Sandra Sánchez Páez
Matrikelnr. 807996
30.08.2022
SoSe 2022 PRO II
Modulprojekt

Ein BK-Baum oder Burkhard-Keller-Baum ist eine spezielle Form einer Datenstruktur, die häufig zur Durchführung von Rechtschreibprüfungen auf der Grundlage des Levenshtein-Abstands verwendet wird. Er wird auch für Autokorrekturfunktionen verwendet. In diesem Projekt wird ein Programm, das die BK-Baumstruktur verwendet, aus einer vom Benutzer auf der Befehlszeile angegebenen Wortliste erstellt, aber hier kann auch eine andere String-Metrik als Grundlage für die Baumberechnungen verwendet werden. Das ist die Damerau-Levenshtein-Distanz. Dieser Parameter kann vom Benutzer beim Ausführen des Programms angegeben werden. Ich habe mich für diese zusätzliche Metrik entschieden, weil sie sich nur geringfügig unterscheidet, so dass ich dachte, sie würde dem Programm mehr Einheitlichkeit verleihen und wäre einfacher zu implementieren.

Nach der Erstellung des Baums wird dieser automatisch gespeichert und kann in späteren Durchläufen des Programms verwendet werden, um Zeit und Ressourcen zu sparen. Am Ende dieser ersten Phase wird dem Benutzer der Baumstatus (einschließlich der Anzahl der Blätter und des längsten Pfades von der Wurzel zum Blatt) angezeigt.

In der zweiten Phase wird das BK-Baum-Objekt in einen Graphen umgewandelt. Sobald dies geschehen ist, wird es gespeichert und gezeichnet, so dass der Benutzer die Baumstruktur auf dem Bildschirm sehen kann. Diese Funktion wird nur für kleine Bäume empfohlen, da sie sonst sehr schwer zu visualisieren wäre. Der Benutzer kann wählen, ob die Bäume visualisiert werden sollen oder nicht, indem er ein Flag in der Befehlszeile setzt.

Im letzten Schritt geht das Programm in einen interaktiven Modus über und bietet dem Benutzer die Möglichkeit, nach den ähnlichsten Wörtern aus der Wortliste, die zur Erstellung des Baums verwendet wurde, zu einem anderen Wort zu suchen, wenn er die gewünschte Distanzschwelle eingibt. Der Benutzer kann das Programm verlassen, indem er einfach auf "Enter" drückt. Diese drei Hauptfunktionen, die den drei Stufen entsprechen, bilden die main()-Funktion.

Die Berechnung der Levenshtein-Distanz war der Ausgangspunkt des Projekts, denn sie ist entscheidend für den Aufbau aller anderen Funktionen. Wenn man nur eine kleine Python-Liste von Wörtern (Strings) verwendet, schien es zunächst ausreichend zu sein, eine Funktion zu schreiben, die den rekursiven Ansatz des Slicing verwendet. Aber bei der Arbeit mit echten Daten (der langen, vollständigen nltk-Liste) wurde mir klar, dass ein effizienterer Ansatz erforderlich war. Dazu später mehr.

Für diesen Teil verwendete ich testgetriebene Entwicklung, da es keinen Sinn machte, weiterzumachen, bevor nicht klar war, dass die Abstände zwischen den Wörtern richtig berechnet wurden. Später habe ich Tests für alle Hauptfunktionen implementiert. Sie sind alle in der gleichen Datei zu finden.

Schon in einem sehr frühen Stadium verstand ich, welche Klassen ich konzeptionell erstellen musste. Es erschien mir sinnvoll, eine BK Tree-Klasse zu haben, sobald der Baum aufgebaut war, und dann eine Graph-Klasse. Ich zögerte, was ich mit anderen Funktionen machen sollte, und dachte an eine String-Klasse, entschied mich dann aber für eine File-Klasse, deren Instanz erzeugt wird, sobald der Benutzer den Namen der Wortlistendatei eingibt. Alle Klassen befanden sich zunächst zusammen in einer Datei, aber ich habe sie später in Dateien organisiert, was logischer aussieht, besonders wenn die Gesamtlänge der Zeilen zunimmt.

Die Klasse File lädt ein Vokabular (wandelt eine Zeichenkette, die aus durch Kommas oder Leerzeichen getrennten Wörtern besteht, in eine Liste von Zeichenketten um) und baut hauptsächlich einen BK Tree aus der Datei auf. Um die Klasse zu instanziieren, benötigen wir nur die Wortliste und einen Namen, der später zum Speichern des Baums verwendet wird.

Die Klasse BKTree benötigt nur eine Wortliste, um das Baumobjekt zu erzeugen. Dieser Ansatz mag fragwürdig sein, da ich eine Wortliste als Baum bezeichne, obwohl der eigentliche Baum noch nicht erstellt wurde (wie in der Rezension erwähnt), aber es erschien mir sinnvoll, das Objekt zu erstellen, sobald ich eine Wortliste habe, denn das ist zusammen mit der Levenshtein-Funktion das Einzige, was ich extern brauche, um den Baum zu erstellen. Ich habe dieses Konzept während des gesamten Projekts beibehalten (siehe z. B. die Klasse Graph).

Das Wichtigste, was ein BKTree tun sollte, ist, sich selbst zu bauen, und so habe ich mit dem Aufbau der Klasse begonnen. Ich schrieb eine Methode insert_word() zum Hinzufügen von Blättern zum Baum und eine Methode build_tree(), die diese Methode in einer for-Schleife aufruft. Der erzeugte Baum (self.tree) ist ein Tupel mit einer Wurzel auf der linken Seite (ein Wort) und einem Wörterbuch der Kinder und ihrer Abstände zur Wurzel auf der rechten Seite. Diese Struktur ist von Anfang an im Konstruktor der Klasse klar.

In diesem frühen Stadium habe ich nur mit der Demo-Wortliste gearbeitet, die acht Wörter hat, aber ich habe später mit dem Paket tqdm einen Fortschrittsbalken implementiert, mit dem der Benutzer den Fortschritt der Baumbildung für größere Listen sehen kann.

Die search_word()-Methode war ein weiterer entscheidender Punkt, da sie im Grunde ein Drittel der Funktionalität des Programms ausmacht. Es war mir klar, dass ich eine Liste von passenden Wörtern zurückgeben musste, also hatte ich den Rückgabetyp. Da ich keine Node-Klasse erstellt habe, war es sinnvoll, innerhalb der Methode eine Suchfunktion zu schreiben, die alle übereinstimmenden Wörter innerhalb eines bestimmten (vom Benutzer eingegebenen) Abstands sucht.

Später habe ich Methoden wie save_tree() und load_tree() implementiert, die es dem Programm ermöglichen, dank des Pickle-Moduls große Strukturen zu speichern und auf diese Weise Zeit und Ressourcen zu sparen.

Die Methode interactive_mode_search_word() ruft search_word auf und ist in eine while-Schleife in main() eingeschlossen, die sicherstellt, dass der Benutzer so viele Abfragen eingeben kann, wie er möchte. Der interessanteste Teil davon ist, wie mögliche Eingabefehler behandelt werden. Dafür habe ich ein paar Ausnahmeklassen erstellt, die in exception.py zusammengefasst sind. Diese stellt die letzte von drei Stufen des Programms dar.

Der Zwischenschritt, die Methode make_graph_from_tree(), übergibt das BKTree-Objekt an die Graph-Klasse, die die Informationen des Baums (mit Knoten, Kanten und den Abständen, die sie zusammenhalten) neu formatiert und auf Wunsch des Benutzers auch den Graphen speichert und zeichnet.

Ich denke, der schwierigste Teil für mich, wenn es um die Architektur des Projekts geht, war/ist die Entscheidung, was ich mit den Funktionen zur Berechnung der String-Metriken machen soll. Ich war mir die ganze Zeit darüber im Klaren, dass das Programm objektorientiert sein muss, weshalb ich mich dafür entschieden habe, statische Methoden statt normaler Funktionen zu verwenden (aber ich frage mich, ob das in diesem Fall in Ordnung ist). Eine andere Möglichkeit, die Verwendung von Klassen beizubehalten, wäre gewesen, eine Klasse für jede dieser Funktionen zu erstellen, aber ich bin mir nicht sicher, ob es sinnvoll ist, eine Klasse dafür zu erstellen, mit nur einer Methode und unklaren Attributen.

Die Arbeit mit Graphviz und Netzwerken war einige Tage lang eine Herausforderung, nicht nur die Umformatierung und das Zeichnen des eigentlichen Graphen, sondern vor allem der Teil der Extraktion der Anforderungen, um das Programm fehlerfrei laufen zu lassen. Dafür habe ich mehrere Tester und verschiedene Formeln verwendet. Eine davon war pip freeze requirements.txt, aber dann sah ich, dass nur das, was durch pip installiert worden war, der Anforderungsdatei hinzugefügt wurde. Mir ist auch aufgefallen, dass die Abhängigkeiten von Netzwerken (graphviz und pydot) nicht zur Datei hinzugefügt worden waren. Ich fügte sie nachträglich hinzu und bearbeitete auch den Code, um dem Benutzer eine informativere Meldung auf dem Bildschirm anzuzeigen. Es dauerte ein paar Tage, bis ich erkannte, dass es sich nicht um einen FileNotFound-Fehler handelte, sondern um ein fehlendes Paket.

Fast am Ende des Projekts, als ich eine Peer-Review erhielt, wurde mir klar, dass ich nicht richtig verstanden hatte, was damit gemeint war, die Verwendung von mindestens einer weiteren Metrik zuzulassen. Bis dahin hatte ich eine Funktion zur Berechnung der Hamming-Distanz und eine weitere für Damerau Levenshtein geschrieben, die ich beide in Beispielen zeigte. Dann verstand ich, dass wir den BK-Baum mit dieser Metrik und nicht mit der Standard-Levenshtein-Metrik erstellen mussten. Dies war jedoch kein Problem, und ich konnte es leicht beheben. Schließlich löschte ich die gesamte Methode calculate_hamming_distance(), denn es war sinnlos, ungenutzten Code im Skript zu haben. Ich behielt jedoch die Idee bei, Beispiele für jede der String-Metriken auszudrucken.

Um auf die Abstandsberechnungen zurückzukommen, entschied ich mich für einen dynamischen Ansatz. Also habe ich eine Matrix implementiert, die die Zeichen in beiden Zeichenketten darstellt. Wie zuvor ist der Abstand für die Kosten der Operation 0, wenn die Zeichen übereinstimmen. Andernfalls erhalten wir die minimale Zahl von drei möglichen Operationen: Einfügung, Löschung oder Ersetzung. Das Gleiche gilt für die Berechnung der Damerau-Levenshtein-Distanz, wenn man zu diesen Operationen die Transposition hinzufügt.

Ich konnte die Effizienz dieser neuen Methode bei der Arbeit mit einer großen Liste testen. Obwohl ich anfangs etwas unsicher war, was "Liste der korrekten Wörter" bedeutet, habe ich das nltk-Korpus genommen und importiert, was eigentlich "Wörter" genannt wird. Diese Liste enthält 235892 Wörter. Ich rief diesen Teil bis zum Ende des Projekts separat auf, dann rief ich ihn von innen heraus auf. Die Wörter werden in einer Textdatei gespeichert und können anschließend sicher als Liste für Zeichenketten gelesen werden.

Eine Kleinigkeit, die ich noch hinzufügen möchte, ist, dass es offenbar zwei gültige Definitionen für die Höhe eines Baumes gibt. Die eine zählt die Kanten und die andere die Gesamtzahl der Knoten (also Kanten + 1). Ich habe mich für die letztere entschieden und bin der Meinung, dass es keinen signifikanten Unterschied gibt.

Die letzte wirkliche Herausforderung, auf die ich beim Schreiben des Programms gestoßen bin, war die Feststellung, dass es sich erheblich verlangsamt hatte (von 5 Minuten auf drei Stunden mit der großen Liste). Das ist eines der vielen Dinge, bei denen mir die Peer Review geholfen hat. Um dieses Problem zu lösen, verwendete ich das Tool git bisect, das ich vorher nicht kannte. Es half mir, meinen Commit-Verlauf durchzugehen und die Leistung des Programms bei verschiedenen Commits zu testen. So stellte ich fest, dass ich beim Speichern der Baumdatei in der save_tree()-Methode einen Einrückungsfehler gemacht hatte, so dass für jedes Wort, das dem Baum hinzugefügt wurde, eine Datei gespeichert wurde.

Das folgende UML-Diagramm zeigt alle im Programm verwendeten Klassen und Methoden. Es gibt die Hauptklassen (die voneinander abhängig sind, da ein BKTree aus einer Datei und ein Graph aus einem BKTree besteht), die Fehlerklassen und die Testklassen. Alle Klassen sind nicht miteinander verwandt (es gibt keine Vererbung) und es kann maximal eine Instanz von jeder der Hauptklassen geben.

Schließlich sind die Links zu den Bewertungen, die ich für zwei Klassenkameraden gemacht habe, die folgenden:

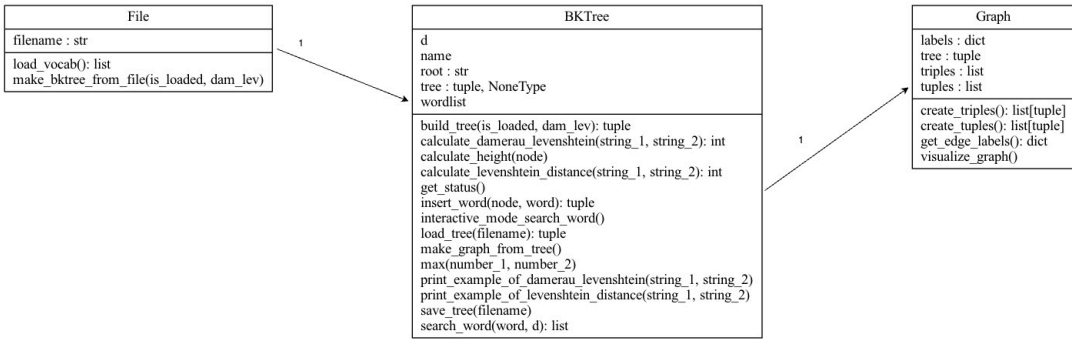
https://gitup.uni-potsdam.de/rasi/coco_pro2_modulprojekt/-/merge_requests/1

https://gitup.uni-potsdam.de/svezhentseva/pro2-bk-tree/-/merge_requests/1

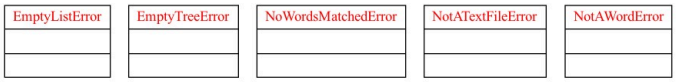
Und dies ist die Bewertung, die ich selbst erstellt habe:

https://gitup.uni-potsdam.de/sanchezpaez/pro2_modulprojekt_sanchezpaez/-/merge_requests/2

MAIN CLASSES



ERROR CLASSES



TEST CLASSES