Analisis Efektivitas Algoritma FAST++ untuk *Test Case Minimization* dalam Pelaksanaan *Regression Testing*

Ilham Gibran Achmad Mudzakir[1], Zikri Ariachandra[2], Ani Rahmani[3]

[123] *Jurusan Teknik Komputer dan Informatika - Politeknik Negeri Bandung Bandung 40012*
E-mail : {ilham.gibran.tif417, zikri.ariachandra.tif417, anirahma} @polban.ac.id

ABSTRAK

Banyak cara yang dapat dilakukan untuk memangkas biaya pengembangan aplikasi. Salah satunya dengan mengupayakan pengurangan *cost* dalam tahap pengujian, yaitu dengan mengurangi *test case*. Pengujian dilakukan untuk menjamin kualitas perangkat lunak yang dibangun, agar sesuai dengan *requirement* yang disepakati. Pengujian memiliki banyak jenis, salah satunya adalah *regression testing, yang dilakukan* sebagai dampak dari penambahan atau modifikasi fitur-fitur pada perangkat lunak yang terus berkembang. Penambahan fitur berdampak pada bertambahnya jumlah *test case* yang harus diujikan. Hal ini kemudian berdampak pada bertambahnya waktu dan *cost* yang dibutuhkan untuk menguji keseluruhan sistem. Beberapa penelitian saat ini mulai mengembangkan metode untuk mengurangi waktu eksekusi pada saat pengujian. Salah satunya adalah dengan memanfaatkan teknik *test case minimization*. Penelitian ini mengkaji efektivitas algoritma FAST++ untuk mengurangi jumlah *test case* pada *test suite* yang diujikan. Efektivitas FAST++ dihitung berdasarkan waktu eksekusi dan jumlah *test case* yang berhasil dikurangi. Digunakan juga *fault detection loss metrics* untuk memastikan algoritma reduksi tidak kehilangan kemampuannya dalam mendeteksi *fault*. Uji coba dilakukan memanfaatkan software berbahasa C, "print token" dari *Software-artifact Infrastructure Repository* (SIR) milik NC State University. Hasil uji coba menunjukkan bahwa algoritma FAST++ lebih cepat 7.02 detik dari *retest all* dalam menemukan seluruh *defect* yang ada.

Kata Kunci
Software testing , test case, test case minimization, regression testing

1. PENDAHULUAN

Dalam pengembangan perangkat lunak, tahap pengujian saat ini merupakan salah satu bidang yang paling diminati untuk diteliti. Pencarian terhadap metode atau teknik yang paling efektif dalam melakukan pengujian adalah salah satu bidang yang sering diteliti [1]. Mencari suatu metode atau teknik yang paling efektif dan efisien memerlukan penelitian yang cukup mendalam. Hal ini karena untuk menyatakan sebuah metode efektif, terdapat beberapa faktor yang harus dipertimbangkan. Salah satu faktor yang harus dipertimbangkan adalah *system under test* (SUT) yang merupakan objek pengujian. Efektivitas juga dapat dihitung dari besarnya biaya untuk melakukan pengujian, karena pada kenyataannya proses *testing* dapat menghabiskan kurang lebih 50% dari keseluruhan biaya pengembangan *software* [2]. Dengan begitu, upaya meningkatkan efektivitas pengujian yang dilakukan dapat membantu mengurangi biaya yang dikeluarkan.

Proses pengujian juga sering dilakukan pada sistem yang telah dikembangkan sebelumnya. Fungsionalitas yang ditambahkan pada sistem umumnya memunculkan *defect* dan *bug* baru ketika diintegrasikan dengan sistem yang telah ada [3]. Hal ini berpotensi untuk menambah biaya pengembangan sistem, karena proses pengujian harus melibatkan *test suite* yang sebelumnya diujikan kepada sistem, untuk memastikan fitur-fitur yang telah ada tidak memunculkan *bug* atau *defect* baru. Teknik pengujian ini disebut sebagai *regression testing*. Terdapat beberapa teknik *regression testing*, yaitu *test case minimization, test case selection,* dan *test case prioritization (TCP).*

Paper ini membahas kajian terhadap *test case minimization*, yang memiliki tujuan untuk mengidentifikasi *test case* yang redundan di dalam *test suite*, dan menghilangkannya agar ukuran *test suite* berkurang [4], sehingga *cost* keseluruhan dalam melakukan proses testing berkurang.

Pendekatan *test case minimization* merupakan salah satu solusi untuk mengatasi ukuran *test suite* yang besar [2]. Pada studi yang dilakukan, algoritma yang diterapkan adalah FAST++, yaitu salah satu bagian dari algoritma yang dikembangkan oleh [1] pada penelitiannya.

Algoritma FAST merupakan *similarity-based test prioritization*, yang umumnya digunakan untuk melakukan *test case prioritization* (TCP). Namun pada penelitian [1] algoritma tersebut dimodifi-kasi untuk melakukan reduksi pada *test suite* yang ada. Algoritma inilah yang kemudian dikenal sebagai algoritma FAST-R. Pendekatan FAST-R merupakan

gabungan *similarity-based* dan *big data*. Pada penelitiannya [1] membangun algoritma FAST-R menjadi algoritma-algoritma lainnya, salah satu algoritma yang dibuat adalah algoritma FAST++ yang menjadi bahasan utama dari penelitian ini.

Setelah pendahuluan, pada bagian 2 akan dibahas penelitian yang relevan dengan penelitian yang dilakukan; bagian 3 membahas metode pelaksanaan; bagian 4 menjelaskan hasil dan pembahasan, dan terakhir kesimpulan.

2. PENELITIAN TERKAIT

Telah banyak penelitian untuk memperoleh teknik *test case minimization* yang lebih efektif. Studi [1] membahas kelengkapan NP atau *NP-Completeness* (non deterministic polynomial time) dalam masalah *test suite minimization* yang mendorong penggunaan *heuristics*. Menurutnya, *heuristics* merupakan *problem solving tools* yang menggunakan metode praktis yang tidak dijamin optimal, sempurna, atau rasional, akan tetapi tetap cukup untuk mencapai tujuan atau pendekatan dalam jangka pendek atau langsung. Cocok untuk kondisi dimana menemukan solusi optimal tidaklah mungkin atau tidak praktis karena membutuhkan waktu yang lama. Metode *heuristic* dapat digunakan untuk mempercepat proses menemukan solusi yang memuaskan [6]. Terdapat variasi dari *greedy algorithm* yang dikenal sebagai *effective heuristics*, yaitu GE dan GRE *heuristics.*

- GE (*Greedy Evolution*) *Heuristics* : memilih semua *test case* yang esensial di dalam *test suite*. Gunakan *greedy algorithm* untuk *test requirements* yang tersisa [1].
- GRE (*Greedy Heuristic*) *Heuristics* : pertama hilangkan semua *test case* redundan didalam *test suite*. Hal ini akan membuat beberapa *test case* menjadi esensial, setelah itu lakukan GE *Heuristic* pada *test suite* yang sudah direduksi [1].

Test case redundant adalah test case yang hanya memenuhi sebagian *test requirement* yang mana sudah dipenuhi oleh test case lain, dan *test case* esensial adalah kebalikannya. Dari semua pendekatan ini tidak ada yang menunjukkan bahwa satu pendekatan lebih baik dari pendekatan lainnya, karena pendekatan ini merupakan pendekatan yang *heuristic* bukan *precise algorithm* [1].

Penelitian [2], menelaah *regression test*, khususnya *Test Case Minimization* yang dikategorikan menjadi 5 (lima), yaitu: (i) *randomized unit testing*, (ii) *user session testing*, (iii) *retargeted compilers testing*, (iv) *integer linear programming*, dan (v) *automated fault-location*. Sedangkan penelitian [3] mengkategorikan *Test Case Minimization* hanya pada satu kategori yaitu *heuristic based*. *Heuristic based* memanfaatkan algoritma *searching* seperti *Greedy Algorithm* dan *Multi Objective Algorithm* untuk mencari dan mengurangi *test case* yang akan digunakan untuk *regression testing*, seperti yang termuat pada penelitian [1] mengenai FAST++.

Pada penelitian [2] dinyatakan bahwa saat ini metode yang paling umum digunakan untuk *Test Case Minimization* adalah *randomized unit testing.* Metode ini fokus untuk menguji komponen terkecil dari *software* yang akan diuji (*software under test* /SUT) dengan memanfaatkan fungsi *random*.

Penelitian [3,5] menyampaikan bahwa selain memanfaatkan *random algorithm*, *Test Case Minimization* dapat dicapai juga melalui pendekatan lainnya seperti *model-based, coverage-base* dan *graph-base*. Pendekatan yang disam-paikan juga memiliki karakteristiknya masing-masing dan telah diuji dengan memanfaatkan SUT yang berbeda-beda.

Penelitian [5] menegaskan bahwa proses *regression testing* tidak terbatas pada fungsionalitas saja. Melalui penelitiannya menunjukan bahwa algoritma *Test Case Minimization* dapat digunakan untuk mereduksi *test suite* yang berkaitan dengan komponen graphical user interface (GUI) dengan mereduksi *event-event* yang tidak berpengaruh besar pada pengujian yang dilakukan. Secara umum penelitian [2-5] mencoba merangkum metode-metode yang umumnya digunakan untuk melakukan *Test Case Minimization*.

Penelitian lain, dilakukan oleh [7] yang menerapkan pendekatan lain berupa *scope-aided test*, yaitu proses pengujian *regression* pada komponen aplikasi yang dapat atau akan digunakan kembali (*reuse*). Pendekatan tersebut memanfaatkan *code coverage* dari aplikasi yang diujikan. Pendekatan ini bekerja pada *black-box testing* maupun *white-box testing*.

3. METODE

3.1 Software Under Test yang Digunakan

Uji coba dilakukan menggunakan *system under test* (SUT) bernama **print tokens,** yaitu sebuah perangkat lunak yang diperoleh dari *Software-artifact Infrastructure Repository* (SIR) milik NC State University [8]. SUT ini dibuat dengan bahasa C dan memiliki kurang lebih 563 *line of code* (LOC).

Pada pelaksanaan pengujian digunakan 2 buah program berupa 2 versi SUT yaitu **original version** dan **version 7**. *Original version* adalah versi SUT yang tidak memiliki *bugs* sedangkan *version 7* adalah versi SUT yang telah ditanami *bugs* di dalamnya. Untuk menguji dua versi program tersebut, SIR telah menyediakan sebuah *test plan* yang mengeksekusi 1107 *test case* untuk menemukan *bugs* di dalam seluruh versi dari SUT.

3.2 FAST++

Penelitian [1] mengenalkan beberapa pendekatan yang scalable untuk mereduksi test suite. Pendekatan yang scalable tersebut disebut dengan pendekatan

keluarga FAST-R. Pendekatan FAST-R merupakan pendekatan yang terdiri dari gabungan similarity-based dan beberapa teknik dari domain big data. Pada pendekatan FAST diterapkan min hashing dan locality-sensitive hashing algorithms. Pendekatan FAST-R mengadopsi heuristic efficient untuk memperoleh set B yang tersebar di dalam ruang big data. Lebih tepatnya, ada satu pendekatan yang bernama FAST++ yang mana mengaplikasikan algoritma k means++.

Kemudian, penelitian oleh [1] memperkenalkan pendekatan lainnya yang bernama FAST-CS yang menggunakan sampling algorithm untuk membangun coresets, sebuah teknik clustering yang dapat scale up ke dataset yang besar. Untuk meningkatkan performa dari pendekatan yang dimiliki, teknik random projection diterapkan. Random projection mereduksi dimensi test suite dengan tetap menjaga titik titik pairwise distance yang ada.

Algoritma FAST++ memiliki dua buah masukan yaitu *test suite* dan *fault matrix*. *Fault matrix* dibuat dalam format text (.txt) dan berisikan daftar *test case* yang menemukan *fault*. *Metrics* yang digunakan untuk menghitung efektivitas dari Algoritma FAST++ adalah *Fault Detection Loss* (FDL) yang akan menghitung jumlah *loss* dari *test suite* yang telah direduksi. Saat ukuran dari *test suite* direduksi, kemampuan *fault detection* atau *code coverage* dari *test suite* mungkin saja hilang [5]. Oleh karena itu, perhitungan FDL digunakan untuk memastikan *test suite* yang telah direduksi tidak kehilangan kemampuannya dalam mendeteksi adanya *fault* pada program.

Algoritma 1

Input : Test Suite *T*, Fault Matrix *F*
Output : Reduced Test Suite File *f*

```
1.  P ← RandomProjection(T)
2.  s ← FirstSelection (P)
3.  R ← List(s)
4.  D ← Distance()
5.  D(s) ← 0
6.  while (Size(R) < B) do
7.    for all t ∈P do
8.      if d(P(t),P(s))^2 < D(t)
    then
9.        D(t) ← d(P(t), P(s))^2
10.   s ← ProprotionalSample(P,D)
11.   R ← Append(R,s)
12.   D(s) ← 0
13. end while
14. fdl ← FDL(R, F)
15. f ← OpenFile('reducedTS.txt')
16. for all tc ∈R do
17.   Write(f, tc)
18. CloseFile(f)
19. return f
```

Secara garis besar, algoritma FAST++ yang digunakan memiliki 3 (tiga) tahapan, yaitu (i) *Preparation*, (ii) *Reduction*, dan (iii) *Completion*. Pada tahap *preparation*, FAST++ akan memetakan *test case* yang ada pada *test suite* ke dalam bentuk *vector* berdasarkan *vector-space model* dan mengecilkan dimensi dari *test case* menggunakan *random projection* [1], tahap ini dapat dikatakan tahap *preprocessing* untuk selanjutnya dimasukkan ke tahap *Reduction*. Pada Algoritm 1, tahap *preparation* dapat dilihat pada baris kode 1 sampai 3.

Pada tahap *reduction*, *vector* yang telah dibuat dimasukkan ke dalam algoritma *k-means* untuk dihitung keterhubungannya. Tahap ini dapat dilihat pada *Algoritm* 1 baris 4 sampai 14. Algoritma *k-means* membutuhkan masukan berupa jumlah *cluster* yang diinginkan dari kumpulan *vector*. Pada algoritma FAST++, nilai *cluster* untuk algoritma *k-means* adalah jumlah *test case* yang diinginkan untuk diambil dalam satu kali pengujian. Besaran ini dapat bervariasi jumlah dan pemaknaannya, pada penelitiannya [1] mengibaratkan nilai ini dengan besaran biaya yang dijadikan batasan pada pengujian, yang dikonversi dalam jumlah *test case*. Pada eksperimen ini, kami mendefinisikan nilai ini dengan 20% dari jumlah *test case* yang dimiliki.

Langkah terakhir yang dilakukan adalah *completion*. Pada tahap ini, *test case* yang telah direduksi oleh Algoritma dimasukkan ke dalam *file* dengan ekstensi text (.txt). Tahap ini terlihat pada Algoritma 1, baris 15 sampai 18. Hasil keluaran pada tahap *reduction* adalah nomor *test case* yang dipilih, sehingga keluarannya adalah kumpulan angka yang menunjukan *test case* yang dipilih. Oleh karena itu diperlukan tahapan untuk mengembalikan *test case* yang telah dipilih ke dalam format yang sama dengan *test suite* masukan. Hal ini dilakukan untuk mempermudah proses eksekusi pada SUT yang dipilih.

3.3 Tools untuk Pengujian

Pelaksanaan pengujian, *retest all* dan *test case minimization* dilakukan dengan memanfaatkan ***mts tools,*** yaitu *tools* yang disediakan oleh SIR untuk melakukan *automation test*. *Tools* ini memerlukan *project directory*, *executable program*, *test plan*, dan *directory* pembanding sebagai masukan.

Cara kerja *tools* ini yaitu dengan membuat sebuah *shell script* yang dapat digunakan untuk menjalankan *automation test* pada program berdasarkan *test plan* yang dimasukkan. *Shell script* yang dihasilkan, digunakan untuk menguji program yang dimiliki dengan seluruh *test suite* yang dijadikan masukan dari ***mts*** *tools*. Eksekusi program menghasilkan kumpulan *output* hasil eksekusi sejumlah *test case* yang diujikan, sehingga hasil eksekusi dari uji coba akan menghasilkan 1107 *output file*.

3.4 Pelaksanaan Uji Coba / Eksperimen

Proses eksperimen menggunakan **mts** *tools* dilakukan sebanyak dua kali, yaitu untuk mengeksekusi program **original** dan **version 7**. Dengan demikian, akan terdapat 1107 *output* untuk setiap eksekusi. Setelah proses eksekusi dilakukan, proses selanjutnya adalah membandingkan *output file* **original program** dan **version 7 program**. Pada penelitian ini, selanjutnya dibandingkan performa *retest all* dan algoritma reduksi memanfaatkan FAST++. Oleh karena itu, kita akan melakukan eksperimen untuk masing-masing pendekatan *regression test* yaitu *retest all* dan FAST++.

3.5 Analisis Efektivitas

Penelitian ini mengkaji efektivitas algoritma FAST++ untuk mengurangi jumlah *test case* pada *test suite* yang diujikan. Efektivitas FAST++ dihitung berdasarkan waktu eksekusi dan jumlah *test case* yang berhasil dikurangi. Digunakan juga *fault detection loss metrics* untuk memastikan algoritma reduksi tidak kehilangan kemampuannya dalam mendeteksi *fault*. Algoritma 2 menunjukkan algoritma perhitungan *Fault Detection Loss* (FDL).

Algoritma 2

Input : Reduced Test Suite *R*, Fault Matrix *F*

Output : Fault Detection Loss *fdl*

```
1.  tf ←0
2.  df ← 0
3.  for all tc ∈ R do
4.     for all f  ∈ F do
5.        if tc = f  then
6.           df ← df + 1
7.           break
8.  tf ← Size(F)
9.  fdl ← (tf - df) / tf
10. return fdl
```

4. HASIL DAN PEMBAHASAN

Proses *retest all* melakukan pengujian program dengan memanfaatkan seluruh *test suite* yang dimiliki. Artinya, proses *retest all* akan memanfaatkan *test suite* yang belum direduksi. Berdasarkan hasil eksekusi *retest all* terhadap seluruh *test suite* yang dimiliki, pendekatan *retest all* membutuhkan 47.48 detik untuk dapat mendeteksi seluruh *fault* yang ada.

Algoritma FAST++ memanfaatkan *random projection* untuk mengecilkan dimensi dari *test suite* yang dimasukkan. Hal ini berdampak pada pemilihan *test suite* awal yang bersifat acak, sehingga algoritma FAST++ perlu untuk dieksekusi beberapa kali agar dapat memperoleh hasil yang optimal.

Oleh karena itu, eksperimen dilakukan dengan perulangan sebanyak 10 kali untuk menemukan *test suite* yang optimal. Tabel 1 menunjukkan hasil eksekusi algoritma FAST++.

Tabel 1. Hasil Eksekusi FAST++

Perulangan	Preparation Time (detik)	Reduction Time (detik)
1	0.2421794	0.0107574
2	0.2278554	0.0119236
3	0.2309506	0.0069575
4	0.2289696	0.0056420
5	0.2378146	0.0086285
6	0.2532936	0.0092390
7	0.2317918	0.0075687
8	0.2405149	0.0073762
9	0.2489350	0.0088430
10	0.2305299	0.0082280

Selanjutnya dilakukan pengujian dengan mengeksekusi *test suite* yang telah direduksi pada SUT yang dipilih. Tabel 2 menampilkan hasil eksperimen yang dilakukan.

Tabel 2. Hasil Eksekusi *Test suite* FAST++

Test Plan	Execution Time (detik)	Fault Detection Rate
1	5.96	4/11
2	5.59	2/11
3	5.53	4/11
4	5.96	2/11
5	5.86	3/11
6	5.53	2/11
7	6.03	3/11
8	5.72	3/11
9	5.73	5/11
10	5.97	3/11

Tabel 2 memperlihatkan jumlah *test plan* yang dibuat berdasarkan perulangan pada algoritma FAST++. Seperti yang telah disampaikan sebelumnya, algoritma FAST++ akan dieksekusi sebanyak 10 kali, hal ini menyebabkan jumlah *test plan* yang dihasilkan

pun sebanyak 10 buah *test plan*. *Test plan* ini kemudian dieksekusi menggunakan *mts tools.* Durasi dari eksekusi setiap *test plan* diperlihatkan pada kolom *execution time* pada Tabel 2. Dapat dilihat dari keseluruhan eksperimen yang dilakukan, setiap *test plan* rata-rata dapat menemukan kurang lebih 2 sampai 3 *defect* dari 11 *defect* yang ada pada program. Hal ini diperlihatkan pada kolom *fault detection rate*. Perhitungan *fault detection rate* dilakukan menggunakn formula 1-FDL, dengan demikian dapat diperoleh dengan pasti jumlah *fault* yang dapat dideteksi oleh masing-masing *test suite* yang telah direduksi.

Berdasarkan Tabel 2, dapat dipahami bahwa dari sepuluh variasi *test suite* hasil reduksi, tidak terdapat satupun yang berhasil menemukan seluruh *fault* yang ada. Untuk lebih memahami hasil eksperimen, dilakukan pemetaan *fault* mana saja yang berhasil ditemukan oleh *test suite* yang telah direduksi. Tabel 3 menunjukkan hasil pemetaan atau *fault matrix* untuk setiap variasi *test suite* yang telah direduksi.

Tabel 3 Pemetaan *fault* pada *reduced test suite*

Nomor Test Suite	Fault Detected
1	3, 7, 10, 11
2	5, 6
3	1, 5, 9, 11
4	1, 4
5	4, 5, 11
6	4, 11
7	2, 3, 8
8	2, 4, 6
9	2, 3, 4, 7, 11
10	1, 4, 5

Berdasarkan hasil dari Tabel 3, dapat dipahami bahwa variasi *test suite* yang dihasilkan dari algoritma FAST++ memungkinkan adanya redudansi antar variasi. Dengan begitu variasi *test suite* bukan dimaksudkan untuk mengisi atau menguji *test suite* yang belum diujikan pada variasi sebelumnya.

Seperti yang telah disampaikan pada bagian uji coba, algoritma FAST++ dibangun memanfaatkan *random projection*, algoritma ini bisa jadi menyebabkan *test case* dipilih secara acak, sehingga hasil setiap perulangan atau repetisi saat mengeksekusi FAST++ akan menghasilkan keluaran yang berbeda dan benar-benar acak. Hal ini berdampak pada proses deteksi *fault* yang menjadi tidak menentu.

Jika diamati Tabel 3 dengan baik, dapat dipahami bahwa dengan mengeksekusi variasi 1 sampai 7 dari *test suite* yang telah direduksi akan menghasilkan 100% *fault detection*, namun selama proses eksekusi terdapat beberapa redundansi *test case* yang diujikan. Hal ini berdampak pada tingkat efektivitas dan efisiensi dari proses pengujian. Selain itu, *execution time* yang dibutuhkan untuk mengeksekusi variasi *test suite* 1 sampai 7 adalah sekitar 40.46 detik, lebih cepat 7.02 detik dari *retest all*. Namun hasil ini diperoleh dengan mengeksekusi 1547 *test case* secara jumlah. Artinya lebih banyak 440 *test case* dari *retest all*.

Hasil uji coba yang diperoleh juga dapat dikatakan bersifat tidak pasti, karena hasil dari proses reduksi bersifat tidak pasti. Nilai rata-rata *fault detection* pun berkisar di angka 3.1 *fault* ditemukan, untuk masing-masing variasi *reduced test suite*.

5. KESIMPULAN

FAST++ adalah algoritma yang dibangun dengan algoritma *heuristic*, sehingga untuk menjalankan FAST++ dibutuhkan data *fault matrix* yang menunjukkan letak *fault* pada program. Meski waktu yang dibutuhkan terhitung cepat, yaitu kurang dari 1 detik untuk mereduksi *test suite* dengan *rate* 3.1/11 *fault* ditemukan, FAST++ masih dirasa kurang efektif karena masih terdapat redudansi pada *test suite* yang dihasilkan. Meski nilai *fault detection loss* bernilai 0.0 yang menunjukkan *test plan* yang dibuat masih mampu mendeteksi adanya *fault*, namun tidak ada satupun *test plan* yang berhasil menemukan seluruh *fault* yang ada.

Selain itu, Variasi *test suite* yang bersifat acak juga menjadikan risiko dalam proses *regression testing* menjadi meningkat, karena *tester* tidak dapat memastikan apakah eksekusi FAST++ yang dilakukan dapat mereduksi dengan efektif *test suite* yang ada, meskipun jumlah *test case* yang direduksi dapat didefinisikan oleh *tester*.

Eksperimen yang dilakukan masih memanfaatkan satu **versi** dari SUT *print token* yang disediakan, disarankan untuk mengkombinasikan penggunaan beberapa **versi** dari SUT yang tersedia untuk memperkaya *fault* yang dapat dideteksi oleh pendekatan yang dipilih (*retest all* dan FAST++). Disarankan pula untuk melakukan eksperimen dengan memanfaatkan SUT yang berasal dari proyek riil,

DAFTAR PUSTAKA

[1] E. Cruciani, B. Miranda, R. Verdecchia, and A. Bertolino, “Scalable Approaches for Test Suite Reduction,” presented at IEEE/ACM 41st ICSE, Montreal, Canada, 25-31 May, 2019.

[2] S. U. R. Khan, S. P. Lee, R. W. Ahmad, A. Akhunzada, V. Chang, "A survey on Test Suite Reduction frameworks and tools" in *International Journal of Information Management*, vol. 36, Dec. 2016, pp 963-975.

[3] R. H. R. Miranda, G. D. Rodriguez, O. S. Gomez, "15 Years of Software Regression Testing Techniques -- A Survey," in *International Journal of Software Engineering and Knowledge Engineering*. Jul. 2016.

[4] S. Yoo, M. Harman, "Regression testing minimization, selection and prioritization: a survey," in *Software Testing, Verification and Reliability*, vol. 22, 2010.

[5] H. Do, "Recent Advances in Regression Testing Techniques" in *Advance in Computers*, vol. 103, 2016, pp. 53-77.

[6] J. J. Lu, M. Zhang, "Heuristic Search" in *Encyclopedia of Systems Biology*, 2013. pp. 875-936.

[7] B. Miranda and A. Bertolino, "Scope-aided test prioritization, selection and minimization for software reuse," *J. Syst. Softw.*, vol. 131, pp. 528–549, 2017, doi: 10.1016/j.jss.2016.06.058.

[8] National Science Foundation, diakses pada 6 Agustus 2020 ,https://sir.csc.ncsu.edu /portal/index.php