Universität Potsdam
Programmierung I
Dozentin:
Jana Götze
Wintersemester 2020/21

Programmteile und Reflektion

String Matching

Name: Katja Konermann
Martikelnummer: 802658
Email: katja.konermann@uni-potsdam.de

Inhaltsverzeichnis

1 Überblick

Für das Projekt habe ich den *Aho-Corasick*-Algorithmus gewählt. Ich habe mich dabei an dem Paper von 1975 orientiert, in dem insbesondere die dort beschriebenen Algorithmen 1, 2 und 3 relevant waren.
Die drei Teile meines Programms werden im Folgenden genauer vorgestellt.

1.1 String Matching Algorithmen

Die Datei *string_matching.py* enthält zum einen die Klasse *NaiveMatching*, die den naiven Algorithmus darstellt. Sie ist relativ kurz gehalten und erhält bei ihrer Instanzierung eine iterierbares Objekt, das ausschließlich Strings enthält, die zudem nicht leer sind. Ist das nicht der Fall, wird ein Fehler geworfen. Ansonsten besitzt die Klasse nur eine Methode *match_pattern*, die einen Eingabetext als Argument erhält und mithilfe des naiven Algorithmus (wie in QUELLE beschrieben) die Startindices der Strings ermittelt, die der Klasse bei der Instanzierung übergeben wurden.
Zum anderen enthält die Datei die Klasse *AhoCorasickMatching*. Diese erbt von der *NaiveMatching* Klasse und damit auch die Bedingungen, die an die Schlüsselwörter gestellt werden. Die *AhoCorasick* Klasse besitzt mehr Attribute und Methoden, die der Vorverarbeitung dienen und als privat gekennzeichnet sind, weil sie für einen Benutzer keine große Bedeutung haben dürften.
Bei der Vorverarbeitung werden in der Methode *construct_functions* die drei wichtigen Funktionen erstellt: Die *goto* Funktion weist einem Zustand und einem Zeichen einen neuen Zustand zu. Die *fail* Funktion weist einem Zustand einen neuen Zustand zu, wenn ein Übergang durch die *goto* Funktion scheitert. Die *output* Funktion enthält für einen Zustand alle Schlüsselwörter, die an diesem Zustand ausgegeben werden können.
Als Datenstruktur wird für die *fail* und *output* Funktion ein *Dictionary* benutzt, in dem die *Keys* jeweils *Integer* sind, die Zustände darstellen. Die Values für das Wörterbuch der *fail* Funktion sind ebenfalls Integers (Zustände), für das Wörterbuch der *output* sind die Values Mengen, die Schlüsselwörter enthalten.
Auch die *goto* beruht zum Teil auf einem Dictionary. Sie besitzte aber einige spezielle Eigenschaften (siehe Pseudocode), sodass sie in ihrer finalen Version durch eine Methode *goto* implementiert ist.
Die Klasse *AhoCorasickMatching* besitzt so wie *NaiveMatching* die öffentliche Methode *match_pattern*, die die gleichen Argument erhält und den gleichen Rückgabewert besitzt, aber intern den Algorithmus *pattern matching machine* implementiert.
Eine Besonderheit beider *match_pattern* Methoden, ist der zusätzliche Parameter *start*, ein *Integer*, das zu dem gefundenen Index eines Schlüsselwortes addiert wird. Für den String *"Hello World"* und einen Wert von 1 für *start*, wird für das Schlüsselwort *"Hello"* so dann beispielsweise 1 statt 0 ausgegeben. Das ermöglicht im Weiteren so etwas wie einen Multiline-Modus.

1.2 Programm Modi

In der Datei *user_commands.py* wird durch die Klasse *Search*, die verschiedenen Modi und Funktionalitäten des Programmes implementiert. Bei den Attributen, die bei der Instanzierung übergeben werden, handelt es sich um eine Liste von Strings, die Schlüsselwörter, einen String, der den Eingabetext darstellt und booleansche Parameter, die bestimmen, ob das Programm mit den naiven Algorithmus (*self.naive*), unabhängig von Groß-und Kleinschreibung (*self.insensitive*) oder mit einer ausführlicheren Ausgabe (*self.verbose*) ausgeführt wird.
Das Programm sollte die Möglichkeit besitzen, Eingabetexte aus Textdateien oder Verzeichnissen auszulesen. Um zwischen einem bloßen String und Datei bzw. Verzeichnissen zu unterscheiden,

werden *Properties* genutzt, die testen, ob Strings mit einer validen Dateiendung oder einem (Back-)Slash enden.
Hier hätte man auch andere Vorgehensweisen wählen können, in dem zum Beispiel getestet wird, ob ein String als Datei oder Verzeichnis existiert. Ich habe aber dieses Vorgehen gewählt, weil es meiner Meinung nach die aussagekräftigsten Fehlermeldungen liefert: Wenn ein Benutzer einen String mit der Endung *.txt* als Eingabe gibt, so kann davon ausgegangen werden, dass er eine Datei einlesen will. Wenn diese Datei dann nicht gefunden wird, kann daraufhin eine entsprechende Fehlermeldung ausgegeben werden. Würde man dagegen testen, ob ein String als Datei existiert und das nicht der Fall ist, so würde mein Programm in den Default-Modus zurückfallen und den String nur als solchen behandeln. Es würde also keine Fehlermeldung geben. Es kann allerdings zu Problemen kommen, wenn der Benutzer einen String durchsuchen will, der zufällig mit einer Dateiendung oder einem (Back-)Slash endet. Um darauf aufmerksam zu machen, habe ich einen Abschnitt in die README eingefügt, wo beschrieben wird, wie man das umgehen kann.
Die Klasse *Search* besitzt private Methoden, die entsprechend der Parameter in einem Verzeichnis (*_match_in_dir*), Datei (*_match_in_file*) oder einem String (*_match_in_str*) matchen. In der öffentlich Methode *run* werden diese Methoden dann je nach Inputtyp aufgerufen.
Bei der Verarbeitung von Dateien ist es gefährlich, den ganzen Inhalt mit *read* oder *readlines*auf einmal einzulesen, denn die Datei könnte sehr groß oder Arbeitsspeicher des Nutzers sehr klein sein. Das würde zu einer Fehlermeldung führen. Stattdessen iteriert mein Programm über die Zeilen einer Datei. Dabei gibt es zwei Modi: Im *verbose* Modi wird die Zeile und die Indices in der Zeile der gefundenen Schlüsselwörter ausgegeben. Im Default Modus dagegen werden absoluten Indices der Schlüsselwörter in der Datei ausgegeben. Dafür wird sich mithilfe des *start* Parameters der *match_pattern* Methode aus den String-Matching Klassen der letzte Index der vorherigen Zeile gemerkt. Ich habe mich dazu entschlossen, dabei EOL-Zeichen und Whitespace am Zeilenende mitzuzählen, damit man, falls man den Inhalt der Datei mit *read* einliest, das Schlüsselwort einfach durch Indexierung finden kann.
Bis jetzt arbeitet das Programm nur mit Textdateien. Dateien mit anderen Endungen werden ignoriert. Das könnte man in Zukunft noch ausbauen. Deshalb ruft die Methode *_match_in_file* wiederum eine Methode *_match_txt_file* auf, wenn die Datei auf *txt* endet. Das könnte man durch weitere *elif* statements und Methoden für die entsprechenden Dateitypen erweitern.
In der *Search* Klasse ist zudem eine Demo Methode enthalten, die als *classmethod* implementiert ist und Beispiele, die in der Klassenvariable *DEMOS* gespeichert sind, ausgibt. Dabei wird genutzt, dass die String Repräsentation der *Search* Klasse die entsprechende Eingabe in der Kommandozeile darstellt. Die Pfade zu den Demo Dateien habe ich hier hart kodiert. Das ist meiner Meinung nach sinnvoll, weil die Demo für einen Benutzer Hilfestellung und Anleitung sein soll und nicht besonders hilfreich ist, wenn der Benutzer auch hier erst selber Pfade angeben muss.

1.3 Benutzerschnittstelle

Die Datei *main.py* sollte zusammen mit den nötigen Kommandozeilenargumenten vom Benutzer aufgerufen werden. Die Argumente werden hier von dem Modul *argparse* gehändelt. Der Benutzer hat die Wahl zwischen zwei Bereichen: Mit dem Aufruf von *main.py demo* erhält er die Demo mit Beispielaufrufen. Mit dem Aufruf von *main.py search* plus dem Eingabetext und Suchwörtern wird das Hauptprogramm ausgeführt. Ich habe hier die Reihenfolge der Argumente, wie sie in der Aufgabenstellung beschrieben war, abgeändert. Statt den Suchwörter folgt zuerst der Eingabetext, erst dann folgt der Eingabetext. Diese Änderung machte es einfacher, mehrere Suchwörter zu definieren, denn der Parameter *nargs*, mit dem die Anzahl der Argumente in *argparse* bestimmt, verhält sich *greedy* und würde, wenn man Eingabetext und Suchwörter tauscht,

das Argument Eingabetext als Suchwort zählen.
Mithilfe von *argparse* werden auch Hilfebeschreibungen gestellt. In *main.py* werden durch *try-except* Blöcke Fehler abgefangen, die etwa im Zusammenhang mit Datei und Verzeichnissen (*OSError*) und ungültigen Eingaben (*ValueError*) auftreten können. Es bietet sich hier an, *try-except* zu nutzen, da im Vorhinein nur schwer vorhergesagt werden kann, was für Eingaben gemacht werden. So kann man etwa testen, ob eine Datei existiert, aber das heißt nicht, dass man sie auch lesen kann oder Zugriff auf sie hat. Genauso könnte ein Kodierungsfehler auftreten. Mit *try-except* können diese Fehler leicht behandelt werden. Hier heißt das, dass die Meldung des Fehler ausgegeben und das Programm beendet wird.

2 Pseudocode

3 Erweiterungen

Das Programm könnte man auf verschiedene Weisen erweitern. Zum einen könnten weitere Matching Algorithmen implementiert werden, die dann über einen zusätzlichen optional Parameter in der Kommandozeile aufgerufen werden könnte. Dazu müsste eine weitere Klasse in auf Basis der *NaiveMatching* Klasse geschrieben und ein *elif* Statement in die Methode *_create_match* der *Search* Klasse eingefügt werden.
Außerdem könnten weitere Methoden zu Einlesen verschiedener Dateitypen geschrieben werden. Die entsprechenden Dateiendungen müssten dazu in die Klassenvariable *INPUT_EXT* eingefügt werden, eine Methode geschrieben werden und ein zusätzliches *elif* statement in die Methode *_match_in_file* eingefügt werden.
Um andere Ausgaben zu erhalten, müsste man die Methode *_print_matches* ändern. Diese Methode gibt die gefundenen Matches schön und gut lesbar aus.
Zusätzlich könnte man - wie schon beim Eingabetext möglich - auch die Suchwörter aus einer Datei auslesen. Dazu würde man wahrscheinlich die *_create_match* erweitern, sodass die die Endung der gegebenen Strings überprüft und gegebenenfalls eine Methode aufruft, die Schlüsselwörter aus einem bestimmten Dateityp ausliest.

4 Reflektion

Als ich an dem Projekt gearbeitet habe, sah mein Arbeitsablauf meistens folgendermaßen aus: Zunächst habe ich mir den Code, den ich bis dahin geschrieben hatte, noch einmal angeguckt und gegebenenfalls ein wenig überarbeitet. Dann habe ich mir einen neuen Teil des Projekts vorgenommen. Zwischendurch habe ich den Code dann getestet, um zu überprüfen, ob er richtig funktioniert. Angefangen habe ich mit den String Matching Algorithmen, dann habe ich die *Search* Klasse geschrieben. Zum Schluss habe ich dann die Benutzerschnittstelle in *main.py* erstellt.
Als ich mir die Aufgabenbeschreibungen und Anforderungen durchgelesen habe, dachte ich, dass die Implementierung des intelligenten Algorithmus der schwierigste Teil des Projektes wird. Damit bin ich aber ziemlich schnell und problemlos durchgekommen. Die meiste Zeit habe ich dagegen in die Gestaltung der Benutzerstelle und die Implementierung der *Search* Klasse gesteckt. Ich glaube, dass lag vor allem daran, dass ich mich bei den String-Matching Algorithmen an Pseudocode entlang hangeln konnte, während die anderen Teile freier waren. Ich habe mir beispielsweise viele Gedanken um die Ausgabe von Matches in Dateien gemacht und das immer wieder geändert.

Mir war außerdem von Anfang an klar, dass ich dem User die Möglichkeit geben wollte, nach mehreren Schlüsselwörtern zu suchen, denn darin liegt die Stärke des Aho-Corasick Algorithmus und rechtfertigt seine nötige Vorverarbeitungszeit. Ich habe dann viele verschiedene Möglichkeiten ausprobiert, aber ich fand es schließlich am einfachsten, die Argumente Eingabetext und Schlüsselwörter zu vertauschen.
Bei meinen nächsten Programmierprojekte würde ich die Dokumentation noch schreiben, während ich am Code arbeite. Für dieses Projekt habe ich Docstrings und Kommentare erst ziemlich zum Ende eingefügt. Das hat aber sehr viel Zeit in Anspruch genommen, weil es teilweise schon länger her, dass ich den Code geschrieben hatte.