Journal of Information System Research (JOSH)
Volume 4, No. 2, Januari 2023, pp 451−457
ISSN 2686-228X (media online)
https://ejurnal.seminar-id.com/index.php/josh/
DOI 10.47065/josh.v4i2.2822

Analisis Efektifitas Algoritma FAST Menggunakan Metrik Average Percentage Fault Detection dan Waktu Eksekusi Pada Test Case Prioritization

Asri Maspupah*, Mumuh Kustino Muharram, Sophia Gianina Daeli

Jurusan Teknik Komputer dan Informatika, Politeknik Negeri Bandung, Bandung
Jl. Gegerkalong Hilir, Ciwaruga, Kec. Parongpong, Kabupaten Bandung Barat, Jawa Barat, Indonesia
Email: [1]asri.maspupah@polban.ac.id, [2]mumuhandev@gmail.com, [3]ideen.ausgehen@gmail.com
Email Penulis Korespondensi: asri.maspupah@polban.ac.id
Submitted: **02/01/2023**; Accepted: **19/01/2023**; Published: **21/01/2023**

Abstrak-Regression testing digunakan untuk memvalidasi hasil modifikasi agar tidak menimbulkan error baru pada fitur yang telah berfungsi dengan baik. Pelaksanaan regression testing umumnya dilakukan dengan mengeksekusi ulang test suite yang digunakan pada pengujian sebelumnya. Salah satu isu penting dalam regression testing adalah penentuaan strategi pendekatan untuk menggunakan kembali test suite yang sudah ada. Pengujian dengan pendekatan retest all akan menimbulkan pengujian yang lama dan mahal. Dengan demikian, perlu digunakan pendekatan pemilihan test case. Fokus kajian penelitian adalah analisis efektivitas algoritma FAST, yaitu algoritma FAST-pw dan FAST-all untuk eksekusi test case berdasarkan prioritas. Metode penelitian dengan menggunakan eksperimen melalui running algoritma pada test case yang telah ditanamkan fault pada software dengan ukuran test case 10.000 s.d. 20.000 baris program. Parameter efektifitas menggunakan waktu eksekusi test case dan persentase deteksi fault dengan menggunakan metrik average percentage fault detection (APFD). Hasil penelitian menunjukkan bahwa algoritma FAST, yaitu algortima FAST-pw dan FAST-all, memiliki nilai efektivitas yang baik saat diterapkan pada pengujian TCP dengan SUT yang berukuran kecil hingga sedang, yaitu test case yang menguji 1.000 – 20.000 baris program.

Kata Kunci: Regression Testing; Test Case Prioritization; Algoritma FAST; Average Percentage Fault Detection (APFD); Efektifitas

Abstract−It uses regression testing to validate the modification results so as not to cause new errors in features that are already functioning correctly. The implementation of regression testing is generally done by re-executing the test suite used in the previous test. One of the crucial issues in regression testing is determining a strategic approach to reuse existing test suites. Testing with retesting all test cases will result in a long and expensive test. Thus, it is necessary to use a test case selection approach. The research study focuses on analyzing the effectiveness of the FAST algorithm, namely the FAST-pw and FAST-all algorithms, for executing test cases based on priority. The research method uses experiments through running algorithms on test cases that have been implanted with faults in the software with a test case size of 10,000 to 20,000 lines of the program. The effectiveness parameter uses the test case execution time and the percentage of fault detection using the average percentage fault detection (APFD) metric. The results showed that the FAST algorithm, namely the FAST-pw and FAST-all algorithms, had good effectiveness values when applied to TCP testing with small to medium-sized SUTs, namely test cases that tested 1,000 – 20,000 program lines.

Keywords: Regression Testing; Test Case Prioritization; Algorithm FAST; Average Percentage Fault Detection (APFD); Effectivities

1. PENDAHULUAN

Selama pengembangan *software*, seringkali terjadi modifikasi fitur *software* khususnya pada jenis *software* yang dikembangkan dengan menggunakan metode agile development, karena perubahan *requirement* dapat terjadi pada setiap fase pengembangan [1]. Oleh karena itu, developer perlu memastikan agar tidak menimbulkan *error* baru pada fitur sebelumnya yang telah berfungsi dengan baik. Salah satu pendekatan yang dapat digunakan adalah *regression testing* yang bertujuan untuk memvalidasi hasil modifikasi dengan menemukan kecacatan *software* [2] [3]. Penyebab terjadi modifikasi *software* adalah terdapat penambahan, perubahan dan penghapusan *requirement*.

Pelaksanaan *regression testing* umumnya dilakukan dengan mengeksekusi ulang *test suite* yang digunakan pada pengujian sebelumnya [4]. Strategi pendekatan penggunaan kembali *test suite* sebelumnya merupakan salah satu isu penting. Terdapat dua pendekatan utama pada *regression testing*, yaitu penggunaan kembali semua test case (*retest all*), dan pemilihan sebagian *test case*. *Regression testing* yang dilakukan dengan pendekatan *retest all* membutuhkan *time* dan *cost* yang besar terutama pada aplikasi yang berukuran besar, karena semua *test case* pada *test suites* akan dieksekusi ulang [4]. Disisi lain, pendekatan *selection test case* dapat mengurangi waktu yang dibutuhkan dengan memilih *subset test case* dari *test suite* untuk menguji fitur aplikasi yang terpengaruh oleh modifikasi, sehingga eksekusi *regression testing* menjadi lebih efisien dan dapat meminimalisir biaya pengujian [5] [6]. Dengan demikian, efisiensi dari setiap pendekatan *regression testing* sangat bergantung pada pemilihan *test case* yang sesuai [7].

Pada peneliti terdahulu telah mengembangkan beberapa teknik pada pemilihan *subset test case* dari *test suite*. Berdasarkan survey yang dilakukan oleh Raul, teknik pemilihan *test case* dibagi ke dalam empat kategori, yaitu: *test case minimization* (TCM), *test case selection* (TCS), *test case optimation* (TCO) dan *test case prioritization* (TCP) [8]. Teknik TCM merupakan teknik pemilihan test case dengan cara menghapus *redundant*

Journal of Information System Research (JOSH)
Volume 4, No. 2, Januari 2023, pp 451−457
ISSN 2686-228X (media online)
https://ejurnal.seminar-id.com/index.php/josh/
DOI 10.47065/josh.v4i2.2822

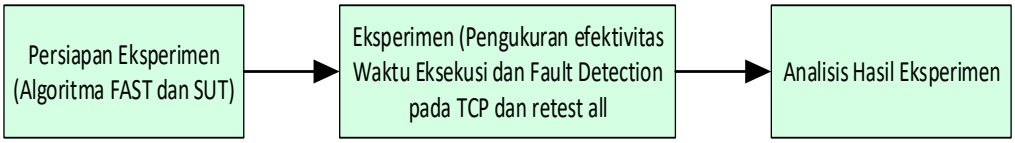
test case, yaitu *test case* yang memiliki objektif testing yang sama [8] [9]. Teknik TCS merupakan teknik pemilihan test case berdasarkan kriteria tertentu yang telah ditetapkan[6]. Misalnya, pemilihan *test case* berdasarkan *fault detection* sejak dini, sehingga *test case* terpilih dari *test suite* adalah *test case* yang dapat menemukan semua *fault* sejak awal sejak *test suites* dijalankan atau test case yang berhubungan dengan kode program yang berubah sebagai akibat dari modifikasi [6] [10]. Sementara, TCO merupakan pemilihan *test case* berdasarkan *multiple-objective optimization* dan artificial intelligent [8], sedangkan TCP merupakan penjadwalan test case dengan mengurutkan test case berdasarkan skala prioritas yang bertujuan untuk meningkatkan kinerja *test execution* [8].

Kajian utama pada artikel ilmiah ini adalah analisis efektivitas algoritma FAST pada TCP terhadap pendekatan *retest all*. Efektivitas algoritma diukur berdasarkan tingkat deteksi *fault* sejak dini dengan menggunakan metric *Average Percentage Fault Detection* (APFD). Hasil pengukuran APFD pada algoritma FAST akan dibandingkan dengan nilai metrics pada saat menggunakan pendekatan *retest all*. APFD merupakan metric untuk menghitung kecepatan *fault detection* dari sebuah test suite dengan mengukur rata-rata eksekusi *test case* menemukan kecacatan dalam bentuk persentase [11] [12]. Dengan demikian, metrik tersebut dapat digunakan sebagai evaluator efektifitas teknik TCP [11].

Secara umum, TCP adalah pendekatan *regression testing* dengan mengurutkan *test case* berdasarkan parameter prioritas, seperti waktu eksekusi, prediksi bug dari *history* sebelumnya, nilai coverage *test case* dan keterhubungan antar komponen *software* yang terdampak ketika terjadi perubahan software [2], [13]. Selanjutnya, *test case* akan dieksekusi secara berurutan satu per satu dari urutan prioritas tertinggi hingga terendah sampai kriteria pengujian terpenuhi [2], [14]. Algoritma FAST menyediakan 12 buah algoritma yang digunakan untuk memilih subset dari *test suite* pada pengujian *black-box* dan *white-box* [15]. Namun, algoritma FAST yang digunakan hanya dua buah algoritma dengan metode pengujian *white-box*.

2. METODOLOGI PENELITIAN

Metode yang digunakan pada penelitian ini adalah metode eksperimen terhadap sebuah studi kasus yang dipilih. Eksperimen dimulai dengan persiapan eksperimen, yaitu persiapan algoritma FAST dan penentuan objek eksperimen sebagai studi kasus eksperimen. Persiapan algoritma FAST dilakukan sebagai tahap pemahaman cara kerja algoritma FAST sehingga diketahui *input*, *output*, dan proses eksekusi algoritma. Selanjutnya dilakukan eksperimen dengan menjalankan algoritma FAST untuk TCP dan restest all. Kemudian dilakukan pengukuran efektifitas berdasarkan waktu eksekusi dan matrik APFD. Tahap terakhir adalah analisis hasil eksperimen dengan melihat nilai efektifitas pada kedua pendekatan. Alur tahapan penelitian ditunjukkan pada Gambar 1.



Gambar 1. Alur Tahapan Penelitian

2.1 Penentuan Case Study

Studi kasus eksperimen yang digunakan merupakan sebuah *system under test* (SUT) yang telah ditanamkan fault dan *test case* yang mampu mendeteksi fault. SUT merupakan sebuah program atau software yang disiapkan untuk kegiatan pengujian. Terdapat 3 sumber utama SUT, yaitu sekumpulan program yang berasal dari algoritma FAST, program berbahasa C yang diambil dari *software-artifact infrastucture repository* (SIR) dan beberapa program dalam bahasa java yang terintegrasi pada framework Defects4J [15]. Daftar SUT yang digunakan sebagai studi kasus eksperimen dijelaskan pada Tabel 1.

Tabel 1. Daftar *System Under Test* (SUT)

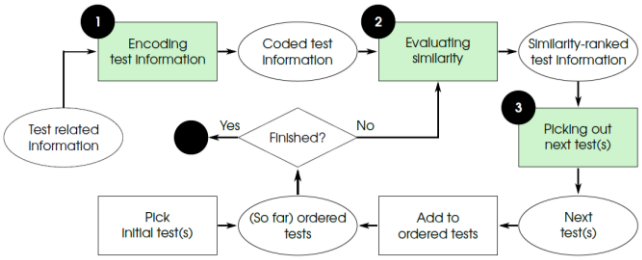
No.	Nama SUT	Sumber	LoC*	Test LoC	Jumlah TM*	Jumlah TC*	Jumlah Kesalahan	Referensi
1.	Flex	FAST	10296	-	670	-	9	[16]
2.	Grep	FAST	10124	-	809	-	8	[16]
3.	Gzip	FAST	4594	-	214	-	7	[16]
4.	Sed	FAST	13413	-	370	-	6	[16]
5.	Make	SIR	14330	-	875	-	19	[17]
6.	Closure Compiler	Defects4J	90697	84585	8124	221	101	[18]
7.	Commons Lang	Defects4J	21787	37957	2322	113	39	[18]
8.	Commons Math	Defects4J	84323	86511	3877	385	7	[18]
9.	JFreeChart	Defects4J	96382	49133	2278	356	26	[18]
10.	Joda-Time	Defects4J	27801	53158	4160	123	27	[18]

*) LoC: line of code; TM: test method (test case); TC: test class

Journal of Information System Research (JOSH)
Volume 4, No. 2, Januari 2023, pp 451−457
ISSN 2686-228X (media online)
https://ejurnal.seminar-id.com/index.php/josh/
DOI 10.47065/josh.v4i2.2822

2.2 Algoritma FAST

Algoritma FAST adalah teknik pengujian TCP untuk memilih subset test case yang beragam dari suatu test suit berdasarkan kemiripan (*similarity*) antar test case [15]. Algoritma FAST dapat dijalankan pada pengujian TCP dengan menggunakan metode *white-box* dan *black-box*. Pada pengujian *white-box*, FAST menyediakan opsi bagi pengguna untuk mengatur prioritas berdasarkan *function*, *statement*, atau *branch coverage*. FAST terdiri dari 12 buah algoritma, lima diantaranya diusulkan oleh penelitian Miranda [15]. Sementara tujuh algoritma lainnya merupakan hasil penelitian lain yang digunakan sebagai pembanding untuk pengukuran performa algoritma FAST. Gambar 2 menunjukkan alur algoritma FAST pada TCP berbasis kemiripan.



Gambar 2. Alur Algoritma FAST pada TCP berbasis similarity [15]

Gambar 2 menjelaskan alur proses pengujian TCP berdasarkan tingkat kemiripan dengan menggunakan algoritma FAST. Pertama, *encoding test information* yaitu pembuatan kode yang dilakukan ketika tahap pembuatan SUT dan test suite. Namun, pada kajian penelitian ini tidak melakukan pengkodean, melainkan pemanfaatan kode program yang sudah ada sebagai SUT termasuk *test suites* yang telah didefinisikan oleh [15], SIR, dan Defects4J. Kedua, evaluasi kemiripan antar *subset test case*, yaitu tahapan untuk mengukur tingkat kemiripan berdasarkan kesamaan dan perbedaan angota pada setiap *subset*. Kemiripan dan perbedaan antar *subset test case* menggunakan perhitungan *jaccard similarity* (JS) Pada persamaan 1 dan *jaccard distance* (JD) pada Persamaan 2.

$$JS(A, B) = |A \cap B| \div |A \cup B| \quad (1)$$

$$JD(A, B) = 1 - JS(A, B) \quad (2)$$

Keterangan: Rumus JS digunakan untuk kemiripan, sedangkan rumus JD digunakan untuk perbedaan. Variable A dan B pada kedua rumus merupakan dua buah set *test case* yang sedang dibandingkan.

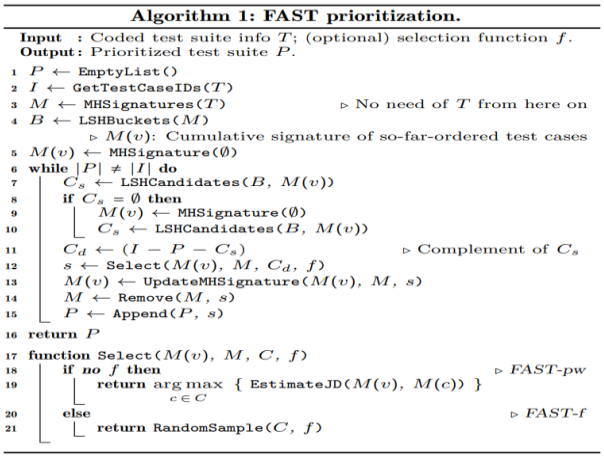
Pada penerapannya FAST memanfaatkan tiga buah algortima untuk membentuk *subset* berdasarkan kemiripan, yaitu *shingling*, *minhashing* dan *locality-sensitive hashing* (LSH) [15]. Algoritma *shingling* digunakan untuk mengelompokkan *test case* ke dalam beberapa set. Pada algoritma tersebut dilakukan pula pengukuran tingkat kemiripan pada setiap set. Setiap *set test case* disebut sebagai *signature*. Sementara, Algoritma *minhashing* digunakan untuk membentuk *signature* dengan memanfaatkan fungsi hash. Pada saat penerapannya, algoritma *minhashing* dijalankan beberapa kali sehingga mengahasilkan beberapa buah *signature*. Fungsi *hash* yang digunakan dapat berbeda-beda agar menghasilkan set *signature* yang lebih beragam. Perhitungan JS antara dua buah *set* dengan menghitung fraksi *minhash* yang sama untuk setiap urutan *test case* pada *set*. Misalnya, set *signature* (1, 1, 3, 2, 4) dan (1, 2, 1, 2, 3) dibandingkan sehingga nilai JS keduanya adalah 2/5. Hal ini karena, anggota dari set kedua signature memiliki kesamaan, yatu pada urutan 1 dan 4 dari lima aggota yang ada. Terakhir, algoritma LSH mengurangi cakupan perbandingan menjadi hanya sebuah *subset* yang terdiri atas kumpulan *test case* yang cenderung mirip. Kemudian, LSH membentuk kandidat *subset* baru dari kumpulan *test case* tersebut. Pada algoritma FAST, kemiripan antar *subset* digunakan untuk mendapatkan set yang paling tidak mirip satu sama lain.

Ketiga, *picking out next test* adalah penentukan *test case* ke dalam *subset* baru. FAST menggunakan tiga algoritma, shingling, minhashing, dan LSH, dengan cara yang berbeda sehingga menghasilkan beberapa subset *test case*. Secara keseluruhan algoritma FAST yang diperkenalkan Miranda dapat dilihat pada Gambar 3.

Gambar 3 menggambarkan aliran proses algoritma FAST pada TCP untuk membentuk subset *test case*. Pada line 17, fungsi f sebagai parameter menjalankan *function select* adalah *generic function* untuk mencapai keseimbangan yang tepat antara efisiensi dan akurasi yang dibutuhkan pada saat melakukan pengurutan berdasarkan prioritas. Fungsi f dapat diturunkan ke dalam lima fungsi, yaitu FAST-pw, FAST-one, FAST-log, FAST-sqrt, dan FAST-all. Setiap fungsi memiliki karakteristik unik, meskipun fungsi tersebut menggunakan konsep TCP yang sama. Perbedaan setiap fungsi terletak pada teknik pemilihan subset untuk masuk ke dalam kelompok *subset test case*. FAST-pw menghitung nilai JD agar mendapatkan *subset* yang paling berbeda dari subset yang dipilih sebelumnya. FAST-one hanya mengambil satu *test case* dari setiap *subset* pada setiap giliran. FAST-log mengambil *test case* sejumlah hasil perhitungan $\log_2 x+1$ dengan x adalah jumlah *test case* pada *subset*. FAST-sqrt memilih *test case* sejumlah hasil perhitungan $\sqrt{x}+1$ dengan x adalah jumlah *test case* pada *subset*. FAST-all mengambil semua *test case* pada *subset*. Namun, pada kajian penelitian ini hanya menggunakan dua

Journal of Information System Research (JOSH)
Volume 4, No. 2, Januari 2023, pp 451−457
ISSN 2686-228X (media online)
https://ejurnal.seminar-id.com/index.php/josh/
DOI 10.47065/josh.v4i2.2822

buah fungsi pada saat eksperimen, yaitu FAST-pw dan FAST-all. Pemilihan fungsi berdasarkan proses mendapatkan varian *subset test case* yang sudah diurutkan berdasarkan prioritas, kemudian subset *test case* tersebut dibandingkan dengan keselurugan *test case* pada *test suit*.



Algorithm 1: FAST prioritization.

Input : Coded test suite info T; (optional) selection function f.
Output: Prioritized test suite P.

```
P ← EmptyList()
I ← GetTestCaseIDs(T)
M ← MHSignatures(T)                          ▷ No need of T from here on
B ← LSHBuckets(M)
        ▷ M(v): Cumulative signature of so-far-ordered test cases
M(v) ← MHSignature(∅)
while |P| ≠ |I| do
    C_s ← LSHCandidates(B, M(v))
    if C_s = ∅ then
        M(v) ← MHSignature(∅)
        C_s ← LSHCandidates(B, M(v))
    C_d ← (I − P − C_s)                      ▷ Complement of C_s
    s ← Select(M(v), M, C_d, f)
    M(v) ← UpdateMHSignature(M(v), M, s)
    M ← Remove(M, s)
    P ← Append(P, s)
return P
function Select(M(v), M, C, f)
    if no f then                                          ▷ FAST-pw
        return arg max_{c ∈ C} { EstimateJD(M(v), M(c)) }
    else                                                  ▷ FAST-f
        return RandomSample(C, f)
```

Gambar 3. Algoritma FAST pada TCP [15]

Algoritma FAST dijalankan menggunakan Kanonizo *tool* yang dikembangkan oleh Miranda dkk. *Tool* tersebut bersifat *open source* pada repository Github yang dapat diakses pada tautan https://github.com/icse18-FAST/FAST. FAST dibangun dengan menggunakan bahasa pemrograman Python 2 dan mampu menguji program dalam bahasa Java dan C [16]. Versi Python yang digunakan pada kegiatan eksperimen ini adalah Python 2.7.

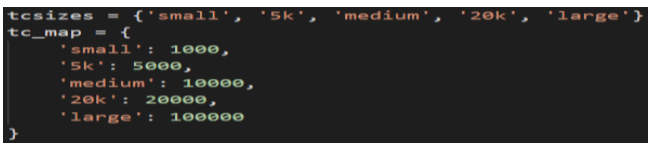
2.3 Persiapan Algoritma FAST

Persiapan algoritma FAST memiliki tiga tahapan agar FAST siap dijalankan pada saat eksperimen. Pertama, *clone project* algoritma FAST atau unduh *source code* secara manual pada tautan https://github.com/icse18-FAST/FAST. Kedua, unduh setiap dependensi yang dibutuhkan saat menjalankan algoritma FAST. Ketiga, percobaan menjalankan algoritma FAST. Percobaan dilakukan dengan eksekusi algoritma menggunakan instruksi instruksi python py/prioritize.py <subject> <entity> < function > < repetition>. Keterangan setiap parameter untuk instryksi menjalankan algoritma FAST ditunjukkan pada Tabel 2. Hasil eksekusi algoritma FAST berupa *test case* yang sudah diurutkan berdasarkan prioritas. Instruksi ini dieksekusi sebanyak dua kali untuk algoritma FAST-pw dan FAST-all dengan iterasi sebanyak 50 kali.

Tabel 2. Parameter Eksekusi Algoritma FAST

Parameter	Keterangan
< subject>	SUT yang akan digunakan pada eksperimen
<entity>	Metode pengujian yang akan digunakan, yaitu black-box TCP, atau white-box TCP (function, line, atau branch).
<function>	Function algoritma yang digunakan untuk TCP. Terdapat 12 jenis input parameter yang dapat dimasukkan, yaitu: FAST-pw, FAST-one, FAST-log, FAST-sqrt, dan FAST-all untuk algoritma FAST; ART-D dan ART-F untuk TCP berbasis ART; STR untuk pendekatan menggunakan test input string; GT, GA, dan GA-S untuk algoritma pendekatan greedy; dan I-TSD untuk pendekatan dengan menggunakan normalized compression distance antarset.
< repetition>	Jumlah iterasi yang akan dilakukan

Tahap selanjutnya, proses pengukuran efektifitas algoritma FAST dengan menggunakan fungsi FAST-pw dan FAST-all. *Test Suite* digunakan sebagai *input* untuk menjalankan algoritma FAST dengan berbagai macam variasi ukuran *test case* yang dikategorikan ke dalam *small*, *medium*, dan *large* dengan masing-masing ukuran *test suit* adalah 1000, 5000, 10.000 dan 20.000 *test case*. Dengan demikian, Kanonizo *tool* dimodifikasi sesuai dengan kebutuhan data. Modifikasi dilakukan pada file scalability.py dan generate-scalability-input.py dengan menambahkan elemen baru pada variabel array tcsizes dan tc_map. Perubahan nilai variable ukuran test case dapat dilihat pada Gambar 4.



```
tcsizes = {'small', '5k', 'medium', '20k', 'large'}
tc_map = {
    'small': 1000,
    '5k': 5000,
    'medium': 10000,
    '20k': 20000,
    'large': 100000
}
```

Gambar 4. Variabel tcsizes dan tc_map

Journal of Information System Research (JOSH)
Volume 4, No. 2, Januari 2023, pp 451−457
ISSN 2686-228X (media online)
https://ejurnal.seminar-id.com/index.php/josh/
DOI 10.47065/josh.v4i2.2822

2.4 Metrik Average Percentage Fault Detection (APFD)

APFD dikembangkan oleh Elbaum et al. yang digunakan untuk mengukur tingkat rata-rata deteksi *fault* per persentase eksekusi test suite [12]. APFD dihitung dengan mengambil rata-rata dari persentase *fault* yang terdeteksi selama pelaksanaan *test suite*. Nilai APFD berkisar dari 0 hingga 1; nilai yang lebih tinggi menyiratkan tingkat deteksi kesalahan yang lebih cepat (lebih baik). Perhitungan APFD dapat dituliskan dengan mengggunakan Persamaan 3. Pada rumus tersebut, m adalah jumlah *fault* yang ada pada SUT, n adalah jumlah test cases, dan TFi adalah posisi suatu *fault* i yang ditemukan pada *test suite*.

$$\text{APFD} = 1 - \frac{\sum_{i=1}^{m} TFi}{n \times \text{m}} + \frac{1}{2 \times \text{n}} \quad (3)$$

Pada perhitungan metrik APFD perlu diketahui terlebih dahulu jumlah fault yang ada pada SUT, jumlah test cases dan posisi (urutan ke berapa) suatu *fault* i ditemukan pada test suite. Perhatikan contoh dibawah ini.

1. Jumlah fault pada system under test (m): 7
2. Jumlah test cases (n): 4142
3. Posisi fault dapat dilihat pada tabel 2.

Tabel 3. Daftar *Fault* Pada Test Suite

Fault ke-i	Posisi ditemukannya fault (test case ke-j)
1	1
2	2
3	2
4	1
5	1
6	1
7	3

Berdasarkan data *fault* di atas, maka rincian perhitungan nilai APFD ditunjukkan pada Persamaan 4. Nilai APFD tersebut menunjukkan presentasi deteksi *fault* yang baik karena mendekati 1.

$$\text{APFD} = 1 - \frac{1+2+2+1+1+1+3}{4127 \times 7} + \frac{1}{2 \times 4127} = 0{,}999741326 \quad (4)$$

3. HASIL DAN PEMBAHASAN

3.1 Eksekusi TCP pada Algoritma FAST

Eksekusi *regression testing* dengan pendekatan TCP pada algoritma FAST untuk fungsi FAST-pw dan FAST-all masing-masing menghasilkan sebuah *file* dengan format.tsv sebanyak 50 buah *test suite*. Jumlah *file* yang terbentuk sama dengan jumlah iterasi percobaan yang dilakukan. *File* .tcp mencantumkan data waktu pembentukkan *signature*, waktu proses TCP (*prioritization time*), dan nilai APFD untuk setiap *subset* yang dibuat selama proses eksekusi algoritma. Jumlah baris pada file untuk kedua fungsi FAST-pw dan FAST-all sebanyak 352 baris, yaitu 1 baris header dan 351 subset yang dihasilkan selama proses TCP. Hasil eksekusi Algoritma FAST ditunjukkan pada Gambar 5.

SignatureTime	PrioritizationTime	APFD
0.6955645	0.1227725	0.899739583333
0.6955645	0.1227725	0.58203125
0.6955645	0.1227725	0.17578125
0.6955645	0.1227725	0.74609375
0.6955645	0.1227725	0.834635416667
0.6955645	0.1227725	0.381510416667
0.6955645	0.1227725	0.454427083333
0.6955645	0.1188111	0.826822916667
0.6955645	0.1188111	0.584635416667
0.6955645	0.1188111	0.173177083333
0.6955645	0.1188111	0.74609375
0.6955645	0.1188111	0.860677083333
0.6955645	0.1188111	0.37890625
0.6955645	0.1188111	0.451822916667
0.6955645	0.0985769	0.826822916667

(a)

SignatureTime	PrioritizationTime	APFD
0.6955645	0.0804236	0.881510416667
0.6955645	0.0804236	0.407552083333
0.6955645	0.0804236	0.303385416667
0.6955645	0.0804236	0.959635416667
0.6955645	0.0804236	0.384114583333
0.6955645	0.0804236	0.80078125
0.6955645	0.0804236	0.11328125
0.6955645	0.0190453	0.561197916667
0.6955645	0.0190453	0.756510416667
0.6955645	0.0190453	0.694010416667
0.6955645	0.0190453	0.907552083333
0.6955645	0.0190453	0.23046875
0.6955645	0.0190453	0.553385416667
0.6955645	0.0190453	0.342447916667
0.6955645	0.0189524	0.717447916667

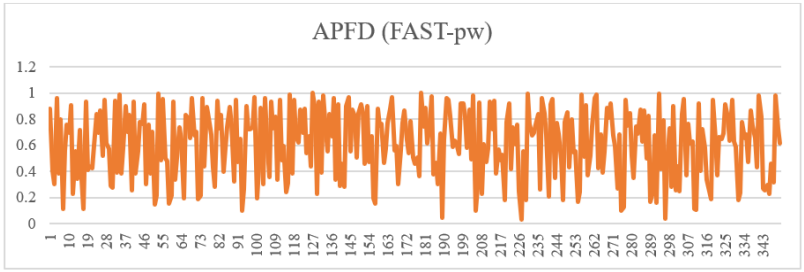
(b)

Gambar 5. Data pada *file* .tsv untuk algoritma FAST-pw (a) dan FAST-all (b)

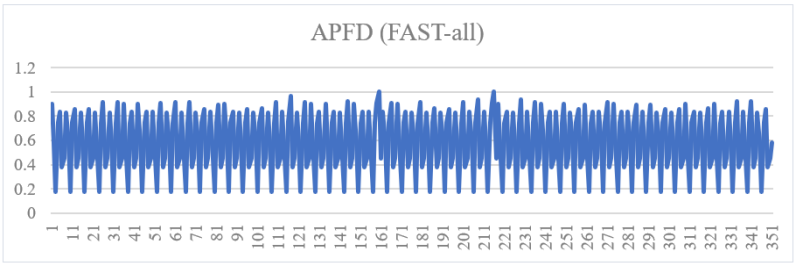
3.2 Pengukuran Efektifitas

Terdapat dua hal yang dianalisis pada pengukuran efektifitas, yaitu nilai APFD pada setiap *subset test case* yang dihasilkan saat eksekusi algoritma FAST dan hasil pengukuran waktu eksekusi pada kedua pendekatan. Hasil perhitungan nilai APFD pada algoritma FAST untuk TCP ditunjukkan pada Gambar 6. Sementara hasil perhitungan nilai APFD pada algoritma FAST untuk *retest all* ditunjukkan pada Gambar 7.

Journal of Information System Research (JOSH)
Volume 4, No. 2, Januari 2023, pp 451−457
ISSN 2686-228X (media online)
https://ejurnal.seminar-id.com/index.php/josh/
DOI 10.47065/josh.v4i2.2822



Gambar 6. Nilai APFD untuk subset hasil eksekusi algoritma FAST fungsi FAST-pw



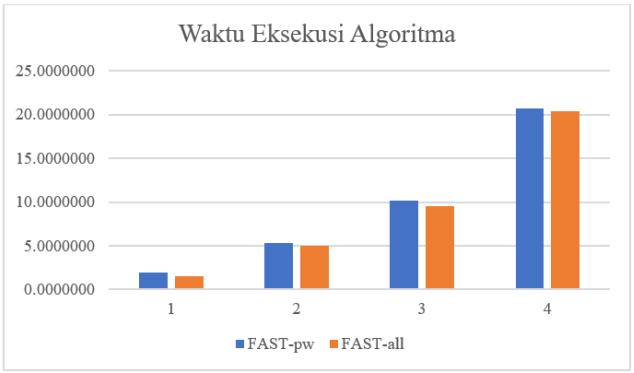
Gambar 7. Nilai APFD untuk subset hasil eksekusi algoritma FAST fungsi FAST-all

Pada Gambar 6 dan Gambar 7 terlihat bahwa fluktuasi perubahan nilai APFD pada FAST-pw lebih tajam dan inkonsisten dibandingkan dengan perubahan nilai APFD pada FAST-all yang cenderung konstan. Nilai APFD pada FAST-pw sering mengalami kenaikan dan penurunan yang tajam, sedangkan APFD pada FAST-all hamper tidak pernah mengalami perubahan yang signifikan. Nilai APFD tertinggi yang dicapai pada kedua algortima terletak pada nilai 0.998698. Rata-rata nilai APFD pada masing-masing FAST-pw dan FAST-all adalah 0.58451 dan 0.61182. Dengan demikian, dapat disimpulkan bahwa algoritma FAST-pw menghasilkan subset yang lebih beragam, jika dibandingkan dengan algoritma FAST-all yang cenderung menghasilkan subset yang mirip satu sama lain.

Hasil pengukuran efektivitas waktu eksekusi algoritma FAST pada TCP dan *retest all* yang menggunakan input *test suite* berukuran 1000, 5000, 10.000 dan 20.000 ditunjukkan pada Tabel 4. Sementara, perbandingan efektivitas waktu eksekusi pada keempat kategori divisualisasikan pada Gambar 6. Penerapan algoritma FAST pada TCP dengan memanfaatkan fungsi FAST-pw, sedangkan pengujian regresi dengan retest-all memanfaatkan fungsi FAST-all.

Tabel 4. Waktu eksekusi algoritma FAST-all dan FAST-pw

Algoritma	Ukuran Test Case (baris)			
	1.000	5.000	10.000	20.000
FAST-pw	1.9687123	5.2712096	10.137608	20.734909
FAST-all	1.4766294	5.0297568	9.5479275	20.3760328



Gambar 8. Perbandingan waktu eksekusi algoritma FAST-pw dan FAST-all

Gambar 8 menunjukkan bahwa algoritma FAST-pw membutuhkan waktu yang lebih lama saat dilakukan eksekusi pengurutan *test case* pada TCP dibandingkan dengan algoritma FAST-all. Namun, perbedaan waktu antara keduanya tidak terlalu signifikan dengan selisih 0.4205231. Disisi lain, rata-rata waktu eksekusi menunjukkan bahwa kedua algoritma stabil untuk digunakan pada *test case* berukuran kecil hingga sedang, karena waktu eksekusi berbanding lurus dengan ukuran *test case*.

Journal of Information System Research (JOSH)
Volume 4, No. 2, Januari 2023, pp 451−457
ISSN 2686-228X (media online)
https://ejurnal.seminar-id.com/index.php/josh/
DOI 10.47065/josh.v4i2.2822

4. KESIMPULAN

Berdasarkan analisa pada pengukuran efektifitas algoritma FAST, yaitu algortima FAST-pw dan FAST-all dapat disimpulkan bahwa presentase deteksi default menunjukkan kinerja yang baik dengan nilai APFD tertinggi mendekati nilai 1. Namun, fluktuasi nilai APFD pada 50 kali percobaan menunjukkan bahwa algoritma FAST-pw menghasilkan subset yang lebih bervariasi dibandingkan dengan algortima FAST-all. Sementara, waktu eksekusi FAST-pw lebih lama dibandingkan dengan waktu eksekusi FAST-all. Perbedaan waktu eksekusi pada kedua fungsi menunjukkan nilai yang tidak terlalu signifikan. Selain itu, kedua algoritma stabil ketika digunakan pada program berukuran kecil dan sedang. Dengan demikian, algoritma FAST-pw dan FAST-all masih memiliki nilai efektivitas yang baik saat diterapkan pada pengujian TCP dengan SUT yang berukuran kecil hingga sedang, yaitu test case yang menguji 1.000 – 20.000 baris. Pada penelitian selanjutnya dapat menerapkan 10 algoritma lainnya yang ada pada FAST untuk dilakukan analisis efektivitas eksekusi TCP.

REFERENCES

[1] S. Al-Saqqa, S. Sawalha, and H. Abdelnabi, "Agile Software Development: Methodologies and Trends," *International Journal of Interactive Mobile Technologies*, vol. 14, no. 11, pp. 246–270, 2020, doi: 10.3991/ijim.v14i11.13269.
[2] A. Ansari, A. Khan, A. Khan, and K. Mukadam, "Optimized Regression Test Using Test Case Prioritization," in *Procedia Computer Science*, 2016, vol. 79, pp. 152–160. doi: 10.1016/j.procs.2016.03.020.
[3] A. Maspupah and A. Bakhrun, "PERBANDINGAN KEMAMPUAN REGRESSION TESTING TOOL PADA REGRESSION TEST SELECTION: STARTS DAN EKSTAZI," *Jurnal Teknologi Terapan)* |, vol. 7, no. 1, pp. 59–67, 2021, [Online]. Available: https://github.comTestingResearchIllinois/starts.
[4] A. Ngah, M. Munro, and M. Abdallah, "An Overview of Regression Testing," *Journal of Telecommunication, Electronic and Computer Engineering*, vol. 9, no. 3, pp. 45–49, 2017, [Online]. Available: https://www.researchgate.net/publication/321243328
[5] I. Ghani, W. M. N. Wan-Kadir, A. F. Arbain, and N. Ibrahim, "Improved Test Case Selection Algorithm to Reduce Time in Regression Testing," *Computers, Materials and Continua*, vol. 72, no. 1, pp. 635–650, 2022, doi: 10.32604/cmc.2022.025027.
[6] S. P. Dongoor, "Selecting an Appropriate Requirements Based Test Case Prioritization Technique," Sweden, Mar. 2019. [Online]. Available: www.bth.se
[7] D. Suleiman, M. Alian, and Prof. A. Hudaib, "A Survey on Prioritization Regression Testing Test Case," in *8th International Conference on Information Technology (ICIT)*, 2017, pp. 854–862.
[8] R. H. Rosero, O. S. Gómez, and G. Rodríguez, "15 Years of Software Regression Testing Techniques - A Survey," *International Journal of Software Engineering and Knowledge Engineering*, vol. 26, no. 5. World Scientific Publishing Co. Pte Ltd, pp. 675–689, Jun. 01, 2016. doi: 10.1142/S0218194016300013.
[9] M. H. Alkawaz and A. Silvarajoo, "A Survey on Test Case Prioritization and Optimization Techniques in Software Regression Testing," in *IEEE 7th Conference on Systems, Process and Control*, 2019, pp. 59–64.
[10] S. Singhal, N. Jatana, B. Suri, S. Misra, and L. Fernandez-Sanz, "Systematic Literature Review On Test Case Selection and Prioritization: A Tertiary Study," *Applied Sciences (Switzerland)*, vol. 11, no. 24, pp. 1–34, Dec. 2021, doi: 10.3390/app112412121.
[11] H. Spieker, A. Gotlieb, D. Marijan, and M. Mossige, "Reinforcement Learning for Automatic Test Case Prioritization and Selection in Continuous Integration," in *ISSTA 2017 - Proceedings of the 26th ACM SIGSOFT International Symposium on Software Testing and Analysis*, Jul. 2017, pp. 12–22. doi: 10.1145/3092703.3092709.
[12] S. Nayak, C. Kumar, S. Tripathi, and L. Jena, "Efficiency Enhancement in Regression Test Case Prioritization Technique," *International Journal of Innovative Technology and Exploring Engineering*, vol. 8, no. 12, pp. 5445–5451, Oct. 2019, doi: 10.35940/ijitee.K1595.1081219.
[13] M. Azizi and H. Do, "A collaborative filtering recommender system for test case prioritization in web applications," in *Proceedings of the ACM Symposium on Applied Computing*, Apr. 2018, pp. 1560–1567. doi: 10.1145/3167132.3167299.
[14] D. K. Yadav and S. Dutta, "Regression test case prioritization technique using genetic algorithm," in *Advances in Intelligent Systems and Computing*, 2017, vol. 509, pp. 133–140. doi: 10.1007/978-981-10-2525-9_13.
[15] B. Miranda, E. Cruciani, R. Verdecchia, and A. Bertolino, "FAST approaches to scalable similarity-based test case prioritization," in *Proceedings - International Conference on Software Engineering*, 2018, vol. 2018-January. doi: 10.1145/3180155.3180210.
[16] B. Miranda, "FAST Approaches to Scalable Similarity-based Test Case Prioritization," *FAST*, 2018. https://github.com/icse18-FAST/FAST (accessed Nov. 08, 2020).
[17] National Science Foundation, "Software-artifact Infrastructure Repository," *SIR*, 2006. https://sir.csc.ncsu.edu/portal/index.php (accessed Nov. 12, 2020).
[18] René Just, "Defects4J -- version 2.0.0," *Defects4J*, 2016. https://github.com/rjust/defects4j (accessed Nov. 20, 2020).