Jurnal Pengembangan Teknologi Informasi dan Ilmu Komputer e-ISSN: 2548-964X
Vol. 2, No. 1, Januari 2018, hlm. 246-255 http://j-ptiik.ub.ac.id

Perbandingan *Test Case Generation* dengan Pendekatan *Genetic Algorithm Mutation Analysis* dan *Sampling*

Christopher Dimas Satrio[1], Mochamad Chandra Saputra[2], Aditya Rachmadi[3]

Program Studi Sistem Informasi, Fakultas Ilmu Komputer, Universitas Brawijaya
Email: [1]christopherdimass@gmail.com, [2]andra@ub.ac.id, [3]rachmadi.aditya@ub.ac.id

Abstrak

Pengujian perangkat lunak memiliki peranan penting dalam mengetahui dan menjaga kualitas suatu perangkat lunak. Dalam proses pengujian perangkat lunak sangat diperlukan adanya *test case*. Namun, pembuatan *test case* membutuhkan waktu yang relatif lama sehingga dilakukan pembuatan *test case* secara otomatis menggunakan algoritme tertentu. Salah satu algoritme yang sering digunakan adalah *genetic algorithm*. Banyak pendekatan *genetic algoritm* yang telah dikembangkan. Namun, belum ada penelitian yang membandingkan pendekatan *genetic algorithm* yang lebih baik. Penelitian ini akan membandingkan pendekatan *genetic algorithm mutation analysis* dan *sampling*. Kedua pendekatan tersebut akan diimplementasikan dan dianalisis hasilnya. Dari segi hasil akhir akan dilihat jumlah pengurangan *test case* yang terjadi. Banyaknya iterasi, akumulasi jumlah individual, banyaknya evaluasi *fitness* yang terjadi, dan ukuran dari *test suites* akan menjadi variabel pembanding di antara kedua pendekatan tersebut. Kesimpulan dari penelitian ini adalah pendekatan *genetic algorithm mutation analysis* lebih baik dibanding *genetic algorithm sampling* pada jumlah pengurangan *test case* dan semua variabel pembanding yang diterapkan.

Kata kunci: *test case, genetic algorithm, mutation analysis, sampling, test suite*

Abstract

Software testing has an important role to knowing and maintaining the quality of a software. In software testing process, test case is necessary. However, test case generation require a relatively long time, so automatic test case generation with a certain algorithm are conduct. The purpose of this research are to compare genetic algorithm mutation analysis and genetic algorithm sampling approach. That's two approach wil be implemented and analyzed the result. In terms of the final result, will be revealed the reduction of the number of test case that occurs. Number of iteration, cumulative number of individuals, number of fitness evaluation, and size of the resulting test suites will be variable comparison between that's two approach. The conclusion of this research are the genetic algorithm mutation analysis approach is better than genetic algorithm sampling approach in terms of the reduction number of test case and for all comparison variable that applied.

Keywords: *test case, genetic algorithm, mutation analysis, sampling, test suite*

1. PENDAHULUAN

Pengujian perangkat lunak memiliki peranan yang sangat penting dalam mengetahui dan menjaga kualitas perangkat lunak. Menurut Homès (2012), pengujian adalah kumpulan dari aktivitas yang bertujuan untuk mengidentifikasi failures di dalam perangkat lunak atau sistem. Dengan kata lain, identifikasi *failures* pada *software* dapat menjaga kualitas perangkat lunak yang bersangkutan.

Menurut survei yang dilakukan oleh Perry yang dikutip dari buku *software quality assurance* oleh Galin (2004) pada November 1994 mendapatkan fakta bahwa waktu pengujian yang dijadwalkan pada keseluruhan waktu proyek yang dilakukan organisasi mencapai 27%. Partisipan pada survei ini juga menyatakan bahwa mereka merencanakan untuk mengalokasikan waktu yang lebih besar, rata-rata 45%, untuk melakukan pengujian.

Dalam proses pengujian perangkat lunak sangat diperlukan adanya *test cases*. pembuatan *test case* memiliki peran yang sangat penting dalam pengujian perangkat lunak. Pembuatan *test case* secara otomatis dapat meningkatkan

kualitas dalam proses pengujian perangkat lunak. Hal ini dikarenakan metode-metode algoritme yang diterapkan pada pembuatan *test case* dapat membantu membuat *test case* yang sesuai dengan kebutuhan (Khan & Amjad, 2015). Pengurangan jumlah *test case* menggunakan pembuatan *test case* secara otomatis membuat waktu yang dibutuhkan untuk membuat *test case* dapat dikurangi dan berdampak pada berkurangnya waktu yang dibutuhkan untuk pengujian (Ansari et al., 2017),

Terdapat banyak pendekatan yang dapat digunakan dalam pembuatan *test case* secara otomatis, misalnya *genetic algorithm.* Prinsip dari *genetic algorithm* didasari oleh prinsip evolusi dan genetik (Khan & Amjad, 2015). Penerapan metode pendekatan *genetic algorithm* memiliki beberapa pendekatan, contohnya adalah *mutation analysis* dan *sampling*.

Penelitian ini diharapakan dapat menjawab masalah-masalah seperti, jumlah pengurangan *test case*, hasil perbandingan variabel banyaknya iterasi, akumulasi jumlah individual, banyaknya evaluasi *fitness* yang terjadi, dan ukuran dari *test suites*, serta analisis perbandingan dua algoritme dengan *mutation analysis* dan *sampling*.

Penelitian ini akan menguji perangkat lunak pada tahap pembuatan *test case* pada *unit testing* dengan objek penelitian perangkat lunak berbasis web. *Sample coding* yang akan digunakan berbahasa program PHP.

Peneliti akan melakukan studi literatur, mempersiapkan data algoritme (*sample coding*), implementasi *genetic algorithm* dengan *mutation analysis* dan *genetic algorithm* dengan *sampling*, analisis hasil, dan terakhir adalah menarik kesimpulan dalam penelitian ini.

2. LANDASAN KEPUSTAKAAN

2.1 Kajian Pustaka

Dalam penelitiannya yang berjudul "Automatic Test Case Generation based on Genetic Algorithm and Mutation Analysis" Haga & Suehiro (2012) membahas proses pembuatan *test case* menggunakan pendekatan *genetic algorithm* dengan *mutation analysis* untuk mendapatkan *test case* dengan kualitas yang baik. Terdapat dua fase besar dalam jurnal ini, yaitu membuat *test case* secara random dan perbaikan *test case* dengan *genetic algorithm. Mutation analysis* digunakan untuk mengukur kecukupan dari *test case* yang dihasilkan dengan menggunakan *mutation score.*

Dalam penelitiannya yang berjudul "Optimization of Automatic Generated Test Cases for Path Testing Using Genetic Algorithm" Khan, Amjad, dan Srivastava (2016) membahas proses pembuatan *test case* menggunakan pendekatan *genetic algorithm*. Pendeketan *genetic algorithm* disini fokus pada *flow graph* yang merepresentasikan progam atau fungsi yang akan diuji. *Genetic algorithm* akan mengoptimalisasi *test case* hingga populasi *test case* yang ada dapat mencakup semua jalur yang ada pada *flow graph*. Selanjutnya penelitian oleh Mohaptra, Bhuyan, dan Mohapatra (2009) yang berjudul "Automated Test Case Generation and Its Optimization for Path Testing Using Genetic Algorithm and Sampling" membahas *random sampling* yang dapat digunakan untuk mendapatkan *test suites* yang optimal dari *test case* yang dihasilkan *genetic algorithm*.

Dalam penelitiannya yang berjudul "A Systematic Review of the Application and Empirical Investigation of Search-Based Test Case Generation" Ali, et al. (2010) membahas tentang perbandingan algoritme yang digunakan untuk *test case generation* pada *search-based software testing* (SBST). Perbandingan dapat dilakukan dengan mengukur *cost* yang harus dikeluarkan dalam mengeksekusi sebuah algoritme SBST.

Berdasarkan ketiga penelitian di atas, maka penulis akan mengadopsi pendekatan *genetic algorithm* dengan *mutation analysis* dan *sampling* dalam penelitian untuk menentukan pendekatan yang lebih baik dalam membantu proses pembuatan *test case* secara otomatis. Penentuan pendekatan yang lebih baik akan dilihat dari ukuran *cost* yang harus dikeluarkan untuk mengeksekusi masing-masing pendekatan algoritme. Ukuran *cost* ini diwakilkan dengan variabel-variabel berikut yaitu banyaknya iterasi, akumulasi jumlah individual, banyaknya evaluasi *fitness* yang terjadi, dan ukuran dari *test suites* yang dihasilkan. Variabel-variabel yang telah disebutkan tadi akan menjadi tolak ukur perbandingan antara pendekatan *genetic algorithm mutation analysis* dengan gen*etic algorithm sampling*.

2.2 Pengujian perangkat lunak

Pengujian adalah kumpulan dari aktivitas yang bertujuan untuk mengindentifikasi *failures*

di dalam perangkat lunak atau sistem. Dengan kata lain, identifikasi *failures* pada *software* dapat menjaga kualitas perangkat lunak yang bersangkutan. *Failures* adalah ketidakmampuan sebuah sistem atau komponen sistem dalam menjalankan fungsi yang dibutuhkan dengan batasan yang telah ditentukan (Homès, 2012).

2.3 *Unit testing*

Unit testing adalah proses dari pengujian komponen terkecil dari program atau perangkat lunak, seperti *method* atau obyek dari suatu *class* (Sommerville, 2011). Pengujian dapat sangat mahal dan memakan waktu, maka sangat penting dalam memilih *unit test case* yang efektif.

2.4 *Test case, test data,* dan *test suites*

Test case adalah kumpulan nilai input, prasyarat eksekusi, expected results, dan kondisi setelah eksekusi, yang dibuat untuk tujuan atau kondisi pengujian tertentu. *Test suites* menunjukan kumpulan anggota test case yang dapat yang dapat dijadikan alat uji untuk program yang akan diuji. *Test data* adalah data yang tersedia sebelum pengujian diekseskusi dan mempengaruhi atau dipengaruhi oleh sistem yang akan diuji (Homès, 2012).

2.5 Genetic algorithm

Prinsip dari *genetic algorithm* didasari oleh prinsip evolusi dan genetik (Khan & Amjad, 2015). G*enetic algorithm* terinspirasi dari proses pencarian dan seleksi alami yang mengarah pada individual paling sesuai yang tersisa (Ghiduk et al., 2010). Individual disini dapat diartikan sebagai *test case* pada pengujian perangkat lunak.

Genetic algorithm secara umum memiliki 4 fase yaitu, *evaluation*, *selection*, *crossover* dan *mutation* (Gupta & Rohil, 2008). Pada fase *evaluation* terdapat prosedur dalam mengukur kecocokan terhadap setiap solusi individual yang ada di dalam populasi dan memberikan nilai relatif yang didasarkan pada penetapan kriteria optimasi. Pada fase *selection* terdapat prosedur dalam memilih individual secara acak dari populasi saat ini untuk pengembangan generasi berikutnya. Pada fase *crossover* terjadi Kombinasi ulang menghasilkan individual baru dengan cara melakukan pertukaran informasi antar individual. Pada fase *mutation* akan terjadi membuat sedikit perubahan terhadap individual baru yang dihasilkan oleh fase *crossover*.

2.6 *Mutation analysis*

Mutation analysis mengevaluasi kualitas dari set *test case*. Inti dari pendekatan ini adalah suatu *error* program sengaja diinjeksikan ke dalam program yang ingin diuji. *Error* program ini disebut sebagai *mutant* (Haga & Suehiro, 2012). Berikut cara menghitung *mutation score*:

$$MS = \frac{detected\ mutants}{all\ mutants} \quad (1)$$

2.6.1 GA dengan *mutation analysis*

Prosedur dalam menerapkan pendekatan *genetic algorithm*(GA) dengan *mutation analysis* dapat dilihat pada gambar sebagai berikut:

Step 1	Generate initial test cases randomly.
Step 2	Generate mutants and compute the detection matrix.
Step 3	Reduce the number of test cases.
Step 4	Compute the mutation score. If it converges, exit.
Step 5	Generate a set of test cases using a GA.
Step 6	Go to step 1.

Gambar 1. Prosedur GA *mutation analysis*

Pertama, *test case* awal dibuat secara random untuk definisi awal dari berbagai input data. Kemudian *mutant* akan dibuat dengan menerapkan *mutant operator*. Berdasarkan *mutant* yang telah dihasilkan maka *detection matrix* dapat dibentuk. *Detection matrix* mengakibatkan berkurangnya jumlah *test case* tanpa mengurangi kemampuan mendeteksi *defects*. Lalu *mutation score* dihitung, jika hasilnya telah memuaskan maka prosedur dapat berakhir. Jika tidak, *test case* baru akan dibuat menggunakan GA lalu kembali ke langkah pertama.

2.7 *Random sampling*

Random sampling digunakan untuk mengekstrak *test set* dari *test case* optimal yang telah dihasilkan *genetic algorithm*, memiliki ukuran sampel yang sama dengan *cyclomatic complexity* (Mohapatra, et al., 2009). Berikut perhitngan menggunakan *random sampling*:

$$Q = \frac{N}{T}c \quad (2)$$

2.7.1 GA dengan *sampling*

Prosedur dalam menerapkan pendekatan GA dengan *random sampling* dapat dilihat pada tabel di bawah ini:

Tabel 1. Prosedur pendekatan GA *sampling*

Langkah 1	Membuat input data awal secara random
Langkah 2	Temukan semua jalur yang ada pada *flowgraph* dan masukan input data yang

	tadi sudah dibuat secara random
Langkah 3	Pilih input data sebagai anggota dari populasi awal jika *path coverage* yang dimiliki lebih dari 20%
Langkah 4	Cek *fitness function* (cek kriteria *coverage*)
Langkah 5	Jika kriteria *coverage* terpenuhi atau *path coverage* melebihi 95% maka langsung menuju langkah 7
Langkah 6	Jika kriteria coverage belum terpenuhi maka terapkan *genetic algorithm operation (crossover & mutation)* untuk membuat *test case* baru. Lalu kembali ke langkah 4
Langkah 7	Terapkan random sampling

Pada Tabel 1 dapat dilihat bahwa langkah awal dalam mengeksekusi pendekatan GA dengan *sampling* adalah membuat input data awal, lalu langkah kedua adalah untuk memastikan apakah input data yang telah dibuat telah meliputi semua jalur pada *independent path*. Pada prosedur langkah ketiga, input data yang telah meliputi semua jalur pada *independent path* akan dijadikan populasi awal *test case*. Populasi awal harus memiliki *path coverage* diatas 20% (Khan, Amjad, & Srivastava, 2016). *Path coverage* adalah persentase perbandingan antara jalur yang dapat di-*cover* oleh *test case* dengan total jumlah jalur pada *independent path*. Selanjutnya dilakukan pengecekan *fitness function* atau kriteria *coverage*. *Path coverage* diatas 95% akan langsung masuk ke populasi akhir, 85%-95% akan diterapkan *mutation*, dan dibawah 75% akan diterapkan *crossover*. Jika semua set input data telah mencapai *path coverage* diatas 95%, maka akan dilakukan perhitungan *random sampling* untuk menentukan *optimal test suites.*

2.8 Perbandingan algoritme

Perbandingan algoritme dapat digunakan untuk membandingkan *test case generation* pada *search-based software testing* (SBST). Perbandingan dapat dilakukan dengan mengukur *cost* yang harus dikeluarkan dalam mengeksekusi sebuah algoritme SBST. *Cost* dapat diukur dengan mengukur indikator-indikator, yaitu banyaknya iterasi, akumulasi jumlah individual, banyaknya evaluasi *fitness* yang terjadi, dan ukuran dari *test suites* yang dihasilkan.

Number of iteration menunjukkan berapa kali SBST harus melakuakan iterasi untuk mencapai solusi terbaik atau *test case final*. Pada *genetic algorithm* hal ini dapat ditunjukan dengan banyak generasi populasi yang terjadi.

Cumulative number of individuals menunjukkan berapa banyak jumlah individual dihasilkan setiap iterasi. Pada SBST, individual yang dimaksud adalah *test case*.

Number of fitness evaluation menunjukkan berapa kali evaluasi *fitness function* yang dibutuhkan algoritme untuk mendapatkan solusi akhir. Hal ini berdasarkan banyaknya individual atau *test case* baru yang dihasilkan.

Size of the resulting test suites dapat dijadikan pengganti pengukuran *cost* dari waktu yang dibutuhkan untuk menghasilkan *test suites*. Hal ini terjadi karena biasanya semakin besar *test suites* yang dihasilkan akan membutuhkan sumber data yang lebih banyak untuk dieksekusi.

3. IMPLEMENTASI ALGORITME

3.1 Persiapan *sample coding*

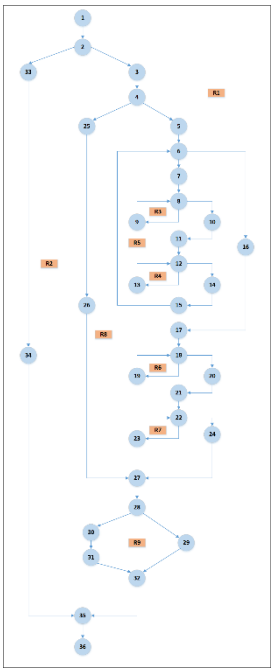
Sample coding yang akan diambil untuk dilakukan pengujian adalah fungsi dalam mengambil detail transaksi bernama *getDetailTransaksi*, Fungsi ini digunakan untuk mengambil data detail transaksi yang telah masuk ke *database* berupa data transaksi, layanan, proses, item, pembayaran, dan rak. Berikut adalah fungsi *getDetailTransaksi* dalam bentuk *pseudocode*:

Tabel 2. *Pseudocode* fungsi *getDetailTransaksi*

```
1   INIT variabel $rb
2   IF check Token
3        INIT variabel array $transaksi
         SET variabel $res_transaksi untuk query
4   mengambil data transaksi
5        IF query memiliki kembalian 1 baris THEN
             SET $transaksi dengan setiap kolom hasil
    kembalian query
6            $res_transaksi
7             SET variabel $res_layanan untuk query
    mengambil data layanan,
              FOR setiap baris pada $res_layanan
                        INIT variabel array $layanan
8                       SET $layanan dengan setiap kolom
9   hasil kembalian query
                  $res_layanan
                        SET variabel $res_item untuk
    query mengambil data item,
10                      FOR setiap baris pada $res_item
11                              INIT variabel array
    $item
12                              SET $item dengan
13  setiap kolom hasil kembalian
                          query $res_layanan
                                PUSH Array $item ke
    variabel $layanan
14                      ENDFOR
15                      SET variabel $res_proses untuk
16  query mengambil data
                  proses
17                      FOR setiap baris pada
    $res_proses
18                              INIT variabel array
19  $proses
                                SET $proses dengan
    setiap kolom hasil kembalian
                          query $res_proses
20                              PUSH Array $proses ke
21  variabel $layanan
22                      ENDFOR
23                      PUSH Array $layanan ke variabel
    $transaksi
             ENDFOR

24           SET variabel $res_pembayaran untuk query
    mengambil data
```

```
25          pembayaran,
26           FOR setiap baris pada $res_pembayaran
                    INIT variabel array $pembayaran
                    SET $pembayaran dengan setiap
27  kolom hasil kembalian
                query $res_pembayaran
28                  PUSH Array $pembayaran ke
29  variabel $transaksi
            ENDFOR
            SET variabel $res_rak untuk query
30  mengambil data rak,
31          FOR setiap baris pada $res_rak
                    INIT variabel array $rak
                    SET $rak dengan setiap kolom
    hasil kembalian query
                $res_rak
32                  PUSH Array $rak ke variabel
33  $transaksi
34          ENDFOR

35      ELSE
            $rb set success = false
36          $rb set message = "Transaksi tidak
37  ditemukan"
            return $rb->getResult();
        ENDIF

38      IF status transaksi query = false THEN
            Transaksi query di-rollback
39          $rb set success = false
40          $rb set message = " Transaksi tidak
    ditemukan"
        ELSE
            Transaksi query di-commit
41          $rb set Row dengan data pada variabel
42  $transaksi
            $rb set success = true
            $rb set message = "Pengambilan data detail
    transaksi berhasil"

        ENDIF
    ELSE
43      $rb set success = false
44      $rb set message = "Token not registered"
45  ENDIF

46  RETURN result dari $rb

47
```

Dari *sample coding* di atas akan dibuat *flowgraph* untuk mempermudah pembacaan algoritme fungsi tersebut. Berikut *flowgraph* dari fungsi *getDetailTransaksi*



Gambar 2. *Flowgraph* fungsi *getDetailTransaksi*

Pada Gambar 2 dapat dilihat bahwa fungsi *getDetailTransaksi* memiliki 36 *nodes*, 43 *edges*, dan 9 *region*. Data tadi akan digunakan untuk menghitung *cyclomatic complexity* yang dimiliki fungsi *getDetailTransaksi*. Hasil perhitungan *cyclomatic complexity* adalah 9. Nilai 9 ini menentukan jumlah jalur *independent path* yang harus dilalui dalam pengujian. Pengujian *test case* mengikuti jalur-jalur *independent path* tentu membutuhkan *test data* atau input yang sesuai. Pembuatan *test data* ini dilakukan agar setiap jalur dapat teruji dan menghasilkan *output* yang diinginkan. Berikut *test data* yang akan digunakan dalam *test case* pengujian fungsi *getDetailTransaksi:*

Tabel 3. *Test data*

Variabel input	No. Input data	Ketersediaan data di database	Input data
Token	1	Aktif	135150400111018
	2	Tidak aktif	135150400111000
idTransaksi	3	Tidak aktif	XVD201701151331TEST
	4	Transaksi saja	XVD201701151331TEST4
	5	Transaksi dan layanan	XVD201701151331TEST5
	6	Transaksi, layanan, item	XVD201701151331TEST6
	7	Transaksi, layanan, item, proses	XVD201701151331TEST7
	8	Transaksi, layanan, item, proses, pembayaran	XVD201701151331TEST8
	9	Transaksi, layanan, item, proses, pembayaran, rak	XVD201701151331TEST9

Pada Tabel 3 dapat dilihat sejumlah input data yang akan diujikan pada setiap jalur pada *independent path*. Variabel terdiri dari token dan idTransaksi. Setiap nilai dari variabel di atas dapat memenuhi semua jalur pada *independent path* yang dijelaskan pada kolom ketersediaan pada *database.*

3.2 GA dengan *mutation analysis*

Test case awal telah dibuat pada subbab persiapan *sample coding*. Langkah selanjutnya dalam prosedur GA dengan *mutation analysis* adalah injeksi *mutant* dan pembuatan *detection matrix*.

Mutant akan diinjeksikan pada program pada baris program yang memiliki unsur percabangan dan perulangan. Operator *mutant* adalah negasi dari operator program yang sebenarnya. Setelah *coding* diinjeksikan *mutant* maka akan dilakukan pengujian kembali

menggunakan *random test case* yaitu *test case* yang telah dihasilkan oleh pengujian *basis path*.

Pada pengujian ini yang dicari adalah apakah *test case* yang ada dapat mendekteksi *mutant* yang telah diinjeksikan. Sebuah *test case* dapat dikatakan berhasil mendeteksi *mutant* apabila *result* dari pengujian awal berbeda dengan *result* dari pengujian fungsi *getDetailTransaksi* setelah diinjeksikan *mutant*. Sebaliknya jika *result* antara sebelum dan sesudah diinjeksikan *mutant* sama, maka *test case* tidak berhasil mendekteksi *mutant* tersebut

Langkah selanjutnya adalah mengeleminasi *test case* yang *redundant* yaitu *test case* yang mendeteksi *mutant* yang sama. Hal ini terus dilakukan hingga tidak ada *test case redundant* di dalam *detection matrix*. Hasil dari eleminasi tadi dapat dilihat pada *detection matrix* di bawah ini:

$$DM = \begin{array}{c} \\ M_1 \\ M_2 \\ M_3 \\ M_4 \\ M_5 \\ M_6 \\ M_7 \\ M_8 \end{array} \begin{array}{c} T_9 \\ \begin{pmatrix} 1 \\ 1 \\ 1 \\ 1 \\ 1 \\ 1 \\ 1 \\ 1 \end{pmatrix} \end{array}$$

Detection matrix akhir di atas membuktikan bahwa jumlah *test case* telah berhasil dikurangi dari 9 menjadi hanya 1 *test case*.

Kualitas dari suatu set *test case* dapat diketahui dengan menghitung *mutation score*(MS) dimana *mutant* yang dideteksi dibagi dengan seluruh jumlah *mutant*. Dapat dilihat pada *detection matrix* akhir bahwa *test case* dapat mendeteksi seluruh *mutant* yang ada sehingga *mutation score* bernilai 100%.

Crossover dilakukan dengan perhitungan untuk mengetahui kualitas dari *test case* yang ada dengan menggunakan perhitungan *evaluation function*(qty). Berikut hasil perhitungan *evaluation function*(qty):

Tabel 4. *Hasil perhitungan evaluation function(qty)*

	T_9	**stg**
M_1	1	1,0
M_2	1	1,0
M_3	1	1,2
M_4	1	1,4
M_5	1	1,6
M_6	1	2,1
M_7	1	3,3
M_8	1	1,2
qty	12,8	

Test case yang memiliki nilai qty tinggi akan digunakan sebagai *test case set* yang baru. Namun dapat dilihat pada Tabel 4, karena hanya dilakukan perhitungan pada satu *test case* maka *test case* T_9 akan menjadi anggota tunggal *test case set* yang baru.

Hasil akhir dari algoritme penerepan *genetic algorithm* dengan *mutation analysis* adalah *test case* berjumlah 1 buah yaitu: {135150400111018, XVD201701151331TEST9}.

3.3 GA dengan *sampling*

Penerapan *genetic algorithm* dengan *random sampling* diawali dengan membuat input data yang dapat meliputi semua jalur pada *independent path*. Hal ini sudah dikerjakan pada subbab persiapan *sample coding* dan akan dijadikan populasi awal *test case*.

Populasi awal harus memiliki *path coverage* diatas 20% (Khan, Amjad, & Srivastava, 2016). *Path coverage* adalah persentase perbandingan antara jalur yang dapat di-*cover* oleh *test case* dengan total jumlah jalur pada *independent path*. Populasi yang memiliki *path coverage* diatas 20% dapat dilihat pada tabel di bawah ini

Tabel 5. Populasi awal

No set	Set of input	Random Test Cases	Path	Path Coverage
1	5	[135150400111000, XVD201701151331TEST7], [135150400111018, XVD201701151331TEST], [135150400111018, XVD201701151331TEST4], [135150400111018, XVD201701151331TEST5], [135150400111018, XVD201701151331TEST6],	1,2,3,4,5,6	66%
2	5	[135150400111000, XVD201701151331TEST9], [135150400111018, XVD201701151331TEST], [135150400111018, XVD201701151331TEST6], [135150400111018, XVD201701151331TEST7], [135150400111018, XVD201701151331TEST8],	1,2,3,4,5,6,7,8	88%
3	5	135150400111000, XVD201701151331TEST5 & 135150400111018, XVD201701151331TEST & 135150400111018, XVD201701151331TEST4 & 135150400111018, XVD201701151331TEST8 & 135150400111018, XVD201701151331TEST9	1,2,34,5,6,7,8,9	100%

Penerapan *fitness function* yang bertugas untuk mengevaluasi populasi atau set pada pendekatan *genetic algorithm* telah direpresentasikan dengan perhitungan terhadap *path coverage* setiap set telah dilakukan. Hasilnya adalah set 1 harus dilakukan *crossover* agar daya lingkup jalurnya dapat meningkat.

Hasil dari *crossover* dapat dilihat pada tabel di bawah ini

Tabel 6. *Crossover* set 1

Test case	Sebelum	Crossover (test case)	Sesudah
1	135150400111000, XVD201701151331TEST7	1 dengan 5	135150400111000, XVD201701151331TEST6
2	135150400111018, XVD201701151331TEST	2 dengan 1	135150400111018, XVD201701151331TEST7
3	135150400111018, XVD201701151331TEST4	3 dengan 2	135150400111018, XVD201701151331TEST
4	135150400111018, XVD201701151331TEST5	4 dengan 3	135150400111018, XVD201701151331TEST4
5	135150400111018, XVD201701151331TEST6	5 dengan 4	135150400111018, XVD201701151331TEST5

Pada Tabel 6 dapat dilihat *crossover* dilakukan dengan cara sing*le point crossover* yaitu sebagian dari *parent* 1 akan digabung dengan sebagian *parent* 2. *Parent* yang dimaksud adalah *test case* anggota dari set 1 dan *crossover* akan mempertemukan satu *parent* dengan *parent* lain untuk menghasilkan *test case* yang baru.

Path coverage pada set 2 dengan hasil 88% membuktikan bahwa diperlukan proses *mutation* untuk menaikan persentase *path coverage* yang dimiliki. Hasil dari proses *mutation* pada set 2 dapat dilihat pada tabel di bawah ini

Tabel 7. *Mutation* set 2

Sebelum	Mutation	Sesudah
135150400111000 XVD201701151331TEST9	135150400111000 XVD201701151331TEST9	135150400111018 XVD201701151331TEST9
135150400111018 XVD201701151331TEST	135150400111018 XVD201701151331TEST	135150400111000 XVD201701151331TEST
135150400111018 XVD201701151331TEST6	135150400111018 XVD201701151331TEST6	135150400111018 XVD201701151331TEST
135150400111018 XVD201701151331TEST7	135150400111018 XVD201701151331TEST7	135150400111018 XVD201701151331TEST5
135150400111018 XVD201701151331TEST8	135150400111018 XVD201701151331TEST8	135150400111000 XVD201701151331TEST8

Setelah dilakukan *mutation*, set 2 telah mencapai *path coverage* 100%. Status set 2 naik menjadi *satisfied* karena telah melewati angka 95% dan langsung masuk ke populasi akhir. Set 1 memiliki *path coverage* 77% sehingga akan dilakukan *mutation* untuk meningkatkan persentase *path coverage* yang dimiliki.

Hasil akhir setelah *mutation* dilakukan dapat dilihat pada tabel di bawah ini:

Tabel 8. Populasi final

No	Set of input	Random Test Cases	Path	Path Coverage
1	5	[135150400111000, XVD201701151331TEST8], [135150400111018, XVD201701151331TEST], [135150400111018, XVD201701151331TEST9], [135150400111018, XVD201701151331TEST7], [135150400111018, XVD201701151331TEST6]	1,2,3,4,5,6,7,8,9	100%
2	5	[135150400111018, XVD201701151331TEST9], [135150400111000, XVD201701151331TEST], [135150400111018, XVD201701151331TEST], [135150400111018, XVD201701151331TEST5], [135150400111000, XVD201701151331TEST8]	1,2,3,4,5,6,7,8,9	100%
3	5	[135150400111000, XVD201701151331TEST5], [135150400111018, XVD201701151331TEST], [135150400111018, XVD201701151331TEST4], [135150400111018, XVD201701151331TEST8], [135150400111018, XVD201701151331TEST9]	1,2,34,5,6,7,8,9	100%

Pada Tabel 8 dapat dilihat bahwa semua set telah memiliki *path coverage* 100% dengan status *satisfied.* Hal ini menunjukan bahwa penerapan *genetic algorihm* lewat *crossover* dan *mutation* untuk menghasilkan *test case* yang optimal telah selesai. Hasil ini akan dilanjutkan dengan proses *random sampling* untuk menghasilkan *optimal test suites.*

Rumus random sampling terdiri dari jumlah *test case* hasil *genetic algorithm* dan *cyclomatic complexity* dari *flow graph* fungsi yang diuji. *Jumlah test case* hasil *genetic algorithm* adalah 10 dan *cyclomatic complexity flow graph* yang diuji adalah 9. *Random sampling* menghasilkan nilai 10, artinya ada 10 kemungkinan *optimal test suites* yang mungkin didapat.

Setelah melakukan eleminasi terhadap *test suite* yang tidak optimal atau tidak meliputi seluruh jalur *independent path*, didapat hasil akhir *test suit* seperti tabel di bawah ini

Tabel 9. *Optimal test suites*

Test suites 1	135150400111018, XVD201701151331TEST	135150400111018, XVD201701151331TEST9	135150400111000, XVD201701151331TEST5
Test suites 2	135150400111000, XVD201701151331TEST8	135150400111018, XVD201701151331TEST	135150400111018, XVD201701151331TEST9

Tabel 9 menunjukan *test suites* optimal yang telah melingkupi seluruh jalur pada *independent path*. Hasil akhir dari algoritme penerepan *genetic algorithm* dengan *random sampling* adalah *test case* berjumlah 4 buah yaitu:

135150400111000, XVD201701151331TEST5
135150400111000, XVD201701151331TEST
135150400111000, XVD201701151331TEST8
135150400111018, XVD201701151331TEST9

4. PEMBAHASAN

4.1 Penerapan algoritme

Penerapan algoritme *genetic algorithm* dengan *mutation analysis* memiliki hasil akhir yaitu 1 *test case*. Penerapan algoritmr *genetic algorithm* dengan *sampling* menghasilkan 4 *test case*

Sebelum menerapkan kedua algoritme di atas, penulis telah melakukan pengujian awal menggunakan *basis path testing* untuk memenuhi langkah awal pada masing-masing algpritme.

Data-data di atas menunjukan bahwa dengan menerapkan pendekatan *genetic algorithm* dengan *mutation analysis* mengakibatkan pengurangan jumlah *test case* sebanyak 8 buah dari pengujian awal *basis path testing*. Penerapan pendekatan *genetic algorithm* dengan *sampling* juga mengakibatkan pengurangan jumlah *test case*, yaitu sebanyak 5 buah dari pengujian awal *basis path testing*.

4.2 Perbandingan algoritme

Perbandingan algoritme dapat dilakukan dengan cara mengukur *cost* dalam menegeksekusi algoritme yang dibandingkan. Pengukuran *cost* pada GA dengan *mutation analysis* dan GA dengan *sampling* dapat dilihat pada tabel di bawah ini:

Tabel 10. Hasil perbandingan algoritme

Algoritme	**Cost of finding the target**			**Cost of executing generated test case**
	Number of iteraion	Cumulative number of individual	Number of fitness evaluation	Size of test suites
GA mutation analysis	1	1	1	1
GA sampling	3	36	19	2

4.3 Analisis perbandingan

GA dengan *mutation analysis* menggungguli GA dengan *sampling* di semua aspek. Hal ini dibuktikan dengan semua angka yang merepresentasi *cost* pada baris GA dengan *mutation analysis* lebih kecil dibanding GA dengan *sampling*.

Variabel *number of iteration*, GA dengan *mutation analysis*, hasil iterasi pertama telah menunjukan *test case* yang sangat berkualitas yang dapat dilihat pada *mutation score*. Pada GA dengan *sampling* harus dilakukan iterasi lebih dari sekali untuk mendapatkan *path coverage* di atas 95%

Variabel *cumulative number of individual*, GA dengan *mutation analysis* memiliki iterasi yang lebih sedikit dibandingkan dengan GA dengan *sampling*. Pada setiap iterasi pasti akan membutuhkan individu. Hal ini mengakibatkan semakin banyak iterasi maka semakin banyak juga jumlah individu yang akan terbentuk.

Variabel *number of fitness evaluation*, GA dengan *mutation analysis* memiliki lebih sedikit iterasi dibandingkan dengan GA dengan *sampling*, maka akan lebih sedikit pula evaluasi *fitness* yang dimiliki GA dengan *mutation analysis.*

Variabel *size of test suites*, ukuran test suites berdasarkan jumlah *test case* akhir yang dihasilkan oleh masing-masing algoritme. GA dengan *mutation analysis* hanya memiliki satu *test case* akhir sehingga memiliki *test suites* yang kecil pula dibandingkan GA dengan *sampling.*

5. KESIMPULAN

Secara keseluruhan berdasarkan penelitian yang dilakukan penulis dapat ditarik kesimpulan sebagai berikut

1. Pengujian perangkat lunak dengan pendekatan *genetic algorithm* dalam *test case generation* sangat membantu dalam membuat *test case* efektif dan berkualitas. *Basis path testing* membutuhkan 9 *test case* untuk memenuhi kualitas test case yang baik. Hal ini menunjukan bahwa terjadi pengurangan yang cenderung signifikan terhadap jumlah *test case*. GA dengan *mutation analysis* membuat pengurangan jumlah *test case* awal dengan *test case* akhir sebanyak 8 *test case*, sedangkan GA dengan *sampling* membuat pengurangan jumlah *test case* awal dengan *test case* akhir sebanyak

5 *test case*.

2. Penelitian yang dilakukan penulis juga berhasil membandingkan kedua *pendekatan genetic algorithm* yang telah digunakan. Variabel banyaknya iterasi, GA dengan *mutation analysis* hanya melakukan 1 kali iterasi sedangkan GA dengan *sampling* membutuhkan 3 kali iterasi. Variabel akumulasi individual, GA dengan *mutation analysis* hanya menghasilkan 1 individual dalam iterasi yang dilakukan sedangkan GA dengan *sampling* menghasilkan 36 individual dalam seluruh iterasi yang dilakukan. Variabel banyaknya evaluasi *fitness* yang terjadi, GA dengan *mutation analysis* menghasilkan 1 *test case* baru sedangkan GA dengan *sampling* menghasilkan 19 *test case* baru. Variabel ukuran dari *test suites* yang dihasilkan, GA dengan mutation analysis menghasilkan ukuran test suites yang kecil yaitu 1 buah *test suites* dengan anggota 1 *test case*, sedangkan GA dengan *sampling* menghasilkan 2 buah *test suites* dengan anggota masing-masing 3 *test case*. Sesuai dengan penelitian lain yang telah dituliskan pada latar belakang, pengurangan jumlah *test case* dengan pembuatan *test case* secara otomatis ini berdampak pada waktu berkurangnya waktu pengujian perangkat lunak yang harus dilakukan.
3. Perhitungan pada *cost measure* pada perbandingan antara GA *mutation analysis* dibandingkan GA *sampling* menunjukan GA *mutation analysis* lebih baik dibandingkan dengan GA *sampling* di semua variabel pembanding. Hal ini dapat dijelaskan dengan mengetahui alasan hasil perbandingan pada GA dengan mutation analysis yang telah dijelaskan pada poin 2. Pada variabel *number of iteration*, hasil iterasi pertama telah menunjukan *test case* yang sangat berkualitas. Pada variabel *cumulative number of individual*, iterasi yang lebih sedikit mengakibatkan sedikit jumlah individu yang akan terbentuk. Pada variable *number of fitness evaluation*, iterasi yang lebih sedikit mengakibatkan sedikit pula jumlah evaluasi *fitness* yang harus dilakukan. Pada variabel *size of test suites*, jumlah *test case* akhir yang sedikit mengakibatkan ukuran *test suites* yang kecil. Maka pada GA dengan *mutation analysis* akan diasumsikan memiliki waktu eksekusi yang lebih cepat walaupun dijalankan pada data *sample* yang lebih besar karena pada penelitian yang dilakukan peneliti, GA dengan mutation analysis memiliki ukuran *test suite* yang kecil yang berbanding lurus dengan waktu pengeksekusian algoritme. Hasil pada poin satu dan tiga ini dapat meyelesaikan permasalahan tentang waktu pengujian perangkat lunak yang sangat besar dengan membantu organisasi di bidang teknologi informasi memilih metode pengujian yang akan digunakan. Penelitian ini dapat digunakan dan dimanfaatkan oleh organisasi yang bergerak di bidang teknologi informasi untuk meningkatkan efektifitas dan efisiensi dalam pengujian perangkat lunak dengan mempersingkat waktu yang harus dialokasikan untuk pengujian yang dilakukan.

6. DAFTAR PUSTAKA

Ali, S., Briand, L. C., Hemmati, H. & Panesar-Walawege, R. K., 2010. A Systematic Review of the Application Empirical Investigation of Search-Based Test Case Generation. *IEEE Transaction on Software Engineering, Vol. 36,* pp. 742 - 762.

Ansari, A., Shagufta, M. B., Fatima, A. S. & Tehreem, S., 2017. Constructing Test Case Using Natural Language Processing. *International Conference on Advances in Electrical, Electronics, Information, Communication and Bio-Informatics (AEEICB17)*, Volume 3, pp. 95-99.

Galin, D., 2014. *Software Quality Assurance.* Harlow: Pearson Education Limited.

Ghiduk, A. S. & Girgis, M. R., 2010. Using Genetic Algorithm and Dominance Concepts for Generating Reduced Test Data. *Informatica 34*, pp. 377-385.

Gupta, N. K. & Rohil, M. K., 2008. Using Genetic Algorithm for Unit Testing of Object Oriented Software. *First International Conference on Emerging Trends in Engineering and Technology*, pp. 308-313.

Haga, H. & Suehiro, A., 2012. Automatic Test Case Generation based on Genetic Algorithm and Mutation Analysis. *IEEE International Conference on Control System, Computing and*

Engineering, pp. 119-123.

Homès, B., 2012. *Fundamentals of Software Testing*. London: ISTE Ltd.

Khan, R. & Amjad, M., 2015. Automatic Test Case Generation fot Unit Software Testing Using Genetic Algorithm and Mutation Analysis. *IEEE UP Section Conference on Electrical Computer and Electronics (UPCON)*, pp. 1-5.

Khan, R., Amjad, M. & Srivastava, A. K., 2016. Optimization of Automatic Generated Test Cases for Path Testing Using Genetic Algorithm. *2016 Second International Conference on Computational Intelligence & Communication Technology*, pp. 32-36.

Mohapatra, D., Bhuyan, P. & Mohaptra, D. P., 2009. Automated Test Case Generation and Its Optimization fot Path Testing Using Genetic Algorithm and Sampling. *WASE International Conference on Information Engineering*, pp. 643-646.

Sommerville, I., 2011. *Software Engineering*. Boston: Addison-Wesley.