Course Recommendation Agent
Description and Evaluation

by

Filippo M. Libardi
Mikhail Ternyuk
Olusanmi A. Hundogan
Samuel Galanakis
Sorin Dragan

Graduate School of Natural Sciences
MSc Artificial Intelligence

University of Utrecht

Utrecht, Netherlands

2019

Abstract

This report aims to give a general understanding as well as a deeper analysis of a Goal-Based agent system that can be used to select appropriate University courses for a certain student.

The knowledge base of the agent consists of a large ontology with several entities and relations between entities. The agent queries the ontology (using logical expressions) to retrieve any needed set of entities.

A student can express preferences and/or restrictions to narrow down the possible course list. The agent, in turn, applies a metric to evaluate the best courses for the student. This metric is based on several factors which are addressed in detail in the report. The agent has a set of finite possible precepts sequences, these are consequently mapped to possible actions the agent can perform.

Additionally the agent's performance is evaluated using a performance metric. A behavioural analysis as well as a complete system evaluation will be carried out comprehensively in the last section of this report.

Contents

1. Introduction

The end goal is to create a system that is able to recommend the best possible combination of courses for a specific student. A goal based agent approach was chosen for this application, as detailed in chapter 3. The recommendation of courses is based on both the preferences of the user and certain strict conditions. Both of which will either be in the knowledge base already or will be entered by the user. The information the agent relies on is stored in an ontology. The ontology contains all the concepts and relations needed for the reasoner to infer the information required by the Agent for identifying and ranking courses for a student. The core concepts of the ontology are Students, Teachers, Courses, Topics and Weekdays/Periods for scheduling purposes. All included concepts, their corresponding properties and how they relate to the Agents operations are detailed in chapter 2.

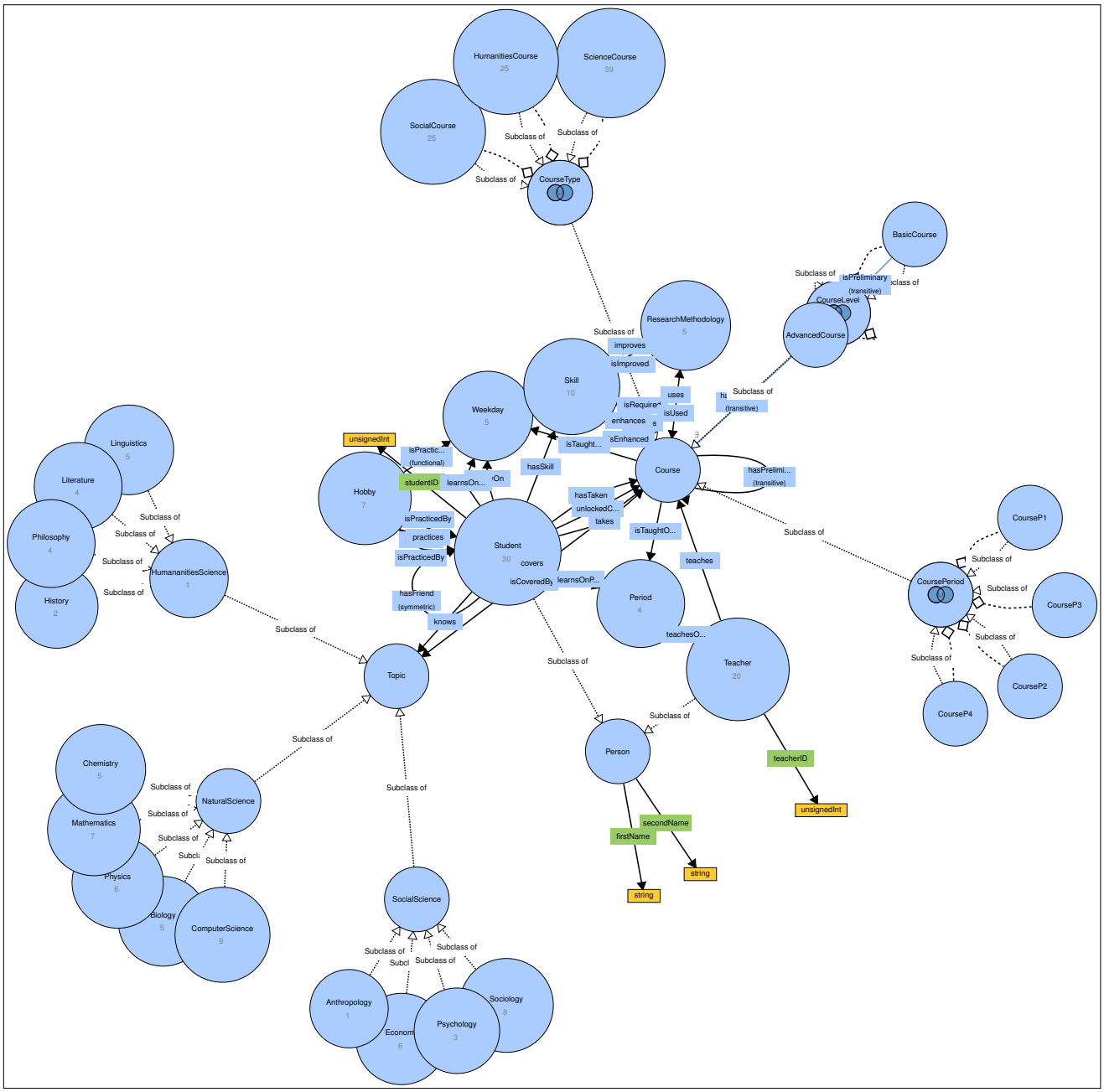
The implementation of the agent was achieved using a finite state machine approach with a text based interface. Details on the various states and the corresponding transitions are found in section 3.2.

The actual recommendation of courses can be summarized as follows. The student starts by providing the agent with filters that are considered non-negotiable which rule out any courses that do not comply with them. For example, the period which they are interested in, filters out all courses not taught in that period. Subsequently, the user is asked to rank a list of preferences based on how much he values them.
After having ranked the preferences the rest of the process is iterative in essence. The agent will ask the user for his preference, starts from the most highly valued preference. Consecutively it will rank and offer the most highly ranked suggestion based on the

preferences gathered so far. If the user does not accept the recommendation the agent asks for the next preference and repeats the process until the user is satisfied with a recommendation, exits or there are no more preferences to specify.

The above process, how ranking is achieved as well as further features such as calendar integration and agent trust are expanded on in chapter 4.

2. Ontology Design



The ontology for this application needs to describe all concepts and relations relevant to courses, students and teachers that play a part in suggesting courses to students. First the class structure and meaning is addressed.

2.1 Concepts

The top level parent classes for the ontology are Course, Topic, Person, Hobby, Period, Research Methodology, Skill and Weekday.

The first class, Course and its children are shown by Figure 2.1. As is depicted, CourseType, CourseLevel and CoursePeriod are the direct children of Course. They all represent different ways of categorizing courses. Specifically, CourseType has disjoint subclasses which each contain all courses of a specific type, for example Science course. CourseLevel on the other hand has subclasses Advanced and Basic course which split the courses, as the names suggest, with respect to how advanced they are. What is meant by advanced and basic in this context will be made clear in section 2.2. Lastly, CoursePeriod simply categorizes the courses depending on which period they are taught in. It is assumed that the agent will be used for a university system with four blocks per academic year and that each course is taught only once per year.

The next class is Topic, depicted by Figure 2.2. The topics class is used to split the topics depending on what field of study they belong to. The direct children of Topic are used to group topics belonging to branches such as the Natural Sciences etc. The next level of sub classes corresponds to sets of topics belonging to specific fields of study, such as Mathematics or History. These topics are used to assess the similarity of courses and can also be used as a preference by the user. This is explained in detail in chapter 4.

The next class is Person which simply contains the subclasses Teacher and Student which split the people in the self evident way.

The two subclasses are disjoint as it is assumed that no person can be both a teacher and a student simultaneously.

The rest of the classes have a simpler structure. The Hobby class is used to represent all the hobbies students may take part in. For example instances may be football and ice

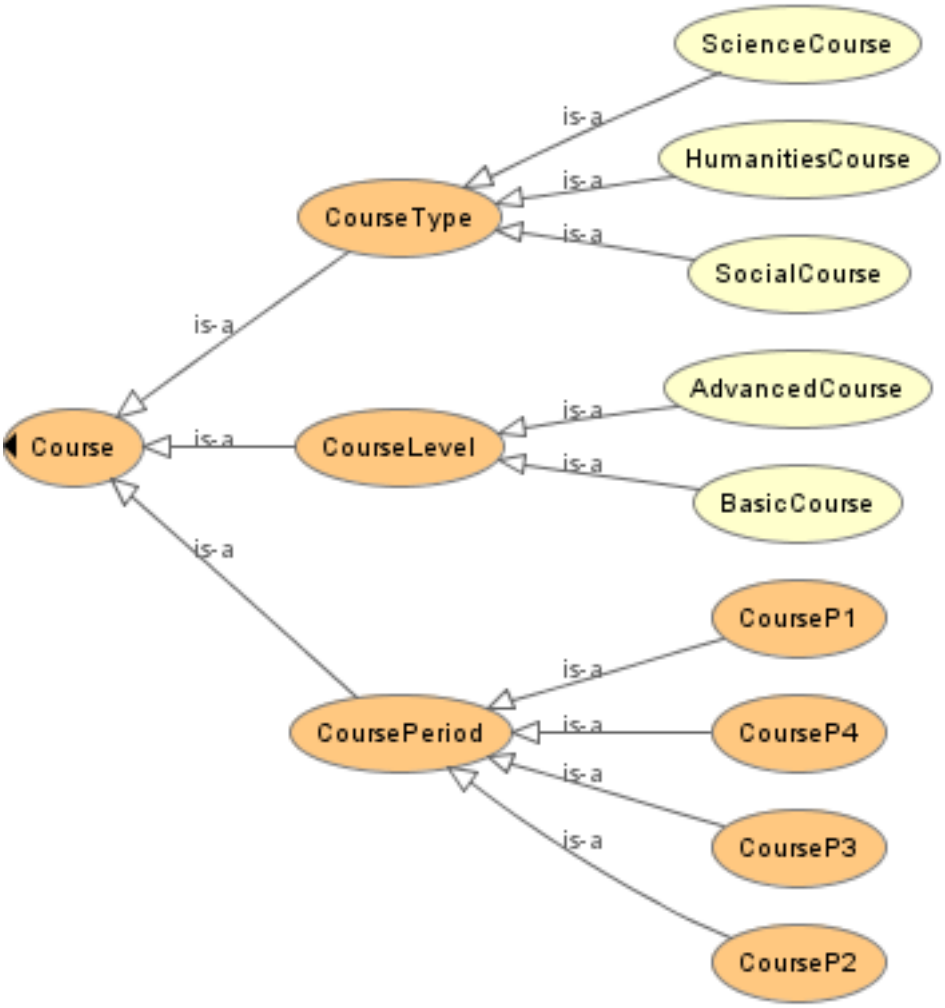


Figure 2.1: Course class and children

Figure 2.2: Topics class and children

skating. These are relevant to course recommendation as they may clash with the schedule of certain courses which is clearly not favorable. Equation 2.1 shows the DL.

$$Hobby \sqsubseteq \exists\, isPracticedBy.Student \tag{2.1}$$

Similarly to the CoursePeriod subclass of Course, the class Period is used for scheduling purposes. It has four instances, each representing an academic period. These are used in order to specify facts such as during which periods Teachers teach, students take classes using relations which will be analyzed in section 2.2. Also this hierarchy has been chosen due to the necessity of representing the impossibility of a Teacher to teach more than one course per period.

The Research Methodology class is a set of instances that represent different research methodologies. Examples are Lab and Workshop. These are connected to Courses and Skills via relations.

The Skill class contains instances of skills such as Academic Writing and Debating. These are directly connected to the research methodologies and in this model the students gain skills when they take courses that use said methodologies.

Lastly, Weekday simply has an instance for each day of the week and is used for scheduling purposes such as acknowledging clashes between courses and hobbies.

2.2 Properties

In order to express the relations between the instances of the classes introduced in the previous section a variety of object properties are used. In table 2.2 all the object properties are listed along with the corresponding range, domain, inverse properties and a summary. The function and purpose of certain properties is less apparent than others, especially those defined as chains. These are the focus of the remainder of this section.

Name	Domain	Range	Inverse	Use summary
CanTake	Student	Course	None	hasTaken ∘ isPreliminary
covers	Course	Topic	isCoveredBy	A course covers a topic
enhances	Course	Skill	isEnhanced	uses ∘ improves
hasFriend	Student	Student	Symmetric	Friendship between students
hasPreliminary	Course	Course	isPreliminary	Course has preliminary
hasTaken	Student	Course	None	hasTaken ∘ hasPreliminary
improves	ResearchMethodology	Skill	isImproved	R. Methodology improves skill
isBusyOn	Student	Weekday	None	practices ∘ PracticedOnWeekday
isTaughtOnPeriod	Course	Period	None	Course is taught during a specific period
isTaughtOnWeekday	Course	Weekday	None	Course is scheduled on specific weekday
isPracticedOnWeekday	Hobby	Weekday	None	Hobby scheduled on day
learnsOnWeekday	Student	Weekday	None	takes ∘ isTaughtOnWeekday
learnsOnPeriod	Student	Period	None	takes ∘ isTaughtOnPeriod
teachesOnPeriod	Teacher	Period	None	teaches ∘ isTaughtOnPeriod
practices	Student	Hobby	practices	Student has hobby
uses	Course	ResearchMethodology	isUsed	Course uses Methodology
takes	Student	Course	None	Student takes course
teaches	Teacher	Course	None	Teacher teaches a course

Firstly, CanTake is expressed as a chain hasTaken ∘ isPreliminary and thus is triggered when a student has already taken a course which is preliminary to another course. The reasoner will check for each student if a course they have taken is preliminary to a course and in such a case infer that the student can take that course.

Secondly, enhances is expressed as a chain uses ∘ improves and thus is triggered when a course used a research methodology which in turn improves a skill. The reasoner will do this check for each course and methodology connected to that course and infer that the course improves the corresponding skills. These inferred connections from courses to skills are then used in the agent in order to rank courses higher if they have common skills with those preferred by the user.

Furthermore, hasTaken is used to denote that a user has taken a course. In addition to this role it is defined as a chain hasTaken ∘ hasPreliminary in order to be able to infer that if a student has taken a course they have also take any preliminaries to that course as otherwise they should not have been able to take that course.

The properties isBusyOn and learnsOn are used to specify that a student has a hobby or class on a weekday. The reasoner will check if a student has a course that is scheduled on a specific weekday and if so it will infer learnsOn that weekday for that student. Similarly for isBusyOn. In the agent if both isBusyOn and learnsOn hold for the same day the user is alerted to the schedule clash and informed that he can not take that course.

The property teachesOnPeriod is defined as a chain teaches ∘ isTaughtOnPeriod and is used to infer on what period(s) a teacher teaches based on the courses they teach and when they are scheduled.

The properties learnsOnPeriod, learnsOnWeekday are used to express that a student takes some course in a period or that is scheduled on a specific weekday respectively.

2.3 Class definitions and restrictions

Now that the classes and properties are established, it is possible to clarify important restrictions on the classes.

Equation 2.2 shows the definition for a course in description logic (DL). A course is

every instance that covers at least one topic, uses exactly one research methodology, is thought on exactly one period and two week days.

$$\begin{aligned} Course \equiv \exists\, & covers.Topic \\ \sqcap\ = & \ 1\ isTaughtOnPeriod.Period \\ \sqcap\ = & \ 2\ isTaughtOnWeekday.Weekday \\ \sqcap\ = & \ 1\ uses.ResearchMethodology \end{aligned} \tag{2.2}$$

The course types AdvancedCourse and BasicCourse are defined by things that have a preliminary course or are preliminary to another course. The Equation 2.3 shows the exact definition. In the case of BasicCourse it is possible to narrow down the definition by restricting the statement to AdvancedCourse.

$$\begin{aligned} AdvancedCourse &\equiv \exists\, hasPreliminary.Course \\ BasicCourse &\equiv \exists\, isPreliminary.AdvancedCourse \end{aligned} \tag{2.3}$$

The Person class is a union of Teacher and Student. (Equation 2.4)

$$Person \equiv Teacher \sqcup Student \tag{2.4}$$

The teacher (Equation 2.5) is defined by anything which teaches at most one course per period. Likewise, the student is defined by anything that takes at least two and at most three courses per period (Equation 2.6).

$$\begin{aligned} Teacher \sqsubseteq\ Person \sqcap (&\leq\ 1\ teaches.CourseP1 \\ \sqcup\ &\leq\ 1\ teaches.CourseP2 \\ \sqcup\ &\leq\ 1\ teaches.CourseP3 \\ \sqcup\ &\leq\ 1\ teaches.CourseP4) \end{aligned} \tag{2.5}$$

$$\begin{aligned} Student \sqsubseteq Person \sqcap (& (\leq 2\, takes.CourseP1 \sqcap \geq 3\, takes.CourseP1) \\ & \sqcup (\leq 2\, takes.CourseP2 \sqcap \geq 3\, takes.CourseP2) \\ & \sqcup (\leq 2\, takes.CourseP3 \sqcap \geq 3\, takes.CourseP3) \\ & \sqcup (\leq 2\, takes.CourseP4 \sqcap \geq 3\, takes.CourseP4)) \end{aligned} \tag{2.6}$$

The classes Period, ResearchMethodology and Weekday are sets of a discrete number of individuals. Equation 2.7 lists all individuals per class.

$$\begin{aligned} Period &\equiv \{P1,\, P2,\, P3,\, P4\} \\ ResearchMethodology &\equiv \{Experiment,\, Lab,\, Lecture,\, Seminar,\, Workshop\} \\ Weekday &\equiv \{Mo,\, Tu,\, We,\, Th,\, Fr\} \end{aligned} \tag{2.7}$$

3. Agent Design

In this section the design of the agent architecture will be assessed. Additionally, it will be explained how that said architecture has been implemented. This means showing how the code has been structured in order to consistently project each component from agent framework to the project design.

3.1 Agent Architecture

The agent design is derived from the goal based agent architecture. The final goal of the agent is to collect the best possible packages of courses according to the student preferences list. This, in the system is modelled as returning packages with the highest score. In order to achieve this in the shortest amount of interactions, the agent will start with asking the most important properties first. So the user set preferences act as a primitive search and planning function. Specifically, the agent takes in the user ranked preference and schedules

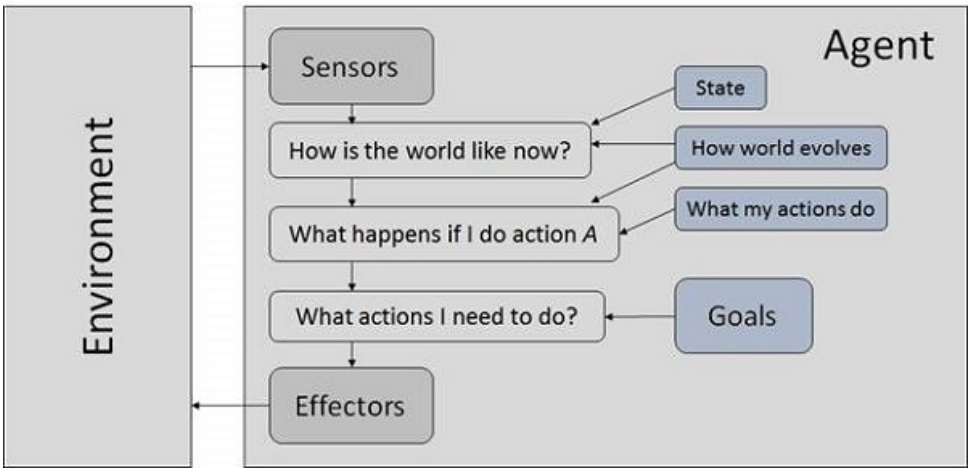


Figure 3.1: Goal Based architecture.

the preference requests in the corresponding order.

Other agent types were considered before selecting the goal based architecture. First, reflex based architecture was considered. This option was dismissed because the the action at any specific state depends heavily on previous input. Secondly, the agent architecture strongly resembles the utility based architecture. However, that would have required calculating the utility for each possible state in the state space. If you consider the full number of course combinations as states, the computation would become intractable for a large number of courses. This is the reason for dismissing the utility based architecture. Additionally BDI architecture was dismissed because the information about the environment that is needed to perform the desired role is reliable. Looking at Figure 3.1 the components of the architecture are mapped as follows. The agent sits in a partially observable and static environment which consists of all the entities within the ontology (covered in the previous chapter).

The system knowledge base ought to represent every possible condition the world can be in. However, the ontology might be missing information about the state of the world, hence it cannot be considered fully-observable (uncertain factors cannot be foreseen).

Additionally, the agent will be the only entity updating its knowledge about the world, so the latter can be considered static as there is no external factor that affects it (world state transition will only happen due to internal operations). For instance: every time the user gives a new preference or applies a new constraint (world evolves), the agent behaves consequently and re-executes its "suggesting pipeline" based on the newly given information.

The task environment consists of the user's need to select appropriate courses for a specific period.

The preferences of the user are collected in run time through a text interface. The agent continues asking for further preferences until the user accepts the proposed courses.

All possible states are considered in development time (Finite State Machine), because the agent sits in a partially observable environment.

	Possible Actions
A1	*DisplayAndPropose(AvailablePackages)*
A2	*AskHardPreference(HardConstraint)*
A3	*AskPreference(SoftConstraint)*

	Agent Functions
F1	*S1 >A1*
F2	*S2 >A1*
F3	S2 >A3
F4	*S3 >A1*
F5	*S4 >A2*
F6	*S5 >A3*
F7	S5 >A1
F8	*S7 >A2*
F9	S7 >A3

$$
\begin{aligned}
HardConstraint &= \{Hobby, Period\} \\
SoftContraint &= \{Topics \sqcup Weekdays \sqcup Lecturers \sqcup Skills \sqcup Friends\} \\
PreviouslyTakenCourses &= \{TakenCourse_1, TakenCourse_2 ... TakenCourse_n\} \\
AvailablePackages &= \{AvailablePackages_1, AvailablePackages_2 ... AvailablePackages_n\}
\end{aligned} \tag{3.1}
$$

$$
\begin{aligned}
S_1 &= < \{H_n, P_n\}, \{SC_1 \ldots \sqcup\ SC_n\}, \{PTC_1 \ldots \sqcup\ PTC_n\}, \{AP_1 \ldots \sqcup\ AP_n\} > \\
S_2 &= < \{H_n, P_n\}, \{\}, \{PTC_1 \ldots \sqcup\ PTC_n\}, \{AP_1 \ldots \sqcup\ AP_n\} > \\
S_3 &= < \{H_n, P_n\}, \{SC_1 \ldots \sqcup\ SC_n\}, \{PTC_1 \ldots \sqcup\ PTC_n\}, \{\} > \\
S_4 &= < \{\}, \{\}, \{\}, \{\} >
\end{aligned} \tag{3.2}
$$

As Equation 3.1 shows, there are several sets which constitute our set of states. A state is composed of a combination of: one hobby and one period, a set of soft constraints, a set of of previously taken courses and a set of available packages. The hard constraints define conditions which need to be met to fulfill the goal, whereas the soft constraints and the previously taken courses are used to score the possible available packages. The available courses is a set of course combinations, which the agent returns.

Equation 3.2 lists six possible states and five actions. If the agent's state is empty (State 4) it will ask for the hard constraints and retrieve the previously taken courses (Action 2). With this information the agent will be able to generate a list of possible

course combinations which corresponds to State 2. In this state, the agent will display the available courses and propose (Action 1) the highest scored course package (Action 1). If the user decides to take the proposed package the agent fulfills his goal. Otherwise, it will ask for soft constrains, which leads to State 1. At this state it will repeat Action 3 and Action 1 until it succeeds in its goal or drops it because there are no courses available (State 4).

3.2 Project Architecture

From the above explanation of how the agent architecture has been designed, it falls the design utilised for the agent implementation. Once studied the deterministic environment and all the states that the agent can possibly be in, the *Finite State Machine* implementation became clear. Beside the state transitioning methodology, the project architecture implements one class representing the agent and one representing the user. The overall high level hierarchy of the classes is explained in the following paragraph.

3.2.1 Class Hierarchy

One of the biggest problems when working with Procedural Programming is “global variables”. The latter’s values can be changed from any function in the code and it becomes very hard to follow their behavior. However, different solutions can be found to avoid global variables.
One of these being variables parsing trough function parameters. While this can, in turn, be computationally stressing, especially when working with heavy data structures, it decreases code readability and increases the creation of temporary variables. Generally speaking, for smaller projects, this might be feasible, however, since this is not the case, objects in our project enforce a “have-a” relationship to these variables.

This is, among many other reasons, why *Object Oriented Programming (OOP)* is a good method to employ when writing code. Besides, it also enhances readability and makes it easier to understand the program structure. The *State Machine* class includes

different class methods, this process is also known as information hiding or *Encapsulation*. Class members can only be accessed from within the class and its child classes. Child classes of a given class share common properties with their parent class through a process called inheritance. Is important to keep in mind that the Finite State Machine handles internal-state transitioning. Said states need to be differentiated from the world state belonging to the agent architecture.

The constructor of the base class, *StateMachine*, is given in Listing 3.1. As illustrated in the code snippet, the class makes use of the *Borg Design Pattern*.

```
class StateMachine(object):
    __shared_state = {}

    def _init_(self):
        self.previousState = Noneconstructor
        self.currentState = None
        self.nextState = None
        self.levenshtein_words = []
        [...]
        self._dict_ = self.__shared_state
```

Listing 3.1: Constructor of the base class

The Borg implementation has a private attribute “shared state” which is stored in the class dictionary when initialization comes. This allows the different class instances to share the class attributes amongst each other. Thus, whenever a new state is created it will still contain the values for its class attributes, inheriting them from the base class.

To understand the functionality of the main loop, it’s necessary to keep in mind the first three variables declared in the constructor: *previousState, currentState and nextState*. The main loop is contained in the *update* function of the *StateMachine* base class. It runs until there is an active current state and it handles state transition as detailed in Listing 3.2.

```
while self.currentState:
    [...]
    self.currentState.update()
```

```
4       if self.nextState:
5           self.previousState = self.currentState
6           self.currentState = self.nextState
7           self.nextState = None
8
```

Listing 3.2: Main loop of the state machine

Due to their relations, the update method is inherited by any subclass of State Machine, meaning that whatever a state is current, its update will be called. This technique in OOP is called *Method Overriding* and it is a form of *Polymorphism*. All the classes share a method, in some languages this is defined as virtual, however, depending on which type of class is being instantiated that particular class will call its own overridden method. This is very useful in our case as it makes it possible to not specify from which class the update method needs to be called. The only necessary bit of information is a valid current state. Moreover, keeping track of state transitions is made very easy by the last lines included in the above snippet. Every time a state transition is necessary, all that is required is to assign a value to the next state. If the next state has been assigned a value, the previous state will become the current state and the current state is set to the next state. This is a very neat technique to handle state swapping.

Is possible to think of the *StateMachine* as merely the state transitioning brain of the Agent.

As shown in Figure 3.2 the State Machine class is where the main loop of the agent and each state transition is constructed. Additionally the class handles the relations between the user and the agent and acts as a data parses between the two. While the agent class takes care of the ranking and dealing with the ontology, the agent methods will be called from within the *StateMachine* class. This is because is the latter class that actually decides (based on the agent state in the world) what action to take next. This class is still part of the agent architecture and is complementary to the Agent class. The latter, in turn, takes care of the ranking system for packages and handling the flow of data from/to the ontology.

The student class, on the other hand, takes data from a student instance (matched trough ID) in the ontology and represent the user. Every time it is asked to take a decision it will

randomly choose a positive or negative answer. Important properties the class inherits from the ontology instance are previously taken courses and friends. Every other preference or hard filter (e.g. hobby) is expressed in a data file (student.json). The system run for every instance of the student in the file (total of 30) and returns the best package for each.

At every turn the user gives a new preference the agent queries the ontology for the list of courses with the expressed restriction/preference applied. After, it will re-calculate the score for each package possible (this procedure is described in depth in chapter 4) and propose to the user. This happens at every turn and the user can decide to exit at any point in time. It is assumed that if the user decides to exit it is because the courses proposed suits his need. When the user decides to exit or when there is no more preference left to be expressed, the system will update the user instance inside the ontology and sync the reasoner. When doing this, the agent also updates the user hobbies (as these are expressed in the json file). If the ontology returns an inconsistency, it means the user cannot take the selected course due to an overlap with his hobbies. At this stage the system will alert the user and warn him about the inconsistency, the user is then asked to pick another package. Once this happens and the new package has no overlap with user hobbies, it will be added to the user Google Calendar (a better explanation of the latter system is found in subsection 4.3.3).

In Figure 3.2 a more complete and clearer view of the above whole process is represented. As noticeable there is also another class playing a part in the system pipeline. This consists of fake agents that are needed for the trust system, this is explained in details in subsection 4.3.1.

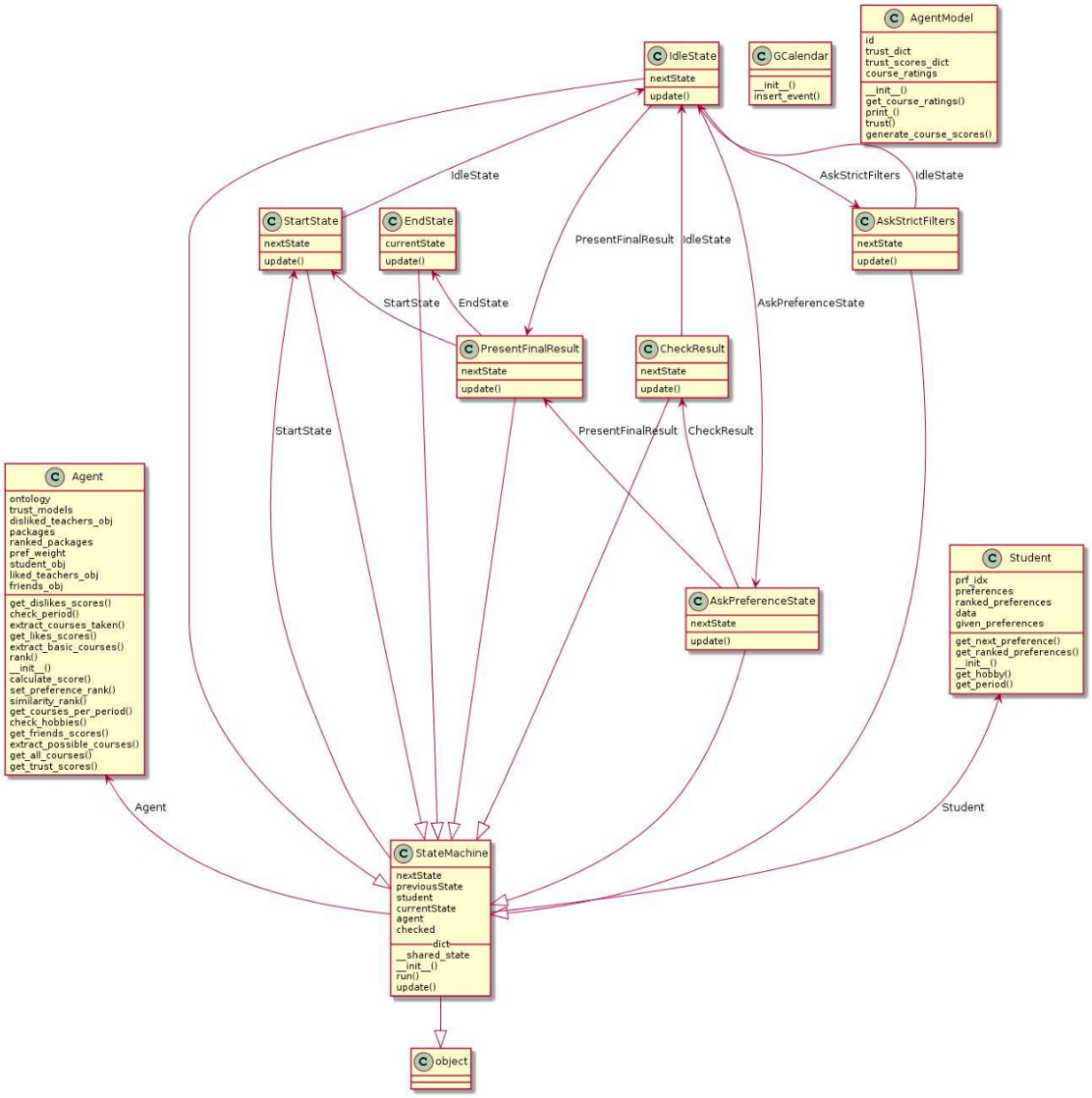


Figure 3.2: Class Diagram

4. Agent Implementation

4.1 Similarity

The student is given the option of receiving courses similar to one of his choice. Courses are classified as similar depending on the number of common topics and research methodologies. Specifically, it is calculated using function 4.1.

$$\frac{Common_T}{Common_T + \neg Common_T + 1} + R.Meth \tag{4.1}$$

Where:

$$R.Meth = \begin{cases} 0 & \text{Courses use different methodologies} \\ \frac{1}{Common_T + \neg Common_T + 1} & \text{Courses use same methodologies} \end{cases} \tag{4.2}$$

4.2 Ranking

After filtering out courses that do not comply with the hard constraint initially set by the user the agent will generate course packages consisting of two to three courses and rank them according to the preferences expressed by the user. This will happen every time a new preference is expressed by the user. The score of course packages is simply the summed score of the courses it contains. Each individual course is scored according to several factors, weighted by the way the user has ordered the preferences. These factors are preferred/disliked topics, skills that the user wants to improve, days to avoid, preferred days, liked and disliked teachers, courses being taken by friends and ratings from peers.

The topics score for preferred is calculated according to function 4.3 and is denoted by $Score_T^+$. Likewise the score for the topics that the user would like to avoid are calculated by function 4.4.

$$Score_T^+ = \frac{Count(Preferred_T \bigcap Course_T)}{Count(Preferred_T)} \tag{4.3}$$

$$Score_T^- = -\frac{Count(Avoid_T \bigcap Course_T)}{Count(Avoid_T)} \tag{4.4}$$

The skills score is calculated according to function 4.5 and is denoted by $Score_S$.

$$Score_S = \frac{Count(Preferred_S \bigcap Improves_S)}{Count(Preferred_S)} \tag{4.5}$$

Concerning whether a student likes or dislikes a teacher the score of the course taught by the teacher is increased, respectively decreased, according to the weight specified by the user in the ranking of preferences.

When the course in question is being taken by a friend of the student it's score is increased in accordance with the assigned weight.

The Weekday preference of the user is taken into account by counting the number of days in common with those the course is scheduled on. The score is then discounted for every common day that the user wants to avoid and increased if the day was preferred.

The ratings of peers are also taken into account. The influence of this factor depends on the trust of the agent with the peers in question and is detailed in section 4.3.

The final score for a given course is calculated by function 4.6 where w_n is the weight determined by the ranking given by the user to the n_{th} preference and $nPref$ is the total number of preferences.

$$CourseScore = \sum_{n=1}^{nPref} Score_n w_n \tag{4.6}$$

The aforementioned scores are assigned to each course and subsequently packages are made and ranked by summing the scores of the included packages. The first package is

presented to the user.

4.3 Trust

In a multi-agent system, a self-interested agent represents the preferences of every user by constructing a model of the world based on personal experience. However, it may also rely on the knowledge of other agents. To validate the knowledge of other agents, a trust function is needed. In the case of this report, this trust function is derived from the model of other agents. The model consists of courses previously taken by the student and the corresponding ratings on a scale from 0 to 10 (for example "A student that took Intelligent Agents and rated it with 9"). The trust function assigns a trust value for each neighbouring agent by taking into account the number of common courses as well as the similarity between the ratings given to those courses. The next sections will explain the details of the trust function and the course scoring system.

4.3.1 Trust Function

The update of the trust function requires the agent to request the internal models of the neighbouring agents. The trust value is stored in a dictionary where the key is the id of the neighbouring agent. For each model, a scaling factor is defined as the ratio between the number of common courses and the total number of courses in the model. The denominator discounts the trust value for neighbouring agents who have a large count of courses in their model.

$$scaling\_factor = \frac{count_{x,y}(common)}{count_{y}(all\_courses)} \tag{4.7}$$

x : agent of the student in discussion, y : neighbouring agent

If there are no courses present in the model of the student's agent and the neighbouring agent in the current iteration, the trust value will be 0. If the are common courses the trust value is calculated using formula 4.8:

$$trust\_dict[model.id] = (10 - mean(diff\_list)) \cdot 0.1 \cdot scale \tag{4.8}$$

The maximum mean of the ratings is 10. $diff\_list$ is a list of absolute differences between the rating given in each model for the common courses. The mean of the formerly mentioned list gets subtracted from the maximum possible mean achieved in the perfect scenario. The result is divided by 10 to bring it between 0 and 1 and is further multiplied by the scaling factor.

4.3.2 Course Score

After the trust value for each neighbouring agent is computed, the student's agent generates a new dictionary that holds each neighbouring agent's courses ratings. These weighted by the trust value of that specific agent. The final score for each course is computed using formula 4.9:

$$\begin{aligned} model\_ratings[course] = & rating\_score(courserating) \cdot trust\_dict[model.id] \cdot \\ & score\_discount(count(neighbouring\_agents)) \end{aligned} \tag{4.9}$$

The <rating_score> function receives as a parameter the course rating from the model. If the rating is lower than 5, the course score will be decreased by $(5 - score)/5$. If the rating is greater than 5, the course score will be increased by $(score - 5)/5$. If the score is 5, noting is added or subtracted from the course score. This suggests that if the student's agent desires a course that was rated poorly in a neighbouring agent's model whom it trusts, the agent should lower the probability of having that course in a recommended package. This will be done by lowering his score and vice-versa.

When the student uses the agent to choose a further course, the agent relies on the trust

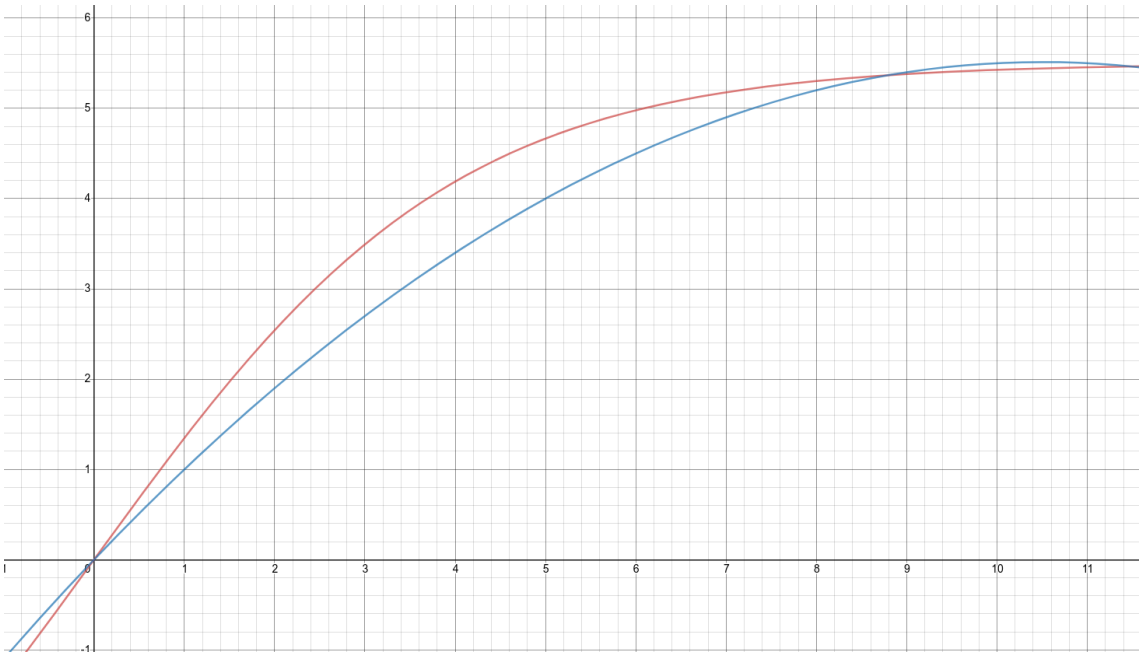


Figure 4.1: Plots of $f_1(x)$ in blue and $f_2(x)$ in red

system to increase or decrease the score of courses present in the recommendation package. A course score will be adjusted using the average of the trust score the same course from each neighbouring model which contains it. Because the importance of the average score for a course increases with the number of neighbouring agents, a new discount factor is required. The *score_discount* functions as a way of valuing the expertise of a large group of neighbouring agents more than the expertise of a smaller group. The upper boundary of the function was defined as 5.5 and it would be reached with 10 neighbouring agents. The function should moderately increase from 0, for no neighbouring agents, to the upper boundary (5.5) for a high number of neighbouring agents. Two functions that have the required characteristics were proposed.

The first one is part of a parabola defined to be constant after 10.

$$f_1(x) = \begin{cases} 5.5 & x < 10 \\ 5.5 - \frac{(10-x)\cdot(10-x+1)}{20} & 0 < x \leq 10 \end{cases} \tag{4.10}$$

The second one is a variation of the sigmoid function.

$$f_2(x) = \frac{11}{1+e^{\frac{-x}{2}}} - 5.5 \tag{4.11}$$

4.3.3 Calendar Integration

If the user indicates that he is satisfied with a suggested course package the agent will automatically add the included courses to the user's calendar. This is done via the google calendar API. For each course in the selected package the provided event template is filled with the corresponding title, a list of attendees and added to the student's calendar on the two days it is taught.

5. Results

A performance metric of the agent objectively assesses an agent's ability to act on a given task. This section will explain the internal performance metric the project uses, alternatives and external metrics.

5.1 The Performance Function

The internal performance metric of this project relies on one assumption. The hypothesis that the more preferences the agent meets, the higher the satisfaction of the user. Equation 5.1 shows a simple formalisation of the objective function.

$$\frac{1}{N}\sum_{i=1}^{N} unitary\_pref_i \qquad unitary\_pref_i \in [-1,1] \tag{5.1}$$

The formula sums the binary values of the unitary preferences specified by the user. It is necessary to normalise by the number of stated preferences to compare different scores.

5.2 Experimental Results

Figure 5.1 shows the performances of different agent configurations. The first boxplot shows an agent randomly sampling two or three courses from all possible ones in the ontology. Thereby disregards any hard constraints and possibly returning invalid course combinations (A1). The second shows an agent that always satisfies the hard constraints mentioned in the chapter 3, but picks the course combination randomly (A2). The third

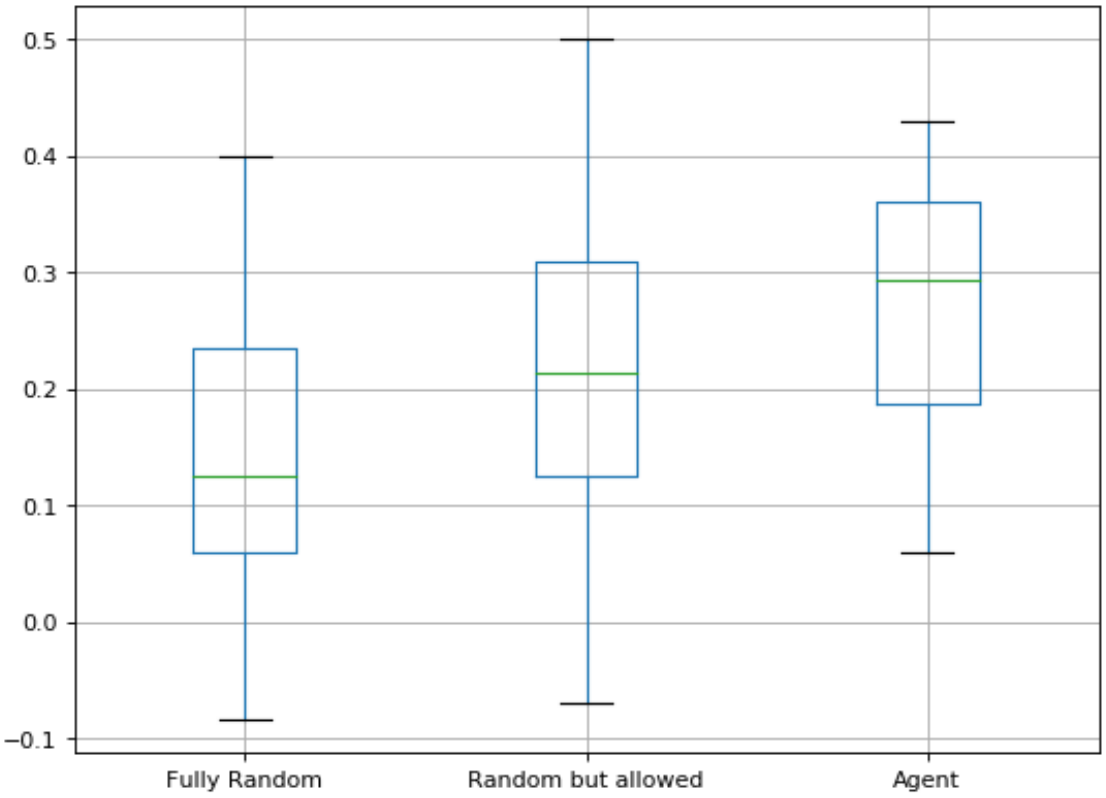


Figure 5.1: A boxplot showing the score values of three agent configurations over 45 trials.

boxplot describes the agent designed in this project (A3). As expected, the average score increases with the sophistication of the selection process. Thus, just making sure that all course packages are valid is better than randomly guessing a package; and deciding on a package based on a scoring metric is generally better than randomly picking a course package. However, the agent A2 does score higher from time to time, evidenced by the variance of the second boxplot. Agent A3's average score is about 30% accuracy, which is significantly different ($t(44)=3.28$, $p < 0.01$) than A2's average score. Agent A1's score cannot compete with either of the other agents' scores.

5.3 Alternative Performance Measuring Approaches

An alternative performance metric would have been measuring the accuracy of the goal completion and returning valid course packages which satisfy the hard constraints. Thus, this is a valid performance metric for many tasks, A2 and A3 were explicitly designed

to at least achieve this goal. The fact that the ontology contains enough valid course combinations further supports the achievability of this task. Those issues caused the decision to reject this alternative metric.

Despite the need for an internal performance metric, it is insufficient to capture the holistic performance of the agents. The latter depends on the final acceptance of the user. An external performance metric may be an actual experiment which consists of asking users for preferences in a real case scenario. The scenario would entail real users with real preferences and an elaborate knowledge base. These would have been out of the scope of the project.

6. Discussion

6.1 Limitations

Even though the agent was able to achieve sufficient results, there are various ways the agent can be further improved.

Firstly, concerning the calculation of trust between agents (previously mentioned in subsection 4.3.1), the selection of a scaling factor function was not considered. A method for choosing such a function needs to be designed so that the function can perform optimally in the specific environment in which the agent could be deployed. This could conceivably be done experimentally or using an accurate simulation.

Secondly, currently, the number of courses in each package can be two or three, and the user is shown the package with the highest score independent of its size. The user experience might improve by adding package size to the hard restrictions as he would not have to manually reject every package of size three if they want to take two courses.

Furthermore, the ordering of the preferences is set by the user. Firstly, this may result in lower user satisfaction as the user might find it time consuming and unnecessary. In order to avoid this, the weights could be generated using alternative methods. Such methods include machine learning and experimental approaches. This will likely come at a cost to the satisfaction of the user with respect to the recommended packages. The agent will aim for the best outcome given the average users' preferences as opposed to an individualised recommendation through user-defined weights.

Additionally, the evaluation function is, in a sense biased towards larger packages. Hence, it merely checks how may preferences are met and are package-size agnostic.

During the ranking of the agent itself, the scoring function avoids this by including a normalisation term. The evaluation function should exhibit the same behaviour.

6.2 Conclusion

Overall, considering the results analysed in chapter 5 and the limitations assessed above, it is possible to conclude that even though agent performs generally good and is fully functional, there is still room for improvement.