KU LEUVEN

FACULTEIT INDUSTRIËLE INGENIEURSWETENSCHAPPEN

CAMPUS DE NAYER SINT-KATELIJNE-WAVER

Een interactieve kennis-banktoepassing voor groepsverdelingen

Simon VANDEVELDE

Promotor: Herman Crauwels

Co-promotor: Kylian Van Dessel

Masterproef ingediend tot het behalen van de graad van master of Science in de industriële wetenschappen: Industrieel Ingenieur E/ICT

Academiejaar 2018 - 2019

Voorwoord

Ik zou graag iedereen willen bedanken die mij gesteund en geholpen heeft bij het maken van deze masterproef.

Om te beginnen wil ik eerst mijn promotoren Herman Crauwels en Kylian Van Dessel hartelijk bedanken voor hun uitstekende begeleiding en feedback. Zij hebben steeds uitgebreid de tijd genomen om tips, hulp en andere input te geven, waarvoor dank.

Daarnaast zou ik graag mijn vriendin, familie en vrienden willen bedanken voor alle steun.

Tot slot wil ik ook iedereen bedanken die ik hier niet expliciet vermeld heb.

Simon Vandevelde, Meise 2019

Abstract

In deze thesis is onderzocht in welke mate het IDP-systeem, ontwikkeld door KU Leuven, werkt in een interactieve context. Hiervoor werd het IDP-systeem ingebouwd in een interactieve toepassing om groepsindelingen van eerstefase bachelorstudenten te vormen. Het IDP-systeem is een implementatie van het kennisbankparadigma (KBP), wat inhoudt dat het informatie opslaat in een kennisbank, en deze informatie kan gebruiken om verschillende problemen op te lossen.

Dit onderzoek is in twee stappen gebeurd. Als eerste stap is bekeken hoe men best een groepsindeling kan implementeren in IDP. De groepsindeling is in twee stukken opgesplitst: de ruwe initiële en de verfijnende incrementele indeling. De initiële indeling zal proberen om een eerste indeling te maken, aan de hand van de woonplaats en middelbare school van de studenten. Vervolgens kunnen studentenvoorkeuren opgegeven worden, en zal het systeem de indeling hermaken zodat aan zo veel mogelijk voorkeuren voldaan zijn. Elke voorkeur heeft een eigen instelbare prioriteit.

Als tweede stap is hier rond een toepassing gecreëerd. Deze toepassing maakt gebruik van Qt5 om een Grafische User Interface (GUI) weer te geven, en de Pyidp3 API om te communiceren met het IDP-systeem. Pyidp3 is in deze thesis uitgewerkt en is een verderzetting van de reeds bestaande Pyidp API. Het IDP-systeem vormt de *backend* van de toepassing. Het IDP-systeem zorgt voor de intelligentie, en de GUI en Pyidp3 zorgen voor een gebruiksvriendelijke interface. Bij het maken van een initiële indeling in de toepassing is kan de gebruiker verscheiden parameters wijzigen, zoals de grootte van de groepen en de hoeveelheid groepen. Bij de incrementele indeling kunnen voorkeuren ingegeven worden, prioriteiten aangemaakt en ingesteld worden, meerdere oplossingen laten berekend worden en kan het systeem het verschil tonen tussen de huidige groepsindeling en een nieuwe groepsindeling.

Keywords: IDP, Kennisbank, Groepsindeling, Interactiviteit

Abstract

The aim of this thesis is to research how well the IDP system, developed at KU Leuven, functions in an interactive context. The IDP system is an implementation of the knowledge base paradigm (KBP), meaning that it stores its information in a knowledge base, to which it can apply inference methods to solve problems. To test the functionality in an interactive context, an interactive application for group assignment of students was developed.

The research is divided in two steps. As a first step, we took a closer look at how to implement a group assignment algorithm using the IDP system. The implementation consists of two parts: the coarse initial group distribution based on residence and former high school, and the refined incremental distribution which takes student preferences into account. Each preference has its own adjustable priority.

As a second step, we developed an application around this IDP implementation. The application uses Qt5 to construct a Graphical User Interface (GUI) and the Pyidp3 API to communicate with the IDP system. In this way the IDP system forms the application's backend. The use of the IDP system ensures the intelligence of our application and the GUI and Pyidp3 provide a userfriendly interface. Before making an initial group layout, it is possible to change the parameters of the system, such as the group size and the amount of groups. For the additional assignment, it is possible to add preferences with a priority, to make one's own priorities, to have multiple next solutions be calculated and have the IDP system show the differences between the current and a new layout.

Keywords: IDP, Knowledge base, Group assignment, Interactivity

English Summary

Introduction

This master's thesis aimed to research how well the IDP system, developed at KU Leuven, performs in an interactive environment. To test this in a practical manner, an interactive application for the group assignment of students is designed using the IDP system as a backend.

The IDP system is an implementation of the Knowledge Base Paradigm (KBS). It is a system which stores all its information in a knowledge base and applies powerful inference tasks to the knowledge to solve different problems. The system uses the FO(·) language, which is basic first order logic but enriched with types, aggregates, inductive definitions, partial functions and more [Bruynooghe et al. (2015)].

The application is responsible for the group assignment of students. At the KU Leuven campus De Nayer, each year around 150 students sign up to enroll in higher studies. These students are divided into groups. Instead of assigning them randomly or in alphabetical order to these groups, the campus takes into the students' background and preferences in consideration so the students feel comfortable in their new environment. Currently, this process is done manually and takes a great deal of time.

The aim of this thesis can be split up into two goals.

- Developing an intelligent, knowledgebased solution for the group assignment.
- Integrating this solution into an interactive application.

Literature study

Before developing a KBS, different KB techniques are taken into consideration. We describe two State-of-the-Art systems in this research area, Answer Set Programming (ASP) and the IDP system. Afterwards some options for interfacing with the IDP system are looked at and compared.

Answer Set Programming is a form of declarative programming. It reduces problems to atomic formulas and relations, after which an answer set solver can be used to find a solution to the

problem [Lifschitz (2008)]. Examples of such solvers are clasp[1] and DLV[2].

The IDP system is an alternative to ASP. The IDP system implements the Knowledge Base Paradigm, which consists of two parts: a knowledge representation language and powerful inference methods.

A knowledge representation language is a natural and readable way of programming. For the IDP system, this language is called the IDP language. It allows us to declare rules, after which we let the IDP system generate (a) model(s) that satisfy the rules. The syntax consists of the FO(·) language. A program written for the IDP system is divided into four or five parts.

- Vocabulary: contains all the needed symbols, predicates and functions needed to construct the rules.
- Structure: where we can assign values to types, relations or functions.
- Theory: the part where we define the constraints which the models have to satisfy. These constraints are formed using FO(·).
- Term: an optional part, where we can declare a term, i.e. a variable or a formula, whose value needs to be minimized.
- Procedure: the only imperative part of the IDP system. Here we program what inference tasks we want to apply to the system. This part is written in Lua.

Interface To use the IDP system as a backend in an application, we need to be able to communicate with it from an imperative programming language. A first option is the Lua language, as it is already available to us in the IDP system. The downside of Lua however, is that there are no recent graphical user interface (GUI) frameworks. A second option is the Java Application Programming Interface (API) made by Calus (2011). This is written for the IDP2 system, which is currently outdated. The best and final option is the Python 2 API by Vennekens (2015). This API has to be ported to Python 3, as the GUI framework we want to use is only available in Python 3.

IDP

In this chapter, our solution of a group assignment implementation is given. We start by splitting the assignment of groups into two parts: an initial assignment and an incremental assignment. The idea here is that a user could make an initial group assignment based on residence and publish this. Students can then give preferences, after which the incremental group assignment can be made by the IDP-system to try and satisfy as much of them as possible. The process of gathering preferences and recalculating the group layout can be done multiple times.

[1] https://potassco.org/clasp/
[2] http://www.dlvsystem.com/

Initial assignment The initial assignment algorithm takes three variables into account. The first variable is the high school at which the students studied at. The algorithm tries to assign students who studied at the same high school to the same group. Concurrently, it tries to assign the rest of the students so that they all live as close together as possible. This is done by looking at their postal code. For instance, 1860 (Meise) neighbors the town of 1840 (Londerzeel). However, 1980 (Eppegem) and 2000 (Antwerp) do not neighbor each other. The reason for this is their zone number, which is the first number of their postal code. Because of the difference in zones, we split up the postal code into the zone number and the rest of the postal code, which form our other two variables. The IDP system will try to assign students from the same zone and same living area into the same group.

Incremental assignment The incremental assignment will take the preferences of students into account when forming the group layout. We consider three kinds of preferences: wanting to be in the same group as another student, not wanting to be in the same group as another student and wanting to be in a specific group, for schedule reasons. The user of the application can decide what priority a preference should have when adding it to the application. For instance, a student with a sports statute who needs to be in a specific group because the schedule is suited best for their sport program, has a high priority.

Pyidp3

Pyidp serves as an interface between Python 2 and the IDP system. It enables a Python programmer to experience the power of the IDP system, without having to learn its full syntax. The interface is able to translate a pythonic way of data representation to a representation for the IDP system and vice versa.

For this work however, we have to port the Python 2 interface to Python 3. In addition, some features of the IDP system currently not provided by the API are required, such as model checking and cost minimization. Because of this, we updated and expanded the Pyidp interface to a Python 3 interface called Pyidp3.

Application

It is important for our application to be interactive. The intelligence of our application is situated in the backend, which is the IDP system. The interaction with the IDP system is made possible by using a GUI in Python and the Pyidp3 API. The application is built according to the Model-View-Controller design pattern.

The model consists of the representation of information and the necessary manipulations on this information. Two main parts of the model are a class to read and write CSV files and a class to read from and write to the IDP system using Pyidp3.

The view consists of multiple tabs, which serve to display information and allow for a userfriendly

way of input. The userfriendlyness is important, as the system has to be operable by a layman.

The controller is responsible for data exchange between IDP and GUI, and managing interaction aspects. A user can make an initial group layout by loading in a CSV file containing a list of students after which they can customize a set of parameters: the amount of groups, minimum and maximum group size, the priority of the three variables in the initial assignment and the maximum calculation time for the IDP system. After having made the initial assignment, the user can start inputting student preferences. Whenever it is time for a recalculation, the user can have the application make an incremental assignment. Here the application allows the user to specify how many new solutions they want to see. For each new solution, a comparison is made with the current solution. The differences are listed and whether the preferences are satisfied or not. The user can then pick the new group layout.

Evaluation

The initial group assignment is slow and its methodology is shallow. The IDP system is not made to perform calculation-intensive tasks and our implementation, containing a fair amount of mathematical calculations, loses the advantages of the IDP system. However, as this is only an initial solution used to build on to, it does not need to be perfect and is sufficient.

The incremental assignment however succeeds in recalculating a group layout within 10 seconds, as long as there are not more than 100 preferences given. This is a good result considering this limit on the number of preferences is normally not exceeded in a realistic setting.

Conclusion

The research question of this master's thesis is as follows: "How well can the IDP knowledge base system be deployed in an interactive application?". We divided this research question further into two goals. Namely, (1) the development of an intelligent knowledgebased solution for the group assignment, and (2) the integration of this solution into an interactive application.

To satisfy the first goal, an implementation for group assignment using the IDP system was made. The implementation consists of two parts: a coarse initial assignment and a refined incremental assignment. The algorithm for our initial assignment has to run fairly long to find a good solution. The incremental assignment however finds the solution within an acceptable time period and correctly takes the students' preferences into account.

For the second goal, an interactive application is developed which uses our IDP implementations for the assignment as a backend. The interactivity is achieved by using Python and the Pyidp3 interface. The application allows for several interactive features, including: adjusting group sizes, adjusting the weights of the initial implementation, adding preferences with their own priority, presenting multiple solutions and allowing for the user to pick one, etc.

We can conclude that the application is sufficiently interactive and that the IDP KBS was successfully deployed in an interactive application.

Inhoudsopgave

Lijst van afkortingen

AI	Artificiële Intelligentie
API	Application Programming Interface
ASP	Answer Set Programming
CL	Collaborative Learning
CSV	comma-separated values
FO	First Order
FO(·)	uitgebreide First Order logic
GA	Genetisch Algoritme
GUI	Graphical User Interface
IDP	Imperative-Declarative Programming
KBP	Kennisbankparadigma
KBS	Kennisbanksysteem
OO	Operationeel Onderzoek

Lijst van figuren

Lijst van tabellen

Hoofdstuk 1

Inleiding

1.1 Situering

In dit werk wordt onderzocht in hoeverre het haalbaar is om een interactieve applicatie te ontwikkelen die gebruik maakt van het IDP-kennisbanksysteem. Om dit praktisch te testen wordt een applicatie voor groepsindeling van studenten ontworpen. Hierbij speelt de interactiviteit van deze applicatie een grote rol.

Het IDP-systeem is een implementatie van het Kennisbank Paradigma (KBP). Een implementatie van KBP is een systeem dat zijn informatie in een kennisbank houdt, en gebruik maakt van allerlei inferentiemethoden om deze kennis toe te passen op problemen. De IDP-kennisbank steunt op de FO(·) taal. Dat houdt in dat het eerste-orde logica (First Order) gebruikt, aangevuld met types, aggregaten, inductieve definities, partieelfuncties, en meer (Bruynooghe et al. (2015)). IDP is ontworpen door de Knowledge Representation and Reasoning (KRR) groep, welke deel is van de Declaratieve Talen en Artificiële Intelligentie (DTAI) onderzoeksgroep[1] van KU Leuven.

De applicatie die ontworpen wordt, zal het maken van de groepsindeling van de eerstejaarsstudenten op KU Leuven campus De Nayer trachten te vereenvoudigen. Momenteel worden ieder jaar alle studenten manueel ingedeeld in groepen. In plaats van deze groepen op basis van alfabetische volgorde of moment van inschrijven te vormen, wordt geprobeerd om de studenten op een student vriendelijke manier aan een groep toe te wijzen.

In het verleden is dit al volgens twee verschillende methoden gebeurd.

De eerste methode gaat de studenten in de eerste plaats in te delen op basis van woonplaats. Groepsgenoten die bij elkaar in de buurt wonen, gaan namelijk (hopelijk) sneller geneigd zijn tot carpooling. Vervolgens wordt gekeken naar de voorkeuren van de student (bij wie ze graag in de groep zouden willen zitten; en bij wie zeker niet), de verdeling man/vrouw en de verdeling van zittenblijvers/nieuwe studenten. Hierbij wordt rekening gehouden met een uniforme verdeling van het aantal studenten per groep.

De tweede methode werkt gelijkaardig, maar focust eerst op de middelbare school van de studen-

[1] https://dtai.cs.kuleuven.be/

ten en vervolgens op de woonplaats. Na deze initiële indeling kunnen de studenten voorkeuren opgeven, en worden deze zo goed mogelijk ingerekend. Deze voorkeuren zijn gelijk aan de eerder beschreven voorkeuren.

1.2 Probleemstelling

Eén van de belangrijkere problemen in dit onderzoek is de mogelijkheid tot interactiviteit van IDP. Met een naïeve IDP oplossing kost het indelen van een realistisch aantal studenten relatief veel rekenkracht en tijd, wat de interactiviteit niet ten goede komt. Verder is het niet evident om een applicatie op te bouwen rond het IDP-systeem, omdat er telkens een vertaling moet gebeuren van imperatief naar declaratief programmeren, en omgekeerd.

Een ander probleem in het onderzoek is het indelen in groepen. Zoals eerder vermeld is dit momenteel een manueel proces, dat bijgevolg veel tijd kost. Maar ook is de heuristiek van het indelen afhankelijk van persoon tot persoon. Het is geen eerlijke manier van indelen, aangezien er vrij veel ruimte is voor arbitraire keuzes. Dit kan in sommige gevallen leiden tot grote verschillen in indelingen. Een voorbeeld waarbij deze arbitraire keuzes merkbaar zijn, is wanneer student A bij student B wil, student B op zijn beurt bij student C wil, maar student C niet bij student A wil. De oplossing die hier gekozen wordt, is afhankelijk van persoon tot persoon.

1.3 Doelstelling

De doelstelling van het onderzoek is tweevoudig:

- het ontwikkelen van een intelligente kennisbank-oplossing voor het indelen van studenten;
- deze oplossing interactief verwerken in een applicatie die door een eindgebruiker gebruikt kan worden.

De uiteindelijke applicatie zal toelaten om een CSV bestand van informatie over studenten in te laden. Het indelen in groepen zal uit twee stappen bestaan. Als eerste stap zal een initiële indeling gemaakt worden. Deze initiële indeling zal op basis van de woonplaats van de student gebeuren. Hoewel de achterliggende gedachte het stimuleren van carpoolen is, zou het perfect oplossen van een volledig carpoolprobleem ons te ver leiden en behoort dit niet tot het onderzoek.

Als tweede stap kan de eindgebruiker voorkeuren van studenten ingeven en het IDP-systeem de indeling laten herrekenen. Telkens een indeling berekend wordt, zal de applicatie weergeven aan welke beperkingen voldaan zijn, en aan welke niet.

Tijdens deze twee stappen krijgt de gebruiker ook de mogelijkheid om het aantal groepen en het minimum en maximum aantal studenten per groep op te leggen.

In het volledige systeem zullen volgende voorkeuren en beperkingen beschikbaar zijn:

- een student die samen in een groep wil met een specifieke andere student;

- een student die zeker niet samen wil met een specifieke andere student;
- een student die in een bepaalde groep wil (omdat het uurrooster beter uitkomt met bijvoorbeeld zijn/haar sportschema);
- het aantal gewenste groepen;
- maximum en minimum aantal studenten per groep.

Eens de gebruiker tevreden is met het bekomen resultaat, kan deze informatie weer weggeschreven worden naar een nieuw CSV bestand dat dan gebruikt kan worden om bijvoorbeeld op een website te tonen.

De implementatie van dit werk is terug te brengen in drie grote onderwerpen.

1. Het IDP-systeem: Het IDP-systeem vormt de intelligentie van de totale implementatie en zal dus instaan voor het de effectieve indeling.
2. De interface: Voor communicatie tussen het IDP-systeem en de applicatie er rond, is nood aan een interface om te fungeren als communicatiekanaal. Deze interface moet data kunnen omzetten naar een leesbaar formaat voor het IDP-systeem, en de output van het IDP-systeem kunnen interpreteren.
3. De eindapplicatie: De eindapplicatie zorgt dat er externe data kan ingelezen worden, dat deze data wordt aangebracht aan het IDP-systeem en dat het resultaat vervolgens weergegeven wordt. Hierbij speelt interactiviteit een grote rol.

Concreet willen we onderzoeken of het mogelijk is een heuristiek te ontwikkelen om studenten onder te verdelen in groepen aan de hand van het IDP-kennisbanksysteem, en of het IDP-kennisbanksysteem in staat is om te werken in een interactieve omgeving door hier een applicatie rond te bouwen.

Hoofdstuk 2

Literatuurstudie

In dit hoofdstuk wordt een beschrijving gegeven van het IDP-systeem (gebruikt in dit werk) en aanverwante technologieën. Verder worden er heuristieken besproken voor het indelen van groepen, en worden de verschillende mogelijkheden om te communiceren met het IDP-systeem bekeken en vergeleken.

2.1 Answer Set Programming

Answer Set Programming (ASP) is een vorm van declaratief programmeren. Bij ASP worden problemen gereduceerd tot atomaire formules en relaties, waarna vervolgens een *answer set solver* een oplossing tot het probleem kan zoeken. Een *answer set solver* is een programma dat alle *answer sets* van een ASP programma zal zoeken, door modellen te genereren en na te gaan of de modellen voldoen aan de formules en relaties (Lifschitz (2008)). Een voorbeeld van zo'n *answer set solver* is clasp[1].

Een ASP programma heeft geen verplichte, vaste structuur, maar wel een gesuggereerde methodologie. Deze heet de *"Generate, Define, Test"*-methodologie. Dit houdt in dat het programma wordt opgedeeld in drie delen:

1. Generate: het genereren van een set van mogelijke oplossingen aan de hand van keuzeregels;
2. Define: definiëren van hulppredicaten (vaak beperkingen);
3. Test: elimineren van alle 'slechte' oplossingen.

Meer informatie hierover kan gevonden worden in de paper van Lifschitz (2008).

De basis van de ASP syntax bestaat uit vier soorten beperkingen/regels (Schaub (2018)).

- Keuzeregels: zorgen voor een keuze over een subset van elementen.

[1] `https://potassco.org/clasp/`

- Integriteitsbeperkingen: om ongewenste oplossingen uit de oplossingsset te elimineren.
- Kardinaliteitsregels: vastleggen van een aantal uit een lijst van specifieke elementen dat moet voorkomen in de oplossing.
- Gewichtsregels: gelijkaardig aan kardinaliteitsregels, maar met gewogen elementen (niet elk element weegt even zwaar door).

ASP laat ons ook toe om *facts* toe te voegen. Dit zijn elementen die altijd `true` moeten zijn.

Stel bijvoorbeeld dat een groep studenten graag een feest zouden willen organiseren. Voor dit feestje zullen ze naar de winkel moeten gaan, waar ze verschillende dingen kunnen kopen. Ze moeten wel zorgen dat ze voldoende dingen kopen om ervoor te zorgen dat het een goed feest is.

We beginnen met het feit dat de studenten naar de winkel moeten gaan. Dit is te schrijven als:

```
in(winkel).
```

Deze regel beschrijft een *fact*, namelijk dat we ons altijd in een winkel bevinden.

Een keuzeregel die hierbij van toepassing is, is dan bijvoorbeeld: “Wanneer we in een winkel staan, kunnen we pizza, wijn of mais kopen”.

```
{ pizza; wijn; mais} :- in(winkel).
```

Dit houdt in dat, wanneer aan `in(winkel)` voldaan is, we elke combinatie van elementen in onze oplossing kunnen hebben, gaande van geen van de elementen tot alle elementen. Dit laat ons toe om een basis aan elementen te genereren. Dit is de Generate-stap. De mogelijke oplossingen zijn dus:

```
niets
pizza
wijn
mais
pizza wijn
pizza mais
mais wijn
pizza wijn mais
```

Omdat `in(winkel)` een *fact* is, zal deze altijd waar zijn. Het is dus ook deel van iedere oplossing, maar is weggelaten voor eenvoud.

Een kardinaliteitsregel kan hierna gebruikt worden om vast te leggen dat in onze oplossing aan een bepaald aantal elementen voldaan moet zijn. “Voor een feestje van studenten moeten minstens twee van de winkelproducten gekocht worden” zouden we bijvoorbeeld kunnen schrijven als:

```
feestje :- 2 { pizza; wijn; mais}.
```

Van de zeven mogelijke oplossingen die onze eerdere keuzeregel gaf, zijn er nu dus vier die leiden tot een feestje. De oplossingen zien er nu als volgt uit:

```
geen
pizza
wijn
mais
mais wijn feestje
pizza mais feestje
pizza wijn feestje
pizza wijn mais feestje
```

Bemerk dat de modellen zonder feestje nog steeds deel zijn van onze oplossingen. Dit komt omdat we nergens de beperking hebben opgelegd dat er altijd een feestje moet zijn. Hiervoor kunnen we een integriteitsbeperking gebruiken.

```
:- not feestje.
```

Een integriteitsbeperking haalt alle modellen waar niet voldaan is aan de *body* van de beperking uit de oplossing. Elk model waar geen feestje was, is nu niet meer deel van onze oplossingsset.

Stel dat de studenten op een strak budget zitten, en maximaal twee van de winkelproducten willen kopen, kunnen we dat als volgt formuleren:

```
feestje :- {pizza; wijn; mais} 2.
```

Een gewichtsregel kan in dit voorbeeld gebruikt worden om te definiëren dat `pizza` meer aanleiding geeft tot een feestje dan `mais`.

```
feestje :- 5 { 5:pizza; 3:wijn; 2:mais}.
```

In dit voorbeeld zou bijvoorbeeld enkel `pizza` al genoeg zijn voor een feestje, terwijl dat wanneer we een feestje met `mais` willen, er sowieso `wijn` en/of `pizza` nodig is. De oplossingen die hier aan voldoen zijn dus:

```
pizza feestje
pizza wijn feestje
pizza wijn mais feestje
mais wijn feestje
```

Verder is het ook mogelijk in ASP om één of meerdere kostfuncties te definiëren, waarna we deze kunnen optimaliseren (door te minimaliseren of te maximaliseren). Als onze studenten dus een zo goedkoop mogelijk feestje willen organiseren, kunnen we dit definiëren als:

```
#minimize{ 3@1:pizza; 1@1:wijn; 1@1:mais}.
```

Ieder element in een optimalisatiefunctie is van de vorm `w@p:I`, met `w` onze kost, `p` de prioriteit en `I` de conditie. De prioriteit p wordt gebruikt om de prioriteit aan te duiden wanneer er meerdere optimalisatiefuncties in één programma voorkomen. De oplossing van dit voorbeeld is dus:

```
wijn mais
```

omdat dit met een kost van `2` de goedkoopste manier is om een feestje te geven.

2.2 Het IDP-Systeem

Het IDP-systeem is een implementatie van het kennisbankparadigma (KBP). Zo'n implementatie bestaat eenvoudig gezien uit twee stukken: een kennisweergave-taal, en een aantal krachtige inferentiemethoden (De Cat et al. (2014)).

2.2.1 Kennisbank Paradigma

Het eerste deel van het KBP is de kennisweergave-taal. De kennisweergave-taal die door het IDP-systeem gehanteerd wordt heet de IDP taal. Het voordeel van zo'n kennisweergave-taal is dat ze natuurlijker en leesbaarder overkomt voor mensen dan een doorsnee programmeertaal.

De IDP taal is een logische, declaratieve programmeertaal. Deze laat ons toe om in het IDP-systeem een aantal regels op te stellen, en vervolgens bijvoorbeeld het systeem modellen te laten genereren die aan deze regels voldoen. Hiervoor maakt het systeem gebruik van een eindige-modellengenerator. De logica die gebruikt wordt bij het formuleren van deze regels is FO($\cdot$). Dit is zoals eerder vermeld eerste-orde logica (First Order logic), maar uitgebreid met inductieve definities, aggregaten, partieelfuncties en types (Bruynooghe et al. (2015)).

Eenvoudig gezien laat eerste-orde logica toe om een IDP-systeem te definiëren, welke bestaat uit een aantal predicaten en symbolen, om vervolgens onder andere na te gaan of aan alle predicaten voldaan is. Dankzij de uitbreidingen die FO($\cdot$) extra toevoegt, kan het gebruikt worden om geavanceerde modellen na te kijken. Deze uitbreidingen zijn:

1. Inductieve Definities: deze laten toe om definities te vormen met nog onbekende variabelen;
2. Aggregaten: zaken zoals `som`, `minimum` en `maximum`;
3. Partieelfuncties: kunnen gebruikt worden om bijvoorbeeld een onvolledige set van antwoorden aan te geven, om vervolgens deze set uit te breiden naar een volledige set met antwoorden;
4. Types: laten toe om bepaalde zaken meer specifiek te modelleren. Een persoon kan bijvoorbeeld niet enkel een persoon zijn, maar ook student of leerkracht.

Het tweede deel van het KBP zijn de inferentiemethoden om toe te passen op de kennis. Enkele van de inferentiemethoden die het IDP-systeem aanbiedt, zijn model expansie, model checking en optimalisatie.

Bij model expansie maakt het IDP-systeem zelf een eindig aantal modellen aan, welke allemaal aan vooropgestelde regels voldoen. Hierbij kunnen ook gekende waarden ingevuld worden, waarna IDP deze aanvult tot één of meerdere correcte modellen (indien oplosbaar).

Model checking zal nagaan of een model aan de gedefinieerde regels voldoet.

Optimalisatie laat toe om optimalisatieproblemen op te lossen. IDP zal het model zo oplossen dat de te minimaliseren waarde zo klein mogelijk is. Een voorbeeld is het zo klein mogelijk krijgen van

de hoeveelheid studenten die bij elkaar willen zitten, maar dit niet kunnen. Of met andere woorden: proberen om voor zoveel mogelijk studenten aan hun voorkeur te voldoen.

Om een kennisbanksysteem praktisch te kunnen gebruiken heeft het nood aan een imperatief programmeerbaar interface. Dit laat onder andere toe om de output uit te lezen en de instelling van het KBS in te stellen. Hiervoor gebruikt het IDP-systeem de programmeertaal Lua[2] (De Cat et al. (2014)). Dit verklaart ook het acroniem "IDP": *Imperative Declarative Programming*. Het declaratieve programmeren gebeurt in de IDP taal, en het imperatieve in Lua.

De versie van IDP die in dit werk gebruikt wordt heet IDP3. Het IDP-systeem bestaat al een tiental jaar, maar is pas sinds 2012 een volwaardig kennisbanksysteem. Voor 2012 was er nog sprake van IDP2. IDP2 was een systeem voor modelexpansie, waar eerste-orde logica kon gebruikt worden om modellen te expanderen. IDP2 ondersteunt eerste-orde logica met enkele uitbreidingen, maar niet zoveel uitbreidingen als FO(·) (De Cat et al. (2014)).

2.2.2 IDP taal

Een programma geschreven in IDP bevat zoals eerder gezegd een imperatief deel in Lua, en een declaratief deel in de IDP taal. Dit imperatieve deel bestaat uit procedures welke de inferentiemethoden kunnen oproepen. Onder het declaratieve deel vallen `vocabularies`, `structures`, `theories` en `terms` (De Cat et al. (2014)). Meer uitleg over wat deze zijn en wat ze doen, volgt hieronder. Hiervoor wordt telkens een voorbeeld gebruikt van toepassing op dit onderzoek. Deze voorbeelden zijn niet de definitieve versies, maar eerder aangepaste versies om brede voorbeelden te tonen. De uitleg die gegeven wordt is slechts beknopt en bevat enkel de zaken die voor dit onderzoek van toepassing zijn. Voor een volledige uitleg over de syntax van IDP zie sectie 1.4 van De Cat et al. (2014).

2.2.2.1 Vocabulary

De vocabulary van een IDP programma bevat alle verschillende symbolen, predicaten en functies die in de loop van het programma gebruikt worden. Een voorbeeld is te zien in listing 2.1.

Op lijn twee tot en met lijn vier worden drie types gedeclareerd: `Student`, `Groep` en `Getal`. Types kunnen gedeclareerd worden met het keywoord `type`, en een type kan op zijn beurt gedeclareerd worden als subtype aan de hand van het keywoord `isa`. Integers en natuurlijke getallen zijn standaard gedefinieerd in elk vocabulary en kunnen dus aan de hand van hun naam (`nat` en `int` respectievelijk) gebruikt worden als supertype.

De volgende declaratie is een relatie. Relaties zijn functies met types als argumenten en een boolean als antwoord. Zo kan de relatie `ZelfdeGroep` gebruikt worden om te definiëren of twee studenten `x` en `y` in dezelfde groep willen. Als `ZelfdeGroep(x,y)` waarde `true` teruggeeft, dan is aan deze relatie voldaan. Indien ze `false` teruggeeft, is niet aan deze relatie voldaan.

Listing 2.1: Voorbeeld van een vocabulary

[2] `https://www.lua.org/`

```
1  Vocabulary V {
2      type Student isa String
3      type Groep isa String
4      type Getal isa nat
5
6      ZelfdeGroep(Student, Student)   //Studenten die bij elkaar willen.
7
8      InGroep(Student): Groep
9      GroepGrootte(Groep): Getal
10     partial Voorkeur(Student): Getal
11 }
```

Lijnen acht tot en met tien dienen hier om functies te definiëren. Functies uiten een functioneel verband tussen elementen van één of meerdere verzamelingen op de elementen van een andere verzameling. Een functioneel verband houdt in dat dit een veel-op-een relatie is. Zo kan iedere `Student` maar aan één `Groep` toegewezen worden. Dit is te schrijven als `InGroep(Student)`: `Groep`. Meerdere studenten kunnen wel dezelfde groep delen, maar geen enkele student kan meerdere groepen of geen groep hebben.

`GroepGrootte` mapt een `Groep` op een `Getal`. De waarde van deze functie zal een `Getal` zijn, en omdat in de typedeclaratie weergegeven is dat de functie als supertype een `nat` heeft, is de waarde van `GroepGrootte` dus een natuurlijk getal. De voornaamste reden om deze tussenstap te maken is om een limiet op te leggen van de bereik waarin dit natuurlijk getal ligt. Een bereik opleggen voor een natuurlijk getal is vaak een goed idee, omdat het huidig IDP-systeem niet goed kan omgaan met oneindige types.

`Voorkeur` is geen gewone functie, maar een partieelfunctie. Dit houdt in dat de functie niet noodzakelijk volledig ingevuld moet zijn, in tegenstelling tot `InGroep` en `GroepGrootte`. Het voornaamste voordeel van een partieelfunctie is dat we deze functie enkel moeten invullen voor de argumenten waarvoor we de functiewaarde kennen en/of waarvoor ze relevant is. Een nadeel van een partieelfunctie is de onzekerheid voor niet gedefinieerde waarden.

2.2.2.2 Structure

De structuur van een IDP programma laat ons toe om de mogelijke waarden van types vast te leggen, om relaties (al dan niet gedeeltelijk) te definiëren en om functies die vastliggen al volledig in te vullen. Een structuur wordt gedeclareerd over het domein van een vocabularium. Een voorbeeld hiervan is te zien in listing 2.2.

In deze listing wordt weergegeven hoe men een type een waarde kan geven. Iedere variabele van het type `Getal` kan bijvoorbeeld enkel maar een waarde hebben tussen 0 en 1000. Het type `Groep` kan in dit voorbeeld de elementen `groep1` of `groep2` zijn. Op deze manier limiteren we het aantal groepen tot maximaal twee. Hetzelfde geldt voor `Student`, welke enkel de strings `Jos`, `Jefke`, `Marie` of `An` zijn. Anders gezegd hebben we dus vier studenten. Merk op dat de naam van de studenten niet uitmaakt. Deze hadden evengoed `s0` tot en met `s3` kunnen zijn. Zolang er geen

twee dezelfde namen voorkomen kunnen we een `Student` zonder problemen definiëren.

Listing 2.2: Voorbeeld van een Structure

```
1 Structure S: V{
2     Getal = {0..1000} //nat tussen 0 en 1000
3     Groep = {groep1; groep2;} //twee groepen
4     Student = {Jos; Jefke; Marie; An;} //vier studenten
5
6     ZelfdeGroep = {Jos,Jefke; Jefke,Marie; Marie,Jefke;}
7 }
```

Vervolgens definiëren we een relatie. De relatie `ZelfdeGroep` is een relatie tussen twee studenten (zie listing 2.1). Deze kunnen we invullen door tussen accolades de twee studenten met een komma te scheiden. Indien er meerdere relaties ingevuld moeten worden, kunnen we deze scheiden met puntkomma's. In het voorbeeld wil `Jos` bij `Jefke` en willen `Jefke` en `Marie` bij elkaar.

2.2.2.3 Theory

De theorie van een IDP programma is het gedeelte waarin we alle beperkingen definiëren waar onze modellen moeten aan voldoen. Dit kunnen zowel harde, als zachte beperkingen zijn. Een harde beperking is een beperking waaraan voldaan moet zijn om een geldige oplossing te hebben. Aan een zachte beperking moet niet verplicht voldaan zijn, maar indien er niet aan voldaan is zal dit tot een strafwaarde leiden. Door deze strafwaarde vervolgens te minimaliseren, kunnen we het aantal zachte beperkingen waaraan niet voldaan is zo klein mogelijk maken.

In het voorbeeld in listing 2.3 zijn drie beperkingen gedefinieerd. Een beperking wordt altijd beëindigd met een punt. De beperkingen worden gevormd met FO(·). Om symbolen van de FO(·) logica (zoals "$\forall$", "$\exists$" of "$\Rightarrow$") te kunnen gebruiken, maakt IDP een vertaling naar een keyboardvriendelijker notatie. Voor een klein overzicht van de vertalingen van sommige van de symbolen gebruikt in dit werk, zie tabel 2.1. Voor een volledig overzicht van alle symbolen, zie De Cat et al. (2014).

Tabel 2.1 Vertaling enkele FO(·) symbolen naar IDP taal

FO(·)	IDP taal	FO(·)	IDP taal	FO(·)	IDP taal
$\forall$	!	$\vee$	|	$\neg$	~
$\exists$	?	$\wedge$	&	$\Rightarrow$	=>
$!=$	=	$\leqslant$	=<	$\leftarrow$	<-
$\neq$	~=	$<$	<	$\Leftrightarrow$	<=>

Zo staat er in listingvoorbeeld 2.3 voor de eerste beperking:

$$\forall g1, g2 \in \text{Groep} : |\text{GroepGrootte}(g1) - \text{GroepGrootte}(g2)| < 3 \tag{2.1}$$

Oftewel, dat voor elke twee groepen, de absolute waarde van het verschil in groepsgrootte kleiner moet zijn dan drie.

De tweede beperking dient om de grootte van groepen te berekenen. Hiervoor maakt het gebruik van de kardinaalsoperator (#), welke één van de vijf aggregaatsfuncties is. Deze telt voor iedere `Student x` het aantal keer dat aan de conditie `InGroep(x) = g` voldaan is. In woorden zegt de beperking: "Voor iedere variabele `g`, welke een `Groep` is, geldt dat de `GroepGrootte` gelijk is aan het aantal student x waarvoor geldt dat student x in de groep zit". Een ander voorbeeld van een aggregaatoperator is terug te vinden in listing 2.5.

De derde beperking zal garanderen dat iedere student een groep krijgt toegewezen. Hier staat namelijk "Voor iedere `Student x` bestaat er een `Groep g` waarvoor geldt dat `x` in `Groep g` zit".

Listing 2.3: Voorbeeld van een theory

```
1  Theory T : V {
2
3      !g1[Groep] g2[Groep]:
4          abs(GroepGrootte(g1) − GroepGrootte(g2)) < 3.
5
6      !g[Groep]: GroepGrootte(g) =
7          #{x[Student]: InGroep(x) = g}.
8
9      !x[Student]: ?g[Groep]: InGroep(x) = g.
10
11     {
12         !x[Student] y[Student]: Samen(x,y) <− InGroep(x) = InGroep(y).
13     }
14
15
16 }
```

2.2.2.4 Term

Eerder in dit werk werd vermeld dat het IDP-systeem ons toelaat om modellen te minimaliseren. Dit is mogelijk aan de hand van een `Term`-blok. In dit blok kunnen we een term schrijven, waarvan we de waarde willen minimaliseren.

Zo is in listing 2.4 een voorbeeld te zien van een term voor de waarde van de variabele `Eindgewicht`, welke dan geminimaliseerd kan worden met behulp van de juiste inferentietaak.

Listing 2.4: Voorbeeld van een term met variabele

```
Term t1 : V{
    Eindgewicht
}
```

In listing 2.5 bestaat de term uit een aggregaatfunctie die groepsgroottes sommeert. De formule in de term werd eerder gebruikt om de groepsdeviatie vast te leggen (zie listing 2.3), behalve dat er hier gebruikt gemaakt wordt van een `sum` operator. Hier wordt de som gemaakt van het verschil

van alle groepgroottes onderling. Het minimaliseren van deze term zou betekenen dat er gezocht wordt naar de modellen met een zo klein mogelijke variatie, maar die toch nog voldoen aan alle regels beschreven in onze theorie.

Listing 2.5: Voorbeeld van een term met formule

```
Term t2 : V{
    sum{g1[Groep] g2[Groep]: g1 ~= g2:
            abs(GroepGrootte(g1) - GroepGrootte(g2))}
}
```

2.2.2.5 Procedure

Het procedure-blok is het enige imperatieve gedeelte van het IDP-systeem. Hierin kunnen we verschillende opties van de kennisbank instellen en configureren, en inferentietaken kiezen zoals het genereren van modellen. Alle code in de procedure is geschreven in Lua[3]. Wanneer we het IDP-systeem starten, start het de `main()`-procedure en voert deze uit. Een voorbeeld van zo'n procedure is te vinden in listing 2.6.

Listing 2.6: Voorbeeld van een procedure main()

```
1 Procedure main(){
2   start = os.clock()
3   stdoptions.nbmodels = 1
4   minimize(T,S,t1)
5   print(string.format("elapsed time: %.2f\n", os.clock()-start))
6 }
```

De variabele `stdoptions.nbmodels` laat ons toe om in te stellen hoeveel modellen we IDP willen laten teruggeven. Door `minimize(T,S,t1)` zal IDP proberen om het/de model(len) te zoeken voor struct S die voldoen aan theorie T, waarvoor term t1 minimaal is.

2.2.3 Interactiviteit in IDP

Een doel van dit werk is om te onderzoeken hoe eenvoudig het is om het IDP-kennisbanksysteem te gebruiken in een interactieve omgeving.

Pieter et al. (2016) onderzochten reeds hoe goed het kennisbank paradigma ingezet kan worden in een interactieve configuratie. In hun werk maken ze gebruik van de IDP-kennisbank en meten ze deze aan een aantal subtaken die een interactief configuratiesysteem moet kunnen aanbieden.

Deze subtaken zijn:

1. informatie van de gebruiker kunnen verkrijgen
2. consistente waarden voor een parameter genereren

[3] `https://www.lua.org/`

3. propagatie van informatie
4. consistentie van een waarde nakijken
5. een configuratie nakijken
6. automatische aanvulling
7. uitleg over waarom een parameter een bepaalde waarde heeft
8. *backtracking*

Elk van deze subtaken zijn mogelijk in het IDP-systeem. Bovendien hebben de onderzoekers een praktische applicatie ontworpen en deze bleek voldoende bruikbaar te zijn voor hun context.

2.2.4 Vergelijking met ASP

Waar IDP altijd dezelfde structuur heeft (in willekeurige volgorde `Vocabulary`, `Structure`, `Theory`, `Procedure` en eventueel nog een `Term`), heeft ASP helemaal geen structurele verplichtingen, enkel de gesuggereerde aanpak zoals vermeld in sectie 2.1. Het IDP-systeem kent maar één negatie, de "$\sim$"-operator, terwijl er in ASP twee negaties mogelijk zijn. In ASP wordt namelijk onderscheid gemaakt tussen de "`not`" en een sterke negatie (Bruynooghe et al. (2015)). In tabel 2.2 staan deze verschillen tussen IDP en ASP opgelijst.

Tabel 2.2 Verschillen tussen IDP en ASP

IDP	ASP
• leesbaarder (natuurlijkere taal)	• minder leesbaar
• functies mogelijk	• geen functies mogelijk
• vaste structuur	• geen vaste structuur
• één vorm van negatie	• twee soorten negatie

2.3 Alternatieve technologieën

Het KBP en ASP bevinden zich allebei in het onderzoeksveld van artificiële intelligentie (AI). Naast kennisbanksystemen zijn er in de literatuur ook verscheidene andere technieken om problemen zoals het toewijzen van studenten aan groepen op te lossen. In dit hoofdstuk wordt kort gekeken naar constraint programming (CP), wat ook uit het veld van AI komt, en genetische algoritmes (GA), wat oorspronkelijk uit het veld van operationeel onderzoek komt (OO) maar tegenwoordig ook in toepassingen van AI gebruikt worden.

2.3.1 Constraint programming

Een andere soort van declaratief programmeren is constraint programming (CP) (Dovier et al. (2007)). Problemen worden eerst herleid tot variabelen, waarna vervolgens aan iedere variabele

een waarde toegekend kan worden. De relaties tussen deze variabelen worden dan vastgelegd aan de hand van beperkingen.

Het programma zal dan aan elke variabele een range van waarden toekennen, en in een volgende stap deze waarden propageren doorheen alle beperkingen, om na te kijken aan welke beperkingen nog voldaan is.

2.3.2 Genetisch algoritme

Genetische algoritmes (GA) zijn geïnspireerd door natuurlijke selectie (Mitchell (1995a)). Een GA zoekt doorheen een ruimte van "chromosomen", welke elk een kandidaat-oplossing voor het op te lossen probleem voorstellen. De zoektocht van het GA gebeurt door het verwerken van generaties van chromosomen in andere generaties van chromosomen. De eerste generatie van chromosomen bestaat uit een volledig willekeurige set. Aan de hand van een fitnessfunctie wordt berekend hoe fit een chromosoom is, waarbij deze fitness uitdrukt hoe goed het chromosoom is in het oplossen van het probleem. Waar in de natuur enkel de sterksten overleven, zal in een genetisch algoritme het chromosoom met de grootste fitnesswaarde de grootste kans hebben om te overleven, en in de volgende generatie terug komen. De bedoeling van het GA is dus om een oplossingsset van chromosomen te bekomen waarbij de fitnesswaarde van de chromosomen altijd zo groot mogelijk is.

Bij het genereren van de volgende generatie wordt in een eenvoudig GA gebruik gemaakt van drie soorten operatoren op de chromosomen.

1. Selectie: een chromosoom wordt integraal gekopieerd naar de volgende generatie.
2. Crossover: twee chromosomen worden samengevoegd tot twee nieuwe chromosomen, elk met eigenschappen van beide vorige chromosomen.
3. Mutatie: een chromosoom wordt willekeurig aangepast.

GA kunnen zeer goed ingezet worden voor zuivere optimalisatieproblemen, maar zijn niet zo bruikbaar in een interactieve context.

2.4 Indelen in groepen

2.4.1 Collaborative Learning

Het doel van de meeste onderzoeken rond het maken van algoritmes om studenten in te delen in groepen is om optimaal *Collaborative Learning* (CL) te stimuleren. De definitie van CL is als volgt:

> *"Collaborative learning is the instructional use of small groups so that students work together to maximize their own and each other's learning"* Johnson et al. (2014).

CL zorgt er dus voor dat studenten, soms zonder dit te weten, elkaar kunnen stimuleren om betere resultaten te behalen. Uit testen is gebleken dat heterogene groepssamenstellingen het beste werken om CL optimaal te stimuleren. Niet enkel moet in een ideale verdeling de intelligentie van een student in rekening gebracht worden, maar ook persoonseigenschappen, achtergrond, gender, houdingen en persoonlijkheden (Bekele (2006)) .

Huxham and Land (2000) stellen dat er drie manieren zijn om studenten in groepen in te delen:

1. compleet willekeurig;
2. door de studenten zelf te laten kiezen;
3. een heuristiek verzinnen die rekening houdt met verscheidene studenteigenschappen.

In theorie zorgt een compleet willekeurige indeling van de groepen voor een gebalanceerde verdeling van de studenten. Maar aangezien we in praktijk enkel met kleine aantallen werken, is dit geen geschikte methode. Ook gaat er zo heel veel nuttige kennis over de studenten verloren.

Wanneer de studenten volledig zelf mogen kiezen, kan het gebeuren dat de groepen meer gemotiveerd zijn omdat ze elkaar al beter kennen. Maar dit zorgt potentieel voor problemen op vlak van sociale isolatie, discriminatie en vooroordelen.

De literatuur raadt aan om voor de derde optie te gaan, omdat we via deze optie *collaborative learning* optimaal kunnen stimuleren, maar in realiteit wordt meestal gekozen voor één van de eerste twee.

2.4.2 Algoritmes

Er bestaan een aantal algoritmes die groepen optimaal proberen in te delen op basis van CL.

Bekele (2006) heeft een Bayesiaans netwerk proberen te trainen om de moeilijker te bepalen studenteigenschappen te voorspellen, om vervolgens deze data te gebruiken om de studenten in te delen. In plaats van ellenlange enquête's af te moeten nemen bij studenten volstonden een paar gerichte vragen per student om zijn/haar verdere eigenschappen te extrapoleren. De resultaten van de studenten in de intelligent gevormde groepen werden vergeleken met de groepen waarbij de studenten zelf mochten kiezen, en de groepen waarbij de studenten willekeurig werden toegewezen.

Niet enkel lag het gemiddeld resultaat van de testgroep hoger, maar de stijging in prestatie lag bij de testgroep ook het hoogst. Het merendeel van de studenten uit de testgroepen vond hun groepsindelingen optimaal, en vond dat de samenwerking goed verliep. Het percentage van studenten dat samenwerktte was 51.2% bij de testgroepen, 44.2% bij de zelf toegewezen groepen en 31.1% bij de willekeurige groepen. Dit toont aan dat studenten beter gemotiveerd zijn om te werken bij het gebruik van het ontwikkelde algoritme (Bekele (2006)).

Een ander werk, van Moreno et al. (2012), heeft een genetisch algoritme ontworpen dat de optimale verdeling van studenten zoekt voor CL, op basis van enkele studenteigenschappen. Dit hebben ze gedaan door iedere student voor te stellen als een array van eigenschappen, en hier hun genetisch

algoritme op los te laten. Uit het onderzoek bleek dat de studenten betere resultaten haalden wanneer ze werden ingedeeld door het algoritme, dan wanneer dit niet het geval was.

Tacadao and S.J. (2015) hebben een generisch model voorgesteld met alles waaraan een beperkingsgebaseerde heuristiek moet voldoen om succesvol *collaborative learning* te stimuleren. De onderzoekers hebben vijf verschillende parameters bepaald die in rekening gebracht moeten worden bij het correct indelen van een groep. Deze zijn:

- individuele bekwaamheid, iets wat kan gemeten worden aan de hand van behaalde resultaten in het secundair onderwijs of een ijkingstest;
- leer- en denkstijlen;
- groep-gerelateerde eigenschappen, bijvoorbeeld kan de student wel samenwerken;
- persoonseigenschappen, zoals motivatie en hoe vlot in omgang een student is;
- voorkeur in groepsgenoten.

Deze parameters werden gebruikt om hun groepsindelingen te maken die optimaal CL stimuleren. De voornaamste moeilijkheid hierbij was het vertalen van de beschreven parameters naar parameters die gebruikt kunnen worden om mee te modelleren.

Uit hun onderzoek is gebleken dat de oplossing een afweging is tussen de eerste vier parameters en de laatste parameter. Het is bijvoorbeeld mogelijk om de voorkeur in groepsgenoten als zachte beperking te gebruiken, en de eerste vier in een harde beperking te verwerken.

2.5 Interface rond IDP

Met enkel het gebruik van het IDP-systeem, zou een gebruiker telkens manueel het geprogrammeerde IDP-systeem moeten aanpassen, en dus kennis nodig hebben van de IDP taal. Dit is niet haalbaar voor een leek. Om een gebruiksvriendelijke toepassing te maken rond IDP hebben we een tussenstap nodig. Voor deze tussenstap zijn er een aantal mogelijkheden. De meest voor de hand liggende oplossing is om de programmeertaal Lua[4] te gebruiken, aangezien deze zoals eerder vermeld zich al in het IDP-systeem bevindt. Ook zijn er voor IDP twee API's beschikbaar: de Java API van Calus (2011) en de Python 2 API van Vennekens (2015) genaamd Pyidp. Voor een overzicht van deze mogelijkheden, zie tabel 2.3.

2.5.1 Lua

De eenvoud van Lua volgt uit het feit dat het niet nodig is om extra software te installeren, bovenop IDP. Doordat het ingebouwd is, is het aanpassen van het IDP bestand en het uitlezen van de output relatief vlot mogelijk.

[4] `https://www.lua.org/`

Het grootste nadeel aan Lua is dat hoewel er GUI frameworks bestaan, de meeste niet meer geüpdate zijn sinds 2012, of enkel werken op een incompatibele Lua versie. Dit zorgt ervoor dat we onze applicatie niet in Lua kunnen maken, en we dus moeten opteren voor een andere taal via een API om daarin onze GUI te maken.

2.5.2 Java API

De Java API van Calus (2011) zou het GUI probleem kunnen oplossen door toe te laten dat we in Java Swing onze GUI maken. Het probleem dat zich hier stelt is dat deze API gemaakt werd in 2011 en enkel werkt voor IDP2, terwijl wij IDP3 gebruiken.

2.5.3 Pyidp

Een andere mogelijkheid is de Python 2 API Pyidp van Vennekens (2015). Deze is ontwikkeld in 2015, en is voldoende recent. Het aanbrengen en uitlezen van data is redelijk eenvoudig, en het dynamisch vormen van een IDP bestand is met Pyidp ook haalbaar. Programmeren via Python laat ons ook toe om het Qt[5] framework te gebruiken voor onze GUI.

Het voornaamste nadeel van de Pyidp API is dat het geschreven is voor Python 2. Niet enkel is Python 2 officieel niet meer ondersteund in 2020 [6] maar de library om te werken met Qt5,[7] het Qt-framework dat wij zullen gebruiken in dit werk, wordt enkel ondersteund in Python 3. Er is in Python 2 een conversie-library[8] om Qt5 te kunnen gebruiken, maar dit is een onofficiëel project en niet van de developers van Qt zelf. Qt4 support is voor Python 2 wél aanwezig, maar Qt5 krijgt natuurlijk de voorkeur omdat deze meer recent is en hier nog actief aan gewerkt wordt. Een andere mogelijke oplossing is om Pyidp om te vormen naar een versie die werkt met Python 3. Dit is dan ook waar wij voor opteren in dit onderzoek.

[5] https://www.qt.io/
[6] https://hg.python.org/peps/rev/76d43e52d978
[7] https://riverbankcomputing.com/software/pyqt/download5/
[8] https://github.com/pyqt/python-qt5

Tabel 2.3 Vergelijking mogelijkheden interface rond IDP

Mogelijkheid	Voordelen	Nadelen
Lua	• makkelijk scriptbaar • ingebouwd aanwezig	• geen moderne GUI toolkits aanwezig • soms wat onhandig programmeren
Java API	• GUI's in Swing • mogelijkheid tot object-gericht programmeren	• ondersteunt enkel IDP2
Pyidp	• makkelijk aanbrengen & uitlezen van data • Qt5 framework aanwezig	• Pyidp ondersteunt enkel Python 2 • Pyqt5 ondersteunt enkel Python 3, dus conversie is nodig

Hoofdstuk 3

IDP

In dit hoofdstuk bespreken we de implementatie van de groepsindeling in het IDP-systeem. Het concreet implementeren in het IDP-systeem gebeurt in twee stappen:

1. de eerste, initiële indeling;
2. de incrementele indeling;

Voor de initiële indeling wordt eerst een naïve implementatie gegeven en wordt vervolgens besproken hoe we deze optimaliseren. Voor de incrementele indeling wordt besproken welke inputs het systeem moet krijgen en hoe we kunnen nagaan aan welke voorkeuren wel en niet voldaan is.

3.1 Programmeer- en testomgeving

De implementatie gebeurt aan de hand van volgende software:

- IDP: versie 3.7.1;
- IDE: Vim[1].

In dit hoofdstuk worden verschillende experimenten met het IDP-systeem uitgevoerd. Om "invloed van de buitenwereld" zoveel mogelijk te kunnen vermijden, zijn al deze experimenten uitgevoerd op dezelfde hardware. Op deze manier zijn de experimenten zo goed mogelijk onderling vergelijkbaar. De hardware voerde tijdens het testen enkel het IDP-systeem uit, en niets extra's op de achtergrond.
Hardware:

- CPU: Pentium G3258 dualcore @3.2Ghz
- RAM: 2x4GB DDR3 @1333MHz

[1] https://www.vim.org/

Software:

- OS: Fedora 29
- kernel: Linux 4.18.16-300.fc29
- IDP: versie 3.7.1

Doordat de testen plaatsvonden op een oudere pc, zijn de absolute waarden van de uitkomsten niet representatief voor hedendaagse hardware. De waarden relatief ten opzichte van elkaar zijn wél representatief.

3.2 Initiële indeling

Zoals vermeld in 1.3 is de initiële indeling gebaseerd op postcodes. Op deze manier kunnen we heel ruwweg de studenten indelen op basis van afstand tot elkaar (postcode 1860, Meise ligt bijvoorbeeld naast 1840, Londerzeel).

3.2.1 Naïve indeling

Een voor de hand liggende implementatie van deze initiële indeling is weergegeven in listing 3.1.

Listing 3.1: Voorbeeld van een naïve initiële indeling voor tien studenten.

```
1  vocabulary V {
2      type Student isa nat
3      type Getal isa nat
4      type Groep isa nat
5      GroepGrootte(Groep): Getal //Iedere groep heeft een grootte.
6      HuidigeGroep(Student): Groep //Iedere student zit in een groep.
7      Afstand(Student,Student): Getal //Afstand tussen twee studenten.
8      TotaleAfstand : Getal //De totale afstand van alle groepen samen.
9      MinGrootte : Getal //Minimum aantal studenten.
10     MaxGrootte : Getal //Maximum aantal studenten.
11 }
12 structure S : V {
13     Student = {0..9} //tien studenten
14     Groep = {0..1} //twee groepen
15     Getal = {0..100000} //range van getallen
16     Afstand = {0,0->0; 0,1->432; 0,2 -> 643; ..; 8,9->200; 9,9->0}
17 }
18 theory T : V {
19     //Bepalen grootte van een groep.
20     !groepnummer[Groep]: GroepGrootte(groepnummer) =
21         #{x[Student]: HuidigeGroep(x)=groepnummer}.
22
```

```
23      //Vastleggen spreiding op een groepgrootte.
24      !groep[Groep] : MinGrootte =< GroepGrootte(groep) =< MaxGrootte.
25
26      //Berekenen TotaleAfstand.
27      TotaleAfstand = sum{x[Student] y[Student]: HuidigeGroep(x) = HuidigeGroep(y):
28          Afstand(x,y)}.
29  }
```

De afstanden tussen studenten zijn aan de hand van preprocessing al omgezet in een afstandenmatrix. Op deze manier is het eenvoudig om de afstand tussen twee studenten te verkrijgen, zonder te hoeven rekenen. De functie `HuidigeGroep` zal ervoor zorgen dat elke student exact één groep krijgt toegewezen. De grootte van een groep wordt berekend door voor elk groepnummer na te gaan hoeveel studenten in die groep zitten. Aan de hand van de groottes van groepen kunnen we hier ook een maximumspreiding op vastleggen: voor iedere groep moet gelden dat zijn grootte ligt tussen de minimum- en maximumgrootte per groep. De totale afstand tot slot wordt berekend door de `Afstand` van alle studentenparen die samen in een groep zitten op te tellen. Deze laten we vervolgens minimaliseren door het IDP-systeem.

3.2.2 Afstanden

Bij deze naïve oplossing stuiten we al meteen op een probleem: de afstandenmatrix gaat voor `n` studenten een grootte van `n` bij `n` hebben. Voor kleine aantallen studenten vormt dit geen probleem, maar voor grotere hoeveelheden wordt dit onoverzichtelijk in het IDP-systeem.

Een oplossing hiervoor is om de afstand tussen studenten niet in preprocessing te berekenen, maar in het IDP-systeem zelf. Dit rekent minstens even snel als bij een afstandenmatrix (zie tabel 3.1) en laat toe om gemakkelijker data aan te brengen. Immers, enkel de postcode van een student moet nu gegeven worden. De functie `Afstand` wordt vervangen door de functie `Woont`, dat voor elke student de postcode weergeeft. De berekening van de `TotaleAfstand` moet vervolgens vervangen worden door een berekening zoals in listing 3.2.

Listing 3.2: Berekenen afstand met de `Woont` functie.

```
TotaleAfstand = sum{x[Student] y[Student]: HuidigeGroep(x) = HuidigeGroep(y):
    abs(Woont(x) - Woont(y))}.
```

Tabel 3.1 Rekentijd voor een initiële indeling met of zonder afstandenmatrix.

hoeveelheid studenten	duratie met matrix	waarde zonder matrix
6	0.005s	0.002s
10	0.033s	0.032s
13	5.037s	4.026s
15	314s	275s

Door de postcode van student `y` van student `x` af te trekken bekomen we de afstand. Hiervan

hebben we de absolute waarde nodig, omdat we anders negatieve getallen verkrijgen.

Door deze aanpassingen kunnen we de afstanden *on-the-fly* berekenen. Op deze manier kunnen we `TotaleAfstand` sneller berekenen omdat we geen nood meer hebben aan pre-processing van de afstanden.

3.2.3 Symmetrie

Een grote zwakte van het IDP-systeem is symmetrie. Devriendt (2017) legt symmetrie uit aan de hand van het *pigeonhole problem*. Wanneer we 101 duiven willen toewijzen aan 100 kooien zodat geen enkele duif een kooi moet delen, voelen wij als mens meteen aan dat dit niet zal gaan, omdat er meer duiven zijn dan kooien. Het maakt niet uit welke duif in welke kooi geplaatst wordt, want dit leidt toch allemaal tot hetzelfde resultaat (1 duif blijft over zonder kooi). Het IDP-systeem daarentegen, zal proberen om kooi per kooi alle duiven eenmaal toe te wijzen, tot het na schijnbaar eindeloos rekenwerk en rekengeheugen tot de ontdekking komt dat het probleem onoplosbaar is. Dit komt doordat er symmetrie aanwezig is in het systeem, maar het IDP-systeem de symmetrie niet herkent en er dus niets aan doet. Het IDP-systeem ziet geen equivalentie tussen het invullen van duif 1, duif 2, duif ... of duif 101.

In onze huidige oplossing van de initiële studentenindeling doen er zich twee symmetrieën voor:

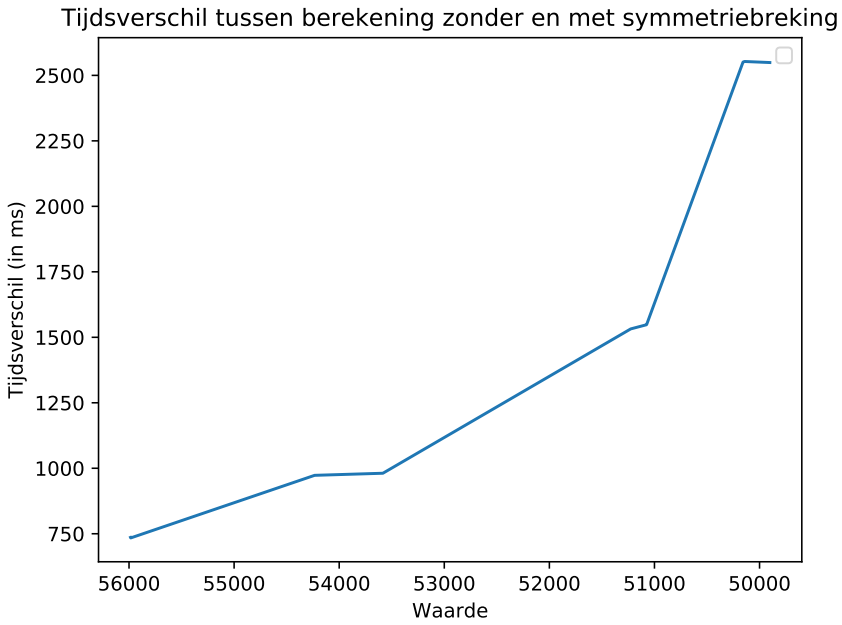
- tussen studenten;
- tussen groepen.

Het IDP-systeem ondersteunt statische symmetriebreking. Hierbij probeert het systeem symmetrie te voorkomen, door automatisch extra beperkingen toe te voegen die equivalente oplossingen ongeldig maken, maar andere oplossingen nog steeds toelaten. Aan de hand van de optie `stdoptions.symmetrybreaking` kunnen we deze op `static`, of op `none` instellen. In tabel 3.2 is de invloed van statische symmetriebreking te zien specifiek op ons probleem. Hieruit is te zien dat het gebruik van deze optie niet voor een kortere rekentijd zorgt. In figuur 3.1 is het tijdsverschil geplot tussen het berekenen van een initiële indeling met, en zonder symmetriebreking per nieuwe oplossing. De implementaties verkregen altijd binnen drie seconden van elkaar dezelfde tussenoplossing. Opvallend is dat de symmetriebreking enkel voor een vertraging van het systeem heeft gezorgd. We gaan het symmetrieprobleem op een andere manier moeten oplossen.

Tabel 3.2 Rekentijd met en zonder symmetriebreking.

aantal studenten	benodigde tijd met breking	benodigde tijd zonder breking
6	18.016s	0.132s
10	19.022s	0.132s
15	34.032s	34.992s

De symmetrie tussen studenten situeert zich in de manier waarop we `TotaleAfstand` berekenen. Zoals al vermeld in 3.2.2 berekenen we voor elk studentenpaar de afstand eigenlijk twee keer. Eén



Figuur 3.1: Rekentijd initiële indeling met en zonder symmetriebreking.

keer van de eerste student tot de tweede student, en één keer van de tweede student tot de eerste student. Het is uiteraard voldoende voor elk paar om enkel de afstand van de ene student tot de andere te berekenen, en zo dubbel werk te vermijden.

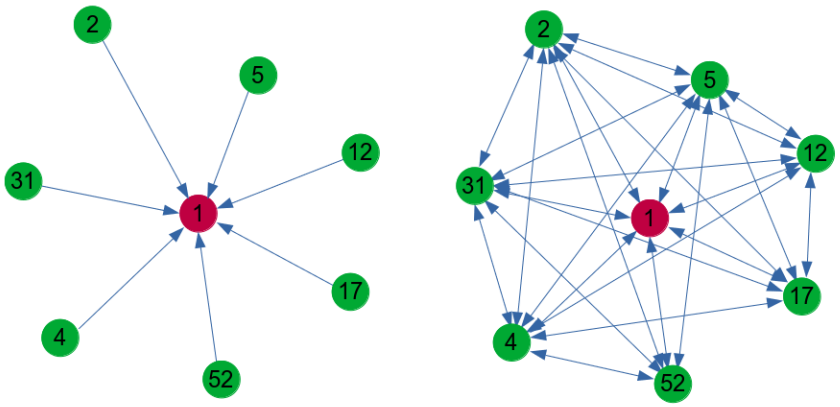
Deze symmetrie is te vermijden door een extra beperking te definiëren in de berekening van `TotaleAfstand`: we berekenen enkel de afstand wanneer de ene student een kleiner studentennummer heeft dan de andere student. Dit kan geïmplementeerd worden zoals in listing 3.3.

Listing 3.3: Berekenen afstand zonder symmetrie.

```
TotaleAfstand = sum{x[Student] y[Student]: HuidigeGroep(x) = HuidigeGroep(y) &
    x < y: abs(Woont(x) - Woont(y))}.
```

Het tweede voorkomen van symmetrie volgt uit het feit dat een groep gedefiniëerd is als een getal. Dit getal heeft geen betekenis over de eigenlijke inhoud van een groep, het doet enkel dienst als een getal om studenten aan toe te kunnen wijzen. Stel dat groep 1 uit Pieter en Astrid bestaat en groep 2 uit Bert en Bram, dan is dit hetzelfde als wanneer Pieter en Astrid in groep 2 zitten, en Bert en Bram in groep 1. Hoewel de groepsnummers anders zijn, zijn de indelingen in de groepen equivalent. Hieruit volgt dat wanneer we studenten in 10 groepen willen indelen, dit leidt tot 10! (3628800) modellen per oplossing waarvoor de groepsindelingen gelijk zijn. Het vermijden van deze symmetrie zorgt voor de beste optimalisatie van onze naïve oplossing.

De manier waarop deze symmetrie wordt vermeden is voor de hand liggend: in plaats van studenten aan een groepsgetal te linken, gaan we studenten onderling linken. We vervangen de functie `HuidigeGroep` door de relatie `Samen`, zoals in listing 3.4 beschreven.



Een groene bol is een blad, een rode bol is een wortel. Een bauwe lijn is een relatie tussen twee studenten. De relatie van een student naar zichzelf is weggelaten.

Figuur 3.2: Visuele weergaven van `GestructureerdSamen` en `Samen`.

Listing 3.4: Vervangen `HuidigeGroep` door `Samen` om symmetrie te vermijden.

```
Samen(Student, Student)
```

Een probleem hierbij is dat we op deze manier symmetrie onder studenten weer introduceren: `Samen(1,2)` komt op hetzelfde neer als `Samen(2,1)`. Dit zouden we kunnen oplossen door de extra beperking toe te voegen dat twee studenten enkel samen kunnen zijn, wanneer de eerste student een kleiner studentnummer heeft dan de tweede student, zoals weergegeven in listing 3.5.

Listing 3.5: Symmetrie vermijden in de `Samen` relatie.

```
!x[Student] y[Student]: Samen(x,y) => x < y.
```

Een nadeel hiervan is dat de `Samen` relatie hierdoor unidirectioneel wordt. `Samen` houdt dan niet langer bij met wie elke student samen zit, maar enkel met welke studenten met een groter studentnummer elke student samenzit. Dit belemmert de uitbreidbaarheid en modulariteit van onze implementatie. In plaats van de `Samen` relatie aan te passen, introduceren we daarom een nieuwe relatie die we `GestructureerdSamen` noemen. `GestructureerdSamen` zal enkel voldaan zijn voor elke kleinste studentnummer naar elk ander studentnummer waaraan `Samen` voldaan is. Dit kleinste studentnummer noemen we de `Wortel`, en elke andere student noemen we een `Blad`. Zo is er voor een groepsindeling maar één mogelijke representatie. Op deze manier kunnen we symmetrie vermijden. Dit nuanceverschil tussen de relaties is visueel weergegeven in figuur 3.2.

Listing 3.6: Definiëren `Wortel Blad` en `GestructureerdSamen`.

```
Wortel(Student)
Blad(Student)
GestructureerdSamen(Student, Student)
```

Om de aanpassingen concreet te implementeren, voegen we eerst drie nieuwe relaties toe aan de `vocabulary`: `Wortel`, `Blad` en `GestructureerdSamen`. Deze zijn te zien in listing 3.6.

Vervolgens voegen we aan de `theory` de volgende definities en regels toe.

- Een student is enkel een wortel wanneer deze het kleinste studentnummer heeft van alle andere studenten waarmee het samen in een groep zit.
- Een blad is een student die geen wortel is.
- Twee studenten voldoen aan `GestructureerdSamen` wanneer de eerste student een wortel is, de tweede een blad en ze allebei voldoen aan `Samen`.
- Twee studenten zijn `Samen` in beide richtingen, en wanneer twee studentenparen een gemeenschappelijke student hebben, zijn de andere twee studenten ook samen.

Listing 3.7: Definities voor `Wortel Blad` en `GestructureerdSamen`.

```
{
    !x[Student]: Wortel(x) <- x = min{y[Student]: Samen(x,y): y}.
}
{
    !x[Student]: Blad(x) <- ~Wortel(x).
}
{
    !x[Student],y[Student]:GestructureerdSamen(x,y) <- Samen(x,y) & Wortel(x)
        & Blad(y).
}
!x[Student],y[Student]: Samen(x,y) => Samen(y,x).
!x[Student],y[Student],z[Student]: Samen(x,y) & Samen(y,z) => Samen(x,z).
```

Merk op dat om aan de eerste regel te voldoen, aan `Samen(x,x)` voldaan moet zijn. Elke student moet dus ook bij zichzelf zitten.

We kunnen vervolgens het aantal groepen vastleggen door te zeggen hoeveel wortels een oplossing mag bevatten: elke groep heeft namelijk maar één wortel, en omgekeerd. Ook is de spreiding op een groepsgrootte eenvoudig te beperken: door vast te leggen hoeveel bladeren een wortel minimaal en maximaal mag hebben. Voor deze regel introduceren we het hulppredicaat `Aantal`, welke voor elke student het aantal andere studenten nagaat waarmee het aan `GestructureerdSamen` voldoet. De implementatie hiervan is te vinden in listing 3.8.

Listing 3.8: Vastleggen aantal groepen en hun spreiding.

```
#{x[Student]: Wortel(x)} = AantGroepen.
!x[Student]: Aantal(x) =
    #{y[Student]: GestructureerdSamen(x,y) | GestructureerdSamen(y,x)}.
!x[Student]: Wortel(x) => MinGrootte <= Aantal(x) <= MaxGrootte.
!x[Student]: Blad(x) => Aantal(x) = 1.
```

3.2.4 Precisie per postcode

De vraag kan gesteld worden of het niet overbodig is om met volledige postcodes te rekenen. IDP is namelijk ontworpen om logisch te redeneren, niet om veel rekenwerk te verrichten. Daarom is er geëxperimenteerd met de hoeveelheid cijfers die gebruikt worden per postcode. Op deze manier is er onderzocht of het IDP-systeem sneller kan rekenen wanneer we onze postcodes afkappen op drie of twee cijfers.

Door drie cijfers precisie te kiezen, zouden bijvoorbeeld 2861 en 2860 allebei afgekapt worden naar 286. Dit is een acceptabele afronding, aangezien het buurgemeenten zijn. Bij twee cijfers zouden bijvoorbeeld 2860 en 2800 allebei 28 worden, wat al ruwer afgerond is, maar misschien wel sneller kan rekenen.

Om te beslissen wat de beste precisie is voor de postcodes, is hiervoor een test uitgevoerd. Deze gaat voor twee, drie en vier cijfers per postcode na wat de gevonden waarde is na drie en twintig minuten zoeken. De gevonden waarden kunnen geïnterpreteerd worden als “de totale afstand tussen studenten in alle groepen opgeteld”. Een kleinere waarde is dus een betere oplossing. Deze waarde wordt berekend door per groep het verschil in postcode tussen de eerste student van een groep en alle andere studenten van dezelfde groep te berekenen, en op te tellen. Logischerwijs verschilt deze waarde per aantal cijfers, dus hebben we na het zoeken telkens de gevonden oplossing herberekend met vier cijfers als postcode.

De resultaten van deze test zijn te vinden in tabel 3.3. Uit deze waarden blijkt dat we bij drie cijfers over het algemeen de beste afstand bekomen. Bij twee cijfers rekent het IDP-systeem sneller, maar worden er grovere fouten gemaakt waardoor er meer spreiding is op de bekomen reële afstand. Bij vier cijfers rekent het IDP-systeem trager, waardoor er minder modellen op eenzelfde tijd kunnen gegenereerd worden, en er dus minder vooruitgang gemaakt wordt. De postcodes definiëren met drie cijfers lijkt de gulden middenweg tussen precisie en rekensnelheid te zijn.

Tabel 3.3 Bekomen afstanden na rekenen, na afkappen van postcode.

aantal cijfers	na 3 min	reële na 3 min	na 20 min	reële na 20 min
2	601	61255	575	58847
3	6272	62670	4716	47102
4	63807	63807	49641	49641

3.2.5 Provinciegrenzen

Het volgende probleem is dat de afstand inschatten op basis van postcodes enkel werkt in één “zone”. Een zone wordt aangeduid door het eerste cijfer van de postcode. Zo ligt bijvoorbeeld 1980 in Vlaams-Brabant, maar ligt 2000 in Antwerpen centrum. Volgens de huidige oplossingsmethode zou dit dus een verschil zijn van 20, wat niet representatief is voor de werkelijke afstand. Om hier rekening mee te houden zijn de postcodes in twee opgesplitst: het eerste cijfer vormt het zonegetal, en de overige drie cijfers vormen de “zonespecifieke postcode”. Bijvoorbeeld, voor een student wonende te 3150 Haacht wordt zijn zonegetal `3` en zijn aangepaste postcode `150`. Dit is

eenvoudig te verwezenlijken door onze vorige functie `Woont` aan te vullen met een nieuwe functie `WoontZone`. `WoontZone` mapt elke student op een zonegetal, en `Woont` mapt elke student op een zonespecifieke postcode.

Verder werken de postcodes hetzelfde als ervoor, behalve dat ze nu voornamelijk werken binnen één zone. Dit betekent dat wanneer we als precisie per postcode drie cijfers kiezen, we één cijfer zonegetal en twee cijfers "postcode" hebben. Oplossingen met studenten die niet in éénzelfde zone zitten maar wel in één groep worden bestraft. In listing 3.9 berekenen we een constante `NietInZone` die weergeeft hoeveel studenten in een andere zone zitten dan hun wortel. Deze constante kunnen we dan vermenigvuldigen met een factor en vervolgens optellen bij `TotaleAfstand` om de de som te bekomen. In de berekening van de `TotaleAfstand` moeten we er wel rekening mee houden dat we niet meer de afstand berekenen tussen studenten die zich in verschillende zones bevinden.

Listing 3.9: Berekening afstraffing van Zone.

```
NietInZone = #{x[Student] y[Student]: x < y & Samen(x,y)
    & WoontZone(x) ~= WoontZone(y)}.
```

Deze implementatie houdt geen rekening met de afstand tussen studenten in aparte zones. Er wordt enkel geprobeerd worden om optimaal een cluster te vormen per zone per groep, zonder te kijken naar hoe ver deze clusters over zonegrenzen heen van elkaar liggen.

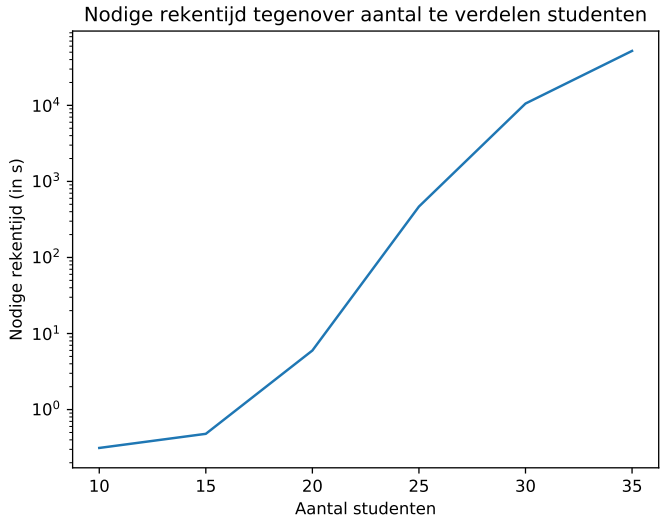
3.2.6 Middelbare school

We kunnen onze oplossingen nog verder verbeteren door rekening te houden met de middelbare scholen waar de studenten afgestudeerd zijn. Dit heeft namelijk twee voordelen. Ten eerste, studenten van dezelfde school wonen vaak bij elkaar. Wanneer we dus zoveel mogelijk studenten van dezelfde school in één groep plaatsen, zal de afstand tussen die studenten vanzelf al klein zijn. Ten tweede zullen hierdoor bij de incrementele indeling (zie sectie 3.3) minder studenten expliciet willen veranderen van groep. De kans is namelijk groot dat ze bij de mensen willen zitten die ze kenden van het middelbaar, waar op deze manier dus al aan voldaan is.

Door een relatie toe te voegen, `School`, die elke student op een middelbare school mapt kunnen we in onze `theory` een constante `NietSamenSchool` berekenen. Deze constante geeft weer hoeveel studenten van dezelfde middelbare school komen, maar niet in dezelfde groep zijn ingedeeld. Als we vervolgens `NietSamenSchool` vermenigvuldigen met een factor, en optellen bij `TotaleAfstand` en `NietInZone`, bekomen we onze uiteindelijke te minimaliseren waarde. Dit wordt weergegeven in listing 3.10.

Listing 3.10: Berekening van `SamenSchool`.

```
NietSamenSchool = #{x[Student] y[Student]: x < y & ~Samen(x,y)
    & School(x) = School(y)}.
Totaal = TotaleAfstand + NietInZone * 10 + NietSamenSchool * 1000.
```



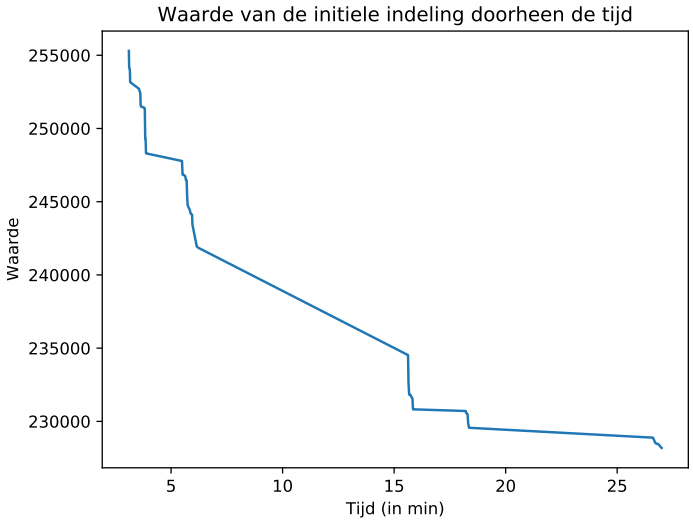
Figuur 3.3: Duratie van zoeken van een optimale oplossing bij bepaald aantal studenten.

3.2.7 Rekentijd

Omdat het de bedoeling is om het maken van een initiële oplossing in een interactieve omgeving te kunnen gebruiken, moet het snel genoeg kunnen werken. Het probleem ligt hier echter bij de rekentijd: het zoeken van de optimale oplossing duurt te lang. In figuur 3.3 is uitgezet hoe lang het duurt om dé optimale oplossing te vinden volgens onze implementatie, telkens voor een dataset van 10, 15, 20, 25, 30 en 35 studenten. Hier is duidelijk zichtbaar dat afhankelijk van de hoeveelheid studenten de nodige tijd enorm stijgt. Aangezien het IDP-systeem zo'n 127 keer meer tijd nodig had bij 30 dan bij 20 studenten, kunnen we concluderen dat het quasi onmogelijk zal zijn om de optimale oplossing van 150 studenten te vinden.

Om hier een oplossing voor te bieden, gaan we niet telkens zoeken naar dé beste oplossing die we in het algemeen kunnen vinden, maar wel naar de beste oplossing die we kunnen vinden in een bepaalde tijd. Hiervoor kunnen we in het IDP-systeem de `stdoptions.mxtimeout` optie gebruiken. Door deze in te stellen op 300 seconden, zoekt het IDP-systeem 5 minuten lang oplossingen. Eens de tijd voorbij is, geeft het de beste gevonden oplossing terug.

Natuurlijk, dit houdt in dat we nooit echt de ideale indeling vinden voor een gegeven set studenten. In grafiek 3.4 is de gevonden afstand geplot tegenover de tijd. Hierop is te zien dat na ongeveer vijf minuten, we aan een oplossing zitten met een kwaliteit van ongeveer 25% van het verschil tussen de beginkwaliteit en de kwaliteit van de oplossing na een halfuur zoeken. Bovendien is het uiteraard onnodig deze initiële indeling volledig te optimaliseren, aangezien in de volgende stap (de incrementele indeling) de groepsindelingen nog gemanipuleerd gaan worden. Dit in rekening houdende, is een oplossing na vijf minuten dus voldoende goed.



Figuur 3.4: Grafiek die tijd tegenover kwaliteit van de oplossing van de testdata weergeeft.

3.2.8 De implementatie

Het volledige IDP bestand met de implementatie van de initiële indeling is terug te vinden in appendix A.1. Om samen te vatten, de implementatie werkt door studenten aan te wijzen die in dezelfde groep zitten, aan de hand van `Samen`. Omdat deze manier van werken symmetrie introduceert, maken we gebruik van `GestructureerdSamen`. `GestructureerdSamen(x,y)` is enkel voldaan indien `x` een wortel is, `y` een blad en `x` en `y` voldoen aan `Samen`. Een wortel is de student met het kleinste studentnummer die in een groep zit, en de bladeren zijn alle andere studenten.

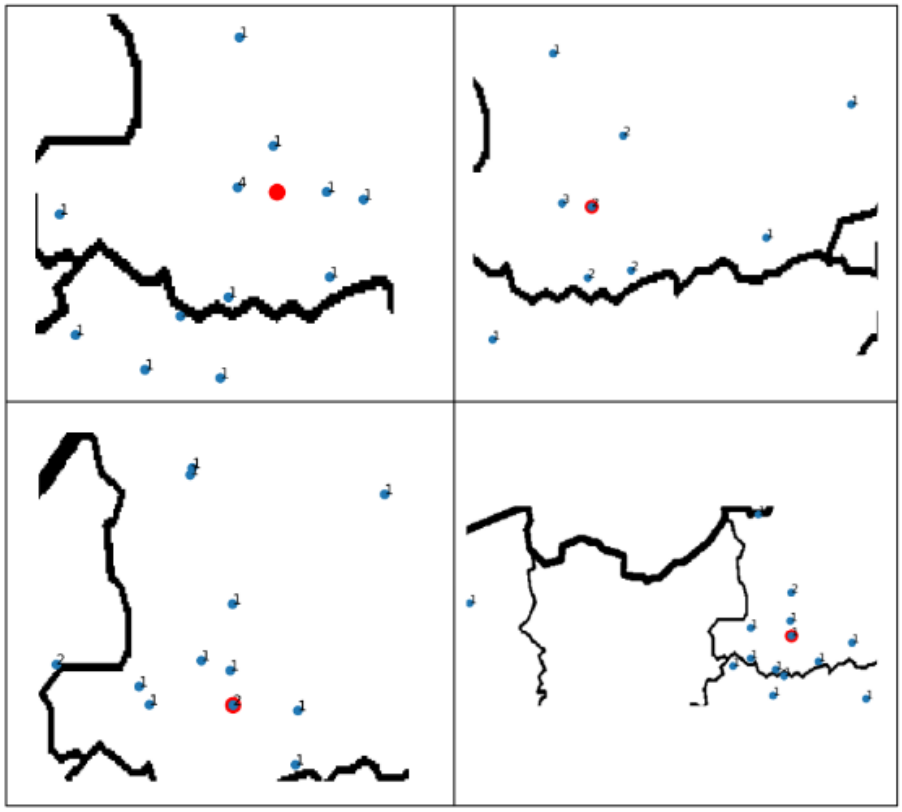
Zoals besproken, houden we rekening met volgende drie zaken om de kwaliteit van een oplossing te bepalen.

1. de afstand in elke groep, op basis van de aangepaste postcode;
2. de hoeveelheid studenten die in een andere zone zitten dan de rest van hun groepsgenoten;
3. de hoeveelheid studenten die van dezelfde school komen maar in een andere groep zitten.

Deze criteria stellen samen de kwaliteit van een oplossing voor. Listing 3.11 toont de implementatie.

Listing 3.11: Berekenen van de kwaliteit van een oplossing

```
1 Afstand = sum{x[Student] y[Student]: x < y & Samen(x,y) &
2             WoontZone(x) = WoontZone(y): abs(Woont(x) - Woont(y))}.
3 NietInZone = #{x[Student] y[Student]: x < y & Samen(x,y)
4             & WoontZone(x) ~= WoontZone(y) }.
5 NietSamenSchool = #{x[Student] y[Student]: x < y & ~Samen(x,y) &
6                  School(x) = School(y)}.
7
8 Totaal = Afstand + NietSamenSchool * 1000 + NietInZone * 100.
```



Figuur 3.5: Visualisatie van de indeling van vier groepen op een kaart van België.

`Afstand` wordt berekend als de som van het verschil van postcodes van alle studenten die in dezelfde groep zitten en in dezelfde zone wonen.

`NietSamenSchool` bevat de studenten die niet in dezelfde groep zitten, maar wel van dezelfde school komen.

`NietInZone` bevat de studenten die in dezelfde groep zitten, maar in een andere zone wonen.

Om zo min mogelijk groepen te hebben met veel studenten in verschillende zones wordt dit afgestraft met 100 per student in een andere zone. Vervolgens wordt `NietSamenSchool` afgestraft met 1000 voor elke student van een school die niet bij een andere student van die school zit. Als laatste term volgt dan de afstand tussen alle studenten van de groepen.

Dit zorgt voor oplossingen zoals te zien in figuur 3.5.

3.3 Incrementele indeling

In de vorige sectie hebben we de eerste groepsindeling gemaakt. Het is nu de bedoeling dat in een volgende stap voorkeuren van studenten hieraan toegevoegd kunnen worden. Het IDP-systeem

moet dan de indeling herrekenen om zo een volgende indeling uit te komen, rekening houdend met de voorkeuren. Aan deze volgende indeling kunnen we dan verder weer voorkeuren toevoegen, om zo wederom een nieuwe indeling te berekenen. Dit proces kunnen we herhalen tot we tevreden zijn met de indeling. Op deze manier kan de gebruiker via het IDP-systeem incrementeel een oplossing verbeteren, door telkens nieuwe voorkeuren in te geven en de groepsindeling te herrekenen. We gaan dus een IDP bestand ontwerpen dat de voorkeuren van studenten en de vorige oplossing als input heeft, om vervolgens een nieuwe oplossing als output te geven.

In sectie 1.3 werd vermeld dat er drie vormen van voorkeuren zijn die ingegeven moeten kunnen worden als inputs aan het IDP-systeem. Bij deze inputs komen er nog drie bij: de eerdere oplossing, de minimum- en maximumgrootte van de groepen, en het aantal gewenste groepen. In totaal zijn er dus zes verschillende inputs om aan het systeem te geven:

1. de vorige groepsindeling;
2. de minimum- en maximumgrootte die een groep mag hebben;
3. het aantal gewenste groepen;
4. alle studenten die samen willen zitten, met de student waar ze bij willen;
5. alle studenten die niet samen willen zitten, met de student waar ze niet bij willen;
6. alle studenten die in een andere groep willen zitten, met in welke groep ze willen.

De bedoeling van de indeling is om aan zoveel mogelijk voorkeuren te voldoen, met zo min mogelijk wijzigingen aan de groepsindeling van de studenten. Om dit te bereiken zal onze te minimaliseren waarde een som zijn van:

- het aantal studenten die niet bij een student zitten waar ze wel bij wilden;
- het aantal studenten die wel bij een student zitten waar ze niet bij wilden;
- het aantal studenten die niet in een groep zitten waar ze wel in wilden;
- het aantal studenten die in een andere groep zitten dan in de vorige verdeling.

Deze voorkeuren worden gemodeleerd als zachte beperkingen.

Om te voorkomen dat ons IDP bestand moeilijk leesbaar wordt door teveel informatie, maken we gebruik van twee `vocabularies`: één voor de input, en één voor de output. De `vocabulary` in listing 3.12 geeft weer hoe we de inputs kunnen definiëren. `Student`, `Getal` en `Groep` definiëren we als natuurlijke getallen, en de minimum- en maximumgrootte als constante van het type `Getal`. Aan de hand van de functie `HuidigeGroep` gaan we een vorige indeling aanbrengen aan het systeem. Deze functie mapt elke student op zijn vorig groepsgetal. Als de vorige indeling de initiële indeling was, dan moeten we een tussenstap maken om elke `Wortel` een groepsgetal te geven aangezien er daar niet met groepen gewerkt werd. De relaties `WilSamen` en `WilNietSamen` laten ons toe om te definiëren welke studenten wel en niet samen willen. `WilInGroep` is een relatie om aan te duiden wanneer een student in een specifieke groep wil.

Listing 3.12: Vocabulary met alle input.

```
1  vocabulary V_data {
2      type Student isa nat
3      type Getal isa nat
4      type Groep isa nat
5
6      MinGrootte: Getal
7      MaxGrootte: Getal
8
9      //De vorige indeling
10     Huidige(Student): Groep
11
12     //De voorkeuren om in te geven
13     WilSamen(Student, Student)
14     WilNietSamen(Student, Student)
15     WilInGroep(Student, Groep)
16
17     //De nieuwe indeling
18     VolgendeGroep(Student): Groep
19 }
```

Belangrijk om op te merken is dat we bij deze oplossing een eerdere oorzaak van symmetrie herintroduceren. Er is namelijk weer een groepsgetal aanwezig, iets wat in de initiële indeling voor 3628800 extra oplossingen zorgde. Het verschil met de initiële indeling is dat het omwisselen van groepsgetallen (bv. alle studenten van groep 1 en groep 2 wisselen) hier niet leidt tot equivalente oplossingen: als alle studenten die eerder in groep 2 zaten nu in groep 1 terecht komen, zou dit voor een extra kost zorgen (zie verder). Hierdoor gaat het IDP-systeem vroegtijdig stoppen met het berekenen van zo'n oplossing, en wordt symmetrie vermeden.

Deze symbolen kunnen we vervolgens waarden toekennen, zoals beschreven in listing 3.13. Belangrijk hierbij is dat de functie `HuidigeGroep` compleet is: er moeten met andere woorden evenveel studenten inzitten als dat er gedefiniëerd zijn. Dit komt doordat we een functie doorgaans ofwel volledig moeten definiëren, ofwel niet. Stel dat we een nieuwe student willen toevoegen die niet in een eerdere groepsindeling aanwezig was, zullen we de student zelf aan een groep moeten toewijzen en dit aanvullen in `HuidigeGroep` om te voorkomen dat we een waarde missen in de functie.

In listing 3.13 wil onder andere student 4 bij student 5. Verder is er geen enkele student die expliciet niet bij een andere wil. Merk op dat wanneer er geen voorkeur van een bepaald type voorkomt, we deze relatie leeg moeten laten, maar niet mogen weglaten. Indien we de relatie zouden weglaten uit onze `structure`, dan zou het IDP-systeem proberen om deze zelf optimaal in te vullen, en zou het dus voorkeuren kunnen "verzinnen".

Listing 3.13: Structure met de invulling van de inputs.

```
1  structure S : V_data {
2      Groep = {1..10}
```

```
3       Student ={1..152}
4       Getal = {0..10000000}
5
6       MinGrootte = 14
7       MaxGrootte = 16
8
9       HuidigeGroep = { 1->2; 2->3; 3->4; ...; 152->6}
10
11      WilSamen = {4,5; 120, 54;}
12      WilNietSamen = {}
13      WilInGroep = {123, 3;}
14  }
```

De volgende stap is het `vocabulary` waar we de predicaten in definiëren die het IDP-systeem zelf moet aanvullen, zoals onder andere de nieuwe indeling. Dit wordt getoond in listing 3.14. Eerst wordt de vorige `vocabulary` bij de nieuwe gevoegd. Zo zitten alle vorige predicaten ook in deze `vocabulary`, maar is het leesbaarder omdat ze gescheiden zijn. De relatie `GroepGrootte` mapt op elke groep de grootte van die groep. De constanten op lijn negen tot en met elf in de vocabulary houden bij aan hoeveel voorkeuren niet voldaan is. Op deze manier kunnen we hier zicht op krijgen. De constante `TotWisselGroep` houdt bij hoeveel studenten in een andere groep zitten dan in de vorige indeling. Deze constante minimaliseren we mee, om ervoor te zorgen dat we de vorige indeling zo min mogelijk veranderen. De volgende vier relaties houden bij voor welke studenten deze voorkeuren niet voldaan zijn of welke studenten in een andere groep zitten dan in de vorige indeling. De functie `VolgendeGroep` gaat voor elke student het groepsgetal, van de groep waar deze inzit, bevatten. De constante `Totaal` is de totale te minimaliseren kostterm, welke gelijk is aan een gewogen som van alle `Tot`-constanten.

Listing 3.14: Vocabulary met alle output en hulppredicaten.

```
1   vocabulary V_optimization{
2       extern vocabulary V_data
3
4       GroepGrootte(Groep): Getal
5
6       TotUnsatSamen: Getal
7       TotUnsatNietSamen: Getal
8       TotUnsatHuidigeGroep: Getal
9       TotWisselGroep: Getal
10
11      UnsatSamen(Student, Student)
12      UnsatNietSamen(Student,Student)
13      UnsatHuidigeGroep(Student, Groep)
14      WisselGroep(Student)  //Studenten die niet meer in dezelfde groep zitten.
15
16      VolgendeGroep(Student): Groep
17
```

```
18      Totaal : Getal
19  }
```

Tot slot volgen onze `theory` en `term` nog. Deze zijn beide te lezen in listing 3.15. In de `theory` berekenen we per groep de grootte, om hiermee vast te leggen hoe groot een groep minimaal en maximaal mag zijn. We gaan voor ieder studentenpaar na of er een voorkeur is en of aan deze voorkeur ook effectief voldaan is. Ook berekenen we hoeveel voorkeuren niet voldaan zijn, om op te nemen in de berekening van de totale kostterm. Deze kostterm is een optelling van elke "`TotUnsat`" vermenigvuldigd met een prioriteitswaarde. Op deze manier is het onder andere mogelijk om studenten die niet samen willen zitten voorrang te geven over studenten die wel samen willen zitten, door deze voorkeur een groter gewicht te geven in de berekening van de kostfunctie.

Listing 3.15: `Theory` en `Term` van de incrementele indeling.

```
1   theory T : V_optimization {
2       !g[Groep]: GroepGrootte(g) = #{x[Student]: VolgendeGroep(x) = g}.
3
4       !g[Groep]: MinGrootte =< GroepGrootte(g) =< MaxGrootte.
5
6       {
7           !x[Student] y[Student]: UnsatSamen(x,y) <- WilSamen(x,y)
8               & VolgendeGroep(x) ~= VolgendeGroep(y).
9       }
10      {
11          !x[Student] y[Student]: UnsatNietSamen(x,y) <- WilNietSamen(x,y)
12              & VolgendeGroep(x) = VolgendeGroep(y).
13      }
14      {
15          !x[Student]: WisselGroep(x) <- HuidigeGroep(x) ~= VolgendeGroep(x).
16      }
17      {
18          !x[Student] g[Groep]: UnsatHuidigeGroep(x,g) <- WilInGroep(x,g)
19              & VolgendeGroep(x) ~= g.
20      }
21
22      TotWisselGroep = #{x[Student]: Wisselgroep(x)}.
23      TotUnsatSamen = #{x[Student] y[Student]: UnsatSamen(x,y)}
24      TotUnsatNietSamen = #{x[Student] y[Student]: UnsatNietSamen(x,y)}
25      TotUnsatHuidigeGroep = #{x[Student] g[Groep]: UnsatWilInGroep(x,g)}
26
27      Totaal = 5*TotUnsatSamen + 10*TotUnsatNietSamen + 15*TotUnsatHuidigeGroep
28          + 2*TotWisselGroep.
29  }
30  term t: V_optimization {
31      Totaal
32  }
```

In deze implementatie bepalen we voor welke studenten niet aan hun voorkeur voldaan is. Er kan de vraag gesteld worden of dit niet overbodig is, aangezien we deze "`Unsat`"-relaties zouden kunnen weglaten en de "`Tot`"-relaties kunnen herschrijven om de regels van de definities te bevatten. Daarbij zouden we bijvoorbeeld ook Python kunnen gebruiken om na te gaan voor welke studenten dit zo is. In tabel 3.4 is te zien wat de snelheid is van het IDP-systeem bij het uitrekenen van indelingen met en zonder deze 4 relaties. Het is duidelijk dat het verlies aan snelheid beperkt blijft. Verder is het eenvoudiger om dit na te kijken in IDP zelf dan het na te kijken in een andere taal: het zijn immers maar vier extra relaties, tegenover een tiental lijnen denk- en programmeerwerk in een andere taal.

Tabel 3.4 Rekentijd incrementele indeling met of zonder unsatrelaties.

aantal voorkeuren	10	20	30
met unsatrelaties	4.091s	14.995s	50m1.566s
zonder unsatrelaties	4.095s	14.999s	49m59.492s

Na samen te zitten met een domeinexpert, mevrouw Koppen, om deze implementatie te valoriseren, is gebleken dat de manier waarop prioriteiten geïmplementeerd worden beter kan. Momenteel gebruikt het IDP-systeem "globale" prioriteiten: één voor alle studenten die samen willen, één voor alle studenten die niet samen willen en één voor alle studenten die in een bepaalde groep willen. Bijvoorbeeld, student A die bij student B wil heeft hetzelfde gewicht als student C die bij student D wil, namelijk de globale prioriteit. Mevrouw Koppen bemerkt dat hier geen rekening gehouden wordt met de reden waarom deze studenten samen willen. Als A en B graag samen willen omdat ze dan kunnen carpoolen, zou dit een grotere prioriteit moeten hebben dan wanneer C en D samen willen zitten omdat ze elkaar ooit eens hebben leren kennen op een kamp 5 jaar geleden.

Gelukkig is dit dankzij de structuur van het IDP-systeem geen moeilijke aanpassing. In listing 3.16 is weergegeven wat er in de `theory` moet veranderen om deze prioriteiten toe te laten. Allereerst definiëren we het type `Prioriteit` als een `Getal`. Dit zorgt ervoor dat een prioriteit altijd een getalwaarde tussen 0 en 10000000 moet hebben (zie listing 3.13). Vervolgens definiëren we partieelfuncties, die voorkeuren mappen op prioriteitswaardes.

Listing 3.16: Aanpassing aan vocabulary om `Prioriteit` toe te voegen.

```
type Prioriteit isa Getal

partial PrioriteitWilSamen(Student,Student): Prioriteit
partial PrioriteitWilNietSamen(Student,Student): Prioriteit
partial PrioriteitWilInGroep(Student,Groep): Prioriteit
```

Tot slot hoeven we in onze `theory` aan te passen hoe we onze kostterm berekenen. We tellen nier meer hoeveel voorkeuren er onvoldaan zijn, maar we berekenen wel de som van de prioriteiten van al de onvoldane voorkeuren. Hierdoor is het ook niet meer nodig om de globale prioriteitswaarden te gebruiken die in de berekening van `Totaal` stonden. Deze aanpassingen zijn te vinden in listing 3.17.

Listing 3.17: Aanpassing aan theory om `Prioriteit` toe te voegen.

```
1     TotWisselGroep = #{x[Student]: WisselGroep(x)}.
2     TotUnsatSamen = sum{x[Student] y[Student]: UnsatSamen(x,y):
3         PrioriteitWilSamen(x,y)}.
4     TotUnsatNSamen = sum{x[Student] y[Student]: UnsatNietSamen(x,y):
5         PrioriteitWilNietSamen(x,y)}.
6     TotUnsatInGroep = sum{x[Student] g[Groep]: UnsatInGroep(x,g):
7         PrioriteitWilInGroep(x,g)}.
8
9     Totaal = TotUnsatSamen + TotUnsatNSamen + TotUnsatInGroep + TotWisselGroep.
```

Om samen te vatten, bij de incrementele indeling proberen we telkens om een nieuwe indeling te maken die zo min mogelijk verschilt van de vorige indeling. Hierbij kunnen we voorkeuren van studenten ingeven en, bij elke voorkeur, een individuele prioriteitswaarde. Het volledige IDP bestand is te vinden in appendix A.2.

Hoofdstuk 4

Pyidp3

In dit hoofdstuk wordt een korte uitleg gegeven over Pyidp/Pyidp3, werking en verschillen. Voor een meer uitgebreide tekst, zie appendix D.

Zoals eerder vermeld in de inleiding hebben we nood aan een interface rond het IDP-systeem, om deze te kunnen gebruiken vanuit Python 3.

Vennekens (2015) heeft hiervoor de Pyidp[1] API ontwikkeld voor Python 2. Deze heeft als doel om toe te laten in IDP te redeneren, maar dan op een imperatieve manier in Python. Hierdoor zouden programmeurs van de voordelen van het IDP-systeem kunnen genieten, zonder expliciet de nieuwe syntax ervan aan te hoeven leren. Het voornaamste nadeel van deze API, is dat het enkel ondersteund wordt voor Python 2. Verder ontbreken er ook bepaalde functionaliteiten (zie 4.2.2) om het voor dit werk bruikbaar te maken.

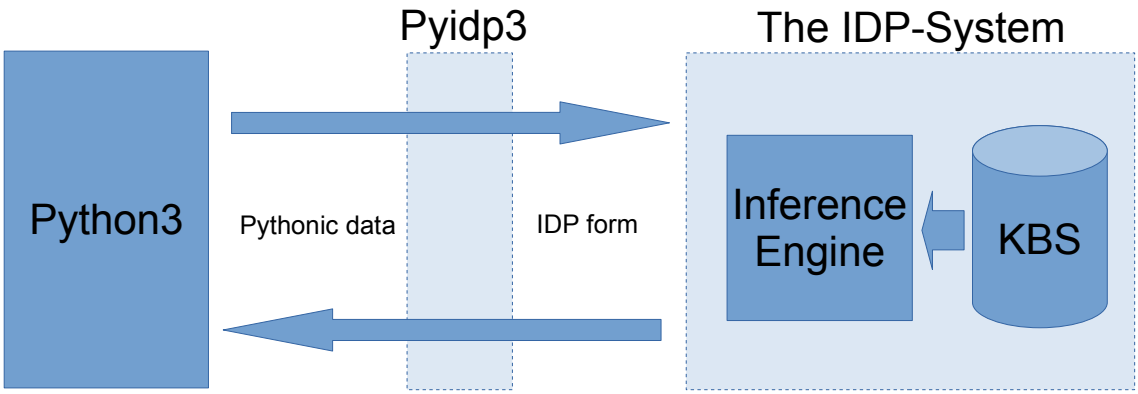
Voor deze thesis hebben we de Pyidp3 API ontwikkeld. Dit is in de eerste plaats een rechtstreekse port van Pyidp, die we vervolgens verder hebben uitgebreid met extra functionaliteiten, “Quality Of Life” (QOL), bugfiksen en documentatie. Hoe deze overzetting gebeurd is en welke extra functionaliteiten zijn toegevoegd, staat beschreven in 4.2.1. Uitleg over de QOL, bugfiksen en documentatie zijn te vinden in appendix D.

Zoals vermeld is het doel van Pyidp3 om een brug te vormen tussen Python 3 en het IDP-systeem. Pyidp3 zal objecten en data van Python kunnen omzetten in een structuur die verstaanbaar is voor het IDP-systeem. Deze structuur is dezelfde als die van een IDP bestand. Vervolgens kan Pyidp3 de output van het IDP-systeem lezen, interpreteren en terug omzetten in Pythonformaat. Dit is weergegeven in figuur 4.1.

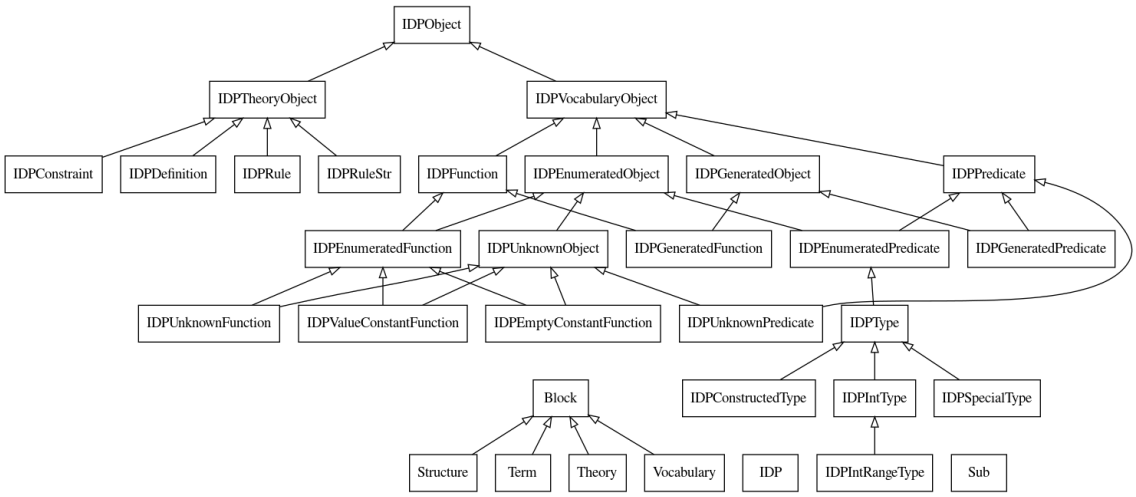
4.1 Werking Pyidp/Pyidp3

Fundamenteel werkt Pyidp/Pyidp3 heel eenvoudig: het laat toe om al de verschillende vormen van IDP symbolen in te geven via Python, waarna het elk van deze symbolen vertaalt naar de string die de “IDP vorm” van het symbool voorstelt. Wanneer we deze strings van al deze symbolen

[1] `https://bitbucket.org/joostv/pyidp`



Figuur 4.1: Schema van Python, Pyidp3 en IDP.



Figuur 4.2: Klassediagramma van Pyidp3.

samenzetten, hebben we een uitgewerkt IDP bestand. Vervolgens kunnen we het IDP bestand doorsturen naar de IDP executable (met een Unix `pipe`), en de output uitlezen en verwerken.

Om Pyidp te gebruiken dient de gebruiker enkel maar één object aan te maken: het `IDP` object. Vervolgens is het mogelijk om via dit `IDP` object alles te definiëren wat in de `vocabulary`, `structure` en `theory` moet komen. Hiervoor bevat de IDP klasse de volgende methodes: `Constant`, `Constraint`, `Define`, `Function`, `Predicate` en `Type`.

Aan de hand van deze zes methoden kunnen volledige bestanden voor het IDP-systeem ontworpen worden. `IDP` maakt elke keer dat zo'n methode wordt gebruikt, een nieuw `IDPObject` aan (of beter gezegd, een kindklasse hiervan, dat een enger type voorstelt). Een `IDPObject` is een Pythonische representatie van een predicaat, definitie, functie, type of constante in IDP. In figuur 4.2 is een volledig overzicht te zien van alle welke klassen die overerven van `IDPObject`.

Op het eerste niveau erft de `IDPObject`-klasse over in objecten die in de `theory` komen, zoals definities en beperkingen, en objecten die het `vocabulary` komen (en eventueel ook `structure`), zoals functies, predicaten, types en constraints. Het `IDPTheoryObject` heeft een abstracte me-

Tabel 4.1 IDP objecten en hun mogelijke invullingen

Mogelijkheid	`Constant`	`Type`	`Predicate`	`Function`	Vervat in object:
waarde toekennen	mogelijk	mogelijk	mogelijk	mogelijk	`IDPEnumerated-Object`
returnwaarde	geen	mogelijk	geen	verplicht	`IDPFunction`
argumenten	geen	geen	verplicht	mogelijk	`IDPFunction`, `IDPPredicate`

thode `in_theory` waarin het object omgezet wordt naar een vorm die in het `theory`-blok van een IDP-systeem kan. Gelijkaardig heeft het `IDPVocabularyObject` abstracte methodes `in_voc` en `in_struct`, die het object omzetten naar respectievelijk een vorm voor het `vocabulary` en een vorm voor de `structure`.

Alle kindklassen van deze objecten verschillen voornamelijk in de vorm waarop ze deze methoden invullen. Bijvoorbeeld, elke beperking in IDP moet eindigen op een punt. Een definitie is een verzameling beperkingen, die tussen twee accolades staan. Beperkingen en definities moeten dus op een andere manier gevormd worden, en dit gebeurt in de `in_theory` methode.

Bij `IDPVocabularyObject` is meer variatie mogelijk. Zo willen we in sommige gevallen een waarde toekennen aan een object (Bv "Student = $\{1..152\}$") en in sommige gevallen net niet (wanneer we IDP een functie zelf willen laten invullen). Verder hebben sommige functiesymbolen een returnwaarde (wanneer er een ":" is) en anderen niet. Tot slot is er een verschil in het gebruik van argumenten. Tabel 4.1 geeft een overzicht van alle mogelijke invullingen van de `Constant`-, `Type`-, `Predicate`- en `Function`-objecten.

De `IDP`-klasse houdt ook drie soorten `Block` objecten bij: een `Structure`, een `Theory` en een `Vocabulary` object. Deze objecten stellen van de gelijknamige blokken van het IDP-systeem voor. In de `Block` klasse zijn een aantal abstracte methodes gedefinieerd om zo'n blokobject als string voor te stellen, bestaande uit een header, een begin, de inhoud en een einde. Elke blok past indien nodig deze methodes aan naar zijn eigen vorm. Zo verschillen de drie blokken bijvoorbeeld in hun header: een vocabulary start met "vocabulary `vocnaam`", een structure met "structure `structnaam`: `vocnaam`" en een theory met "theory `theorynaam`: `vocnaam`". Verder introduceert de klasse `Block` ook de `show()` methode, die een blok omzet in stringvorm aan de hand van de net vermelde methoden (zoals te zien in 4.1).

Listing 4.1: `show()` methode van de `Block` klasse.

```
def show(self, objects):
    return (self.header() + self.begin() + self.content(objects)
        + self.end())
```

Door bij het genereren van een blok een lijst van `IDPobjecten` mee te geven, kan de `Block`-klasse, met behulp van de methode in `content()`, alle strings in IDP vorm genereren die in het blok moeten.

Ter verduidelijking bekijken we een klein voorbeeld, weergegeven in listing 4.2. In dit voorbeeld

lossen we volgende letterpuzzel op: ”AI + BA = CDE”. De bedoeling is om elke letter een waarde te geven, en zo de formule te doen uitkomen. Bijkomend, mogen verschillende letters niet dezelfde waarde hebben, moeten klinkers een even waarde en medeklinkers een oneven waarde krijgen en mogen de eerste letters niet nul zijn.

Hiervoor definiëren we eerst een `Type Decimaal`, welke een waarde zal hebben tussen 0 en 9. Verder moeten we voor iedere letter een constante definiëren, en het predicaat `Even(Decimaal)`. De constraints zijn hier al meteen ingegeven in de vorm van het IDP systeem, om dit te signaleren is het tweede argument van `Constraint()` telkens de boolean `True`. Indien we deze niet op `True` zetten, denkt Pyidp dat we onze constraint in het Pyidp formaat gemaakt hebben, en dat deze nog moet omgezet worden. Hierover meer aan het einde van deze subsectie.

Listing 4.2: Oplossing van de cde letterpuzzel in Pyidp3.

```
1  from pyidp3.typedIDP import *
2
3  idp = IDP("pad/naar/idp")
4
5  # Decimaal en letters definieren
6  idp.Type("Decimaal", list(range(0,9)))
7  idp.Constant("A: Decimaal")
8  idp.Constant("B: Decimaal")
9  idp.Constant("C: Decimaal")
10 idp.Constant("D: Decimaal")
11 idp.Constant("E: Decimaal")
12 idp.Constant("I: Decimaal")
13
14 # Definieren welke decimalen even zijn.
15 idp.Predicate("Even(Decimaal)", [0, 2, 4, 6, 8])
16
17 # Geen twee letters mogen dezelfde waarde hebben.
18 idp.Constraint("A ~= B & A ~= C & A ~= I & A ~= D & A ~= E & "
19                "B ~= C & B ~= I & B ~= D & B ~= E & C ~= I & "
20                "C ~= D & C ~= E & I ~= D & I ~= E & D ~= E", True)
21
22 idp.Constraint("A ~= 0 & B ~= 0 & C ~= 0", True)
23 idp.Constraint("Even(A) & ~Even(B) & ~Even(C) & ~Even(D) & Even(E) & Even(I)",
24                True)
25 # De formule die moet opgaan.
26 idp.Constraint("10 * A + I + 10 * B + A = 100 * C + 10 * D + E", True)
27
28 # IDP-systeem uitvoeren
29 idp.refresh()
```

Bij uitvoering van bovenstaand programma, wordt er IDP code gegenereerd en doorgegeven aan de IDP executable. Door de debugopties te gebruiken kunnen we deze data ook laten wegschrijven naar een IDP file. Deze ziet er als volgt uit:

Listing 4.3: cde.idp file gegenereerd bij cde.py.

```
vocabulary V {
type Decimaal isa int
satisfiable()

A() : Decimaal
B() : Decimaal
C() : Decimaal
D() : Decimaal
E() : Decimaal
I() : Decimaal
Even(Decimaal)
}

theory T : V {
satisfiable().

A ~= B & A ~= C & A ~= I & A ~= D & A ~= E & B ~= C & B ~= I & B ~= D
    & B ~= E & C ~= I & C ~= D & C ~= E & I ~= D & I ~= E & D ~= E.

A ~= 0 & B ~= 0 & C ~= 0.

Even(A) & ~Even(B) & ~Even(C) & ~Even(D) & Even(E) & Even(I)

10 * A + I + 10 * B + A = 100 * C + 10 * D + E
}

structure S : V {
Decimaal = {0; 1; 2; 3; 4; 5; 6; 7; 8}
Even = {0; 2; 4; 6; 8}
}

procedure main(){
stdoptions.nbmodels = 1
stdoptions.xsb = true
allsols = modelexpand(T,S)
print(allsols[1])
}
```

Dit is een letterlijke implementatie: we hebben zo goed als iedere lijn in het IDP systeem zelf moeten definiëren. Stel dat we onze puzzel moeilijker willen maken, dan zouden we per nieuwe letter een constante moeten toevoegen dat deze niet nul mag zijn. Daarnaast moet de formule die aangeeft dat elke twee verschillende letters een verschillende waarde moeten krijgen, gewijzigd worden. Dit is dus een slecht schaalbare implementatie.

Net omdat we met Pyidp/Pyidp3 in Python kunnen werken, is het mogelijk om dit veel efficiënter

en schaalbaarder op een imperatieve manier te implementeren. Wanneer we hier een letter willen toevoegen, hoeven we deze enkel aan de juiste `*_letters` lijst toe te voegen en de formule aan te passen. Deze implementatie is te vinden in listing 4.4.

Listing 4.4: Efficiëntere implementatie van de cde puzzel.

```
1  letters = ["A", "B", "C", "D", "E", "I"]
2  nietnul = ["A", "B", "C"]
3  for i, letter in enumerate(letters):
4      idp.Constant(letter + ": Decimaal")  # Elke constante definieren
5      if letter in nietnul:
6          idp.Constraint(letter + " ~= 0", True)  # Sommige letters zijn niet nul
7      for j, letter2 in enumerate(letters):
8          if i < j:
9              # Letters mogen niet dezelfde waarde hebben
10             idp.Constraint(letter + " ~= " + letter2, True)
11
12     if letter in even_letters:  # Even letters
13         idp.Constraint("Even(" + letter + ")", True)
14     else:  # Oneven letters
15         idp.Constraint("~Even(" + letter + ")", True)
```

Zoals eerder vermeld, biedt Pyidp ook nog een bijkomende functionaliteit, naast het vormen van een directe API. Pyipd heeft namelijk een eigen syntax om beperkingen en definities te schrijven op een Pythonische manier te schrijven. Deze syntax kan het zelf omvormen naar beperkingen en definities voor het IDP systeem. Op deze manier hoeft een Python programmeur niet de IDP syntax voor beperkingen en definities te leren. In dit werk wordt hier verder niets mee gedaan. Beperkingen en definities worden altijd gedefinieerd in IDP-vorm, om zo efficiënter te kunnen werken en makkelijker te kunnen debuggen.

4.2 Implementatie Pyidp3

4.2.1 Overzetting naar Python 3

Het overzetten van de originele Pyidp naar een versie voor Python 3, gebeurde in twee stappen. Eerst hebben we gebruik gemaakt van `2to3`[2]. Dit is een tool gemaakt door de ontwikkelaars van Python zelf, dat zo goed mogelijk probeert om Python 2 code om te zetten in Python 3 code. Het scant alle bestanden, haalt code die niet meer zou werken in Python 3 eruit en vervangt deze door werkende code. Hierbij produceert het een .log file van alle veranderingen, welke te vinden is in appendix D.22.

Als tweede stap hebben we manueel nog enkele kleine foutjes moeten oplossen die `2to3` niet automatisch aanpast.

[2] `https://docs.python.org/3.7/library/2to3.html`

- De `popen` functie om data door te sturen aan de hand van een Linux `pipe`, accepteert in Python 3 enkel byte-objecten. Elke strings die we willen doorgeven, moeten we dus eerst omgezet worden aan de hand van de `encode()` methode.
- Ook de output van het IDP systeem gebeurt in byte-vorm, deze moeten we decoderen aan de hand van de `decode()` methode.

4.2.2 Nieuwe functionaliteiten

De bestaande Pyidp bevatte een beperkt aantal functionaliteiten, waardoor we genoodzaakt waren functionaliteiten te implementeren. Eerst nagegaan welke functionaliteiten er al waren in Pyidp, en welke er nog nodig zijn. Op basis hiervan is een lijst gemaakt van funtionaliteiten om te implementeren in Pyidp3. Volgende features zijn allemaal toegevoegd aan Pyidp3, omdat ze nodig waren voor dit werk.

- Ondersteuning voor de `term` blok: deze is nodig indien we willen minimaliseren.
- Ondersteuning voor het **isa** keywoord: hiermee kunnen we supertypes aanduiden.
- Meerdere modellen per oplossing toelaten, in plaats van enkel één.
- Minimalisatie implementeren.
- Toelaten om opties van IDP-systeem in te stellen.
- Een `compare` methode voor het IDP object, die twee lijsten/*dictionaries* kan vergelijken en zo het verschil tussen twee oplossingen kan weergeven.

Op de `compare` methode na, zijn al deze features momenteel succesvol geïmplementeerd. De `compare` methode kan momenteel enkel `dictionaries` vergelijken.

Voor een volledige uitleg van deze features en hoe ze zijn geïmplementeerd, zie D.2.3.

Belangrijk te vermelden is dat hoewel de oorspronkelijke Pyidp ondersteuning had voor het omzetten van Pythonische syntax naar IDP-vorm, dit in Pyidp3 niet meer ondersteund is. Al de oorspronkelijke code is nog aanwezig, maar is nooit getest geweest.

Hoofdstuk 5

De eindtoepassing

Hoofdstuk 3 gaf een overzicht over onze implementatie in het IDP-systeem, hoofdstuk 4 over hoe we kunnen interfacen met IDP vanuit Python. In dit hoofdstuk zal ingegaan worden op de uiteindelijke toepassing om interactief groepen te indelen. Hiervoor zal de toepassing via Pyidp3 communiceren met het IDP-systeem.

Deze toepassing is geschreven volgens het Model-View-Controller (MVC) ontwerppatroon. Hierbij splitsen we onze toepassing op in drie grote delen.

1. Model: de representatie van informatie, en de bewerkingen die hierop nodig zijn.
2. View: de concrete, visuele weergave van informatie.
3. Controller: ontvangt input, en stuurt hierbij de juiste functies van het view- en modelgedeelte aan.

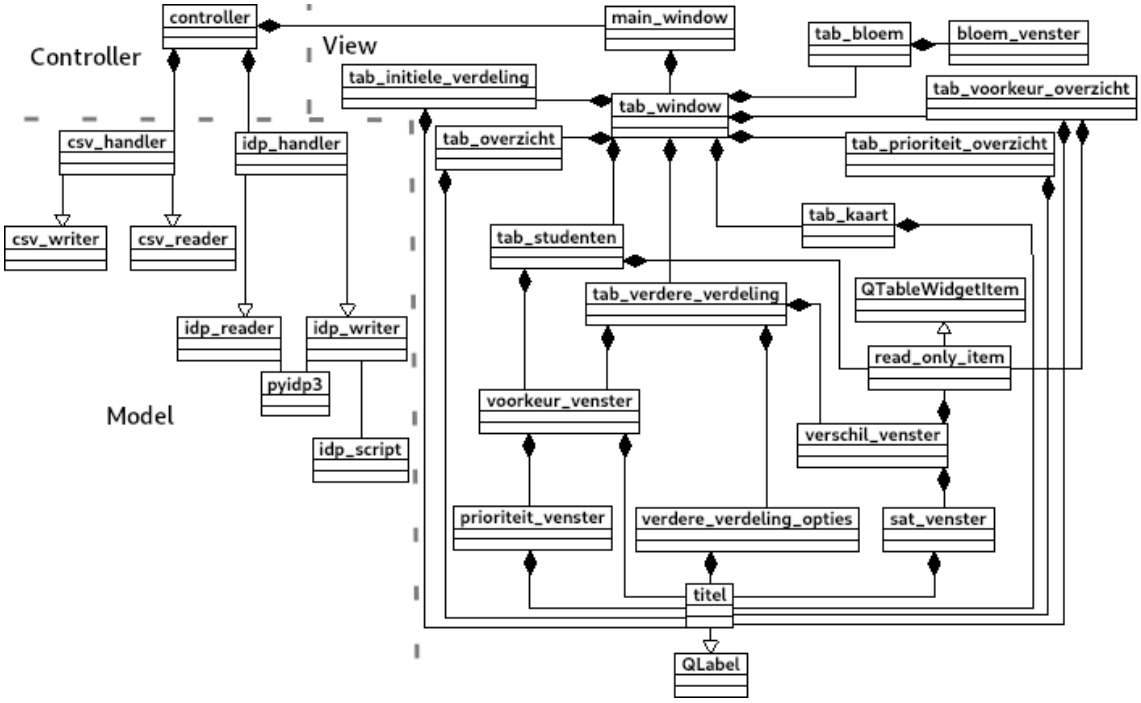
5.1 Model

Om gestructureerd te werk te gaan, is een beknopt Unified Model Language (UML) klassediagramma opgesteld. Door alle gebruikte klassen en hun onderlinge relaties weer te geven, beschrijft een klassediagramma de structuur van een programma. Dit diagramma is te zien in figuur 5.1.

Het modelgedeelte bestaat uit twee delen: een deel in het IDP-systeem, en een deel in Python 3. Concreet gebeurt het zoeken naar indelingen in IDP, en het verwerken van de input/output-data in Python. De intelligentie van de toepassing bevindt zich in het IDP-systeem, en de interactie met het IDP-systeem wordt verzorgd door Python (met view, controller en Pyidp3).

De eindtoepassing is in staat om informatie over de studenten in te lezen, en om informatie over de groepsindelingen weg te schrijven. De in- en uitvoer van bestanden wordt afgehandeld in de `csv_handler`.

Verder is voor het gebruik van het IDP-systeem via Python een vertaling nodig van onze eerdere IDP bestanden naar Pyidp3. Alle communicatie tussen de toepassing en het IDP-systeem verloopt via de `idp_handler` klasse.



Figuur 5.1: UML klassediagramma van de eindtoepassing.

5.1.1 CSV verwerken

Omdat het de bedoeling is dat onze toepassing CSV bestanden kan lezen en schrijven, hebben we hiervoor de `csv_handler` klasse gemaakt. Deze klasse erft over van twee ouderklassen: `csv_writer` en `csv_reader`, zoals te zien op figuur 5.1. `csv_handler` combineert de functionaliteiten van beide ouders. Deze splitsing is gemaakt om het lezen onafhankelijk van het schrijven te laten gebeuren.

De `csv_reader` klasse verzorgt het inlezen van CSV bestanden. De klasse gaat lijn per lijn de tekst af, en vult daarbij de nodige datastructuren. Er zijn twee mogelijkheden bij het inlezen van CSV bestanden. Ofwel is de lijst om in te lezen een lijst van studenten zoals deze van KU Loket gehaald wordt, ofwel is het een lijst van een eerder gevormde oplossing, die weggesschreven werd aan de hand van de applicatie. Afhankelijk van het bestand dat we in te lezen, vullen we andere datastructuren. Zo heeft een lijst van KU Loket nog geen groepsindelingen of voorkeuren van studenten, en bevat een lijst weggeschreven door de applicatie een volledig ander formaat.

De `csv_writer` klasse laat toe om onze indelingen terug weg te schrijven naar een CSV bestand. Belangrijk hierbij is dat deze bestanden worden niet hetzelfde CSV formaat bevatten als een oorspronkelijk bestand van KU Loket. In de oorspronkelijke CSV bestanden staat namelijk veel informatie die niet gebruikt wordt bij de groepsindeling, zoals geslacht, jaar van afstuderen, leeftijd en meer. Deze voor dit werk overbodige data wordt niet ingeladen, en kan dus ook niet mee weggeschreven worden. Het formaat waarin de groepsindelingen worden weggeschreven is “`Naam,Groep,Woonplaats,School,Voorkeuren`”.

5.1.2 communicatie IDP

De `idp_handler` verzorgt zowel het lezen van als het schrijven naar het IDP-systeem. Deze klasse heeft een gelijkaardige structuur als de `csv_handler`. De `idp_handler` erft over van twee andere ouderklassen: de `idp_reader` en de `idp_writer`. De splitsing is ook hier gemaakt om het lezen onafhankelijk van het schrijven te laten gebeuren. Alleen deze klassen communiceren met het IDP-systeem, en gebruiken dus Pyidp3.

De `idp_writer` klasse staat in voor het initialiseren van het IDP object en het maken van een initiële of incrementele indeling. Hierbij gebruikt `idp_writer` functies uit `idp_script`. Dit is een bestand dat de Pyidp3 implementaties van de idp-bestanden om indelingen te genereren bevat. Deze implementaties zijn terug te vinden in appendix B.1. Bij het maken van indelingen kunnen er parameters zoals minimum en maximum groepgrootte en aantal groepen meegegeven worden.

De `idp_reader` klasse bevat methoden die we gebruiken om de output van Pyidp3 bij groepsindelingen om te vormen naar voor ons bruikbare datastructuren. Bijvoorbeeld, bij de initiële indeling maken we geen gebruik van groepsnummers, maar definiëren we enkel welke studenten samen zitten in een groep. Dit om symmetrie te vermijden, zoals besproken in sectie 3.2.3. Om deze studenten een groepsnummer toe te wijzen, gebruiken we de `samen_naar_groep_dict` methode van de `idp_reader`. Deze methode zal ons een *dictionary* geven, met daarin voor elke student zijn groepsnummer.

5.2 View

Pyqt5 is een raamwerk ontwikkeld door *Riverbank Computing*[1] welke ons toelaat om eenvoudig GUI's te ontwikkelen in Python 3.

De view bestaat uit 1 groot venster, `tab_window`, met hierin acht verschillende tabbladen die elk als een afzonderlijke klasse geïmplementeerd werden.

1. Overzicht: op dit tabblad gebeurt inlezen van en schrijven naar CSV bestanden.
2. Studenten: geeft een overzicht van alle studenten in een tabelvorm, en laat toe om voorkeuren toe te voegen en prioriteiten aan te maken.
3. Initiële indeling: bevat velden om de parameters van de initiële indeling in te stellen, en een knop om deze te starten.
4. Incrementele indeling: bevat velden om de parameters van de incrementele indeling in te stellen, een knop om deze te starten, een overzicht van de verschillen tussen een huidig en een nieuwe oplossing en velden om voorkeuren toe te voegen en prioriteiten aan te maken.
5. Voorkeuren: bevat een tabel met daarin alle voorkeuren die reeds zijn ingegeven, plus knoppen om voorkeuren te verwijderen of te wijzigen.

[1] `https://www.riverbankcomputing.com/news`

6. Prioriteiten: bevat een tabel met daarin alle prioriteiten en hun waarde, met een knop om deze waarde te wijzigen.

7. Kaart: een kaart van België waarop de groepen kunnen getekend worden, om zo visueel de kwaliteit van een groepsindeling te kunnen inschatten.

8. Bloemdiagramma: een tekening met daarop alle studenten gegroepeerd in “bladeren”, waarbij ook de voorkeuren worden getekend aan de hand van lijnen tussen studenten.

Sommige tabbladen bestaan op hun beurt uit meerdere vensters, om zo op een overzichtelijke manier te kunnen programmeren. Ook laat dit ons toe om views te herbruiken, zoals het `voorkeur_venster` dat in zowel de `tab_studenten` als de `tab_verdere_indeling` voorkomt.

5.3 Controller

De controller zorgt voor alle interactiviteit. Wanneer op een knop in de view geklikt wordt, voert de controller een methode uit. Zo’n methode bestaat uit het inlezen van informatie uit de view, nakijken of de informatie juist is, het uitvoeren van de nodige manipulaties op de data en vervolgens de data weergeven in de correcte views.

In het vervolg van deze sectie worden alle tabbladen overlopen, met hierbij de functionaliteiten die de controller in deze tabs aanbiedt. Screenshots van deze tabbladen zijn te vinden in appendix B.

5.3.1 Overzicht

Door in een invulbalk het pad naar een CSV file op te geven, kunnen we deze inladen. Dit kan zowel een bestand zonder indeling (vanuit KU Loket) als een bestand met een al eerder gemaakte indeling zijn (vanuit de applicatie).

Gelijkaardig kunnen we de huidig gemaakte indeling weer wegschrijven naar een CSV bestand.

5.3.2 Studenten

Het ”Studenten-tabblad bevat een tabel met daarin voor elke student zijn naam, groep (indien van toepassing), woonplaats, middelbare school en eventuele voorkeuren. Door op de hoofding van een kolom te klikken kunnen we deze kolom sorteren.

Ook bevat dit tabblad een venster om voorkeuren op te geven (`voorkeur_venster`), rechts in het tabblad Dit venster bestaat uit twee delen.

- In het eerste gedeelte kan de gebruiker voorkeuren ingeven. Dit gebeurt door een naam in te vullen, vervolgens in het dropdown-menu een voorkeur te kiezen (wil samen met, wil niet samen met, wil in groep), een tweede naam of een groepsgetal in te vullen en een prioriteit te kiezen. Met de checkbox “Wederzijds” kan men aangeven of de voorkeur voor beide studenten geldt. Bijvoorbeeld wanneer student A bij B wil, en B bij A, hoeven we dit

maar eenmaal in te geven door de checkbox aan te vinken. De controller maakt dan twee voorkeuren aan, A bij B, en B bij A.

- Het tweede gedeelte van het `voorkeur_venster` is het `prioriteit_venster`. In het `prioriteit_venster` kan de gebruiker zelf een prioriteit toevoegen, door een naam en een waarde in te geven. Stel bijvoorbeeld dat er nood is aan een prioriteit “Sportstatuut” met waarde 15, dan kan de gebruiker de naam invullen samen met de gewenste prioriteitswaarde en deze toevoegen. De nieuwe prioriteit verschijnt dan in de lijst van prioriteiten om uit te kiezen bij het ingeven van voorkeuren.

5.3.3 Initiële indeling

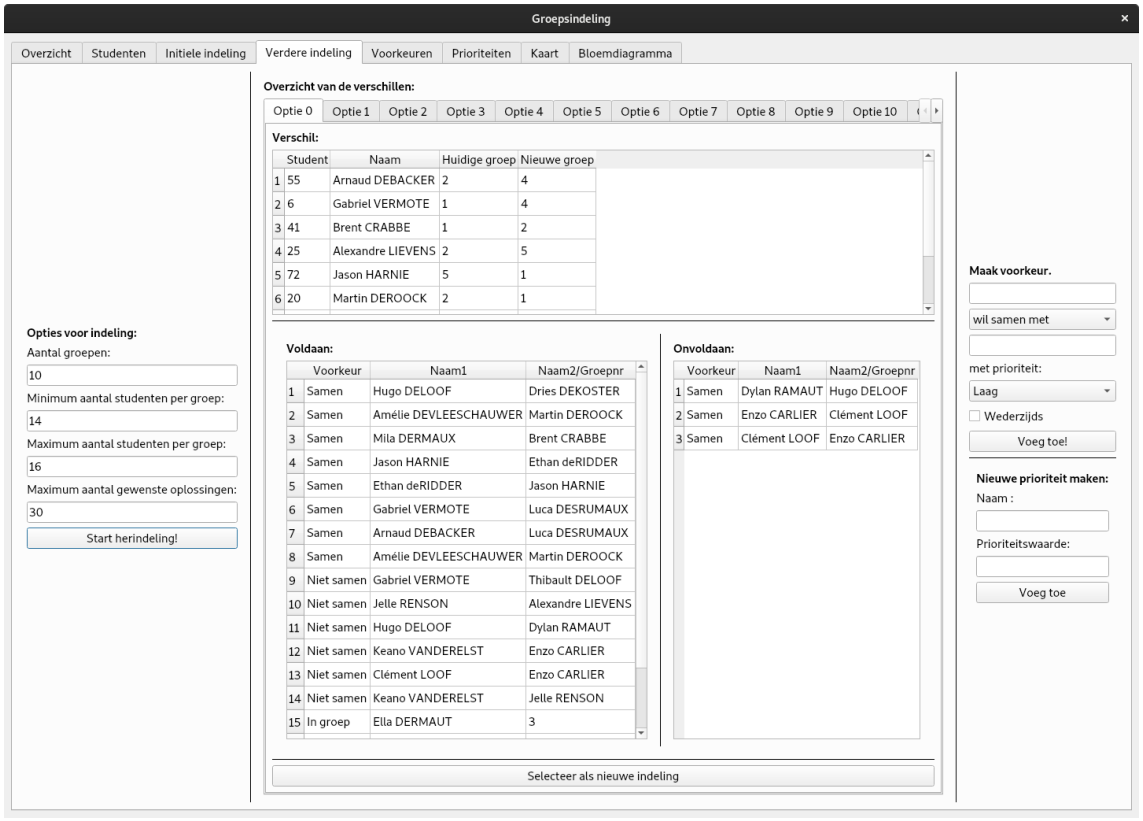
Het tabblad “Initiële indeling” laat toe om de verschillende parameters van de initiële indeling in te stellen. De gebruiker kan zo hier het aantal groepen, het minimum aantal studenten per groep en het maximum aantal studenten per groep instellen. Ook kan men de gewichten van de woonplaats, zone en school instellen. Deze gewichten bepalen hoe zwaar een bepaalde factor meeweegt bij het uitrekenen van de indeling in het IDP-systeem (zie 3.2.8). Stel dat de gebruiker bijvoorbeeld geen rekening wil houden met de middelbare scholen waarvan de studenten komen, dan volstaat het om dit gewicht op nul in te stellen.

Tot slot is het ook mogelijk om een maximum zoektijd in te stellen. Zoals eerder in sectie 3.2.7 vermeld, is het quasi onmogelijk om binnen een aanvaardbare tijdslimiet dé optimale oplossing te vinden. Het instellen van een maximum zoektijd laat ons toe om te zoeken naar de beste oplossing gevonden na `x` aantal seconden. Zo heeft de gebruiker de vrijheid om een snelle, maar oppervlakkigere, of een langere, maar grondigere indeling te maken afhankelijk van de beschikbare tijd.

5.3.4 Incrementele indeling

De kern van de applicatie zit in het “Incrementele indeling”-tabblad. Dit tabblad bestaat uit drie luiken:

- In het eerste luik kunnen we de parameters instellen voor de incrementele indeling. De eerste drie parameters zijn dezelfde als bij de initiële indeling en worden automatisch overgenomen. Deze parameters zijn wel nog wijzigbaar, wat interessant kan zijn wanneer er bijvoorbeeld een groep bij zou moeten komen. De vierde parameter laat ons toe om te kiezen hoeveel verschillende oplossingen we maximaal willen krijgen. In het geval dat het IDP-systeem voor de optimale waarde meerdere oplossingen vindt, dan kunnen we hiermee beperken hoeveel van deze oplossingen we te zien krijgen in de toepassing. Wanneer bijvoorbeeld student A’s enige voorkeur is om van groep te wisselen om niet meer bij student B te zitten, kan student A naar elke andere groep verplaatst worden. Dit zorgt voor meerdere oplossingen, met als enige verschil tussen de oplossingen de groep waarnaar de student verhuist.



Figuur 5.2: “Incrementele indeling”-tabblad.

- Het tweede luik is het `verschil_venster`. Dit venster toont de nieuwe oplossing(en) die het IDP-systeem bekomen heeft bij het berekenen van een incrementele indeling. Per oplossing wordt het verschil tussen de nieuwe indeling en de huidige indeling weergegeven, in tabelvorm. Elke student die van groep verandert, staat hierin opgelijst. In deze oplijsting staat ook de huidige groep van de student, en de groep waarin de student zou komen als we de oplossing accepteren. Onder deze tabel bevinden zich twee lijsten: een lijst van alle voorkeuren die in de oplossing voldaan zijn, en een lijst van alle voorkeuren die in de oplossing onvoldaan zijn. Op die manier kan een gebruiker snel zien wat de consequenties zouden zijn van het accepteren van een nieuwe indeling. De gebruiker kan dan de oplossing kiezen dat de beste volgende indeling lijkt.

  Het zelf kiezen van de volgende indeling geeft extra flexibiliteit en interactiviteit. De huidige IDP-implementatie maakt bijvoorbeeld geen onderscheid in de richting waarin een voorkeur gaat: wanneer student A bij student B zou willen, zijn zowel A naar B verhuizen als B naar A verhuizen voor het IDP-systeem gelijkwaardige oplossingen. Echter, als mens is het voor ons enkel logisch om A naar B proberen te verschuiven.

- Het derde luik is het `voorkeur_venster`, welke op dezelfde manier gebruikt wordt als het gelijknamige venster in het “Studenten”-tabblad.

Figuur 5.2 toont een screenshot van het tabblad, met respectievelijk van links naar rechts deze luiken.

5.3.5 Voorkeuren

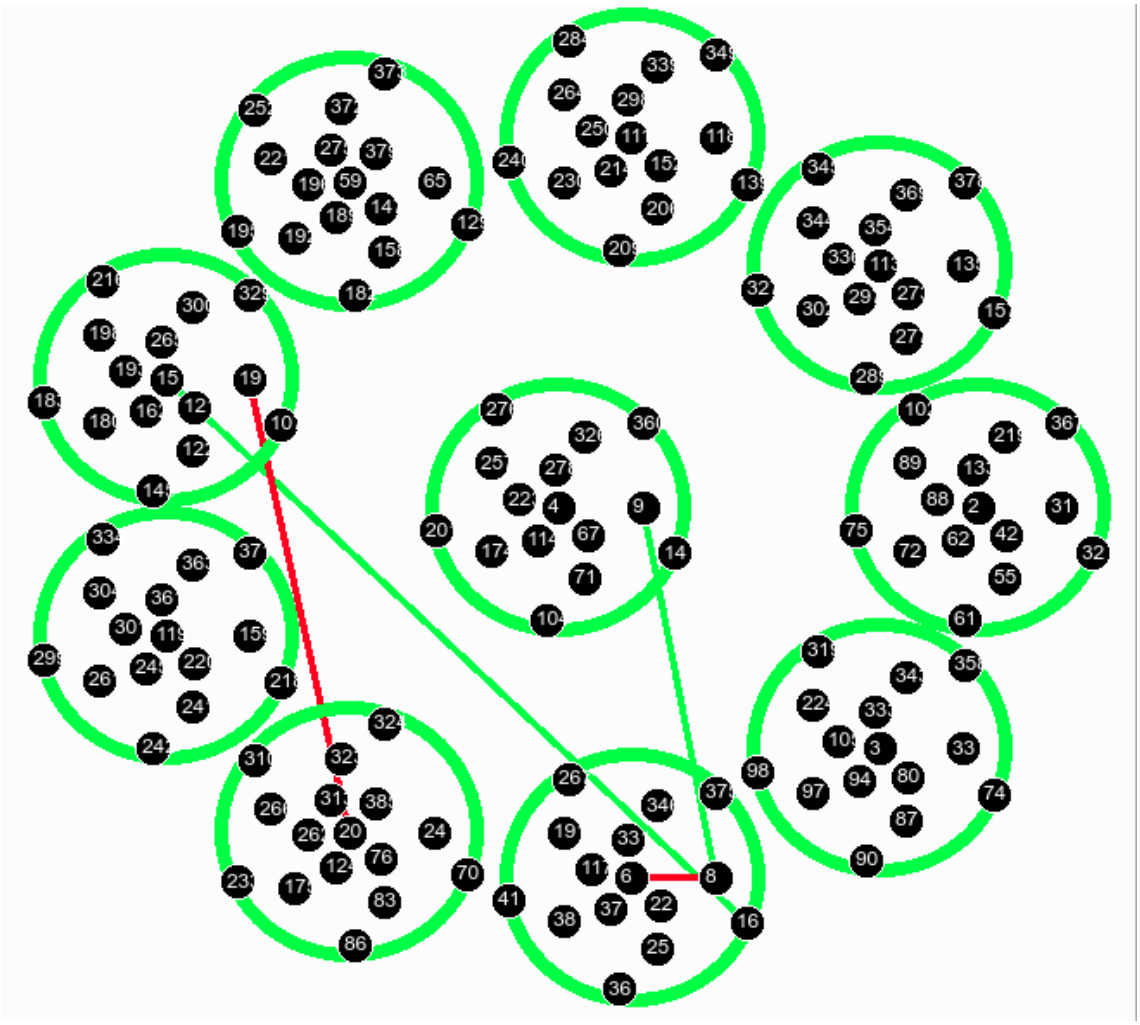
Het “Voorkeuren”-tabblad geeft alle voorkeuren weer, in tabelvorm. Op dit tabblad kunnen we ook prioriteiten van voorkeuren wijzigen, of verwijderen. Voorkeuren waaraan voldaan is worden in het groen getoond, voorkeuren waaraan niet voldaan is, in het rood, en voorkeuren die net zijn toegevoegd in het grijs.

5.3.6 Prioriteiten

Op het “Prioriteiten”-tabblad worden de prioriteiten en hun prioriteitswaarde weergegeven. Hier is het mogelijk om de waarde van een prioriteit nog aan te passen, indien de gebruiker dit nodig vindt.

5.3.7 Kaart

Het “Kaart”-tabblad laat toe om de groepsindelingen uit weer te geven op een kaart van België. De gebruiker kan aan de hand van checkboxen kiezen welke groepen te tekenen op de kaart. Elke stip stelt een gemeente voor, en het getal bij de stip duidt aan hoeveel studenten er in die gemeente wonen. Door de groepsindelingen uit te tekenen, kan visueel nagegaan worden hoe succesvol de initiële indeling verliep. Idealiter zou elke groep binnen één zone een cluster moeten vormen.



Figuur 5.3: Voorbeeld van een bloemdiagramma.

5.3.8 Bloemdiagramma

Het laatste tabblad van de toepassing, is het “Bloemdiagramma”. Dit is een grafische weergave van de groepsindeling, waarbij elke student wordt voorgesteld door een zwarte stip. De stippen van studenten die in dezelfde groepen zitten worden geclusterd in groene cirkels, welke een “blad” van de bloem voorsteld. Op deze manier bestaat wordt iedere groep voorgesteld door een blad, met in dat blad de studenten van die groep.

Vervolgens worden twee soorten voorkeuren getekend: de voorkeur om bij een student te zitten met een groene lijn, en de voorkeur om niet bij een student te zitten met een rode lijn. Een groene lijn is een voorkeur waaraan voldaan is wanneer deze zich in één blad bevindt, en een rode lijn is een voorkeur waaraan voldaan is indien deze zich over meerdere bladeren spant.

De gebruiker kan vervolgens kiezen welke groep in de kern van de bloem te tekenen. Zo is snel zichtbaar aan welke voorkeuren wel, en niet voldaan zijn voor de gekozen groep. Een voorbeeld van zo’n bloem is te zien in figuur 5.3.

Hoofdstuk 6

Evaluatie

In dit hoofdstuk worden onze implementaties van de initiële en incrementele indeling geëvalueerd. Eerst wordt gekeken naar hoe de gevonden waarde van een oplossing tijdens het maken van de initiële indeling varieert doorheen de tijd. Vervolgens vergelijken we onze implementatie van de initiële verdeling in het IDP-systeem met onze implementatie in clasp. Daarna bekijken we hoe snel een incrementele indeling gemaakt kan worden, afhankelijk van de hoeveelheid opgegeven voorkeuren. Tot slot beschrijven we de kwaliteit van de indelingen bekomen met onze implementaties.

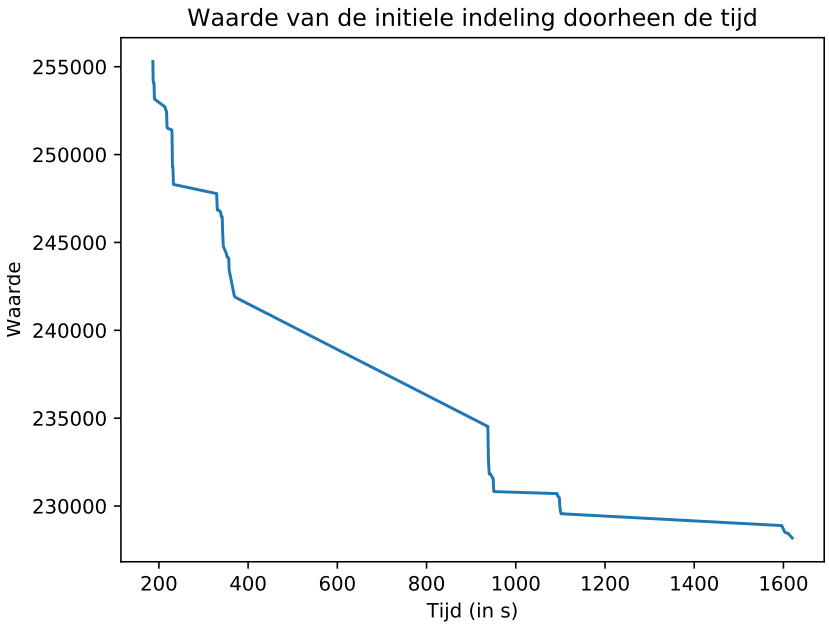
6.1 Waarde van oplossing tijdens initiële indeling

Om te bekijken hoe de oplossingswaarde gevonden door de initiële indeling varieert doorheen de tijd, hebben we een uur lang een initiële verdeling proberen maken en deze uitgezet op een grafiek (fig 6.1). Hierop is te zien dat het net minder dan 200 seconden kost om een eerste oplossing te vinden. De laatste oplossing is gevonden na zo'n 1600 seconden. Na deze oplossing heeft het IDP-systeem nog 33 minuten gerekend, maar geen nieuwe oplossing meer gevonden. In totaal zijn er 62 betere oplossingen gevonden doorheen het uur. De kwaliteit van de oplossing die hierbij gevonden wordt, wordt verderop besproken in sectie 6.4.1.

6.2 Vergelijking met clasp

Om het IDP-systeem te vergelijken met een ASP technologie, hebben we de initiële indeling in clasp geïmplementeerd. Deze implementatie is te vinden in appendix C.1. Een belangrijke kanttekening hierbij is dat wegens beperkte ervaring met clasp, deze implementatie eerder naïef is. Het is een vrij letterlijke vertaling van de initiële indeling, en mogelijk dus niet optimaal voor clasp.

De twee systemen hebben we vervolgens de oplossing laten zoeken voor 6, 8, 10, 12, 14 en 15 studenten, en hierbij de rekentijd gemeten. Hieruit is gebleken dat de implementatie in het IDP-systeem over het algemeen sneller werkt dan de implementatie in clasp. Deze resultaten zijn te zien



Figuur 6.1: Verloop van de initiële verdeling doorheen de tijd.

in tabel 6.1. Het aantal geteste studenten is klein gehouden, omdat anders de clasp implementatie niet meer uitrekenbaar is wegens te hoog RAM gebruik.

Tabel 6.1 Duratie zoektocht optimale waarde van IDP en clasp.

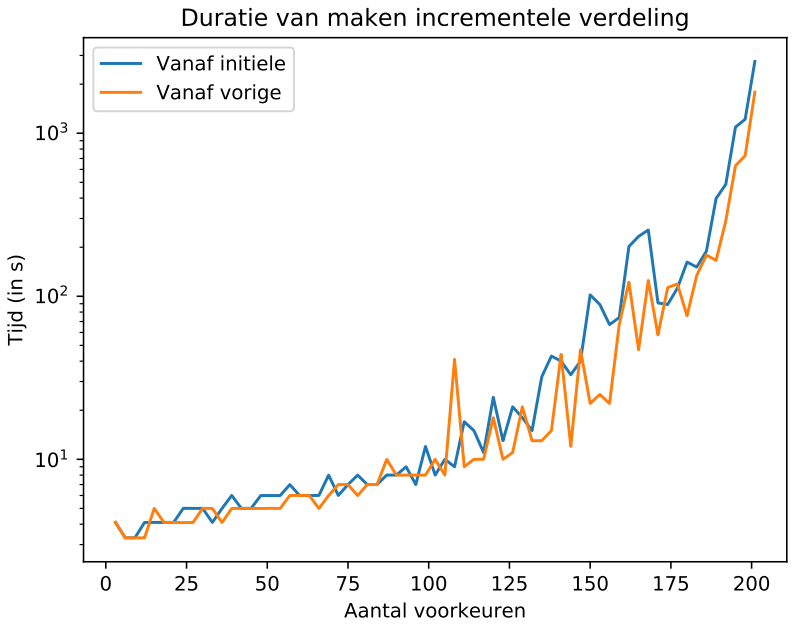
aantal studenten	6	8	10	12	14	15
IDP	0.246s	0.245s	0.787s	3.290s	3m49.023s	21m40.695s
clasp	0.143s	0.992s	24.612s	11m17.036s	40m5.342s	niet gevonden
% sneller	58%	404%	3127%	20578%	1050%	/

6.3 Rekentijd incrementele indeling

Bij de incrementele indeling is het belangrijk dat deze snel kan berekend worden. Om de snelheid van de implementatie van de incrementele indeling te onderzoeken, hebben we gemeten hoe lang het duurt om een indeling te maken bij een groeiend aantal voorkeuren. We onderscheiden twee vormen van het maken van een incrementele indeling.

1. Het maken van een incrementele indeling waarbij we telkens vanaf de initiële indeling starten.
2. Het maken van een incrementele indeling waarbij we telkens met de vorige indeling verder gaan.

De voorkeuren die werden toegevoegd werden telkens willekeurig gegenereerd. Bij de eerste vorm



Figuur 6.2: Aantal voorkeuren tegenover rekentijd bij incrementele verdeling.

werd er bij elke incrementele indeling één voorkeur toegevoegd, bij de tweede werden er drie voorkeuren toegevoegd.

Bij het maken van een incrementele indeling, startende van de initiële indeling, zien we op figuur 6.2 dat we tot ongeveer 100 voorkeuren kunnen toevoegen alvorens de berekening langer duurt dan tien seconden. Bij het uitrekenen van 200 voorkeuren, duurt het ongeveer 2700 seconden.

Wanneer we de incrementele indeling steeds opnieuw uitrekenen gebaseerd op de vorige indeling, gaat het rekenwerk over het algemeen net iets sneller. Pas na zo'n 120 voorkeuren (op een paar uitschieters na) duurt het langer dan 10 seconden om een indeling te maken.

Uit deze grafieken kunnen we stellen dat het op vlak van rekentijd niet meteen uitmaakt of de gebruiker vanaf de vorige of de initiële verdeling begint. Natuurlijk, de voorkeur gaat uit naar het incrementeel toevoegen van voorkeuren en herrekenen.

6.4 Kwaliteit van een indeling

Niet enkel de rekentijd is belangrijk bij het maken van een indeling, maar ook de kwaliteit ervan. Daarom wordt in deze sectie bekeken wat de kwaliteit van onze indelingen is.

6.4.1 Initiële indeling

Bij het maken van de initiële indeling is het de bedoeling om te prioriseren om zoveel mogelijk studenten van dezelfde scholen samen te plaatsen. Vervolgens worden op basis van woonplaats

	Naam	Groep	Woonplaats	School
8	Alexandre LIEVENS	6	2820 Bonheiden	College Hagelstein 2
9	Mila DERMAUX	6	2860 Sint-Katelijne-Waver	College Hagelstein 2
14	Ilias DEVOGHEL	2	2861 Sint-Katelijne-Waver	College Hagelstein 2
15	Wout MERTENS	6	2860 Sint-Katelijne-Waver	College Hagelstein 2
16	Rune SEGERS	9	2860 Sint-Katelijne-Waver	College Hagelstein 2
19	Aaron DEVLEESCHOUWER	9	2860 Sint-Katelijne-Waver	College Hagelstein 2
20	Alexis ROESEMS	10	2800 Mechelen	College Hagelstein 2
22	Mats VANDEVELDE	9	2860 Sint-Katelijne-Waver	College Hagelstein 2
24	Jarne ZELDERLOO	5	2860 Sint-Katelijne-Waver	College Hagelstein 2
25	Milan DEZUTTER	10	2820 Bonheiden	College Hagelstein 2
31	Emiel VANDEVELDE	8	2861 Sint-Katelijne-Waver	College Hagelstein 2
32	Mattéo DUPONT	8	2800 Mechelen	College Hagelstein 2

Figuur 6.3: Groepen van studenten die aan College Hagelstein 2 gestudeerd hebben.

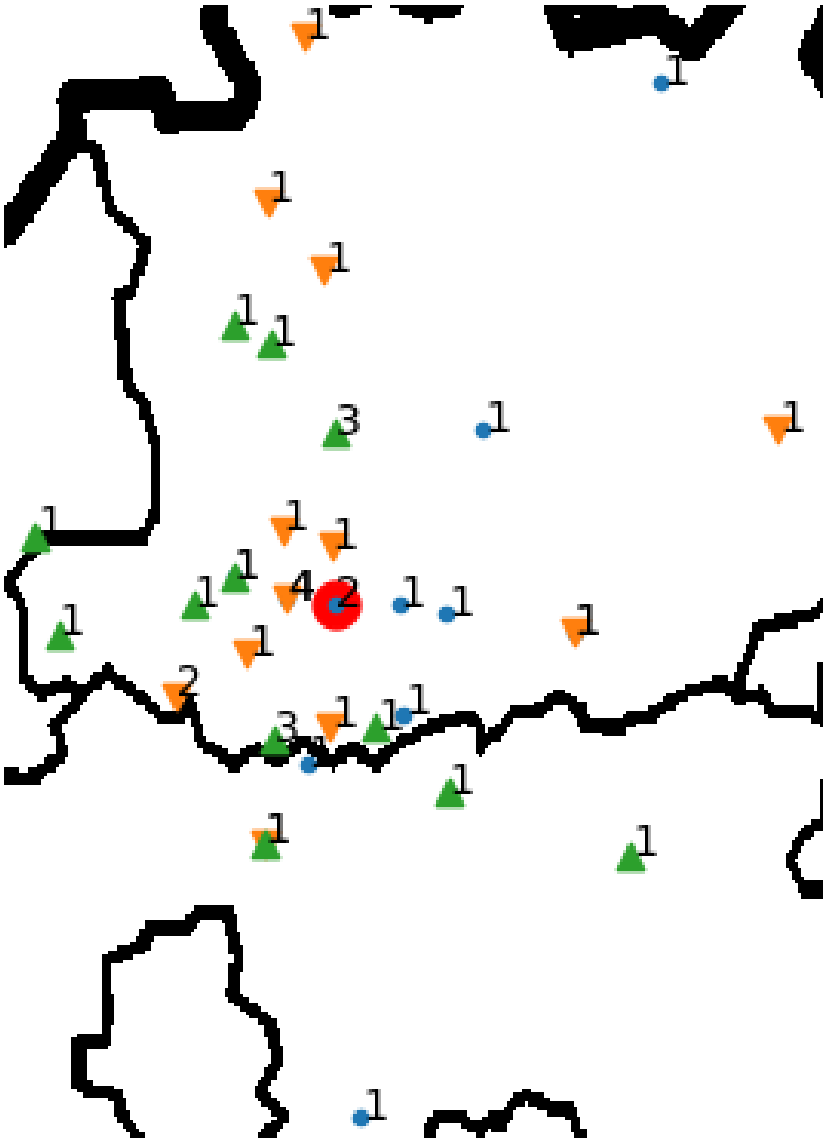
de rest van de indeling gemaakt, zodat alle studenten van een groep zo dicht mogelijk bij elkaar wonen.

In de huidige implementatie heeft het IDP-systeem veel tijd nodig om dit te berekenen. Door deze tijd te beperken op 300 seconden, is de bekomen oplossing niet optimaal. Wanneer we de studenttabel sorteren op school, zien we dat voor de meeste scholen de studenten verspreid zitten over de groepen. In figuur 6.3 is bijvoorbeeld te zien hoe 12 studenten van dezelfde middelbare school en met gelijkaardige postcodes in zes verschillende groepen ingedeeld worden. Idealiter zouden deze studenten over één of hoogstens twee verschillende groepen verspreid zijn.

Ook liggen de woonplaatsen van studenten van dezelfde groepen nogal ver uit elkaar. Door deze weer te geven op een kaart van België, kan men door naar de kaart te kijken quasi onmiddellijk een betere indeling bedenken. Dit zou geen probleem zijn wanneer de slechte indeling van woonplaatsen komt doordat de studenten in groepen wel uit dezelfde middelbare school komen, maar zoals eerder vermeld is dit ook niet het geval. Figuur 6.4 toont een voorbeeld van zo'n kaart.

6.4.2 Incrementele indeling

Bij een incrementele indeling is het de bedoeling om aan zoveel mogelijk voorwaarden te voldoen, met zo min mogelijk rekentijd. Eerder in het hoofdstuk, in sectie 6.3, is getoond hoe het IDP-systeem voldoende snel rekent afhankelijk van de hoeveelheid voorkeuren we in één keer opgeven, en hoeveel voorkeuren er in totaal al ingegeven zijn. Deze limieten waren tijdens het testen respectievelijk 100 en 120 voorkeuren. Aangezien het niet de bedoeling is van de implementatie om 100 voorkeuren per keer in te geven, maar eerder zo'n twee- of drietal, kunnen we stellen dat dit geen drempel zal zijn. Verder, in een reële situatie zullen nooit 120 voorkeuren in totaal voorkomen, en presteert onze implementatie van de incrementele indeling dus zeer goed.



Figuur 6.4: Kaart van België met daarop woonplaatsen van studenten van drie groepen.

Hoofdstuk 7

Conclusie en future work

De hoofdvraag van deze thesis is als volgt: “In hoeverre is het haalbaar om een interactieve toepassing te maken die gebruik maakt van het IDP-kennisbanksysteem?”. Om dit praktisch na te gaan, maakten we een toepassing voor de groepsindeling van studenten. We bekwamen twee doelstellingen.

1. Het schrijven van een intelligente kennisbank-oplossing voor het indelen van studenten.
2. Deze oplossing verwerken in een interactieve applicatie die voor een gebruiker bruikbaar is.

Om aan de eerste doelstelling te voldoen, is in dit werk een implementatie in het IDP-systeem gemaakt voor het indelen van groepen. Dit werd opgesplitst in twee vormen van indelingen. Eerst werd gekeken hoe we het beste een initiële indeling maken van studenten, op basis van woonplaats en middelbare school. Hierna is een implementatie van een incrementele indeling gemaakt, waarbij we de studenten herindelen rekening houdende met voorkeuren van de studenten.

De initiële indeling die in deze thesis gemaakt werd, werkt voldoende goed. Om een optimale oplossing te vinden moet het IDP-systeem met de huidige implementatie lang rekenen. Wanneer we de rekentijd limiteren, bekomen we een oppervlakkige oplossing. Maar, deze oplossing moet niet optimaal zijn, omdat er bij de incrementele verdelingen nog veel aan zal veranderen. We kunnen dus stellen dat de huidige implementatie van een initiële indeling voldoende goed werkt. Daarbovenop, vergeleken met een naïve implementatie in clasp werkt de initiële indeling in het IDP-systeem beter.

De incrementele indeling werkt uitstekend. Bij het rekenen houdt de implementatie rekening met alle voorkeuren, en berekent het de optimale waarde binnen een aanvaardbare tijd (twee tot tien seconden voor een reële situatie).

Voor de tweede doelstelling is een interactieve toepassing gemaakt om groepen te indelen. De intelligentie van de toepassing volgt uit het gebruik van het IDP-systeem. De interactie met het IDP-systeem wordt bereikt door gebruik van Python 3 en de Pyidp3 API, en biedt volgende mogelijkheden.

- De mogelijkheid om gewichten aan te passen in de initiële indeling.

- De mogelijkheid om harde beperkingen aan te passen in zowel de initiële als de incrementele indeling.
- Voorkeuren ingeven met een prioriteit.
- Eigen prioriteiten aanmaken.
- Meerdere mogelijke oplossingen weergeven bij de incrementele indeling.
- Weergeven wat er concreet verandert als de gebruiker een wijziging zou toepassen.

We kunnen besluiten dat de toepassing voldoende interactief is. Het IDP-kennisbanksysteem is dus goed te gebruiken in een interactieve toepassing.

Uiteraard, verdere uitbreidingen en aanpassingen zijn mogelijk. Het IDP-systeem werkt momenteel enkel op Linux-architectuur, met als gevolg hiervan dat de toepassing ook enkel ondersteund is op Linux. De toepassing zou wel kunnen werken op Windows indien we een client-serverstructuur implementeren. Het IDP-systeem zou dan op afstand aanwezig kunnen zijn op een server, of lokaal in een Docker. Pyidp3 zouden we dan aanpassen zodat het gebruik kan maken van sockets om te communiceren met het IDP-systeem.

Een andere interessante verbetering zou het laten verderrekenen van de initiële verdeling zijn. Stel dat een gebruiker na 300 seconden niet tevreden is en langer wil zoeken, dan moet in de huidige implementatie helemaal van het begin opnieuw gezocht worden. Het zou interessant kunnen zijn om het IDP-systeem verder te laten zoeken vanaf een al bestaande initiële indeling.

Het maken van de initiële indeling zou verbeterd kunnen worden door niet te starten vanaf nul, maar door alle studenten die van dezelfde school komen al meteen samen te plaatsen in een groep.

Nog een mogelijke verbetering zou zijn dat de gebruiker kan zien waarom niet aan een voorkeur voldaan is. Doordat de voorkeuren in de incrementele indelingen zachte beperkingen zijn, is dit niet eenduidig te achterhalen. Een mogelijk implementatie zou zijn om de voorkeur als harde beperking vast te leggen, en verder te minimaliseren zoals normaal. Het verschil tussen de huidige indeling en de oplossing geeft dan de minimale subset van voorkeuren die zouden moeten veranderen om de niet voldane voorkeur wel voldaan te maken.

Als een volgende verbetering zouden we het ingeven van voorkeuren door studenten zelf kunnen laten gebeuren, via een online webpagina. Op deze manier moet de gebruiker enkel maar incrementele indelingen kiezen.

Tot slot zou is het momenteel omslachtig om een nieuwe student aan een incrementele indeling toe te voegen. De gebruiker moet het CSV bestand openen, manueel de student en zijn informatie toevoegen en vervolgens de student aan een willekeurige groep toewijzen. Omdat in de realiteit late inschrijvingen kunnen voorkomen, zou het een meerwaarde zijn als we een student konden toevoegen via de toepassing. Hierbij zouden we bijvoorbeeld de student kunnen toewijzen aan een groep 0, en de implementatie aanpassen dat bij het maken van een incrementele indeling, groep 0 altijd leeg gemaakt moet worden door de nieuwe studenten te verplaatsen.

Bibliografie

Bekele, R. (2006). *Computer-Assisted Learner Group Formation Based on Personality Traits*. PhD thesis, Universiteit van Hamburg.

Bruynooghe, M., Blockeel, H., Bogaerts, B., De Pooter, S., Jansen, J., Labarre, A., Ramon, J., Denecker, M., and Verwer, S. (2015). Predicate logic as a modeling language: Modeling and solving some machine learning and data mining problems with idp3. *Theory and Practice of Logic Programming*, 15(6).

Calus, N. (2011). Ontwikkeling van een java api voor een kennisbanksysteem.

De Cat, B., Bogaerts, B., Bruynooghe, M., Janssens, G., and Denecker, M. (2014). Predicate logic as a modelling language: The idp system.

Devriendt, J. (2017). Exploiting symmetry in model expansion for predicate and propositional logic.

Dovier, A., Formisano, A., and Pontelli, E. (2007). An experimental comparison of constraint logic programming and answer set programming. volume 2, pages 1622–1625.

Huxham, M. and Land, R. (2000). Assigning students in group work projects. can we do better than random? *Innovations in Education and Teaching International*, 37(1):17–22.

Johnson, D. W., Johnson, R. T., and Smith, K. A. (2014). Cooperative learning: Improving university instruction by basing practice on validated theory. *Journal on Excellence in College Teaching*, 25(1):85–118.

Lifschitz, V. (2008). What is answer set programming? In *Proceedings of the 23rd National Conference on Artificial Intelligence - Volume 3*, AAAI'08, pages 1594–1597. AAAI Press.

Mitchell, M. (1995a). Genetic algorithms: An overview. volume 1, pages 31–39.

Mitchell, M. (1995b). Genetic algorithms: An overview.

Moreno, J., Ovalle, D. A., and Vicari, R. M. (2012). A genetic algorithm approach for group formation in collaborative learning considering multiple student characteristics. *Computers & Education*, 58(1):560–569.

Pieter, V., Dasseville, I., Janssens, G., and Denecker, M. (2016). The kb paradigm and its application to interactive configuration. volume 9585, pages 13–29. Springer Verlag.

Schaub, T. (2018). Answer set solving in practice.

Tacadao, G. S. and S.J., R. P. S. T. (2015). A generic model for the group formation problem using constraint logic programming. pages 307–308. IEEE.

Vennekens, J. (2015). Lowering the learning curve for declarative programming: a python api for the idp system.

Bijlage A

Files IDP

Listing A.1: Volledige IDP file om de initiële indeling te maken.

```
vocabulary V {
    type Student isa nat
    type Getal isa int
    type Postcode isa nat
    type Zone isa nat

    //De constanten die de parameters van onze initiele verdeling voorstellen
    AantGroepen: Getal
    MaxGrootte: Getal
    MinGrootte: Getal

    //Informatie over studenten
    Woont(Student): Postcode
    WoontZone(Student): Zone
    School(Student): string

    //Een wortel is een student waar andere studenten (welke bladeren zijn) mee samenzitten
    Wortel(Student)
    Blad(Student)

    //GestructureerdSamen is unidirectioneel, van elke student enkel naar zijn Wortel
    //Samen is bidirectioneel, van elke student naar elke groepsgenoot
    GestructureerdSamen(Student,Student)
    Samen(Student,Student)

    //Functie om de hoeveelheid groepsgenoten van een student mee te bepalen
    Aant(Student): Getal

    //De constanten die we gaan gebruiken om te minimaliseren
    NietSamenSchool: Getal //Aantal studenten die van dezelfde school komen,
                           // maar niet in dezelfde groep zitten.
    Afstand: Getal         //Afstand tussen alle studenten in dezelfde groep.
    NietInZone: Getal      //Aantal studenten die in dezelfde groep zitten,
                           // maar niet in dezelfde zone wonen.
    Totaal : Getal         //De gewogen som om te minimaliseren.
}


structure S : V {
```

```
    Postcode = {0..99}
    Student ={1..152}
    Getal = {0..10000000}
    Zone = {1..10}
    AantGroepen = 10
    MinGrootte = 14
    MaxGrootte = 15
    Woont ={1->87;2->98;...152->80;}
    WoontZone={1->2;2->1;...152->2;}
    School = {1->"GO! atheneum Anderlecht";...;152->"TSM-Bovenbouw";
   }
}


theory T : V {
    //Een student is enkel een wortel wanneer deze het kleinste studentnummer heeft
    // van alle andere studenten waarmee het samen in een groep zit.
    {
        !x[Student]: Wortel(x) <- x = min{y[Student]: Samen(x,y): y}.
    }
    //Een blad is een student die geen wortel is.
    {
        !x[Student]: Blad(x) <- ~Wortel(x).
    }
    //Twee studenten voldoen aan GestructureerdSamen wanneer de eerste student een wortel is,
    // de tweede een blad en ze allebei voldoen aan Samen.
    {
        !x[Student],y[Student]:GestructureerdSamen(x,y) <- Samen(x,y) & Wortel(x) & Blad(y).
    }

    !x[Student],y[Student]: Samen(x,y) => Samen(y,x).
    !x[Student],y[Student],z[Student]: Samen(x,y) & Samen(y,z) => Samen(z,x).


    !x[Student]: Aant(x) = #{y[Student]: GestructureerdSamen(x,y) | GestructureerdSamen(y,x)}.
    #{x[Student]: Wortel(x)} = AantGroepen.
    !x[Student]: Wortel(x) => MinGrootte =< Aant(x) =< MaxGrootte.
    !y[Student]: Blad(y) => Aant(y) = 1.


    NietSamenSchool = #{x[Student] y[Student]: x < y & ~Samen(x,y) & School(x) = School(y)}.
    Afstand = sum{x[Student] y[Student]: x < y & Samen(x,y) & WoontZone(x) = WoontZone(y):
        abs(Woont(x) - Woont(y))}.
    NietInZone = #{x[Student] y[Student]: x < y & Samen(x,y) & WoontZone(x) ~= WoontZone(y) }.

    Totaal = Afstand  + NietInZone * 10 + NietSamenSchool*1000.
}
term t: V{
    Totaal
}
```

Listing A.2: Volledige IDP file om een incrementele indeling te maken.

```
//Vocabulary met al de gegeven waarden.
vocabulary V_data {
    type Student isa nat
    type Getal isa nat
    type Groep isa nat
    type Prioriteit isa Getal
```

```
7
8       //Minimum- en maximumgroottes van de groepen.
9       MinGrootte: Getal
10      MaxGrootte: Getal
11
12      //De huidige groep.
13      HuidigeGroep(Student): Groep
14
15      //De voorkeuren van de studenten.
16      WilSamen(Student,Student)
17      WilNietSamen(Student,Student)
18      WilInGroep(Student, Groep)
19
20      //De prioriteiten die bij de voorkeuren horen.
21      partial PrioriteitWilSamen(Student,Student): Prioriteit
22      partial PrioriteitWilNietSamen(Student,Student): Prioriteit
23      partial PrioriteitWilInGroep(Student,Groep): Prioriteit
24
25  }
26
27
28  //Invullen van hoeveelheid studenten & groepen + voorkeuren
29  structure S : V_data {
30      //Groepsnummers lopen van 2 tot 11.
31      Groep = {2..11}
32      Student ={1..152}
33      Getal = {0..10000000}
34
35      MinGrootte = 14
36      MaxGrootte = 16
37
38      HuidigeGroep = { 1->2; 2->3; ...; 152->6 }
39
40      WilSamen = {142, 139; 139, 143; 139,142}
41      PrioriteitWilSamen = {142, 139 -> 2; 139, 143 -> 1; 139, 142->1}
42
43      WilNietSamen = {142, 143; 142, 141; 142, 140;}
44      PrioriteitWilNietSamen = {142, 143->2; 142, 141-> 2; 142, 140->2}
45
46      WilInGroep = {}
47      PrioriteitWilInGroep = {}
48  }
49
50
51  //Vocabulary met de te bekomen data.
52  vocabulary V_optimization{
53      extern vocabulary V_data //De inputvocabulary erbij laden.
54
55      //Functie die voor elke groep het aantal studenten geeft.
56      GroepGrootte(Groep): Getal
57
58      //Constanten die de prioriteiten van de niet voldane voorkeuren optellen.
59      TotUnsatSamen: Getal
60      TotUnsatNSamen: Getal
61      TotUnsatInGroep: Getal
62      TotWisselGroep: Getal
63
64      //Constante die we gaan minimaliseren, som van alle TotUnsats.
65      Totaal : Getal
```

```
66
67      //Relaties om de onvoldane voorkeuren weer te geven.
68      UnsatSamen(Student, Student)
69      UnsatNietSamen(Student,Student)
70      UnsatInGroep(Student, Groep)
71      WisselGroep(Student)     //Studenten die niet meer in dezelfde groep zitten.
72
73      VolgendeGroep(Student): Groep
74  }
75
76
77  theory T : V_optimization {
78      //Groepsgrootte berekenen.
79      !g[Groep]: GroepGrootte(g) = #{x[Student]: VolgendeGroep(x) = g}.
80
81      //Groepsgrootte vastleggen.
82      !g[Groep]: MinGrootte =< GroepGrootte(g) =< MaxGrootte.
83
84      //Unsat voorkeuren nagaan.
85      {
86          !x[Student] y[Student]: UnsatSamen(x,y) <- WilSamen(x,y)
87              & VolgendeGroep(x) ~= VolgendeGroep(y).
88      }
89
90      {
91          !x[Student] y[Student]: UnsatNietSamen(x,y) <- WilNietSamen(x,y)
92              & VolgendeGroep(x) = VolgendeGroep(y).
93      }
94
95      {
96          !x[Student] g[Groep]: UnsatInGroep(x,g) <- WilInGroep(x,g) & VolgendeGroep(x) ~= g.
97      }
98
99      {
100         !x[Student]: WisselGroep(x) <- HuidigeGroep(x) ~= VolgendeGroep(x).
101     }
102
103     //Som van de prioriteiten van unsat voorkeuren nagaan.
104     TotWisselGroep = #{x[Student]: WisselGroep(x)}.
105     TotUnsatSamen = sum{x[Student] y[Student]: UnsatSamen(x,y): PrioriteitWilSamen(x,y)}.
106     TotUnsatNSamen = sum{x[Student] y[Student]: UnsatNietSamen(x,y): PrioriteitWilNietSamen(x,y)}.
107     TotUnsatInGroep = sum{x[Student] g[Groep]: UnsatInGroep(x,g): PrioriteitWilInGroep(x,g)}.
108
109     //Totaal berekenen.
110     Totaal = TotUnsatSamen + TotUnsatNSamen + TotUnsatInGroep + TotWisselGroep.
111 }
112
113
114 term t: V_optimization {
115     Totaal
116 }
```

Bijlage B

Eindtoepassing

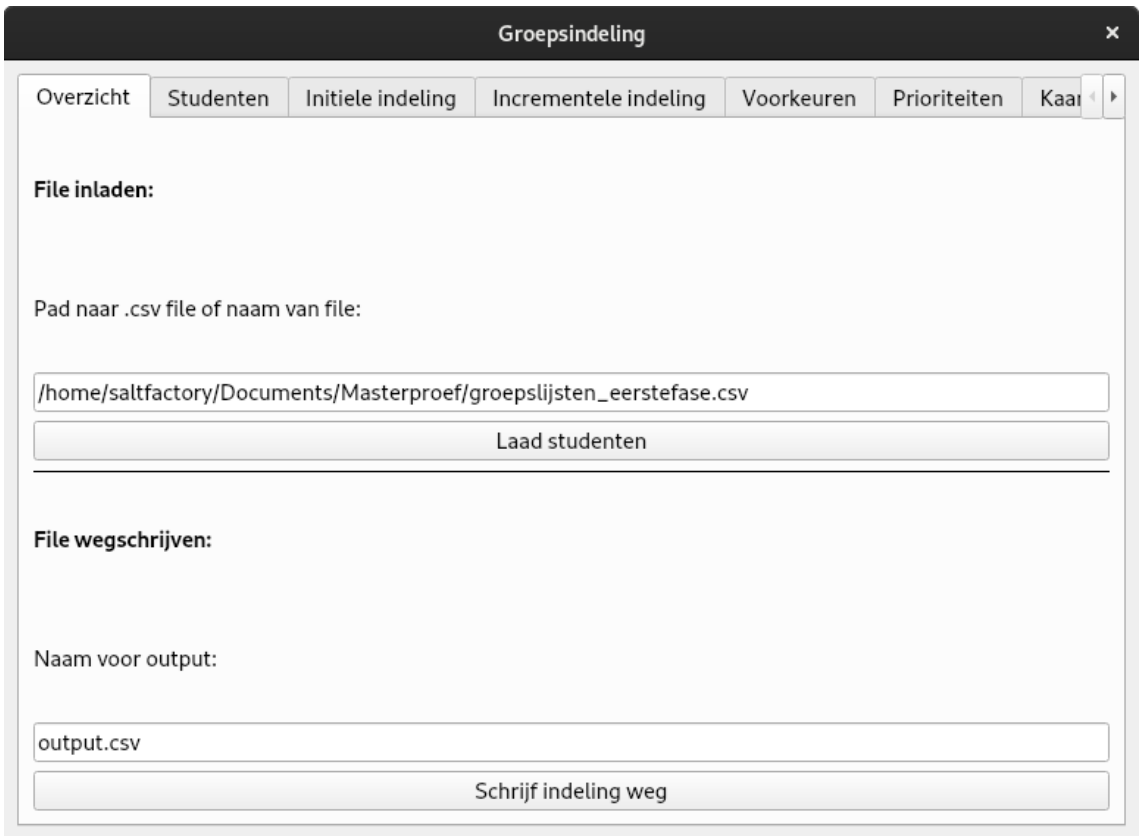
Listing B.1: De Pyidp3 implementatie van de initiële en incrementele verdelingen.

```
1
2
3  def verdere_verdeling(idp, groep_dict, samen_lijst, niet_samen_lijst,
4                        in_groep_lijst, prio_samen_dict, prio_niet_samen_dict,
5                        prio_in_groep_dict, min_studenten, max_studenten,
6                        aant_groepen, student_lijst, nbmodels=5):
7      idp.nbmodels = int(nbmodels)
8
9      # All the inputvars
10     idp.Type("Student", student_lijst)
11     idp.Type("Number", (0, 10000000))
12     idp.Type("Group", (1, (int(aant_groepen))))
13     idp.Type("Priority", "Number", isa=True)
14     idp.Constant("MinSize: Number", min_studenten)
15     idp.Constant("MaxSize: Number", max_studenten)
16
17     idp.Function("InGroup(Student): Group", groep_dict)
18     idp.Predicate("WantsTogether(Student, Student)", samen_lijst)
19     idp.Predicate("NotWantsTogether(Student, Student)",
20                   niet_samen_lijst)
21     idp.Predicate("WantsInGroup(Student, Group)", in_groep_lijst)
22     idp.Function("PriorityWantsTogether(Student, Student): Priority",
23                  prio_samen_dict, partial=True)
24     idp.Function("PriorityNotWantsTogether(Student, Student): Priority",
25                  prio_niet_samen_dict, partial=True)
26     idp.Function("PriorityWantsInGroup(Student, Group): Priority",
27                  prio_in_groep_dict, partial=True)
28
29     # All the inner workings + output
30     idp.Function("GroupSize(Group): Number")
31
32     idp.Constant("Total: Number")
33     idp.Constant("TotUnsatTogether: Number")
34     idp.Constant("TotUnsatNotTogether: Number")
35     idp.Constant("TotUnsatInGroup: Number")
36     idp.Constant("TotUnsatNewGroup: Number")
37
38     idp.Predicate("UnsatTogether(Student, Student)")
39     idp.Predicate("UnsatNotTogether(Student, Student)")
```

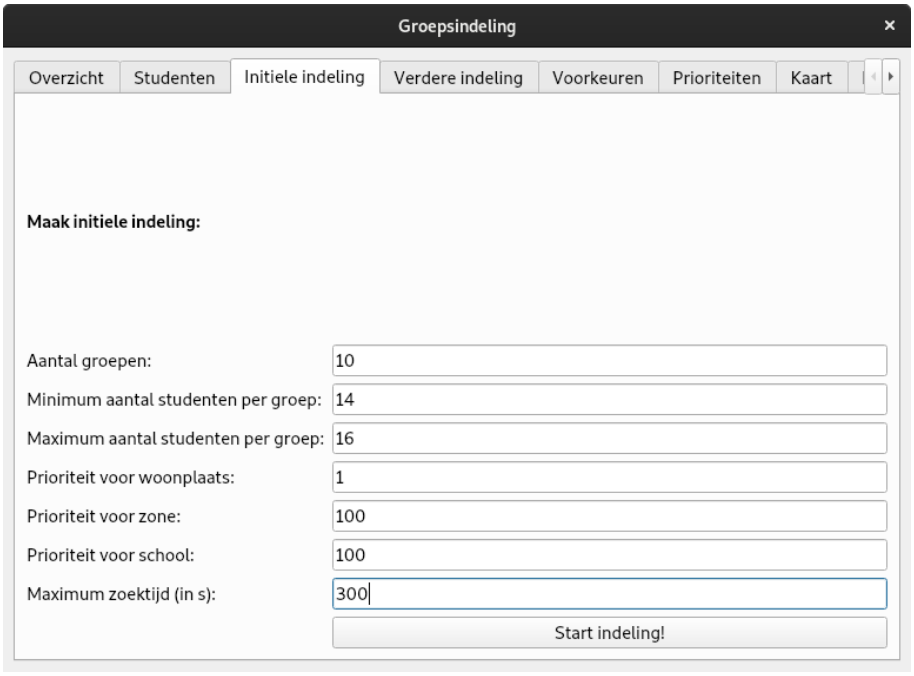
```
40      idp.Predicate("UnsatInGroup(Student, Group)")
41      idp.Predicate("UnsatNewGroup(Student)")
42
43      idp.Function("NewInGroup(Student): Group")
44
45      # All the necessary constraints
46      idp.Constraint("!g[Group]: GroupSize(g) = #{x[Student]: "
47                     "NewInGroup(x) = g}",
48                     True)
49      idp.Constraint("!g[Group]: MinSize =< GroupSize(g) =< MaxSize", True)
50      idp.Constraint("TotUnsatNewGroup = #{x[Student]: "
51                     "InGroup(x) ~= NewInGroup(x)}",
52                     True)
53      idp.Constraint("TotUnsatTogether = sum{x[Student] y[Student]: "
54                     "WantsTogether(x,y) & NewInGroup(x) ~= NewInGroup(y): "
55                     "PriorityWantsTogether(x,y)}",
56                     True)
57      idp.Constraint("TotUnsatNotTogether = sum{x[Student] y[Student]: "
58                     "NotWantsTogether(x,y) & "
59                     "NewInGroup(x) = NewInGroup(y): "
60                     "PriorityNotWantsTogether(x,y)}",
61                     True)
62      idp.Constraint("TotUnsatInGroup = sum{x[Student] g[Group]: "
63                     "WantsInGroup(x,g) "
64                     "& NewInGroup(x) ~= g: PriorityWantsInGroup(x,g)}", True)
65      idp.Constraint("Total = TotUnsatTogether + TotUnsatNotTogether + "
66                     "TotUnsatInGroup + TotUnsatNewGroup", True)
67
68      idp.Define("!x[Student] y[Student]: UnsatTogether(x,y) "
69                 "<- WantsTogether(x,y) "
70                 "& NewInGroup(x) ~= NewInGroup(y).", True)
71      idp.Define("!x[Student] y[Student]: UnsatNotTogether(x,y) <- "
72                 "NotWantsTogether(x,y) & NewInGroup(x) = NewInGroup(y).",
73                 True)
74      idp.Define("!x[Student]: UnsatNewGroup(x) <- InGroup(x) ~= NewInGroup(x).",
75                 True)
76      idp.Define("!x[Student] g[Group]: UnsatInGroup(x,g) <- WantsInGroup(x,g) "
77                 "& NewInGroup(x) ~= g.", True)
78
79      sols = idp.minimize("Total")
80      return sols
81
82
83  def initiele(idp, student_lijst, woon_dict, zone_dict, school_dict,
84               aant_groepen, min_studenten, max_studenten, afstand_gewicht,
85               zone_gewicht, school_gewicht,
86               mxtimeout=300, nbmodels=1):
87      """
88  ____:param idp: De IDP klasse van Pyidp3
89  ____:param student_lijst: lijst van alle studenten
90  ____:param woon_dict: voor iedere student zijn postcode
91  ____:param zone_dict: voor iedere student zijn zone
92  ____:param school_dict: voor iedere student zijn school
93  ____:param aant_groepen: het aantal groepen om in te verdelen
94  ____:param min_studenten: minimum aantal studenten per groep
95  ____:param max_studenten: maximum aantal studenten per groep
96  ____:param mxtimeout: maximum timeout voor het IDP-systeem
97  ____:param nbmodels: aantal models nodig van het IDP-systeem
98  ____:type idp: IDP
```

```
99  ␣␣␣␣:type␣student_lijst:␣list
100 ␣␣␣␣:type␣woon_dict:␣dictionary
101 ␣␣␣␣:type␣zone_dict:␣dictionary
102 ␣␣␣␣:type␣school_dict:␣dictionary
103 ␣␣␣␣:type␣aant_groepen:␣int
104 ␣␣␣␣:type␣min_studenten:␣int
105 ␣␣␣␣:type␣max_studenten:␣int
106 ␣␣␣␣:type␣mxtimeout:␣int
107 ␣␣␣␣:type␣nbmodels:␣int
108 ␣␣␣␣:returns:␣solutions
109 ␣␣␣␣:rtype:␣dictionary␣met␣daarin␣alle␣oplossingen
110 ␣␣␣␣"""
111     idp.nbmodels = nbmodels
112     idp.mxtimeout = mxtimeout
113     idp.Type("Student", student_lijst)
114     idp.Type("Getal", (0, 10000000))
115     idp.Type("Postcode", (0, 100))
116     idp.Type("Zone", (0, 10))
117     idp.Constant("MinGrootte:␣Getal", min_studenten)
118     idp.Constant("MaxGrootte:␣Getal", max_studenten)
119     idp.Constant("AantGroepen:␣Getal", aant_groepen)
120
121     idp.Predicate("Samen(Student,Student)")
122     idp.Predicate("SSamen(Student,␣Student)")
123
124     idp.Function("Woont(Student):␣Postcode", woon_dict)
125     idp.Function("WoontZone(Student):␣Zone", zone_dict)
126     idp.Function("School(Student):␣string", school_dict)
127
128     idp.Predicate("Wortel(Student)")
129     idp.Predicate("Blad(Student)")
130     idp.Function("Aant(Student):␣Getal")
131
132     idp.Constant("NietSamenSchool:␣Getal")
133     idp.Constant("Afstand:␣Getal")
134     idp.Constant("NietInZone:␣Getal")
135     idp.Constant("Totaal:␣Getal")
136
137     idp.Define("!x[Student]:␣Wortel(x)␣<-␣x␣=␣min{y[Student]:␣Samen(x,y):y}.", True)
138     idp.Define("!x[Student]:␣Blad(x)␣<-␣~Wortel(x).", True)
139     idp.Define("!x[Student],␣y[Student]:␣SSamen(x,y)␣<-␣"
140                "Samen(x,y)␣&␣Wortel(x)␣&␣Blad(y).", True)
141
142     idp.Constraint("!x[Student],␣y[Student]:␣Samen(x,y)␣=>␣Samen(y,x).", True)
143     idp.Constraint("!x[Student],␣y[Student],␣z[Student]:␣"
144                    "Samen(x,y)␣&␣Samen(y,z)␣=>␣Samen(z,x).", True)
145
146     idp.Constraint("!x[Student]:␣Aant(x)␣=␣#{y[Student]:"
147                    "␣SSamen(x,y)␣|␣SSamen(y,x)}.",
148                    True)
149     idp.Constraint("#{x[Student]:␣Wortel(x)}␣=␣AantGroepen.", True)
150     idp.Constraint("!x[Student]:␣Wortel(x)␣=>␣MinGrootte␣=<␣Aant(x)␣=<"
151                    "␣MaxGrootte.", True)
152     idp.Constraint("!y[Student]:␣Blad(y)␣=>␣Aant(y)␣=␣1.", True)
153
154     idp.Constraint("NietSamenSchool␣=␣#{x[Student]␣y[Student]:␣x␣<␣y␣&␣"
155                    "~Samen(x,y)␣&␣School(x)␣=␣School(y)}", True)
156     idp.Constraint("Afstand␣=␣sum{x[Student]␣y[Student]:␣x␣<␣y␣&␣Samen(x,y)"
157                    "&␣WoontZone(x)␣=␣WoontZone(y):␣abs(Woont(x)␣-␣Woont(y))}",
```

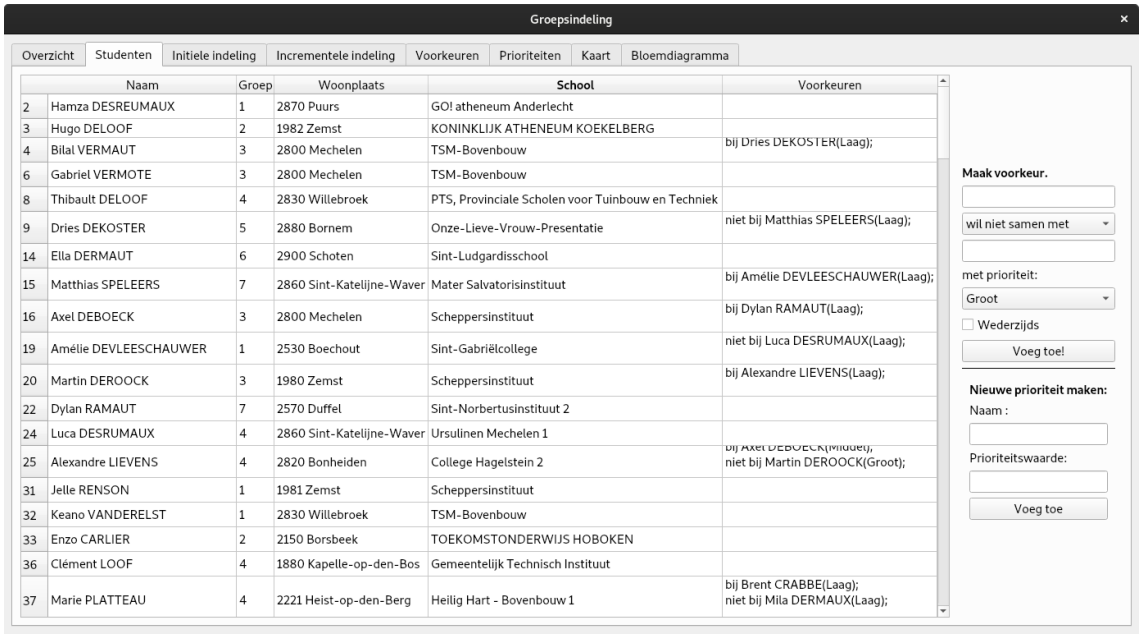
```
158                     True)
159     idp.Constraint("NietInZone␣=␣#{x[Student]␣y[Student]:␣x␣<␣y␣&␣Samen(x,y)␣&"
160                     "␣WoontZone(x)␣~=␣WoontZone(y)␣}", True)
161
162     totaal = "Totaal␣=␣Afstand␣*␣{:s}␣+␣NietInZone␣*␣{:s}␣+␣NietSamenSchool␣*␣{:s}"\
163             .format(afstand_gewicht, zone_gewicht, school_gewicht)
164     idp.Constraint(totaal,
165                     True)
166
167     solutions = idp.minimize("Totaal")
168     return solutions
```



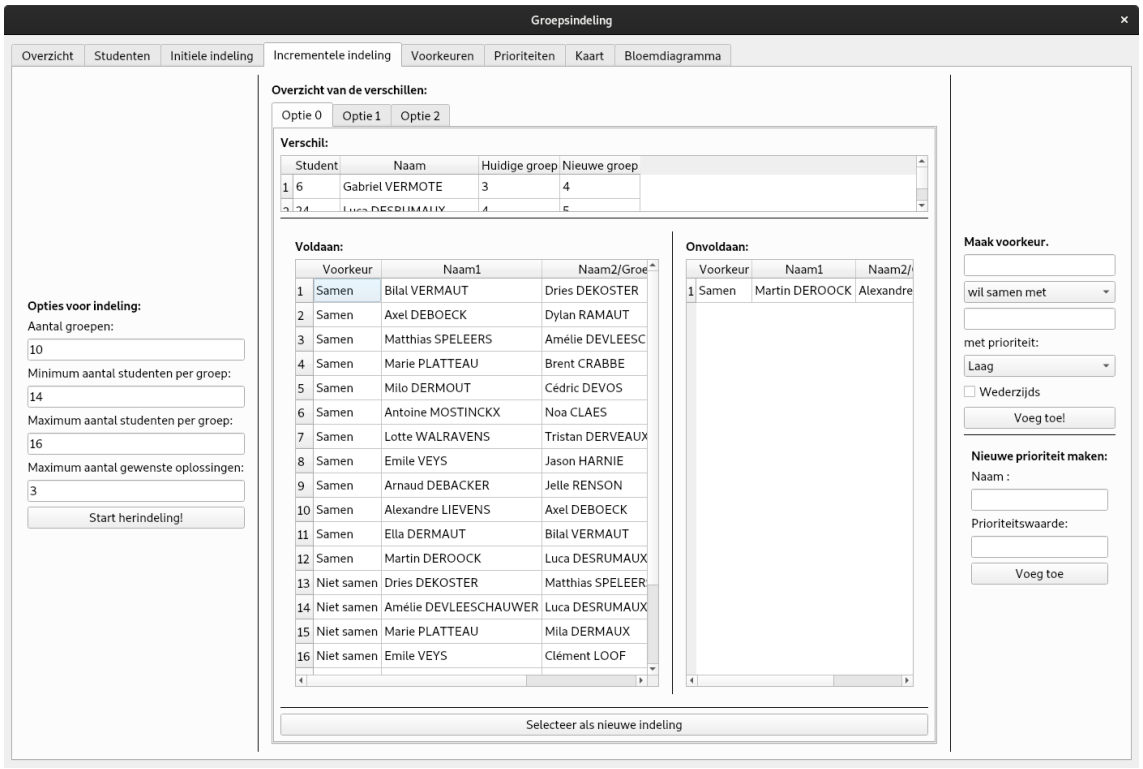
Figuur B.1: “Overzicht”-tabblad.



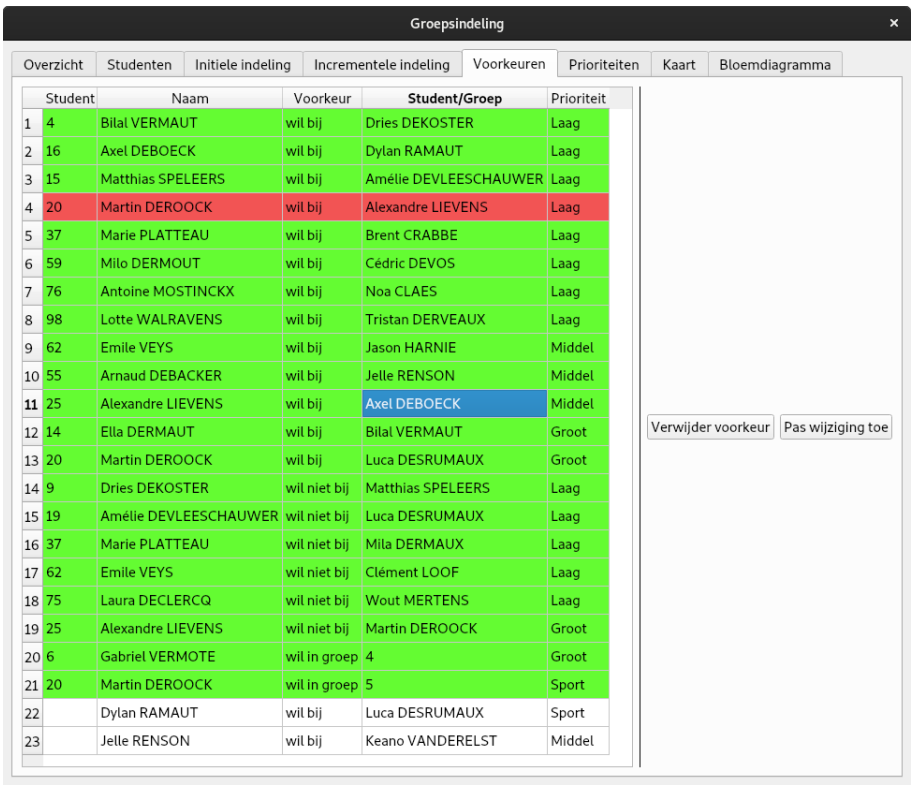
Figuur B.3: “Initiële verdeling”-tabblad.



Figuur B.2: “Studenten”-tabblad.

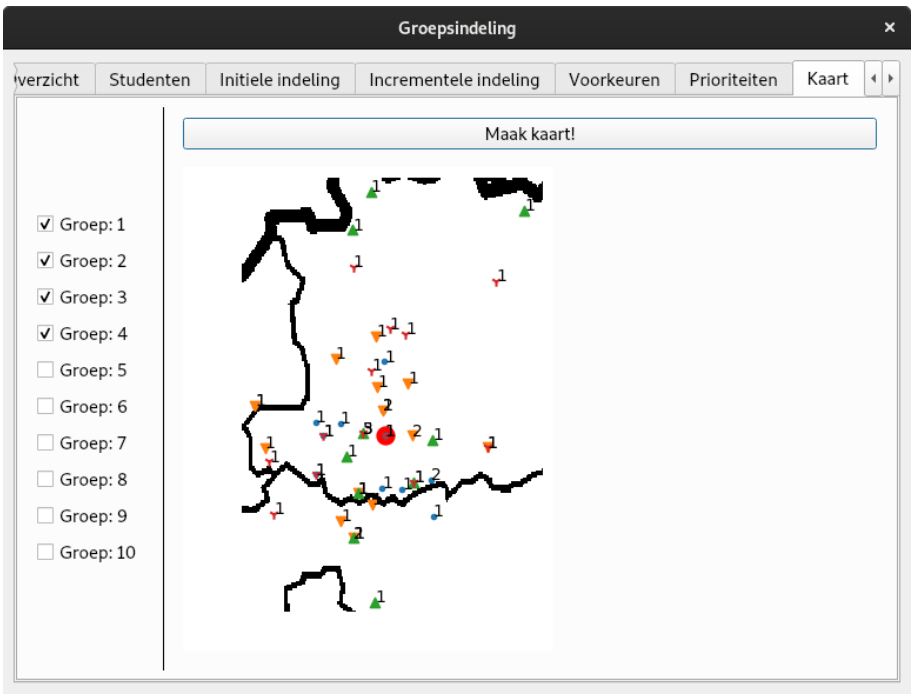


Figuur B.4: “Incrementele verdeling”-tabblad.



Figuur B.5: “Voorkeuren”-tabblad.

Figuur B.6: “Prioriteiten”-tabblad.



Figuur B.7: “Kaart”-tabblad.

Figuur B.8: “Bloemdiagramma”-tabblad.

Bijlage C

clasp implementatie

Listing C.1: De clasp implementatie van de initiële indeling.

```
#const mingrootte = 4.
#const maxgrootte = 5.
#const aantgroep = 3.
student(1..15).
postcode(0..99).
woont(1, 87; 2, 98; 3, 80;...; 152, 80;).
zone(1, 2; 2, 1; 3, 2; ...;152, 2;).
school(1, "GO! atheneum Anderlecht";...;152, "TSM-Bovenbouw";).

% Generate
blad(X) :- not wortel(X), student(X).
X < Y :- samen(X,Y), wortel(X), blad(Y).

{wortel(X): student(X)} = aantgroep.
mingrootte <= {samen(X,Y): blad(Y) } <= maxgrootte :- wortel(X).
{samen(Y,X): wortel(Y)} = 1 :- blad(X).

volsamen(X,Y) :- samen(X,Y).
volsamen(Y,Z) :- samen(X,Y), samen(X,Z), Y < Z.


afstand(A) :- A = #sum{|P-Q|, X, Y: volsamen(X,Y), woont(X,P), woont(Y,Q), not notzone(X,Y)}.
aantnotschool(S) :- S = #count{X,Y: A != B, volsamen(X,Y), school(X,A), school(Y,B)}.
aantnotzone(Z) :- Z = #count{X,Y: A!= B, volsamen(X,Y), zone(X,A), zone(Y,B)}.

som(T) :- T = A + 100*S + 100*Z, afstand(A), aantnotschool(S), aantnotzone(Z).

#minimize {T : som(T)}.
```

Bijlage D

Pyidp3

Dit hoofdstuk is de volledige versie van het Pyidp3 hoofdstuk besproken in de thesistekst. In deze versie volgt eerst een herhaling, maar gaan we vervolgens overal in meer detail op in.

Zoals eerder vermeld in de inleiding hebben we nood aan een interface rond het IDP-systeem, om deze te kunnen gebruiken vanuit Python 3. Vennekens (2015) heeft hiervoor de Pyidp[1] API ontwikkeld voor Python 2. Deze heeft als doel om toe te laten in IDP te redeneren, maar dan op een imperatieve manier in Python. Hierdoor zouden programmeurs van de voordelen van het IDP-systeem kunnen genieten, zonder expliciet de nieuwe syntax ervan aan te hoeven leren. Het voornaamste nadeel van deze API, is dat het enkel ondersteund wordt voor Python 2. Verder ontbreken er ook bepaalde functionaliteiten (zie D.2.3) om het voor dit werk bruikbaar te maken.

Voor deze thesis hebben we de Pyidp3 API ontwikkeld. Dit is in de eerste plaats een rechtstreekse port van Pyidp, die we vervolgens verder hebben uitgebreid met extra functionaliteiten, “Quality Of Life” (QOL), bugfiksen en documentatie.

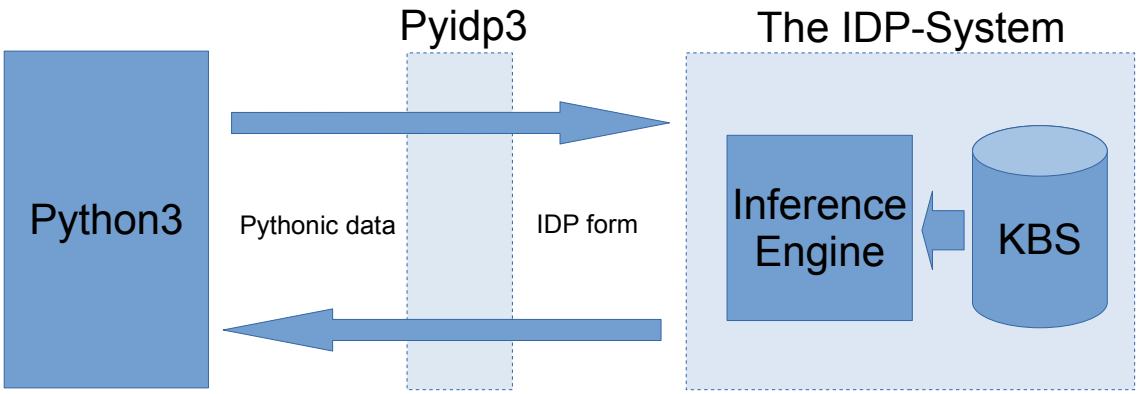
Zoals vermeld is het doel van Pyidp3 om een brug te vormen tussen Python 3 en het IDP-systeem. Pyidp3 zal objecten en data van Python kunnen omzetten in een structuur die verstaanbaar is voor het IDP-systeem. Deze structuur is dezelfde als die van een idp-bestand. Vervolgens kan Pyidp3 de output van het IDP-systeem lezen, interpreteren en terug omzetten in Pythonformaat. Dit is weergegeven in figuur D.1.

D.1 Werking Pyidp/Pyidp3

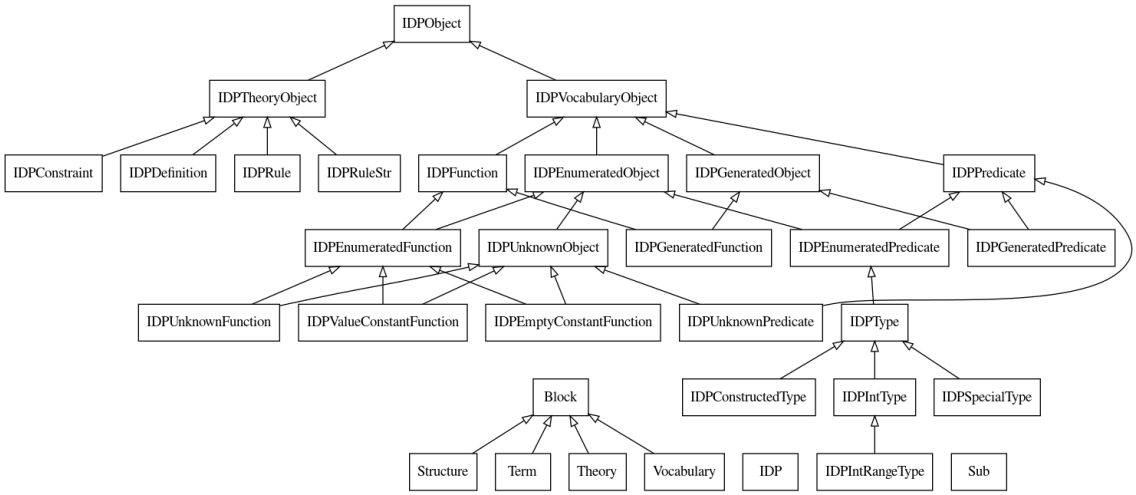
Fundamenteel werkt Pyidp/Pyidp3 heel eenvoudig: het laat toe om al de verschillende vormen van IDP symbolen in te geven via Python, waarna het elk van deze symbolen vertaalt naar de string die de “IDP vorm” van het symbool voorstelt. Wanneer we deze strings van al deze symbolen samenzetten, hebben we een uitgewerkt IDP bestand. Vervolgens kunnen we het IDP bestand doorsturen naar de IDP executable (met een Unix `pipe`), en de output uitlezen en verwerken.

Om Pyidp te gebruiken dient de gebruiker enkel maar één object aan te maken: het `IDP` object. Ver-

[1] `https://bitbucket.org/joostv/pyidp`



Figuur D.1: Schema van Python, Pyidp3 en IDP.



Figuur D.2: Klassediagramma van Pyidp3.

volgens is het mogelijk om via dit `IDP` object alles te definiëren wat in de `vocabulary`, `structure` en `theory` moet komen. Hiervoor bevat de IDP klasse de volgende methodes: `Constant`, `Constraint`, `Define`, `Function`, `Predicate` en `Type`.

Aan de hand van deze zes methoden kunnen volledige bestanden voor het IDP-systeem ontworpen worden. `IDP` maakt elke keer dat zo'n methode wordt gebruikt, een nieuw `IDPObject` aan (of beter gezegd, een kindklasse hiervan, dat een enger type voorstelt). Een `IDPObject` is een Pythonische representatie van een predicaat, definitie, functie, type of constante in IDP. In figuur D.2 is een volledig overzicht te zien van alle welke klassen die overerven van `IDPObject`.

Op het eerste niveau erft de `IDPObject`-klasse in objecten die in de `theory` komen, zoals definities en beperkingen, en objecten die het `vocabulary` komen (en eventueel ook `structure`), zoals functies, predicaten, types en constraints. Het `IDPTheoryObject` heeft een abstracte methode `in_theory` waarin het object omgezet wordt naar een vorm die in het `theory`-blok van een IDP-systeem kan. Gelijkaardig heeft het `IDPVocabularyObject` abstracte methodes `in_voc` en `in_struct`, die het object omzetten naar respectievelijk een vorm voor het `vocabulary` en een vorm voor de `structure`.

Tabel D.1 IDP objecten en hun mogelijke invullingen

Mogelijkheid	`Constant`	`Type`	`Predicate`	`Function`	Vervat in object:
waarde toekennen	mogelijk	mogelijk	mogelijk	mogelijk	`IDPEnumerated-Object`
returnwaarde	geen	mogelijk	geen	verplicht	`IDPFunction`
argumenten	geen	geen	verplicht	mogelijk	`IDPFunction`, `IDPPredicate`

Alle kindklassen van deze objecten verschillen voornamelijk in de vorm waarop ze deze methoden invullen. Bijvoorbeeld, elke beperking in IDP moet eindigen op een punt. Een definitie is een verzameling beperkingen, die tussen twee accolades staan. Beperkingen en definities moeten dus op een andere manier gevormd worden, en dit gebeurt in de `in_theory` methode.

Bij `IDPVocabularyObject` is meer variatie mogelijk. Zo willen we in sommige gevallen een waarde toekennen aan een object (Bv "Student = $\{1..152\}$") en in sommige gevallen net niet (wanneer we IDP een functie zelf willen laten invullen). Verder hebben sommige functiesymbolen een returnwaarde (wanneer er een ":" is) en anderen niet. Tot slot is er een verschil in het gebruik van argumenten. Tabel D.1 geeft een overzicht van alle mogelijke invullingen van de `Constant`-, `Type`-, `Predicate`- en `Function`-objecten.

De `IDP` klasse houdt ook drie soorten `Block` objecten bij: een `Structure`, een `Theory` en een `Vocabulary` object. Deze objecten stellen van de gelijknamige blokken van het IDP-systeem voor. In de `Block` klasse zijn een aantal abstracte methodes gedefinieerd om zo'n blokobject als string voor te stellen, bestaande uit een header, een begin, de inhoud en een einde. Elke blok past indien nodig deze methodes aan naar zijn eigen vorm. Zo verschillen de drie blokken bijvoorbeeld in hun header: een vocabulary start met "vocabulary `vocnaam`", een structure met "structure `structnaam`: `vocnaam`" en een theory met "theory `theorynaam`: `vocnaam`". Verder introduceert de klasse `Block` ook de `show()` methode, die een blok omzet in stringvorm aan de hand van de net vermelde methoden (zoals te zien in D.1).

Listing D.1: `show()` methode van de `Block` klasse.

```
1 def show(self, objects):
2     return (self.header() + self.begin() + self.content(objects)
3         + self.end())
```

Door bij het genereren van een blok een lijst van `IDPobjecten` mee te geven, kan de `Block`-klasse, met behulp van de methode in `content()`, alle strings in IDP vorm genereren die in het blok moeten.

Ter verduidelijking bekijken we een klein voorbeeld, weergegeven in listing D.2. In dit voorbeeld lossen we volgende letterpuzzel op: "AI + BA = CDE". De bedoeling is om elke letter een waarde te geven, en zo de formule te doen uitkomen. Bijkomend, mogen verschillende letters niet dezelfde waarde hebben, moeten klinkers een even waarde en medeklinkers een oneven waarde krijgen en mogen de eerste letters niet nul zijn.

Hiervoor definiëren we eerst een `Type Decimaal`, welke een waarde zal hebben tussen 0 en 9. Verder moeten we voor iedere letter een constante definiëren, en het predicaat `Even(Decimaal)`. De constraints zijn hier al meteen ingegeven in de vorm van het IDP systeem, om dit te signaleren is het tweede argument van `Constraint()` telkens de boolean `True`. Indien we deze niet op `True` zetten, denkt Pyidp dat we onze constraint in het Pyidp formaat gemaakt hebben, en dat deze nog moet omgezet worden. Hierover meer aan het einde van deze subsectie.

Listing D.2: Oplossing van de cde letterpuzzel in Pyidp3.

```
1  from pyidp3.typedIDP import *
2
3  idp = IDP("pad/naar/idp")
4
5  # Decimaal en letters definieren
6  idp.Type("Decimaal", list(range(0,9)))
7  idp.Constant("A: Decimaal")
8  idp.Constant("B: Decimaal")
9  idp.Constant("C: Decimaal")
10 idp.Constant("D: Decimaal")
11 idp.Constant("E: Decimaal")
12 idp.Constant("I: Decimaal")
13
14 # Definieren welke decimalen even zijn.
15 idp.Predicate("Even(Decimaal)", [0, 2, 4, 6, 8])
16
17 # Geen twee letters mogen dezelfde waarde hebben.
18 idp.Constraint("A ~= B & A ~= C & A ~= I & A ~= D & A ~= E & "
19                "B ~= C & B ~= I & B ~= D & B ~= E & C ~= I & "
20                "C ~= D & C ~= E & I ~= D & I ~= E & D ~= E", True)
21
22 idp.Constraint("A ~= 0 & B ~= 0 & C ~= 0", True)
23 idp.Constraint("Even(A) & ~Even(B) & ~Even(C) & ~Even(D) & Even(E) & Even(I)",
24                True)
25 # De formule die moet opgaan.
26 idp.Constraint("10 * A + I + 10 * B + A = 100 * C + 10 * D + E", True)
27
28 # IDP-systeem uitvoeren
29 idp.refresh()
```

Bij uitvoering van bovenstaand programma, wordt er IDP code gegenereerd en doorgegeven aan de IDP executable. Door de debugopties te gebruiken kunnen we deze data ook laten wegschrijven naar een IDP file. Deze ziet er als volgt uit:

Listing D.3: cde.idp file gegenereerd bij cde.py.

```
1 vocabulary V {
2 type Decimaal isa int
3 satisfiable()
```

```
4
5  A() : Decimaal
6  B() : Decimaal
7  C() : Decimaal
8  D() : Decimaal
9  E() : Decimaal
10 I() : Decimaal
11 Even(Decimaal)
12 }
13
14 theory T : V {
15 satisfiable().
16
17 A ~= B & A ~= C & A ~= I & A ~= D & A ~= E & B ~= C & B ~= I & B ~= D
18     & B ~= E & C ~= I & C ~= D & C ~= E & I ~= D & I ~= E & D ~= E.
19
20 A ~= 0 & B ~= 0 & C ~= 0.
21
22 Even(A) & ~Even(B) & ~Even(C) & ~Even(D) & Even(E) & Even(I)
23
24 10 * A + I + 10 * B + A = 100 * C + 10 * D + E
25 }
26
27 structure S : V {
28 Decimaal = {0; 1; 2; 3; 4; 5; 6; 7; 8}
29 Even = {0; 2; 4; 6; 8}
30 }
31
32 procedure main(){
33 stdoptions.nbmodels = 1
34 stdoptions.xsb = true
35 allsols = modelexpand(T,S)
36 print(allsols[1])
37 }
```

Dit is een letterlijke implementatie: we hebben zo goed als iedere lijn in het IDP systeem zelf moeten definiëren. Stel dat we onze puzzel moeilijker willen maken, dan zouden we per nieuwe letter een constante moeten toevoegen dat deze niet nul mag zijn. Daarnaast moet de formule die aangeeft dat elke twee verschillende letters een verschillende waarde moeten krijgen, gewijzigd worden. Dit is dus een slecht schaalbare implementatie.

Net omdat we met Pyidp/Pyidp3 in Python kunnen werken, is het mogelijk om dit veel efficiënter en schaalbaarder op een imperatieve manier te implementeren. Wanneer we hier een letter willen toevoegen, hoeven we deze enkel aan de juiste `*_letters` lijst toe te voegen en de formule aan te passen. Deze implementatie is te vinden in listing D.4.

Listing D.4: Efficiëntere implementatie van de cde puzzel.

```
letters = ["A", "B", "C", "D", "E", "I"]
nietnul = ["A", "B", "C"]
for i, letter in enumerate(letters):
    idp.Constant(letter + ": Decimaal")  # Elke constante definieren
    if letter in nietnul:
        idp.Constraint(letter + " ~= 0", True)  # Sommige letters zijn niet nul
    for j, letter2 in enumerate(letters):
        if i < j:
            # Letters mogen niet dezelfde waarde hebben
            idp.Constraint(letter + " ~= " + letter2, True)

    if letter in even_letters:  # Even letters
        idp.Constraint("Even(" + letter + ")", True)
    else:  # Oneven letters
        idp.Constraint("~Even(" + letter + ")", True)
```

Zoals eerder vermeld, biedt Pyidp ook nog een bijkomende functionaliteit, naast het vormen van een directe API. Pyipd heeft namelijk een eigen syntax om beperkingen en definities te schrijven op een Pythonische manier te schrijven. Deze syntax kan het zelf omvormen naar beperkingen en definities voor het IDP systeem. Op deze manier hoeft een Python programmeur niet de IDP syntax voor beperkingen en definities te leren. In dit werk wordt hier verder niets mee gedaan. Beperkingen en definities worden altijd gedefinieerd in IDP-vorm, om zo efficiënter te kunnen werken en makkelijker te kunnen debuggen.

D.2 Implementatie Pyidp3

D.2.1 Omgeving

Bij het programmeren van deze implementatie zal Python3 gebruikt worden. Opmerkbaar hierbij is dat we zoveel mogelijk proberen om de code leesbaar te houden door het volgen van de PEP8[2] standaard. Dit zijn richtlijnen van het `Python Enhancement Project` over quasi alles wat met gestructureerd programmeren in Python te maken heeft.

Om overzicht te houden over het project en om aan versiecontrole te doen is gebruik gemaakt van `Git`[3] en `GitLab`[4]. Bij deze laatste is er zoveel mogelijk geprobeerd om gestructureerd te werk te gaan, door het gebruik van `issues` en een `issueboard`[5]. Al de `issues` werden onderverdeeld in vier pijlers: `Bug`, `Quality Of Life (QOL)`, `Feature` en `Documentation`. Alle bugs die we tijdens het implementeren tegenkwamen, konden we zo bijvoorbeeld oplijsten en als 'todo' markeren in de `Bug` lijst. Hetzelfde geldt voor de features waarvan we nodig vonden ze te implementeren, en specifieke zaken in verband met documentatie die aangevuld moesten worden. `Issues` die onder

[2] `https://www.python.org/dev/peps/pep-0008/`
[3] `https://git-scm.com/`
[4] `https://about.gitlab.com/`
[5] `https://gitlab.com/Salt_Factory/pyidp3/boards`

QOL vallen zijn features die niet per se nodig zijn voor implementeren van Pyidp3 toepassingen, maar die de levenskwaliteit van de Pyidp3 programmeur wel kunnen verbeteren.

D.2.2 Overzetting naar Python 3

Het overzetten van de originele Pyidp naar een versie voor Python 3, gebeurde in twee stappen. Eerst hebben we gebruik gemaakt van 2to3[6]. Dit is een tool gemaakt door de ontwikkelaars van Python zelf, dat zo goed mogelijk probeert om Python 2 code om te zetten in Python 3 code. Het scant alle bestanden, haalt code die niet meer zou werken in Python 3 eruit en vervangt deze door werkende code. Hierbij produceert het een .log file van alle veranderingen, welke te vinden is in appendix D.22.

Als tweede stap hebben we manueel nog enkele kleine foutjes moeten oplossen die 2to3 niet automatisch aanpast.

- De popen functie om data door te sturen aan de hand van een Unix pipe, accepteert in Python 3 enkel byte-objecten. Elke strings die we willen doorgeven, moeten we dus eerst omgezet worden aan de hand van de encode() methode.
- Ook de output van het IDP systeem gebeurt in byte-vorm, deze moeten we decoderen aan de hand van de decode() methode.

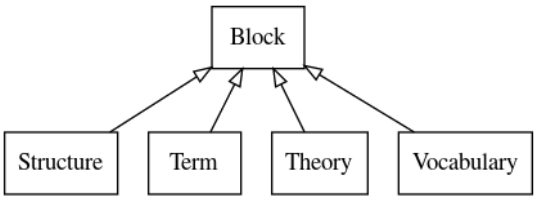
D.2.3 Nieuwe functionaliteiten

De bestaande Pyidp bevatte een beperkt aantal functionaliteiten, waardoor we genoodzaakt waren nieuwe functionaliteiten te implementeren. Op basis hiervan is een lijst gemaakt van funtionaliteiten om te implementeren in Pyidp3. Hiervoor is telkens een issue aangemaakt. Volgende features zijn allemaal toegevoegd als issues.

- Ondersteuning voor de term blok: deze is enkel nodig indien we willen minimalizeren.
- Ondersteuning voor het **constructed from** keywoord.
- Ondersteuning voor het **isa** keywoord: hiermee kunnen we supertypes aanduiden.
- Meerdere modellen per oplossing toelaten, in plaats van enkel één.
- Minimalisatie implementeren.
- Satchecken implementeren.
- Toelaten om opties van IDP-systeem in te stellen.
- Een compare methode voor het IDP object, die twee lijsten/dictionaries kan vergelijken en zo het verschil tussen twee oplossingen kan weergeven.

Op de compare methode na, zijn al deze features momenteel succesvol geïmplementeerd.

[6] https://docs.python.org/3.7/library/2to3.html



Figuur D.3: UML Schema van de `Block` klasse en zijn kinderen.

D.2.3.1 `term` blok

Zoals al eerder vermeld, ondersteunt Pyidp drie blokken: de `theory`, de `vocabulary` en de `structure`. Zolang we enkel aan modelexpansie willen doen, is dit voldoende. Maar om te minimaliseren is de `term` blok ook nodig. De implementatie hiervan was relatief voor de hand liggend. Hiervoor is er een nieuwe klasse aangemaakt, `Term`, die net zoals de andere drie blokken overerft van de `Block` klasse, zoals te zien in figuur D.3.

In de `term` blok hoefden we enkel aan te passen hoe zo'n blok gevormd moest worden, en ervoor zorgen dat we de te minimaliseren term als parameter kunnen meegeven. Deze kan vervolgens gebruikt worden om te minimaliseren (zie D.2.3.5 voor meer).

D.2.3.2 constructed from

In het IDP-systeem kan gebruik gemaakt worden van het keywoord **constructed from** om de waarden van types die voor alle structuren gelijk moeten zijn vast te leggen. Dit is gedemonstreerd in listing D.5

Listing D.5: Voorbeeld van gebruik van het keywoord constructed from in het IDP-systeem.

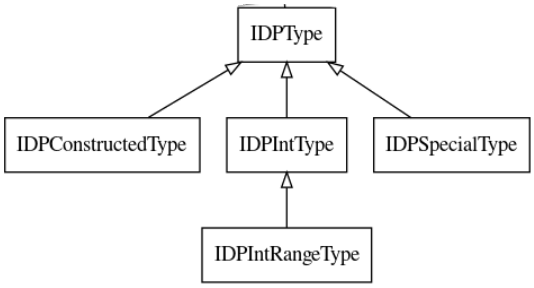
```
1 vocabulary V {
2     type Weekend constructed from {Zaterdag, Zondag}
3 }
```

In Pyidp is er geen ondersteuning om dit soort types te declareren. Om dit op te lossen in Pyidp3 hebben we een nieuwe klasse `IDPConstructedType` gemaakt, welke overerft van de `IDPType` klasse. Deze overerving is gevisualiseerd in figuur D.4. Een `IDPConstructedType` wordt aangemaakt wanneer er bij het aanmaken van het type de `constructed_from` boolean op `True` staat. In listing D.6 is weergegeven hoe het vorige voorbeeld er in Pyidp3 zou uitzien.

Listing D.6: Voorbeeld van implementatie het keywoord constructed from in Pyidp3.

```
1 dagen_van_het_weekend = ["Zaterdag", "Zondag"]
2 idp.Type("Weekend", dagen_van_het_weekend, constructed_from=True)
```

Om dit concreet te implementeren hebben we de `in_voc` en `in_struct` methodes die de nieuwe klasse overerft van de `IDPType` klasse herschreven zodat ze de juiste stringwaarden teruggeven. De methode `in_voc` geeft de typenaam, "constructed from" en de waarden terug, en de methode `in_struct` geeft helemaal niets terug omdat deze bij **constructed from** leeg is.



Figuur D.4: UML Schema van de `Type` klasse en zijn kinderen.

D.2.3.3 isa

Gelijkaardig aan **constructed from**, is **isa** een keywoord dat gebruikt kan worden bij het definiëren van een type. In het IDP-systeem wordt dit keywoord gebruikt om een supertype aan te duiden. Zo wordt bijvoorbeeld in de code om de verdere verdeling te berekenen het type `Getal` als een subtype van `int` gedefinieerd, en het type `Prioriteit` als een subtype van `Getal`. Zoals reeds vermeld heeft dit tot voordeel om de waarden van `Prioriteit` af te bakenen op de waarden van `Getal`.

Listing D.7: Voorbeeld van gebruik van het keywoord isa in het IDP-systeem.

```
1 type Getal isa int
2 type Prioriteit isa Getal
```

In de oorspronkelijke Pyidp wordt een type waarvan alle waarden een getal zijn automatisch van het type `IDPIntType`. Dit houdt in dat het in de vocabulary wordt aangevuld met "`isa int`". Types met andere waarden dan uitsluitend natuurlijke getallen, worden gevormd zonder een supertype. In Pyidp is het voorkomen van het keywoord `isa` dus enkel mogelijk wanneer het om integers gaat.

Om Pyidp3 uit te breiden met functionaliteit en omdat er concreet nood aan is in dit werk, biedt deze wel ondersteuning voor gebruik van **isa** met andere supertypes dan enkel `int`. Hiervoor hebben we de klasse `IDPSpecialType` toegevoegd (zie figuur D.4). Dit is een klasse die overerft van de algemene `Type` klasse, en waarvan de `in_voc` methode is aangepast om correct het keywoord en de supertype te genereren in de `vocabulary`.

Wanneer in Pyidp3 een type gedefinieerd wordt, kan een supertype meegegeven worden en de **isa** vlag op `True` gezet worden, om te signaleren dat er een subtype als argument meegegeven is. Een voorbeeld hiervan is gegeven in listing D.8.

Listing D.8: Voorbeeld van implementatie het keywoord isa in Pyidp3.

```
1 idp.Type("Getal", (0, 10000))
2 idp.Type("Priority", "Number", isa=True)
```

D.2.3.4 Meerdere modellen per oplossing

In Pyidp wordt het hele IDP-systeem ge-abstraheerd in de `IDP` klasse. Wat wil zeggen dat niet enkel de inputvariabelen maar na het uitvoeren van het IDP-systeem, ook de outputvariabelen hierin terecht komen. Alle output gegenereerd door het IDP-systeem wordt lijn per lijn omgezet in Python en vervolgens aan de hand van een keywoord toegevoegd aan het `__dict__` attribuut van het IDP object. Dit is een speciaal dictionary attribuut[7] dat elke klasse in Python bezit. Het keywoord dat gebruikt wordt, is de naam van de variabele dat Pyidp inleest. Door intelligent gebruik te maken van de `__setattr__` magische methode, kunnen we deze dictionary automatisch laten vullen. Wanneer we bij het IDP object een nog niet-bestaand attribuut een waarde willen geven, wordt deze door `__setattr__` toegevoegd aan de `__dict__` dictionary. Zo hoeft een gebruiker enkel maar de naam van deze variabele op te zoeken in de `dictionary` om de waarde(n) van de variabele in de oplossing te bekomen. Een voorbeeld van het opvragen van de waarde van een symbool is weergegeven in listing D.9.

Listing D.9: Voorbeeld van het uitlezen van een oplossing in Pyidp.

```
1 ...
2 idp.Function("Oplossing(Rij, Kolom): Getal")
3 ...
4 idp.model_expand()
5 print("Oplossing:", idp['Oplossing'])
```

Doordat alle output van het IDP-systeem in deze ene `dictionary` terecht komt, is er telkens maar ruimte om één model als oplossing op te slaan. In het geval dat we er meerdere zouden genereren (door de tot nu toe vast gecodeerde main aan te passen) zou Pyidp alle output lezen, deze omzetten in Pythonische vorm, en vervolgens op basis van de naam van de gelezen variabele deze opslaan in de dictionary. Maar omdat er hier meerdere variabelen dezelfde naam hebben, worden de waarden van de variabele telkens overschreven met die van de variabele in de volgende oplossing.

Om dit te voorkomen hebben we Pyidp3 aangepast zodat het deze waarden niet in het IDP-systeem opslaat, maar per model in een aparte dictionary, waarna het deze tesamen teruggeeft in de vorm van een lijst. Listing D.10 toont hoe we met deze aanpassing de oplossing(en) van een model expansie kunnen berekenen, en printen.

Listing D.10: Voorbeeld van het uitlezen van een oplossing in Pyidp3.

```
1 ...
2 idp.Function("Oplossing(Rij, Kolom): Getal")
3 ...
4 oplossingen = idp.model_expand()
5 for i, opl in enumerate(oplossingen):        # Itereer over oplossingen.
6     print("Oplossing {:d}:".format(index))   # Print ranggetal.
7     print(opl["Oplossing"])                  # Print waarde van "Oplossing".
```

[7] https://docs.python.org/3/reference/datamodel.html

De `main` gebruikt in het IDP-systeem, is hier ook aangepast, om tussen twee modellen de string ”next:” te printen. Op deze manier kunnen we de output eerst splitsen op deze string, om zo een lijst van al onze verschillende modellen te bekomen.

D.2.3.5 Minimalisatie

In alle vorige voorbeelden in dit hoofdstuk werd het IDP-systeem telkens gebruikt om aan model expansion te doen, met andere woorden er worden regels opgelegd en startwaarden aangegeven om vervolgens het IDP-systeem te laten proberen deze aan te vullen tot een volwaardig model. Het IDP-systeem kan meer dan enkel model expansie, zoals minimalisatie en satchecking, maar Pyidp ondersteunt deze niet.

Omdat er in de eindapplicatie van dit werk nood is aan minimalisatie, is deze functionaliteit toegevoegd aan Pyidp3. Hiervoor is de manier waarop Pyidp3 zijn idp-file genereert aangepast. Er zijn twee methoden toegevoegd: `minimize_script` en `minimize`. De eerste methode genereert de string die nodig is om naar het IDP-systeem te sturen, en geeft deze terug. Het heeft als argument een `Term` blok klasse, waarin de te minimaliseren term zich bevindt. De `minimize` functie genereert de string aan de hand van de `minimize_script` methode, geeft deze door aan de IDP executable en leest vervolgens de output van het IDP-systeem weer uit. Deze output zet het dan om in een lijst van modellen, om ze daarna terug te geven. Wanneer er niet expliciet de vlag ”-O” wordt meegegeven om de debug code te negeren, wordt de string gegenereerd door `minimize_script` ook weggeschreven naar `IDPscript.txt`.

In listing D.11 is een codesnipper weergegeven van hoe minimalisatie gebruikt wordt in de `group_assign` testfile (zie sectie D.3.

Listing D.11: Voorbeeld van het gebruik van minimalisatie in Pyidp3.

```
1 ...
2 idp.Constant("Totaal: Getal")  # Totaal defini"eren.
3 # Totaal berekenen.
4 idp.Constraint("Totaal = TotUnsatSamen + TotUnsatNSamen + ... ")
5 ...
6 oplossingen = idp.minimize("Totaal")
7 for i, opl in enumerate(oplossingen):        # Itereer over oplossingen.
8     print("Oplossing {:d}:".format(index))    # Print ranggetal.
9     print(opl["NieuwInGroep"])                # Print nieuwe groepsverdeling.
```

D.2.3.6 Satchecken

Net zoals voor minimalisatie, was er voor satchecking geen ondersteuning in de oorspronkelijke Pyidp. Dit is noodzakelijk voor dit werk, maar is wel toegevoegd voor de volledigheid. Deze implementatie van deze inferentietaak werkt volledig gelijkaardig aan de manier waarop we minimaliseren. Er zijn twee nieuwe methoden: `check_sat_script` om de string te vormen, en `check_sat` om deze door te geven aan het IDP-systeem en de output ervan te lezen.

D.2.3.7 Opties in het IDP-systeem

In het IDP-systeem is het mogelijk om verschillende opties te gebruiken. De in dit werk meest gebruikte opties zijn `nbmodels`, `verbosity.solving` en `mxtimeout`. In Pyidp zijn de opties hard-coded: `nbmodels` is altijd gelijk aan 1, en `xsb` is altijd gelijk aan `True`.

Echter, in dit werk hebben we in sommige gevallen nood aan meerdere modellen per oplossing. Hiervoor zouden we `nbmodels` moeten kunnen veranderen. Daarom hebben we de functionaliteit om opties in te stellen toegevoegd. Bij het aanmaken van een IDP object, wordt het `_options_dict` geinitialiseerd. Dit is een `dictionary` gevuld met telkens als keywoord de naam van een optie, en als waarde na initialisatie `None` (op `nbmodels` en `xsb` na).

Deze opties kunnen veranderd worden door hun 'attribuut' in de IDP klasse een waarde te geven. Dankzij de `__setattr__` methode (zie sectie D.2.3.4), kunnen we nagaan of dit attribuut voorkomt in onze `_options_dict`, om vervolgens deze hieraan toe te voegen.

De nieuwe methode `generate_options` genereert de string die nodig is om deze opties in het IDP-systeem te zetten. Het gaat per keywoord in de `_options_dict` na of het een waarde gekregen heeft (en dus niet None is). Indien het een waarde heeft, voegt het "stdoptions.keywoord = waarde" aan een string toe. Na het overlopen van alle opties, geeft het deze string terug. De methodes die gebruikt worden om scripts te genereren zijn ook allemaal aangepast om deze `generate_options` methode te gebruiken bij het aanmaken van hun mainmethode.

In listing D.12 is weergegeven hoe we de opties in Pyidp3 kunnen gebruiken.

Listing D.12: Voorbeeld van het gebruik van opties in Pyidp3.

```
1 ...
2 idp.mxtimeout = 300  # Maximum timeout van 5 minuten.
3 idp.nbmodels = 5  # We willen vijf models in onze oplossing.
4 idp.verbosity_solving = 1  # Verbositeit van het solving gedeelte.
5 ...
```

D.2.3.8 De compare methode

Nu dat het in Pyidp3 mogelijk is om meerdere modellen te genereren die allemaal een oplossing zijn van hetzelfde probleem, kunnen we ons de vraag stellen wat nu net het verschil is tussen deze modellen. Om deze vraag gemakkelijker te kunnen beantwoorden, hebben we de `compare` methode toegevoegd. Dit is een vrij simpele methode die het verschil tussen twee `enumerations` (momenteel enkel `dictionaries`) nagaat, en deze teruggeeft en print. Op deze manier hoeft een gebruiker deze methode niet zelf te implementeren.

Het gebruik van de `compare` methode is vrij voor de hand liggend. De twee te vergelijken `dictionaries` worden meegegeven aan de methode. Een voorbeeld is weergegeven in listing D.13 en de output ervan in listing D.14.

Listing D.13: Voorbeeld van het gebruik van compare in Pyidp3.

```
1 idp.compare({1:'een', 2:'twee', 3:'drie'}, {1:'een', 2:'drie', 3:'twee'})
```

Listing D.14: Output van voorbeeld D.13.

```
1 key     | enum1     | enum2
2       2| twee      | drie
3       3| drie      | twee
```

D.2.4 Quality Of Life

Alles wat onder `QOL` valt, zijn de features die niet nodig zijn om Pyidp3 te doen werken, maar die wel de levenskwaliteit van de Pyidp3 programmeur verbeteren.

Deze QOL verbeteringen zijn:

- toevoegen van PEP8 conformiteit;
- `IDPIntTypes` mogelijkheid tot range van waarden geven;
- geautomatiseerd testen (zie D.3);
- rudimentaire IDP naar .py converteerder geschreven;
- enkel een '.' toevoegen aan een constraint wanneer dit nodig is;
- genereren van een UML schema;
- toevoegen van de Pyidp3 module aan de `Python Package Index`.

D.2.4.1 range bij `IDPIntTypes`

Bij het definiëren van een type in het IDP-systeem kunnen we deze in de `structure` een waarde geven. Indien deze waarde een range is van getallen, kunnen we deze verkort noteren aan de hand van "start..einde". Een voorbeeld hiervan is gegeven in listing D.15.

Listing D.15: Toekennen van range aan type in IDP-systeem.

```
1 Getal1 = {1; 2; 3; 4; 5; 6; 7; 8; 9; 10}
2 Getal2 = {1..10} // Beide zijn equivalent!
```

Om in Pyidp een range van waarden te geven aan een type, kunnen we de `range` functie gebruiken. Het nadeel hiervan is dat Pyidp automatisch een lijst van alle waarden in deze range maakt. De code die het genereert voor het IDP-systeem is dus zoals die van `Getal1` in listing D.15. Dit is normaal geen probleem omdat deze code niet zichtbaar is voor de gebruiker, maar wanneer we als programmeur deze code ook laten wegschrijven naar een IDP bestand om ze te debuggen, dan staan hier overbodig veel getallen in. In het scenario dat er een range nodig is van 1 tot 1000000,

zou dit een lijst maken met daarin elk getal tussen deze twee waarden. Dit maakt de code minder leesbaar.

Daarom is er in Pyidp3 de nieuwe klasse `IDPIntRangeType` toegevoegd, welke overerft van van `IDPIntTypes` (zie D.4). Deze nieuwe klasse accepteert een `tuple` bestaande uit de startwaarde en de eindwaarde als argument, en gebruikt deze om in zijn `in_struct` methode een range te genereren aan de hand van "..". Het gebruik hiervan is gedemonstreerd in listing D.16. De IDP code die gevormd zou worden bij het uitvoeren van deze listing, is dezelfde als de code in D.15.

Listing D.16: Definiëren van types met range in IDP-systeem.

```
1 idp.Type("Getal1: int", list(range(1,10)))
2 idp.Type("Getal2: int", (1, 10)) # Beide zijn equivalent in werking!
```

D.2.4.2 rudimentaire IDP naar .py converteerder

Tijdens dit werk hebben we een oppervlakkige IDP naar .py converteerder gemaakt. Op deze manier hoefden we niet telkens manueel deze overgang te maken. De converteerder splitst eerst de IDP file op in de 4 of 5 grote stukken: `vocabulary`, `structure`, `theory`, `main` en indien aanwezig de `term`.

Deze worden dan elk doorgegeven aan een voor hen specifieke *parser*. Deze leest en interpreteert lijn per lijn wat er staat. Per soort variabele houdt de converteerder een *dictionary* bij. In deze dictionary houdt de converteerder voor elk element de naam bij en, indien van toepassing, de waarden, de returnwaarde, de argumenten en/of de speciale keywoorden die de converteerder tegenkomt.

Aan de hand van al deze *dictionaries* genereert het de .py code, door per *dictionary* te overlopen wat erin zit, en dit in de vorm van Pyidp3 weg te schrijven.

Momenteel is deze converteerder nog experimentele code, en bevindt het zich op de `QOL/ISSUE-12` branch van het Pyidp3 *GitLab* repository.

D.2.4.3 Enkel een punt indien nodig

In het IDP-systeem eindigt iedere beperking en definitieregel op een punt, om aan te geven dat het niet verderloopt op de volgende lijn. De oorspronkelijke Pyidp plaats automatisch bij definiëren van een `Constraint` een punt achteraan de lijn. Wanneer de gebruiker in het argument van `Constraint` al een punt geplaatst heeft, plaatst Pyidp alsnog een punt aan de lijn, waardoor het niet meer leesbaar is voor het IDP-systeem. Definities daarentegen moeten wel expliciet eindigen op een punt, want hier voegt Pyidp er geen automatisch aan toe.

In Pyidp3 is deze inconsistentie verholpen door telkens na te kijken of het nodig is om een punt te plaatsen. Indien al een punt aanwezig is, wordt er geen extra bijgevoegd. Dit wordt gedaan voor zowel definities als beperkingen.

D.2.4.4 UML schema

Omdat het altijd handig is om een UML schema bij de hand te hebben tijdens het werken aan een module, hebben we een automatisch UML klassediagramma laten genereren van Pyidp3. Het Python linterprogramma *Pylint*[8] biedt een script genaamd *pyreverse* aan dat een module kan tekenen in een UML diagramma, aan de hand van *Graphviz*[9]. Het genereren is gebeurd aan de hand van het commando te zien in listing D.17. De afbeelding die hierdoor gegenereerd wordt, is te zien in figuur D.5. Om het leesbaar te houden, zijn enkel de klassenamen weergegeven. Bemerk dat hier alle klassen om python syntax om te zetten in IDP zijn weggelaten, omdat deze niet meer ondersteund worden in Pyidp3.

Listing D.17: `pyreverse` commando om UML te genereren.

```
1 $ pyreverse -Sk --output=png pyidp3
```

D.2.4.5 *Python Package Index*

De *Python Package Index* (PyPi) is een *repository* voor Pythonmodules. Software-ontwikkelaars kunnen hier hun modules uploaden, en softwaregebruikers kunnen deze vervolgens downloaden en automatisch laten installeren door Python. Door onze Pyidp3 module te uploaden in PyPi kunnen we niet enkel makkelijker in een nieuwe omgeving aan de module werken, maar is de module toegankelijker voor de gemiddelde gebruiker. Het is niet langer nodig om telkens de broncode van *GitLab* te downloaden, en deze te installeren in de juiste folder om de software te gebruiken.

Om Pyidp3 te installeren volstaat het nu om op de meeste besturingssystemen het commando in listing D.18 uit te voeren.

Listing D.18: Pyidp3 downloaden en installeren via PyPi.

```
1 $ python3 pip install pyidp3
```

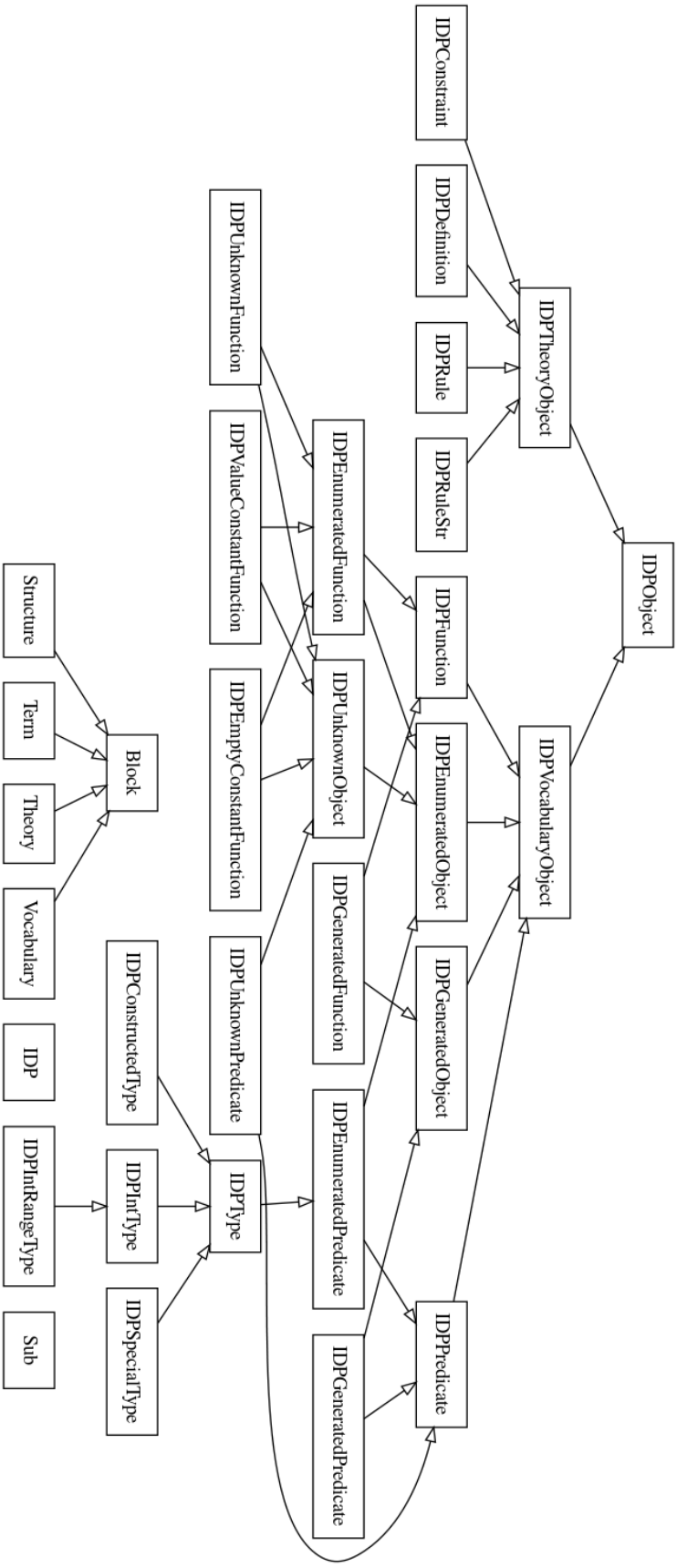
D.2.5 Bugfiksen

Gedurende de implementatie van alle features, is er gewerkt aan het oplossen van gevonden bugs in Pyidp.

Deze bugs zijn:

- “string“ werd niet herkend als mogelijk type;
- “nat“ werd niet herkend als mogelijk type;
- String moest met extra aanhalingstekens gegeven worden;
- Constanten kunnen nu een waarde toegewezen krijgen.

[8] `https://www.pylint.org/`
[9] `https://graphviz.org/`



Figuur D.5: UML schema van Pyidp3.

Om de bug te fiksen dat constanten geen waarde konden krijgen, hebben we de nieuwe klassen `IDPValueConstantFunction` en `IDPEmptyConstantFunction` toegevoegd. Deze zijn beide zichtbaar in het UML klassediagramma in figuur D.5 op pagina 94.

D.2.6 Documentatie

Om de Pyidp3 module zo duidelijk en zo bruikbaar mogelijk te houden, hebben we in dit werk er ook uitvoerige documentatie voor geschreven. De voor de hand liggende manier van documenteren is om zogenaamde *docstring* te gebruiken. Een *docstring* is een stukje commentaar dat gebruikt kan worden om modules, functies, klassen of methodes te beschrijven. Belangrijk hierbij is dat het telkens het eerste statement moet zijn van hetgene dat de *docstring* wil beschrijven. Niet enkel maakt zo'n commentaar de code leesbaarder, het laat ook toe om eenvoudige, automatische documentatie te genereren van de volledige module en zijn inhoud. Conventies rond *docstring* staan beschreven in PEP 257[10].

In dit werk gaan we nog een stapje verder dan zomaar *docstring* te gebruiken: we gaan namelijk al onze docstring in het *reStructuredText*[11]-formaat schrijven, zoals wordt aangeraden in PEP 287[12]. *reStructuredText* is een markup taal, zoals HTML of Markdown, wat ons toelaat onze documentatie op te maken. Het voornaamste voordeel hiervan is dat we nu gebruik kunnen maken van de *Sphinx*[13] documentatie generator. Deze kan onze code scannen, overal de docstrings interpreteren, en deze mooi formatteren tot een API referentiehandleiding, zonder dat we hiervoor extra werk moeten verheffen.

Sphinx laat ook toe om eigen pagina's te schrijven. Dit kan handig zijn om zaken zoals een inleiding, een overzicht van functionaliteiten, een pagina met voorbeeldcode, enz toe te voegen aan onze documentatie. Op deze manier kunnen we een handleiding tot onze module schrijven. Omdat deze module open-source is, mogen we onze handleiding gratis laten hosten door *ReadTheDocs*[14]. Zo is de volledige documentatiehandleiding van de Pyidp3 module te vinden op `https://pyidp3.readthedocs.io/`. De handleiding bestaat uit een inleiding, een oplijsting van features, codevoorbeelden, de stappen die ondernomen zijn tijdens het overzetten en de Pyidp3 referentiehandleiding. Door *ReadTheDocs* te linken met onze *GitLab*-pagina, genereert het na iedere aanpassing aan de code automatisch een vernieuwde versie van de documentatie. Er is ook een pdf versie van deze documentatie, deze is op de USB-stick terug te vinden.

D.3 Testen

Een belangrijk onderdeel bij het schrijven van goed werkende code, is natuurlijk grondig testen. Voor dit werk is gebruik gemaakt van *Continous Integration/Continuous Delivery* (CI/CD). Dit houdt

[10] `https://www.python.org/dev/peps/pep-0257/`
[11] `http://docutils.sourceforge.net/rst.html`
[12] `https://www.python.org/dev/peps/pep-0287/`
[13] `http://www.sphinx-doc.org/en/master/`
[14] `https://readthedocs.org/`

in dat we niet enkel op het einde van onze implementatie testen uitvoeren, maar dat we gedurende het hele implementatie-proces testen uitvoeren. Meer nog, deze testen gebeuren volautomatisch. Hiervoor biedt *GitLab* de CI/CD pijplijn aan. *GitLab* laat toe om gratis een 3000-tal minuten per maand testen uit te voeren op hun eigen hardware. Zo kunnen we na het configureren van onze testfile *GitLab* na ieder keer dat we code uploaden, deze automatisch laten testen op eventuele fouten. Dit bespaart ons de moeite van telkens bij iedere aanpassing te moeten kijken of we geen andere functionaliteiten per ongeluk buiten dienst gebracht zouden hebben.

GitLab geeft geen garantie dat alle testen altijd op dezelfde hardware gebeuren, en aangezien deze hardware door veel verschillende projecten gedeeld wordt, kan het ook geen garantie geven dat de software-omgeving altijd hetzelfde is. Daarom maakt het gebruik van *Docker*[15] en draait het alle testen in een *Docker* container. Dit is een softwarepakket met voorbereide code en afhankelijkheden dat toelaat om bepaalde processen (in ons geval de testen) volledig geïsoleerd van de rest van het systeem te laten draaien. Op deze manier ondervinden onze testen geen invloed van het verschil in hardware, of van het verschil in software-omgeving.

Voor dit werk was er dus nood aan een *Docker* container met daarop een linuxomgeving aanwezig, met in deze linuxomgeving alle benodigdheden voor het IDP-systeem. De KRR groep van de KU Leuven heeft al drie *Docker* containers met daarin het IDP-systeem geinstalleerd, maar deze voeren allen een service bovenop het IDP-systeem uit. Twee van de containers draaien de webIDE van IDP, en de derde draait de autoconfig software van KRR. Voor dit werk hebben we nood aan een container met puur IDP, zonder extra diensten die automatisch starten. Daarom hebben we besloten een eigen container te maken.

Een *Docker* container wordt gemaakt door een Dockerfile aan te maken (zie listing D.19). In dit bestand geven we eerst weer welk besturingssysteem we als basis willen gebruiken, in dit geval is dat Ubuntu. Vervolgens gebruiken we de pakketbeheerder van Ubuntu, *apt*, om de pakketten *wget* en *git* te installeren. Aan de hand van *wget*, downloaden we dan de bestanden van het IDP-systeem, om deze vervolgens te openen en te verplaatsen naar de `$HOME` folder (welke zich in /root/ bevindt). Tot slotte is het belangrijk om de omgevingsvariabele `LANG` op `C.UTF-8` te zetten, omdat deze standaard op ASCII staat en dus geen Unicode ondersteunt (een probleem dat pas na lang zoeken opgelost was).

Listing D.19: Dockerfile voor pure-IDP.

```
1  FROM ubuntu
2  MAINTAINER Salt_Factory
3  LABEL Remarks="This is a dockerfile for my testing environment using IDP"
4  RUN apt update && \
5  apt -y install wget git
6
7  RUN wget https://dtai.cs.kuleuven.be/krr/files/releases/idp/idp-linux-latest.tar.gz
8  RUN tar -xf idp-linux-latest.tar.gz
9  RUN rm idp-linux-latest.tar.gz
10 RUN mv idp* /root/idp
```

[15] https://www.docker.com/

```
11 ENV LANG=C.UTF-8
12 ENV LC_MESSAGES=POSIX
```

Deze Dockerfile is dan omgezet in een *Docker* container, en vervolgens geupload naar de *Docker Hub*. Van hieruit kunnen wij, maar ook andere gebruikers de container downloaden via de naam "saltyfactory/pure-idp"[16].

Nu we onze *Docker* container gemaakt en geupload hebben, moeten we nog onze testomgeving op *GitLab* opzetten. Dit gebeurt aan de hand van een *.gitlab-ci.yml* bestand. Het bestand gemaakt voor dit werk is te vinden in listing D.20. Hier definiëren we eerst welke *Docker* container we willen gebruiken en welke *stages* nodig zijn. In dit geval hebben we maar 1 *stage*: **test**.

Listing D.20: *.gitlab-ci.yml* bestand voor de CI/CD omgeving van *GitLab*.

```
1  image: saltyfactory/pure-idp
2
3  stages:
4      - test
5
6  before_script:
7      - apt update
8      - apt -y install python3 python3-pip
9      - python3 setup.py install
10
11 testing:
12     stage: test
13     script:
14         - pwd
15         - cd tests
16         - chmod +x starttest.sh
17         - ./starttest.sh
```

Vooraleer we beginnen aan de testing *stage*, installeren we eerst python3, python3-pip en vervolgens de eigen Pyidp3-module door de zelfgemaakte setup.py file te installeren. Het testen zelf is niet meer dan het uitvoeren van een *Bash* script, genaamd starttest.sh. Deze is te vinden in listing D.21.

In dit bestand maken we eerst een lijst *files* met daarin de namen van al onze testbestanden. Hierna voeren we deze allemaal eens uit, kijken we naar de output, en doen we `exit -1` wanneer er meer dan één keer het keywoord **Error** gevonden is. Er is altijd één voorkomen van dit woord wanneer er geen error is.

Listing D.21: Bestand dat alle testen uitvoert.

```
1  #!/bin/bash
2
3  files="simple_inference.py tougher_inference.py definition_test.py
```

[16] https://hub.docker.com/r/saltyfactory/pure-idp

```
4  constructed_from.py small_initial.py big_initial.py group_assign.py
5  sudoku.py masyu.py"
6
7  for file in $files; do
8      echo $file
9      out=$(./$file 2>&1)
10     echo $out
11     error_amount=$(echo $out | grep -E 'Error' | wc -l )
12     if  [[ ($error_amount -ne 1) ]]; then
13         echo "$error_amount errors!"
14         exit -1
15     fi
16     echo "all good!"
17 done
18 exit 0
```

De gebruikte testfiles zijn telkens implementaties van Pyidp3. `small_initial.py`, `big_initial.py` en `group_assign.py` zijn de Pyidp3 vormen van de idp-files die in het vorige hoofdstuk besproken zijn (de initiële verdeling en de incrementele verdeling). Zo weten we dat als deze allemaal succesvol werken, Pyidp3 ten minste alles wat we nodig hebben voor dit werk ondersteunt. De overige tesfiles zijn implementaties van oefeningen uit de handleiding van het IDP labo (Mitchell (1995b)).

Dankzij de CI/CD pijplijn, konden we gedurende deze volledige implementatiestap met een gerust hart programmeren, wetende dat als we een fout introduceerden dit niet door onze pijplijn zou geraken. Doordat tijdens dit werk gebruik gemaakt is van een *branch* per *issue*, konden we de regel stellen dat code enkel in de *masterbranch* mocht gemerged worden wanneer deze geen fouten vertoonde. Op deze manier bevat onze *masterbranch* altijd werkende code.

D.4 Overzetting

Listing D.22: De logfile geproduceert door *2to3* bij het overzetten van Pyidp.

```
1  --- idpobjects.py       (original)
2  +++ idpobjects.py       (refactored)
3  @@ -55,7 +55,7 @@
4
5      def product_of_types(self):
6          import itertools
7  -       types = map(lambda x: getattr(self.idp,x), self.typing)
8  +       types = [getattr(self.idp,x) for x in self.typing]
9          return itertools.product(*types)
10
11 class IDPEnumeratedObject(IDPVocabularyObject):
12 @@ -96,7 +96,7 @@
13
14     def product_of_types(self):
15         import itertools
16 -       types = map(lambda x: getattr(self.idp,x), self.typing)
17 +       types = [getattr(self.idp,x) for x in self.typing]
```

```
18          return itertools.product(*types)
19
20      @property
21  @@ -162,7 +162,7 @@
22          IDPGeneratedObject.__init__(self,idp,name,args,impl)
23
24      def __getitem__(self, key):
25  -       args = map(self.idp.object_for_name, key)
26  +       args = list(map(self.idp.object_for_name, key))
27          try:
28              res = self.implementation(args)
29          except AttributeError:
30  @@ -275,4 +275,4 @@
31          self.rules = rule_list
32
33      def in_theory(self):
34  -       return "{\n" + "\n".join(map(lambda x: x.in_theory(), self.rules)) + "\n}"
35  +       return "{\n" + "\n".join([x.in_theory() for x in self.rules]) + "\n}"
36  --- idp_parse_out.py    (original)
37  +++ idp_parse_out.py    (refactored)
38  @@ -126,7 +126,7 @@
39   def parse_tuple(tup):
40       tup = tup.strip()
41       elements = tup.split(',')
42  -    parsed = map(parse_element, elements)
43  +    parsed = list(map(parse_element, elements))
44       if len(parsed) == 1:
45           return parsed[0]
46       return tuple(parsed)
47  @@ -144,7 +144,7 @@
48   def parse_enumeration(s):
49       s = s.strip()
50       elements = s.split(';')
51  -    parsed = map(parse_enumerated, elements)
52  +    parsed = list(map(parse_enumerated, elements))
53       if "->" in s: # Function
54           return dict(parsed)
55       else: # Predicate
56  @@ -153,7 +153,7 @@
57   def parse_range(s):
58       s = s.strip()
59       low, up = s.split('..')
60  -    return range(int(low),int(up))
61  +    return list(range(int(low),int(up)))
62
63   def parse_contents(s):
64       stripped = s.strip().lstrip('{').rstrip('}').strip()
65  --- idp_py_syntax.py    (original)
66  +++ idp_py_syntax.py    (refactored)
67  @@ -44,7 +44,7 @@
68          self.formula = formula
69
70      def __str__(self):
71  -       var_tuple =  flatten(map(lambda x: x[0], self.vars))
72  +       var_tuple =  flatten([x[0] for x in self.vars])
73          it = (" ".join(var_tuple) + ": " +
74                  " & ".join(map(tuple_to_atom, self.vars)))
75          if self.agg == "card":
76  @@ -63,10 +63,10 @@
```

```
77              symbols = "+−*%/^"
78              return "(" + x + ")" if any([(s in x) for s in symbols]) else x
79          if self.symbol == "/":
80  −           (l,r) = map(lambda x: add_pars(str(x)), self.children)
81  +           (l,r) = [add_pars(str(x)) for x in self.children]
82              return "(" + l + "−" + l + "%" + r + ")␣/␣" + r
83          else:
84  −           return ("␣" + self.symbol + "␣").join(map(lambda x: add_pars(str(x)), self.children))
85  +           return ("␣" + self.symbol + "␣").join([add_pars(str(x)) for x in self.children])
86
87   class BooleanFormula(Formula):
88
89  @@ −75,7 +75,7 @@
90          self.children = children
91
92      def __str__(self):
93  −       return ("␣" + self.symbol + "␣").join(map(lambda x: "(" + str(x) + ")", self.children))
94  +       return ("␣" + self.symbol + "␣").join(["(" + str(x) + ")" for x in self.children])
95
96   class UnaryFormula(Formula):
97
98  @@ −100,7 +100,7 @@
99              return "&"
100
101     def __str__(self):
102 −       var_tuple =  flatten(map(lambda x: x[0], self.vars))
103 +       var_tuple =  flatten([x[0] for x in self.vars])
104         return (self.kind + "␣" + "␣".join(var_tuple) + ":␣" +
105                 "␣&␣".join(map(tuple_to_atom, self.vars)) +
106                 self.guard_sym() + "␣"+ str(self.formula))
107 @@ −179,7 +179,7 @@
108         return UnaryFormula(symb, self.visit(node.operand))
109
110     def visit_Tuple(self, node):
111 −       return tuple(map(lambda x: self.visit(x), node.elts))
112 +       return tuple([self.visit(x) for x in node.elts])
113
114     def visit_GeneratorExp(self,node):
115         return self.visit_ListComp(node)
116 @@ −199,7 +199,7 @@
117             symb = "?" if func == "any" else "!"
118             return QuantifiedFormula(symb, *self.visit(node.args[0]))
119         aggregates = { 'sum' : 'sum', 'len' : 'card', 'product' : 'prod', 'max' : 'max', 'min' :
120                        'min' }
121 −       if func in aggregates.keys():
122 +       if func in list(aggregates.keys()):
123             return AggregateFormula(aggregates[func], *self.visit(node.args[0]))
124         return str(func) + "(" + ",␣".join(map(str, [self.visit(arg) for arg in node.args])) + ")"
125
126 −− test.py     (original)
127 +++ test.py     (refactored)
128 @@ −1,11 +1,11 @@
129  #!/usr/bin/python3
130 −from typedIDP import *
131 +from .typedIDP import *
132
133
134  idp = IDP('/home/saltfactory/Documents/Masterproef/IDP/idp3−3.7.1−Linux/usr/local/bin/idp')
135
```

```
136 -idp.Type("Student", range(int(5)))
137 -idp.Type("Groepnummer",range(int(1)))
138 +idp.Type("Student", list(range(int(5))))
139 +idp.Type("Groepnummer",list(range(int(1))))
140  idp.Function("InGroep(Student): Groepnummer")
141  idp.Constraint("""!groep[Groepnummer]: #{student[Student]:
142 ␣␣␣␣␣␣␣␣␣␣␣␣␣␣␣␣␣␣␣␣␣␣␣␣InGroep(student) = groep} =< """ + str(10),True)
143 --- typedIDP.py (original)
144 +++ typedIDP.py (refactored)
145 @@ -1,8 +1,9 @@
146  IDP_LOCATION = "/home/saltfactory/Documents/Masterproef/IDP/idp3-3.7.1-Linux/usr/local/bin/idp"
147
148 -from idp_py_syntax import parse_formula
149 -
150 -from idpobjects import *
151 +from .idp_py_syntax import parse_formula
152 +
153 +from .idpobjects import *
154 +from functools import reduce
155
156  class Block(object):
157
158 @@ -56,7 +57,7 @@
159          return "vocabulary " + self.name
160
161  def subclasses(cls):
162 -    return reduce(lambda x,y: x + y, map(lambda x: subclasses(x) + [x], cls.__subclasses__()), [])
163 +    return reduce(lambda x,y: x + y, [subclasses(x) + [x] for x in cls.__subclasses__()], [])
164
165  class IDP(object):
166
167 @@ -103,7 +104,7 @@
168      #if the second argument 'enumeration' is given, then the Type is an int
169      #else, it's left blank
170      def Type(self, name, enumeration):
171 -        if len(enumeration) > 0 and all([isinstance(x, (int,int)) for x in enumeration]):
172 +        if len(enumeration) > 0 and all([isinstance(x, int) for x in enumeration]):
173              res = IDPIntType(self, name, enumeration)
174          else:
175              res = IDPType(self, name, enumeration)
176 @@ -166,7 +167,7 @@
177          return str(thing)
178
179      def assign_name(self, object_):
180 -        if isinstance(object_, (int,int,str,bool,float)): # Primitive type
181 +        if isinstance(object_, (int,str,bool,float)): # Primitive type
182              return object_
183          name = "o" + str(id(object_))
184          self.object_names[name] = object_
185 @@ -222,7 +223,7 @@
186          self.know(old_class(old.typedName, new))
187
188      def __str__(self):
189 -        return "\n".join(map(lambda bl: bl.show(self.idpobjects.values()), self.blocks)) + "\n"
190            + IDP.gen_models + "\n"
191 +        return "\n".join([bl.show(list(self.idpobjects.values())) for bl in self.blocks]) + "\n"
192            + IDP.gen_models + "\n"
193
194      def fillIn(self, pred):
```

```
195           if self.dirty:
196 @@ -272,13 +273,13 @@
197                   print("GOT OUTPUT:")
198                   print(out)
199                   print("END OF IDP OUTPUT")
200 -         import idp_parse_out
201 +         from . import idp_parse_out
202           if out.strip() == "nil":
203               print("UNSATISFIABLE!")
204               self.cache = {'satisfiable' : []}
205           else:
206  #             self.Predicate("satisfiable", [()])
207 -             self.cache = idp_parse_out.idp_parse(out, map(lambda x: x.name, self.wanted))
208 +             self.cache = idp_parse_out.idp_parse(out, [x.name for x in self.wanted])
209           self.dirty = False
210
211       def checkSat(self):
212 @@ -297,8 +298,8 @@
213           out,err = idp.communicate(input=str(script))
214
215           if __debug__:
216 -             print("err:" + err)
217 -             print("out:" + out)
218 +             print(("err:" + err))
219 +             print(("out:" + out))
220           #check wether the output is true or false
221           if out.find("true") != -1:
222               return True
223 @@ -343,7 +344,7 @@
224  def type(idp):
225      def foo(cls):
226          clsname = cls.__name__
227 -         for name, method in cls.__dict__.iteritems():
228 +         for name, method in cls.__dict__.items():
229               if hasattr(method, "_idp_return_type"):
230                   rt = getattr(method, "_idp_return_type")
231                   if isinstance(rt, str):
```

Bijlage E

Praktische tips voor de beginnende IDP programmeur

Doorheen het masterjaar heb ik veel met het IDP-systeem gewerkt. Hierdoor heb ik bijgeleerd over hoe ik iets implementeer in IDP, en trukjes die het leven van de IDP-programmeur kunnen vergemakkelijken. Ik kan wel niet geranderen dat wat ik hier oplijst correct is. Het zijn tips gebaseerd op mijn ervaringen, en niet op theorieën (vandaar *praktische* tips). De tips in deze lijst mogen niet als absolute waarheid aanschouwd worden.

1. **Floats vermijden.**

   Het IDP-systeem houdt niet van floats, omdat deze kunnen zorgen voor oneindig rekenen. Tussen 0.0 en 1.0 liggen ligt namelijk een oneindige range aan getallen.

2. **Vermijd negatieve getallen.**

   Net zoals floats, heeft het IDP-systeem niet graag negatieve getallen. Vooral in de context van minimalizatie kan dit leiden tot problemen (want nul is plots niet meer het kleinst mogelijke getal).

3. **Verbosity aanzetten.**

   Door gebruik van

```
1 stdoptions.verbosity.solving = 1.
```

   kan de basisverbositeit van het IDP-systeem worden aangezet. Dit is handig om volgende redenen:

   - Wanneer het IDP-systeem lang in de ground-stap zit, zal het ook lang moeten solven.
   - In het geval van optimalizatie geeft het IDP-systeem telkens een update wanneer een nieuw model gevonden is, met daarbij zijn waarde en het tijdstip waarop deze gevonden werd, in milliseconden. Deze output kan gemakkelijk omgezet worden in een grafiek, om zo weer te kunnen hoe de gevonden waarde doorheen de tijd evolueert.

- Op deze manier krijgen we als programmeur meer feedback over de werking van het IDP-systeem. Als het vastzit in een oneindige ground-stap, kunnen we zonder verbositeit enkel gissen dat het hier vastzit.

4. **Abs() kan in sommige situaties voor vertraging zorgen.**

   Hoewel het soms een nodige functie is, kan de invloed van abs() wel merkbaar zijn. Best hier eens mee experimenteren hoe merkbaar het verschil is.

5. **Schrijf zoveel mogelijk als definitie.**

   De definitie werkt veruit het vlotst en is het meest overzichtelijk, ten koste van enkele gevolgen (zie theorie). Als iets herschreven kan worden als definitie zonder dat het last heeft van de gevolgen, doe dit dan.

6. **Genereren van unsatcores en explainunsat kan helpen met debuggen.**

   Wanneer het systeem unsat is, en je echt geen idee meer hebt waarom, kan je unsatcores genereren. Hiervoor is er:

```
1 printunsatcore(theory, structure)
```

   Deze geeft een uitleg van welke elementen elkaar nog tegenwerken en zo een satisfiable model voorkomen. Hoewel deze uitleg vrij cryptisch is, kan het toch helpen om de fout in de code te vinden.

   Om een minder cryptische uitleg te krijgen, kan `explainunsat` gebruikt worden. Deze werkt als volgt:

```
1 explainunsat(theory, structure, vocabulary)
```

7. **String tegenover integers.**

   Afhankelijk van het probleem kan het sneller werken om types te definieren als string, of als int (of nat). Vier studenten kunnen we bijvoorbeeld definiëren op twee manieren:

```
1 Theory T{
2     Type studint isa nat //definieer student als een nat
3     Type studstring isa string //definieer student als een string
4 }
5 Structure S:T{
6     studint = {0..4} //vier studenten
7     studstring = {s0, s1, s2, s3} //vier studenten
8 }
```

   Soms kan een systeem sneller werken met ints, soms met strings, afhankelijk van hoe ze gebruikt worden. Belangrijk om hierbij te doen is dus om te **experimenteren**.

8. **Tussenstappen gebruiken voor termen.**

In sommige scenario's is het gewenst om een term te minimalizeren die afhankelijk is van veel verschillende variabelen. Hier kan het handig zijn om "tussenvariabelen" te gebruiken, in plaats van 1 grote berekening te doen voor de term. Zo is

```
1 Afstand = sum{x,y: Samen(x,y): Woont(x) - Woont(y)}.
2 Samen = #{x,y: Samen(x,y) & Samen(y,x}.
3 Uit = #{x,y: Samen(x,y) & Zone(x) ~= Zone(y)}.
4
5 Totaal = Afstand + 10*Samen + 3*Uit
```

bijvoorbeeld veel leesbaarder dan

```
1 Totaal = sum{x,y: Samen(x,y): Woont(x) - Woont(y)} + 10*#{x,y:
2     Samen(x,y) & Samen(y,x)} + 3*#{x,yL Samen(x,y) & Zone(x)
3     ~= Zone(y)}.
```

Doordat het makkelijker leesbaar is, is het ook makkelijker te debuggen. Wanneer IDP een model gevonden heeft, zijn de drie waarden ook afzonderlijk te lezen. Dit laat toe tot het beter inschatten van wat er mis kan lopen. Het is wel mogelijk dat de tweede oplossing efficiënter zal werken, maar om te testen in het IDP-systeem is de eerste oplossing makkelijker.

9. **Bij gebruik van pipe komt de verbosity output in stderr.**

   Wanneer we de IDP output in de terminal willen wegschrijven naar een file, komt de "normale" output in `stdout`, en de verbosity output in `stderr`. Om beide outputs in dezelfde tekstfile te schrijven, kunnen we een volgend commando gebruiken in een Bash terminal:

```
1 ./idp idp_bestand.idp > output.txt 2>&1
```

10. **Meerdere vocabularies.** Een groot IDP bestand kan al snel onleesbaar worden. Om de leesbaarheid te behouden is het mogelijk om meerdere `vocabularies` te maken. Dit laat toe om bijvoorbeeld 2 `vocabularies` te maken: eentje voor alle inputdata, en eentje voor al de te genereren data. Deze `vocabularies` moeten samengevoegd worden door gebruik van het keyword `extern`. Bijvoorbeeld:

```
1 vocabulary V_optimization{
2         extern vocabulary V_data
3 }
```

11. **Handleiding lezen.**

    In de handleiding staan talloze verborgen features van het IDP systeem, zoals de verbosity en de explainunsat. Ik kan iedereen aanraden om deze te lezen.

12. **EXPERIMENTEREN!**

    Vrijwel het belangrijkste bij het schrijven van IDP code is experimenteren. Door te experimenteren met hoe de IDP code geschreven is kan je leren wat nu net efficiënt is, en wat niet.

FACULTEIT INDUSTRIËLE INGENIEURSWETENSCHAPPEN
CAMPUS DE NAYER SINT-KATELIJNE-WAVER
J. De Nayerlaan 5
2860 SINT-KATELIJNE-WAVER, België
tel. + 32 15 31 69 44
iiw.denayer@kuleuven.be
www.iiw.kuleuven.be