Softwareprojekt Stundenplan

–

Projektbericht

Ismail Akgün, Lenja Dröge, Kai Gräf, Linus Hase, Roman Jacob,
Frederic Pierre Lamp, Henry Müller, Tom Oliver Mohr, Philipp Raschka, Finn Weigand

FH Wedel – Sommersemester 2024

Inhaltsverzeichnis

1 Vorstellung der Idee und geplantes Vorgehen

Bereits vor mehreren Semestern wurde der Grundstein für dieses Projekt gelegt, indem Studierende sich mit ein und derselben Problematik befassten: Ist es grundsätzlich möglich, die Verteilung von Vorlesungen eines bestimmten Semesters auf die Räume und Zeitslots der FH Wedel zu automatisieren und in Form eines Algorithmus zu implementieren?

Ursächlich für diese Fragestellung ist nach wie vor der Umstand, dass zu Beginn jeden Semesters alle Vorlesungen in einer mittlerweile stark veralteten Software gesammelt und aufwändig per Hand auf die einzelnen Zeitslots gelegt werden müssen. Da die Anzahl an Veranstaltungen pro Semester und die der möglichen Räume über die Jahre zunehmend gewachsen ist (nach heutigem Stand etwa 300 Veranstaltungen bei mehr als 40 Räumen), stellt dies einen sehr zeit- und nervenraubenden Prozess dar, welcher jedes Semester aufs Neue wiederholt werden muss. Problematisch ist zudem die Vielzahl an Sonderfällen, Bedingungen und Dozentenwünschen, die der momentane Verantwortliche Birger Wolter genau kennen und einplanen können muss.

Tendenziell ist dieses Problem nicht neu: Seit Jahrzehnten beschäftigen sich weltweit viele Mathematiker und Informatiker mit der Problematik von Lösungen für große Datensätze. Beispielsweise basiert die Kryptographie vereinfacht gesagt fundamental auf dem Problem, dass kein effizienter Algorithmus existiert, um eine Lösung für ein Problem mit vielen richtigen oder falschen Ergebnissen zu finden. Ähnlich sieht dies bei der Erstellung eines Stundenplanalgorithmus aus: Das Ausrechnen aller möglichen Lösungen und deren Bewertung ist mit der Rechenleistung heutiger Computer schlicht nicht möglich, weil es eine immense Zeit in Anspruch nehmen würde. Daher wurden für dieses Projekt zwei verschiedene Ansätze ausgewählt und verfolgt: Genetische Algorithmen und Künstliche Intelligenz (KI).

1.1 Genetische Algorithmen

Genetische Algorithmen basieren auf der natürlichen Funktionsweise der DNA in Lebewesen. Wie auch andere Vertreter sind sie der Gruppe der evolutionären Algorithmen zuzuordnen und machen sich die Konzepte der natürlichen Selektion und genetischen Mutation zu Nutze. Dabei beinhaltet jede Lösung eine spezielle Strukturierung von Teilhabern oder auch sogenannten Chromosomen, welche wie in der DNA zusammen eine gebündelte Lösung ergeben. Grundsätzlich werden am Anfang mehrere zufällige Genverteilungen erstellt, die noch keinem regulären Muster folgen, geschweige denn spezielle Bedingungen erfüllen. Je nach Anwendungsfall können diese aus menschlicher Sicht komplett unsinnig sein, was allerdings zunächst einmal irrelevant ist. Tendenziell gibt es im Rahmen von genetischen Algorithmen vier bzw. fünf Phasen, die für jede Generation durchlaufen werden:

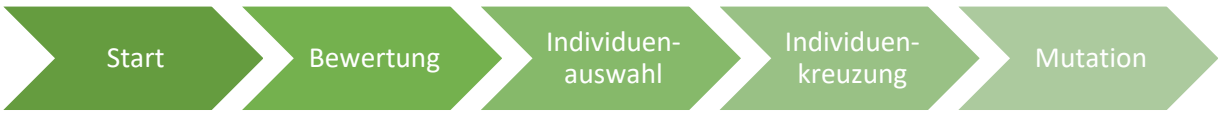

Abb. 1: Typische Phasen eines genetischen Algorithmus

In der eben bereits angerissenen Startphase wird der Initialzustand, auch bekannt als Startbevölkerung, erzeugt. Bei der ersten Ausführung handelt es sich dabei um eine mehr oder weniger zufällige Anordnung von Genen, in späteren Phasen basiert die Bevölkerung bereits auf Erkenntnissen und durchgeführten Mutationen.

In der zweiten Phase folgt die Bewertung der einzelnen Lösungen bzw. Individuen. Hierfür gibt es eine sogenannte Fitnessfunktion, welche alle Teile der Lösung betrachtet und anhand einer vordefinierten Skala jedem Individuum einen Gesamtwert zuordnet. Je nach implementierten Verfahren bedeutet eine höhere Bewertung meist eine bessere Lösung. Abhängig von der konkreten Problemstellung können hier zum Beispiel auch verletzte Bedingungen bestraft werden, was somit zu einer niedrigeren Punktzahl führen würde.

Die dritte Phase handelt von der Auswahl der zuvor bewerteten Individuen. Hierfür gibt es mehrere Ansätze, wie genau diese ausgewählt werden, angefangen bei einer reinen Zufallsauswahl bis hin zur spezifischen Auswahl besonders geeigneter Elite-Individuen. Dabei muss allerdings beachtet werden, dass die strikte Auswahl der besten Lösungen nicht zwingend zum besten Gesamtergebnis führen muss, weshalb die Wahl der richtigen Methode essentiell für einen funktionierenden Algorithmus ist.

Darauf folgt die vierte Phase, in der die Kreuzung von verschiedenen Individuen stattfindet. Genau wie in der Biologie wird hier durch die Kreuzung von zwei ursprünglichen Individuen ein neues Individuum geschaffen. Dieses beinhaltet nun die Teilanordnung von seinen Eltern und wird für die nächste Generation wiederum einen Teil seiner DNA abgeben.

Die fünfte Phase beschäftigt sich mit Mutationen. Hier werden per Zufall einzelne Chromosomen verändert, was im Ergebnis ein leicht anderes Individuum erzeugt. Tendenziell sind dies nur sehr leichte Veränderungen und die Chance für eine Mutation ist vergleichsweise gering. Auf diesem Wege werden so tausende Generationen gebildet, wo jede eine gewisse Anzahl an Individuen beinhaltet. Da hier der Zufall der maßgebliche Ausschlag für eine neue Generation ist, ist dieser Algorithmus in hohem Maße nicht-deterministisch und würde bei wiederholter Ausführung wohlmöglich zu einem anderen Ergebnis führen. Auch ist es möglich, dass ein suboptimales Ergebnis erreicht wird, sollte sich der Algorithmus in einem lokalen Optimum verfangen, obwohl in der Realität ein besseres globales Optimum existiert.

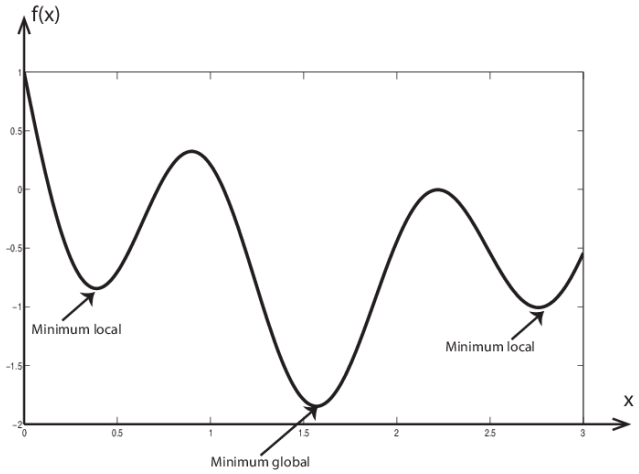

Abb. 2: Vergleich lokales und globales Optimum

1.2 Künstliche Intelligenz (KI)

Das Themengebiet der künstlichen Intelligenz – oder einfach kurz KI – ist nun schon seit etlichen Jahren auf Platz 1 der technischen Neuerungen und damit ein vielversprechendes Werkzeug für eine Vielzahl an Problemen. Gleichzeitig stellt es allerdings auch ein riesiges Forschungsgebiet dar, weshalb sich dieser Bericht nur mit der Funktionalität der sogenannten Graph Neural Networks (GNNs) auseinandersetzt. Sie repräsentieren eine fortschrittliche Klasse von neuronalen Netzwerken, die speziell für die Verarbeitung von Daten in Graphenform geeignet sind. Herkömmliche neuronale Netzwerke stoßen oft an ihre Grenzen, wenn besonders viele und komplexe Datenstrukturen verwendet werden, wohingegen GNNs diese im Vergleich besser verarbeiten können. Dies wird durch die Nutzung von Graphen erwirkt, welche als Netz aus Knoten und Kanten dargestellt werden. Besonders in den letzten Jahren werden diese immer mehr in Feldern wie der Vorhersage von Proteinstrukturen, Molekülinteraktionen oder der Verbesserung von Empfehlungssystemen angewandt. Genau wie bei den primitiven neuronalen Netzwerken ist allerdings auch hier eine große Menge an Daten erforderlich, um die Leistung und Genauigkeit der Vorhersagen signifikant zu verbessern.

2 Projektverlauf

Unser Team bestand von Anfang an aus zehn Personen. Der geplante Umfang des Projektes belief sich auf etwa zweieinhalb Monate Entwicklungszeit, beginnend am 24.04.2024 bis zum 10.07.2024. Aus Gründen der Effizienz und Schnittstellenbildung haben wir dazu folgende Teamaufteilung vorgenommen:

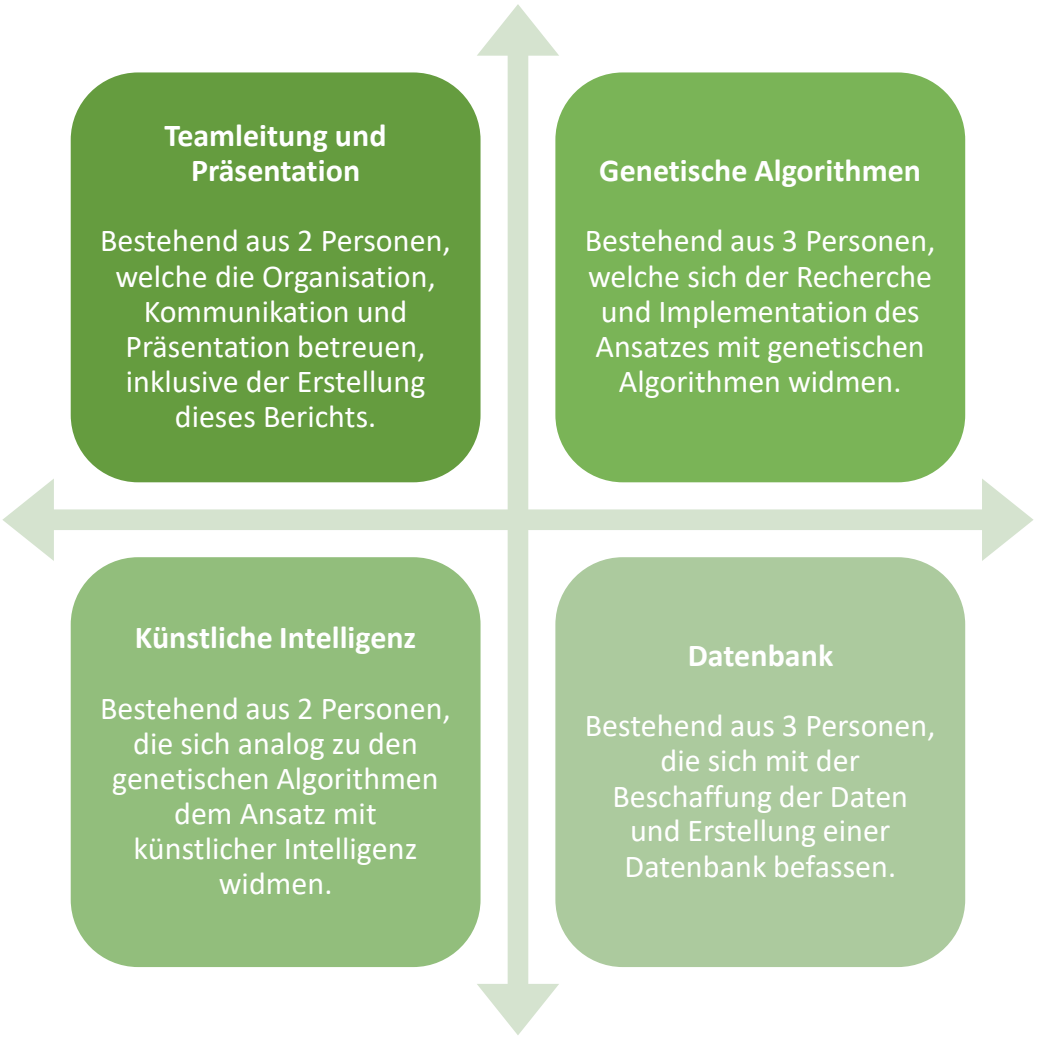

Abb. 3: Teamaufteilung

An erster Stelle stand die genaue Festlegung, was im Rahmen des Projekts entwickeln werden sollte. Unsere Kernanforderungen waren dabei wie folgt:

- Das Endprodukt in Form eines Algorithmus soll basierend auf einer Datenquelle eine funktionierende und sinnvolle Zuordnung von Veranstaltungen zu Räumen für ein Semester liefern.
- Jeder Dozent kann nur eine Vorlesung gleichzeitig halten, welche in einem passenden Raum in einem festgelegten Zeitslot stattfindet. Manche Dozenten können nur an bestimmten Tagen ihre Vorlesungen halten.
- Es darf keine Überschneidungen in der Zuhörerschaft geben, wenn zwei unterschiedliche Veranstaltungen zur selben Zeit stattfinden.
- Jeder Raum kann nur eine Veranstaltung gleichzeitig beherbergen.

Ausgehend von all diesen Bedingungen haben wir uns die Grundstruktur für unser weiteres Vorgehen überlegt. Es war uns allen wichtig, dass wir nicht planlos und unstrukturiert in die Entwicklung einsteigen, daher wurden die ersten beiden Wochen hauptsächlich mit der Recherche von geeigneten Methoden und Tools verbracht. Ebenso wurde uns schnell bewusst, dass wir auf jeden Fall eine Datenquelle in Form einer Datenbank brauchen, um die beiden entwickelten Algorithmen zu einem späteren Zeitpunkt testen zu können. Eine Kernentscheidung, die in diesem Zeitraum gefällt wurde, war die verwendete Programmiersprache: Unsere Wahl fiel auf Python, eine dynamisch typisierte, einsteigerfreundliche und üblicherweise interpretierte Hochsprache, die aufgrund einer hohen Anzahl an frei verfügbaren Bibliotheken bzw. Frameworks und ihrer Plattformunabhängigkeit häufig im Bereich Data Science zum Einsatz kommt.

2.1 Datenbank

Da wir effektiv planten, zwei verschiedene Algorithmen benutzen, war es essentiell, eine universelle Schnittstelle zwischen den Algorithmen und der Datenbank zu erstellen. Dies dient der Unabhängigkeit und Wartbarkeit und soll gewährleisten, dass Veränderungen an einer Stelle der Software keine oder möglichst geringe Auswirkungen auf andere Bereiche haben. Um auf aufwändige Serverstrukturen zu verzichten, haben wir uns für das leichtgewichtige und weitverbreitete Framework SQLite (bzw. SQLite3, siehe https://docs.python.org/3/library/sqlite3.html) entschieden, welches seine Datenbank wahlweise in-memory (also zur Laufzeit im Arbeitsspeicher) oder in Form einer einzigen Datei verwaltet.

Die Datenbank selbst wurde basierend auf den aktuellsten Daten des vergangenen Sommer- und Wintersemesters (2023) erstellt. Viele Informationen zu den Mitarbeitern und der Infrastruktur entstammen dabei der offiziellen Webseite der FH Wedel (https://www.fh-wedel.de/), der internen Seite (https://intern.fh-wedel.de/), aber vor allem auch der Stundenplanseite (https://intern.fh-wedel.de/~splan/), die nach aktuellem Stand nach wie vor zur Anzeige der manuell erstellten Stundenpläne verwendet wird. Das Datenbankschema wurde im Rahmen der Entwicklung mehrfach angepasst und erweitert. Ausgangspunkt war ein ER-Diagramm, dessen finale Version auf der folgenden Seite dargestellt ist.

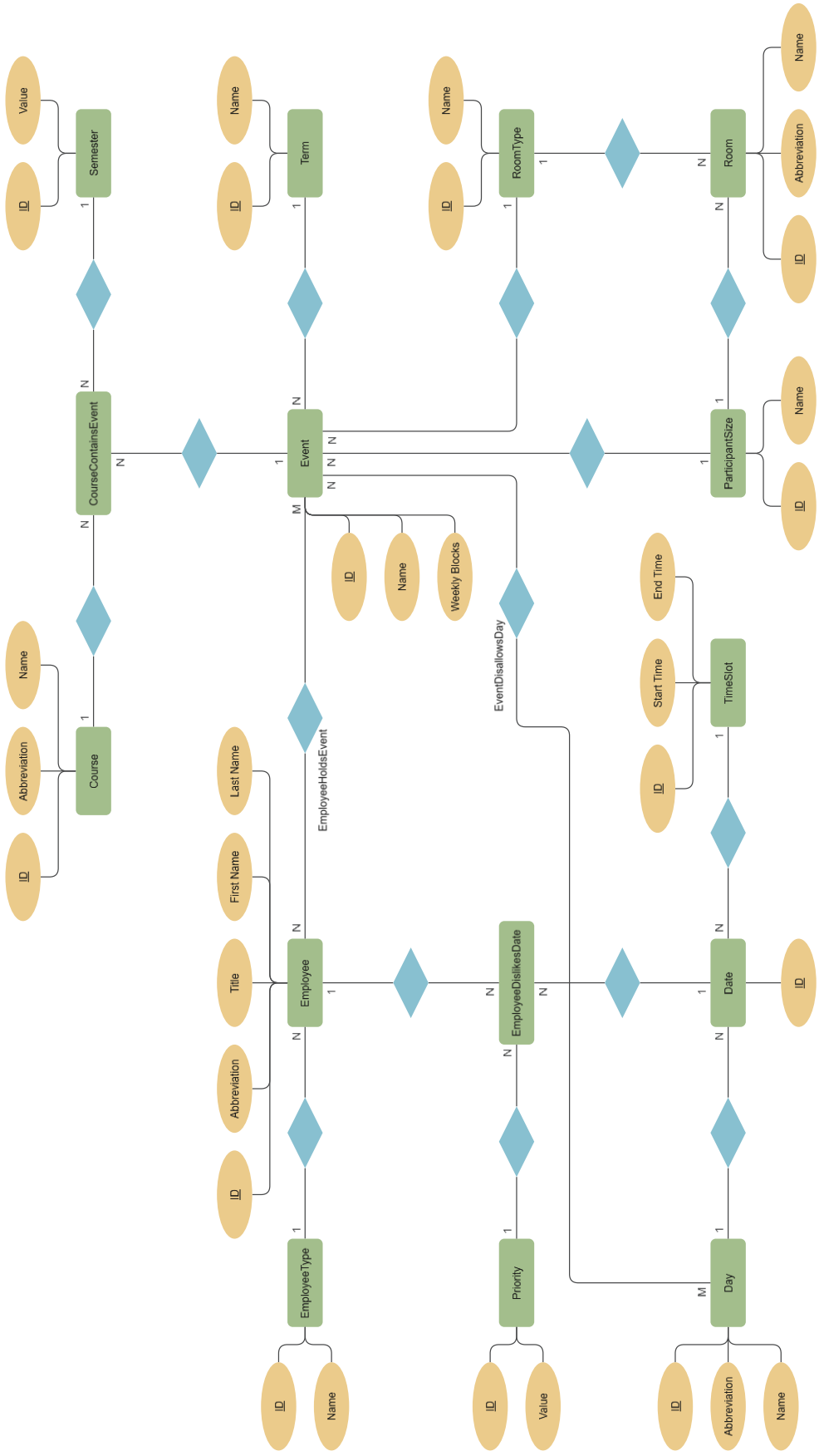

Abb. 4: Struktur der Datenbank als ER-Diagramm

Insbesondere bei der Datenbank, die als Fundament des gesamten Projekts dient, war ein modularer und erweiterbarer Entwurf unverzichtbar. Aus diesem Grund haben wir uns dazu entschieden, sogenannte Data Access Objects (DAOs) zu verwenden, die von der konkreten Datenbank abstrahieren und eine einheitliche Menge von Operationen gemäß dem CRUD-Zyklus (Create, Read, Update, Delete) anbieten. So kann die darunterliegende Datenbank in Zukunft mit geringerem Entwicklungsaufwand ausgetauscht werden.

Die DAOs erzeugen bzw. verwenden wiederum sogenannte Data Transfer Objects (DTOs), welche die in der Datenbank gespeicherten Entitäten in Form von objektorientierten Nutzlasten modellieren. Sie stellen bestimmte Invarianten (Attributwerte und -bereiche, Beziehungen, etc.) sicher und befinden sich aber noch auf einem eher niedrigen Abstraktionsniveau (vergleichsweise nah am Datenbankschema). Vorteilhaft ist hingegen die Objektorientierung, welche eine komfortable Verarbeitung und Erweiterung der Entitäten ermöglicht.

Eines der beiden großen Fragezeichen bestand darin, wie wir die eigentlichen Daten sammeln und in der Datenbank speichern sollen. Nach einigen Abwägungen fiel unsere Wahl auf eine spezielle, eigenhändig erstellte Exceldatei, die alle für den Algorithmus benötigten Daten speichern sollte. Dieses Vorgehen hat gegenüber einfachen Text- oder CSV-Dateien einige Vorteile:

- Vertraute, grafische Benutzeroberfläche zur Anpassung von Daten.
- Automatisierte Sortierungs-, Filterungs- oder Bearbeitungsmöglichkeiten durch Funktionen und Makros.
- Vergleichsweise einfache Erweiterung um neue Daten, alles zentral an einem Ort gespeichert.

Für das Lesen und Speichern der Daten haben wir einen Excelparser entwickelt, der das weitverbreitete Data-Science-Framework Pandas (https://pandas.pydata.org/) nutzt. Dieses ermöglicht ein komfortables Einlesen der gesamten Exceldatei und stellt die Arbeitsblätter für die einzelnen Entitäten der Datenbank in Form sogenannter Dataframes bereit. Dabei handelt es sich um eine zweidimensionale, tabellarische Datenstruktur beliebiger Daten, die anschließend vom Excelparser zeilenweise verarbeitet wird. Aus jeder Zeile wird (sofern die Felder gültig sind) ein entsprechendes DTO erzeugt und mithilfe der DAOs in der Datenbank für die spätere Verwendung abgespeichert.

Im Folgenden sei hier eine vereinfachte Darstellung des Ablaufs anhand der Tabelle für die Räume der FH Wedel angeführt. Diese besitzt grundsätzlich folgenden Aufbau (beispielhafte Auszüge):

Abbreviation	Name	Capacity	Room Type
EXT	Externes Unternehmen	Normal	Externes Unternehmen
HS06	Hörsaal 6	Normal	Hörsaal
HS07	Audimax	Large	Hörsaal
LR10	Virtual-Reality-Labor	Normal	Virtual-Reality-Labor
LR12	Audiolabor	Normal	Audiolabor
OL01	Onlineveranstaltung	Normal	Onlineveranstaltung
PC01	PC-Pool 1	Normal	PC-Pool
SR01	Konferenzraum	Normal	Konferenzraum
SR02	Aristoteles	Small	Seminarraum

Abb. 5: Aufbau der Tabelle für die Räume der FH Wedel

Für das eigentliche Parsen im Excelparser wird je Entität ein Data Transfer Object (DTO) benötigt, das aus den Daten einer einzelnen Zeile erstellt und anschließend in der Datenbank gespeichert wird.

```
# Data Transfer Object (DTO) für einen Raum der FH Wedel (vereinfacht).

@dataclass(frozen=True, kw_only=True)
@final
class RoomDTO:
    id: int
    abbreviation: str
    name: str
    participant_size_id: int
    room_type_id: int

# Parsen der Räume im Excelparser (vereinfacht).

def parse_rooms(data_frame: DataFrame) -> None:
    participant_sizes_by_name: dict[str, ParticipantSizeDTO] = {
        participant_size_dto.name: participant_size_dto
        for participant_size_dto in ParticipantSizeDAO().select_all()
    }
    room_types_by_name: dict[str, RoomTypeDTO] = {
        room_type_dto.name: room_type_dto
        for room_type_dto in RoomTypeDAO().select_all()
    }
    room_dtos: list[RoomDTO] = [
        RoomDTO(
            id=0,
            abbreviation=row["Abbreviation"],
            name=row["Name"],
            participant_size_id=participant_sizes_by_name[row["Capacity"]].id,
            room_type_id=room_types_by_name[row["Room Type"]].id,
        )
        for row in data_frame.to_dict(orient="records")
    ]
    for room_dto in room_dtos:
        RoomDAO().insert(room_dto)

def parse(excel_file_path: str = DEFAULT_EXCEL_FILE_PATH) -> None:
    data_frames: dict[str, DataFrame] = pandas.read_excel(
        excel_file_path, sheet_name=None
    )
    for data_frame in data_frames.values():
        data_frame.fillna("", inplace=True)
    parse_rooms(data_frames["Room"])

def main() -> None:
    logging_config.configure_logging()
    Database().initialize(delete_database_file=True)
    parse()
```

Abb. 6: Grundlegender Ablauf des Parsens und Speicherns von Daten in der Datenbank

Daneben war zu klären, wie bzw. in welcher Form die Algorithmen auf die Datenbank zugreifen sollten. Die denkbar einfachste Variante wäre die direkte Bereitstellung und Nutzung der DTOs in der Logik gewesen. Obgleich dies vermutlich den geringsten Entwicklungsaufwand bedeutet hätte, haben wir uns aufgrund folgender Gründe dafür entschieden, dies nicht so umzusetzen und stattdessen eine zusätzliche Abstraktionsebene in Form einer API (Application Programming Interface) umzusetzen:

- Die DTOs befinden sich auf einer vergleichsweise niedrigen Abstraktionsebene und liegen insbesondere aus Gründen der Normalisierung in verteilten Tabellen vor. Dies würde für die Algorithmen an vielen Stellen ein manuelles Zusammenführen (engl. join) der Objekte notwendig machen.
- Ohne Zwischenschicht werden die Algorithmen und die Datenbank mit ihren DAOs und DTOs enger gekoppelt, was die Austauschbarkeit in Zukunft erschweren dürfte.

Somit sitzt zwischen der Datenbank und den Algorithmen eine einheitliche Schnittstelle, die für Unabhängigkeit und Wartbarkeit sorgen soll. Die API bietet über ihre lesenden Operationen sogenannte Models an, welche die eigentlichen Objekte der Anwendungslogik darstellen. Letztere sind insgesamt ähnlich zu den DTOs aufgebaut, befinden sich nun allerdings auf einer höheren Abstraktionsebene: Sie kombinieren Daten mehrerer Tabellen (z. B. ein Datum, dass sich aus einem Tag und einem Zeitslot zusammensetzt) und können – sofern notwendig – auch zusätzliche Operationen zur Verarbeitung anbieten, was bei den primitiven DTOs eher unüblich ist (reine Nutzlast).

2.2 Künstliche Intelligenz

Zunächst haben wir uns im Rahmen der Recherche damit beschäftigt, wie genau KI in diesem Feld angewandt werden kann. Uns ist relativ schnell bewusst geworden, dass GNNs der richtige Ansatz für diesen Anwendungsfall sind und haben uns vollständig auf diese konzentriert. Dabei bestand unsere erste Idee darin, alle Kurse, Professoren und Räume als Knoten darzustellen und die jeweiligen Beziehungen als Kanten. Auf diesem Wege hätte jeder Knoten eine eindeutige ID und zu jeder Veranstaltung können spezielle Anforderungen definiert werden, wie eine spezielle Raumart oder ein Zeitslot. Damit das GNN die Daten jedoch effizient verarbeiten kann, müssen alle Daten in numerische Werte umgewandelt werden, inklusive der verschiedenen Tage und Zeitslots.

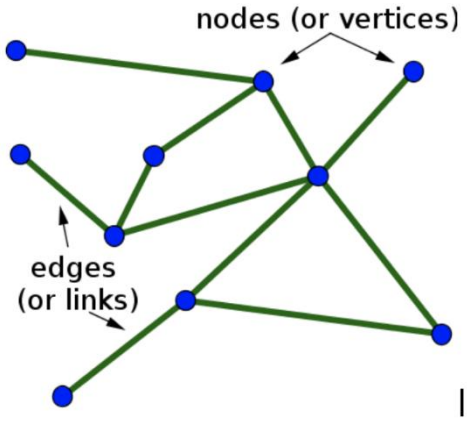

Abb. 7: Aufbau eines GNN aus Knoten und Kanten

Der nächste Schritt bestünde in der Aufstellung des Netzes und der Bereitstellung der Testdaten. Generell gilt hier der ungefähre Richtwert, dass 70% der Daten zum Trainieren, 15% zum Validieren und 15% zum Testen verwendet werden.

Damit das neuronale Netz nicht nur auf die Testdaten spezialisiert ist, würde eine Methode wie die k-Fold-Kreuzvalidierung hier passen, um das Ergebnis zu generalisieren. Dadurch würden wir eine generelle Einschätzung bekommen, wie divers unser Algorithmus ist. Als letzten Schritt würden wir das Endergebnis evaluieren, um zu schauen, wie genau das Ergebnis ist und ob die Anforderungen eingehalten wurden. Daraufhin könnten wir dann das GNN verbessern, bis wir eine voll funktionsfähige Software für die Erstellung eines Vorlesungsplans erstellt hätten.

Für diese Umsetzung haben wir uns zwei verschiedene Frameworks herausgesucht, die eine solche Implementation ermöglichen würden. Bei dem ersten handelt es sich um PyTorch Geometric (PyG), welches auf dem normalen PyTorch aufbaut und daher leichter zu implementieren ist. Die Alternative ist die Deep Graph Library (DGL), welche um einiges umfangreicher ist und somit mehr Funktionen für die Implementierung anbietet.

Die Umsetzung sah zunächst vielversprechend aus und in diesem Sinne haben wir uns mit Marco Pawlowski (Dozent mit KI-Hintergrund an der FH Wedel) getroffen, um seine Einschätzung und Meinung zu dem Thema zu bekommen. Nach einem sehr informativen Gespräch haben wir einen noch besseren Überblick über das Thema bekommen und eine Einschätzung, was uns genau erwartet.

Das große Problem stellte die mangelnde Zeit dar. Nach etwa drei Wochen Recherche und Aufbereitung der Daten sind wir zu dem Entschluss gekommen, dass dieser komplexe Ansatz in den nunmehr knapp zwei Monaten verbleibender Zeit nicht realistisch umsetzbar ist. Ein weiteres Problem war das Fehlen einer großen Menge an Testdaten, die wir für das Training des GNN verwendet hätten. Da ein händischer Entwurf bereits eines einzigen gültigen Stundenplans momentan mehrere Tage braucht, um erzeugt zu werden und wir hunderte bis tausende verschiedene Beispiele bräuchten, war dies ein weiteres KO-Kriterium für diesen Ansatz.

Daraus ergaben sich zwei Möglichkeiten: Entweder koppelt sich das KI-Team ab und forscht weiter in Richtung GNNs – ohne jedoch ein fertiges Produkt zu entwickeln – oder es tritt dem Team der genetischen Algorithmen bei. Letztendlich haben wir uns für die zweite Variante entschieden, sodass wir unsere gesamte Leistung auf einen Algorithmus fokussieren konnten.

2.3 Genetische Algorithmen

Ähnlich wie beim KI-Team haben wir hier erst einmal recherchiert, wie genau genetische Algorithmen funktionieren und welche Implementationsansätze grundsätzlich möglich sind. Da dieses Thema deutlich einfacher und übersichtlicher ist, haben wir schnell verstanden, was die Vorteile und die Limitationen dieses Ansatzes sind. Beispielsweise bestand ein großer Vorteil im Vergleich zur KI darin, dass wir keine Testdaten brauchten, um einen funktionierenden Ansatz zu erzeugen. Zudem ist die Bewertung sehr leicht zu deuten und der Implementationsaufwand ist verglichen mit einem neuronalen Netz deutlich geringer. Eine große Problematik stellt jedoch die Laufzeit dar, da für jede Lösung tausende Generationen gebildet werden müssen, was abhängig von der Rechenleistung der verwendeten Hardware Stunden oder theoretisch sogar Tage dauern kann.

Unsere ersten Überlegungen haben sich mit den Komponenten und dem Aufbau des Algorithmus beschäftigt. In unserem Fall war das Gen die Verbindung von einer Veranstaltung zu einen Raum-Zeitslot-Paar, da dies die Lösung am besten widerspiegelt. So ist es möglich, über die Generationen hinweg die Verbindung aufzulösen und einer Veranstaltung beispielsweise einen anderen Zeitslot zuzuordnen, wohingegen andere damit verbundene Daten wie ein Dozent oder die benötigte Raumgröße erhalten bleiben. Da es hier auch keinen (direkten) Unterschied zwischen einem gleichen Raum zu verschiedenen Zeit und einem anderen Raum zur gleichen Zeit gibt, können wir die beiden miteinander verknüpfen. Die gesamte Anzahl an Verknüpfungen spiegelt dann ein Chromosom wider, also eine mögliche Anordnung der Veranstaltungen zu einem gewissen Raum-Zeitslot. Verschiedene Chromosome sind dann die Population einer Generation.

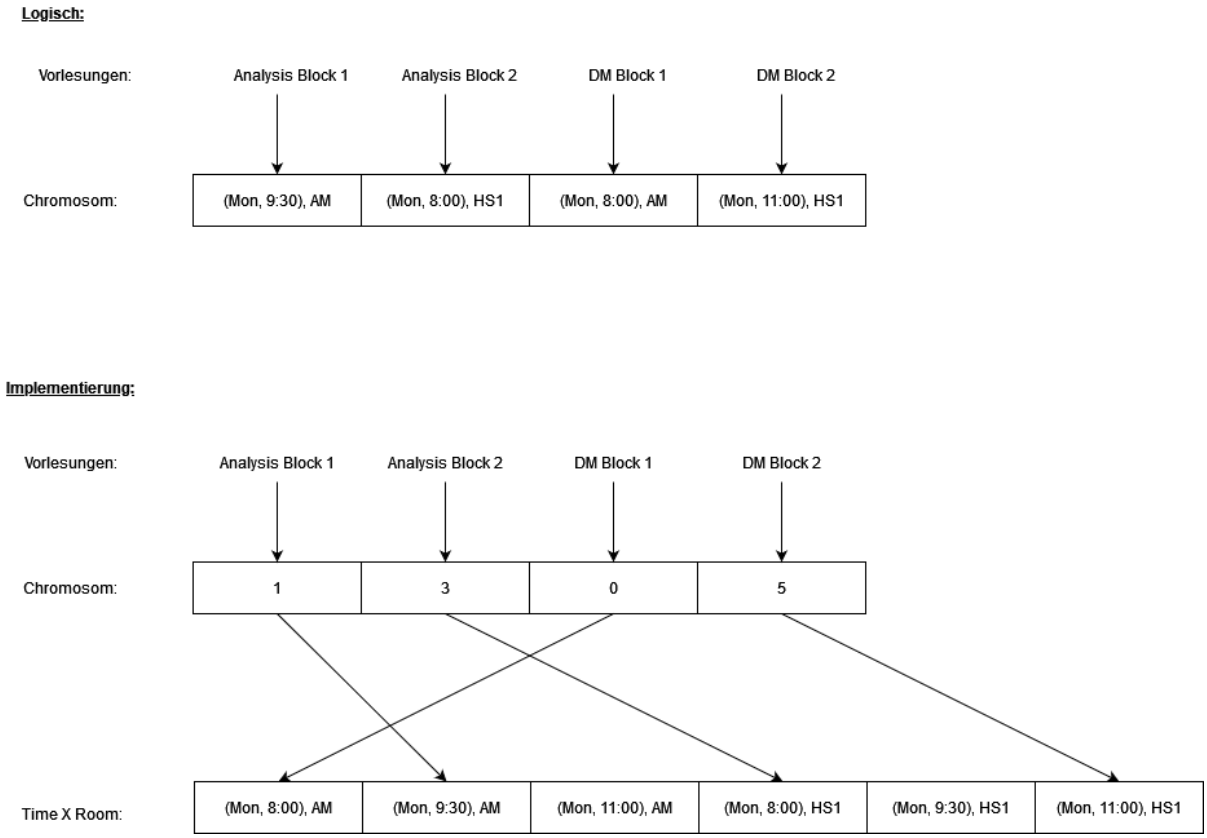

Abb. 8: Beispielhafter Aufbau der Chromosome im Algorithmus

Wie auch in Kapitel 2 erwähnt, würde dann der Ablauf von Kreuzungen und Mutationen beginnen, bis eine (hoffentlich) perfekte Lösung erreicht wird. Fundamental dafür ist die Fitnessfunktion, welche für jedes Chromosom eine Bewertung erstellt, um daraufhin am Ende eine neue Population zu bilden. Unser erster Ansatz daher darin, eine Bestrafung für nicht eingehaltene Regeln einzuführen, d. h. für jede verletzte Regel bzw. Bedingung wird eine Anzahl an Minuspunkten verteilt. Abhängig von der Wichtigkeit dieser Bedingung fällt diese größer oder kleiner aus, um so zuerst die Kernbedingungen mit großen Auswirkungen abzuarbeiten und sich später um die weniger relevanten zu kümmern.

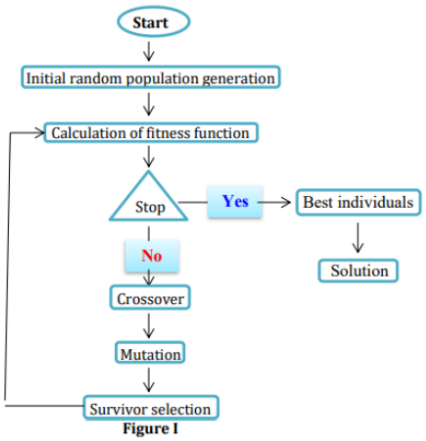

Abb. 9 Typischer Ablauf eines genetischen Algorithmus

Anfänglich war unser Anspruch, sowohl Hard- als auch Soft-Constraints zu implementieren, wobei ein Hard-Constraint zum Beispiel die Verhinderung einer doppelten Belegung eines Raum-Zeitslots-Paars und ein Soft-Constraint eine Abneigung eines Dozenten für einen speziellen Zeitslot wäre, die aber technisch gesehen trotzdem eine valide Lösung darstellen würde. Eine sinnvolle Umsetzung besteht darin, in der Fitnessfunktion zunächst nur die Hard-Constraints zu betrachten und daraus die jeweilige Bewertung der Lösung zu berechnen. Danach könnten auch die Soft-Constraints betrachtet und miteinbezogen werden. Hierbei ist es wichtig, eine angemessene Verteilung von Punkten festzulegen, sodass sich der Algorithmus nicht an kleineren, möglicherweise weniger relevanten Details festbeißt. Generell gilt, dass Hard-Constraints einen deutlich höheren Malus bekommen, sodass viele verletzte Soft-Constraints diese nicht überschatten können. Schlussendlich würde am Ende ein perfektes Ergebnis die Punktzahl 0 haben. Anhand der Bewertung gibt es dann verschiedene Verfahren zur Auswahl der richtigen Gene, die wir alle einmal ausprobieren wollten.

Der nächste Schritt wäre das Kreuzen und Mutieren der für die nächste Population ausgewählten Chromosomen. Da ein Standardwert für die Mutation 5% pro Chromosom ist, haben wir uns erst einmal daran orientiert. Für die Mutation würden dann zufällig 0 – 10 Gene ausgewählt werden, denen dann ein neues Raum-Zeitslot-Paar zugewiesen würde. Für die Kreuzung werden anschließend jeweils eine Anzahl von Genen aus einem Chromosom genommen und mit den Genen eines anderen Chromosoms getauscht, sodass beide neuen Chromosome danach die gleiche Anzahl an Genen haben (1 Gen pro Veranstaltung).

Da, nach der ausführlichen Recherche prinzipiell keine Probleme mehr im Weg standen, haben wir mit der Entwicklung begonnen. Als Framework haben wir uns für PyGAD entschieden, da dies eineVielzahl von verschiedenen Methoden für die Auswahl, Kreuzung und Mutation bereitstellte.

Um nicht direkt mit einer unüberschaubaren Menge an Problemen konfrontiert zu werden, haben wir als erstes ein einfaches Testsystem mit minimalen Bedingungen geschaffen: Auf zwei Tage mit jeweils 7 Zeitslots sollte der Algorithmus zehn Veranstaltungen verteilen, ohne den Hard-Constraint „Jeder Raum darf zurzeit nur eine Vorlesung haben“ zu verletzen. Eine gewisse Herausforderung stand anfänglich mit der Verknüpfung der Datenbank zum Algorithmus an, weil zu diesem Zeitpunkt nur die DAOs und DTOs weitestgehend entwickelt waren, die sich aber aus den bereits angeführten Gründen nur bedingt für die Anwendungslogik eignen (siehe Datenbank).

Glücklicherweise konnten wir in der darauffolgenden Woche die Vereinfachung in Form der API mit ihren lesenden Operationen fertigstellen, sodass das Team der genetischen Algorithmen nach einer kurzen Entwicklungspause wieder die Arbeit aufnehmen konnte. Somit konnten wir unsere ersten Ergebnisse erstellen, die wie erwartet schnell eine Lösung mit dem Fitnesswert 0 zurücklieferten. Um ein bisschen mehr Abwechslung einzubringen, haben wir im Anschluss die Bedingung hinzugefügt, dass sich zwei Veranstaltungen im gleichen vorgesehenen Semester für einen Studiengang zeitlich nicht überschneiden dürfen. Zusätzlich haben wir den Hard-Constraint implementiert, welcher die Verfügbarkeit von Dozenten zu einem gewissen Zeitslot überprüft. Zuletzt haben wir die Bedingung, dass ein Dozent nur eine Vorlesung zurzeit halten kann, um weitere Probleme auszuschließen.

Im Endeffekt haben wir uns auf die Auswahlmethode Tournament festgelegt, welche kleine Gruppen von Chromosomen bildet und den besten dieser Gruppe in die nächste Generation mitaufnimmt. Somit werden nicht immer nur die allerbesten Chromosome ausgewählt, was in unserem Falle zu einer größeren Diversität geführt hat. Für die Kreuzung haben wir die Variante Scattered Selection benutzt, welche anhand von einem zufällig erstellten Chromosom über Werte von zwei Ausgewählten geht, umso zwei neue Chromosome zu erstellen: Eines, welches nur Überschneidungen aufweist und eines, dass überhaupt keine Überschneidungen besitzt. Analog haben wir für die Mutation das Adaptive-Mutations-System benutzt, das Chromosome zwar per Zufall mutiert, aber eine Fallunterscheidung anhand der Fitnessbewertung vornimmt: Chromosome mit einer niedrigen Bewertung werden eher und stärker mutiert als solche mit einer höheren Bewertung.

Ein immer wiederkehrendes Problem war die spezielle Raumzuordnung. Da einige Veranstaltungen wie z. B. Analysis deutlich mehr Teilnehmer besitzen als andere, mussten wir einen Weg finden, dass ein Raum für eine Veranstaltung mindestens so viele Sitze zur Verfügung stellt, wie für die Teilnehmeranzahl benötigt wird. Unsere erste Implementation war ein genauer Vergleich der Sitze mit der geschätzten Anzahl der Teilnehmer. Dies brachte allerdings eine Herausforderung mit sich: Zwar war die Anzahl der Sitze innerhalb der Räume durch unsere Recherchen bekannt, aber es bestand keine Möglichkeit, konkrete Zahlen für die geschätzte Teilnehmerzahl herauszufinden. Daher haben wir zunächst zu Testzwecken jeder Veranstaltung die niedrigste Teilnehmerzahl zugewiesen, um unsere Implementation mit einigen Ausnahmefällen erfolgreich testen zu können. Im Nachhinein haben wir den Ansatz mit konkreten Zahlen verworfen und anstelle dessen einen Aufzählungstyp mit den Werten Small, Normal und Large benutzt, welchen wir anhand der Anzahl an Studiengängen pro Veranstaltung schätzungsweise vergeben haben.

Da wir aus Zeitgründen nicht mehr die Funktionalität aller Soft-Constraints implementieren konnten und die Hard-Constraints auf keinen Fall bei der Endlösung verletzt sein dürfen, haben wir momentan den Vorteil, dass wir den Algorithmus so lange laufen lassen können/müssen, bis der Wert eines Chromosoms bei 0 landet, was wiederum ein perfektes Ergebnis bedeutet. So sind spätere Anpassungen deutlich leichter zu messen, da wir nie in ein lokales Minimum fallen.

Zuletzt haben wir dann in Phasen den Algorithmus in seiner Vollständigkeit mit allen 250 Veranstaltungen und 40 Räumen für ein gesamtes Semester getestet. Nach immer mehr kleineren Verbesserungen sind wir im Durchschnitt nach ungefähr 8000 Generationen auf eine Bewertung von 0 gestoßen.

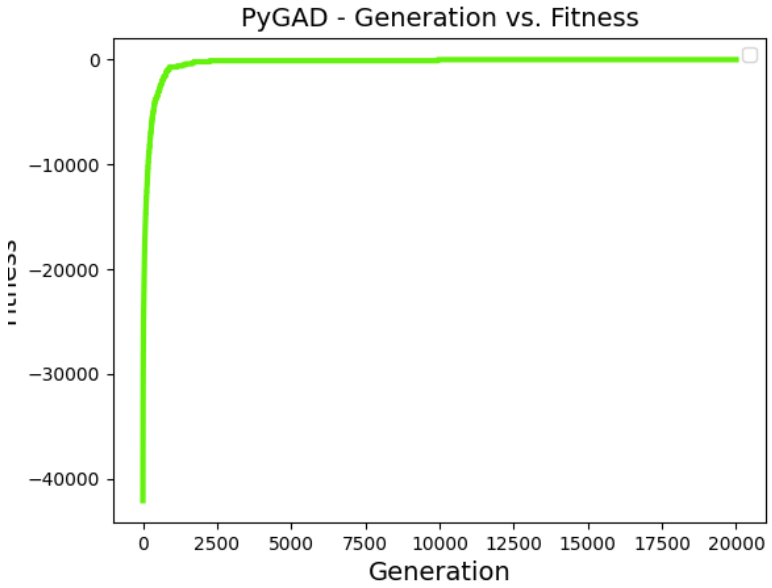

Abb. 10: Verlauf der erzielten Fitness bei zunehmender Generationsanzahl

3 Aktueller Stand, Ausblick und Verbesserungsmöglichkeiten

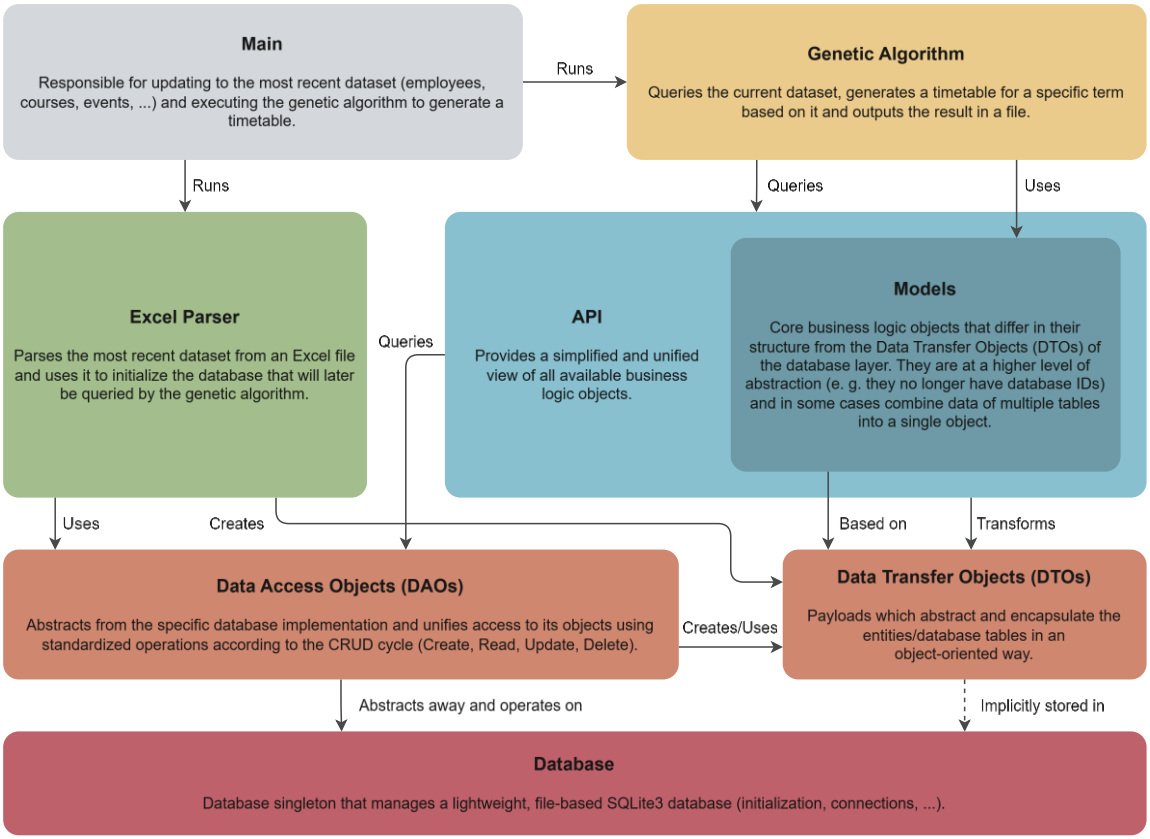

Abb. 11: Übersicht der einzelnen Komponenten der Anwendung und deren Verbindungen

Nach aktuellem Stand kann unser genetischer Algorithmus für aus einer Exceldatei stammende Tabellen, bestehend aus Veranstaltungen, Räumen und Dozenten, eine Verteilung auf die einzelnen Räume und Zeitslots erstellen, ohne jegliche Hard-Constraints zu brechen. Die für einen vollständigen Durchlauf nötige Laufzeit beläuft sich auf etwa ein bis drei Stunden und die Ausgabe erfolgt momentan in Form einer einfachen Textdatei, in der sowohl der gesamte Stundenplan als auch die einzelnen Pläne der Studiengänge sortiert nach Semestern enthalten sind.

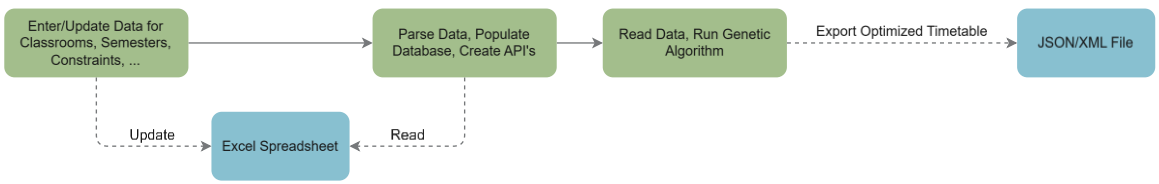

Abb. 12: Darstellung des aktuellen/geplanten Programmablaufs

Der Algorithmus funktioniert tendenziell gut, wir konnten allerdings wegen der recht kurzen Projektdauer nicht alle unsere Anforderungen erfüllen. Daher sei im Anschluss noch eine Auflistung von möglichen Verbesserungen und Erweiterungen genannt (aufsteigend sortiert von geringem bis hohem Implementationsaufwand).

3.1 Ausgabe als JSON-/XML-Datei

Die Ausgabe in Form einer einfachen Textdatei ist zurzeit noch sehr rudimentär. Um den Vorlesungsplan in das aktuell genutzte System (https://intern.fh-wedel.de/~splan/) einspielen zu können, muss das Ergebnis in eine XML-Datei mit einem speziellen Aufbau konvertiert werden. Alternativ könnte die Ausgabe auch zusätzlich im deutlichen schlankeren JSON-Format erfolgen, das von modernen Webbrowsern ohnehin nativ unterstützt wird. In beiden Fällen müsste das Endergebnis des genetischen Algorithmus (z. B. durch Parsen) in ein jeweils passendes Ausgabeformat umgewandelt werden, was tendenziell mit relativ geringem Aufwand umzusetzen sein sollte.

3.2 Blockveranstaltungen

Gewisse Veranstaltungen müssen nacheinander als Blockveranstaltung stattfinden, wohingegen andere nicht am selben Tag stattfinden dürfen. Diese Bedingung ist zum jetzigen Zeitpunkt noch nicht implementiert und stellt gewissermaßen einen weiteren Hard-Constraint dar, der für eine schlussendlich erfolgreiche Benutzung des Algorithmus erfüllt sein muss.

3.3 Fest eingeplante Vorlesungen

Möglicherweise wäre es sinnvoll, dem Algorithmus eine Menge bereits fest eingeplanter Vorlesungen zu übergeben, darunter beispielsweise Kolloquien oder Wochenendveranstaltungen. Ebenso sollte die Möglichkeit bestehen, die entsprechenden Timeslots und Räume dafür zu sperren. Dies würde die Zuweisung dieser Vorlesungen erheblich erleichtern und sich vermutlich positiv auf die Laufzeit des Algorithmus auswirken, da durch das Sperren der einzelnen Raum-Zeit-Paare eine geringere Anzahl an Hard-Constraints geprüft werden müsste.

3.4 Soft Constraints

Momentan sind keine Soft-Constraints wie „Möglichst wenig Freistunden zwischen den Veranstaltungen“ oder „Veranstaltungen lieber früher als später“ implementiert. Möglich wäre eine einfache Implementation mit Hilfe von niedrigen Malussen, die in die Fitness miteinbezogen werden. Da dies einige Probleme mit sich bringt, ist andernfalls die bereits thematisierte Methode von zwei verschiedenen Bewertungen, jeweils für Hard- und Soft-Constraints, denkbar. Dies ist deutlich zeitaufwändiger, aber höchstwahrscheinlich notwendig für einen real einsetzbaren Algorithmus.

3.5 Wahl des Frameworks für genetischen Algorithmus

Die wohl aufwändigste Verbesserung wäre die Implementierung eines genetischen Algorithmus auf Basis eines anderen Frameworks. PyGAD bietet zwar eine gute Grundlage für die Verarbeitung der Daten, hat aber als Hauptproblem die rein sequentielle Verarbeitung zur Laufzeit. So ist es nicht möglich, die Laufzeit mit Methoden wie Multithreading oder Multiprocessing gewinnbringend zu verkürzen. Andere Libraries ermöglichen hingegen eine solche Umsetzung und wären daher höchstwahrscheinlich effizienter. Da dieser Algorithmus aber effektiv nur einmal pro Semester ausgeführt werden soll, ist dies weniger relevant. Außerdem könnte die Implementation eines neuen Algorithmus durch die Schnittstelle zur Datenbank vereinfacht werden, welche nicht neuimplementiert werden muss.

4 Abbildungsverzeichnis