Hybride Systeme

Implementing a CSP Solver

Alexander Camu Jörn Pochodaj

Svenja Schuirmann

Inhalt

[Einleitung 2](#_Toc16703307)

[Problemstellung 2](#_Toc16703308)

[Lösung 3](#_Toc16703309)

[Theoretischer Teil 3](#_Toc16703310)

[Praktischer Teil 6](#_Toc16703311)

[Fazit 11](#_Toc16703312)

[Probleme 11](#_Toc16703313)

[Zeiterfassung 12](#_Toc16703314)

# Einleitung

Constraint Satisfaction Problems (CSP) beschreibt ein Problem als Modell. Das Modell beinhaltet dabei Variablen, deren Domänen und Constraints. Die Constraints beschreiben die Beziehungen der Variablen zueinander. Diese CSPs werden vielfältig eingesetzt. Zum Beispiel finden sie Anwendung in der Molekularen Biologie, in der Elektrotechnik und netzwerk Management.

Ziel ist es für ein CSP einen Zustand zu finden für welchen alle Constraints erfüllt sind. Es werden also eine oder mehrere Lösungen gesucht die alle Constraints erfüllen und nicht nur möglichst viele wie es bei Optimierungsproblemen der Fall ist. Existiert ein Zustand für den alle Constraints erfüllt sind nennt man dieses CSP erfüllbar, ist es nicht der Fall ist es unerfüllbar. Diese Ausarbeitung befasst sich nun mit der Theorie von CSPs und der Entwicklung und Implementierung eines Lösers für diese.

# Problemstellung

Die CSPs mit welchen es sich in dieser Ausarbeitung beschäftigt wird, sind sehr eingeschränkt. Diese einfachen CSPs sind so aufgebaut, dass die simple bounds welche ein Constraint bilden folgende Form haben x ≥ y + k. X und y sind die Variablen mit jeweils ihren Domänen. X und y sind nur ganze Zahlen. Alle simple bounds sind miteinander verodert. Damit ein Constraint erfüllt ist muss also nur ein simple bound wahr sein. Es soll nun ein Löser entwickelt werden, welche für diese CSPs ausgibt, ob sie satisfiable oder not satisfiable sind. Um dies zu erleichtern sind zwei Algorithmen gegeben. Algorithmus A besteht aus drei Schritten:

* Consistency check
* Backtrack
* Decision

Im ersten Schritt werden die Constraints überprüft. Dazu müssen die simple Bounds kontrolliert werden. Es wird überprüft, ob min(x) >= max(y) + k ist und nicht max(x) < min(y) + k. Ist beides der Fall, ist der Bound wahr und der ganze Constraint somit wahr. Ist dies nicht der Fall ist der Constraint falsch und es geht weiter im zweiten Schritt. Ist dagegen bei der Überprüfung beide Abfragen wahr oder falsch ist der Bound inconclusive und es geht im dritten Schritt weiter.

Im dritten Schritt wird eine Variable ausgesucht und von dieser die Domain aufgeteilt. Mit einem Teil wird der erste Schritt von vorne gestartet und die andere wird abgespeichert.

Wird der zweite Schritt ausgeführt, wird der letzte abgespeicherte Teil einer Domain gegen den genutzten Teil ausgetauscht. Ist keine Domain mehr abgespeichert ist das CSP not satisfiable. Sind alle Constraint wahr ist das CSP satisfiable.

Dieser Algorithmus soll implementiert und verbessert werden. Eine solche Verbesserung ist Algorithmus B, wo der erste Schritt durch einen Deduction Schritt erweitert wird. Beide Algorithmen sollen nun implementiert und verbessert werden. Außerdem wird die Laufzeit der Algorithmen miteinander verglichen. Die Ausarbeitung beschäftigt sich mit der Theorie hinter diesen Algorithmen.

# Lösung

## Theoretischer Teil

1a)

Proposition 1: Sei p eine Intervallabschätzung. Wenn alle Simple Constraints eines CSP P von p erfüllt sind, dann existiert eine (Punkt-)Lösung von P und damit ist P erfüllbar.

Aufgabenstellung a Part 1: Warum gilt Proposition 1?

Sei p eine Intervallabschätzung und P = <X, D, C> mit X = <x1, x2, … , xn>. Dann ist p definiert als p(x1) = [j1n, j2m], p(x2) = [j2n, j2m], … , p(xn) = [jnn, jnm], wobei p(xi) eine Intervallabschätzung für die jeweilige Variable ist. Nach Proposition 1 sind alle c1, c2, …, cn = C erfüllt. Da ein Simple Constraint aus veroderten Simplen Bounds besteht, bedeutet dies, dass jedes ci mindestens ein Simple Bound sbij enthalten muss, welches True unter p ist.

Da jedes Simple Constraint mindestens einen Simple Bound enthält, der unter p true ist, muss eine Wertebelegung aus dem Intervall p(xi) für jedes Xi aus X existieren, für welche die einzelnen Simple Bounds true sind.

Daraus folgt, dass ein n-Tupel A = (a1, a2, …, an) existiert aus dem Wertebereich des Intervalls p. Da jedes p(xi) eine Teilmenge von Di ist mit D = <D1, D2, …, Dn>. Damit ist die Lösung A auch aus dem Wertebereich D und somit ist P erfüllbar.

Darum gilt Proposition 1.

Aufgabenstellung a Part 2: Wie kann eine Lösung aus p extrahiert werden?

Da P erfüllbar ist unter p, gilt folgendes:

Jeder Wert aus dem Intervall p(Xi) für das Xi ist für P erfüllbar. Denn jede Kombination von Werten aus den Intervallen p(Xi) ergibt ein n-Tupel, welches P erfüllt. Daher kann eine Lösung extrahiert werden, indem die jeweils unterste Grenze der p(Xi) Intervalle gewählt wird.

Somit ist A = (min(p(x1)), min(p(x2)), …, min(p(xn))) eine Lösung von P. Auf diese Weise kann eine Lösung aus p extrahiert werden.

1b) Ist der Algorithmus sound und complete, d.h. liefert der Algorithmus für jedes CSP P das richtige Ergebnis?

Der Algorithmus prüft für die gegebene Intervallabschätzung p, ob die Simple Constraints true, false oder inconclusive sind. Wenn ein Simple Constraint für p inconclusive ist, wird das Intervall einer Variablen in zwei Unterintervalle p‘ und p‘‘ geteilt, welche das zu teilende Intervall vollständig abbilden. Dieser Vorgang wird so lange wiederholt, bis für ein daraus resultierendes neues p entschieden werden kann, ob ein Simple Constraint false ist oder alle Simple Constraints true sind. Wenn ein Simple Constraint false ist, wird das jeweils andere Intervall geprüft und nach den oben beschriebenen Schritten behandelt.

Sobald ein p gefunden wird, für welches alle Simple Constraints true sind, terminiert der Algorithmus mit der Ausgabe, dass das P erfüllbar ist. Wenn kein Intervall übrig ist, welches noch nicht vollständig vom Algorithmus behandelt wurde und für welches bislang kein p gefunden wurde für welches alle Simple Constraints true sind, terminiert der Algorithmus mit der Lösung, dass P nicht erfüllbar ist.

Der Algorithmus splittet alle Intervalle nacheinander in eine Baumstruktur auf, sofern er noch nicht terminiert ist und prüft diese Intervalle auf der jeweils unterste Ebene. Im Worst Case besteht ein Intervall dabei aus einem einzelnen Wert. Da für einen einzelnen Wert ein Simple Constraint nicht inconclusive sein kann, findet spätestens auf dieser Ebene die Entscheidung statt, ob ein Simple Constraint false ist oder ob alle Simple Constraints true sind.

Selbst wenn das Ursprungsintervall p solange aufgesplittet werden muss bis alle Werte in einzelnen Intervallen geprüft wurden, so terminiert der Algorithmus dennoch mit der jeweils richtigen Lösung, da alle Intervalle einzeln geprüft wurden.

Des Weiteren gibt es keinen Wertebereich, der nicht geprüft wird, da die Aufteilung einer Intervallabschätzung p immer in p‘ und p‘‘ erfolgt, welche p vollständig abdecken.

Somit wird keine inkorrekte Lösung ausgegeben. Jedoch ist mit der Lösung nicht sichergestellt, dass die Lösung vollständig ist.

Der Algorithmus liefert für jedes CSP eine korrekte Lösung.

1c) Warum ist Constraint Solving NPC für die gegebenen Einschränkungsformel:

Zu beweisen: Constraint Solving ist in diesem Kontext NPC.

Vorgehen: Um zu beweisen, dass CS Element NPC ist müssen zwei Dinge gezeigt werden:

1. CS ist Element NP
2. CS ist Element NP-Schwer

Anforderung a) sagt aus, dass sich Constraint Solving nichtdeterministisch zu polynomieller Laufzeit lösen lässt. Dies wird über die Bestimmung der Komplexitätsklasse O(CS) gezeigt.

Ein CSP besteht aus 1…n Simple Constraints mit jeweils m1…mn Simple Bounds. Das Solving eines solchen Problems kann am uneffizientesten über Brute Forcing gelöst werden. Dies geschieht, indem jedes Simple Bound in einem Simple Constraint separat für jeden Wert in p geprüft wird.

Die Prüfung eines einzelnen Simple Constraint für einen einzelnen Wert liegt in der Aufwandsklasse O(m). Diese Prüfung muss nun für alle Simple Constraints durchgeführt werden, damit verändert sich der Aufwand zu O(n²). Diese Prüfung muss nun auch für jeden Wert in p durchgeführt werden. Damit befindet sich Contraint Solving in der Größenordnung der Aufwandsklasse O(n³).

CS ist damit Element von NP.

Anforderung b) sagt aus, dass sich Constraint Solving zu nichtdeterministisch polynomieller Laufzeit schwerer lösen lässt, als die schwersten Probleme, die Element von NP sind.

Um dies zu zeigen wenden wir eine Reduktion an und zeigen, dass diese Reduktion sich über Algorithmen zu nichtdeterministischer Laufzeit lösen lassen.

Wir reduzieren das Constraint Solving auf ein CSP P1, welches aus einem einzigen Simple Constraint mit n Simple Bounds besteht.

P1 lässt sich nun mit Algorithmus A lösen. Algorithmus A prüft ein entsprechendes Intervall p nun für P1. Der Algorithmus A baut dazu einen Baum auf, in welchem das Intervall immer wieder erneut geprüft wird. Für jedes Intervall werden alle Simple Bounds von P1 geprüft. Dabei werden weniger proportional zu log(#p) viele Halbierungen des Intervalls vorgenommen, sodass die Anzahl an Aufteilungen in der Aufwandsklasse O(log(n)) liegt. Es muss weiterhin im Worst Case jeder Wert geprüft werden, sodass der Aufwand in O(nlog(n)) liegt. Dies muss für jeden Simple Bound durchgeführt werden, sodass der Aufwand im Worst Case in O(n²log(n)) liegt.

Damit liegt dieses Problem in NP und CS ist NP-schwer.

Damit ist gezeigt, dass CS sowohl in NP als auch in NP-schwer liegt und damit NP-vollständig ist.

1d) i) Funktioniert Algorithmus A korrekt auf dem neu definierten P, bzw. ist das Ergebnis korrekt?

Der Algorithmus A funktioniert korrekt auf P. Die Funktionsweise des Algorithmus ist weiterhin, dass die Intervallgrenzen p auf true, false oder inconclusive geprüft werden. Da hierbei nicht jeder einzelne Wert im Intervall geprüft wird, ist der Umstand, dass in diesem Intervall unendlich viele Zahlen sind, für die Prüfung des Intervalls nicht relevant. Unter der Voraussetzung, dass der Algorithmus nicht auf einem ressourcenlimiterten Endgerät ausgeführt wird, kann der Algorithmus auch immer ein Ergebnis finden, wenn er unendlich lange läuft. Wenn der Algorithmus ein Ergebnis findet, ist dieses korrekt, da wie bereits erwähnt lediglich die Intervallgrenzen geprüft werden.

1d) ii) a) Terminiert der Algorithmus A auf P?

Der Algorithmus A terminiert nicht zwingend auf P. Angenommen es gibt ein Intervall p, welches von A geprüft wird und dieses Intervall ist inconclusive. Dann gibt es mindestens einen Wert in p, für welchen P lösbar ist. Wenn in diesem Intervall nur ein Wert existiert, für den P lösbar ist, dann müsste dieser exakt auf einer Intervallgrenze liegen. Dies ist dadurch gegeben, dass die Simple Bounds eine > oder >= Relation enthalten.

In diesem Falle würde Algorithmus A das Intervall p unendlich oft teilen und sich dem Lösungswert beliebig nah annähern, diesen jedoch nie erreichen. Dadurch würde der Algorithmus nicht terminieren.

1d) ii) b) Wie lässt sich Algorithmus A anpassen?

Eine Möglichkeit den Algorithmus A anzupassen ist diesen terminieren zu lassen. Wenn der Algorithmus A ein Intervall p als inconclusive klassifiziert, wird dieses in zwei Intervall aufgeteilt. Wenn über die Entscheidungsebenen nun eine gewisse Anzahl an Durchläufen immer wieder eine Intervallgrenze identisch bleibt, steigt damit die Wahrscheinlichkeit, dass der Wechsel von wahren und falschen Werten und vice versa genau auf der Intervallgrenze liegt. Wenn dies nun beispielweise 1337 Mal am Stück passiert, teilt der Algorithmus das Intervall nicht erneut auf, sondern führt eine Punktprüfung auf die Intervallgrenzen durch und gibt den wahren Wert zurück. Auf diese Weise wird immer eine Lösung gefunden. Dies ist nicht zwangsläufig die einzige Lösung, jedoch ist dies auch nicht der Anspruch des Algorithmus A.

1e) Welche Heuristiken ergeben für CSP am meisten Sinn unserer Meinung nach?

* Variablenauswahl: Es wird zunächst die Variable mit der kleinsten Domain und dem häufigsten Vorkommen in der Anzahl der Constraints gewählt. Auf diese Weise werden zum einen Deadends schneller gefunden und zum anderen kann die erste Variable dadurch schnellstmöglich minimiert werden, sodass für die Prüfung für Variablen mit größerer Domain, die Iterationen auf andere Variablen geringer sind.
* Variablenauswahl: Wenn ein Simple Constraint evaluiert wird, welcher inconclusive ist, dann ist es sinnvoll die Variable, die an den entsprechenden inconclusive Simple Bounds am häufigsten beteiligt ist, als erstes auszuwählen. Bei gleichem Aufkommen mehrerer Variablen kann die obere Möglichkeit gewählt werden.
* Intervallentscheidung: Bevor sich für ein Intervall entschieden wird, werden zunächst beide Intervalle geprüft. Auf diese Weise kann ein wahres Intervall schneller gefunden werden.

1f) Prinzipiell haben CDCL und Algorithmus B zwei verschiedene Vorgehensweisen. Unter B werden alle Constraints separat betrachtet und unter CDCL würden zunächst eine Variable und alle zugehörigen Simple Bounds betrachtet werden. So ist ein signifikanter Unterschied, dass B Constraints-getrieben agiert und CDCL Variablen-getrieben. Eine komplette Integration von CDCL in B ist somit nicht möglich. Jedoch sind durchaus Hybridlösungen denkbar oder Teilintegrationen von CDCL in B.

Eine Möglichkeit wäre es, zunächst in Schritt 1 alle Constraints für eine Intervallabaschätzung zu prüfen. Für alle Constraints, die inconclusive oder false sind, wird CDCL angewendet, indem nur die Variablen betrachtet werden, welche in den entsprechenden Constraints vorkommen. Sobald es hier zu einem Conflict kommt, wird die Ursache auf den vorherigen Entscheidungsleveln nach CDCL als zusätzliche Conflict Clause gespeichert. Je nachdem ob zuerst ein falses oder ein inconclusive Constraint gefunden wird, wird mit Schritt 2 oder 3 weiterverfahren.

## Praktischer Teil

2a)

Das CSP besteht aus zwei ArrayLists. Eine vom Typ Variable und die andere vom Typ Simple Constraint. Zudem gibt es einen Stack vom Typ Variable für die zunächst nicht verwendete Hälfte der Intervalle. Eine Variable besteht aus drei Integern. Die untere und obere Grenze der Domain und eine Position, zur Identifikation. Ein SimpleConstraint besteht aus einer ArrayList von SimpleBounds. Jeder SimpleBound besteht aus zwei Variablen x und y sowie zwei Integern, die die Konstanten auf der linken beziehungsweise rechten Seite des Simple Bounds darstellen.

2b)

Der implementierte Parser ist in der Lage Eingabedateien zu lesen, die dem formalen Template entsprechen. Diese warden in die zuvor beschriebenen Datenstrukturen umgewandelt. Auch Constraints der Form 10 >= 4 werden erkannt und umgewandelt.

Dies geschieht, indem die Simple Constraints in Simple Bounds geteilt werden und alle unrelevanten Zeichen entfernt werden. Anschließend werden auf die gleiche Art und Weise die einzelnen Variablen gesplittet bis lediglich die Identifier übrig sind, diese warden dann in die internen Datenstrukturen übernommen.

2c)

Im Folgenden werden die Implementierungen der einzelnen Schritte des Algorithmus A beschrieben. Der erste Schritt liest den aktuellen SimpleBound aus. Die Domain der Variablen und die Konstanten werden ausgelesen. Mit den Werten werden die beiden Gleichungen aus Definition 4 ausgewertet und entsprechende Hilfvariablen gesetzt. Daraufhin wird der nächste Schritt des Algorithmus entschieden und ausgeführt.

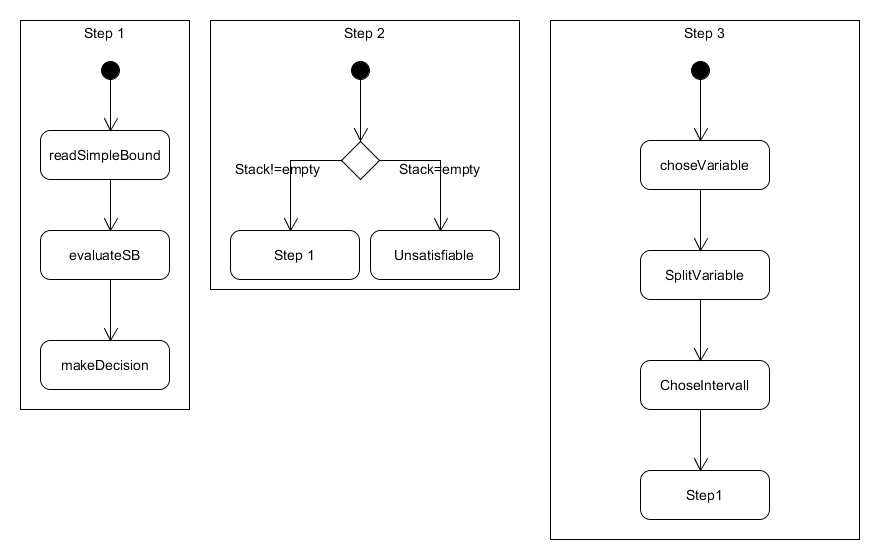
Der Zweite Schritt prüft, ob der Stack mit den zweiten Invervallhälften leer ist. Ist er leer terminiert der Algorithmus, sonst wird die Variable auf dem Stack genutzt.

Im Dritten Schritt wird zuerst eine Variable ausgewählt, deren Bounds verkleinert werden sollen. In unserem Falle wird die erste Mögliche Variable aus der Variablenliste genommen. Die Bounds der gewählten Variablen werden daraufhin halbiert. Die untere Hälfte wird als neue Bounds benutzt und die obere Hälfte wird auf den Stack gelegt. Anschließend wird mit Alogrithmus A fortgefahren.

2d)

Algorithmus B wurde wie Algorithmus A implementiert. Es wurde jedoch der erste Schritt erweitert und eine Methode zur Deduction implementiert.

Der erste Schritt speichert nun für jeden Constraint, ob bisher nur ein Inconclusive SimpleBound gefunden wurde. Ist dies beim letzten SimpleBound noch der Fall, so wird die Deduction Methode aufgerufen. Diese berechnet neue Grenzen und prüft, ob diese Grenzen das Intervall verkleinern. Ist dies der Fall warden verkleinernde Grenzen verwendet. Bei neuen Grenzen wird Schritt 1 ausführt, ansonsten Schritt 3.



2e)

In diesem Aufgabenteil wurden Zeitmessungen der Algorithmen A und B zur Laufzeit vorgenommen. Dies wurde durchgeführt, indem eine Zeitmessung zu Anfang der Algorithmen und eine zu Ende der Algorithmen durchgeführt wurde. Anschließend wurde die Differenz gebildet. Daraus ergaben sich folgende Zeiten:

|  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- |
| Beispielnr. | Runtimes Algorithmus A | Durchschnitt Algorithmus A | Runtimes Algorithmus B | Durchschnitt Algorithmus B | Speedup |
| 1 CSP2 | **77124, 73018, 100877** | 83673 | 81522, 88853, 76830 | 82401 | 1,54% |
| 2 CSPB1 | **167150, 156300, 155420** | 159623 | 159232, 159819, 159526 | 159525 | 0,06% |
| 3 test | **162457, 150434, 143691** | 152194 | 150434, 174774, 147209 | 157472 | -3,46% |
| 4 Example1 | 1170635**,** 1107879**,** 1138964 | 1139159 | 657748, 485614, 623732 | 589031 | 93,39% |
| 5 | 183278**,** 190316**,** 185624 | 186406 | 208498, 204392, 209963 | 207617 | -10,32% |
| 6 | 1200252**,** 1295557**,** 1293211 | 1263006 | 594994, 629010, 803785 | 675929 | 86,85% |
| 7 | 526082**,** 669478**,**  502035 | 621295 | 597927, 631650, 641033 | 623536 | -0,37% |
| 8 | 582385**,** 545436**,** 667426 | 598415 | 438695, 436056, 448079 | 440943 | 35,71% |
| 9 | 484147**,** 452183**,** 458342 | 464890 | 471245, 439574, 453650 | 454823 | 2,21% |
| 10 | 248672**,** 257176**,** 231370 | 245406 | 182692, 172428, 188556 | 181225 | 35,41% |

|  |  |  |
| --- | --- | --- |
| Beispielnr. | Runtimes Algorithmus C | Durchschnitt Algorithmus C |
| 1 CSP2 | **410250, 498223, 481508** | 463327 |
| 2 CSPB1 | **528721, 535759, 594115** | 552865 |
| 3 test | **517284, 487960, 514939** | 506727 |
| 4 Example1 | **982371, 897330, 971228** | 950309 |
| 5 | **786190, 705548, 671531** | 721089 |
| 6 | **1114331, 1171514, 1272977** | 1186274 |
| 7 | **1073277, 1152160, 1159785** | 1128407 |
| 8 | **1013455, 847772, 958325** | 9398850 |
| 9 | **1050697, 995274, 919030** | 988333 |
| 10 | **713465, 643380, 601739** | 652861 |

2f)

In diesem Aufgabenteil wurde die Lazy Clause Evaluation implementiert. Spezielle Decision Heuristics wurden nicht implementiert. Die gemessenen Zeiten zeigen, dass die alleinige Implementation einer Lazy Clause Evaluation für die verwendeten CSPs (siehe Anhang) keine Beschleunigung zur Folge hat. Dies kann daran liegen, dass die einzelnen CSPs keine Lazy Clauses enthalten.

# Fazit

Das Entscheiden und Lösen von Problemen sind essentielle Aufgaben der theoretischen Informatik, die damit das Fundament für weitere Teilbereiche der Informatik bildet. In diesem Kontext ist das Lösen von Constraint Satisfaction Problems (CSP) eine sehr wichtige und gut erforschte Herausforderung. Die vielfältigen Anwendugsbereiche im Gebiet künstliche Intelligenz und die unterschiedlichen Ansätze zum Lösen von CSPs tragen dazu bei, dass CSPs eine gute Möglichkeit bieten, um sich die Fähigkeit des Lösens von Problemen zu vertiefen.

Der Rahmen dieses Projektes hat uns die Möglichkeit eröffnet an einem konkreten Teilgebiet der Informatik unsere Fähigkeiten im Bereich logischen Denken und Lösen von Problemen zu verbessern. Der theoretische Teil dieser Arbeit hat uns dazu veranlasst uns gemeinsam als Gruppe in ein uns bislang fremdes Themengebiet einzuarbeiten. Dabei haben wir unter anderem mit Aufwandsklassen beschäftigt und in diesem Kontext die Aufwandsklassen NP-schwer, NP und P kennengelernt. Die Recherche in diesem Kontext hat uns ein abstraktes Verständnis des P/NP-Problems gegeben, das dies notwendig war um die Aufwandsklasse NP-schwer zu verstehen und ein Problem in dieser formal einordnen zu können.

Weitere Erkenntnisse bestehen in den Arbeitsweisen der einzelnen Algorithmen, sowohl auf theoretischer als auch auf praktischer Ebene. Das Auseinandersetzen mit Determinismus, Completeness und Soundness führten zu einem hohen theoretischen Verständnis von Algorithmus A. Anschließende Implementationen dieser Algorithmen haben nicht nur die Programmierkenntnisse im Allgemeinen aufgewertet, sondern auch das Debuggen komplexer rekursiver Funktionen.

Insgesamt hat sich das theoretische Verständis und die Fähigkeit Probleme in bislang unbekannten Teilbereichen zu Lösen bei allen Gruppenmitgliedern verstärkt ausgeprägt.

# Probleme

1. In der Beschreibung des Algorithus A stand “wähle eine Variable x”. Die Wahl der Variable war nicht weiter beschrieben, weshalb die Implementation unklar war.

Lösung: Es wurde keine Heuristik für die Variablenauswahl genutzt. Es wurde vielmehr eine zufällige Variable ausgewählt (nach der Reihenfolge in der Variablendeklaration).

2. Die Implementierung der Lazy Clause Evaluation passte nicht mit der vorhandenen Datenstruktur zusammen. Es konnten mit den derzeitigen Hilfsvariablen nicht die nötigen Entscheidungen getroffen werden.

Lösung: Es wurden neue Hilfsstruktur eingeführt. Die Evaluation der einzelnen Simple Constraints wurde in einer Liste gespeichert, um die Entscheidungen treffen zu können. Demnach wurde für jeden Simple Constraint gespeichert, ob der Constraint true, false, Inconclusive oder noch nicht ausgewertet ist.

3. NP-Vollständig und NP-Schwer bedurften einer Auffrischung dieser Thematik.

Lösung: Es war notwendig die bis dato in Vergessenheit geratenen Grundkenntnisse bezüglich der Thematik “Aufwandsklassen und Abschätzung” neu zu behandeln.

# Zeiterfassung

|  |  |  |  |
| --- | --- | --- | --- |
| Aufgabenteil | Beteiligte | Gesamte Dauer | Anmerkungen |
| 1a) | Svenja, Jörn, Alex | 1 Stunde | Die Aufgaben 1a) - 1c) wurden in einer einzelnen Sitzung am 16.07 von 20:00 – 23:30 Uhr gemeinsam bearbeitet |
| 1b) | Svenja, Jörn, Alex | 1 Stunde |
| 1c) | Svenja, Jörn, Alex | 1,5 Stunden |
| 1d) | Svenja, Jörn, Alex | 1,5 Stunden | Die Aufgaben 1d), 1e) und 2a) wurden in der darauffolgenden Sitzung am 20.07 von 13:15 – 16:00 Uhr gemeinsam bearbeitet.  In dieser Sitzung wurde festgelegt, wer im praktischen Teil was implementiert. |
| 1e) | Svenja, Jörn, Alex | 0,75 Stunden |
| 2a) | Svenja, Jörn, Alex | 0,5 Stunden |
| 2b) | Alex | ~3 Stunden | Der Solver ist chronologisch als nächsten entstanden. Die Zeit wurde nicht getrackt. |
| 2c) | Svenja | ~3 Stunden | Der Algorithmus A ist im Anschluss implementiert worden. |
| 2b) | Alex, Svenja, Jörn | 3 Stunden | Am 25.07 von 16:00 – 19:00 Uhr wurde Bugs in Algorithmus A behoben und dieser fertiggestellt. |
| 2d) | Jörn | 3 Stunden | Algorithmus B wurde fertiggestellt nachdem Algorithmus A vollständig funktionierte. |
| 2d) | Alex, Svenja, Jörn | 1,5 Stunden | Am 01.08 von 16:00 – 19:00 Uhr haben wir Bugs in Algorithmus B behoben und diesen vervollständigt. Daraufhin wurden die Zeitmessungen durchgeführt. Und Aufgabe 1f) bearbeitet. Die einzelnen Aufgaben geschahen teilweise parallel. |
| 2e) | Alex, Svenja, Jörn | 0,5 Stunden |
| 1f) | Alex, Svenja, Jörn | 1 Stunde |
| 2f) | Alex, Svenja, Jörn | 3 Stunden | Am 08.08 wurde in gemeinsamer Arbeit erledigt und vollständig funktionsfähig gemacht. |
| Einleitung und Problemstellung | Svenja | 1 Stunde |  |
| Fazit | Alex | 1 Stunde |  |
| CSPs | Jörn | 1 Stunde |  |
| Dokumentation der Implementation | Alex, Jörn, Svenja | 0,5 Stunden | Diese Aufgaben wurden parallel zueinander erledigt am 14.08 von 17:45 – 18:45 Uhr. |
| Aufbereitung des Reports | Alex, Jörn, Svenja | 0,5 Stunden |
| Zeiterfassungstabelle | Jörn, Svenja, Alex | 0,5 Stunden |
| Probleme und Herausforderungen | Jörn, Svenja, Alex | 0,5 Stunden |
|  |  |  |  |
|  |  |  |  |