Burkhard-Keller Bäume zur approximativen Suche

Modulprojektbericht Programmierung II

Humanwissenschaftliche Fakultät
Department Linguistik

vorgelegt von: Simon Bross
Matrikelnummer: 809648

Link zum Projekt-Repository
Link zum letzten Commit
Link zum gegebenen Review

Berlin, den 5. September 2022

1. Einleitung

Der vorliegende Bericht dokumentiert die durchgeführte Implementation der Burkhard-Keller-Bäume Datenstruktur in Python. BK-Bäume stellen aufgrund ihrer inhärenten Eigenschaften, insbesondere des Zunutzemachens der Dreiecksungleichheit, ein effizientes Verfahren zur approximativen Suche von ähnlichen Wörtern bereit, das zur Rechtschreibprüfung und zur Generierung von Wortvorschlägen verwendet wird. Innerhalb der Datenstruktur werden Wörter als Baumknoten repräsentiert, die mit anderen Kindsknoten (Wörter) über Kanten verbunden sind. Diese Kanten sind mit einer Stringmetrik-Distanz etikettiert und spiegeln die Ähnlichkeit zwischen den zwei Wörtern wider.

Die Implementierung umfasst neben der konkreten Datenstruktur und zwei Stringmetriken (Levenshtein und LSC-Distanz) eine Kommandozeilenschnittstelle zur Erstellung von Bäumen aus einer .txt Datei, einen interaktiven Modus zur Baumdurchsuchung im Terminal, eine als .dot Datei abgespeicherte Visualisierung, sowie eine direkte grafische Visualisierung, die sich parallel zum Terminalprogramm als separates Matplotlib Fenster öffnet. Erstellte Bäume werden automatisch mit pickle serialisiert und abgespeichert und können mit der Kommandozeile deserialisiert werden, um sie ohne erneutes Aufbauen wieder zu visualisieren oder zu durchsuchen. Dem Projekt ist eine Demo-Anwendung mit einer kleinen Wortliste im Repository beigelegt, sowie Unittests für die Abstandsberechnungen mithilfe der implementierten Stringmetriken.

2. Aufbau und Programmstruktur

2.1 MVC-Muster

Das Projekt wurde anhand der Model-View-Controller Klassenarchitektur strukturiert, um Darstellung (View), Datenmanagement (Model) und die prinzipielle Programmlogik (Controller) voneinander zu trennen. Außerdem erleichtert das Muster das Einarbeiten von späteren Änderungen und die Implementation von Erweiterungen. Die drei Komponenten treffen im Hauptprogramm `main.py` aufeinander, welches den Programmablauf ausgehend von dem Input der Kommandozeilenschnittstelle initialisiert und verwaltet. Die folgende Tabelle gibt einen Überblick über die Aufgaben der einzelnen Komponenten:

Komponente	**Funktion/Aufgabe**
Model (BKTree Klasse)	Verwaltet die Datenstruktur (BK-Tree)
View (BKView Klasse)	Präsentiert/visualisiert die Daten des Models im Command-Line-Interface und nimmt User-Input zur Baumdurchsuchung entgegen
Controller (BKController Klasse)	Mittler zwischen Model und View

Tabelle 1: MVC-Komponenten

Die genaue Spezifikation und Beziehungen zwischen den Objekten der MVC-Architektur kann dem nachfolgenden UML-Diagramm entnommen werden:

2.2 UML-Diagramm

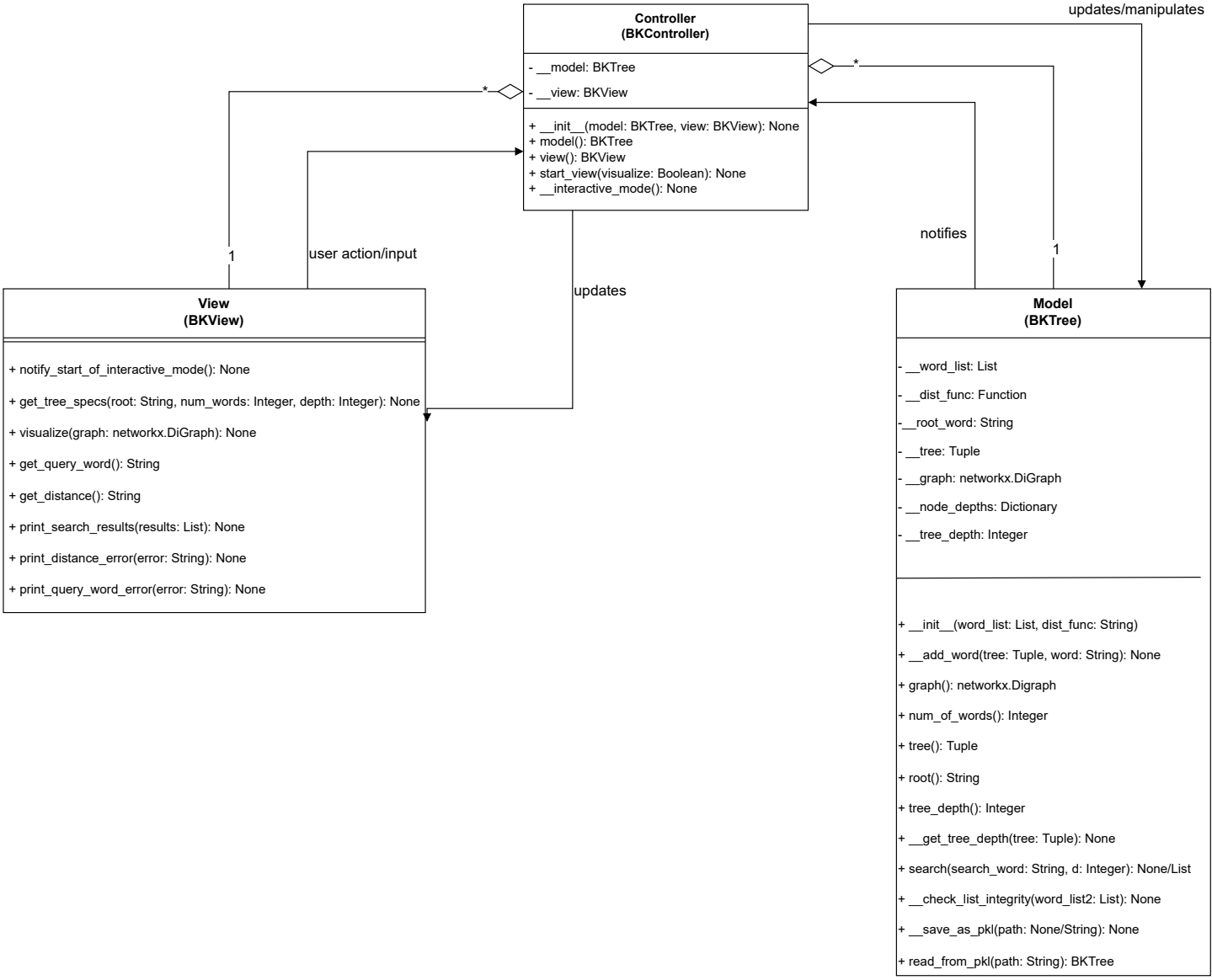

Abbildung 1: UML-Diagramm der MVC-Klassenarchitektur

Der Controller stellt ein Aggregat aus genau einer View und einem Model dar, dessen Lebensdauer abhängig von derer seiner einzelnen Teile ist. Model und View können auch unabhängig vom Controller existieren, wobei dies bei der View nicht sehr sinnhaft ist, weil sie speziell auf Input-Daten des Models ausgerichtet ist. Das Model hingegen könnte autark beziehungsweise in gänzlich neuen (MVC) Kontexten verwendet werden. View und Model können in keinem oder beliebig vielen BKControllern existieren.

Wie erkenntlich ist, besteht keine direkte Verbindung zwischen View und Model. Die Programmlogik des interaktiven Modus, in welchem der Nutzer ein Suchwort und eine maximale Distanz zu Wörtern im Baum eingeben kann, ist in der Controller Klasse definiert. Dabei mittelt der Controller zwischen dem Model und View: Die View nimmt den User-Input (d.h. Suchwort und Distanz mit `get_query_word und get_distance`) entgegen, welcher vom Controller an das Model weitergeleitet wird, sofern er valide ist. Ist dies unzutreffend, wird die View aktualisiert und der Nutzer gebeten, korrekte Eingaben für die Suchfunktionalität bereitzustellen.

Bei validem Input und erfolgreicher Vermittlung wird das Suchergebnis über denselben Weg (Model → Controller → View) zurück übertragen und durch die View im CLI dargestellt.

2.3 Funktionalitäten und Datenstrukturen: Überblick

2.3.1 BKTree Klasse (Model)

Die BKTree Klasse baut bei der Instantiierung eines Objektes mit einer gegebenen Wortliste und Stringmetrik eine rekursive 2-Tupel-Repräsentation des Burkhard-Keller-Baumes auf und speichert diese als privates Objektattribut `self.__tree` ab, um es vor Modifikationen von außen abzukapseln. Parallel dazu wird auch ein plottbarer und in .dot umwandelbarer gerichteter Graph mittels networkx erstellt (`self.__graph`). Das erste Element des Tupels repräsentiert dabei einen mit einem Wort etikettierten (Wurzel-)Knoten, wohingegen das zweite Element ein Dictionary darstellt, dessen keys die Abstände in der gegebenen Stringmetrikeinheit vom Mutterknoten zu den Kindsknoten/Wörtern (values) widerspiegeln. Die Kindsknoten sind wieder rekursive 2-Tupel derselben Struktur. Auf dieser Grundlage wurden die rekursiven Funktionalitäten zur Baumdurchsuchung und zur Bestimmung der Baumtiefe umgesetzt. Für eine Wortliste, bspw. `[help, hell, loop, helps, troop, shell, helper]`, sieht die Tupel-Repräsentation wie folgt aus (Listenreihenfolge unbeachtet):

```
('help', {2: ('shell', {4: ('helper', {})}), 1: ('hell', {2: ('helps', {})}), 4: ('troop', {}), 3: ('loop', {})})
```

Graphisch visualisiert durch networkx/matplotlib::

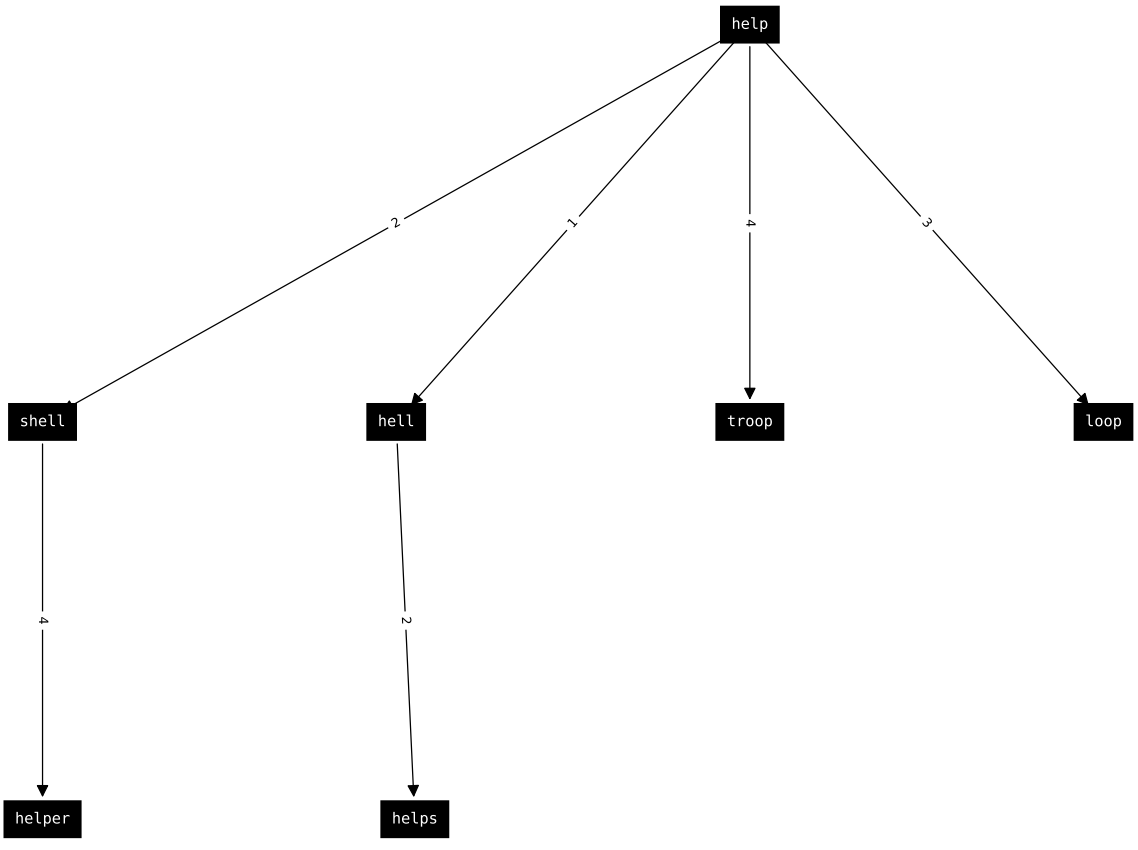

Abbildung 2: Graphische Visualisierung des BK-Baums (Generiert im Model, Präsentation durch View)

2.3.2 BKView Klasse

Die BKView Klasse benachrichtigt den Benutzer über den Start des interaktiven Modus und präsentiert die Spezifikationen des BK-Baumes (Wort des Wurzelknotens, Anzahl der Knoten/Wörter, maximale Baumtiefe). Außerdem nimmt sie den User-Input entgegen, verarbeitet diesen jedoch nicht, i.e. er wird weder umgewandelt (die gegebene Distanz zur Baumdurchsuchung muss beispielsweise in einen Integer umgewandelt werden) noch auf seine Validität geprüft - dies geschieht im Controller/Model. Für die graphische Visualisierung des DiGraph-Objektes (siehe 2) (`self.__graph`) legt sie das Layout des Plots fest und präsentiert diesen in einem separat öffnenden Matplotlib interactive plot Fenster, falls durch den Benutzer erwünscht.

2.3.3 BKController Klasse

Als Schnittstelle zwischen Model und View ist der Controller für den Datenaustausch zwischen den beiden Komponenten verantwortlich. Er startet die Funktionalitäten der View und liefert ihr die zu darstellenden Daten des Models. Falls der Benutzer über die Kommandozeile im Hauptprogramm `main.py` anzeigt, dass eine graphische Visualisierung erwünscht ist und der Baum 5000 oder weniger Knoten besitzt, initialisiert der Controller diese, indem er das DiGraph-Attribut (`self.__graph`) des Models an die View weiterleitet. Außerdem initialisiert und verwaltet der Controller den interaktiven Modus des Programmes, in welchem der Benutzer ein Suchwort und maximale Distanz eingibt, die zur approximativen Suche von ähnlichen Wörtern benötigt werden. Die Logik des Modus wird durch eine zentrale while-Schleife gesteuert, die die Validität des Benutzer-Inputs sicherstellt und dem User durch die View aufzeigt beziehungsweise um Verbesserung bittet, falls dies nicht der Fall ist.

3. Reflexion

3.1 Arbeitsprozess

Der Arbeitsprozess startete mit der Sichtung und Einarbeitung in das bereitgestellte Arbeitsmaterial zu den Burkhard-Keller-Bäumen. Nach der Festlegung der Softwarearchitektur wurde mit der Implementierung des Models begonnen, wobei zuerst die Metriken zur Abstandsberechnungen zwischen zwei Strings eingebaut wurden. Unglücklicherweise implementierte ich neben der Levenshtein-Distanz zunächst die jaro similarity, bei der ich erst nach einer äußerst langen Implementations- und Testdauer - es gab für sie keinen Pseudocode - realisierte, dass sie das Axiom der Dreiecksungleichheit nicht erfüllt und folglich nicht als Metrik für die BK-Bäume geeignet ist. Daraufhin wurde auf die LSC-Distanz ausgewichen, die auf der längsten gemeinsamen Teilsequenz zweier Strings beruht und alle erforderlichen Axiome für die Datenstruktur erfüllt.

Schwierigkeiten bereiteten insbesondere, aufgrund ihrer Abstraktion und Datenstrukturkomplexität der rekursiven 2-Tupel-Repräsentation des Baumes, die rekursiven Funktionalitäten zum Baumaufbau, Baumdurchsuchung und Bestimmung der maximalen Baumtiefe. Wertvolle Abhilfe verschaffte dabei die Benutzung der Debugging-Funktionen der IDE sowie das händische Schritt-für-Schritt-Durchrechnen und Vergegenwärtigen von einigen Beispielen. Da `networkx` keine direkte Funktionalität zur Bestimmung der Baumtiefe bereitstellte, habe ich diese ausgehend von der Tupel-Repräsentation implementiert und dabei die Tiefe eines jeden Knotens in dem Objektattribut `self.__node_depths` abgespeichert. Die generierten Informationen zu den Knotentiefen könnten in etwaigen Erweiterungen benutzt werden, beispielsweise zur Einschränkung der Suchtiefe.

Nachdem das Modell fertiggestellt war, bereitete die Implementierung der View und des Controllers keine signifikanten Schwierigkeiten mehr. Dabei stellten sich vielmehr konzeptionelle Fragen, beispielsweise in welcher Komponente der Benutzer-Input für die Suchfunktion übermittelt werden soll. In Applikationen mit GUI wird dieser in der View empfangen, weshalb ich mich dazu entschied, dem gleichzutun. Die View nimmt die Eingaben jedoch nur entgegen, notwendige Umwandlungen, preprocessing und Validitätsprüfungen übernimmt dabei der Controller.

3.2 Review

Das erhaltene Review hat Stärken, Schwächen, und etwaige Fehler und Unstimmigkeiten des Projektes aufgezeigt, für das man aus subjektiver Perspektive oftmals etwas blind ist. Durch das Feedback konnten jedoch einige Aspekte verbessert beziehungsweise strukturierter gelöst werden, andere Anmerkungen wiederum wurden nicht übernommen:

1. *Stringmetriken*: Aufgrund des numba decorators, der die Funktionen der Abstandsberechnungen kompiliert und diese dadurch erheblich verschnellert, wenn diese sehr oft aufgerufen werden müssen (i.e. bei sehr großen Wortlisten), wurde von einem objektorientierten

Ansatz abgesehen. Mit ihm hätte sich jedoch die Abfrage der Stringmetrikvalidität im Model vereinfacht (mit `getattr()`). Zugunsten der Performance wurde dagegen entschieden. Weiterhin muss für jede neu implementierte Stringmetrik ein neues Dictionary-Element in `all_metrics` hinzugefügt werden, bei der Abfrage innerhalb BKTrees muss jedoch nichts verändert werden. Durch den Hinweis der Reviewerin wurden in den Algorithmen zur Abstandsberechnung die Schleifen mit `range()`-Funktion mit `np.arange` und `nd.nditer` ersetzt, da sie aufgrund der C-Implementierung von `numpy` inhärent schneller als die Python-Funktionalitäten sind.

2. *Graphische Visualisierung*: Durch die Reviewerin wurde darauf hingewiesen, dass das Plot-Fenster auch parallel zum interaktiven Modus geöffnet bleiben kann und nicht wie zuvor geschlossen werden muss, damit der interaktive Modus erst gestartet wird. Zugunsten des Programmlaufes wurde dies übernommen. Außerdem wurde die Demo-Wortliste auf 30 Wörter verkleinert, um den Baum schöner darstellen zu können, die Visualisierung sollte jedoch im Vollbild betrachtet werden. Bei Knoten-Überlappungen kann jedoch weiterhin die Zoom-Funktion verwendet werden. Entgegen der Kommentare der Reviewerin blieben die Funktionalitäten zum Plotting in der View und auch die Daten des Graphs und seine Instantiierung weiterhin im Model, um dies strikt voneinander zu trennen.

3. *Hardcodierung der Levenshtein-Metrik*: Die `__init__` Methode der BKTree Klasse instanziierte Objekte standardmäßig mit der Levenshtein-Metrik, dessen Stringrepräsentation fest im Konstruktor codiert war. Diese Hardcodierung wurde entfernt. Das Hauptprogramm `main.py` respektive Kommandozeilenschnittstelle nutzt bereits standardmäßig die Levenshtein-Distanz bei der Erstellung eines BKTree Objektes.

4. *Baumaufbau*: Der Konstruktor der BKTree Klasse erstellte eine redundante Kopie der übergebenen Wortliste, um in ihr das Wort des Wurzelknotens zu entfernen, damit dieses nicht doppelt im Baum erscheint. Dieser Vorgang wurde mittels einer List-Comprehension vereinfacht.

5. *Generische Print-Methode in der View*: Anstelle einer einzigen Print-Methode zur Daten- und Statusausgabe wurden spezifische Ausgabemethoden in der View implementiert, um jeder Methode nur eine Aufgabe zuzuweisen. Die Aufsplittung erleichtert das Debuggen, insbesondere bei etwaigen Erweiterungen.

6. *Controller-Instantiierung*: Im Hauptprogramm wurde zunächst entgegen der MVC-Logik eine View-Instanz erstellt, um von ihr aus den Benutzer über die Erstellung des BK-Baumes zu benachrichtigen. Dabei agierte sie noch autark, ohne in einem Controller eingebettet zu sein. Die Funktionalität wurde entfernt und von einer Funktion im Hauptprogramm übernommen. Model und View arbeiten nun nur auf Befehl des Controllers.

7. *Einheitlichkeit*: Die BkTree Klasse wurde in BKTree umbenannt und auch alle Erwähungen ihrer angepasst. Damit ist sie namentlich konform mit den anderen beiden MVC-Komponenten (BKView und BKController).