend(K);
    SortNachAbstandAufsteigend(k);
    for(int i = 0; i < k.size();i++){
        V.add(k.get(i));
    }
    Gibt V zurück;
ende

funktion SortNachAbstandAufsteigend (Liste<Knoten>k)
    /* Ordnet die Knoten, die den selben Grad haben,
       nach dem Abstand zum ersten Knoten der Liste aufsteigend
    *\
```

Hypothesen

Hypothese 1. Knoten mit einem hohen Grad im Kommunkationsgraphen sollen in der Variablenordnung am Anfang stehen.

In diesem Experiment wollen wir nochmal unsere Hypothese überprüfen, die wir in Experiment A 4.1 weder bestätigen noch wiederlegen konnten. Wir betrachten hierzu diesen Graph 7.
Der Graph besteht diesmal aus zwei Kreisen, die jeweils die zentralen Knoten a1 bzw. a21 haben. Der Graph besteht aus insgeamt 40 Knoten. Die zwei Kreise haben mit ihren zentralen Knoten jeweils 20 Knoten. Jeder Knoten eines Kreises besitzt eine Kante zu seinem zentralen Knoten und zwei weiteren Knoten. Außerdem besteht eine Kante zwischen den beiden zentralen Knoten a1 und a21. a1 und a21 haben daher den Grad 20, während alle anderen Knoten den Grad 3 haben.

Ordnung A Dies ist die Ordnung, wie sie von JPF-BDD aus dem Programm eingelesen wird, wenn wir keine eigene Ordnung angeben. Sie soll uns zum Vergleich mit anderen Ordnungen dienen.

Ordnung B In Ordnung B haben wir Variable a21 an die zweite Stelle gesetzt, damit die 2 Variablen mit dem höchstem Grad an erster und zweiter Stelle stehen. Dadurch wollen wir Hypothese 1 überprüfen. Wir erwarten eine Verbesserung der Laufzeit im Vergleich zu Ordnung A.

Ordnung C In Ordnung C setzen wir die Variablen mit dem höchsten Grad an die letzten zwei Stellen der Ordnung. Damit erstellen wir eine Gegenprobe zu Hypothese 1 und erwarten eine Steigerung der Laufzeit.

Ordnung D In Ordnung D haben wir Variable a21 an die letzte Stelle gesetzt. Damit erhöht sich der Abstand zwischen den zwei Variablen mit dem höchstem Grad im Kommunikationsgraph auf ein Maximum. Wir erwarten deshalb eine starkes Ansteigen der Laufzeit im Vergleich zu Ordnung B.

Ordnung E In Ordnung E haben wir die Variablen des Kreis um a1 bzw. um a21 nach dem Abstand geordnet. Begonnen haben wir mit a1 und dann a2. Danach kommen die Variablen die den Abstand 1 zu a2 besitzen, dann die Variablen mit Abstand 2 usw. Dasselbe wird dann auch für den Kreis um a21 gemacht. Dadurch verringern wir den durchschnittlichen Abstand, weshalb wir eine Verbesserung der Laufzeit erwarten. Vergleiche hierzu Experiment A Ordnung I4.1.

Ordnung F In Ordnung F haben wir von Ordnung E die Variable a21 an zweite Stelle gesetzt damit die Variablen mit höchstem Grad an vorderster Stelle der Variablenordnung stehen. Wir erwarten dadruch eine Verbesserung der Laufzeit im Vergleich zur Laufzeit bei Ordnung E.

Ordnung G In Ordnung G haben wir die Variablen Der 2 Kreise abwechselnd angeordnet. Beginnend mit den 2 Variablen mit dem höchsten Grad a1 und a21, kommen dann abwechselnd je eine Variable aus dem Kreis um a1 und dem Kreis um a21. Damit verringern wir den durchschnittlichen Abstand zwischen den Variablen in der Ordnung. Wir erwarten deshalb eine Laufzeitverbesserung im Vergleich zu Ordnung B.

Ordnung H In Ordnung H haben wir statt a1, a21 an die erste Stelle der Variablenordnung gesetzt. Variable a1 steht dadurch an zweiter Stelle der Ordnung. Wir erwarten hier keinen großen Unterschied der Laufzeit im Vergleich zu Ordnung B.

Ordnung I In Ordnung I wurden die Variablen a21 bis a40 zwischen a1 und a2 gesetzt. Dadurch stehen die Variablen mit höchstem Grad an den ersten zwei Stellen. Allerdings stehen in dieser Ordnung die Variablen, die zum Kreis um Variable a21 gehören vor den Variablen, die zum Kreis um Varaible a1 gehören. Wir erwarten keinen großen Unterschied bei der Laufzeit im Vergleich zu Ordnung H.

Ergebnisse Laufzeiten des Programms mit den verschiedenen Ordnungen können wir in dieser Tabelle 8 betrachten.

Wie erwartet verbessert sich die Laufzeit mit Variablenordnung B im Vergleich zu Ordnung A. Allerdings fällt die Verbesserung der Laufzeit um ca 27 Sekunden im Vergleich zur ursprünglichen Laufzeit von ca 6 Minuten und 28 Sekunden relativ gering aus. Darum lässt sich Hypothese 1 nicht allein mit Ordnung B bestätigen. Bei Ordnung C allerdings verdoppelt sich die Laufzeit im Vergleich zu Ordnung B fast. Daraus können wir schließen, dass es besser ist Variablen mit einem hohen Grad im Kommunkationsgraph an die vorderen Stellen der Variablenordnung zu setzten. Somit bestätigt sich Hypothese 1.

Für Ordnung D haben wir ein starkes Ansteigen der Laufzeit im Vergleich zu Ordnung B erwartet. Allerdings messen wir hier sogar eine leichte Verbesserung der Laufzeit. Dies ist aber nur bei UND-Verknüpfungen der Fall. Bei dem Experiment mit ODER-Verknüpfungen verschlechtert sich die Laufzeit, was unseren Erwartungen entspricht. Warum sich das Verhalten hier bei UND- und ODER-Verknüpfungen unterscheidet können wir uns nicht erklären. Aber weil dies ein Einzelfall in unseren Experimenten ist, betrachten wir dieses Ergebniss nicht weiter.

Bei Ordnung E und Ordnung F haben wir entgegen unseren Erwartungen einen starken Anstieg der Laufzeit im Vergleich zu Ordnung A. Dies steht im Wiederspruch mit den Ergebnissen aus Experiment A mit Ordnung I 4.1. Daraus schließen wir, dass die Ordnung, wie in Ordnung E und F, für einen kreisförmigen Graphen bzw. Teilgraphen nicht immer gut ist. Ordnung F hat allerdings eine Laufzeitverbesserung im Vergleich zu Ordnung E. Dieses Ergebniss unterstützt noch einmal Hypothese 1.

Durch Ordnung G ändert sich die Laufzeit im Vergleich zu Ordnung A fast gar nicht. Aus diesem Ergebniss können wir keine Schlussfolgerungen ziehen.

Bei Ordnung H messen wir eine leicht stärkere Verbesserung der Laufzeit zu Ordnung A, als wir zwischen Ordnung A und Ordnung B gemessen haben. Wir vermuten das es daran liegt, dass wir in Ordnung H nach Variable a21 zuerst die Variablen des größten inneren Teilgraphen von a21 gesetzt haben. In Ordnung I haben wir dasselbe gemacht, mit a1 als erste Variable der Ordnung. Hier messen wir eine sehr starke Verbesserung der Laufzeit. Den Grund, warum bei dieser Ordnung die Laufzeitverbesserung um sovieles größer ist als bei Ordnung H, vermuten wir bei der Reihenfolge der if-Schleifen im Programm.

Abgesehen von Ordnung D hatte auch bei diesem Experiment, die Variante mit den ORDER-Verknüpfungen, die selben Tendenzen bei den Laufzeiten.

In diesem Experiment haben wir Hypothese 1 bestätigt. Außerdem verändern wir den Algorithmus aufgrund der Ergebnisse von Ordnung H und I wie folgt. Wir berechnen zuerst einen Hauptknoten des Kommunikationsgraphen und berechnen von diesem aus die inneren Teilgraphen. Auf diese Teilgraphen wird dieses Verfahren dann rekursiv angewendet, bis die inneren Teilgraphen eine Größe von 1 haben.

Außerdem stellen wir eine Hypothese für diesen Algorithmus auf.

Hypothese 1: Es ist besser die inneren Teilgraphen der Größe nach absteigend zu sortieren.

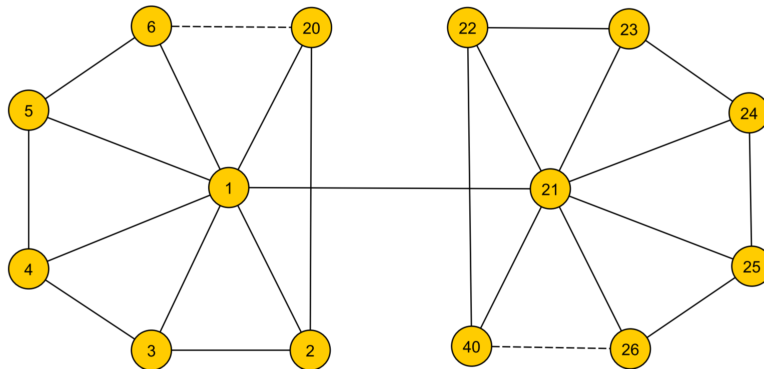

Abbildung 7: Kommunikationsgraph zu ExperimentB

Bezeichnung	Variablenordnung	Laufzeit bei UND-Verknüpfung	Laufzeit bei ODER-Verknüpfung
A	1>2>3>...>40	6:28m	6:24m
B	1>21>2>3>...>20>22>...>40	6:01m	6:07m
C	2>3>...>20>22>...>40>1>21	11:52m	12:24m
D	1>2>3>...>19>20>22>...>40>21	5:45m	6:53m
E	1>2>20>3>19>...>21>22>40>...	10:30m	12:10m
F	1>21>2>20>3>19>...>22>40>...	10:01m	10:52m
G	1>21>2>22>3>23>4>...>40	6:27m	6:49m
H	21>1>2>3>...>40	5:53m	5:59m
I	1>21>22>...>40>2>3>4>...>20	1:30m	1:39m

Abbildung 8: Laufzeiten zu ExperimentB

4.3 ExperimentC

Algorithmus

```
gegeben:  Kommunikationsgraph Graph K, Variablenordnung V
funktion BerechneVariablenordnung() start
    Knoten k = K.BerechneHauptknoten();
    V.add(k);
    Liste<Graphen> innTeilgraphen =
        K.BerechneInnereTeilgraphen(k);
    sortiereInnereTeilgraphenAbsteigend(innTeilgraphen);
    for(int j = 0; j < innTeilgraphen.size(); j++){
        Graph innG = innTeilgraphen.get(j);
        BerechneOrdnungInnererTeilgraphen(innG);
    }
    Gibt V zurück;
ende
funktion BerechneOrdnungInnererTeilgraphen (Graph g) start
    Knoten k = innG.BerechneHauptknotenInnererTeilgraphen();
    V.add(k);
    if(g.size() > 1){
        Liste<Graphen> innTeilgraphen =
```

```
            innG.BerechneInnereTeilgraphen(k);
        sortiereInnereTeilgraphenAbsteigend(innTeilgraphen);
        for(int i = 0; i < innteilgraphen.size(); i++){
            Graph innG = innTeilgraphen.get(i);
            BerechneOrdnungInnererTeilgraphen(innG);
        }
    }
ende

funktion BerechneHauptknoten()
/* Berechnet den Hauptknoten eines Kommunikationsgraphen nach
   folgenden Kriterien:
    - Wähle den Knoten mit dem höchsten Grad
    - Wenn mehrere Knoten den selben höchsten Grad besitzen
      wähle einen von ihnen aus
*\

funktion BerechneHauptknotenInnerenTeilgraphen()
/* Berechnet den Hauptknoten eines inneren Teilgraphen nach
   folgenden Kriterien:
    - Wähle den Knoten mit dem höchsten Grad
    - Wenn mehrere Knoten den selben höchsten Grad besitzen
      dann wähle aus diesen Knoten den Knoten mit dem geringsten
      Abstand zum Hauptknoten des unabhängigen Teilgraphen zu dem
      der innere Teilgraph gehört.
    - Wenn mehrer Knoten den selben Abstand besitzen wähle einen
      von ihnen aus.
    *\
```

Hypothesen

Hypothese1: Es ist besser bei unserem Algorithmus die inneren Teilgraphen der Größe nach absteigend zu sortieren.

Mit Experiment C wollen wir unseren bisherigen Algorithmus überprüfen. Außerdem wollen wir auch Hypothese 1 überprüfen. Dazu betrachten wir diesen Kommunikationsgraphen 9. Der Graph besteht aus 40 Knoten. Variable a1 ist hier der Hauptknoten für unseren Algorithmus. Der Knoten a1 hat 7 innere Teilgraphen mit unterschiedlicher Größe.

Ordnung A Dies ist die Ordnung, wie sie von JPF-BDD aus dem Programm eingelesen wird, wenn wir keine eigene Ordnung angeben. Sie soll uns zum Vergleich mit anderen Ordnungen dienen.

Ordnung B In Ordnung B haben wir den Hauptknoten a1 an die letzte Stelle der Variablenordnung gesetzt. Wir erwarten hier einen Anstieg der Laufzeit im Vergleich zu Ordnung A.

Ordnung C Bei dieser Ordnung haben wir nur nach dem Grad eines Knoten geordnet. Beginnend bei der Variable mit dem höchstem Grad, kommen dann die Variablen mit dem nächst höchsten Grad usw. Damit wollen wir untersuchen

ob es besser ist nur nach dem Grad einer Variable zu ordnen oder nach unserem Algorithmus.

Ordnung D Ähnlich wie bei Ordnung C haben wir auch bei Ordnung D nur nach dem Grad des Knoten geordnet. Diesesmal aber aufsteigend statt absteigend. Hier erwarten wir eine Anstieg der Laufzeit im Vergleich zu Ordnung C.

Ordnung E Bei Ordnung E haben wir nur nach dem Abstand zum Hauptknoten a1 geordnet. Das heißt zuerst kommt Variable a1 und dann alle Variablen die den Abstand 1 zu a1 im Kommunkationsgraphen haben, dann die Variablen mit Abstand 2 usw. Mit dieser Ordnung wollen wir untersuchen ob dieses Ordnungsschema besser oder schlechter ist als unser bisheriger Algorithmus.

Ordnung F Wie in Ordnung E, haben wir auch hier nur nach dem Abstand zu Variable a1 geordnet. Allerdings beginnend mit den Variablen, die den größten Abstand zu a1 haben. Wir erwarten hier einen Anstieg der Laufzeit im Vergleich zu Ordnung E.

Ordnung G Hier haben wir nach unserem bisherigen Algorithmus geordnet. dabei haben wir die inneren Teilgraphen der Größe nach aufsteigend sortiert. Wir erwarten eine schlechtere Laufzeit als bei Ordnung H.

Ordnung H Wie in Ordnung G haben wir hier nach unserem bisherigen Algoritmus geordnet. Allerdings haben wir diesmal die inneren Teilgraphen der Größe absteigend sortiert. Wir erwarten im Vergleich zu Ordnung G eine Verbesserung der Laufzeit.

ErgebnisseLaufzeiten des Programms mit den verschiedenen Ordnungen können wir in dieser Tabelle 10 betrachten.

Bei Ordnung B können wir einen sehr kleinen Anstieg der Laufzeit messen. Der Unterschied ist allerdings zu gering um eine Aussage treffen zu können.

Bei Ordnung C steigt die Laufzeit im Vergleich zu Ordnung A sehr stark an. Bei Ordnung D übersteigt die Laufzeit sogar 15 Minuten. Aus diesen zwei Ordnungen können wir schließen, dass eine reine Ordnung nach dem Grad nicht optimal ist. Weiter zeigt uns der Vergleich der beiden Ordnungen, dass das Ordnen nach aufsteigendem Grad um einiges schlechter ist, als das Ordnen nach absteigendem Grad. Auch zeigen uns diese beiden Ordnungen, dass die Variablenordnung einen bedeutenden Einfluss auf die Laufzeit haben kann.

Bei Ordnung E und F steigt die Laufzeit ebenfalls sehr stark an. Daraus schließen wir, dass auch eine Ordnung nur nach dem Abstand zum Hauptknoten nicht optimal ist. Wenn wir Ordnung E und F miteinander vergleichen, können wir daraus schließen, dass das Ordnen nach aufsteigendem Abstand zum Hauptknoten besser ist als das Ordnen nach absteigendem Abstand.

Wenn wir Ordnung G und H miteinander vergleichen, können wir eine geringfügig bessere Laufzeit für Ordnung H feststellen. Damit können wir Hypothese 1 nicht bestätigen. Allerdings zeigen beide Ordnungen ,die aus unserem bisherigen Algorithmus entstanden sind, eine sehr gute Laufzeit im Vergleich zu den Ordnungen C,D,E und F.

In diesem Experiment konnten wir Hypothese 1 weder bestätigen noch wiederlegen. Darum wird diese Hypothese in einem weiteren Experiment untersucht.

Außerdem zeigt uns der Vergleich zwischen der besten und der schlechtesten Laufzeit, die wir in diesem Experiment gemessen haben, dass die Variablenordnung einen bedeutenden Einfluss auf die Laufzeit haben kann.

Wie in den vorherigen Experimenten, können wir auch in Experiment C keinen Unterschied im Verhalten zwischen UND- und ODER-Verknüpfungen feststellen. Darum werden wir diese Unterscheidung in den weiteren Experimenten nicht weiter untersuchen.

Wir haben außerdem festgestellt das unsere bisheriger Algorithmus in diesem Experiment gute Ergebnisse geliefert hat. Darum wird der Algorithmus weiter verfeinert.

Falls es beim Bestimmen des Hauptknoten von inneren Teilgraphen mehrere Knoten mit dem selben höchsten Grad gibt, ordnen wir sie nach ihrem Abstand zum Hauptknoten des ganzen Graphen. Außerdem ordnen wir die inneren Teilgraphen der Größe nach absteigend, bevor wir sie weiter bearbeiten.

Wir haben außerdem bemerkt, dass der Kommunikationsgraph aus Experiment C 9 ein kürzester Wegegraph ist, ausgehend vom Hauptknoten a1. Daher fragen wir uns, ob es einen Unterschied macht, den Algorithmus auf dem kürzesten Wegegraphen des Hauptknoten oder auf dem Kompletten Graphen anzuwenden.

Daraus ergeben sich folgende Variationen des Algorithmus die wir weiter untersuchen.

Variation 1: Führe den Algorithmus auf dem Kompletten Graphen aus.

Variation 2.1: Berechne zuerst den kürzesten Wegegraphen des ausgewählten Hauptknoten. Führe dann den Algorithmus aus, verwende aber bei der Auswahl der Hauptknoten der inneren Teilgraphen die Grade, wie sie im kompletten Kommunikationsgraph vorkommen.

Variation 2.2: Berechne zuerst den kürzesten Wegegraphen des ausgewählten Hauptknoten. Führe dann den Algorithmus aus, verwende aber bei der Auswahl der Hauptknoten der inneren Teilgraphen die Grade, wie sie im kürzesten Wegegraphen vorkommen.

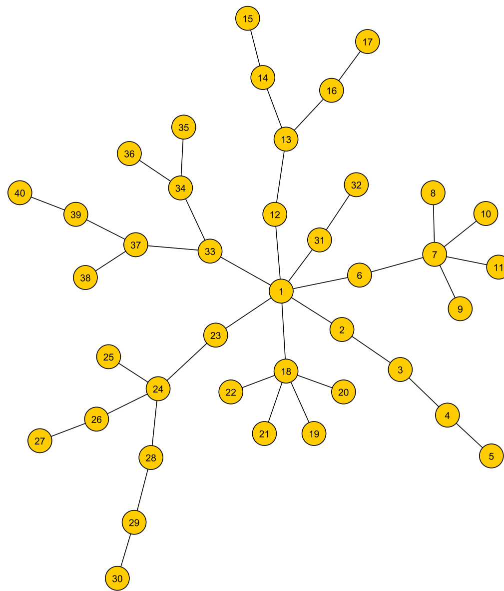

Abbildung 9: Kommunikationsgraph zu ExperimentC

Bezeichnung	Variablenordnung	Laufzeit bei UND-Verknüpfung	Laufzeit bei ODER-Verknüpfung
A	1>2>3>...>40	2s	3s
B	40>1>2>3>...>39	2s	2s
C	1>7>18>24>13>33>...>40	5:25m	7:39m
D	5>8>9>10>11>...1	>15m	>15m
E	1>2>6>12>18>23>31>33>...	4:07m	5:31m
F	1>30>40>15>17>...	>15m	>15m
G	1>31>32>2>3>...	3s	3s
H	1>23>24>25>26>27>...	1s	1s

Abbildung 10: Laufzeiten zu ExperimentC

4.4 ExperimentD

Algorithmus

```
gegeben:  Kommunikationsgraph Graph K, Variablenordnung V
funktion BerechneVariablenordnung() start
    Knoten k = K.BerechneHauptknoten();
    V.add(k);
    Variante 2.1 und 2.2:[
        Graph kK = K.BerechneKürzestenWegeGraph(k);
        Liste<Graphen> innTeilgraphen =
            kK.BerechneInnereTeilgraphen(k);
    ]
    Variante 1:[
        Liste<Graphen> innTeilgraphen =
            K.BerechneInnereTeilgraphen(k);
    ]
    sortiereInnereTeilgraphenAbsteigend(innTeilgraphen);
    for(int j = 0; j < innTeilgraphen.size(); j++){
        Graph innG = innTeilgraphen.get(j);
        BerechneOrdnungInnererTeilgraphen(innG);
    }
    Gibt V zurück;
ende
funktion BerechneOrdnungInnererTeilgraphen (Graph g) start
    Knoten k = innG.BerechneHauptknotenInnererTeilgraphen();
    V.add(k);
    if(g.size() > 1){
        Liste<Graphen> innTeilgraphen =
            innG.BerechneInnereTeilgraphen(k);
        sortiereInnereTeilgraphenAbsteigend(innTeilgraphen);
        for(int i = 0; i < innteilgraphen.size(); i++){
            Graph innG = innTeilgraphen.get(i);
            BerechneOrdnungInnererTeilgraphen(innG);
        }
    }
ende

funktion BerechneHauptknoten()
/* Berechnet den Hauptknoten eines Kommunikationsgraphen nach
   folgenden Kriterien:
    - Wähle den Knoten mit dem höchsten Grad
    - Wenn mehrere Knoten den selben höchsten Grad besitzen
      wähle einen von ihnen aus
*\
```

```
funktion BerechneHauptknotenInnerenTeilgraphen()
/* Berechnet den Hauptknoten eines inneren Teilgraphen nach
   folgenden Kriterien:
     - Wähle den Knoten mit dem höchsten Grad
     - Wenn mehrere Knoten den selben höchsten Grad besitzen
       dann wähle aus diesen Knoten den Knoten mit dem geringsten
       Abstand zum Hauptknoten des unabhängigen Teilgraphen zu
       dem der innere Teilgraph gehört
     - Wenn mehrer Knoten den selben Abstand besitzen wähle
       einen von ihnen aus

       Variante 2.1: Hier werden die Grade des ursprünglichen
       Graphen bei der Wahl des Hauptknoten verwendet.
       Variante 2.2: Hier werden die Grade des kürzesten
       Wegegraphen bei der Wahl des Hauptknoten verwendet. *\
```

In Experiment D wollen wir untersuchen, welche der 3 Variationen unseres Algorithmus die beste ist. Dazu haben wir den Graph aus Experiment C 9 um einige Kanten erweitert, damit er nicht mehr ein kürzester Wegegraph von a1 ist 11.

Die unbestätigte Hypothese aus Experiment C werden wir in diesem Experiment nicht untersuchen. Der Grund dafür ist das sich die Kommunikationsgraphen von Experiment C und D zu ähnlich sind. Wir werden die Hypothese in Experiment E weiter untersuchen.

Ordnung A Dies ist die Ordnung, wie sie von JPF-BDD aus dem Programm eingelesen wird wenn, wir keine eigene Ordnung angeben. Sie soll uns zum Vergleich mit anderen Ordnungen dienen.

Ordnung B Bei dieser Ordnung haben wir nach dem Algorithmus mit Variation 1 geordnet.

Ordnung C Bei dieser Ordnung haben wir nach dem Algorithmus mit Variation 2.1 geordnet.

Ordnung D Bei dieser Ordnung haben wir nach dem Algorithmus mit Variation 2.2 geordnet.

Ergebnisse Laufzeiten des Programms mit den verschiedenen Ordnungen können wir in dieser Tabelle 12 betrachten.

Beim Vergleich der Laufzeit von Ordnung B, mit den Laufzeiten der Ordnungen C und D stellen wir eine leichte Verbesserung der Laufzeit auf Seiten von Ordnung C und D fest. Allerdings ist der Unterschied zu gering um daraus eine fundierte Schlussfolgerung zu ziehen.

Beim Vergleich zwischen Ordnung C und D können wir keinen Unterschied feststellen. Das liegt daran, dass sich die beiden Ordnungen kaum unterscheiden.

Insgesamt können wir aus Experiment D keine Schlussfolgerungen ziehen. Dadurch kann auch der Algorithmus nach diesem Experiment nicht verfeinert werden.

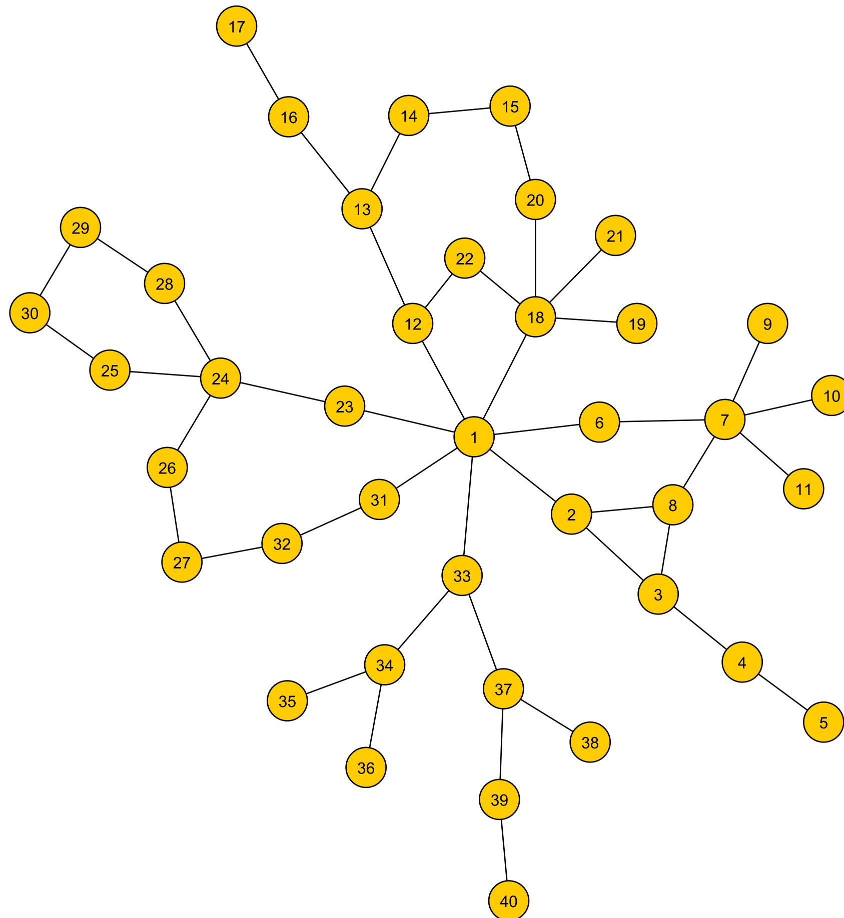

Abbildung 11: Kommunikationsgraph zu ExperimentD

Bezeichnung	Variablenordnung	Laufzeit
A	1>2>3>...>40	5s
B	1>18>12>13>14>15>20>...	8s
C	1>24>...>11>12>13>14>15>16>17>...	5s
D	1>24>...>11>13>14>15>16>17>12>...	5s

Abbildung 12: Laufzeiten zu ExperimentD

4.5 ExperimentE

Algorithmus

```
gegeben:  Kommunikationsgraph Graph K, Variablenordnung V
funktion BerechneVariablenordnung() start
    Knoten k = K.BerechneHauptknoten();
    V.add(k);
    Variante 2.1 und 2.2:[
        Graph kK = K.BerechneKürzestenWegeGraph(k);
        Liste<Graphen> innTeilgraphen =
            kK.BerechneInnereTeilgraphen(k);
    ]
    Variante 1:[
        Liste<Graphen> innTeilgraphen =
            K.BerechneInnereTeilgraphen(k);
    ]
    sortiereInnereTeilgraphenAbsteigend(innTeilgraphen);
    for(int j = 0; j < innTeilgraphen.size(); j++){
        Graph innG = innTeilgraphen.get(j);
        BerechneOrdnungInnererTeilgraphen(innG);
    }
    Gibt V zurück;
ende
funktion BerechneOrdnungInnererTeilgraphen (Graph g) start
    Knoten k = innG.BerechneHauptknotenInnererTeilgraphen();
    V.add(k);
    if(g.size() > 1){
        Liste<Graphen> innTeilgraphen =
            innG.BerechneInnereTeilgraphen(k);
        sortiereInnereTeilgraphenAbsteigend(innTeilgraphen);
        for(int i = 0; i < innteilgraphen.size(); i++){
            Graph innG = innTeilgraphen.get(i);
            BerechneOrdnungInnererTeilgraphen(innG);
        }
    }
ende

funktion BerechneHauptknoten()
/* Berechnet den Hauptknoten eines Kommunikationsgraphen nach
   folgenden Kriterien:
    - Wähle den Knoten mit dem höchsten Grad
    - Wenn mehrere Knoten den selben höchsten Grad besitzen
      wähle einen von ihnen aus
*\
```

```
funktion BerechneHauptknotenInnerenTeilgraphen()
/* Berechnet den Hauptknoten eines inneren Teilgraphen nach
   folgenden Kriterien:
     - Wähle den Knoten mit dem höchsten Grad
     - Wenn mehrere Knoten den selben höchsten Grad besitzen dann
       wähle aus diesen Knoten den Knoten mit dem geringsten
       Abstand zum Hauptknoten des unabhängigen Teilgraphen zu dem
       der innere Teilgraph gehört.
     - Wenn mehrer Knoten den selben Abstand besitzen wähle
       einen von ihnen aus.

       Variante 2.1: Hier werden die Grade des ursprünglichen
       Graphen bei der Wahl des Hauptknoten verwendet.
       Variante2.2: Hier werden die Grade des kürzesten
       Wegegraphen bei der Wahl des Hauptknoten verwendet. *\
```

In Experiment E testen wir nochmal, welche Variation unseres Algorithmus die Beste ist. Dazu betrachten wir diesen Kommunikationsgraph 13. Er besteht aus 40 Knoten mit dem Hauptknoten a1. Abbildung 14 zeigt den kürzesten Wegegraph, ausgehend von Knoten a1. Im vollständigen Graphen gibt es 2 innere Teilgraphen. Im kürzesten Wegegraph sind es 5 innere Teilgraphen.

Außerdem wollen wir noch unsere Hypothese weiter untersuchen, die wir in Experiment C nicht bestätigen konnten.

Hypothesen

Hypothese 1: Es ist besser bei unserem Algorithmus die inneren Teilgraphen der Größe nach absteigend zu sortieren.

Ordnung A Dies ist die Ordnung, wie sie von JPF-BDD aus dem Programm eingelesen wird, wenn wir keine eigene Ordnung angeben. Sie soll uns zum Vergleich mit anderen Ordnungen dienen.

Ordnung B Bei dieser Ordnung haben wir nach dem Algorithmus mit Variante 1 geordnet.

Ordnung C Bei dieser Ordnung haben wir nach dem Algorithmus mit Variante 2.1 geordnet.

Ordnung D Bei dieser Ordnung haben wir nach dem Algorithmus mit Variante 2.2 geordnet.

Ordnung E In Ordnung E haben wir die inneren Teilgraphen der Größe nach aufsteigend geordnet. Ansonsten haben wir beim Algorithmus Variante 2.2 verwendet. Dadurch wollen wir unsere Hypothese überprüfen. Wir erwarten, durch die aufsteigende Ordnung der inneren Teilgraphen, einen Anstieg der Laufzeit im Vergleich zu Ordnung D.

Ergebnisse Laufzeiten des Programms mit den verschiedenen Ordnungen können wir in dieser Tabelle 15 betrachten.

Mit den Varianten 2.1 und 2.2 die in Ordnung C und D verwendet wurde, verringert sich die Laufzeit um die Hälfte im Vergleich zu Ordnung B, die Variante 1 des Algorithmus verwendet. Also müssen wir uns zwischen Variante 2.1 und 2.2 entscheiden.

Zwischen Ordnung C und Ordnung D lässt sich nur ein geringer Unterschied zu Gunsten von Ordnung D messen. Da wir auch in diesem Experiment keinen signifikanten Unterschied zwischen Variante 2.1 und 2.2 feststellen konnten, entscheiden wir uns für Variante 2.2.

Mit Ordnung E messen wir eine deutliche Steigerung der Laufzeit im Vergleich zu Ordnung D. Damit bestätigt sich unsere Hypothese 1, dass es besser ist, die inneren Teilgraphen der Größe nach absteigend zu sortieren.

Durch Experiment E konnten wir unsere Hypothese 1 bestätigen. Außerdem haben wir uns für Variante 2.2 unseres Algorithmus entschieden. Da alle unsere bisherigen Kommunikationsgraphen zusammenhängende Graphen war, stellt sich uns die Frage wie wir unabhängige Teilgraphen sortieren sollen. Wir vermuten, dass es auch besser ist die unabhängigen Teilgraphen der Größe nach absteigend zu sortieren.

Wir passen den Algorithmus entsprechend unserer neuen Hypothese an. Dass heißt es werden zuerst alle unabhängigen Teilgraphen berechnet und der Größe nach absteigend sortiert. Dann wird auf jeden der Teilgraphen unser bisheriger Algorithmus angewendet.

Außerdem wird noch folgende Eigenschaft bei der Wahl des Hauptknoten hinzugefügt. Wenn es mehrere Knoten mit dem selben höchsten Grad gibt, dann berechne für diese Knoten den kürzesten Wegegraphen und den maximalen Abstand dieses kürzesten Wegegraphen. Wähle den Knoten mit dem minimalen maximalen Abstand als Hauptknoten. Wenn mehrere Knoten denselben Abstand haben wähle einen von ihnen aus.

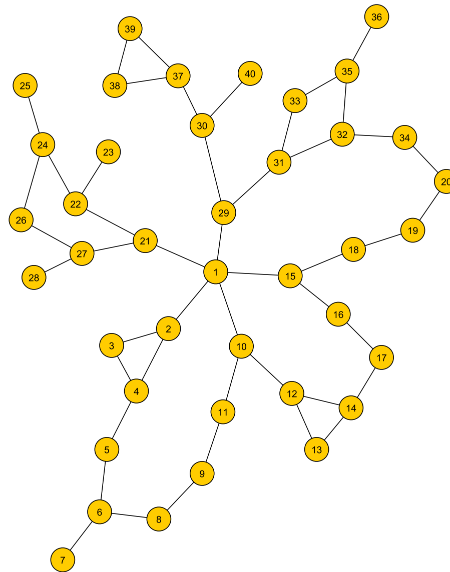

Abbildung 13: Kommunikationsgraph zu ExperimentE

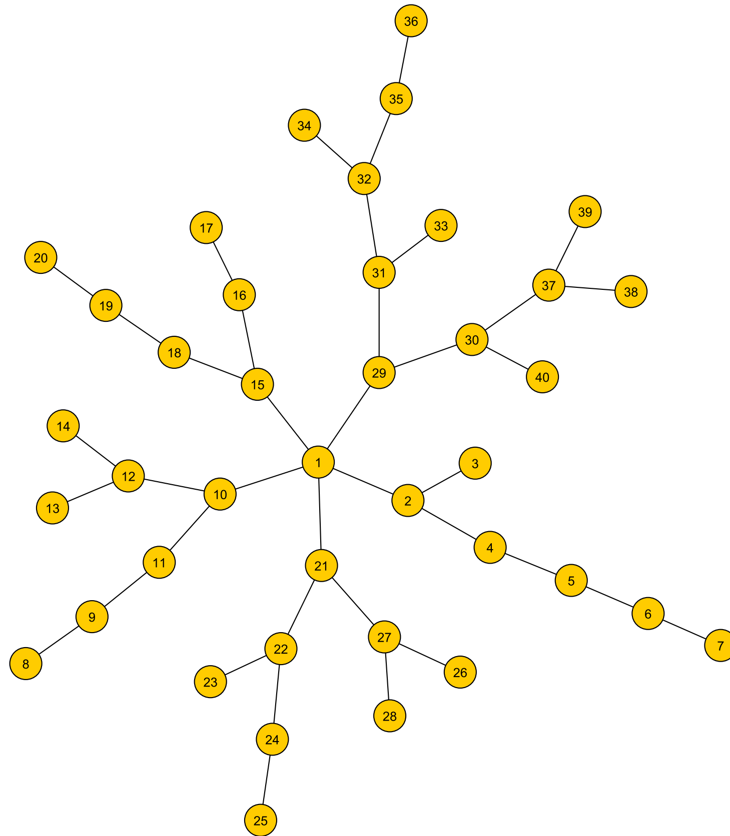

Abbildung 14: kürzeste Wegegraph von 13 ausgehend von Knoten a1

Bezeichnung	Variablenordnung	Laufzeit
A	1>2>3>...>40	1:10m
B	1>2>10>15>29>31>32>...	54s
C	1>29>31>...>12>14>13>11>...	25s
D	1>29>31>...>12>13>14>11>...	24s
E	1>15>16>17>18>19>20>...	10:23m

Abbildung 15: Laufzeiten zu ExperimentE

4.6 Experiment F, G, H

Algorithmus

```
gegeben:  Kommunikationsgraph Graph K, Variablenordnung V
funktion BerechneVariablenordnung() start
    Liste<Graphen> unabGraphen =
        berechneUnabhängigeTeilgraphen(K);
    sortiereUnabhängigeTeilgraphenAbsteigend(unabGraphen);
    for(int i= 0; i < unabGraphen.size(); i++){
        Graph g = unabGraphen.get(i);
        Knoten k = g.BerechneHauptknoten();
        V.add(k);
        Graph kg = g.BerechneKürzestenWegeGraph(k);
        Liste<Graphen> innTeilgraphen =
            kg.BerechneInnereTeilgraphen(k);
        sortiereInnereTeilgraphenAbsteigend(innTeilgraphen);
        for(int j = 0; j < innTeilgraphen.size(); j++){
            Graph innG = innTeilgraphen.get(j);
            BerechneOrdnungInnererTeilgraphen(innG);
        }
    }
    Gibt V zurück;
ende
funktion BerechneOrdnungInnererTeilgraphen (Graph g) start
    Knoten k = innG.BerechneHauptknotenInnererTeilgraphen();
    V.add(k);
    if(g.size() > 1){
        Liste<Graphen> innTeilgraphen =
            innG.BerechneInnereTeilgraphen(k);
        sortiereInnereTeilgraphenAbsteigend(innTeilgraphen);
        for(int i = 0; i < innteilgraphen.size(); i++){
            Graph innG = innTeilgraphen.get(i);
            BerechneOrdnungInnererTeilgraphen(innG);
        }
    }
ende

funktion BerechneHauptknoten()
/* Berechnet den Hauptknoten eines Kommunikationsgraphen nach
   folgenden Kriterien:
    - Wähle den Knoten mit dem höchsten Grad.
    - Wenn mehrere Knoten den selben höchsten Grad besitzen, dann
      berechne den Kürzesten Wegegraphen für diese Knoten und
      den maximalen Abstand zu diesen Knoten.
    - Wähle dann den Knoten mit dem minimalen maximalen
```

```
      Abstand aus.
    - Wenn mehrer Knoten des selben Abstand haben dann wähle
      einen dieser Knoten.
*\

funktion BerechneHauptknotenInnerenTeilgraphen()
/* Berechnet den Hauptknoten eines inneren Teilgraphen nach
   folgenden Kriterien:
    - Wähle den Knoten mit dem höchsten Grad.
    - Wenn mehrere Knoten den selben höchsten Grad besitzen
      dann wähle aus diesen Knoten den Knoten mit dem geringsten
      Abstand zum Hauptknoten des unabhängigen Teilgraphen zu
      dem der innere Teilgraph gehört
    - Wenn mehrer Knoten den selben Abstand besitzen wähle
      einen von ihnen aus */
```

In diesem Kapitel fassen wir die 3 Experimente F,G und H zusammen. Mit allen 3 Experimenten wollen wir Hypothese 1 untersuchen. Abbildung 16 zeigt den Graphen von Experiment F. Der Graph von Experiment G besteht aus dem Graphen von Experiment F, sowie den unabhängigen Teilgraphen die Abbildung 17 zeigt. Der Graph von Experiment H wiederrum besteht aus dem Graphen von Experiment G, sowie den unabhängigen Teilgraphen die Abbildung 18 zeigt. Bei jedem Experiment werden 2 Ordnungen miteinander verglichen.

Hypothesen

Hypothese 1: Es ist besser unabhängige Teilgraphen der Größe nach absteigend zu sortieren.

Ordnung A In Ordnung A werden die unabhängigen Teilgraphen der Größe nach absteigend sortiert.

Ordnung B In Ordnung B werden die unäbhängigen Teilgraphen der Größe nach aufsteigend sortiert.

Ergebnisse Laufzeiten des Programms mit den verschiedenen Ordnungen können wir in dieser Tabelle 19 betrachten.

Entgegen unseren Erwartungen liefert Ordnung B in allen Experimenten eine bessere Laufzeit als Ordnung A. In Experiment F ist der Unterschied noch sehr gering. Aber mit ansteigender Anzahl von unabhänigen Teilgraphen in Ordnung G und H wird der Unterschied größer.

Dadurch ist Hypothese 1 wiederlegt und wir passen den Algorithmus entsprechend an. Das heißt, dass unabhängige Teilgraphen nun der Größe nach aufsteigend sortiert werden.

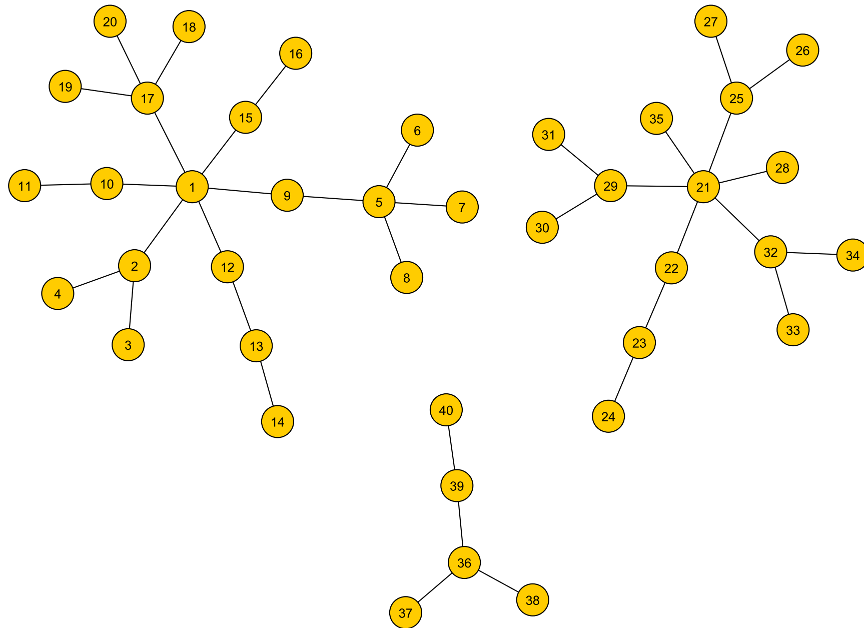

Abbildung 16: Kommunikationsgraph zu ExperimentF

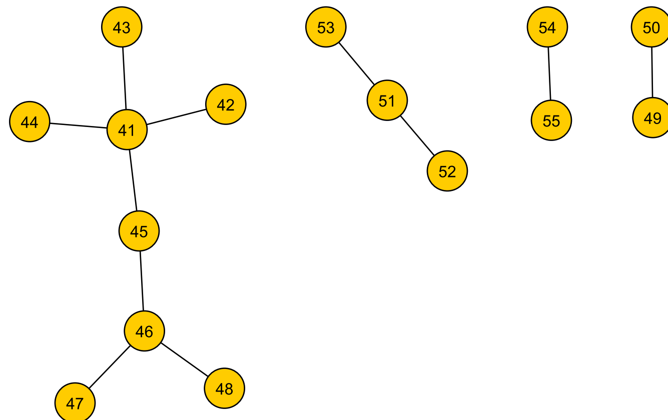

Abbildung 17: Erweiterung zu Graph 16

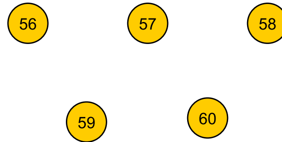

Abbildung 18: Erweiterung zu Graph 17

Experiment	Laufzeit bei Ordnung A	Laufzeit bei Ordnung B
F	3s	1s
G	9s	1s
H	11s	1s

Abbildung 19: Laufzeiten zu ExperimentFGH

4.7 Algorithmus für die Variablenordnung

Der Vollständige Algorithmus sieht so aus:

```
gegeben:  Kommunikationsgraph Graph K, Variablenordnung V
funktion BerechneVariablenordnung() start
    Liste<Graphen> unabGraphen =
        berechneUnabhängigeTeilgraphen(K);
    sortiereUnabhängigeTeilgraphenAufsteigend(unabGraphen);
    for(int i= 0; i < unabGraphen.size(); i++){
        Graph g = unabGraphen.get(i);
        Knoten k = g.BerechneHauptknoten();
        V.add(k);
        Graph kg = g.BerechneKürzestenWegeGraph(k);
        Liste<Graphen> innTeilgraphen =
            kg.BerechneInnereTeilgraphen(k);
        sortiereInnereTeilgraphenAbsteigend(innTeilgraphen);
        for(int j = 0; j < innTeilgraphen.size(); j++){
            Graph innG = innTeilgraphen.get(j);
            BerechneOrdnungInnererTeilgraphen(innG);
        }
    }
    Gibt V zurück;
ende
funktion BerechneOrdnungInnererTeilgraphen (Graph g) start
```

```
    Knoten k = innG.BerechneHauptknotenInnererTeilgraphen();
    V.add(k);
    if(g.size() > 1){
        Liste<Graphen> innTeilgraphen =
            innG.BerechneInnereTeilgraphen(k);
        sortiereInnereTeilgraphenAbsteigend(innTeilgraphen);
        for(int i = 0; i < innteilgraphen.size(); i++){
            Graph innG = innTeilgraphen.get(i);
            BerechneOrdnungInnererTeilgraphen(innG);
        }
    }
ende

funktion BerechneHauptknoten()
/* Berechnet den Hauptknoten eines Kommunikationsgraphen
   nach folgenden Kriterien:
    - Wähle den Knoten mit dem höchsten Grad.
    - Wenn mehrere Knoten den selben höchsten Grad besitzen,
      dann berechne den Kürzesten Wegegraphen für diese
      Knoten und den maximalen Abstand zu diesen Knoten.
    - Wähle dann den Knoten mit dem minimalen maximalen
      Abstand aus.
    - Wenn mehrer Knoten des selben Abstand haben dann
      wähle einen dieser Knoten.
*\

funktion BerechneHauptknotenInnerenTeilgraphen()
/* Berechnet den Hauptknoten eines inneren Teilgraphen nach
   folgenden Kriterien:
    - Wähle den Knoten mit dem höchsten Grad.
    - Wenn mehrere Knoten den selben höchsten Grad besitzen,
      dann wähle aus diesen Knoten den Knoten, mit dem
      geringsten Abstand zum Hauptknoten des unabhängigen
      Teilgraphen zu dem der innere Teilgraph gehört
    - Wenn mehrer Knoten den selben Abstand besitzen
      wähle einen von ihnen aus.
*/
```

5 Testreihe zur Prüfung der Optimierung

Jetzt müssen wir noch testen in welchen Fällen es sinnvoll ist die Analyse anzuwenden. Außerdem ist es notwendig, dass wir überprüfen, ob der Algorithmuse auch bei Programmen gute Variablenordnungen berechnet, die nicht aus einem speziellen Kommunikationsgraphen konstruiert wurden.

Leider haben wir keine Programme aus der Praxis zum verifizieren gefunden,

die genügend statische boolsche Variablen verwenden. Da aus diesem Grund die BDD's zu klein geblieben sind, konnten bei verschiedenen Variablenordnungen keine Unterschiede der Laufzeit gemessen werden.

Aus diesem Grund haben wir eine Reihe von Testprogrammen geschrieben. Diese Programme sind nach der Anzahl ihrer boolschen Variablen geordnet. Die Reihe beginnt mit 10 Variablen und steigt in 10er Schritten bis 70 Variablen. Für jede Variablenanzahl wurden 3 verschiedene Programme geschrieben, damit die Varianz zwischen Programmen mit gleicher Variablenanzahl sichtbar wird.

Wie in den Programmen aus den Experimenten, besitzen auch diese Programme eine Main-Methode. In der Main-Methode gibts es verschiedene if-Anweisungen in denen eine globale Zählvariable um 1 erhöht wird, falls die Bedingung der if-Anweisung erfüllt ist. Die Zählvariable wird am Ende des Programms ausgegeben.

Der wichtige Unterschied zu den vorherigen Programmen ist, dass die Variablen in den if-Anweisungen, die Anzahl und die Reihenfolge der if-Anweisungen zufällig bestimmt wurden. Dazu sind wir wie folgt vorgegangen. Zuerst haben wir eine Reihe von if-Anweisungen erstellt, bei denen die Zählvariable um 1 erhöht wird. Dann haben wir immer 2 Variablen, die wir zufällig bestimmt haben, in den Bedingungen der if-Anweisung mit einer UND-Verknüpfung verbunden. Dann haben wir die Reihefolge der if-Anweisung zufällig verändert. Zum Schluss haben wir noch zufällig einige if-Anweisungen verändert, gelöscht oder neue hinzugefügt.

Damit erreichen wir, dass viele verschiedene zufällige Kommunikationsgraphen entstehen. Dadurch hat diese Testreihe eine höhere Aussagekraft als die Experimente, bei denen die Programme aus Kommunkationsgraphen mit speziellen Strukturen konstruiert wurden.

In den einzelnen Tests der Testreihe vergleichen wir immer die Laufzeiten von 2 Variablenordnungen miteinander. Die erste Variablenordnung ist die Ordnung, die entsteht, wenn wir keine eigene Ordnung angeben. Die zweite Variablenordnung ist die optimierte Ordnung, die unser Algorithmus berechnet.

Mit dieser Testreihe wollen wir untersuchen ab welcher Variablenanzahl, die Variablenordnung einen erkennbaren Unterschied bei der Verifikation zeigt. Außerdem wollen wir auch untersuchen in wie vielen Fällen die optimierte Variablenordnung, die wir durch die Analyse bekommen, eine Verbesserung der Laufzeit bewirkt.

5.1 Auswertung des Testreihe

Für die Auswertung der Testreihe betrachten wir das Diagramm 20 und die dazugehörige Tabellen 21 und 22. In den 2 Tabellen wird auch die Analysezeit angegeben. Dabei besteht die Analysezeit immer zu mindestens 90 Prozent aus der Zeit, die benötigt wird um den Kommunikationsgraphen aus den XML-Dateien zu erstellen.

Als erstes fällt uns auf, dass die Tests bis 20 Variablen keine messbaren Unterschiede zwischen den zwei Ordnungen aufweisen. Das liegt daran ,dass hier die erzeugten BDD's zu klein sind um starke Unterschiede zu erzeugen.

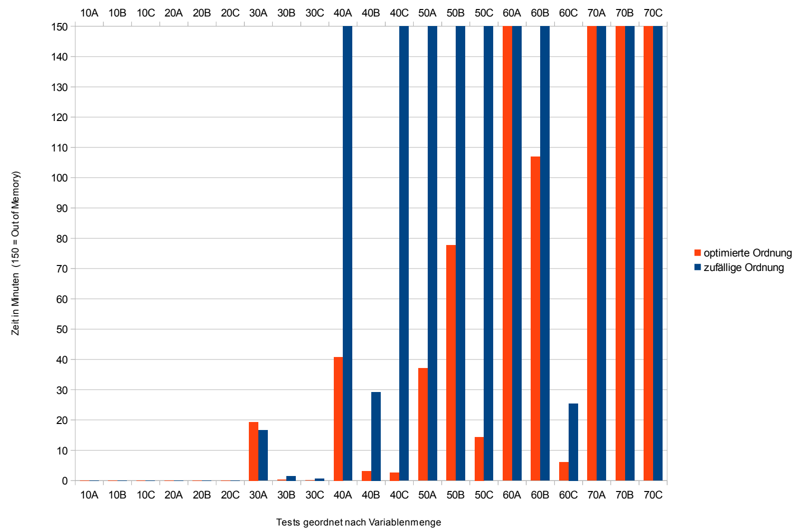

Abbildung 20: Ergebnisse der Testreihe als Diagramm ohne Analysezeiten

Test	Laufzeit mit zufällige Ordnung	Laufzeit mit optimierter Ordnung	Analyse-Zeit
Test10A	0s	0s	2s
Test10B	0s	0s	2s
Test10C	0s	0s	2s
Test20A	3s	3s	5s
Test20B	1s	1s	3s
Test20C	2s	1s	4s
Test30A	16:34m	19:20m	8s
Test30B	1:32m	17s	6s
Test30C	40s	5s	6s

Abbildung 21: Ergebnisse der Testreihe als Tabelle mit Analysezeiten, Teil 1

Test	Laufzeit mit zufällige Ordnung	Laufzeit mit optimierter Ordnung	Analyse-Zeit
Test40A	Out Of Memory	40:46m	12s
Test40B	29:12m	3:12m	8s
Test40C	Out Of Memory	2:41m	8s
Test50A	Out Of Memory	37:06m	16s
Test50B	Out Of Memory	1:17:43h	13s
Test50C	Out Of Memory	14:22m	11s
Test60A	Out Of Memory	Out Of Memory	21s
Test60B	Out Of Memory	1:46:54h	18s
Test60C	25:24m	6:00m	13s
Test70A	Out Of Memory	Out Of Memory	28s
Test70B	Out Of Memory	Out Of Memory	22s
Test70C	Out Of Memory	Out Of Memory	21s

Abbildung 22: Ergebnisse der Testreihe als Tabelle mit Analysezeiten, Teil 2

Bei 30 Variablen sehen wir die ersten Unterschiede. Bei Test 30B und 30C sind die Laufzeiten allerdings so klein das wir die Unterschiede nur in der Tabelle ablesen können. Zwar können wir bei den beiden Test eine Verbesserung der Laufzeit durch die optimierte Variablenodrnung feststellen, allerdings ist die eingesparte Zeit so gering ,dass sie den Aufwand der Analyse nicht rechtfertigt.

Bei Test 30A liegt die Laufzeit bei beiden Ordnungen über 15 Minuten. Das zeigt uns, wie stark sich die Laufzeiten bei Tests mit gleicher Anzahl von Variablen unterscheiden können. Außerdem beobachten wir bei diesem Test eine Verschlechterung der Laufzeit mit der optimierter Variablenordnung. Das weist auf ein mögliches Versagen des Algorithmus hin. Dieses Versagen diskutieren wir in Kapitel 6 noch genauer.

Bei den Test mit 40 und 50 Variablen können wir durchgehend eine starke Verbesserung der Laufzeit mit optimierter Variablenordnung verzeichnen. In 5 von den 6 Tests wird die Verifikation bei den zufälligen Ordnungen sogar wegen zu wenig Speicher abgebrochen. Die optimierte Variablenordnung ermöglicht in diesen Fällen das Beenden der Verifikation. Hier sehen wir welch großes Potenzial der Algorithmus hat.

Auch bei den Tests mit 60 Variablen können wir Laufzeitverbesserungen beobachten. Bei Test 60A allerdings wird das BDD trotz optimierter Ordnung so groß, dass der Speicher nicht mehr ausreicht.

Ab den Tests mit 70 Variablen werden bei allen Tests die BDD's zu groß. Dadurch bricht die Verifikation in allen Fällen wegen zu wenig Speicher ab.

Zusammenfassend können wir durch diese Tests 2 Aussagen treffen. Erstens ist es erst ab 30 bis 40 Variablen sinnvoll die Ordnung zu optimieren, da bei weniger Variablen die BDD's zu klein sind um signifikante Unterschiede festzustellen.

Zweitens konnten wir feststellen das beim Großteil der Tests, die Optimierung der Ordnung mit dem Algorithmus eine Verbesserung der Laufzeit zur Folge hatte. Wir müssen aber auch ganz klar hervorheben, dass der Algorithmus versagen kann, was sogar zu einer Verschlechterung der Laufzeit führen kann.

6 Grenzen des Algorithmus

In der Testreihe in Kapitel 5.1 haben wir beobachten, dass der Algorithmus in vielen Fällen eine Variablenordnung berechnet, die eine bessere Laufzeit liefert, als wenn wir keine Ordnung angeben. Allerdings haben wir auch gesehen, dass die Heuristik manchmal eine schlechtere Variablenordnung berechnet. Wir diskutieren nun darüber, wann ein solches Versagen des Algorithmus auftreten kann.

Zuerst wollen wir noch einmal hervorheben, dass dies eine heuristischer Algorithmus ist. Das hat zur Folge das die Ordnung, die durch den Algorithmus berechnet wird, nicht die bestmögliche Ordnung sein muss. So könnte bei Test30A aus obiger Testreihe die nichtoptimierte Ordnung zufällig besser gewesen sein als die optimierte Ordnung.

Ein weiterer Grund für das Versagen des Algorithmus kann auch die Beschaffenheit des Graphen sein. So könnte zum Beispiel der Hauptknoten den der Algorithmus berechnet ungünstig sein, obwohl er den höchsten Grad hat. Das kann passieren wenn es viele Knoten im Graphen gibt, deren Grad nur um 1 niedriger ist als der des Hauptknoten.

Neben diesem Sonderfall gibt es bestimmt noch weitere Fälle, die wir bis jetzt nicht identifiziert haben. Einige dieser Sonderfälle können wahrscheinlich durch eine Optimierung des Algorithmus behandelt werden, um damit die Erfolgschancen der Analyse zu steigern. Allerdings ist ein Auftreten solcher Sonderfälle eher unwahrscheinlich. Daher lohnt sich die Optimierung des Algorithmus auf diese Sonderfälle vermutlich nicht.

Außerdem können wir noch sagen, dass das Versagen des Algorithmus immer wahrscheinlicher wird, je näher sich der Kommunikationsgraph an einen vollständigen Graphen annähert. Solche Graphen entstehen besonders dann, wenn im Programm lange boolsche Audrücke mit vielen verschiedenen Variablen vorkommen. Der Grund weshalb der Algorithmus hier versagen könnte ist, dass bei einem nahezu vollständigen Kommunikationsgraph viele Knoten vom Algorithmus gleich bewertet werden. Dadurch wird die Ordnung dieser Knoten zufällig bestimmt. Die Variablenordnung, die man dann erhält kann sowohl gut als auch schlecht sein.

7 Ausblick

Betrachten wir nun die Praxistauglichkeit der Analyse im Gesamten. Da die Analyse bis jetzt nur mit statischen boolschen Variablen arbeitet, gibt es in der Praxis keine Programme, bei denen sich der Einsatz dieser Analyse lohnen würde. Der nächste Schritt ist also die Analyse dynamisch während der Verifikation durchzuführen. Auch wenn wir das in dieser Arbeit nicht gemacht haben, wollen wir kurz erläutern welche Schritte dazu durchgeführt werden müssen und welches Potenzial dadurch entstehen könnte. Wir erläutern ebenfalls die Probleme denen man sich stellen muss.

Der erste Schritt ist es den Kommunikationsgraphen dynamisch, während der Verifikation zu erzeugen und ihn zu aktualisieren, wenn neue Kanten und Knoten hinzukommen. Dadurch würden wir uns die Verwendung von Soot sparen. Somit müssten wir keine XML-Dateien von Soot mehr auslesen, was bei der aktuellen Analyse die meiste Zeit verbraucht.

Als nächstes wird dann die Ordnung während der Verifikation berechnet und das BDD umgeordnet. Hier tritt das Hauptproblem auf, dem man sich stellen muss. Wenn man bei jeder Änderung im Kommunikationsgraphen die Ordnung neu berechnet und dann gegebenenfalls das BDD umordnet, kann man schnell die Zeit wieder verlieren, die man durch die optimierte Ordnung einspart.

Es ist also nötig ein Bewertungssystem einzuführen, welches entscheidet, ob sich eine Neuberechnung der Ordnung lohnt, wenn sich etwas im Kommunikationsgraphen ändert. Hier könnte man zum Beispiel einführen, dass die Ordnung erst nach einer bestimmten Anzahl von Änderungen neu berechnet wird. Ebenfalls könnte man erst ab einer bestimmten Anzahl an Variablen anfangen überhaupt eine Ordnung zu berechnen. Denn wie wir gesehen haben lohnt sich eine Optimierung der Ordnung bei wenigen Variablen kaum.

Es ist aber noch wichtiger ein Bewertungssystem einzuführen, welches entscheidet ob es sich lohnt bei einer veränderten Ordnung das BDD auch wirklich umzuordnen. Der Grund dafür ist, dass das Umordnen von BDD's mehr Zeit in Anspruch nimmt, als die Neuberechnung der Ordnung. Deshalb müsste man hier die alte und neue Ordnung miteinander vergleichen und bewerten, ob die Veränderungen eine Neuordnung rechtfertigen. Wenn sich zum Beispiel nur 2 Variablen, die in der Ordnung sehr weit hinten stehen vertauschen lohnt sich ein umordnen eher nicht. Wenn dagegen aber eine Variable, die weit hinten in der Variablenordnung steht, zum neuen Hauptknoten wird, sollte das BDD umgeordnet werden. Ein solches Bewertungssystem zu entwickeln wird wohl die größte Schwierigkeit beim Erweitern der Analyse werden.

Aber auch mit einer dynamischen Analyse bleibt das Problem bestehen, dass sich die Analyse nur bei ausreichend vielen Variablen im BDD lohnt. Daher ist es fraglich, ob die Analyse nach der Erweiterung einen praktischne Nutzen hat.

8 Fazit

In dieser Arbeit haben wir untersucht wie stark sich die Variablenordnung auf die Laufzeit von JPF-BDD auswirkt. Dabei haben wir festgestellt das sich die Laufzeit bei verschiedenen Ordnungen sehr stark ändern kann. In Experiment C ist die Laufzeit von ca 2s auf über 15 Minuten angestiegen, als wir eine schlechte Ordnung angegeben haben. Siehe hierzu die Ergebnisse von Experiment C in Abbildung 10. Allerdings ergeben sich solche Unterschiede erst bei BDD's mit vielen Variablen.

Außerdem haben wir einen heuristischen Algorithmus entwickelt, der aus einem Kommunikationsgraphen eine optimierte Variablenordnung berechnet. Der Kommunikationsgraph wird von unserem Analyseprogramm berechnet. Auch den Algorithmus haben wir in unserem Analyseprogramm implementiert.

Mit einer Testreihe haben wir dann unseren Algorithmus auf seine Erfolgsaussichten geprüft, eine gute Variablenordnung zu berechnen. In den meisten Fällen hat der Algorithmus dabei eine Variablenordnung berechnet, die zu einer Verbesserung der Laufzeit führte. Diese Fälle haben uns das Potenzial des Algorithmus gezeigt. In den anderen Fällen konnte keine Unterschied der Laufzeit gemessen werden, weil die BDD's zu klein waren.

Außerdem gab es auch einen Testfall, bei dem der Algorithmus sogar eine schlechter Variablenordnung berechnet hat, die zu einer Erhöhung der Laufzeit geführt hat. Dieses Ergebniss hat uns gezeigt, dass der Algorithmus auch versagen kann. Mögliche Gründe für ein solches Versagen haben wir in Kapitel 6 diskutiert.

Leider ist unser Analyseprogramm für die Praxis noch nicht nützlich. Das liegt hauptsächlich an der Einschränkung auf statische boolsche Variablen. Um auch für dynamische Variablen eine Ordnung berechnen zu können, müssten wir die BDD's während der Verifikation umordnen. Die Zeit, die für die Umordnung gebraucht wird, darf dabei nicht unterschätzt werden. Aber auch nachdem die Analyse erweitert wird, ist es fraglich, ob dieses Analyseverfahren in der Praxis nützlich ist.

Literatur

[1] Adnan Aziz, Serdar Tasiran, and Robert K. Brayton. Bdd variable ordering for interacting finite state machines. In *IN PROC. OF THE DESIGN AUTOMATION CONF*, pages 283–288, 1994.

[2] Dirk Beyer. Improvements in BDD-based reachability analysis of timed automata. In *Proc. FME*, LNCS 2021, pages 318–343. Springer, 2001.

[3] Beate Bollig and Ingo Wegener. Improving the variable ordering of obdds is np-complete. *IEEE Transactions on Computers*, 45:993–1002, 1996.

[4] Randal E. Bryant. Graph-based algorithms for boolean function manipulation. *IEEE Transactions on Computers*, 35:677–691, 1986.

[5] Seon-Woong Jeong, Bernard Plessier, Gary D. Hachtel, and Fabio Somenzi. Variable ordering and selection for fsm traversal. In *ICCAD*, pages 476–479, 1991.

[6] Richard Rudell. Dynamic variable ordering for ordered binary decision diagrams. In *Proceedings of the 1993 IEEE/ACM international conference on Computer-aided design*, ICCAD '93, pages 42–47, Los Alamitos, CA, USA, 1993. IEEE Computer Society Press.

[7] Hervé J. Touati, Hamid Savoj, Bill Lin, Robert K. Brayton, and Alberto L. Sangiovanni-Vincentelli. Implicit state enumeration of finite state machines using bdds. In *ICCAD*, pages 130–133, 1990.

[8] Raja Vallée-Rai, Phong Co, Etienne Gagnon, Laurie Hendren, Patrick Lam, and Vijay Sundaresan. Soot - a java bytecode optimization framework. In *Proceedings of the 1999 conference of the Centre for Advanced Studies on Collaborative research*, CASCON '99, pages 13–. IBM Press, 1999.

[9] Willem Visser, Matthew B. Dwyer, and Michael W. Whalen. The hidden models of model checking. *Software and System Modeling*, 11(4):541–555, 2012.

[10] Alexander von Rhein, Sven Apel, and Franco Raimondi. Introducing binary decision diagrams in the explicit-state verification of Java code. In *Proc. Java Pathfinder Workshop*, 2011.

A Anhang

Der Anhang ist auf der beiliegenden CD.
Der Anhang besteht aus:

- Source Code der Experimente
(Anlage/Experimente/src/gov/nasa/jpf/test/bdd/)
- jpf-Dateien der Experimente
(Anlage/Experimente/src/gov/nasa/jpf/test/bdd/)
- Variablenordnungen der Experimente
(Anlage/Experimente/orders/)
- Soot-XML-Dateien der Experimente
(Anlage/Experimente/sootOutput/)
- Ergebnisse der Experimente in Form der Ausgabe von JPF-BDD
(Anlage/Experimente/Ergebnisse/)
- Source Code der Test aus der Testreihe
(Anlage/Experimente/src/)
- jpf-Dateien der Tests aus der Testreihe
(Anlage/Experimente/src/)
- Variablenordnungen der Tests aus der Testreihe
(Anlage/Experimente/orders/orderTestsWithXVars/)
- Soot-XML-Dateien der Tests aus der Testreihe
(Anlage/Experimente/sootOutput/)
- Ergebnisse der Testreihe in Form der Ausgabe von JPF-BDD
(Anlage/Experimente/ErgebnisseTestsWithXVars/)
- Source Code des Analyseprogramms
(Anlage/bdd-Analyse/)
- Source Code von JPF-BDD
(Anlage/jpf-bdd/)
Der Source Code von JPF-BDD ist ebenfalls auf https://bitbucket.org/rhein/jpf-bdd erhältlich (Branch: variable_detection, Commit: 200d19d)