StringMatcher Report

Thomas N. T. Pham

University of Potsdam, Germany
nhpham@uni-potsdam.de
April 2021

1 Aufgabe

Für dieses String-Matching-Kommandozeilen-Tool wurde als String-Search-Algorithmus der Boyer-Moore-Algorithmus (BM) implementiert. Als Literatur wurde die auf Moodle zur Verfügung gestellte Version von Cormen et al. (1990) genutzt.

2 Programmaufbau

Das Programm besteht aus `stringmatcher.py`, das die `StringMatcher`-Klasse und eine Demo enthält, einer Datei `errors.py` mit einer Fehlerklasse und zusätzlich `main.py`, das die Interaktion mit der Kommandozeile konfiguriert. Im Ordner *testdata* sind zudem txt-Dateien beigefügt um die String-Suche zu testen.

2.1 `class StringMatcher`

In dieser Klasse sind der naive Suchalgorithmus `naive()` und der BM `boyer_moore()` implementiert.

Ein Objekt dieser Klasse wird mit einem Suchpattern initialisiert. Das bedeutet, dass man für verschiedene Suchpatterns, verschiedene Objekte erstellen muss. Ebenso muss man sich von vornherein entscheiden, ob man Groß- und Kleinschreibung beachten oder ignorieren möchte; per Default gilt ersteres. Das ist notwendig, da BM den Suchstring vorverarbeiten muss, um eine *Bad-Character-Heuristic* und eine *Good-Suffix-Heuristic* zu erstellen. Diese werden als Instanzattribute gespeichert, worauf während der Suche zugegriffen wird, da sie Informationen darüber enthalten, wieviele Shifts vorgenommen werden können, ohne dass ein möglicher Fund übersprungen wird.

Alternativ hätte man auch erst bei der eigentlichen Suche, die Heuristiken berechnen können, dann müsste man nicht für jede Suche mit einem anderen Pattern ein neues Klassenobjekt erstellen. Der Nachteil daran wäre gewesen, dass diese berechneten Heuristiken nicht weiter genutzt werden könnten für mögliche andere Suchen mit dem gleichen String. Da die Aufgabe zudem darin bestand ein Kommandozeilen-Tool zu implementieren, würde bei jeder Suche ohnehin ein neues Objekt erstellt werden.

`boyer_moore()` Mit dieser Methode ist eine Suche in einem anderen String möglich. Ihre Implementation und die der *Bad-Character-Heuristic* sind zwar an dem Pseudocode orientiert, die Berechnung der *Good-Suffix-Heuristic* allerdings nicht (Abschnitt dazu folgt). Natürlich mussten Änderungen vorgenommen werden, zum Beispiel an Indizes oder das print-Statement ersetzt werden durch ein Return einer Liste von Startindizes (ebenso beim `naive()`).

Bad-Character-Heuristic **(`_rightmost_index_table()`)** Diese Heuristik kommt zum Tragen, wann immer ein Mismatch auftaucht. Da der Text zwar von links nach rechts gelesen wird, das Pattern aber von rechts nach links mit dem Text verglichen wird, kann man bei einer Suche von “hallo”, wenn man auf das Zeichen “x” im Text stößt, das Suchpattern so weit nach rechts verschieben, sodass es mit dem rechtesten Vorkommen im Suchpattern aligniert. In diesem Fall enthält das Suchpattern das Zeichen im Text gar nicht und kann deshalb bis hinter das Zeichen verschoben werden und überspringt somit unnötige Shifts. Anstatt, dass wie im Pseudocode ein Alphabet übergeben muss, werden in dieser Implementation nur die Zeichen im Suchpattern beachtet und zu jedem dieser Zeichen sein rechtestes Index in einem `dict` gespeichert. Beim Zugriff muss deshalb ein Defaultwert (hier `-1`) mitgegeben werden, falls das Mismatch-Zeichen nicht enthalten ist. Hier der Code-Ausschnitt aus der `boyer_moore()`-Methode:

```
100 else: # mismatch at index j
101     shift += max(
102         self._good_suffix_heuristic[j],
103         j - self._bad_char_heuristic.get(text[shift+j], -1)
104         )
```

Der Vorteil daran ist die Einsparung von Speicherplatz, da die Größe der Datenstruktur nicht von der Größe des Alphabets abhängt, und zwar nur von dem Suchpattern, das sowieso nicht mehr verschiedene Zeichen als das Alphabet haben kann.

Mögliche Erweiterung: Da das rechteste Vorkommen (des Mismatch-Zeichens aus dem Text) im Suchpattern auch rechts vom Mismatch sein kann, würde diese Heuristik einen negativen Shift vorschlagen. Man könnte die Heuristik verbessern, indem man den Index des rechtesten Vorkommen, das zusätzlich links vom Mismatch ist, nehmen würde. (Die Literatur hat das auch nicht beachtet in der Beschreibung.)

Good-Suffix-Heuristic (`_good_suffix_shifts()`) Diese Heuristik basiert auf den bereits gematchten Zeichen (= good suffix). Die Implementation ist an der Beschreibung der angegebenen Literatur von Cormen et al. (1990) orientiert, jedoch nicht am Pseudocode. Im Gegensatz zur Literatur werden hier explizit zwei Fälle unterschieden:

1. Das *good suffix* ist in dem Suchpattern an einer anderen Stelle noch einmal vorhanden (siehe `_rightmost_substr_start()`). Dann wird das rechteste Vorkommen mit dem bereits gefundenen *good suffix* aligniert.
2. Sonst: Das *good suffix* hat ein Suffix, das ein Präfix des Suchpatterns ist (siehe `_start_of_longest_sfx_as_pfx`). Diese werden dann aligniert.

Zurückgegeben wird eine `list`, die die Anzahl der möglichen Shifts enthält, wobei die jeweilige Indexposition die Stelle des Mismatches ist. Am Anfang während der Implementierungsphase war dies noch ein `dict`, mit dem Gedanken, dass die Datenstrukturen für die beiden Heuristiken einheitlicher wären. Diese Idee wurde verworfen aus Gründen der Effizienz.

Mögliche Erweiterung: Das *good suffix* im Text matcht zwar mit dem neuen Alignment, man kann sich aber zusätzlich zunutze machen, dass man bereits den Mismatch an dem Zeichen vor dem *good suffix* hatte. Das bedeutet, ein Shift zu einem anderen *good suffix*, das das gleiche vorhergehende Zeichen hat, würde auch zu einem Mismatch führen. (Die Literatur hat das auch nicht beachtet in der Beschreibung.)

2.2 Sonstige mögliche Erweiterungen

– Man könnte in Ordnern auch nach anderen plain Textdateien suchen wie zum Beispiel csv-, log- oder py-Dateien. Man könnte das mit einem Argument spezifizieren wie `end=".py"` oder `end=[".py", ".json"]`. Hier ein Ausschnitt aus der `search_dir()`-Methode:

```
177 for file in tqdm(file_list, desc="search directory...", leave=False):
178     if file.endswith(".txt"):
179         filepath = os.path.join(dir, file)
```

Die Zeile 178 müsste man mit dem `end`-Argument verändern (ggf. for-loop hinzufügen im Falle einer Liste).

– Bei der Ordnersuche könnte man rekursiv auch in Unterordnern nach txt-Dateien suchen.
– Man könnte die gefundenen Positionen in einer Outputdatei ausgeben.
– Man könnte die Sätze, in denen das Suchpattern gefunden wurde mit ausgeben oder in einer separaten Datei. Für diesen Zweck könnte man den Sentence-Tokenizer von `nltk` verwenden. Man könnte die Zeichen in jedem Satz zählen und die Sätze, die die gefundenen Positionen beinhalten, auswählen. Allerdings wäre es effizienter, wenn man beim Durchgehen des Textes, selbst nach Satzenden schaut und den aktuellen Satz schon speichern würde im Falle eines Fundes. Viel einfacher zu implementieren jedoch wäre einfach das Ausgeben der Zeile (falls überhaupt in einer Datei gesucht wurde).
– Weitere Suchalgorithmen könnten ergänzt werden. Anstatt des boolschen Wertes als das bisherige Keyword-Argument von `naive`, könnte man einen String als `algorithm="naive"` übergeben. Oder man hat so viele Keyword-Arguments wie Algorithmen, die alle boolsche Werte sind.
– Als Option könnte man reguläre Ausdrücke hinzufügen.
– Eventuell ist man gar nicht (oder nicht nur) an den Positionen der Vorkommen interessiert, sondern eher an der Anzahl. Zusätzlich zum Sammeln der Shifts in einer Liste, würde man einen Counter mitzählen lassen (oder man wendet einfach `len()` auf die Liste an; das wäre mit einer konstanten Laufzeit effizienter).

– Vielleicht möchte man mehrere Strings gleichzeitig suchen. Dafür würde sich aber ein anderer Algorithmus eher anbieten, z.B. Aho-Corasick. Man könnte defaultmäßig den Aho-Corasick benutzen, wenn mehr als ein Suchstring übergeben werden.

3 Reflektion

3.1 Workflow & Schwierigkeiten

Angefangen habe ich tatsächlich gar nicht mit der Aufgabenbeschreibung, sondern mit dem Lesen der Literatur um den Boyer-Moore-Algorithmus (BM) zu verstehen. Das Konzept erschien mir recht intuitiv und schnell war der Großteil klar. Allerdings hatte ich große Probleme den Abschnitt über die *Good-Suffix-Heuristic* in Cormen et al. (1990) zu verstehen, besonders als Abschätzungen mit der Präfixfunktion gemacht und über eine halbe Seite (p.882) Indizes hergeleitet wurden. Am Ende wurde zwar ein fertiger Pseudocode gezeigt, die Bedeutung dahinter war mir jedoch nicht ganz klar. Leider habe ich erst, nachdem ich bereits (zu) viel Zeit in das Verstehenwollen investiert habe, mich dazu entschieden, die *Good-Suffix-Heuristic* selbst so zu implementieren, wie es mit der Beschreibung im Text passt (auch wenn es nicht lang war) und es für mich auch Sinn macht. Das hat den Vorteil, dass ich Bugs einfacher ausfindig machen und selbstbewusst behaupten kann, dass ich auch alle Suchstrings finde.

Die Hauptarbeit für mich persönlich fand auf dem Blatt Papier statt, und zwar beim Zählen der Startindizes von Suffixen zu diversen Beispielpattern und wie das zusammenhängt mit einer Verschiebung. Ich habe dafür mehrere kleine unabhängige Funktionen geschrieben und auf Korrektheit getestet und sie erst danach in eine Klasse gepackt. Dafür habe ich auch u.a. mehrere 100.000 zufällige Strings generiert, in zufällig generierten Texten suchen lassen und mit dem Ergebnis des naiven Algorithmus verglichen. Als die Funktionalität des BM sichergestellt war, habe ich mir Gedanken über die Programmstruktur gemacht. Dazu habe ich mir überlegt, was der Zweck des Programms ist und mich in Benutzer*innen hineinversetzt. Zuerst musste ich mich entscheiden, ob man einen Suchstring an mehreren Orten suchen möchte, oder lieber an einem Ort mehrere Suchstrings

ausprobieren möchte. Je nach Auswahl hätte man die jeweilige Information im Objekt speichern müssen. Ich habe mich für ersteres entschieden (also ein String an mehreren Orten), damit man mehr von der Vorverarbeitung profitieren kann.

Wenn ich auf Probleme gestoßen bin, gab es entweder informative Fehlermeldungen oder aufschlussreiche Threads in diversen Foren, denn es gibt kaum Probleme, die eine andere Person nicht auch schon hatte (z.B. das Reraisen eines `UnicodeDecodeError` erfordert 5 Argumente, die zwar in der Python-Dokumentation angegeben sind, aber nicht in der richtigen Reihenfolge).

3.2 Was hätte ich nächstes mal anders gemacht?

Ich hätte mir zwischendurch nochmal die Aufgabenbeschreibung detaillierter durchgelesen, damit ich nicht am Ende noch eventuell Programmstrukturen aufbrechen muss um eine Funktionalität zu erfüllen. Zum Beispiel habe ich erst spät gesehen, dass nur txt-Dateien in Ordnern beachtet werden (wobei ich dafür nur eine Zeile hinzufügen musste), oder dass es eine case-insensitive-Option geben soll.

Das größte Problem daran ist aber, wenn man denkt, man wäre schon fertig und die Docstrings geschrieben sind, sowie Demo, main.py und Readme und danach noch Funktionalitäten hinzufügt und an jeder Stelle noch die Beschreibung anpassen muss (bzw. diese Stellen erst suchen).

Bei jedem Projekt unterschätze ich die Zeit, die in das Schreiben der Demo, der Docstrings, der main.py und der Readme fließt. Nächstes Mal mehr Zeit dafür einplanen!

3.3 Was ist gut gelaufen?

Unfertige, aber nicht dringende Codeabschnitte mit "# TODO" markieren, um ein grobes Programmgerüst aufzubauen, hilft.

Ein ToDo-Liste mit Bugs und Erweiterungsideen ist auch hilfreich, weil man nicht jede Idee sofort und gleichzeitig umsetzen kann.

Zum Verständnis des Algorithmus haben auf jeden Fall die Gespräche mit Kommiliton*innen geholfen, denn wenn man die Vorgehensweise erklären kann, kann man sie auch implementieren (Danke Natalie).