Testy parsera plików mmCif

Walidacja działania parsera struktur molekularnych zapisanych w formacie mmCif z biopythona (Bio.PDB.MMCIFParser) na całej bazie danych.

Monika Wiech

I. CEL 3
II. REALIZACJA.....3
1. Użyte narzędzia.....3
2. Użyte technologie 4
3. Prototyp testera parserów mmCif - Tester 0.9 4
4. Tester 1.0 5
6. Pomiar.....6
7. Przygotowanie wyników (regexy, issue_counter).....6
III. WYNIKI 10
1. Czas odczytu względem długości struktury dla obu parserów: 10
2. Czas odczytu względem ilości ostrzeżeń dla obu parserów: 11
3. Ostrzeżenia - FastMMCIFParser 13
4. Błędy 14

I. Cel

- Przepuszczenie wszystkich zdeponowanych w PDB struktur przez klasy MMCIFParser oraz FastMMCIFParser z parametrem QUIET ustawionym na “False”
- Zbadanie czasu potrzebnego na odczyt i stworzenie obiektu struktury i stworzenie zestawienia zależności czasu od rozmiaru struktury oraz wywołanej metody
- Przeanalizowanie i opisanie ostrzeżeń zwracanych przez parser
- Przeanalizowanie błędów struktur i parsera, które uniemożliwiają parserowi poprawne odczytanie danych zawartych w tych strukturach
- Rozwiązanie kompatybilne z Pythonem 3.6 oraz 2.7

II. Realizacja

1. Użyte narzędzia

Dane do parsera

Źródło: Protein Data Bank (www.rcsb.org/pdb). Dane zostały pobrane w postaci spakowanych plików (“Coordinates & Experimental Data”). Pobrano całą zawartość bazy.

Parser

MMCIFParser, FastMMCIFParser (biopython)

Środowisko programistyczne
IDE: PyCharm Community Edition 2016.3
Kontrola wersji: GitHub (https://github.com/monika244/mmcif-parser-tester)

Przygotowanie wyników
Przetwarzanie danych: SublimeText 3 (b3126)
Wizualizacja wyników: DataGraph 4.2.1

2. Użyte technologie

Programowanie: Python 3.6 (oraz Python 2.7 dla sprawdzenia kompatybilności)
Przetwarzanie danych: Regex

3. Prototyp testera parserów mmCif - Tester 0.9

Pierwsza wersja testera parsera mmCif ma za zadanie wykonać pomiar czasu parsowania pojedynczego pliku struktury '100d'. Plik jest rozpakowany i przetwarzany przez parser. Czas operacji jest wypisany do konsoli systemowej. Prototyp służył przede wszystkim do sprawdzenia, jak parser mmCif działa oraz do wstępnego oszacowania, ile może potrwać parsowanie wszystkich struktur w bazie PDB.

```
import gzip
import time
from Bio.PDB.MMCIFParser import MMCIFParser
from Bio.PDB.MMCIFParser import FastMMCIFParser

parser = MMCIFParser()

file_dir = './mmCIF/00/'
str_id = '100d'
gz_file_name = file_dir + str_id + '.cif.gz'
cif_file_name = file_dir + str_id + '.cif'


total_start = time.time()

in_file = gzip.open(gz_file_name, 'rb')
out_file = open(cif_file_name, 'wb')
out_file.write(in_file.read())
in_file.close()
out_file.close()

file = open(cif_file_name)
parse_start = time.time()
structure = parser.get_structure(str_id, file)
parse_stop = time.time()
file.close()

total_stop = time.time()

print('structure:', structure.get_id(), '\tparse time:', parse_stop - parse_start,
      '\ttotal time: ', total_stop - total_start)
```

4. Tester 1.0

Następnym krokiem było obsłużenie parsowania wielu plików w czasie jednej sesji testera. Program w tej wersji potrafi przechodzić iteracyjnie przez wszystkie struktury w zadanym folderze i zapisywać wyniki pomiaru czasu do pliku. To pozwala na wydajne przetwarzanie większej ilości struktur.

```
40      print('parser:', parser.__class__, '\nSTART')
41      # loop through all dirs in root data dir
42      for data_dir in glob.glob('%s/*/' % root_data_dir):
43          if dir_count == limit:
44              break
45
46          dir_parse_start = time.time()
47          print('\nstarting', data_dir, '\n')
48
49          # extract all files
50          for gz_file_name in glob.glob('%s/*.gz' % data_dir):
51              with gzip.open(gz_file_name, 'rb') as in_file:
52                  cif_file_name = gz_file_name[:-3]
53                  with open(cif_file_name, 'wb') as out_file:
54                      out_file.write(in_file.read())
55                  # process extracted file
56                  with open(cif_file_name) as cif_file:
57                      str_id = cif_file_name[len(data_dir):-4]
58                      str_parse_start = time.time()
59                      structure = parser.get_structure(str_id, cif_file)
60                      str_parse_stop = time.time()
61
```

Poza tym, wersja 1.0 zapisuje do plików log o unikalnych nazwach informacje o ostrzeżeniach parsera w kontekście parsowanej struktury. To pozwala na uwzględnienie informacji o błędach w końcowych wynikach projektu. Program w tej wersji jest kompatybilny z Pythonem 2.7 oraz 3.6.

```
23      # create log file
24      log_file_pattern = './tester%s.log'
25      log_file_num = 0
26      while os.path.exists(log_file_pattern % log_file_num):
27          log_file_num += 1
28
29      # redirect output to log file
30      sys.stdout = open(log_file_pattern % log_file_num, 'w')
31
```

6. Pomiar

Pomiar sprawności parsera mmCif został przeprowadzony testerem 1.0 na pythonie w wersji 3.6. Pomiar czasu parsowania wersją normalną trwał ok. 87,5 godzin, a wersją FAST trwał ok. 8,25 godzin. Wynikiem pomiaru jest seria plików tekstowych z zapisaną długością i czasem parsowania dla każdej struktury.

7. Przygotowanie wyników (regexy, issue_counter)

Dane testowe zostały pogrupowane w folderach zorganizowanych wg alfabetu. Dzięki temu wyniki pomiaru są podzielone na alfabetyczne pliki, co pozwoliło uniknąć obrabiania dużych plików tekstowych oraz umożliwiło łatwe powtórzenie pomiaru dla struktur w danym folderze w razie potrzeby.

Tester 2.0 produkuje wyniki w postaci 3 plików wyjściowych dla każdego alfabetycznego folderu:

- tester.log: zawiera wyniki pomiaru czasu parsowania w odniesieniu do struktury oraz do folderu
- tester.out zawiera przekierowaną zawartość konsoli z znacznikami struktur, co pozwala na określenie, których struktur dotyczą poszczególne ostrzeżenia parsera
- tester.err zawiera opisy złapanych wyjątków w odniesieniu do struktury

Wszystkie pliki wyjściowe zawierają znaczniki początku i końca alfabetycznego folderu wraz z nazwą użytej klasy parsera, co ułatwia przetwarzanie danych.

Następnym krokiem było przetworzenie zebranych informacji do postaci plików csv, żeby umożliwić stworzenie wykresów zależności czasu parsowania od rozmiaru i poprawności struktur.

Żeby to osiągnąć, każdy z plików został przetworzony przez serię podmian z użyciem wyrażeń regularnych (Regex). Podmiany zostały wykonane edytorem tekstu Sublime Text 3.

Przetwarzanie plików log oraz err:

1. Usunięcie znaczników początku i końca folderu alfabetycznego oraz zbędnych pustych linii:

ZNAJDŹ	^>>.+?$\n
ZASTĄP	<pusty string>
ZNAJDŹ	\n\n
ZASTĄP	\n

2. Usunięcie opisu kolumn z rzędów danych:

ZNAJDŹ	\s?[!>|].+?:\s
ZASTĄP	,
ZNAJDŹ	^,
ZASTĄP	<pusty string>

celem jest zamiana linii:

```
> STR_ID: 1a01 |STR_LEN: 776 |STR_TIME: 1.114242
```

na linię:

```
1a01,776,1.114242
```

3. Połączenie wszystkich plików w jeden oraz wstawienie opisu kolumn w pierwszym rzędzie:

```
STR_ID, STR_LEN, STR_TIME
```

Przetwarzanie plików out:

1. Usunięcie znaczników początku i końca folderu alfabetycznego oraz zbędnych pustych linii (jak powyżej)

2. Usunięcie znaczników końca struktury:

ZNAJDŹ	^>\sFINISH_STR.+?$\n
ZASTĄP	<pusty string>

3. Uproszczenie danych o ostrzeżeniach:

ZNAJDŹ	/Library.+?PDBConstructionWarning:\sWARNING:\s
ZASTĄP	PDBConstructionWarning:\s
ZNAJDŹ	/Library.+?PDBConstructionWarning:\s
ZASTĄP	PDBConstructionWarning:\s
ZNAJDŹ	\n\n
ZASTĄP	\n
ZNAJDŹ	\n^PDBConstruction.+?$
ZASTĄP	|PDBConstructionWarning
ZNAJDŹ (**pętla**)	^(.+?:\s)(….)(.*)|PDBConstructionWarning(.+?)$
ZASTĄP (**pętla**)	\1\2\3|\2,PDBConstruction

celem jest zamiana sekcji:

```
> START_STR: 1a00
/Library/Frameworks/Python.framework/Versions/3.6/lib/
python3.6/site-packages/Bio/PDB/StructureBuilder.py:84:
PDBConstructionWarning: WARNING: Chain A is discontinuous at
line 4382.
  PDBConstructionWarning)
/Library/Frameworks/Python.framework/Versions/3.6/lib/
python3.6/site-packages/Bio/PDB/StructureBuilder.py:84:
PDBConstructionWarning: WARNING: Chain B is discontinuous at
line 4425.
  PDBConstructionWarning)
```

na linię:

```
> START_STR: 1a00|1a00,PDBConstructionWarning|
1a00,PDBConstructionWarning
```

4. Usunięcie opisu kolumn z rzędów danych:

ZNAJDŹ	>\sSTART_STR:\s….|
ZASTĄP	<pusty string>
ZNAJDŹ	>\sSTART_STR\s(….)
ZASTĄP	\1

5. Rozdzielenie rzędów:

ZNAJDŹ	|
ZASTĄP	\n

celem jest zamiana linii:

```
1a00,PDBConstructionWarning|1a00,PDBConstructionWarning
```

na sekcję:

```
1a00,PDBConstructionWarning
1a00,PDBConstructionWarning
```

6. Połączenie plików i dodanie nazw kolumn:

```
STR_ID, ISSUE
```

Uzupełnienie informacji o ilości ostrzeżeń i błędów w pliku out
W obecnej postaci, plik out zawiera osobne rzędy dla każdego ostrzeżenia.
Do wykresu zależności czasu parsowania od ilości ostrzeżeń potrzebne jest zliczenie ostrzeżeń dla każdej struktury.
Zajmuje się tym program issue_counter. Liczy ilość wystąpień ostrzeżenia dla każdej struktury i tworzy plik, w którym sekcja:

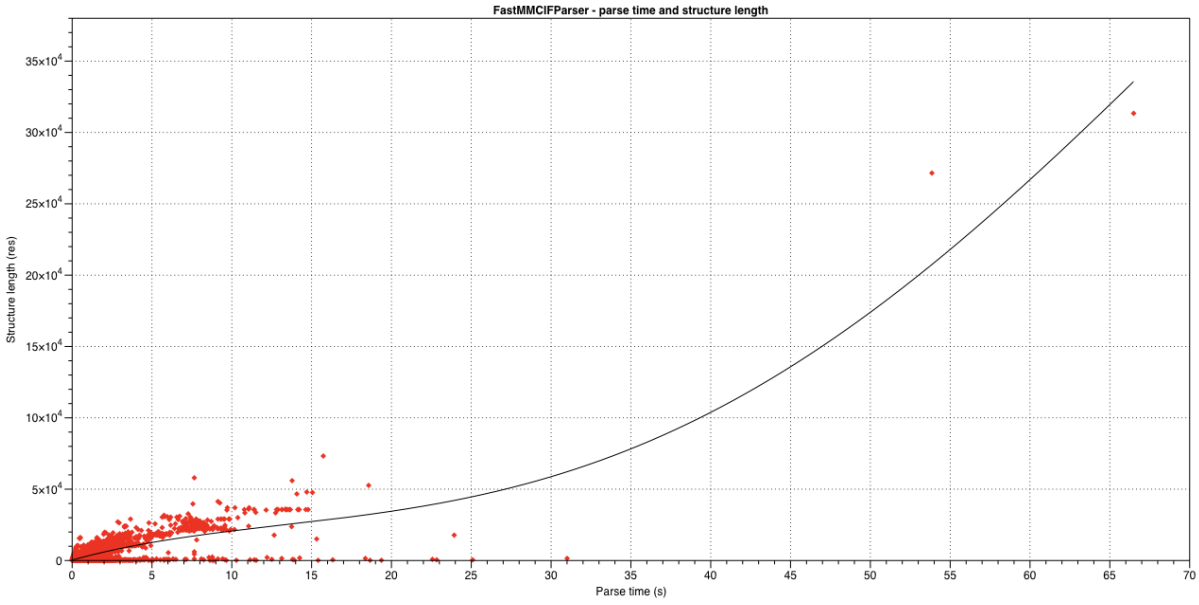
```
1a00,PDBConstructionWarning
1a00,PDBConstructionWarning
```

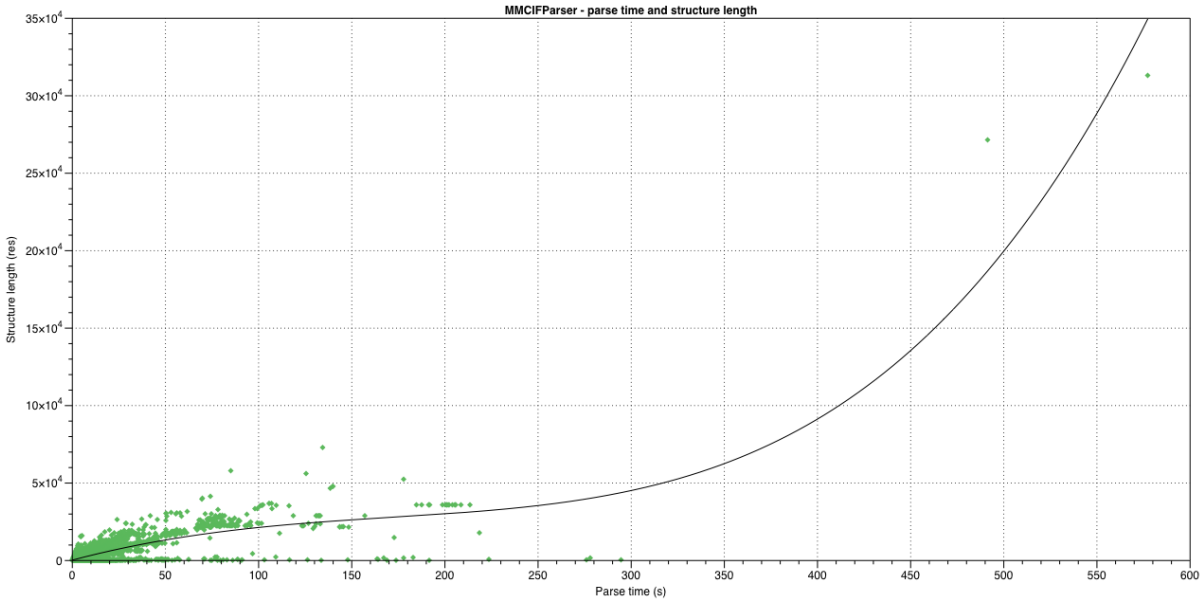
jest zamieniana na linię:

```
1a00,2
```

III. Wyniki

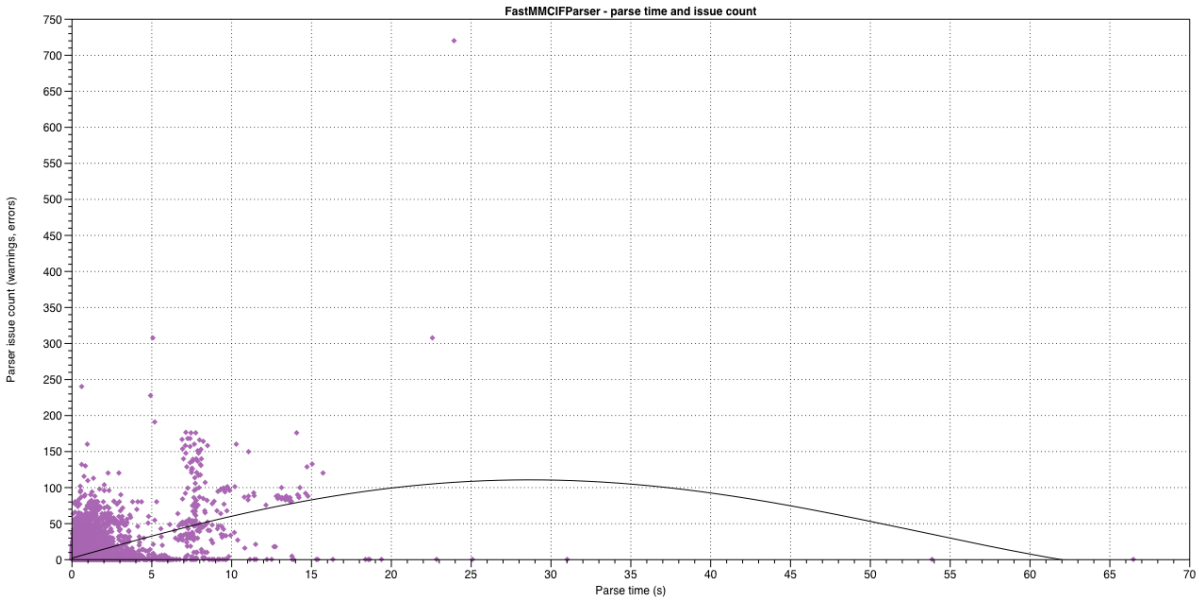
1. Czas odczytu względem długości struktury dla obu parserów:

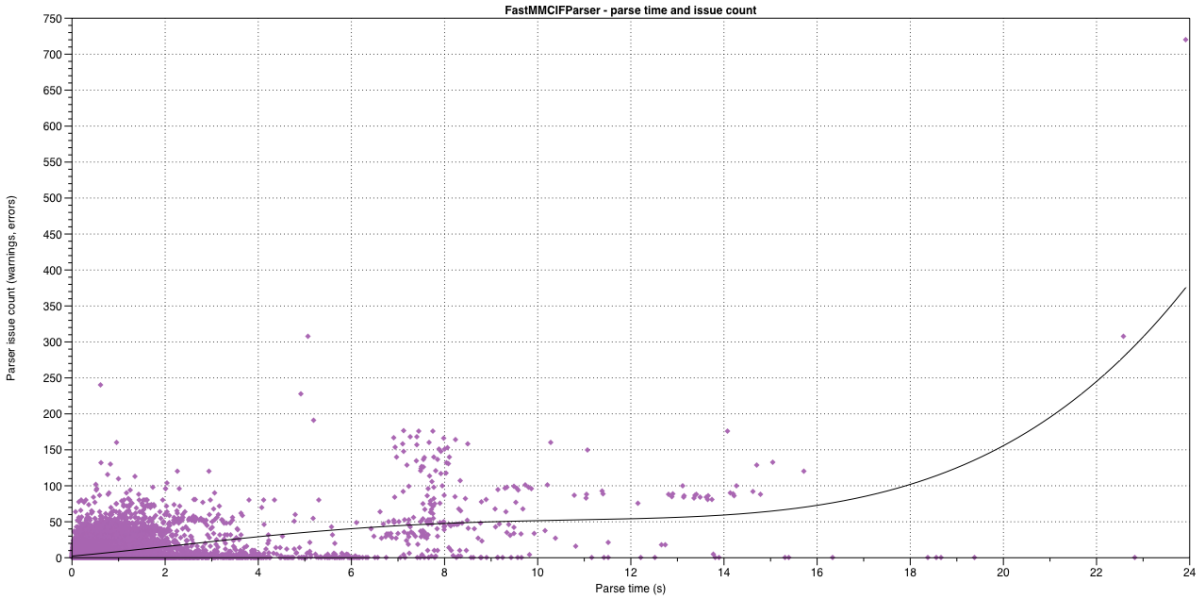


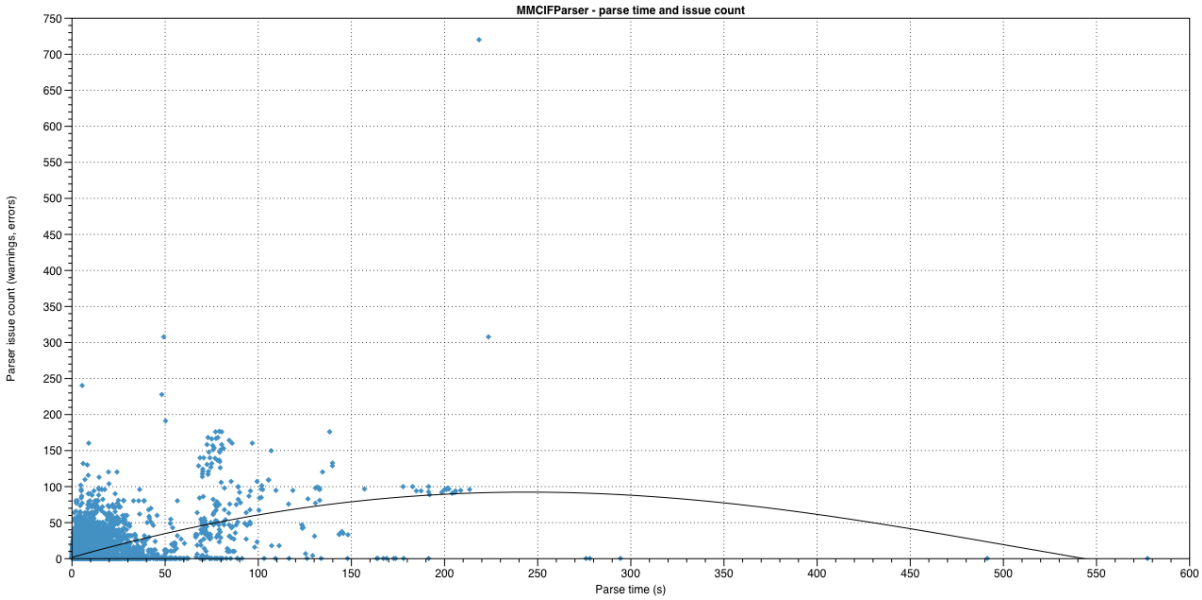


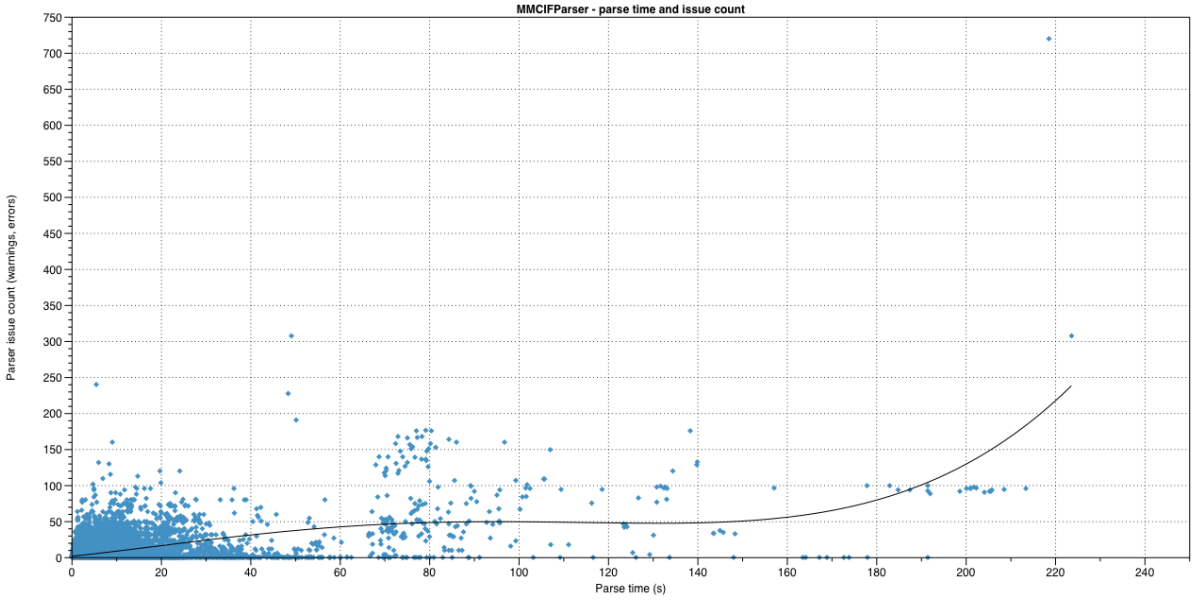
Widoczna jest zależność czasu parsowania od długości struktury - im dłuższa struktura, tym większy czas odczytu.
W prawej części wykresów można znaleźć wyniki, których czas parsowania jest znaczny mimo niewielkich struktur lub nieproporcjonalnie do nich duży.

2. Czas odczytu względem ilości ostrzeżeń dla obu parserów:









Można zauważyć pewną zależność czasu parsowania od ilości ostrzeżeń (znaczna ilość zdaje się przedłużać proces).

Po porównaniu wykresów można zaobserwować, że złożenie obu czynników (duża ilość ostrzeżeń i znaczna długość struktury) jednoznacznie wydłuża czas parsowania. Krótkie struktury o długim czasie parsowania okazały się powodować dużo ostrzeżeń parsera.

3. Ostrzeżenia - FastMMCIFParser

Oba parsery wykryły ostrzeżenia 4 typów:

1. “discontinuous chain’:
`WARNING: Chain A is discontinuous at line 258.`

2. “could not assign element”:
`Could not assign element 'UNK' for Atom (name=UNK) with given element 'X'`

3. “residue redefined”:
`Residue (' ', 31, ' ') redefined at line 3555.`

4. “disordered atom”:
`WARNING: disordered atom found with blank altloc before line 833.`

TYP	PARSER	ILOŚĆ
1. “discontinuous chain’	FastMMCIFParser	461057
1. “discontinuous chain’	MMCIFParser	460993
2. “could not assign element”	FastMMCIFParser	574
2. “could not assign element”	MMCIFParser	574
3. “residue redefined”	FastMMCIFParser	520
3. “residue redefined”	MMCIFParser	520
4. “disordered atom”	FastMMCIFParser	6
4. “disordered atom”	MMCIFParser	6

Zdecydowanie dominują błędy typu 1.
Parser FastMMCIFParser zgłosił nieznacznie więcej ostrzeżeń typu 1 niż drugi typ parsera. Poza tym, ilość znalezionych przez oba parsery problemów jest identyczna.

4. Błędy

FastMMCIFParser znalazł 3 błędy w całej zawartości bazy PDB i zgłosił je jako PDBConstructionException:

STR_ID	BŁĄD
2bwx	Blank altlocs in duplicate residue CSO (' ', 249, ' ')
1ejg	Blank altlocs in duplicate residue SER (' ', 22, ' ')
4udf	Atom C defined twice in residue <Residue LYS het= resseq=91 icode= >

MMCIFParser znalazł te same błędy typu PDBConstructionException; ponadto, wykrył 8 błędów typu ValueError:

STR_ID	BŁĄD
2a9w	No closing quotation
3b3p	No closing quotation
3dqt	No closing quotation
1mom	No closing quotation
1n5m	No closing quotation
3oj3	No closing quotation
1tsl	No closing quotation
4udf	No closing quotation