PENGEMBANGAN *LIBRARY GENETIC ALGORITHM* MENGGUNAKAN GCC *COMPILER* DENGAN PENDEKATAN *PROCEDURAL PROGRAMMING*

SKRIPSI

Diajukan untuk memenuhi persyaratan
memperoleh gelar Sarjana Teknik

Disusun Oleh :

NORMAN NATANAEL

NIM. 0610630073 - 63

KEMENTERIAN PENDIDIKAN DAN KEBUDAYAAN

UNIVERSITAS BRAWIJAYA

FAKULTAS TEKNIK

MALANG

2012

LEMBAR PERSETUJUAN

PENGEMBANGAN *LIBRARY GENETIC ALGORITHM* MENGGUNAKAN GCC *COMPILER* DENGAN PENDEKATAN *PROCEDURAL PROGRAMMING*

SKRIPSI

Diajukan untuk memenuhi persyaratan
memperoleh gelar Sarjana Teknik

Oleh :
NORMAN NATANAEL
0610630073 - 63

Telah diperiksa dan disetujui oleh :

Dosen Pembimbing I	Dosen Pembimbing II
Hadi Suyono, ST., MT., Ph.D.	**Adharul Muttaqin, ST., MT.**
NIP. 19730520 200801 1 013	**NIP. 19760121 200501 1 001**

PENGANTAR

Segala puji syukur bagi Tuhan Yesus Kristus, atas kasih karunia-Nya sehingga penulis dapat menyelesaikan Tugas Akhir dengan judul "**Pengembangan *Library Genetic Algorithm* Menggunakan GCC *Compiler* dengan Pendekatan *Procedural Programming***". Tugas Akhir ini disusun untuk memenuhi sebagian persyaratan memperoleh gelar Sarjana Teknik di Jurusan Teknik Elektro Program Studi Rekayasa Komputer Fakultas Teknik Universitas Brawijaya Malang.

Tidak banyak yang bisa penulis sampaikan kecuali ungkapan terima kasih kepada berbagai pihak yang telah dengan tulus ikhlas memberikan bimbingan, arahan, dan dukungan hingga penulisan tugas akhir ini dapat terselesaikan. Pada kesempatan kali ini, penulis mengucapkan banyak terima kasih kepada:

1. Ayahanda dan Ibunda tercinta, juga kak Kiki, Jeje, serta seluruh keluarga yang senantiasa memberikan doa dan restu demi terselesaikannya skripsi ini.
2. Bapak Sholeh Hadi Pramono, DR., Ir., MS. selaku Ketua Jurusan Teknik Elektro, Fakultas Teknik Universitas Brawijaya.
3. Bapak M. Azis Muslim, ST., MT., Ph.D selaku Sekretaris Jurusan Teknik Elektro, Fakultas Teknik Universitas Brawijaya.
4. Bapak Moch. Rif'an. ST., MT. selaku Ketua Program Studi Jurusan Teknik Elektro, Fakultas Teknik Universitas Brawijaya.
5. Bapak Waru Djuriatno, ST. MT. selaku Ketua Kelompok Dosen Keahlian Rekayasa Komputer Jurusan Teknik Elektro Universitas Brawijaya.
6. Bapak Hadi Suyono, ST., MT., Ph.D. selaku dosen pembimbing I yang telah banyak memberikan bimbingan, masukan dan arahan dalam penyusunan tugas akhir ini.
7. Bapak Adharul Muttaqin, ST., MT. selaku dosen pembimbing II dan dosen pembimbing akademik yang telah bersedia memberikan bimbingan, masukan dan arahan dalam penyusunan tugas akhir ini maupun selama penulis dalam pendampingan akademis di kampus.
8. Bapak dan Ibu Dosen, Staff Administrasi Jurusan Teknik Elektro Fakultas Teknik Universitas Brawijaya.
9. Eka, Didit, Adityo, Tito, Abdur, Jumadil, Felix, mas Prima, mas Diriga, mas Runi, Krisna, juga semua teman-teman asisten, serta Ka. Lab dan Laboran dari

Laboratorium Komputasi dan Jaringan yang telah memberikan banyak bantuan dan dukungan selama ini.

10. Saudara Angga dan Aflah, atas bantuan dan dukungan dalam menyelesaikan tugas akhir ini.
11. Rekan-rekan SuperYouth MP3 dan Musik Pujian Bethany Malang atas segala bantuan, inspirasi, serta motivasi yang diberikan.
12. Teman-teman angkatan 2006 (Ge-Force), khususnya paket E 06 dan anggota RisTIE 2009-2010.
13. Semua pihak yang tidak dapat penulis sebutkan satu per satu yang terlibat baik secara langsung maupun tidak langsung demi terselesaikannya tugas akhir ini.

Penulis menyadari sepenuhnya bahwa Tugas Akhir ini masih banyak kekurangan. Segala kritik dan saran yang membangun akan penulis terima demi kesempurnaan skripsi ini. Harapan penulis semoga skripsi ini bermanfaat bagi pembaca dan khususnya mahasiswa jurusan teknik elektro dimasa yang akan datang.

Malang, Februari 2012

Penulis

DAFTAR ISI

PENGANTAR ..... **i**
DAFTAR ISI ..... **iii**
DAFTAR TABEL ..... **vii**
DAFTAR GAMBAR ..... **viii**
ABSTRAK ..... **x**
I PENDAHULUAN **1**
1.1 Latar Belakang 1
1.2 Rumusan Masalah 2
1.3 Batasan Masalah 2
1.4 Tujuan 3
1.5 Manfaat 3
1.6 Sistematika Penulisan 3
II TINJAUAN PUSTAKA **5**
2.1 Komputasi Evolusioner 5
2.1.1 Pengantar Komputasi Evolusioner 5
2.1.2 Keunggulan Komputasi Evolusioner 7
2.1.2.1 Kesederhanaan Konseptual 7
2.1.2.2 Menyelesaikan Masalah Yang Tidak Memiliki Solusi 8
2.2 Algoritma Genetika 8
2.2.1 Parameter-parameter Algoritma Genetika 9
2.2.1.1 Ukuran Populasi 10
2.2.1.2 Jumlah Generasi 10
2.2.1.3 Probabilitas *Crossover* 10
2.2.1.4 Probabilitas Mutasi 11
2.2.2 Siklus Algoritma Genetika 11
2.2.3 Komponen-komponen Utama Algoritma Genetika …..... 11
2.2.3.1 Pengkodean Kromosom 12
2.2.3.2 Pembangkitan Populasi Awal 12
2.2.3.3 Nilai *Fitness* 13
2.2.3.4 Seleksi 13
2.2.3.4.1 *Rank Selection* 13

2.2.3.4.2 *Roulette Wheel Selection* 14
2.2.3.4.3 *Tournament Selection* 14
2.2.3.5 *Crossover* (Pindah-silang) 15
2.2.3.5.1 *Crossover* 1 titik (*Single Point Crossover*) 15
2.2.3.5.2 *Crossover* 2 titik (*Two Point Crossover*) 15
2.2.3.5.3 *Crossover* Seragam (*Uniform Crossover*) 16
2.2.3.5.4 *Arithmatic Crossover* 16
2.2.3.5.5 *Crossover* Pengkodean Permutasi 16
2.2.3.6 Mutasi 17
2.3 Rekayasa Perangkat Lunak 17
2.3.1 *Library* 17
2.3.1.1 *Static Library* 19
2.3.2 Pendekatan *Procedural Programming* 20
2.3.2.1 Prosedur 20
2.3.2.1 Jenis Variabel Dalam Prosedur 21
2.3.3 GCC *Compiler* 22
III METODE PENELITIAN 24
3.1 Studi Literatur 24
3.2 Analisis Kebutuhan 24
3.3 Perancangan 24
3.4 Implementasi 24
3.5 Pengujian 24
3.6 Kesimpulan dan Saran 25
IV PERANCANGAN DAN IMPLEMENTASI 26
4.1 Sasaran Perancangan *Library* 26
4.2 Kebutuhan Sistem 26
4.3 Perancangan Sistem *Library* Secara Global 27
4.3.1 Cara Kerja *Library* `GASLib.a` 27
4.3.2 Perancangan Sistem Algoritma Genetika 29
4.4 Perancangan Terinci 30
4.4.1 Perancangan Prosedur Pembacaan Parameter 30
4.4.2 Perancangan Prosedur Pemberian Nilai Awal 31
4.4.3 Perancangan Prosedur Pembangkit Nilai Acak 33

4.4.4 Perancangan Prosedur Penyimpan Nilai Terbaik 35
4.4.5 Perancangan Prosedur *Elitist* 36
4.4.6 Perancangan Prosedur Seleksi 38
4.4.7 Perancangan Prosedur *Crossover* 40
4.4.8 Perancangan Prosedur *Parental Crossover* 42
4.4.9 Perancangan Prosedur *Swapper* 43
4.4.10 Perancangan Prosedur Mutasi 44
4.4.11 Perancangan Prosedur Pembuat Laporan 46
4.4.11.1 Perancangan Prosedur Pembuat Kepala Laporan 46
4.4.11.2 Perancangan Prosedur Pembuat Badan Laporan 47
4.4.11.2 Perancangan Prosedur Pembuat Kesimpulan Laporan 47
4.5 Implementasi 48
4.5.1 Lingkungan Implementasi 48
4.5.1.1 Lingkungan Implementasi Perangkat Keras 49
4.5.1.2 Lingkungan Implementasi Perangkat Lunak 49
4.5.2 Implementasi Prosedur Library 49
4.5.2.1 Implementasi Prosedur Pembacaan Parameter 49
4.5.2.2 Implementasi Prosedur Pemberian Nilai Awal 51
4.5.2.3 Implementasi Prosedur Pembangkit Nilai Acak 52
4.5.2.4 Implementasi Prosedur Penyimpan Nilai Terbaik 53
4.5.2.5 Implementasi Prosedur *Elitist* 54
4.5.2.6 Implementasi Prosedur Seleksi 55
4.5.2.7 Implementasi Prosedur *Crossover* 57
4.5.2.8 Implementasi Prosedur *Parental Crossover* 58
4.5.2.9 Implementasi Prosedur *Swapper* 59
4.5.2.10 Implementasi Prosedur Mutasi 60
4.5.2.11 Implementasi Prosedur Pembuat Laporan 61
4.5.2.11.1 Implementasi Prosedur Pembuat Kepala Laporan 61
4.5.2.11.2 Implementasi Prosedur Pembuat Badan Laporan 61
4.5.2.11.3 Implementasi Prosedur Pembuat Kesimpulan Laporan 63

4.5.3 Implementasi Program Perantara 64
V PENGUJIAN 65
5.1 Validasi *Library* 65
5.1.1 Kasus Uji 1 65
5.1.2 Kasus Uji 2 74
5.2 Analisis Kompleksitas Algoritma 76
5.2.1 Algoritma pada Prosedur Pembangkit Nilai Acak 77
5.2.2 Algoritma pada Prosedur Pemberian Nilai Awal 78
5.2.3 Algoritma pada Prosedur Elitist 80
5.2.4 Algoritma pada Prosedur Seleksi 82
5.2.5 Algoritma pada Prosedur *Crossover* 83
5.2.5 Algoritma pada Prosedur Mutasi 85
5.3 Pengujian *Execution Time* 86
VI PENUTUP 89
6.1 Kesimpulan 89
6.2 Saran 90
DAFTAR PUSTAKA 91

DAFTAR TABEL

Tabel 5.1 Hasil implementasi GASLib pada kasus uji 1 73
Tabel 5.2 Nilai parameter yang digunakan untuk pengujian *execution time* 87
Tabel 5.3 Spesifikasi perangkat keras komputer dan perangkat lunak pada pengujian *execution time* 88
Tabel 5.4 Hasil pengujian *execution time* 88

DAFTAR GAMBAR

Gambar 2.1	Teknik Pencarian	6
Gambar 2.2	Diagram alir dari Algoritma Evolusioner	7
Gambar 2.3	Siklus algoritma genetika	11
Gambar 2.4	*Roulette Wheel Selection*	14
Gambar 2.5	Contoh *Single Point Crossover*	15
Gambar 2.6	Contoh *Two Point Crossover*	16
Gambar 2.7	Contoh *Uniform Crossover*	16
Gambar 2.8	Contoh *Arithmatic Crossover*	16
Gambar 2.9	Proses *Compile-Linking* dalam C	18
Gambar 4.1	*Flowchart* prosedur pembacaan parameter	31
Gambar 4.2	*Flowchart* prosedur pemberian nilai awal	32
Gambar 4.3	*Flowchart* prosedur pembangkit nilai acak	34
Gambar 4.4	*Flowchart* prosedur penyimpan nilai terbaik	35
Gambar 4.5	*Flowchart* prosedur *elitist*	37
Gambar 4.6	*Flowchart* prosedur seleksi	39
Gambar 4.7	*Flowchart* prosedur *crossover*	41
Gambar 4.8	*Flowchart* prosedur *parental crossover*	42
Gambar 4.9	*Flowchart* prosedur *swapper*	43
Gambar 4.10	*Flowchart* prosedur mutasi	45
Gambar 4.11	*Flowchart* prosedur pembuat kepala laporan	46
Gambar 4.12	*Flowchart* prosedur pembuat badan laporan	47
Gambar 4.13	*Flowchart* prosedur pembuat kesimpulan laporan	48
Gambar 4.14	Deskripsi prosedur pembacaan parameter	50
Gambar 4.15	Deskripsi prosedur pemberian nilai awal	51
Gambar 4.16	Deskripsi prosedur pembangkit nilai acak	52
Gambar 4.17	Deskripsi prosedur penyimpan nilai terbaik	53
Gambar 4.18	Deskripsi prosedur *elitist*	55
Gambar 4.19	Deskripsi prosedur seleksi	56
Gambar 4.20	Deskripsi prosedur *crossover*	57
Gambar 4.21	Deskripsi prosedur *parental crossover*	58
Gambar 4.22	Deskripsi prosedur *swapper*	59

Gambar 4.23	Deskripsi prosedur mutasi	60
Gambar 4.24	Deskripsi prosedur pembuat kepala laporan	61
Gambar 4.25	Deskripsi prosedur pembuat badan laporan	62
Gambar 4.26	Deskripsi prosedur pembuat kesimpulan laporan	63
Gambar 5.1	Grafik fungsi kasus uji 1	65
Gambar 5.2	*Project Options* pada Bloodshed Dev-C++	69
Gambar 5.3	Isi dari *file* `boundlist.txt` untuk kasus uji 1	70
Gambar 5.4	*Console display* saat program dijalankan	71
Gambar 5.5	Isi bagian awal file `galist.txt` pada eksekusi pertama kasus uji 1	71
Gambar 5.6	Isi bagian akhir dari file `galist.txt` pada eksekusi pertama kasus uji 1	72
Gambar 5.7	Grafik perubahan nilai *fitness* terbaik selama 10 kali percobaan pengujian dan perbandingannya	73
Gambar 5.8	Isi dari *file* `boundlist.txt` untuk kasus uji 2	75
Gambar 5.9	Isi dari *file* `galist.txt` untuk kasus uji 2	76
Gambar 5.10	*Flowchart* prosedur pembangkit nilai acak	77
Gambar 5.11	*Flowchart* prosedur pemberian nilai awal	79
Gambar 5.12	*Flowchart* prosedur *elitist*	81
Gambar 5.13	*Flowchart* prosedur seleksi	82
Gambar 5.14	*Flowchart* prosedur *crossover*	84
Gambar 5.15	*Flowchart* prosedur mutasi	85

ABSTRAK

NORMAN NATANAEL. 2012: Pengembangan *Library Genetic Algorithm* Menggunakan GCC *Compiler* dengan Pendekatan *Procedural Programming*. Skripsi Jurusan Teknik Elektro, Fakultas Teknik, Universitas Brawijaya. Dosen Pembimbing: Hadi Suyono, ST., MT., Ph.D. dan Adharul Muttaqin, ST., MT.

Genetic Algorithm merupakan pendekatan komputasional untuk menyelesaikan suatu masalah dengan memodelkan masalah tersebut menjadi proses evolusi biologis. Algoritma genetika bekerja berdasarkan set solusi dengan memunculkan sifat-sifat baru melalui proses biologis berupa persilangan dan mutasi. Sifat yang dianggap baik atau kuat akan terus bertahan hidup melalui proses seleksi alam. Solusi yang dianggap terbaik adalah set *genotype* dengan nilai *fitness* tertinggi. *Library Genetic Algorithm* merupakan perangkat lunak yang dapat digunakan seorang *programmer* untuk membantu penyelesaian masalah menggunakan algoritma genetika. Perancangan dan implementasi *library* ini dilakukan dengan pendekatan *procedural programming*. *Library* GASLib ditulis menggunakan bahasa pemrograman C dan dibangun dengan GNU GCC *compiler*. *Library* ini berisi sekelompok prosedur untuk *population-handling* dan operasi genetika, juga beberapa prosedur pembantu untuk operasi *non-genetic*.

Pengujian *Library Genetic Algorithm* dilakukan untuk memastikan *library* berjalan sesuai dengan fungsi yang ditentukan dan benar perhitungannya. Pengujian dilakukan dengan membandingkan perhitungan sebuah aplikasi yang dihubungkan secara *static-link* dengan GASLib, dengan hasil menggunakan metode konvensional untuk kasus uji tertentu. Dari kasus uji 1 diperoleh nilai *fitness* terbaik yang konsisten untuk nilai parameter generasi maksimum sebesar 50.000 generasi, sedangkan pada kasus uji 2 didapatkan perbedaan nilai variabel solusi sebesar 0,2% untuk x_1 dan 2,5% untuk x_2. Waktu eksekusi dipengaruhi secara signifikan oleh parameter berupa jumlah populasi dan jumlah generasi. Semakin tinggi nilai parameternya, maka proses eksekusi akan berlangsung lebih lama.

Kata kunci : *Library*, pustaka, *procedural programming*, algoritma genetika

BAB I
PENDAHULUAN

1.1 Latar Belakang

Genetic Algorithm merupakan pendekatan komputasional untuk menyelesaikan suatu masalah dengan memodelkan masalah tersebut menjadi proses evolusi biologis. Alam memiliki proses untuk seleksi individu berdasarkan kemampuannya beradaptasi dengan perubahan lingkungan, yang memacu setiap spesies untuk dapat merubah sifat-sifat yang dianggap kurang baik dan memunculkan gen-gen baru pada generasi selanjutnya agar tidak punah. Hal inilah yang mendasari terjadinya proses evolusi biologis pada alam. Hanya yang kuat akan bertahan hidup dan berpeluang untuk melestarikan jenisnya melalui siklus panjang reproduksi dari generasi ke generasi. *Genetic algorithm* memodelkan set solusi dari sebuah permasalahan, dan memacu solusi-solusi tersebut untuk terus berevolusi agar tercipta set solusi yang lebih baik pada generasi-generasi berikutnya yang diharapkan dapat memberikan hasil paling optimal secara stokastik.

Salah satu aplikasi algoritma genetika adalah pada permasalahan optimasi kombinasi, yaitu memperoleh suatu nilai solusi optimal terhadap suatu permasalahan yang memiliki banyak kemungkinan solusi. Secara garis besar tahapan dalam algoritma genetika ini dimulai dengan menetapkan suatu set solusi yang potensial dan melakukan perubahan dengan beberapa iterasi untuk mencapai solusi terbaik. Set solusi potensial ini ditetapkan di awal secara acak sebagai *sample* populasi yang direpresentasikan dengan *genotype* yang merupakan bagian paling penting dari kromosom tempat menyimpan sifat dari suatu individu. Keseluruhan set dari *genotype* ini mewakili suatu populasi. Selanjutnya, kromosom-kromosom ini akan berevolusi dalam beberapa kali iterasi yang disebut dengan generasi. Generasi baru (*offspring*) dibangkitkan melalui proses pindah-silang (*crossover*) dan mutasi. Sifat-sifat dari *offspring* ini berevolusi dengan suatu kriteria kesesuaian (*fitness*) yang telah ditetapkan dimana hasil terbaik akan dipilih sedangkan yang lainnya diabaikan. Sifat-sifat yang dianggap baik dari individu akan terus beradaptasi menjadi lebih baik sesuai *rating* dari nilai *fitness*.

Untuk mempermudah penggunaan algoritma genetika dalam mencari solusi optimal dari suatu fungsi, diperlukan suatu perangkat lunak yang dapat membantu pengguna dalam menentukan solusi optimal berdasarkan metode optimasi tersebut. Diharapkan dengan implementasi perangkat lunak ini, pengguna dapat memiliki kemudahan untuk penyelesaian sebuah kasus tertentu menggunakan algoritma genetika.

1.2 Rumusan Masalah

Berdasarkan latar belakang yang telah dipaparkan di atas, maka rumusan masalah ditekankan pada:

1. Bagaimana merancang perangkat lunak berupa *library* yang mengimplementasikan algoritma genetika dengan fungsi obyektif dan parameter diberikan oleh *user library*.
2. Bagaimana mengimplementasikan *library genetic algorithm* dengan pendekatan *procedural programming* menggunakan GCC *compiler*.
3. Bagaimana menguji *library genetic algorithm* dengan suatu aplikasi pada kasus tertentu.

1.3 Batasan Masalah

Beberapa hal yang menjadi batasan masalah dalam penulisan tugas akhir ini antara lain:

1. *Library genetic algorithm* dibuat dalam bentuk *file Archive* yang ditulis dengan pendekatan *procedural programming* menggunakan bahasa pemrograman C.
2. Algoritma yang digunakan adalah *simple genetic algorithm* (*simple* GA).
3. Metode genetik yang digunakan adalah *multi-point crossover*, mutasi *random uniform*, dan seleksi *roulette-wheel* untuk pengkodean *real-value*.
4. *Compiler* yang digunakan adalah GNU GCC, dengan IDE Bloodshed Dev-C++.
5. Pengujian dilakukan dengan membandingkan perhitungan sebuah aplikasi yang terhubung secara *static-link* dengan *archive*, dengan hasil

perhitungan referensi dan dengan metode konvensional pada kasus uji tertentu.

1.4 Tujuan

Tujuan dari penyusunan tugas akhir (skripsi) ini yaitu:

Mengimplementasikan algoritma genetika dalam suatu *library* perangkat lunak.

1.5 Manfaat

Manfaat yang diharapkan dapat diperoleh yaitu:

a. Bagi penyusun
   1. Meningkatkan pemahaman penggunaan algoritma genetika.
   2. Mampu mengimplementasikan algoritma genetika dalam *library* perangkat lunak.
   3. Mampu membangun suatu *library* perangkat lunak yang dapat digunakan untuk menyelesaikan permasalahan yang ada.

b. Bagi pengguna

Untuk pengguna, bermanfaat mempermudah implementasi algoritma dan proses pencarian solusi optimal berdasarkan parameter-parameter yang diberikan oleh pengguna.

1.6 Sistematika Penulisan

Sistematika penulisan laporan skripsi ini adalah sebagai berikut:

BAB I Pendahuluan

Menjelaskan latar belakang, rumusan masalah, ruang lingkup, tujuan dan sistematika penulisan skripsi.

BAB II Dasar Teori

Menjelaskan kajian pustaka dan dasar teori yang digunakan.

BAB III Metodologi

Menjelaskan metode yang digunakan dalam pengerjaan skripsi.

BAB IV Perancangan

Menjelaskan analisa kebutuhan perangkat lunak, perencanaan dan perancangan *library* algoritma genetika pada perangkat lunak.

BAB V Implementasi

Membangun sebuah *library* algoritma genetika yang sesuai dengan perancangan sistem sebelumnya.

BAB VI Pengujian

Menjelaskan langkah-langkah pengujian dari sistem yang telah dibuat dan membahas hasil pengujiannya.

BAB VII Penutup

Berisi kesimpulan tentang hasil implementasi algoritma genetika ke dalam *library* dan analisis kinerja *library* pada pengujian, serta saran untuk penelitian dan pengembangan perangkat lunak.

BAB II
TINJAUAN PUSTAKA

2.1 Komputasi Evolusioner

Teori evolusi Charles Darwin pada 1859, memiliki esensi sebuah mekanisme optimisasi dan pencarian yang efektif pada alam. Prinsip Darwin tentang "*Survival of the fittest*" dapat digunakan sebagai titik awal untuk masuk dalam komputasi evolusioner. Makhluk hidup yang berevolusi menunjukkan perilaku kompleks yang optimal pada tiap-tiap tingkatan, mulai dari tingkat sel, organ, individu, hingga populasi.

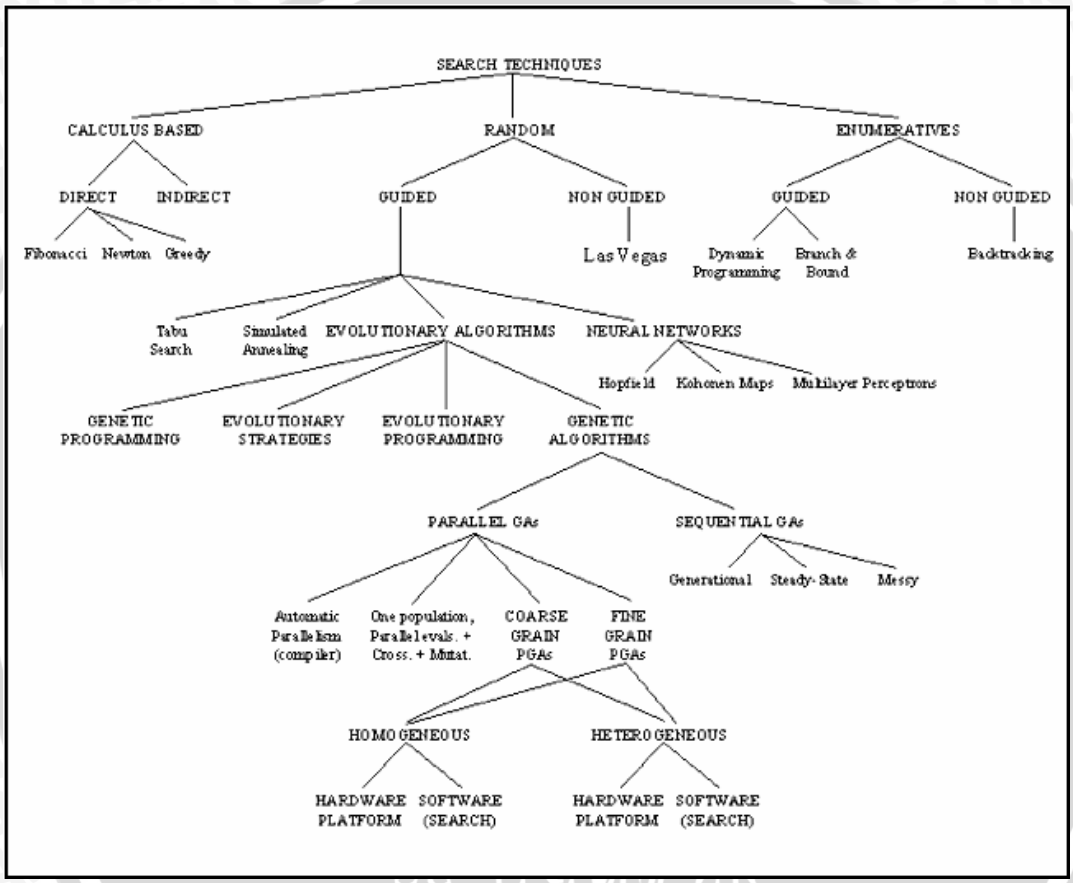
Spesies biologis telah memecahkan permasalahan yang berbasis kemungkinan atau acak yang terbukti dapat dibandingkan setara dengan metode klasik untuk proses optimisasi. Konsep evolusioner dapat diaplikasikan untuk masalah yang tidak memiliki solusi yang dapat diprediksikan, atau permasalahan yang kemungkinan memberikan hasil yang kurang memuaskan dengan metode konvensional. Sebagai hasilnya, algoritma evolusioner banyak digunakan untuk pendekatan praktis dari pemecahan suatu permasalahan.

2.1.1 Pengantar Komputasi Evolusioner

Teori seleksi alam mengemukakan bahwa semua tanaman dan hewan yang hidup saat ini adalah hasil dari proses adaptasi terhadap perubahan kondisi lingkungan selama jutaan tahun. Dalam satu rentang waktu yang sama, beberapa jenis organisme berbeda dapat hidup dan saling berkompetisi untuk mendapatkan sumber daya yang terdapat dalam ekosistem. Hanya organisme yang paling mampu mendapatkan sumber daya ini dan berhasil berkembang biak yang akan memiliki keturunan terbanyak pada generasi mendatang. Sedangkan yang kurang mampu beradaptasi cenderung kalah bersaing dan memiliki sedikit atau bahkan tidak memiliki keturunan di masa mendatang. Organisme yang pertama ini dapat disebut memiliki nilai *fitness* lebih tinggi daripada yang berikutnya. Dengan kata lain, organisme yang memiliki sifat-sifat yang dianggap baik ini yang akan memenuhi populasi dalam ekosistem.

Teknik komputasi evolusioner mengaplikasikan prinsip evolusi ini dalam sebuah algoritma yang dapat digunakan untuk mencari solusi optimal dari suatu permasalahan. Aspek utama yang membedakan algoritma pencarian secara

evolusioner dengan algoritma tradisional adalah basis kerja dari algoritma evolusioner yaitu pencarian berbasis populasi atau set nilai. Melalui keturunan dalam jumlah besar yang beradaptasi pada generasi selanjutnya, algoritma evolusioner melakukan sebuah proses pencarian secara langsung yang efisien. Secara umum pencarian secara evolusioner berjalan lebih baik daripada *random search* murni, dan tidak memiliki kelemahan mendasar dari teknik pencarian *local search* seperti *hill-climbing* yang berdasarkan gradien atau kemiringan.



Gambar 2.1 Teknik pencarian [SIV-13]

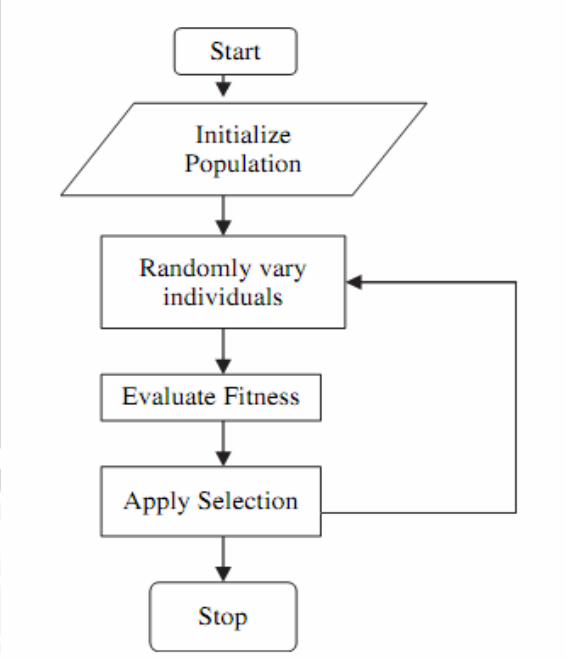
Dalam komputasi evolusioner, ada empat buah paradigma yang menjadi basis di lapangan, yaitu *genetic algorithms* (Holland, 1975), *genetic programming* (Koza, 1992, 1994), *evolutionary strategies* (Recheuberg, 1973), dan *evolutionary programming* (Forgel et al., 1966). Perbedaan yang mendasar dari paradigma ini terletak pada sifat dari skema pemodelan, operator reproduksi, dan metode seleksi. [SIV-13].

2.1.2 Keunggulan Komputasi Evolusioner

Komputasi evolusioner mendeskripsikan daerah pencarian dari semua algoritma evolusioner dan memberikan kelebihan praktis untuk beberapa permasalahan optimisasi. Beberapa keunggulan ini meliputi kesederhanaan konseptual dan kemampuan untuk menyelesaikan masalah yang tidak memiliki solusi. Pada bagian berikut ini akan dijelaskan sekilas tentang kelebihan ini dan menawarkan konsep perancangan algoritma evolusioner dalam penyelesaian permasalahan dunia nyata.

2.1.2.1 Kesederhanaan Konseptual

Keunggulan utama dari komputasi evolusioner adalah kesederhanaan konseptual Gambar 2.2 menunjukkan aplikasi dari algoritma evolusioner dalam fungsi optimisasi. Algoritma ini terdiri dari proses inisialisasi, variasi iterasi, dan seleksi berdasarkan indeks performa dari set solusi yang dihasilkan. Tidak diperlukan adanya informasi mengenai gradien di dalam algoritma ini. Melalui proses iterasi dari variasi yang acak dan proses seleksi, populasi dapat dibuat dibuat konvergen untuk menentukan solusi yang dianggap optimal. Efektifitas dari algoritma evolusioner tergantung dari variasi dan operator seleksi yang diaplikasikan pada representasi yang dipilih.



Gambar 2.2 Diagram alir dari Algoritma Evolusioner [SIV-13]

2.1.2.2 Menyelesaikan Masalah yang Tidak Memiliki Solusi

Keunggulan dari algoritma evolusioner meliputi kemampuan untuk melaukan referensi terhadap suatu permasalahan saat tidak ada kepakaran manusia dalam masalah tersebut. Walaupun kepakaran dari manusia seharusnya digunakan jika dibutuhkan dan tersedia, tetapi seringkali dapat dianggap kurang layak untuk keterlibatannya dalam proses penyelesaian masalah secara otomatis. Sistem kepakaran memiliki beberapa permasalahan sendiri, diantaranya jika pakar yang menjadi referensi tidak memiliki kualifikasi, tidak setuju dengan model, ataupun justru menyebabkan terjadinya error. Kecerdasan buatan bisa diaplikasikan pada beberapa permasalahan yang sukar yang membutuhkan kecepatan perhitungan sangat tinggi, tetapi tetap saja belum bisa bersaing dengan kecerdasan manusia. Fogel pada tahun 1995 menyatakan bahwa kecerdasan buatan mampu menyelesaikan masalah, tetapi mereka tidak menjawab permasalahan tentang bagaimana cara menyelesaikan masalah tersebut. Sebaliknya, komputasi evolusioner, menyediakan metode penyelesaian permasalahan tentang bagaimana cara menyelesaikan suatu masalah.

2.2 Algoritma Genetika

Algoritma Genetika adalah algoritma evolusioner yang memanfaatkan proses seleksi alamiah yang dikenal dengan proses evolusi. Dalam proses evolusi, individu secara terus-menerus mengalami perubahan gen untuk menyesuaikan dengan lingkungan hidupnya. “Hanya individu-individu yang kuat yang mampu bertahan”. Proses seleksi alamiah ini melibatkan perubahan gen yang terjadi pada individu melalui proses perkembang-biakan. Dalam algortima genetika ini, proses perkembang-biakan ini menjadi proses dasar yang menjadi perhatian utama, dengar dasar berpikir: “Bagaimana mendapatkan keturunan yang lebih baik” [BAS-03]. Algoritma Genetika ini ditemukan oleh John Holland yang idenya tertulis dalam buku “Adaptation in natural and artificials systems” pada tahun 1975. Holland mengusulkan algoritma genetika sebagai metode heuristik berdasar pada “Kebertahanan pada *fittest* (*Survival of fittest*)” [SIV-13].

Prinsip algoritma genetika diambil dari teori Darwin yaitu setiap makhluk hidup akan menurunkan satu atau beberapa karakter ke keturunannya

(anak). Algoritma ini dimulai dengan kumpulan solusi yang disebut dengan populasi. Populasi disusun dari bermacam-macam individu. Setiap individu berisi *phenotype* (parameter), *genotype* (kromosom buatan atau *bit string*) dan *fitness* (fungsi objektif). Solusi-solusi dari sebuah populasi diambil dan digunakan untuk membentuk populasi yang baru. Hal ini dimotivasi dengan harapan bahwa populasi yang baru dibentuk tersebut akan lebih baik daripada yang lama. Dalam hal ini, individu dilambangkan dengan sebuah nilai *fitness* yang akan digunakan untuk mencari solusi terbaik dari persoalan yang ada.

Beberapa istilah penting dalam algoritma genetika [BAS-03], antara lain:

1) *Genotype* adalah sebuah nilai yang menyatakan satuan dasar yang membentuk suatu arti tertentu dalam satu kesatuan gen yang dinamakan kromosom. Dalam algoritma genetika, gen ini bisa berupa nilai biner, *float*, *integer* maupun karakter, atau kombinatorial.
2) Allele, yaitu nilai dari gen.
3) Kromosom, merupakan gabungan gen-gen yang membentuk nilai tertentu.
4) Individu, menyatakan suatu nilai atau keadaaan yang menyatakan salah satu solusi yang mungkin dari permasalahan yang diangkat.
5) Populasi, merupakan sekumpulan individu yang akan diproses bersama dalam satu siklus evolusi.
6) Generasi, menyatakan satu-satuan siklus proses evolusi.
7) Nilai fitness, menyatakan seberapa baik nilai dari suatu individu atau solusi yang didapatkan.

2.2.1 Parameter-parameter Algoritma Genetika

Algoritma genetika bekerja berdasarkan parameter-parameter tertentu yang akan mempengaruhi kinerja dan perilaku dari algoritma ini. Beberapa parameter penting yang secara langsung mempengaruhi performa dari algoritma genetika adalah ukuran populasi, jumlah generasi, probabilitas crossover, dan probabilitas mutasi.

2.2.1.1 Ukuran Populasi

Ukuran populasi menunjukkan seberapa banyak kromosom yang ada pada populasi (dalam satu generasi). Bila ada terlalu banyak kromosom, maka algoritma genetika memiliki beberapa kemungkinan untuk melakukan *crossover* dan hanya sebagian kecil *space* pencarian yang dieksplorasi. Dengan kata lain, jika ada terlalu banyak kromosom, algoritma genetika akan bergerak lambat. Penelitian menunjukkan setelah batas tertentu (terutama tergantung pada pengkodean dan masalahnya) tidak berguna jika ukuran populasinya terus ditambah, karena tidak membuat masalah terselesaikan dengan cepat.

2.2.1.2 Jumlah Generasi

Generasi dapat dikatakan sebagai jumlah iterasi yang dilakukan terhadap proses evaluasi tiap-tiap populasi. Seperti halnya ukuran populasi, besarnya generasi akan mempengaruhi kecepatan konvergensi. Semakin besar jumlah generasi, mengakibatkan konvergensi yang lambat, tetapi bila jumlah generasi awal semakin kecil maka dapat terjadi konvergensi prematur. Untuk itu jumlah generasi yang tepat juga harus diperhitungkan dalam melakukan proses optimasi menggunakan algoritma genetika. Proses algoritma genetika akan dihentikan apabila jumlah generasi sudah terpenuhi.

2.2.1.3 Probabilitas *Crossover*

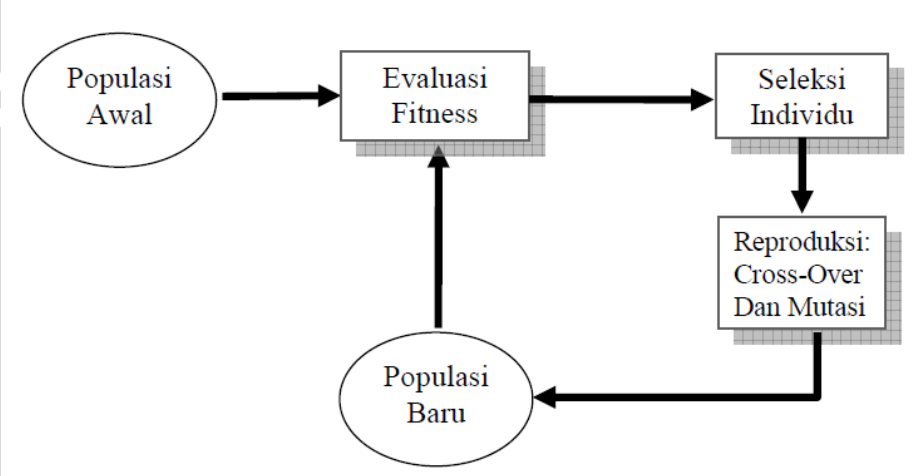
Probabilitas *Crossover* menunjukkan seberapa persen dari total kromosom yang akan melalui proses *crossover*. Bila tidak terjadi *crossover,* maka *offspring* (sifat anak hasil *crossover*) merupakan salinan yang serupa dari induk (*parent*). Bila terjadi *crossover*, *offspring* disusun dari bagian-bagian kromosom induk (*parent*). Bila probabilitas *crossover* mencapai 100%, maka semua *offspring* disusun dari hasil *crossover*. Bila probabilitas *crossover* adalah 0%, maka keseluruhan generasi baru disusun dari salinan *genotype* populasi terdahulu yang serupa (namun hal ini tidak berarti generasi baru sama dengan generasi terdahulu). *Crossover* dilakukan dengan harapan agar tercipta sifat-sifat baru dalam *genotype* pada generasi selanjutnya yang memiliki sifat lebih baik daripada generasi induk.

2.2.1.4 Probabilitas Mutasi

Probabilitas mutasi menunjukkan seberapa sering bagian-bagian kromosom akan bermutasi. Bila tidak terjadi mutasi, *offspring* yang akan memasuki fase seleksi alam adalah *offspring* setelah proses *crossover* (atau disalin) tanpa adanya perubahan. Bila mutasi terjadi, bagian-bagian gen yang terpilih secara acak akan berubah. Bila probabilitas mutasi 100%, maka keseluruhan anggota di dalam populasi akan mengalami perubahan *genotype*, sedangkan bila probabilitas mutasinya yaitu 0%, maka tidak ada yang berubah.

2.2.2 Siklus Algoritma Genetika

Siklus algoritma genetika pertama kali dikenalkan oleh David Goldberg. Gambar 2.2 di bawah ini menunjukkan siklus tersebut.



Gambar 2.3 Siklus algoritma genetika [BAS-03]

Siklus algoritma genetika tersebut merupakan inti dari algoritma *simple genetic algorithm* yang berdasarkan pada rumusan awal yang diberikan oleh Holland pada tahun 1975.

2.2.3 Komponen-komponen Utama Algoritma Genetika

Ada 6 komponen utama dalam algoritma genetika, yaitu pengkodean kromosom atau representasi *genotype*, nilai *fitness*, pembangkitan populasi awal, seleksi, pindah-silang (*crossover*) dan mutasi.

2.2.3.1 Pengkodean Kromosom

Pengkodean merupakan proses merepresentasikan gen-gen individu. Proses tersebut dapat dilakukan menggunakan bit, bilangan real, pohon (*tree*), larik (*array*), *list* atau objek yang lainnya [SIV-13]. Pengkodean yang tepat sangat menentukan berhasil atau tidaknya proses algoritma genetika dalam menyelesaikan suatu permasalahan. Pengkodean yang tepat juga akan menentukan tingkat efisiensi komputasi yang digunakan. Beberapa pengkodean yang umum digunakan antara lain adalah pengkodean *binary* dan pengkodean *real value*. Pengkodean ini akan menentukan bentuk representasi data yang diolah di dalam operasi genetik.

2.2.3.2 Pembangkitan Populasi Awal

Membangkitkan populasi awal adalah proses membangkitkan sejumlah individu secara acak atau melalui prosedur tertentu. Syarat-syarat yang harus dipenuhi untuk menunjukkan suatu solusi harus benar-benar diperhatikan dalam pembangkitan setiap individunya [BAS-03]. Ukuran untuk populasi tergantung pada masalah yang akan diselesaikan dan jenis operator genetika yang akan diimplementasikan. Setelah ukuran populasi ditentukan, selanjutnya dilakukan pembangkitan tiap individunya.

Ada beberapa teknik dalam pembangkitan populasi awal yaitu random generator, pendekatan tertentu memasukkan nilai tertentu ke dalam gen dan permutasi gen.

1) Random Generator

   Inti dari cara ini adalah melibatkan pembangkitan bilangan random untuk nilai setiap gen sesuai dengan representasi kromosom yang digunakan. Bila menggunakan representasi biner, salah satu contoh penggunaan random generator adalah penggunaan rumus berikut untuk pembangkitan populasi awal:

   $$IPOP = \mathbf{round}\{\mathbf{random}(N_{ipops}, N_{bits})\}$$

   dengan IPOP adalah gen yang nantinya berisi pembulatan dari bilangan random yang dibangkitkan sebanyak N_{ipop} (Jumlah populasi) X N_{bits} (Jumlah gen dalam tiap kromosom).

2) Pendekatan Tertentu (Memasukkan Nilai Tertentu ke dalam Gen)

Cara ini adalah dengan memasukkan nilai tertentu ke dalam gen dari populasi awal yang dibentuk.

3) Permutasi Gen

Salah satu cara permutasi gen dalam pembangkitan populasi awal adalah penggunaan permutasi Josephus dalam kasus TSP.

2.2.3.3 Nilai *fitness*

Nilai *fitness* adalah nilai yang menyatakan baik tidaknya suatu solusi (individu). Nilai *fitness* ini yang dijadikan acuan dalam mencapai nilai optimal dalam algoritma genetika. Algoritma genetika bertujuan mencari individu dengan nilai *fitness* yang paling tinggi [BAS-03]. Nilai *fitness* suatu kromosom dapat dihitung dengan menggunakan fungsi objektif.

Dalam TSP, karena TSP bertujuan meminimalkan jarak, maka nilai *fitness*nya adalah inversi dari total jarak dari jalur yang didapatkan [BAS-03]. Cara melakukan inversi bisa menggunakan rumus $1/x$ atau $100000-x$, dengan x adalah total jarak dari jalur yang didapatkan.

2.2.3.4 Seleksi

Setiap anggota yang terdapat dalam populasi akan melalui proses seleksi untuk menentukan mana yang bertahan ke generasi selanjutnya. Sesuai dengan teori Evolusi Darwin, maka kromosom yang baik akan bertahan dan menghasilkan keturunan yang baru untuk generasi selanjutnya. Langkah pertama yang dilakukan dalam seleksi ini adalah pencarian nilai fitness. Semakin tinggi nilai fitness suatu individu, semakin besar kemungkinannya untuk terpilih [BAS-03].

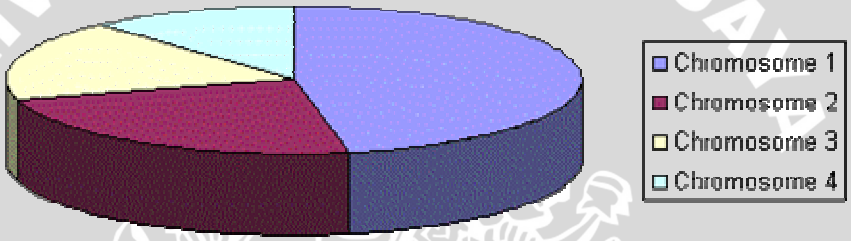
Terdapat beberapa metode yang digunakan pada proses seleksi, antara lain *Rank Selection, Roulette Wheel Selection,* dan *Tournament Selection*.

2.2.3.4.1 *Rank Selection*

Pada pengkodean ranking, kromosom pada populasi diranking sesuai dengan nilai *fitness*-nya, kemudian kromosom diberi nilai *fitness* yang baru sesuai dengan rankingnya. Kromosom dengan ranking terbawah akan mendapat nilai *fitness* 1, ranking terbawah kedua mendapat nilai *fitness* 2, demikian seterusnya. Kromosom dengan ranking terbaik akan mendapat nilai *fitness* N [SIN-12].

2.2.3.4.2 *Roulette Wheel Selection*

Pada *Roulette Wheel Selection*, kromosom akan dipilih secara acak ditentukan dengan memperhitungkan nilai *fitness* masing-masing kromosom. Semakin besar nilai *fitness* suatu kromosom, semakin besar pula peluang kromosom tersebut untuk terpilih dalam proses seleksi. Pengkodean *roulette wheel* dapat dianalogikan seperti permainan roda putar (*roulette wheel*). Pada permainan roda putar, lingkaran roda dibagi menjadi beberapa wilayah. Pada *roulette wheel selection*, lebar suatu wilayah kromosom ditentukan menurut nilai *fitness*-nya, semakin besar nilai *fitness*-nya maka akan semakin besar wilayahnya, dan semakin besar pula peluang kromosom tersebut untuk terpilih [SIN-12].



Gambar 2.4 *Roulette Wheel Selection*

Roulette Wheel Selection dapat disimulasikan dengan algoritma sebagai berikut [SIN-12]:

1) **[Sum]** Jumlahkan semua nilai *fitness* tiap-tiap anggota pada populasi.
2) **[Select]** *Generate* bilangan *random* pada interval (0,S)-r.
3) **[Loop]** secara sekuensial dari kromosom pertama, jumlahkan nilai *fitness* relatif dari setiap indeks yang dilewati. Apabila nilai acak pada indeks ke-*i* s>r, maka seleksi berhenti dan semua gen pada lokasi indeks *i* terpilih.

2.4.4.3 *Tournament Selection*

Pada metode *Tournament Selection*, ditetapkan suatu nilai *tour* untuk individu-individu yang dipilih secara random dari suatu populasi. Individu-individu yang terbaik dalam kelompok ini akan diseleksi dan terpilih dalam

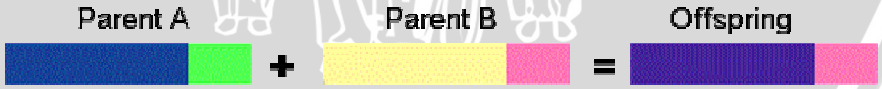
proses seleksi. Parameter yang digunakan pada metode ini adalah ukuran *tour* yang bernilai antara 2 sampai N (jumlah individu dalam suatu populasi).

2.2.3.5 *Crossover* (Pindah-silang)

Crossover merupakan salah satu operator dalam algoritma genetika yang melibatkan dua induk untuk menghasilkan keturunan yang baru. *Crossover* dilakukan dengan melakukan pertukaran gen dari dua induk secara acak. Proses *Crossover* dilakukan pada setiap individu dengan probabilitas *Crossover* yang telah ditentukan [BAS-03]. Operasi ini tidak selalu dilakukan pada semua individu yang ada. Bila *crossover* tidak dilakukan, maka nilai dari induk akan diturunkan kepada keturunannya. Probabilitas keberhasilan operasi *crossover* dinyatakan dengan P_c . Terdapat beberapa metode *crossover* yang umum digunakan untuk *binary encoding*, diantaranya adalah *crossover* 1 titik, *crossover* 2 titik, *crossover* seragam, dan *arithmetic crossover*. Sedangkan untuk pengkodean permutasi hanya terdapat *single point crossover*. Dalam skripsi ini akan digunakan *crossover* dari pengkodean *real value* secara *multi*-point *crossover* yang menukar nilai gen secara acak berdasarkan jumlah variabel solusi.

2.2.3.5.1 *Crossover* 1 titik (*Single Point Crossover*)

Proses *crossover* dilakukan dengan memisahkan suatu *string* menjadi dua bagian dan selanjutnya salah satu bagian dipertukarkan dengan salah satu bagian dari *string* yang lain yang telah dipisahkan dengan cara yang sama.



11001011 + 11011**111** = **11001111**

Gambar 2.5 Contoh *Single Point Crossover*

2.2.3.5.2 *Crossover* 2 titik (*Two Point Crossover*)

Proses *crossover* ini dilakukan dengan memilih dua titik *crossover*. Kromosom keturunan kemudian dibentuk dengan barisan bit dari awal kromosom sampai titik *crossover* pertama disalin dari orangtua pertama, bagian

dari titik *crossover* pertama dan kedua disalin dari orangtua kedua, kemudian selebihnya disalin dari orang tua pertama lagi.

11001**01**1 + 11**011**111 = **11011111**

Gambar 2.6 Contoh *Two Point Crossover*

2.2.3.5.3 *Crossover* seragam (*Uniform Crossover*)

Pada *crossover* ini, bit-bitnya secara acak disalin dari kedua orangtua kepada kromosom keturunannya.

1**1**001**01** + **1**1011**1**01 = 11011111

Gambar 2.7 Contoh *Uniform Crossover*

2.2.3.5.4 *Arithmatic Crossover*

Kromosom *offspring* (kromosom anak) diperoleh dengan melakukan operasi aritmatika terhadap *parent* (induk). Operasi yang dapat dilakukan antara lain AND, OR, XOR dan lain-lain.

11001011 + 11011111 = 11001001 (AND)

Gambar 2.8 Contoh *Arithmatic Crossover*

2.2.3.5.5 *Crossover* Pengkodean Permutasi

Pada pengkodean permutasi, jenis pindah silang hanya *single point crossover*. Metode ini dilakukan dengan memilih satu titik *crossover p* pada kromosom *parent* (kromosom induk) secara acak. Gen ke-1 sampai gen ke-*p* pada *parent* 1 disalin menjadi gen *offspring* (kromosom anak). Sisa gen yang belum terpenuhi diambil dari *parent* 2 dengan cara sekuensial dari gen ke-1 *parent* 2 sampai gen terakhir, dengan syarat gen tersebut belum ada dalam

kromosom *offspring*. Contoh dari pindah silang pengkodean permutasi, (**1 2 3 4 5** 6 7 8 9) + (**4 5 3 6 8** 9 7 2 1) = (**1 2 3 4 5** 6 8 9 7).

2.2.3.6 Mutasi

Setelah *crossover* dilakukan, proses reproduksi dilanjutkan dengan mutasi. Hal ini dilakukan untuk menghindari solusi-solusi dalam populasi memiliki nilai lokal optimum. Mutasi adalah proses mengubah gen dari keturunan secara random. Tidak setiap gen selalu dimutasi tetapi mutasi dikontrol dengan probabilitas tertentu yang disebut dengan *mutation rate* (probabilitas mutasi) dinotasikan dengan P_m. Pada pengkodean biner, mutasi mengubah bit 0 menjadi 1 dan sebaliknya bit 1 menjadi bit 0. Contoh mutasi pada pengkodean biner: 1**1**001001 => 1**0**001001. Sedangkan mutasi secara *random uniform* menggunakan satu nilai acak yang menggantikan nilai gen secara langsung.

2.3 Rekayasa Perangkat Lunak

Algoritma genetika dapat diimplementasikan ke dalam sebuah perangkat lunak untuk membantu menyelesaikan permasalahan komputasi dengan pendekatan stokastik. Perangkat lunak merupakan kumpulan data yang menyediakan set instruksi untuk komputer, tentang apa yang harus dilakukan dan bagaimana cara melakukannya. Dalam pengertian yang lebih sempit di dunia perangkat komputer, perangkat lunak seringkali mengacu kepada set instruksi pada sebuah program untuk menangani suatu fungsi komputasi yang spesifik.

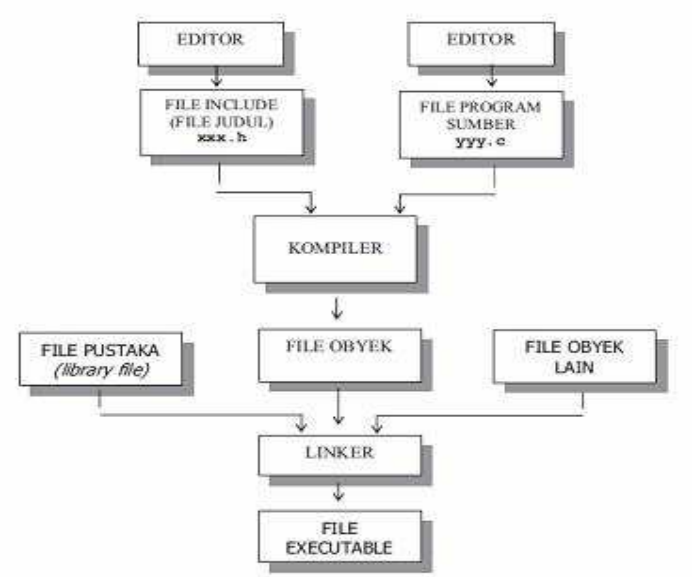
Sebuah program yang menggunakan algoritma genetika harus memiliki fungsi-fungsi dan representasi solusi yang mendukung untuk dijalankannya proses genetika. Agar *programmer* dapat lebih mudah membuat sebuah perangkat lunak yang menggunakan algoritma genetika di dalamnya, maka dapat digunakan sebuah modul tertentu berupa kumpulan kode yang mendukung operasi genetik yang siap untuk langsung digunakan dalam bentuk sebuah *library*.

2.3.1 *Library*

Sistem yang kompleks mengakibatkan peningkatan beban dari proses pembuatan maupun *maintenance* dari suatu program. Untuk dapat memperjelas

dan memudahkan proses pengembangan *project* perangkat lunak, maka dikembangkan suatu proses rekayasa perangkat lunak secara modular yaitu dengan menggunakan *library*. *Library* merupakan berkas-berkas dari kode yang siap untuk proses kompilasi yang akan digabungkan, atau dihubungkan melalui proses linking, dengan sebuah program saat proses kompilasi. Contohnya, *library* untuk fungsi matematika ataupun *library* untuk operasi masukan dan keluaran. Beberapa bahasa pemrograman memiliki *library* standart masing-masing yang tersedia untuk digunakan *programmer* dalam mengembangkan perangkat lunak, tetapi juga dimungkinkan untuk membuat library sendiri yang khusus untuk menangani fungsi tertentu.

Setiap *library* memiliki sebuah referensi yang harus diketahui saat akan digunakan oleh modul yang mereferensikan *library* tersebut, yang dapat berupa sebuah program ataupun *library* lainnya. Pada bahasa C, *library* memiliki *header* yang harus disertakan dalam bagian *preprocessor directive* untuk menyatakan bahwa *library* yang dimaksud akan digunakan sebagai referensi untuk mendefinisikan beberapa prosedur, makro, ataupun variabel global yang terdapat dalam *library* dan akan digunakan oleh program. Saat header sudah terbaca oleh perintah #include, maka isi dalam *header* tersebut secara efektif sudah ditambahakan dalam daftar *reserved words* dan *commands*, yaitu kata-kata dalam bahasa manusia yang sudah dipesan untuk secara khusus dan didefinisikan di dalam *library*.



Gambar 2.9 Proses *Compile-Linking* dalam C

Walaupun pada prinsipnya tidak ada batasan atas berapa jumlah *library* yang dapat dihubungkan ke dalam program, tetapi pada praktiknya keterbatasan memori merupakan faktor yang paling menentukan karena semakin banyak *library* yang dihubungkan akan meningkatkan konsumsi penggunaan memori oleh program. Beberapa sistem operasi secara otomatis hanya melakukan proses *load* dari bagian *library* yang dibutuhkan program, tetapi sistem lainnya membutuhkan *load* keseluruhan isi *library* yang dapat membuat program dieksekusi lebih lambat.

Secara umum, *library* terdiri dari dua jenis, yaitu *static library* dan *shared library*. Keseluruhan kode dari *library* yang terhubung secara statis akan digabungkan pada *object code* dari program yang menggunakannya, sedangkan *library* yang terhubung secara dinamis hanya memasukkan *stub* ke dalam program, yang merupakan referensi pemanggilan bagian tertentu saja saat program memintanya. Dengan kelebihan dan kekurangan masing-masing, keduanya masih umum digunakan hingga saat ini. Sesuai dengan lingkup perancangan, jenis *library* yang akan dibahas lebih lanjut adalah *static library*.

2.3.1.1 *Static Library*

Static library merupakan jenis *library* yang terhubung secara statis, artinya kode di dalam library akan disalin ke dalam modul yang mereferensikan *link* dengan *library* tersebut. Penambahan *static library* dalam jumlah besar akan memberikan penambahan ukuran file dari *executable* secara signifikan. Tetapi *static library* masih umum digunakan karena mempermudah proses distribusi dari program yang menggunakan *library* ini, terutama pada program dengan *custom library*. Selain itu, karena *library* harus ikut disertakan saat proses kompilasi program, maka IDE dapat memastikan terlebih dahulu bahwa semua komponen yang dibutuhkan oleh program memang terdapat dalam *library* sebelum *executable* dibuat, sehingga resiko untuk *run-time error* dari program yang dibuat dapat ditekan. Dalam praktiknya, seringkali proses *debugging* dengan GDB pada *static library* berjalan lebih mudah daripada *library* yang ditujukan untuk terhubung secara dinamis.

Perangkat lunak *static library* dapat dibuat melalui beberapa pendekatan pemrograman, dua paradigma yang paling populer adalah pendekatan secara *object-oriented*, dan pendekatan *procedural programming*. Sesuai dengan ruang

lingkup penulisan skripsi, maka yang akan dibahas lebih lanjut adalah pendekatan *procedural programming*.

2.3.2 Pendekatan *Procedural Programming*

Procedural Programming adalah salah satu paradigma yang populer di dalam pemrograman komputer, yang berbasis pada konsep pemanggilan *procedure* (disebut juga sebagai *subroutine* ataupun fungsi). Pemrograman secara prosedural menggunakan langkah-langkah yang terstruktur, dan setiap fungsi dapat dipanggil kapan saja dalam eksekusi program, selama fungsi tersebut sudah dikenali oleh prosedur utama atau fungsi lain yang memanggilnya.

2.3.2.1 Prosedur

Fungsi atau prosedur merupakan kumpulan dari beberapa pernyataan yang dikumpulkan menjadi satu dalam sebuah pengenal atau nama. Dengan adanya fungsi akan didapatkan beberapa keunggulan dalam pemrograman, antara lain :

1) memudahkan dalam pembuatan program,
2) memudahkan pelacakan kesalahan,
3) dapat menghemat penggunaan memori.

Pendeklarasian fungsi secara umum adalah sebagai berikut,

```
tipe_data1 nama_fungsi (tipe_data2 nama_argumen)
{
    tipe_data3 nama_variabel;

    pernyataan_1;
    pernyataan_2;
    .
    .
    .
    pernyataan_N;
}
```

Suatu fungsi dapat memililki nilai kembalian (*return value*). *tipe_data1* merupakan tipe data dari nilai kembalian fungsi. Pengembalian hasil proses di dalam fungsi dilakukan dengan mengggunakan pernyataan *return*. *nama_fungsi*

merupakan pengenal fungsi. Sebuah fungsi akan dipanggil menurut nama pengenal fungsi tersebut, dan juga menurut format dari argumen untuk fungsi *overload* (fungsi dengan nama pengenal yang sama).

Argumen merupakan suatu variabel atau nilai yang akan diproses di dalam fungsi. *tipe_data2* merupakan tipe data dari argumen, sedangkan *nama_argumen* merupakan pengenal dari argumen. Sebuah fungsi dapat memiliki argumen lebih dari satu atau tidak memiliki argumen sama sekali. Untuk memisahkan antara argumen satu dengan yang lainnya dipergunakan tanda "," (koma).

Argumen di dalam bahasa C disalin ke *program stack* pada saat *run time*, ketika argumen tersebut dibaca oleh sebuah fungsi. Argumen tersebut dapat berupa nilai (*value*) mereka sendiri, atau berupa sebuah *pointer* yang menunjuk satu area memori berisi data yang diinginkan. *Passing* argumen yang berupa nilai dari sebuah variabel disebut sebagai *passing by value*. *Passing by reference* adalah *passing* argumen menggunakan alamat dari variabel dengan memberikan tanda '&' sebelum nama variabel, kecuali jika argumen itu berbentuk sebuah *pointer*.

2.3.2.2 Jenis Variabel Dalam Prosedur

Sebuah fungsi tidak selalu dapat mengakses semua variabel yang dideklarasikan dalam satu program. Pada bahasa pemrograman C, variabel dibedakan atas :

1) **variabel *automatic*** (disebut juga variabel lokal), variabel ini hanya dapat diakses oleh fungsi yang mendeklarasikan variabel tersebut.
2) **variabel *extern*** (disebut variable global), variabel yang dapat diakses oleh semua fungsi di dalam satu program.
3) **variabel statis**, yaitu variabel yang nilainya adalah tetap meskipun fungsi yang mendeklarasikannya sudah selesai dieksekusi.
4) **variabel referensi**, yaitu variabel yang dijadikan sebagai argumen dalam fungsi dengan memasukkan alamat (*address*) dari variabel tersebut. Apabila terjadi perubahan pada variabel ini di dalam suatu fungsi, maka nilai variabel ini di luar fungsi tersebut juga akan berubah.

Penulis program dapat mendeklarasikan variabel lokal dengan nama yang sama dengan variabel global yang berada di luar fungsi tersebut. Jika

terjadi hal tersebut, variabel global tidak dapat diakses di dalam fungsi dengan cara biasa. Untuk mengakses variabel global tersebut harus digunakan operator ruang lingkup (::).

2.3.3 GCC *Compiler*

Agar suatu program dalam bahasa pemrograman dapat dimengerti oleh komputer, program haruslah diterjemahkan dahulu ke dalam kode mesin. Adapun penterjemah yang digunakan bisa berupa *interpreter* ataupun *compiler*. Sebuah *compiler* adalah program komputer yang mengubah *source code* yang ditulis menggunakan bahasa pemrograman tingkat tinggi (*high-level programming language*) ke dalam bahasa yang memiliki level lebih rendah (bahasa assembly ataupun bahasa mesin). *Compiler* bekerja dengan cara menterjemahkan seluruh instruksi dalam program sekaligus. Keuntungan menggunakan *compiler* adalah proses eksekusi bisa berjalan dengan cepat sebab tak ada lagi proses penterjemahan. Selain itu program sumber (*souce code*) dapat dirahasiakan sebab yang dieksekusi adalah program yang sudah berbentuk kode mesin. Kelemahannya, proses pembuatan dan pengujian relatif lebih lama, sebab ada waktu untuk kompilasi dan ada pula waktu untuk *linking*. Program hanya dapat melalui proses *compile* hanya jika program tak mengandung kesalahan secara kaidah sama sekali.

GNU Compiler Collection (GCC) adalah sebuah produk *compiler* dari GNU Project. GCC telah digunakan sebagai *compiler* standart oleh kebanyakan sistem operasi bertipe Unix (*Unix-like*).

Secara garis besar, *compiler* bekerja melalui beberapa langkah untuk merubah *source code* menjadi *machine code*.

1) *Lexical Analysis*

   Proses *lexing* (atau disebut juga *scanning*) mengubah teks *source code* untuk dipecah menjadi sekumpulan token yang dapat berupa *keyword, identifier* ataupun *symbol*.

2) *Syntax Analysis*

   Pada tahapan analisis secara *syntax* dilakukan proses *parsing*, yaitu melakukan pengecekan apakah token-token sudah sesuai dengan tata bahasa ataupun aturan-aturan dalam bahasa pemrograman yang digunakan.

Dalam beberapa bahasa pemrograman (misalnya C) sebelum proses *parsing* akan dilakukan terlebih dahulu *preprocessing*.

3) *Code Generation*

Untuk proses kompilasi yang tidak memerlukan optimasi, *byte-code (intermediate language)* dari *parser* dapat langsung diberikan pada *Code Generator* untuk dikonversi menjadi *machine code*.

BAB III
METODE PENELITIAN

Dalam penyusunan skripsi ini, dirancang suatu Pengembangan *Library Genetic Algorithm* Menggunakan GCC *Compiler* Dengan Pendekatan *Procedural Programming.* Metode penelitian yang digunakan pada penyusunan skripsi ini adalah:

3.1 Studi Literatur

Mempelajari literatur-literatur atau buku serta dokumen-dokumen yang berhubungan dengan algoritma genetika, *procedural programming* dan literatur-literatur terkait.

3.2 Analisis Kebutuhan

Analisis kebutuhan bertujuan untuk mendapatkan semua kebutuhan yang diperlukan dari sistem yang akan dibangun. Metode analisis yang digunakan adalah *Structured Analysis* dengan membuat daftar spesifikasi sistem.

3.3 Perancangan

Perancangan sistem dilakukan setelah semua kebutuhan sistem didapatkan melalui tahap analisis kebutuhan. Perancangan *library* algoritma genetika yang merupakan bagian inti dari sistem dilakukan dengan pendekatan *Structured Design* yang memiliki fokus pada klasifikasi semua proses yang diperlukan dan dependensi antar prosedur yang dirancang.

3.4 Implementasi

Implementasi dilakukan dengan mengacu kepada perancangan perangkat lunak. Sistem dapat mengalami perubahan dari rancangan sebelumnya pada tahapan ini, bila ternyata diperlukan pergantian.

3.5 Pengujian

Pengujian perangkat lunak pada skripsi ini dilakukan agar dapat menunjukkan bahwa perangkat lunak telah mampu bekerja sesuai dengan spesifikasi dari kebutuhan yang melandasinya. Pengujian yang dilakukan meliputi validasi *library* dengan beberapa kasus uji, pengujian waktu yang

diperlukan untuk eksekusi (*execution time*), dan analisis kompleksitas algoritma dari prosedur inti.

3.6 Kesimpulan dan Saran

Pada tahap ini, diambil kesimpulan dari hasil pengujian dan analisis terhadap Pengembangan *Library Genetic Algorithm* Menggunakan GCC *Compiler* Dengan Pendekatan *Procedural Programming*. Tahap selanjutnya adalah pembuatan saran untuk perbaikan terhadap kekurangan-kekurangan yang ada, untuk menyempurnakan penelitian selanjutnya.

BAB IV
PERANCANGAN DAN IMPLEMENTASI

Library yang dikembangkan dalam tugas akhir ini adalah *archive* dari seperangkat *procedure* yang dapat digunakan untuk membantu *programmer* dalam membuat perangkat lunak yang membutuhkan operasi komputasi berbasis algoritma genetika. Dengan adanya *library* ini, *programmer* memiliki kemudahan dan kebebasan mengimplementasikan algoritma genetika secara umum sekaligus memiliki kerangka untuk pengembangan program yang menggunakan algoritma genetika pada implementasi kasus yang lebih spesifik.

Bab ini membahas perancangan dan implementasi *Library Genetic Algorithm* menggunakan GCC *compiler* dengan pendekatan *procedural programming* yang berisi fungsi-fungsi yang mendukung proses pencarian solusi secara *stochastic* berdasarkan seleksi alam dan proses evolusi, beberapa fungsi tambahan dan *file pointer* untuk proses pembacaan masukan dan penulisan keluaran pada *text file*, dan fungsi pembantu lain untuk *value* maupun *variable-handling*.

4.1 Sasaran Perancangan *Library*

Tujuan utama dibuatnya perangkat lunak ini adalah untuk mendapatkan sebuah *static library* yang mampu menyediakan beberapa *procedure* yang dibutuhkan seorang *programmer* untuk mengimplementasikan algoritma genetika dalam program yang akan ia buat. *Library* ini didesain dengan pendekatan *procedural programming* agar *programmer* yang terbiasa bekerja di lingkungan program yang terstruktur memiliki pilihan *archive* yang tepat, sesuai dengan kebutuhan perangkat lunak yang akan dibangun.

4.2 Kebutuhan Sistem

Archive yang dibuat merupakan sebuah *file* yang berdiri sendiri dan tidak dapat langsung dijalankan oleh sistem operasi sebagai sebuah *executable*, karena itu disertakan pula sebuah program sebagai jembatan antara *programmer* dengan *library* yang terhubung secara *static* untuk membangun sebuah *executable* yang dapat langsung digunakan oleh *user*. Keseluruhan sistem yang dibuat ini bekerja di dalam lingkup perangkat lunak menggunakan GCC *compiler* sebagai media translasi ke bahasa mesin.

Selain itu, sistem ini juga membutuhkan lingkungan sistem operasi yang mendukung GCC *compiler* dan memiliki kemampuan untuk membaca maupun menulis *text file*.

4.3 Perancangan Sistem *Library* Secara Global

Dalam pembuatan sebuah sistem atau perangkat lunak, pasti melalui langkah-langkah perancangan sistem itu terlebih dahulu yang dapat digambarkan dan dijabarkan dalam berbagai cara. Perancangan sistem secara global diperlukan sebagai langkah awal untuk membangun *library* sebagai bagian penting dari sistem itu sendiri.

4.3.1 Cara Kerja Library `GASLib.a`

Cara kerja dari *library Genetic Algorithm* ini tidak lepas dari peranan program perantara yang terhubung secara *static* dengan *archive*. Hampir seluruh prosedur di dalam *library* ini merupakan fungsi yang *stand-alone* dan tidak membutuhkan argumen sebagai parameter, sehingga *programmer* dapat dengan bebas menentukan sendiri fungsi mana yang perlu untuk dipanggil melalui program perantara.

Algoritma dimulai dengan proses representasi genotype yang tersedia di dalam *library* sesuai dengan parameter genetik yang ditentukan oleh *programmer* yang menggunakan *library* ini. Dalam hal ini parameter yang sifatnya cenderung statis atau tidak berubah-ubah, yaitu fungsi objektif untuk menentukan nilai *fitness*, akan ditulis sesuai kebutuhan *programmer* pada fungsi perantara. Untuk parameter yang sifatnya lebih dinamis seperti ukuran populasi dan jumlah maksimum generasi akan ditentukan user yang menjalankan aplikasi hasil dari kompilasi program perantara yang menggunakan *library* ini melalui *console* saat *run-time*. Masukan yang menjadi *domain* dari setiap variabel solusi yang direpresentasikan dalam batas atas dan batas bawah pada *genotype*, akan diambil dari *text file* eksternal sehingga untuk program dengan jumlah variabel solusi yang besar tidak memakan waktu terlalu lama dalam proses penentuan *domain* dibandingkan jika harus menuliskannya melalui *console*. Masukan dari *text file* ini dapat ditulis sendiri oleh *user* ataupun memanfaatkan keluaran dari sistem lain yang terbuat secara *auto-generate* dan disimpan dalam bentuk teks.

Di dalam setiap program perantara dengan *library*, saat proses genetik berlangsung yang akan dijalankan pertama kali adalah fungsi untuk konfigurasi parameter. Nilai parameter akan digunakan sebagai acuan dari proses pemberian nilai pada representasi genotype, yang memberikan nilai awal kepada populasi pertama untuk

menyediakan satu set solusi potensial bagi *user*. Nilai ini akan berupa nilai acak bergantung pada beberapa parameter yang dibaca oleh fungsi pemberi nilai awal.

Setelah terbentuk populasi pertama, dilakukan proses evaluasi untuk menentukan tingkat kebaikan masing-masing solusi yang direpresentasikan dalam nilai *fitness* tiap gen dalam populasi sesuai dengan fungsi objektif yang ditentukan. Saat satu set gen dianggap baik, maka nilai fitness akan lebih tinggi daripada *genotype* lainnya dalam satu generasi. Set solusi yang terbaik akan disimpan nilainya untuk dibandingkan dengan generasi selanjutnya. Saat keseluruhan proses awal ini sudah dijalankan, maka program akan mulai membuat laporan berdasarkan data-data yang telah diperoleh dari generasi pertama ini sebelum mulai masuk ke dalam daur reproduksi sampai pada keturunan terakhir.

Dalam *library* yang dibuat ini, kumpulan individu dalam populasi yang direpresentasikan melalui *genotype* mereka akan dianggap sebagai satu-satunya populasi solusi di alam. Maka setiap generasi akan terdiri dari satu komposisi populasi baru sesuai dengan nilai *fitness* masing-masing gen. Tiap *genotype* yang lebih dominan akan memiliki peluang besar untuk berkembang biak dan bertahan sampai ke generasi selanjutnya di alam (*survival of the fittest*). Proses pemilihan individu yang bertahan ke populasi selanjutnya ini terdapat di dalam prosedur tertentu untuk seleksi berdasarkan *fitness* tiap *genotype* secara relatif terhadap *fitness* semua anggota dalam populasi secara kumulatif.

Proses reproduksi akan terus dilakukan hingga keturunan terakhir yang ditentukan berdasarkan jumlah generasi, atau dalam hal ini juga berarti jumlah populasi keseluruhan yang akan dibentuk. Proses ini dimulai dengan cara memilih *parent* yang akan mengalami proses *crossover* sesuai dengan angka probabilitas *crossover*. Semakin tinggi nilai probabilitas maka akan lebih banyak individu dalam populasi yang akan menjadi *parent* bagi generasi selanjutnya. Setelah selesai proses *coupling*, fungsi *crossover* akan kemudian memanggil fungsi lain dalam *library* yang bertugas melakukan proses pindah-silang dari tiap pasang *parent* yang telah terpilih secara acak pada masing-masing variabelnya.

Proses reproduksi aseksual di alam akan diwakilkan melalui proses mutasi, yaitu munculnya individu dengan sifat baru tanpa adanya interaksi seksual antara dua individu dalam populasi. Karena peluang terjadinya mutasi di alam secara umum sangatlah kecil, maka nilai probabilitas mutasi sebagai parameter algoritma genetik juga disarankan untuk menggunakan nilai paramater yang kecil. Tetapi user dapat melakukan

simulasi untuk kondisi alam yang memungkinkan mutasi terjadi lebih sering daripada umumnya dengan cara memberikan nilai probabilitas mutasi yang lebih besar. Semakin tinggi nilai probabilitas mutasi maka akan lebih banyak *genotype* yang bermutasi sehingga lebih banyak individu yang mengalami proses mutasi pada tiap generasi. *Genotype* yang terpilih dalam proses mutasi akan mendapatkan nilai kombinasi gen baru yang diberikan secara acak oleh fungsi pembangkit bilangan acak di dalam *library*.

Data dari kumpulan anggota dalam populasi yang sudah banyak berubah ini akan melalui proses dokumentasi yang langsung ditulis pada sebuah *text file* agar tidak membebani kerja program dengan memakan banyak *resource* untuk menyimpan nilai-nilai yang dihasilkan. Hanya nilai terbaik dalam tiap generasi yang akan disimpan untuk nantinya dibandingkan dengan generasi berikutnya pada fungsi *elitist*. Setelah melalui proses dokumentasi, maka akan sekali lagi terjadi lagi proses evaluasi nilai-nilai baru untuk memberikan nilai fitness bagi tiap anggota baru populasi.

Jika sudah terbentuk populasi baru, maka program akan memanggil fungsi *elitist* untuk menentukan apakah anggota terbaik, atau dalam populasi di alam disebut *Alpha*, dari generasi sebelumnya lebih kuat daripada *Alpha* dari generasi yang baru terbentuk. Jika *Alpha* dari generasi lama masih lebih baik dari generasi terbaru, maka ia berhak tetap ada dalam populasi menggantikan posisi dari individu paling lemah dalam populasi terbaru, dan tetap menjadi pimpinan dari generasi yang selanjutnya ini.

Proses reproduksi dan seleksi alam akan terus terjadi hingga jumlah generasi maksimum tercapai. Saat *loop* ini selesai, seluruh proses pencarian set solusi akan dihentikan dan program akan menulis kesimpulan dari semua data yang telah dikumpulkan. Nilai gen terbaik dan nilai *fitness*-nya yang sudah ditemukan akan dianggap sebagai solusi paling optimal dari algoritma ini. User akan mendapatkan informasi bahwa simulasi alam telah berhasil dan dapat melihat data yang direkam pada *text file* berisi *log* dari keluaran program.

4.3.2 Perancangan Sistem Algoritma Genetika

Perancangan sistem yang meliputi *library* maupun program perantaranya dibangun dengan menggunakan bahasa pemrograman C dan kompiler *GNU GCC*. Sistem ini dirancang dengan spesifikasi sebagai berikut:

1. Membaca parameter yang dibutuhkan algoritma genetik.
2. Mengecek keberadaan *file* masukan berisi parameter berupa *domain* variabel solusi.

3. Membuat sebuah *file pointer* untuk lokasi penampung hasil keluaran.
4. Membuat representasi *genotype*.
5. Menyediakan fungsi pemberi nilai awal.
6. Menyediakan pembangkit bilangan acak.
7. Menyediakan fungsi-fungsi reproduktif yaitu *crossover* dan mutasi.
8. Menyediakan fungsi seleksi dan *elitist* untuk simulasi seleksi alam.
9. Menyediakan fungsi pembantu untuk proses *swap* alamat memori.
10. Menyediakan pemesanan alamat memori dalam *heap* dan prosedur pembebasan memori.
11. Menulis *log* pada sebuah *text file* sebagai laporan hasil simulasi.
12. Memberi informasi pada *user* setelah simulasi berhasil dilakukan.

Spesifikasi lain yang berhubungan dengan *linker* seperti *static link* antara file dari program perantara dengan *archive*, dan lokasi atau *directory library* yang harus diketahui oleh program perantara, akan diatur melalui *integrated development environment* (IDE).

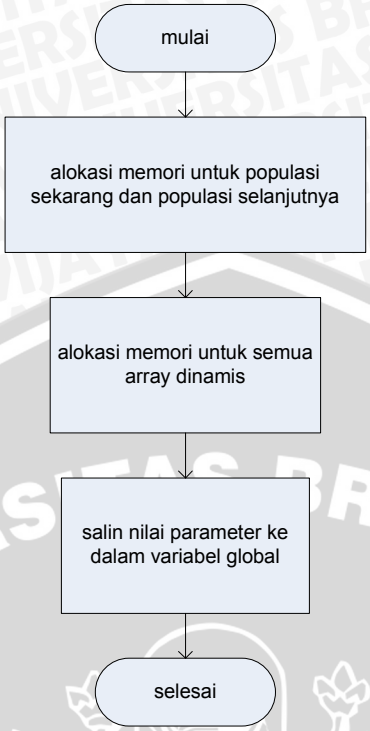
4.4 Perancangan Terinci

Library Genetic Algorithm ini dirancang secara detail dengan membagi program menjadi beberapa sub-program atau prosedur. Hal ini dilakukan dalam rangka memudahkan dalam perancangan dan implementasi *library*. Berikut ini beberapa prosedur yang dirancang.

4.4.1 Perancangan Prosedur Pembacaan Parameter

Saat program hasil kompilasi dijalankan, *user* akan diminta untuk memasukkan beberapa nilai yang digunakan sebagai parameter algoritma genetika. Parameter ini akan dibaca oleh *library* sebagai sebuah argumen untuk proses alokasi memori bagi *dynamic array* pointer pada heap. Nilai parameter lain yang mengatur probabilitas akan disalin ke dalam kelompok variabel global dalam *library*.

Berikut ini merupakan *flowchart* prosedur pembacaan parameter:



Gambar 4.1 *Flowchart* prosedur pembacaan parameter

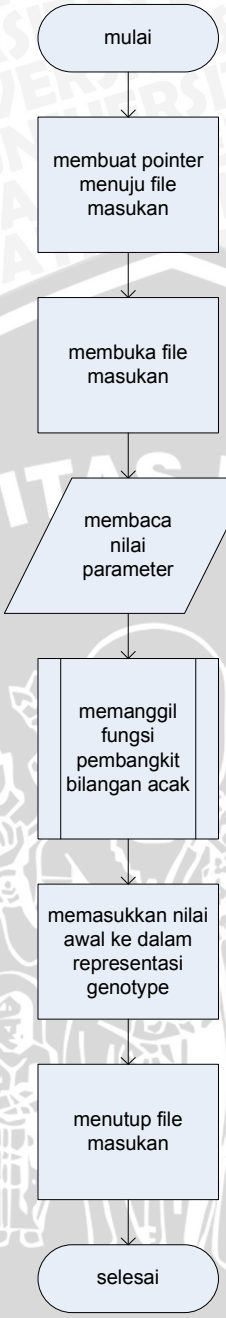
Berikut penjelasan dari *flowchart* prosedur pembacaan parameter:

1. Memori dalam *heap* dipesan sebesar ukuran yang dibutuhkan untuk menampung dua populasi.
2. Seluruh *array* dalam populasi menggunakan *array* dinamis yang harus memiliki lokasi dalam memori sebanding dengan jumlah indeks semua *array* yang baru didapatkan saat *run-time*.
3. Nilai parameter yang dijadikan argumen prosedur disalin ke dalam variabel global agar dapat digunakan semua prosedur dalam *library* tanpa perlu banyak *value passing*.

4.4.2 Perancangan Prosedur Pemberian Nilai Awal

Dalam sub program ini dilakukan inisialisasi atau pemberian nilai awal bagi tiap representasi *genotype* dalam populasi. Prosedur ini juga meliputi proses akses dan terminasi *file* masukan.

Berikut ini merupakan *flowchart* prosedur pemberian nilai awal:



Gambar 4.2 *Flowchart* prosedur pemberian nilai awal

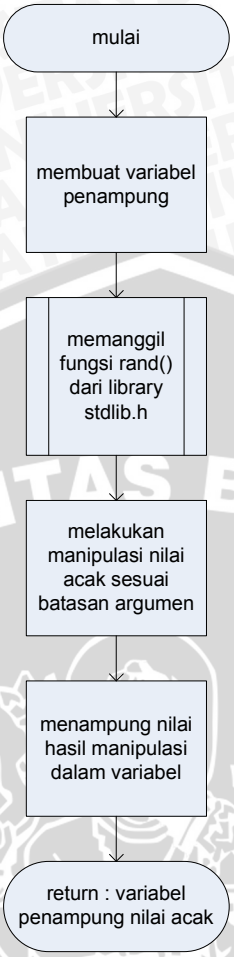
Berikut penjelasan dari *flowchart* prosedur pemberian nilai awal

1. *Library* akan membuat sebuah *pointer* yang menghubungkan program perantara dengan *file* masukan. *Directory* secara *default* akan memiliki lokasi dalam *folder* yang sama dengan program perantara.

2. Setelah diperoleh lokasi memori dari *file* masukan, prosedur ini akan mencoba membuka *file* tersebut. *Console* atau *terminal* akan memberikan informasi *error* jika *file* gagal dibuka.
3. Isi dari file tersebut yang berupa batas bawah dan batas atas dari setiap variabel solusi yang terlibat akan dibaca per baris dan nilainya ditampung dalam sebuah *array* sebagai *domain* tiap variabel.
4. Nilai acak akan diberikan dengan cara memanggil prosedur pembangkit bilangan acak.
5. Semua nilai yang didapat akan dimasukkan sebagai nilai awal ke dalam representasi *genotype*.
6. File masukan akan ditutup setelah pemberian nilai awal selesai dilakukan.

4.4.3 Perancangan Prosedur Pembangkit Nilai Acak

Dalam sub program ini dilakukan proses pembangkitan sebuah nilai acak sesuai dengan argumen yang diberikan sebagai parameter saat prosedur ini dipanggil. Berikut ini merupakan *flowchart* prosedur pembangkit nilai acak:



Gambar 4.3 *Flowchart* prosedur pembangkit nilai acak

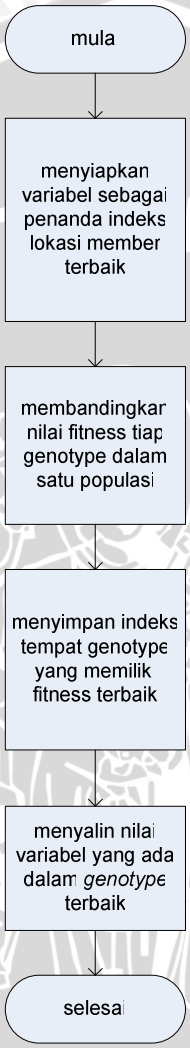
Berikut penjelasan dari *flowchart* prosedur pembangkit nilai acak:

1. Variabel untuk menampung nilai acak disiapkan lebih dahulu.
2. Memanggil fungsi `rand()` yang terdapat dalam *library* standart dengan *header* `stdlib.h` untuk memilih satu bilangan acak.
3. Melakukan manipulasi nilai acak agar sesuai dengan *domain* yang ditentukan melalui batasan argumen yang dijadikan parameter fungsi.
4. Nilai hasil manipulasi akan ditampung dalam variabel yang sudah disiapkan sebelumnya.
5. Kembalian fungsi berupa sebuah variabel dengan nilai acak.

4.4.4 Perancangan Prosedur Penyimpan Nilai Terbaik

Setiap generasi memiliki *genotype* terbaik menurut nilai *fitness*-nya. Sebuah prosedur dibutuhkan untuk menyimpan nilai tersebut.

Berikut ini merupakan *flowchart* prosedur penyimpan nilai terbaik:



Gambar 4.4 *Flowchart* prosedur penyimpan nilai terbaik

Berikut penjelasan dari *flowchart* prosedur penyimpan nilai terbaik:

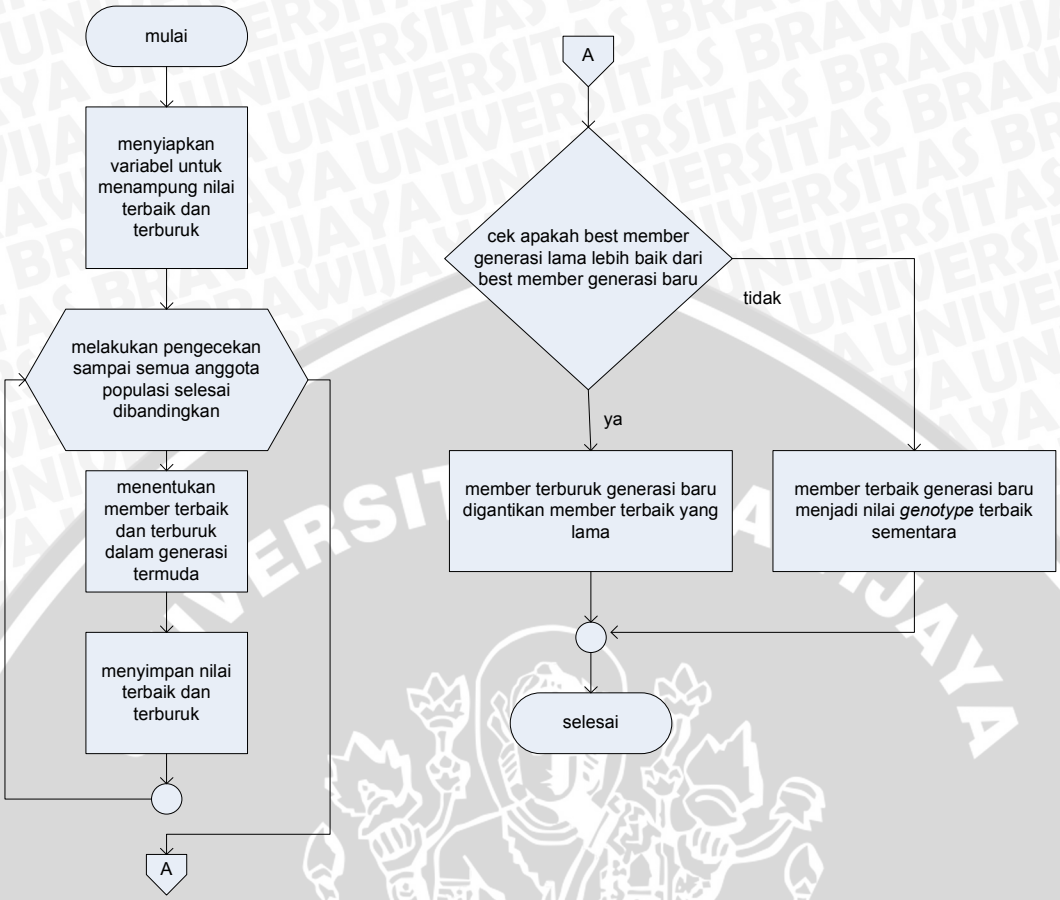
1. Sebuah variabel disiapkan sebagai penanda indeks lokasi member dengan nilai *fitness* tertinggi dalam populasi.
2. Dilakukan proses komparasi dari semua *genotype* dalam satu generasi menurut nilai *fitness* masing-masing.

3. Indeks dari lokasi genotype dengan fitness terbaik akan disimpan dalam variabel yang sudah disiapkan.
4. Nilai semua variabel solusi yang terdapat di dalam genotype dengan fitness terbaik akan disalin dalam tempat paling ujung yang disediakan dalam array memanfaatkan lokasi yang tidak terpakai.

4.4.5 Perancangan Prosedur *Elitist*

Elitist merupakan sebuah sistem di alam dimana anggota terkuat dari satu generasi atau *Alpha* berhak untuk tetap tinggal dalam populasi yang berisi keturunan terbaru apabila ia lebih kuat daripada *Alpha* dalam generasi selanjutnya tersebut. Jika nilai *fitness* dari *genotype* terbaik dari generasi lama lebih baik daripada generasi baru, maka *genotype* itu akan menggantikan posisi *genotype* dengan *fitness* paling buruk di generasi terbaru. Hal ini untuk mencegah hilangnya sifat-sifat yang sudah baik yang mungkin terjadi melalui proses *crossover* maupun mutasi yang murni berdasarkan probabilitas.

Berikut ini merupakan *flowchart* prosedur *elitist*:



Gambar 4.5 *Flowchart* prosedur *elitist*

Berikut penjelasan *flowchart* dari prosedur *elitist*:

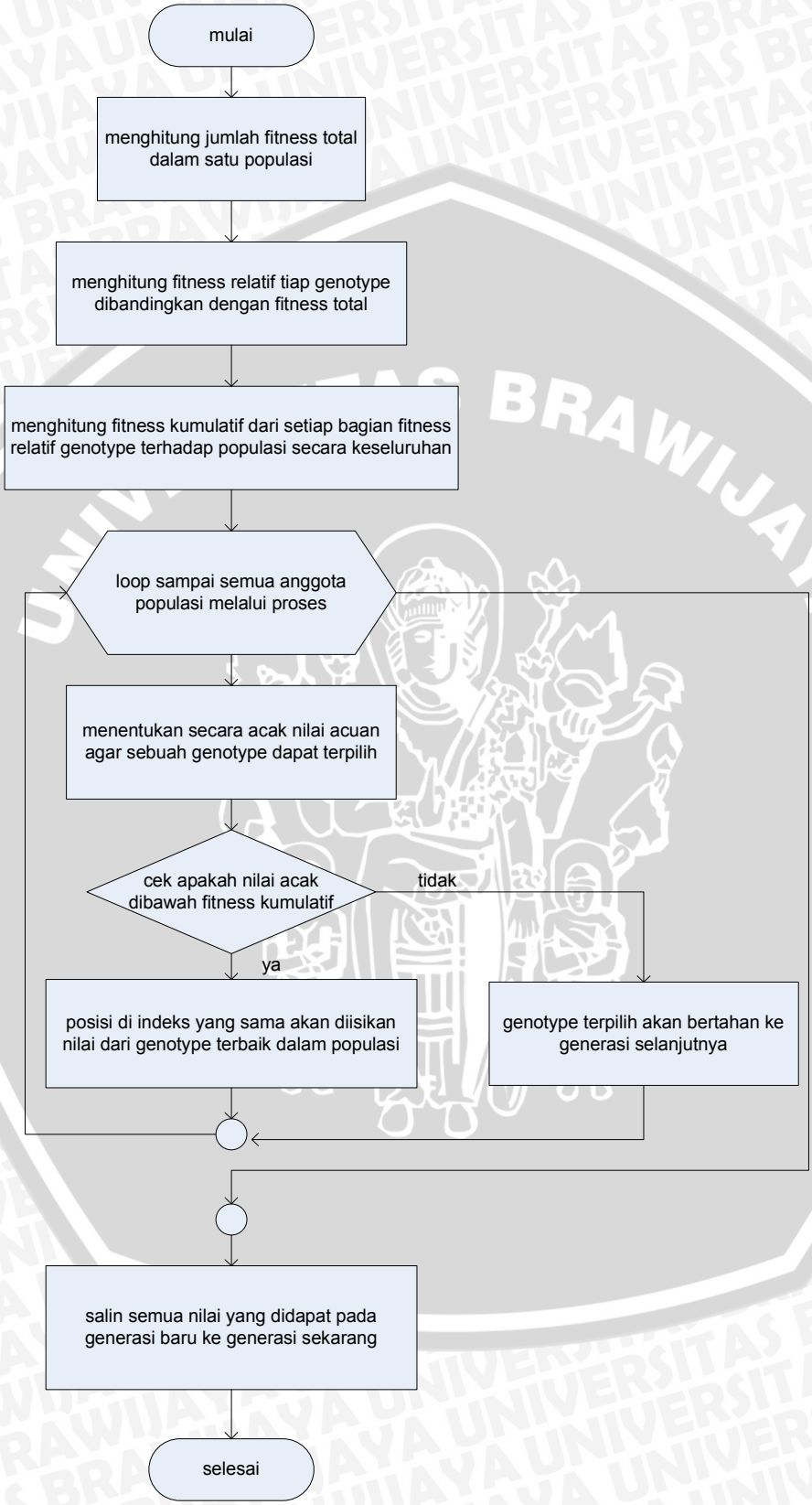
1. Sepasang variabel disiapkan untuk menampung nilai terbaik dan terburuk dari generasi yang paling muda.
2. Proses loop hingga semua anggota dalam populasi selesai melalui proses *check*.
3. *Genotype* dengan nilai *fitness* terbaik dan terburuk dicari dengan membandingkan semua anggota dalam satu populasi.
4. Lokasi dan nilai *genotype* terbaik dan terburuk disimpan dalam variabel yang sudah disiapkan.
5. Proses bercabang dengan seleksi kondisi perbandingan *best member* dari generasi yang lama dengan *best member* dari generasi yang baru.
6. Bila *best member* dari generasi yang baru lebih baik, maka *genotype* itu akan menjadi yang terbaik sementara.

7. Bila *best member* dari generasi yang lama lebih baik, maka *genotype* itu akan menggantikan posisi anggota terburuk dari generasi yang baru di dalam populasi.

4.4.6 Perancangan Prosedur Seleksi

Alam memiliki proses seleksi secara natural, dimana yang kuat akan memiliki peluang bertahan hidup lebih tinggi daripada yang lemah. *Library Genetic Algorithm* menyediakan sebuah prosedur untuk menentukan *genotype* mana yang dapat bertahan ke dalam generasi yang baru untuk membentuk populasi baru. *Genotype* dengan *fitness* yang tinggi memiliki peluang hidup lebih tinggi walaupun tidak selalu bisa bertahan menuju generasi berikutnya karena proses seleksi yang melibatkan unsur acak atau *random*.

Berikut ini *flowchart* prosedur seleksi:



Gambar 4.6 *Flowchart* prosedur seleksi

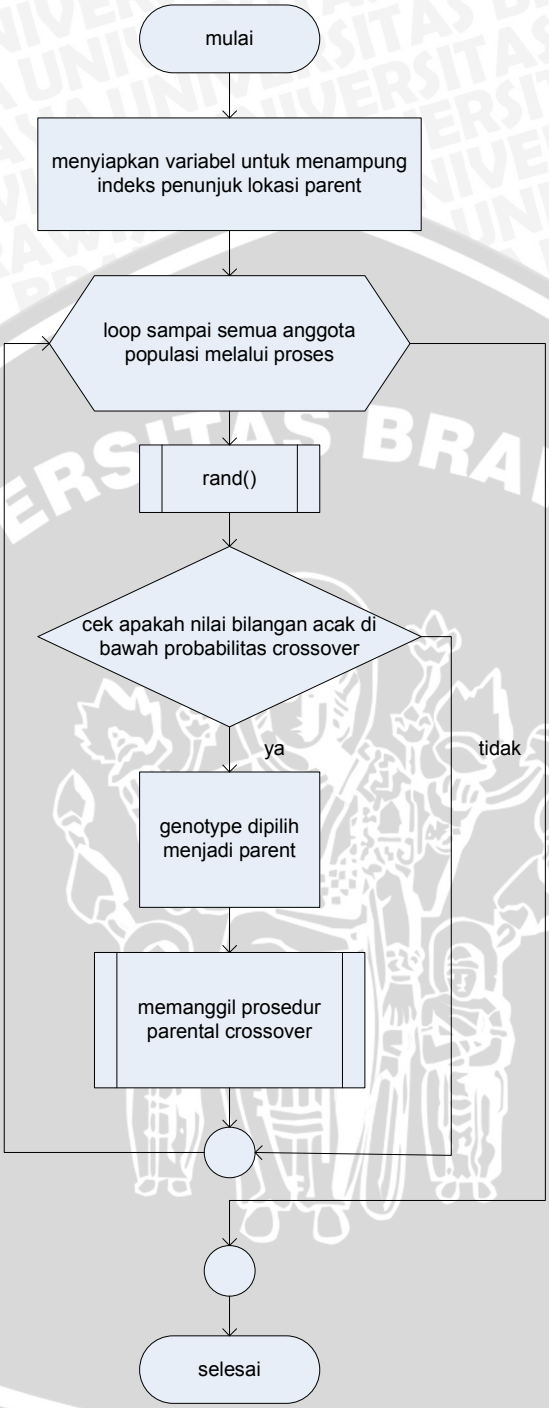
Berikut ini penjelasan *flowchart* prosedur seleksi:

1. Semua nilai *fitness* dalam populasi dijumlahkan hingga didapatkan nilai *fitness* total.
2. Nilai *fitness* tiap *genotype* dibandingkan dengan *fitness* total untuk mendapatkan nilai *fitness* relatif.
3. Setiap bagian dari komponen *fitness* relatif tiap *genotype* dijumlahkan untuk mendapatkan nilai *fitness* kumulatif setiap indeks.
4. Proses *loop* sesuai jumlah *genotype* dalam populasi agar semua anggota melalui proses seleksi.
5. Menentukan satu angka acak yang menjadi acuan apakah *genotype* yang sedang ditunjuk layak untuk bertahan ke generasi selanjutnya.
6. Cek nilai acak terhadap *fitness* kumulatif dari indeks yang sedang ditunjuk *loop*.
7. Jika *fitness* kumulatif lebih tinggi dari bilangan acak yang didapatkan, posisi *genotype* di indeks yang sedang ditunjuk akan diisikan data dari anggota terbaik.
8. Jika nilai bilangan acak yang lebih tinggi maka *genotype* dengan indeks yang sedang ditunjuk akan bertahan ke generasi yang baru.
9. Generasi baru menjadi generasi yang sekarang.

4.4.7 Perancangan Prosedur *Crossover*

Prosedur *crossover* menangani proses *coupling* atau pemilihan *parent* yang akan mengalami pindah-silang.

Berikut ini *flowchart* prosedur *crossover*:



Gambar 4.7 *Flowchart* prosedur *crossover*

Berikut ini penjelasan *flowchart* prosedur *crossover*:

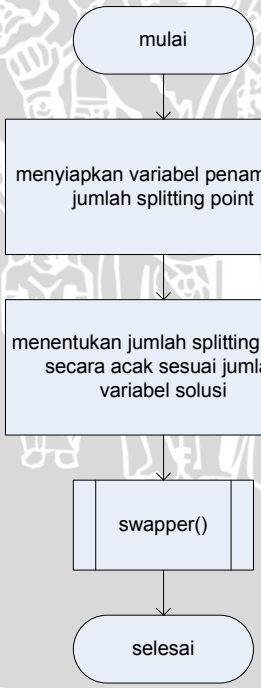
1. Variabel untuk menampung indeks penunjuk lokasi parent disiapkan lebih dahulu.

2. Loop dijalankan hingga semua anggota populasi selesai diproses.
3. Memanggil fungsi `rand()` yang terdapat dalam *library* standart dengan *header* `stdlib.h` untuk memilih satu bilangan acak.
4. Cek apakah nilai bilangan acak keluaran fungsi `rand()` lebih besar dari parameter probabilitas *crossover*.
5. Jika nilai acak lebih besar, maka *genotype* yang ditunjuk oleh indeks *loop* dipilih menjadi *parent*.
6. Prosedur untuk *parental crossover akan* dipanggil, dengan *parent* yang dipilih sebagai argumen.

4.4.8 Perancangan Prosedur *Parental Crossover*

Prosedur *crossover* sebelumnya adalah bagian dari *crossover* yang menangani proses pemilihan *parent*. Kedua parent yang dipilih akan disilangkan dalam prosedur *parental crossover*. Prosedur ini menggunakan algoritma *multi-point crossover*.

Berikut ini *flowchart* prosedur *parental crossover*:



Gambar 4.8 *Flowchart* prosedur *parental crossover*

Berikut ini penjelasan *flowchart* prosedur *parental crossover*:

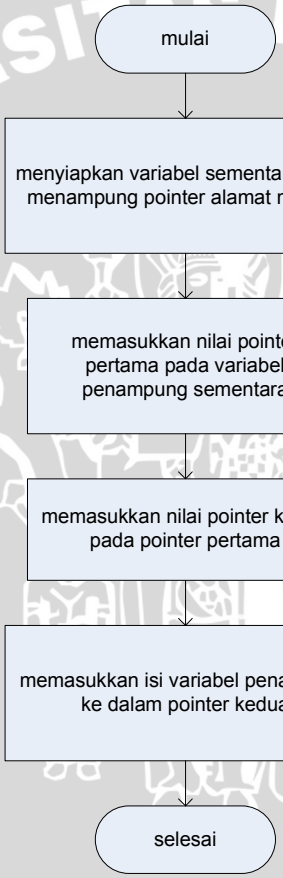
1. Variabel untuk menampung jumlah *splitting point* disiapkan lebih dahulu.

2. Jumlah splitting point ditentukan secara acak sesuai jumlah variabel solusi.
3. Memanggil prosedur `swapper()` untuk menukar alamat memori dari kedua parent.

4.4.9 Perancangan Prosedur *Swapper*

Prosedur *swapper* adalah sebuah prosedur pembantu untuk menukar alamat memori dari dua variabel.

Berikut ini *flowchart* dari prosedur *swapper*:



Gambar 4.9 *Flowchart* prosedur *swapper*

Berikut ini penjelasan *flowchart* prosedur *swapper*:

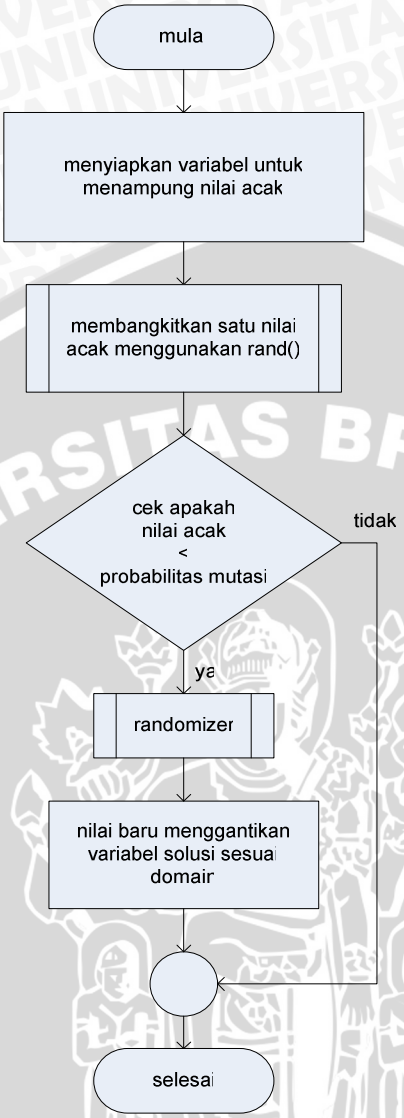
1. Sebuah variabel penampung sementara disiapkan terlebih dahulu.
2. Nilai alamat memori dari *pointer* pertama dimasukkan dalam variabel penampung sementara.

3. Nilai alamat memori dari *pointer* kedua dimasukkan dalam *pointer* pertama.
4. Isi dari variabel penampung sementara, yaitu nilai alamat memori dari *pointer* pertama, dimasukkan dalam *pointer* kedua.

4.4.10 Perancangan Prosedur Mutasi

Prosedur mutasi adalah sebuah prosedur yang menangani proses pembuatan satu nilai *genotype* baru tanpa melalui proses persilangan. Prosedur ini merupakan simulasi dari proses mutasi di alam, dan dapat digunakan untuk simulasi proses reproduksi aseksual yang menghasilkan keturunan baru dengan sifat yang sama sekali berbeda dari induk tunggal generasi sebelumnya. Prosedur mutasi yang digunakan adalah mutasi *uniform*. Mutasi dilakukan untuk setiap variabel solusi dalam *genotype* yang merupakan anggota dari satu populasi.

Berikut ini *flowchart* dari prosedur mutasi:



Gambar 4.10 *Flowchart* prosedur mutasi

Berikut ini penjelasan *flowchart* prosedur mutasi:

1. Sebuah variabel penampung disiapkan untuk nilai acak.
2. Memanggil fungsi `rand()` yang terdapat dalam *library* standart dengan *header* `stdlib.h` untuk memilih satu bilangan acak.
3. Melakukan seleksi kondisi apakah nilai bilangan acak lebih kecil daripada parameter probabilitas mutasi.
4. Jika memenuhi syarat, maka proses dilanjutkan dengan pemanggilan prosedur pembangkit nilai acak yang menggunakan dua buah argumen sebagai batasan.

5. Nilai baru menggantikan nilai gen dari variabel solusi pada *genotype* yang terpilih.

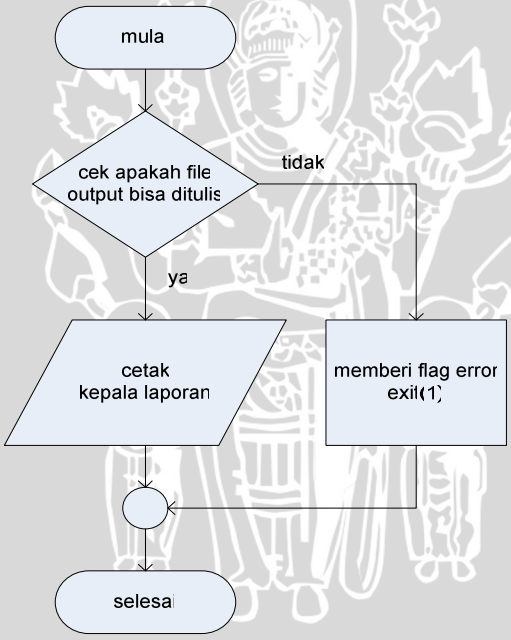
4.4.11 Perancangan Prosedur Pembuat Laporan

Hasil dari *library genetic algorithm* ini akan disimpan dalam bentuk *log* yang terdapat dalam sebuah *text file*. Mekanisme penulisan laporan akan ditangani oleh tiga buah prosedur pembuat laporan.

4.4.11.1 Perancangan Prosedur Pembuat Kepala Laporan

Sebelum proses masuk ke dalam loop seleksi dan reproduksi, diperlukan sebuah kepala laporan yang berfungsi membuka *file output* dan menuliskan bagian awal dari *log*.

Berikut ini *flowchart* dari prosedur pembuat kepala laporan:



Gambar 4.11 *Flowchart* prosedur pembuat kepala laporan

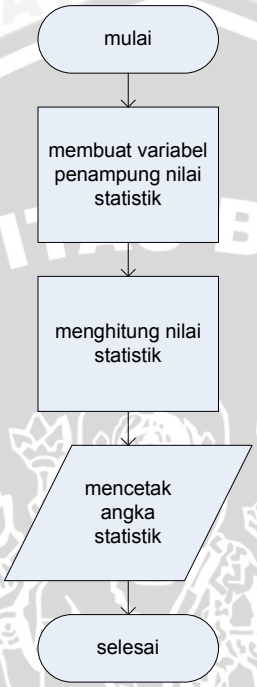
Berikut ini penjelasan *flowchart* prosedur pembuat kepala laporan:

1. Melakukan seleksi kondisi apakah *file output* dapat ditulis.
2. Jika *file* dapat dibuka dengan mode tulis, maka kepala laporan dicetak di dalamnya.
3. Jika *file* tidak dapat dibuka dengan mode tulis, maka program akan *terminate* dan memberi pesan *error* berupa `exit(1)` pada *compiler*.

4.4.11.2 Perancangan Prosedur Pembuat Badan Laporan

Prosedur pembuat badan laporan berfungsi untuk mencetak data ke dalam *log* pada saat proses *loop* dijalankan. Prosedur ini juga sekaligus berisi operasi statistik seperti menghitung nilai *fitness* rata-rata dan standart deviasi.

Berikut ini *flowchart* dari prosedur pembuat badan laporan:



Gambar 4.12 *Flowchart* prosedur pembuat badan laporan

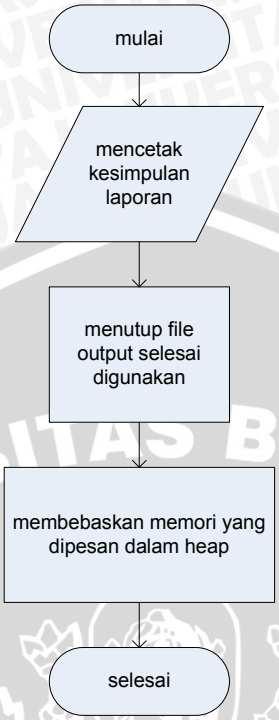
Berikut ini penjelasan dari *flowchart* prosedur pembuat badan laporan:

1. Menyiapkan variabel yang akan digunakan dalam operasi mencari nilai statistik dari tiap generasi.
2. Melakukan operasi menghitung angka statistik.
3. Mencetak angka-angka yang didapatkan ke dalam *log*.

4.4.11.3 Perancangan Prosedur Pembuat Kesimpulan Laporan

Prosedur pembuat kesimpulan laporan berfungsi untuk mencetak kesimpulan dari data-data yang didapatkan ke dalam *log* pada saat proses *loop* selesai dijalankan. Prosedur ini juga sekaligus bertugas menutup *file output* saat program selesai dijalankan. Sebagai prosedur terakhir yang dijalankan, semua pointer yang dialokasikan selama program berjalan akan dibebaskan melalui prosedur ini.

Berikut ini *flowchart* dari prosedur pembuat kesimpulan laporan:



Gambar 4.13 *Flowchart* prosedur pembuat kesimpulan laporan

Berikut ini penjelasan dari *flowchart* prosedur pembuat kesimpulan laporan:

1. Mencetak kesimpulan dari laporan ke dalam *log*.
2. Menutup *file output* setelah seluruh proses selesai dijalankan.
3. Semua memori yang dipesan untuk *array* dinamis dibebaskan.

4.5. Implementasi

Setelah tahap perancangan selesai dilakukan, tahap selanjutnya adalah tahap implementasi. Tujuan dari tahap implementasi ini merupakan proses transformasi hasil perancangan perangkat lunak. Pembahasan terdiri dari lingkungan implementasi (spesifikasi perangkat lunak dan perangkat keras), implementasi antarmuka aplikasi dengan sintaks dari bahasa pemrograman yang digunakan.

4.5.1 Lingkungan Implementasi

Library dan program pengantar dibuat dalam bahasa pemrograman C. Penulis menggunakan lingkungan implementasi dengan perangkat keras berupa sebuah laptop dan beberapa perangkat lunak.

4.5.1.1 Lingkungan Implementasi Perangkat Keras

Sebuah laptop dengan spesifikasi sebagai berikut:

- *Processor* : Intel(R) Core(TM) i5-2410M; 4 CPU @2.30 GHz
- *Memory* : 4096MB RAM
- *System type* : 32-bit Operating System
- *System model* : Satellite L735

4.5.1.2 Lingkungan Implementasi Perangkat Lunak

Spesifikasi perangkat lunak yang digunakan:

- Sistem operasi : Windows 7 Ultimate 32-bit (6.1, Build 7600)
- Bahasa pemrograman : C
- *Integrated Development Environment* (IDE) : Bloodshed Dev-C++ 4.9.9.2
- GCC *software port* : MinGW 4.6.2
- *Compiler* : GNU GCC 3.2

4.5.2 Implementasi Prosedur *Library*

Library Genetic Algorithm ini menggunakan bahasa pemrograman C. Semua prosedur yang ditulis akan diterjemahkan oleh GCC compiler dan dibangun dengan pendekatan *procedural programming. Archive* yang terbentuk berisi kumpulan prosedur yang diimplementasikan dalam bahasa pemrograman C.

4.5.2.1 Implementasi Prosedur Pembacaan Parameter

Prosedur pembacaan parameter merupakan prosedur pertama yang dipanggil dalam *library*. Prosedur ini dalam bagian *header archive* ditulis sebagai prosedur `gaparam(int, int, int, double, double)`. Semua argumen dari prosedur ini merupakan beberapa parameter yang dibutuhkan algoritma genetika, yang dimasukkan user saat *executable* dijalankan. Prosedur ini tidak memberikan nilai kembalian.

DESCRIPTION:

```
1    void gaparam(int nvar, int popsize, int maxgens, double
2    pcross, double pmutation)
3    {
4    int n;
5    population = (mygen*) malloc(sizeof(mygen)*(popsize+1));
```

```
6    for(n=0; n<(popsize+1); n++)
7            {
8            population[n].gen = (double*) malloc
9    (nvar*sizeof(double));
10           population[n].upbound = (double*) malloc
11   (nvar*sizeof(double));
12           population[n].lowbound = (double*) malloc
13   (nvar*sizeof(double));
14           }
15   nextpop = (mygen*) malloc(sizeof(mygen)*(popsize+1));
16   for(n=0; n<(popsize+1); n++)
17           {
18           nextpop[n].gen = (double*) malloc
19   (nvar*sizeof(double));
20           nextpop[n].upbound = (double*) malloc
21   (nvar*sizeof(double));
22           nextpop[n].lowbound = (double*) malloc
23   (nvar*sizeof(double));
24           }
25
26   NVAR = nvar;
27   POPSIZE = popsize;
28   MAXGENS = maxgens;
29   PCROSS = pcross;
30   PMUTATION = pmutation;
31 }
32
```

Gambar 4.14 Deskripsi prosedur pembacaan parameter

Berikut ini penjelasan dari deskripsi prosedur pemberian nilai awal:

1. Baris 1 dan 2 adalah nama dan tipe data prosedur, juga argumen yang dibutuhkan. Prosedur `void initialisation(void)` tidak memberi nilai kembalian.
2. Baris 4 adalah variabel lokal untuk proses *loop*.
3. Baris 5-14 adalah proses alokasi memori untuk pointer `*population` yang digunakan sebagai penanda indeks pertama array dinamis populasi sekarang, dan alokasi memori semua *array* dinamis anggota dari struktur.
4. Baris 16-24 adalah proses alokasi memori untuk pointer `*nextpop` yang digunakan sebagai penanda indeks pertama array dinamis populasi untuk generasi selanjutnya, dan alokasi memori semua *array* dinamis anggota dari struktur.
5. Semua nilai dalam argumen disalin dalam variabel global pada baris 26-30 agar dapat digunakan semua prosedur dalam *library*.

4.5.2.2 Implementasi Prosedur Pemberian Nilai Awal

Prosedur pemberian nilai awal ditulis dalam *archive* sebagai prosedur `initialisation()`yang tidak memerlukan argumen dan tidak memberikan nilai kembalian.

DESCRIPTION:

```
1   void initialisation(void)
2   {
3   srand(time(0));                 //random seeder
4   double lbound, ubound;
5   int i,j;
6
7   FILE *infile;
8   if ((infile = fopen("boundlist.txt","r"))==NULL)
9   //open boundlist file
10     {
11     fprintf(galist,"\nError in opening boundlist.txt!\n");
12     exit(1);
13     }
14
15  for (i=0; i<NVAR; i++)
16     {
17       fscanf(infile, "%lf", &lbound);
18       fscanf(infile, "%lf", &ubound);
19       for (j=0; j<POPSIZE; j++)
20           {
21           population[j].lowbound[i] = lbound;
22           population[j].upbound[i] = ubound;
23           population[j].gen[i] =
24  randomizer(population[j].lowbound[i],population[j].upbound[i]);
25           population[j].fitness = 0;
26           population[j].rfitness = 0;
27           population[j].cfitness = 0;
28           }
29       }
30  fclose(infile);     //close boundlist file
31 }
32
```

Gambar 4.15 Deskripsi prosedur pemberian nilai awal

Berikut ini penjelasan dari deskripsi prosedur pemberian nilai awal:

1. Baris 1 adalah nama dan tipe data prosedur, juga argumen yang dibutuhkan. Prosedur `void initialisation(void)` tidak membutuhkan argumen dan tidak memberi nilai kembalian.
2. Baris 3 adalah proses *seed* dari fungsi `rand()`di dalam `stdlib.h` agar fungsi pengacak menghasilkan nilai yang berbeda-beda setiap kali dijalankan. Parameter yang digunakan untuk `srand()` adalah fungsi `time()` yang ada di dalam library `time.h`.

3. Baris 4-5 adalah deklarasi variabel lokal yang digunakan di dalam prosedur ini.
4. Baris 7-13 adalah proses pembacaan *file* masukan `boundlist.txt` diawali dengan menggunakan pointer `FILE *infile` dan memberikan pesan error berupa terminasi `exit(1)` dan pesan error dalam *file output* saat *file* yang ditunjuk tidak dapat dibuka dalam mode *read*.
5. Baris 15-29 adalah proses pembacaan batas bawah dan batas atas dari dalam file masukan `boundlist.txt` setiap variabel solusi.
6. Baris 30 adalah proses menutup file masukan `boundlist.txt` sebelum prosedur selesai dijalankan.

4.5.2.3 Implementasi Prosedur Pembangkit Nilai Acak

Prosedur pembangkit nilai acak ditulis dalam bagian deklarasi *prototype* pada *archive* sebagai prosedur `double randomizer(double, double)` yang membutuhkan sepasang argumen dengan tipe data *double* dan memberikan nilai kembalian sebuah nilai acak dengan tipe data *double*.

```
DESCRIPTION:

1    double randomizer(double lb, double hb)
2    {
3    double ranval=((double)(rand()%1000)/1000.0)*(hb-lb)+lb;
4    return(ranval);
5    }
6
```

Gambar 4.16 Deskripsi prosedur pembangkit nilai acak

Berikut ini penjelasan dari deskripsi prosedur pemberian nilai awal:

1. Baris 1 adalah nama dan tipe data prosedur, juga argumen yang dibutuhkan. Prosedur `randomizer` membutuhkan dua argumen yang digunakan sebagai batasan nilai acak paling kecil dan nilai paling besar.
2. Baris 3 terdiri dari beberapa proses yang ditulis dalam satu baris. Proses pertama adalah deklarasi *inline* dari variabel penampung nilai acak di sebelah kiri operator penugasan. Proses berikutnya adalah pembangkitan sebuah nilai acak melalui fungsi `rand()` berdasarkan nilai *seed* yang ditentukan dalam prosedur `initialisation`. Proses terakhir di baris ini adalah manipulasi nilai acak dengan operator aritmatika agar hasil akhir

sesuai dengan batasan nilai paling besar dan paling kecil yang ditentukan melalui argumen prosedur.

3. Nilai acak yang ditampung dalam variabel lokal `ranval` digunakan sebagai nilai kembalian untuk proses yang memanggil prosedur ini.

4.5.2.4 Implementasi Prosedur Penyimpan Nilai Terbaik

Prosedur penyimpan nilai terbaik ditulis dalam *archive* sebagai prosedur `bestmember(void)` yang tidak membutuhkan argumen dan tidak memberikan nilai kembalian.

```
DESCRIPTION:

1    void bestmember(void)
2    {
3    int i, j;
4    int the_best = 0;
5    for (i=0; i<POPSIZE; i++)
6        {
7        if (population[i].fitness >
8    population[POPSIZE].fitness)
9            {
10           the_best=i;
11           population[POPSIZE].fitness =
12   population[i].fitness;
13           }
14       }
15   for (j=0; j<NVAR; j++)
16       population[POPSIZE].gen[j] =
17   population[the_best].gen[j]; //copy all best genes
18   }
19
```

Gambar 4.17 Deskripsi prosedur penyimpan nilai terbaik

Berikut ini penjelasan deskripsi prosedur penyimpan nilai terbaik:

1. Baris 1 adalah nama dan tipe data prosedur, juga argumen yang dibutuhkan. Prosedur `bestmember` tidak membutuhkan argumen dan tidak memberi nilai kembalian.
2. Baris 3 dan 4 adalah variabel lokal yang digunakan dalam prosedur ini.
3. Baris 5-14 adalah proses mencari lokasi *genotype* dalam populasi yang memiliki nilai *fitness* terbaik. Proses berlangsung dengan cara *loop* mulai dari indeks pertama hingga terakhir pada populasi, nilai terbaik disimpan pada bagian akhir *array*. Semua nilai *fitness* dalam populasi akan

dibandingkan dengan nilai ini, jika nilainya lebih baik maka nilai itu akan menggantikan nilai terbaik.

4. Baris 15-17 adalah proses menyalin semua nilai gen terbaik berdasarkan perbandingan *fitness* yang ditemukan.

4.5.2.5 Implementasi Prosedur *Elitist*

Prosedur *elitist* ditulis dalam *archive* sebagai prosedur `elitist(void)` yang tidak membutuhkan argumen dan tidak memberikan nilai kembalian.

```
DESCRIPTION:

1    void elitist()
2    {
3    int i;
4    double best, worst;
5    int best_mem, worst_mem;
6
7    best = population[0].fitness;
8    worst = population[0].fitness;
9    for(i=0; i < POPSIZE - 1; ++i)
10           {
11           if (population[i].fitness >
12   population[i+1].fitness)
13              {
14              if (population[i].fitness >= best)
15                 {
16                 best = population[i].fitness;
17                 best_mem = i;
18                 }
19              if (population[i].fitness <= worst)
20                 {
21                 worst = population[i+1].fitness;
22                 worst_mem = i+1;
23                 }
24              }
25           else
26               {
27                if (population[i].fitness <= worst)
28                 {
29                 worst = population[i].fitness;
30                 worst_mem = i;
31                 }
32                if (population[i+1].fitness >= best)
33                 {
34                 best = population[i+1].fitness;
35                 best_mem = i+1;
36                 }
37               }
38           }
39   if (best >= population[POPSIZE].fitness)
40      {
41      for(i=0; i<NVAR; i++)
42               population[POPSIZE].gen[i] =
43   population[best_mem].gen[i];
44      population[POPSIZE].fitness =
```

```
45   population[best_mem].fitness;
46      }
47   else
48      {
49      for(i=0; i<NVAR; i++)
50               population[worst_mem].gen[i] =
51   population[POPSIZE].gen[i];
52      population[worst_mem].fitness =
53   population[POPSIZE].fitness;
54      }
55  }
56
```

Gambar 4.18 Deskripsi prosedur *elitist*

Berikut ini penjelasan deskripsi prosedur *elitist*:

1. Baris 1 adalah nama dan tipe data prosedur, juga argumen yang dibutuhkan. Prosedur `elitist` tidak membutuhkan argumen dan tidak memberi nilai kembalian.
2. Baris 3 -5 adalah variabel lokal yang digunakan dalam prosedur ini.
3. Baris 9-38 adalah proses untuk mencari anggota terbaik dan terburuk dalam generasi yang paling muda. Anggota terbaik dan terburuk ditentukan dengan membandingkan nilai *fitness genotype*.
4. Baris 39-54 adalah proses membandingkan nilai terbaik dari generasi sebelumnya yang disimpan pada indeks *array* paling belakang, dengan nilai terbaik generasi termuda yang didapatkan dari proses nomor 3 di atas. Jika nilai *fitness* dari *genotype* terbaik generasi termuda lebih baik daripada nilai terbaik yang dimiliki generasi sebelumnya, maka nilai *fitness* dan gen yang terbaik ini menggantikan nilai terbaik yang sebelumnya ada. Tetapi jika nilai terbaik dari generasi termuda tidak lebih baik, maka posisi *genotype* terburuk dari generasi sekarang akan digantikan posisinya oleh *genotype* terbaik generasi sebelumnya.

4.5.2.6 Implementasi Prosedur Seleksi

Prosedur seleksi ditulis dalam *archive* sebagai prosedur `selection(void)` yang tidak membutuhkan argumen dan tidak memberikan nilai kembalian.

```
DESCRIPTION:

1    void selection(void)
2    {
3    int mem, i, j, k;
```

```
4    double p, sum=0;
5
6    for(mem=0; mem<POPSIZE; mem++)
7        sum += population[mem].fitness;
8    //total fitness
9    for(mem=0; mem<POPSIZE; mem++)
10       population[mem].rfitness =
11   population[mem].fitness/sum;
12   population[0].cfitness = population[0].rfitness;
13   //relative fitness
14   for(mem=1; mem<POPSIZE; mem++)
15    population[mem].cfitness = population[mem-
16   1].cfitness+population[mem].rfitness;
17   //total cfitness
18
19   for(i=0; i<POPSIZE; i++)
20       {
21       p = rand()%1000/1000.0;
22       if(p<population[0].cfitness)
23           nextpop[i] = population[0];
24       else
25           {
26           for(j=0; j<POPSIZE; j++)
27                {
28                if(p>=population[j].cfitness &&
29    p<population[j+1].cfitness)
30                     nextpop[i] = population[j+1];
31                }
32           }
33       }
34    for(i=0; i<POPSIZE; i++)
35       population[i] = nextpop[i];
36   }
37
```

Gambar 4.19 Deskripsi prosedur seleksi

Berikut ini penjelasan deskripsi prosedur seleksi:

1. Baris 1 adalah nama dan tipe data prosedur, juga argumen yang dibutuhkan. Prosedur `selection` tidak membutuhkan argumen dan tidak memberi nilai kembalian.
2. Baris 3 dan 4 adalah variabel lokal yang digunakan dalam prosedur ini.
3. Baris 6-7 adalah proses menghitung *fitness* total dalam populasi.
4. Baris 9-12 adalah proses menghitung *fitness* relatif terhadap *fitness* total dengan cara membagi *fitness* dari anggota yang ditunjuk index dengan *fitness* total.
5. Baris 14-16 adalah proses menghitung nilai *fitness* kumulatif tiap urutan *genotype* dimulai dari pasangan *genotype* awal dan kedua, hingga semuanya selesai dijumlahkan. Hal ini untuk menentukan lebarnya

daerah kemungkinan suatu *genotype* dapat dipilih tanpa harus mengurutkan semua *genotype* terlebih dahulu. *Genotype* dengan nilai *fitness* yang besar akan memiliki daerah yang lebih lebar sehingga peluang terpilih dalam seleksi lebih tinggi.

6. Baris 19-33 adalah proses menentukan daerah mana yang dipilih berdasarkan sebuah nilai acak dibandingkan dengan nilai *fitness* kumulatif dari proses 5 di atas. Semua genotype terpilih ditampung di dalam array `nextpop`.
7. Baris 34-35 adalah proses memindah semua anggota baru dalam array `nextpop` ke dalam populasi yang sekarang.

4.5.2.7 Implementasi Prosedur *Crossover*

Prosedur seleksi ditulis dalam *archive* sebagai prosedur `crossover (void)` yang tidak membutuhkan argumen dan tidak memberikan nilai kembalian.

```
DESCRIPTION:

1    void crossover()
2    {
3    int i, mem, one, pertamax =0;
4    double x;
5
6    for(mem=0; mem<POPSIZE; ++mem)
7        {
8        x = rand()%1000/1000.0;
9        if(x < PCROSS)
10           {
11           ++pertamax;
12           if(pertamax%2 == 0)
13               parentalcross(one,mem);
14           else
15               one = mem;
16           }
17       }
18   }
19
```

Gambar 4.20 Deskripsi prosedur *crossover*

Berikut ini penjelasan deskripsi prosedur *crossover*:

1. Baris 1 adalah nama dan tipe data prosedur, juga argumen yang dibutuhkan. Prosedur `crossover` tidak membutuhkan argumen dan tidak memberi nilai kembalian.
2. Baris 3 dan 4 adalah variabel lokal yang digunakan dalam prosedur ini.

3. Baris 6-17 adalah proses *coupling*, mencari pasangan *genotype* yang akan menjadi *parent* dalam *crossover*. Proses dimulai dengan membangkitkan sebuah nilai acak antara 0 hingga 1 dan membandingkannya dengan nilai probabilitas *crossover* `PCROSS`. Jika nilai acak lebih kecil maka *genotype* itu terpilih menjadi *parent*. Sebuah *parent* akan melalui proses persilangan saat pasangannya sudah ditemukan. Prosedur ini memanggil `parentalcross(int one, int two)` setiap kali sepasang *parent* terpilih.

4.5.2.8 Implementasi Prosedur *Parental Crossover*

Prosedur *Parental Crossover* ditulis dalam *archive* sebagai prosedur `void parentalcross(int, int)` yang membutuhkan sepasang argumen dengan tipe data *integer*. Prosedur ini tidak memberikan nilai kembalian.

DESCRIPTION:

```
1    void parentalcross(int one, int two)
2    {
3    int i, point;
4    if(NVAR>1)
5        {
6        if(NVAR ==2)
7            point =1;
8        else
9            point = (rand() % (NVAR-1))+1;
10       for(i=0; i<point; i++)
11
12   swapper(&population[one].gen[i],&population[two].gen[i]);
13       }
14   }
15
```

Gambar 4.21 Deskripsi prosedur *parental crossover*

Berikut ini penjelasan deskripsi prosedur *crossover*:

1. Baris 1 adalah nama dan tipe data prosedur, juga argumen yang dibutuhkan. Prosedur `parentalcrossover` membaca dua argumen berisi indeks dari sepasang *parent* yang terpilih untuk proses *crossover*.
2. Baris 3 adalah variabel lokal yang digunakan dalam prosedur ini.
3. Baris 4 adalah seleksi kondisi yang menjadi syarat agar proses persilangan dapat dilakukan, yaitu hanya jika jumlah variabel solusi lebih dari satu.

4. Baris 6-9 adalah proses mencari jumlah pasangan gen yang akan ditukar. Jumlah ini akan ditentukan secara acak jika jumlah variabel solusi ada lebih dari dua.
5. Baris 10-12 adalah proses penukaran gen yang memiliki indeks sama antara kedua *parent* terpilih. Proses penukaran ini dilakukan di dalam prosedur `swapper` dengan kedua alamat memori dari gen *parent* sebagai argumen referensi.

4.5.2.9 Implementasi Prosedur *Swapper*

Prosedur *swapper* ditulis dalam *archive* sebagai prosedur `void swapper(double *, double *)` yang membutuhkan sepasang argumen berupa pointer yang menunjuk lokasi alamat variabel dengan tipe data *double*. Prosedur ini tidak memberikan nilai kembalian.

```
DESCRIPTION:

1    void swapper(double *x, double *y)
2    {
3    double temp;
4    temp= *x;
5    *x = *y;
6    *y = temp;
7   }
8
```

Gambar 4.22 Deskripsi prosedur *swapper*

Berikut ini penjelasan deskripsi prosedur *swapper*:

1. Baris 1 adalah nama dan tipe data prosedur, juga argumen yang dibutuhkan. Prosedur `swapper` membaca dua argumen berupa pointer yang menunjuk lokasi alamat kedua variabel referensi.
2. Baris 3 adalah variabel lokal yang digunakan sebagai penampung sementara dalam prosedur ini.
3. Baris 4-6 adalah proses penukaran alamat dari dua variabel referensi dengan menggunakan bantuan `temp`.

4.5.2.10 Implementasi Prosedur Mutasi

Prosedur mutasi ditulis dalam *archive* sebagai prosedur `void mutation(void)` yang tidak membutuhkan argumen dan tidak memberikan nilai kembalian.

DESCRIPTION:

```
1    void mutation(void)
2    {
3    int i,j;
4    double lbound, hbound, x;
5
6    for(i=0; i<POPSIZE; i++)
7        for(j=0; j<NVAR; j++)
8            {
9            x = rand()%1000/1000.0;
10           if(x<PMUTATION)
11                   {
12                   lbound = population[i].lowbound[j];
13                   hbound = population[i].upbound[j];
14                   population[i].gen[j] = randomizer(lbound,
15   hbound);
16                   }
17           }
18   }
19
```

Gambar 4.23 Deskripsi prosedur mutasi

Berikut ini penjelasan deskripsi prosedur mutasi:

1. Baris 1 adalah nama dan tipe data prosedur, juga argumen yang dibutuhkan. Prosedur `mutation` tidak membutuhkan argumen dan tidak memberi nilai kembalian.
2. Baris 3 dan 4 adalah variabel lokal yang digunakan dalam prosedur ini.
3. Baris 6 dan 7 adalah proses *loop* untuk mengakses semua indeks gen dalam populasi.
4. Baris 9 adalah proses memilih satu angka acak antara 0 sampai 1.
5. Baris 10 adalah proses seleksi kondisi, jika nilai acak pada proses 4 di atas menghasilkan nilai yang lebih kecil dari probabilitas mutasi `PMUTATION`, maka gen yang sedang ditunjuk indeks *loop* terpilih untuk proses mutasi.
6. Baris 12-15 adalah proses mengganti nilai gen terpilih dengan sebuah nilai acak sesuai batas atas dan batas bawah gen itu melalui prosedur `randomizer`.

4.5.2.11 Implementasi Prosedur Pembuat Laporan

Prosedur pembuat laporan dibagi menjadi tiga bagian yang semua *prototype*-nya ditulis pada bagian *header* dari *library*.

4.5.2.11.1 Implementasi Prosedur Pembuat Kepala Laporan

Prosedur pembuat kepala laporan ditulis dalam *archive* sebagai prosedur `void reporthead(void)` yang tidak membutuhkan argumen dan tidak memberikan nilai kembalian.

DESCRIPTION:

```
1   void reporthead(void)
2   {
3   if((galist=fopen("galist.txt","w"))==NULL)
4       exit(1);
5   fprintf(galist,"\n\t\t    LOG\n");
6   fprintf(galist,"\n_____________________________________________
7    _________\n\n");
8   fprintf(galist,"\n generation   best     average    standart
9    \n");
10  fprintf(galist,"\n(population)   value    fitness    deviation
11   \n");
12  fprintf(galist,"\n_____________________________________________
13   _________\n");
14  }
```

Gambar 4.24 Deskripsi prosedur pembuat kepala laporan

Berikut ini penjelasan deskripsi prosedur pembuat kepala laporan:

1. Baris 1 adalah nama dan tipe data prosedur, juga argumen yang dibutuhkan. Prosedur `reporthead` tidak membutuhkan argumen dan tidak memberi nilai kembalian.
2. Baris 3 dan 4 adalah terminasi program jika file keluaran `galist.txt` tidak dapat dibuka dengan mode *write*.
3. Baris 5-13 adalah penulisan kepala laporan dalam file keluaran `galist.txt`

4.5.2.11.2 Implementasi Prosedur Pembuat Badan Laporan

Prosedur pembuat badan laporan ditulis dalam *archive* sebagai prosedur `void report (int)` yang membutuhkan argumen dengan tipe data *integer*. Prosedur ini tidak memberikan nilai kembalian.

.

DESCRIPTION:

```
1    void report(int generation)
2    {
3    int i;
4    double best_val;              //best fitness value
5    double avg;                   //avg fitness
6    double stddev;                //square root of variance
7    double sum_square=0.0, square_sum, sum=0.0, extra;
8
9    for(i=0; i<POPSIZE; i++)
10       {
11       sum += population[i].fitness;
12       sum_square += population[i].fitness *
13   population[i].fitness;
14       }
15
16   avg = sum/(double)POPSIZE;
17   square_sum = avg*avg*(double)POPSIZE;
18   if(sum_square>square_sum)
19           stddev = sqrt((sum_square-square_sum)/(POPSIZE-
20   1));
21   else
22           stddev = sqrt((square_sum-sum_square)/(POPSIZE-
23   1));
24   best_val = population[POPSIZE].fitness;
25
26   fprintf(galist, "\n%5d.    %9.4f, %9.4f, %9.4f,
27   \n\n",generation,best_val,avg,stddev);
28   }
29
```

Gambar 4.25 Deskripsi prosedur pembuat badan laporan

Berikut ini penjelasan deskripsi prosedur pembuat badan laporan:

1. Baris 1 adalah nama dan tipe data prosedur, juga argumen yang dibutuhkan. Prosedur `report` membutuhkan nilai penghitung generasi sebagai argumen.
2. Baris 3-7 adalah variabel lokal yang digunakan dalam prosedur ini.
3. Baris 9-14 adalah proses penjumlahan semua nilai *fitness* dan penjumlahan kuadrat nilai *fitness* semua anggota dalam populasi.
4. Baris 16 adalah proses perhitungan nilai *fitness* rata-rata dalam satu populasi.
5. Baris 17 adalah proses untuk menghitung acuan varian.
6. Baris 18-23 adalah proses menghitung standart deviasi, yaitu akar kuadrat dari varian.
7. Baris 24 adalah proses menyalin nilai fitness terbaik dalam populasi.

8. Baris 26-27 adalah proses penulisan nilai-nilai yang akan ditampilkan di dalam log.

4.5.2.11.3 Implementasi Prosedur Pembuat Kesimpulan Laporan

Prosedur pembuat kesimpulan laporan ditulis dalam *archive* sebagai prosedur `void summary(void)` yang tidak membutuhkan argumen dan tidak memberikan nilai kembalian.

DESCRIPTION:

```
1    void summary (void)
2    {
3    fprintf(galist,"\n_____________________________________________
4    ________\n");
5    fprintf(galist,"\n\n\t\t [ REPORT SUMMARY ]\n");
6    fprintf(galist,"\n\n Best members: \n");
7    int d;
8    for(d=0; d<NVAR; d++)
9         {
10       fprintf(galist,"\n >> x[%d] =
11   %9.4f",d+1,population[POPSIZE].gen[d]);
12        }
13   fprintf(galist,"\n\n Best fitness =
14   %9.4f",population[POPSIZE].fitness);
15   fclose(galist);
16
17   free(population->gen);
18   free(population->upbound);
19   free(population->lowbound);
20   free(nextpop->gen);
21   free(nextpop->upbound);
22   free(nextpop->lowbound);
23   free(population);
24   free(nextpop);
25   }
26
```

Gambar 4.26 Deskripsi prosedur pembuat kesimpulan laporan

Berikut ini penjelasan deskripsi prosedur pembuat kesimpulan laporan:

1. Baris 1 adalah nama dan tipe data prosedur, juga argumen yang dibutuhkan. Prosedur `summary` tidak membutuhkan argumen.
2. Baris 3-6 adalah proses penulisan pembukaan dari kesimpulan laporan.
3. Baris 7-12 adalah proses penulisan semua variabel solusi dan nilainya.
4. Baris 13-14 adalah proses penulisan nilai fitness terbaik dari seluruh generasi.

5. Baris 17-24 adalah proses pembebasan memori dalam *heap* yang telah dipesan sebelumnya.

4.5.3 Implementasi Program Perantara

Library Genetic Algorithm tidak bisa langsung digunakan tanpa adanya sebuah program yang terhubung secara *static link* dengan `GASLib.a` dan menyertakan *header* `gaslib.h` pada bagian *preprocessor directive*. Implementasi program perantara akan tergantung kebutuhan *user* dan tujuan spesifik dari program yang akan dibuat. Pada bab berikutnya, akan diberikan satu contoh program perantara untuk pengujian *library* sesuai dengan kasus tertentu.

BAB V
PENGUJIAN

Rancangan yang telah diimplementasikan pada bab sebelumnya memerlukan suatu pengujian. Uji coba dilakukan untuk mengetahui hasil dari penelitian yang dilakukan, serta sebagai sarana pemunculan ide-ide untuk pengembangan selanjutnya.

5.1 Validasi *Library*

Pengujian ini bertujuan untuk memastikan apakah *library* GASLib berjalan sesuai dengan fungsi yang telah ditentukan dan benar perhitungannya. Pengujian yang akan dilakukan yaitu dengan membandingkan hasil perhitungan dengan referensi lain yang sudah diketahui hasilnya menggunakan data yang sama.

5.1.1 Kasus Uji 1

Validasi *library* dengan kasus uji 1 dilakukan dengan membandingkan *library* GASLib dengan hasil perhitungan yang diberikan oleh Fadlisyah, dkk[FAD-05] yang juga menggunakan algoritma genetika untuk menyelesaikan kasus ini.

Diberikan suatu kasus sebagai berikut:

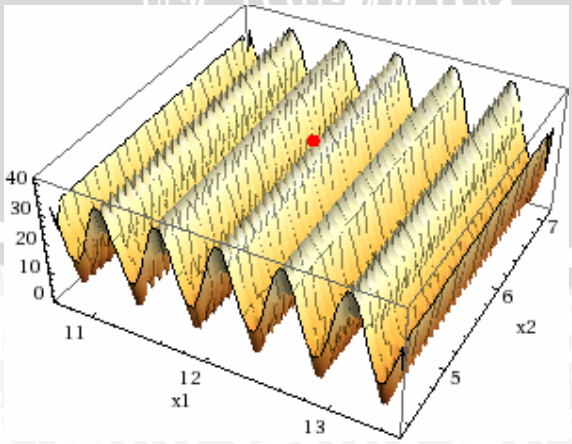
$$\text{Max } f(x_1,x_2) = 21{,}5 + x_1 \sin(4\pi x_1) + x_2 \sin(20\pi x_2)$$

dengan:

fungsi kendala: $-3{,}0 \le x_1 \le 12{,}1$

$4{,}1 \le x_2 \le 5{,}8$

probabilitas mutasi 1% dan probabilitas *crossover* 25%.



Gambar 5.1 Grafik fungsi kasus uji 1

Penyelesaian masalah tersebut menggunakan GASLib dipaparkan secara urut sebagai berikut:

Seluruh prosedur dalam GASLib didesain semaksimal mungkin untuk dapat bekerja secara independen sehingga memberikan kebebasan bagi *programmer* yang menggunakan *library* ini dalam menentukan susunan prosedur genetika yang akan ia gunakan di dalam program yang ditulisnya. Untuk kasus di atas, penulis akan menggunakan program yang ditulis dalam *file* `gaslibtest.c` yang mengikuti kaidah dari *simple genetic algorithm* dengan memanfaatkan prosedur yang sudah tersedia dalam *library*.

Pengujian kali ini menggunakan *Integrated Development Environment* (IDE) Bloodshed Dev-C++ 4.9.9.2 untuk membuat sebuah project yang berisi sebuah program yang dihubungkan secara statis dengan *archive* `GASLib.a`.

Langkah pertama yang perlu dilakukan *user* dari *library* adalah menuliskan semua preprocessor directive yang dibutuhkan program `gaslibtest`, termasuk *header inclusion* dari `gaslib.h` melalui `#include` dan makro yang bisa ditambahkan untuk membantu penulisan program.

```
#include <stdio.h>
#include <stdlib.h>
#include <math.h>
#include <time.h>
#include "gaslib.h"
#define PI 3.14
```

Langkah berikutnya adalah melakukan deklarasi variabel yang akan digunakan untuk menampung semua nilai parameter genetika. Dalam program ini, semua parameter itu akan ditampung lebih dahulu di dalam variabel global sebelum digunakan sebagai argumen dari prosedur `gaparam`. Semua deklarasi variabel penampung ini harus sesuai tipe data yang diperlukan prosedur `gaparam(int, int, int, double, double)`. Jika *user* membutuhkan variabel global lain, ataupun *prototype* prosedur pembantu, semuanya dituliskan pada bagian ini. Dalam kasus ini, penulis menggunakan dua *prototype* prosedur pembantu untuk proses *setting* nilai *fitness* dan parameter masukan.

```
int nvars, popsizes, maxgens;
double pcross, pmutation;
void setfitness(void);
void setparam(void);
```

Prosedur pembantu `setparam` berisi *user-interface* sederhana yang meminta masukan parameter dari pengguna program. Semua variabel penampung yang telah dideklarasikan sebelumnya digunakan dalam prosedur ini untuk menampung nilai masukan *user* melalui *console*.

```
void setparam(void)
{
printf("\nGENETIC ALGORITHM PARAMETERS\n\n");
printf("\nMasukkan Jumlah Variabel = ");
scanf("%d",&nvars);
printf("\nMasukkan Ukuran Populasi = ");
scanf("%d",&popsizes);
printf("\nMasukkan Jumlah Generasi Maksimum = ");
scanf("%d",&maxgens);
printf("\nMasukkan Probabilitas Crossover = ");
scanf("%lf",&pcross);
printf("\nMasukkan Probabilitas Mutasi = ");
scanf("%lf",&pmutation);
}
```

Prosedur pembantu `setfitness` berisi proses untuk memberikan nilai fitness setiap genotype sesuai dengan nilai variabel solusi yang disimpan dalam gen, dalam kasus ini nilai x_1 dan x_2. Di dalam prosedur ini, *user* harus menuliskan fungsi objektif sesuai dengan kasus yang dihadapi.

```
void setfitness(void)
{
int i, j;
double x[NVAR+1];

for (i=0; i<POPSIZE; i++)
    {
    for (j=0; j<NVAR; j++)
        x[j+1] = population[i].gen[j];
    population[i].fitness =
```

```
/*-------------------------------------------------------/
/                OBJECTIVE FUNCTION                     /
/-------------------------------------------------------*/

21.5+((x[1])*(sin(4*PI*(x[1]))))+((x[2])*(sin(20*PI*(x[2]))));

/*-------------------------------------------------------/
/            END OF OBJECTIVE FUNCTION                  /
/-------------------------------------------------------*/

   }
}
```

Fungsi utama dari `gaslibtest` terdiri dari beberapa bagian yang akan menggunakan prosedur-prosedur di dalam *library*. Bagian pertama adalah semua proses yang terjadi sebelum program memasuki *loop* genetika yang memanggil semua prosedur yang berkaitan dengan siklus seleksi alam dan reproduksi.

```
setparam();
gaparam(nvars, popsizes, maxgens, pcross, pmutation);
printf("\n\nProses Genetik Berlangsung\n\n");
initialisation();
setfitness();
bestmember();
reporthead();
```

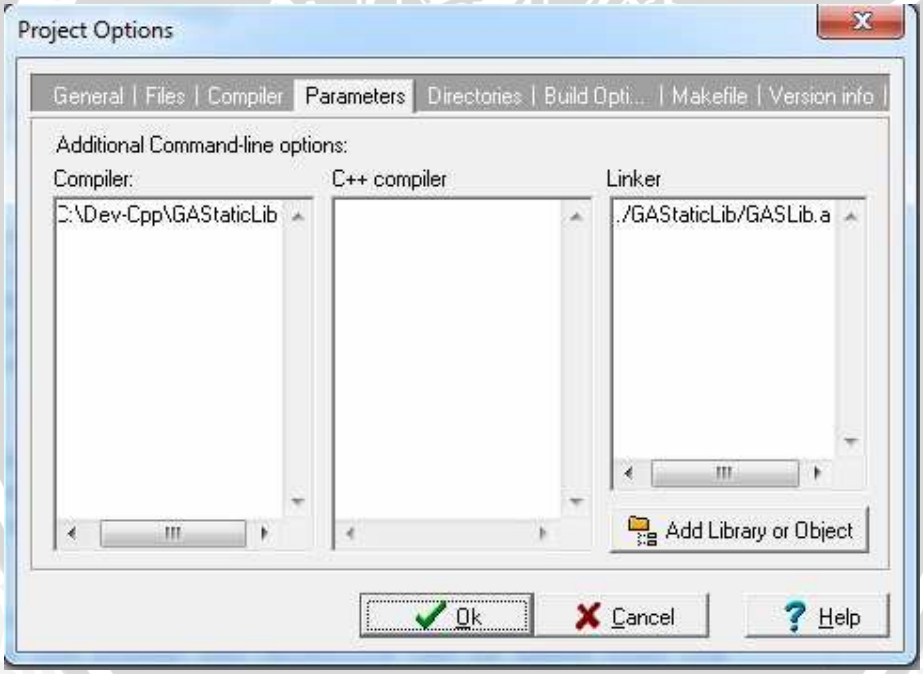
Bagian berikutnya dari fungsi utama `gaslibtest` merupakan proses yang terjadi saat *loop* seleksi alam dan reproduksi berlangsung. Dalam kasus ini, digunakan penghitung iterasi generasi mulai dari 0 hingga jumlah generasi maksimum sesuai parameter masukan *user*.

```
int generation=0;
while(generation<MAXGENS)
{
generation++;
selection();
crossover();
mutation();
report(generation);
setfitness();
elitist();
}
```

Bagian terakhir dari fungsi utama `gaslibtest` adalah proses pemanggilan prosedur penutup dari *library* untuk proses pembuatan kesimpulan, menutup *file* keluaran, dan membebaskan memori dalam *heap*. Akhir dari fungsi utama menampilkan informasi di *console* saat simulasi selesai dilakukan.

```
summary();
/*console display*/
printf("\n\n\nSimulation Finished! Log dumped to
'galist.txt'\n\n\n");
system("PAUSE");
return 0;
```

Sekarang *file* `gaslibtest.c` sudah siap untuk melalui proses kompilasi. Untuk proses *linking* saat pembuatan *object file*, Bloodshed Dev-C++ menyediakan fasilitas penulisan *additional command-line option* untuk menunjukkan *directory* dan nama *archive* yang dibutuhkan untuk proses *linking* pada bagian *project options*.



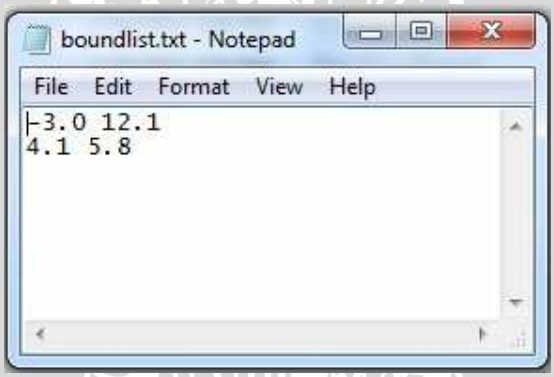
Gambar 5.2 *Project Options* pada Bloodshed Dev-C++

Jika kompilasi *file* `gaslibtest.c` menggunakan GCC *compiler* standart di sistem operasi berbasis UNIX, dapat digunakan penulisan *compiler option* menggunakan '`-lNAMA_ARCHIVE`' pada terminal.

```
$ gcc –Wall gaslibtest.c –lgaslib –o gaslibtest
```

Archive `GASLib.a` didesain dengan jalur masukan *domain* yang khusus dipisah untuk memudahkan user saat permasalahan yang ingin diselesaikan memiliki jumlah variabel solusi yang sangat banyak sesuai dengan fungsi kendala yang diminta oleh kasus tertentu. Dalam kasus ini, semua masukan parameter berjenis *domain* yang merupakan batas bawah dan batas atas dari x_1 dan x_2 akan dibaca oleh prosedur `void initialisation(void)` dari dalam *file* `boundlist.txt` yang harus diletakkan dalam *directory* yang sama (secara *default*) dengan lokasi program perantara `gaslibtest`.

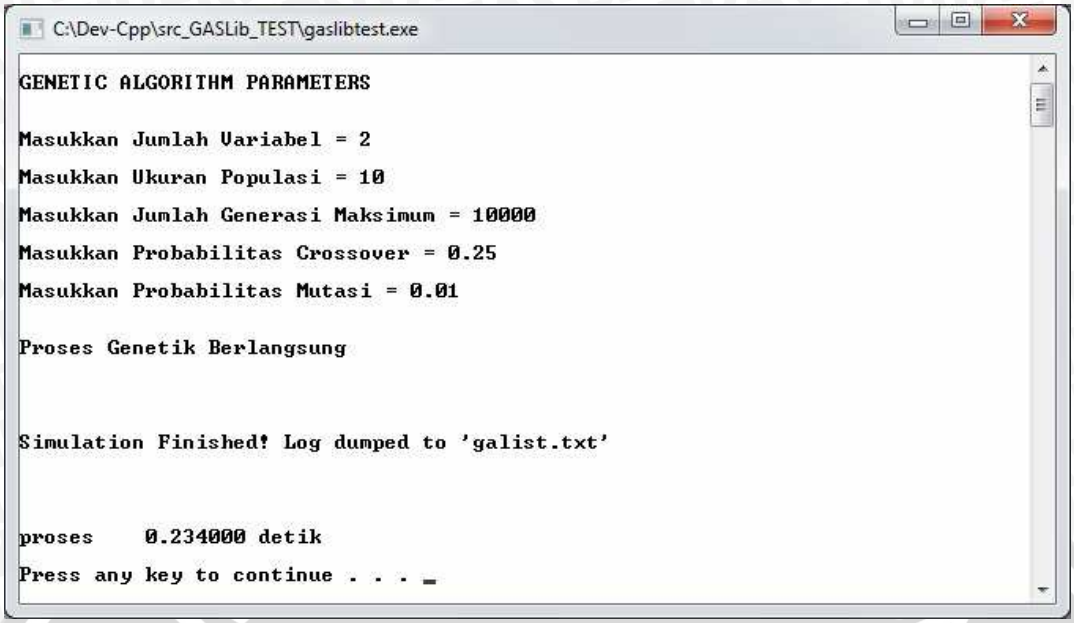
Pada kasus ini, isi dari *file* `boundlist.txt` akan ditulis secara manual sebelum program dieksekusi. Semua *domain* sesuai fungsi kendala ditulis secara berurutan dimulai dari batas bawah x_1 dan batas atasnya pada baris pertama dipisahkan spasi, kemudian batas bawah x_2 dan batas atasnya pada baris berikutnya.



Gambar 5.3 Isi dari *file* `boundlist.txt` untuk kasus uji 1

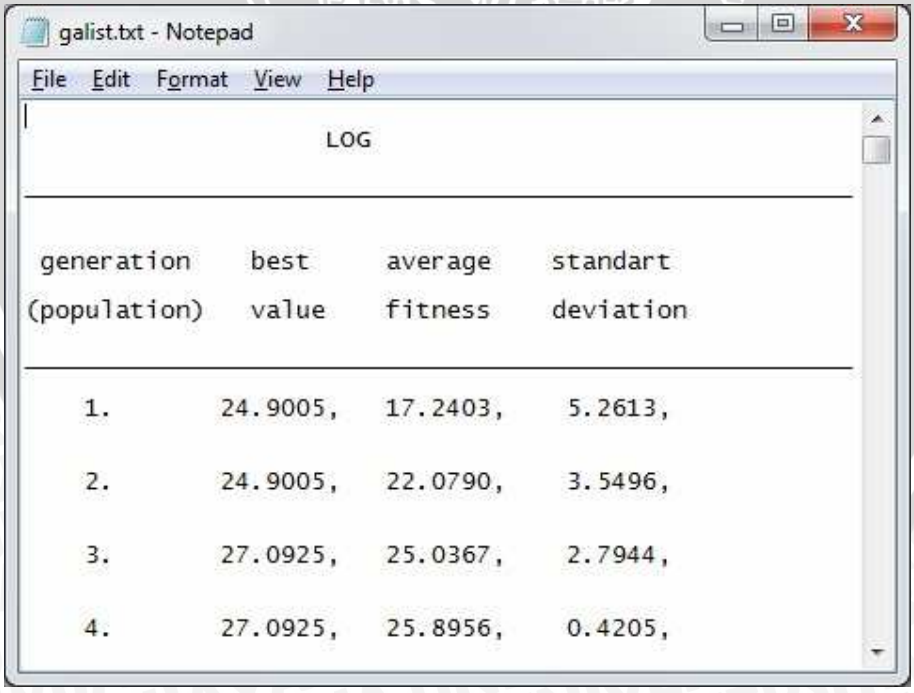
Jika proses *linking* dan kompilasi *file* `gaslibtest.c` berjalan dengan sukses, maka akan terbentuk sebuah *executable* `gaslibtest.exe` pada *directory* yang sama dengan *source file* `gaslibtest`. Sekarang program perantara yang menggunakan *library* GASLib ini siap untuk dijalankan oleh *user*.

Saat program dieksekusi, *console* akan menampilkan teks meminta masukan parameter dari *user*. Untuk pengujian kali ini, semua parameter akan dimasukkan sesuai dengan referensi.

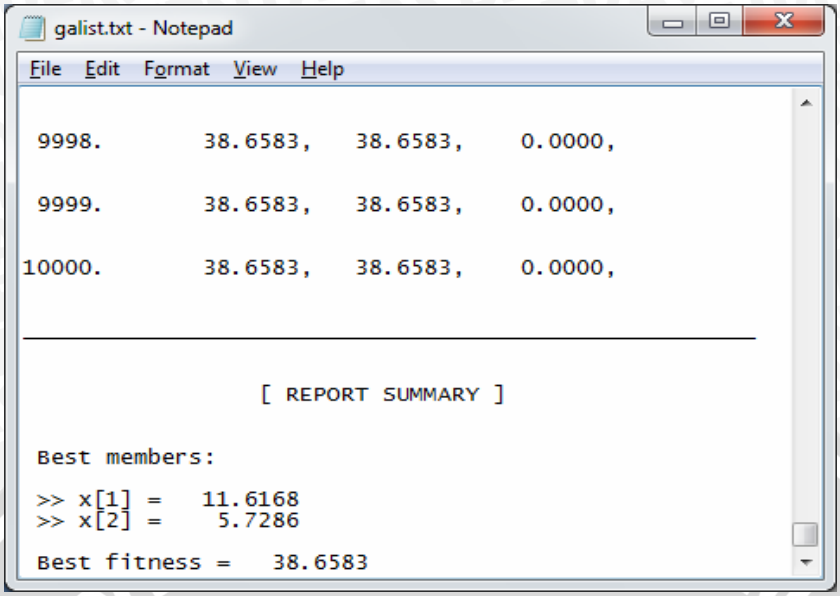


Gambar 5.4 *Console display* saat program dijalankan

Setelah simulasi berhasil dilakukan, data disimpan dalam bentuk *log* pada *file* `galist.txt` yang secara *default* akan ditulis oleh prosedur pembuat laporan pada *directory* yang sama dengan *executable* `gaslibtest.exe`.



Gambar 5.5 Isi bagian awal dari *file* `galist.txt` pada eksekusi pertama kasus uji 1



Gambar 5.6 Isi bagian akhir dari *file* `galist.txt` pada eksekusi pertama kasus uji 1

Pada validasi ini implementasi *library* GASLib dengan kasus tersebut dijalankan sebanyak 9 kali dengan nilai generasi maksimum berbeda, yaitu sebanyak 10.000, 50.000, dan 100.000 untuk mengetahui nilai parameter generasi yang dibutuhkan agar library memberikan hasil yang konsisten, kemudian diambil nilai *fitness* dan variabel solusinya untuk dibandingkan dengan referensi. Tabel 5.1 menunjukkan nilai *fitness* dan nilai variabel solusi dari *genotype* terbaik.

Tabel 5.1 Hasil implementasi GASLib pada kasus uji 1

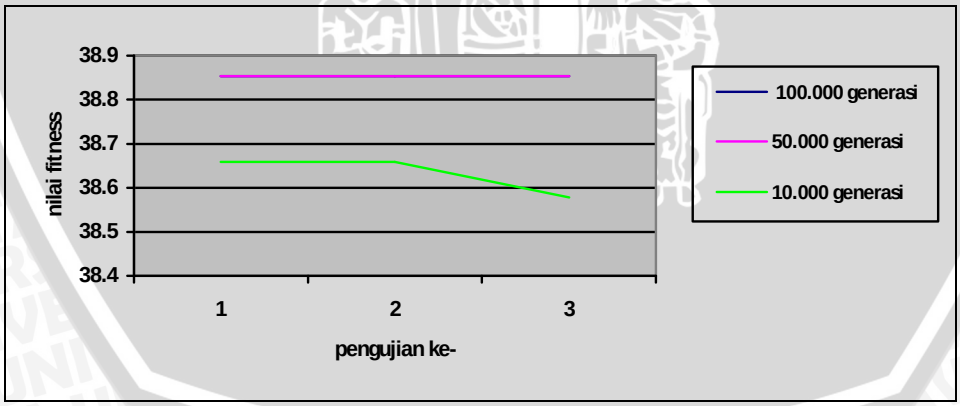
Jumlah Generasi Maksimum	Pengujian ke-	Nilai Variabel Solusi		Nilai Fitness Terbaik	Tercapai Pada Generasi ke-
		x_1	x_2		
10.000	1	11,6168	5,7286	38,6583	7530
	2	11,6168	5,7286	38,6583	2230
	3	11,6168	5,7252	38,5781	7293
50.000	1	11,6319	5,7286	38,8541	15093
	2	11,6319	5,7286	38,8541	10243
	3	11,6319	5,7286	38,8541	7777
100.000	1	11,6319	5,7286	38,8541	43148
	2	11,6319	5,7286	38,8541	2437
	3	11,6319	5,7286	38,8541	12743

Sebagai perbandingan, nilai terbaik yang diperoleh dalam percobaan yang dilakukan Fadlisyah, dkk [FAD-05]

x_1 = 11,631407

x_2 = 5,724824

fitness terbaik = 38,818208



Gambar 5.7 Grafik perubahan nilai *fitness* terbaik pada pengujian dan perbandingannya

Kesimpulan dari validasi *library* dengan kasus uji 1 menunjukkan bahwa *library* GASLib berjalan dengan baik sesuai dengan fungsi yang telah ditentukan, dengan nilai *fitness* terbaik dalam kasus ini sebesar 38,8541 dengan nilai variabel solusi terbaik x_1 =

11,6319 dan x_2 = 5,7286. Hasil yang didapatkan akan konsisten dimulai dari jumlah generasi maksimum sebanyak 50.000 generasi sebagai nilai parameter kasus uji pertama.

5.1.2 Kasus Uji 2

Validasi *library* dengan kasus uji 2 bertujuan untuk membuktikan bahwa *library* GASLib memberikan hasil yang benar dibandingkan dengan perhitungan menggunakan metode konvensional.

Diberikan suatu kasus sebagai berikut:

$$\text{Max } f(x_1,x_2) = 2x, - x_2$$

dengan:

fungsi kendala: $0 \leq x_1 \leq 5$

$2 \leq x_2 \leq 7$

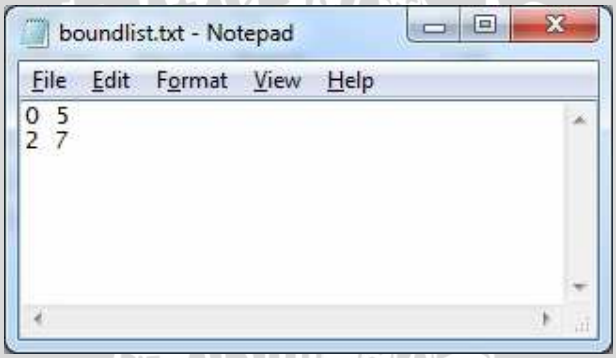
Kasus uji di atas dapat dengan mudah diselesaikan secara matematis, yang menghasilkan solusi maksimisasi untuk nilai *x*, yang sebesar mungkin dan nilai x_2 yang sekecil mungkin. Dengan batasan fungsi kendala pada kasus ini, maka solusi akan didapatkan pada x_1 = 5 dan x_2 = 2.

Penyelesaian masalah tersebut menggunakan *library* GASLib dipaparkan secara berurutan sebagai berikut:

Secara garis besar, langkah-langkah penyelesaian kasus ini sama dengan cara menggunakan *library* GASLib pada penyelesaian kasus sebelumnya. Perbedaan terletak pada fungsi objektif dan domain variabel solusi. Berikut ini merupakan fungsi objektif yang dituliskan untuk menyelesaikan kasus uji 2.

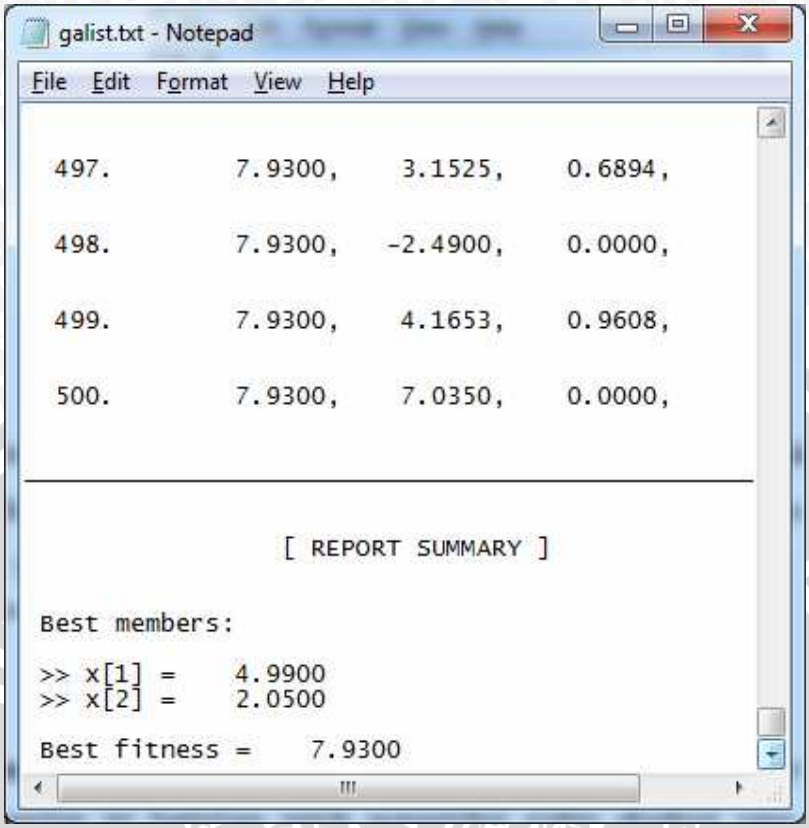
```
/*-------------------------------------------------------/
/                  OBJECTIVE FUNCTION                   /
/-------------------------------------------------------*/

                    (2*x[1] - x[2]);

/*-------------------------------------------------------/
/              END OF OBJECTIVE FUNCTION                /
/-------------------------------------------------------*/
```

Pada kasus ini, isi dari *file* `boundlist.txt` akan ditulis secara manual sebelum program dieksekusi. Semua *domain* sesuai fungsi kendala ditulis secara berurutan dimulai dari batas bawah x_1 dan batas atasnya pada baris pertama yang dipisahkan spasi, kemudian batas bawah x_2 dan batas atasnya pada baris berikutnya.



Gambar 5.8 Isi dari *file* `boundlist.txt` untuk kasus uji 2

Untuk kasus uji 2, penulis menentukan sendiri parameter genetik yang digunakan dengan jumlah generasi sebanyak 500. Setelah kompilasi berhasil dan program simulasi berhasil dilakukan, data disimpan dalam bentuk *log* pada *file* `galist.txt` yang secara *default* akan ditulis oleh prosedur pembuat laporan pada *directory* yang sama dengan *executable* `gaslibtest.exe`.



Gambar 5.9 Isi dari *file* `galist.txt` untuk kasus uji 2

Nilai yang diperoleh dalam pengujian untuk kasus uji 2

x_1 = 4,9900

x_2 = 2,0500

fitness terbaik = 7,9300

Kesimpulan dari validasi *library* dengan kasus uji 2 menunjukkan bahwa *library* GASLib berjalan dengan baik dan memberikan hasil yang benar.

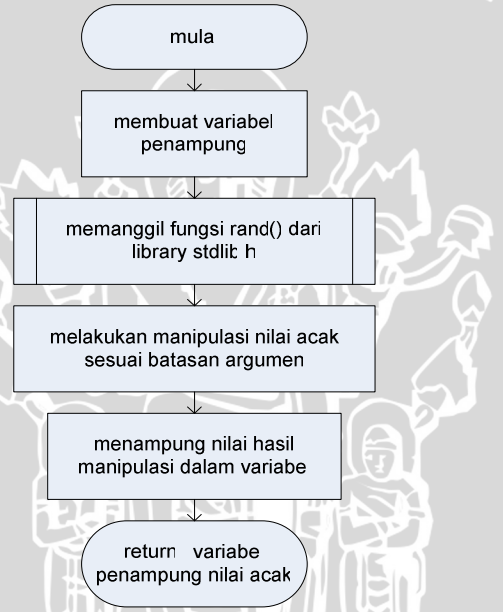
5.2 Analisis Kompleksitas Algoritma

Pengujian ini bertujuan untuk mengetahui kompleksitas algoritma yang digunakan pada prosedur-prosedur inti yang menangani operasi genetik dalam *library* GASLib. Kompleksitas algoritma yang diuji hanya kompleksitas waktu dikarenakan kompleksitas ruang sudah tidak menjadi isu utama dalam penerapan algoritma dengan perkembangan teknologi sekarang. Kompleksitas waktu tersebut akan dinyatakan dalam notasi O-besar (*Big-O*). Algoritma yang akan diuji antara lain:

1. Algoritma pada prosedur pembangkit nilai acak.
2. Algoritma pada prosedur pemberian nilai awal.
3. Algoritma pada prosedur *Elitist.*
4. Algoritma pada prosedur seleksi.
5. Algoritma pada prosedur *crossover*.
6. Algoritma pada prosedur mutasi.

5.2.1 Algoritma pada Prosedur Pembangkit Nilai Acak

Prosedur ini bertugas memberikan satu nilai acak sesuai dengan argumen yang diberikan sebagai parameter saat prosedur ini dipanggil. Berikut ini *flowchart* secara detail dari hasil implementasi prosedur pembangkit nilai acak:



Gambar 5.10 *Flowchart* prosedur pembangkit nilai acak

Berikut merupakan analisis kompleksitas algoritma pada prosedur ini:

1. Pada bagian membuat variabel lokal untuk penampung nilai acak didapatkan kompleksitas algoritma O(1) karena bagian ini hanya dilakukan sekali dalam satu siklus kerja pembangkitan nilai acak.
2. Pada bagian pemanggilan fungsi `rand()` dari *library* standart `stdlib.h` didapatkan kompleksitas algoritma O(1) karena bagian ini hanya dilakukan sekali dalam satu siklus kerja pembangkitan nilai acak.

3. Pada bagian manipulasi nilai acak dengan operator aritmatika didapatkan kompleksitas algoritma O(1) karena bagian ini hanya dilakukan sekali dalam satu siklus kerja pembangkitan nilai acak.
4. Pada bagian menampung nilai acak hasil dari manipulasi aritmatika didapatkan kompleksitas algoritma O(1) karena bagian ini hanya dilakukan sekali dalam satu siklus kerja pembangkitan nilai acak.

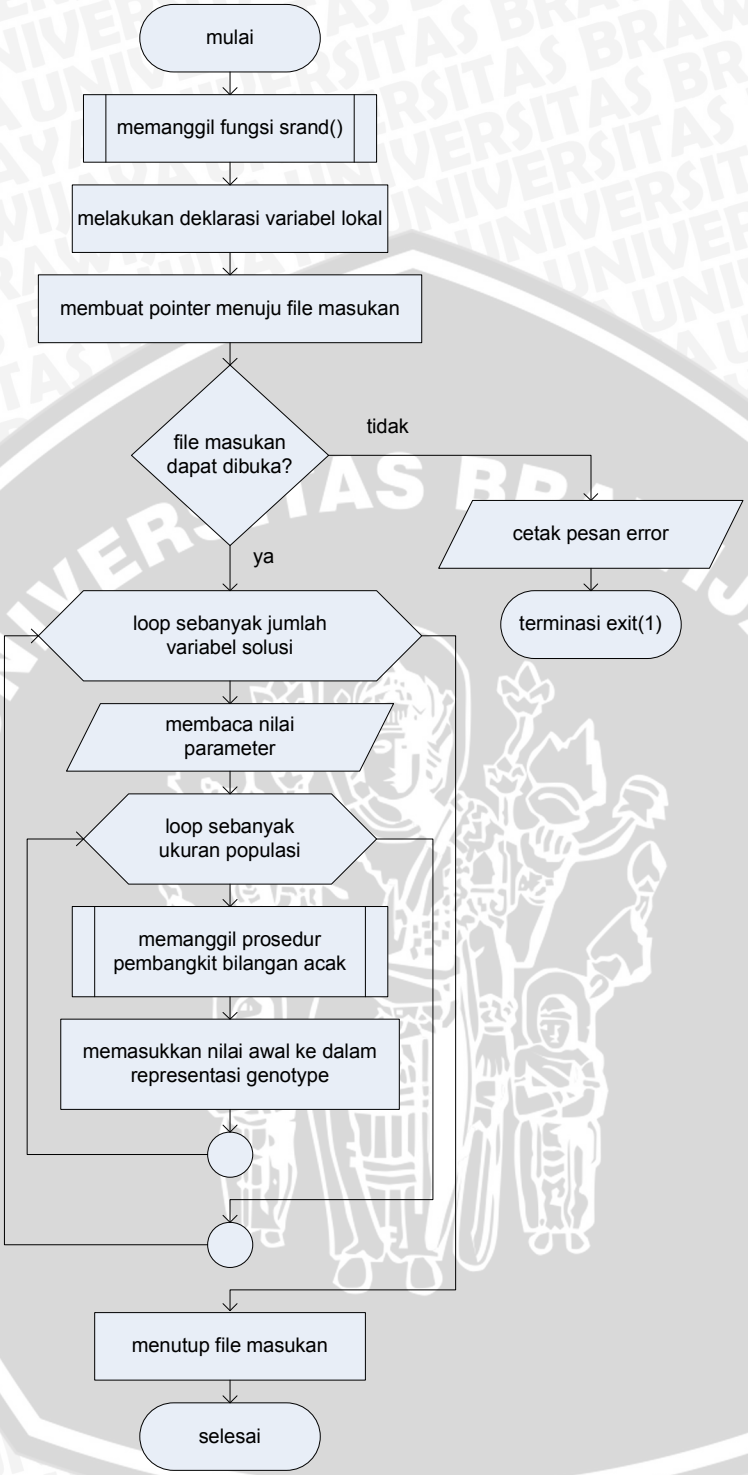
Setelah diketahui kompleksitas algoritma untuk setiap bagian, bisa diketahui kompleksitas algoritma pada prosedur ini dengan cara:

O(1) + O(1) + O(1) + O(1) = O(1)

Setelah dilakukan perhitungan diketahui bahwa kompleksitas algoritma untuk prosedur pembangkit nilai acak adalah O(1).

5.2.2 Algoritma pada Prosedur Pemberian Nilai Awal

Prosedur ini bertugas memberikan nilai awal berupa set solusi potensial pada bagian awal algoritma. Berikut ini *flowchart* secara detail dari hasil implementasi prosedur pemberian nilai awal:



Gambar 5.11 *Flowchart* prosedur pemberian nilai awal

Berikut merupakan analisis kompleksitas algoritma pada prosedur ini:

1. Pada bagian memanggil fungsi `srand()` dari *library* standart `stdlib.h` didapatkan kompleksitas algoritma O(1) karena bagian ini hanya dilakukan sekali dalam satu siklus kerja pemberian nilai awal.
2. Pada bagian deklarasi variabel lokal didapatkan kompleksitas algoritma O(1) karena bagian ini hanya dilakukan sekali dalam satu siklus kerja pemberian nilai awal.
3. Pada proses membuka file masukan `boundlist.txt` didapatkan kompleksitas algoritma O(1) karena bagian ini hanya dilakukan sekali dalam satu siklus kerja pemberian nilai awal.
4. Pada bagian membaca nilai parameter didapatkan kompleksitas algoritma O(n) karena bagian ini dilakukan sebanyak n kali jumlah variabel solusi yang dibaca parameter domainnya.
5. Pada bagian memanggil prosedur pembangkit bilangan acak didapatkan kompleksitas algoritma $O(n^2)$ karena bagian ini dilakukan sebanyak n kali jumlah variabel solusi dan n kali ukuran populasi.
6. Pada bagian memasukkan nilai awal ke dalam representasi *genotype* didapatkan kompleksitas algoritma $O(n^2)$ karena bagian ini dilakukan sebanyak n kali jumlah variabel solusi dan n kali ukuran populasi.
7. Pada bagian menutup *file* masukan didapatkan kompleksitas algoritma O(1) karena bagian ini hanya dilakukan sekali dalam satu siklus kerja pemberian nilai awal.

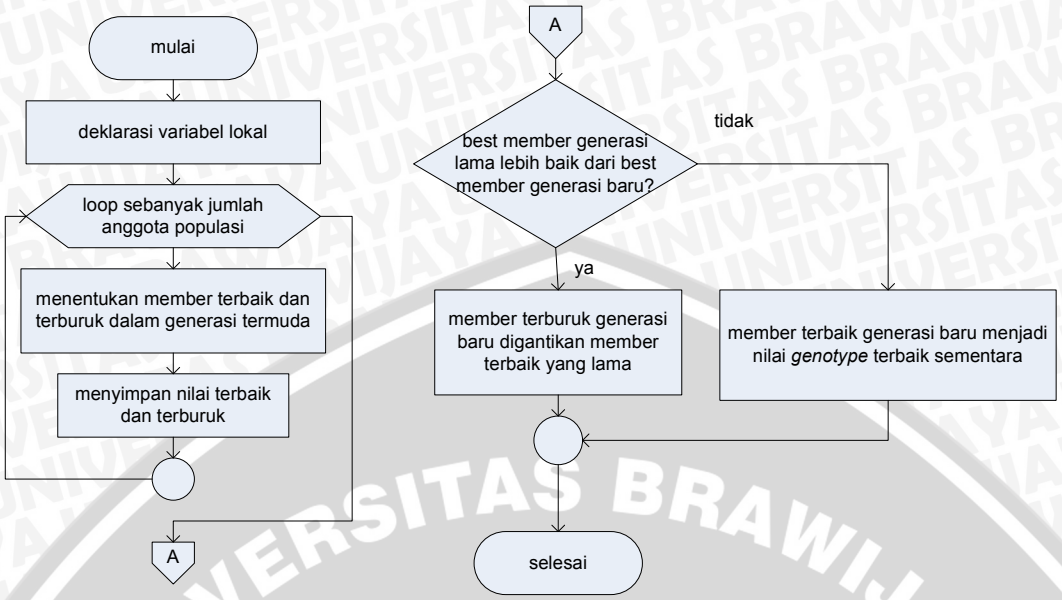
Setelah diketahui kompleksitas algoritma untuk setiap bagian, bisa diketahui kompleksitas algoritma pada prosedur ini dengan cara:

$O(1) + O(1) + O(1) + O(n) + O(n^2) + O(n^2) + O(1) = O(n^2)$

Setelah dilakukan perhitungan diketahui bahwa kompleksitas algoritma untuk prosedur pemberian nilai awal adalah $O(n^2)$.

5.2.3 Algoritma pada Prosedur *Elitist*

Prosedur ini merupakan implementasi proses *elitisme* dalam algoritma genetika. Berikut ini *flowchart* secara detail dari hasil implementasi prosedur *elitist*:



Gambar 5.12 *Flowchart* prosedur *elitist*

Berikut merupakan analisis kompleksitas algoritma pada prosedur ini:

1. Pada bagian deklarasi variabel lokal didapatkan kompleksitas algoritma O(1) karena bagian ini hanya dilakukan sekali dalam satu siklus kerja *elitist.*
2. Pada bagian menentukan member terbaik dan terburuk dalam generasi termuda didapatkan kompleksitas algoritma O(n) karena bagian ini dilakukan sebanyak n kali jumlah anggota populasi generasi termuda.
3. Pada bagian menyimpan nilai terbaik dan terburuk didapatkan kompleksitas algoritma O(n) karena bagian ini dilakukan sebanyak n kali jumlah anggota populasi generasi termuda.
4. Pada bagian penukaran posisi member didapatkan kompleksitas algoritma O(1) karena bagian ini hanya dilakukan sekali dalam satu siklus kerja *elitist.*

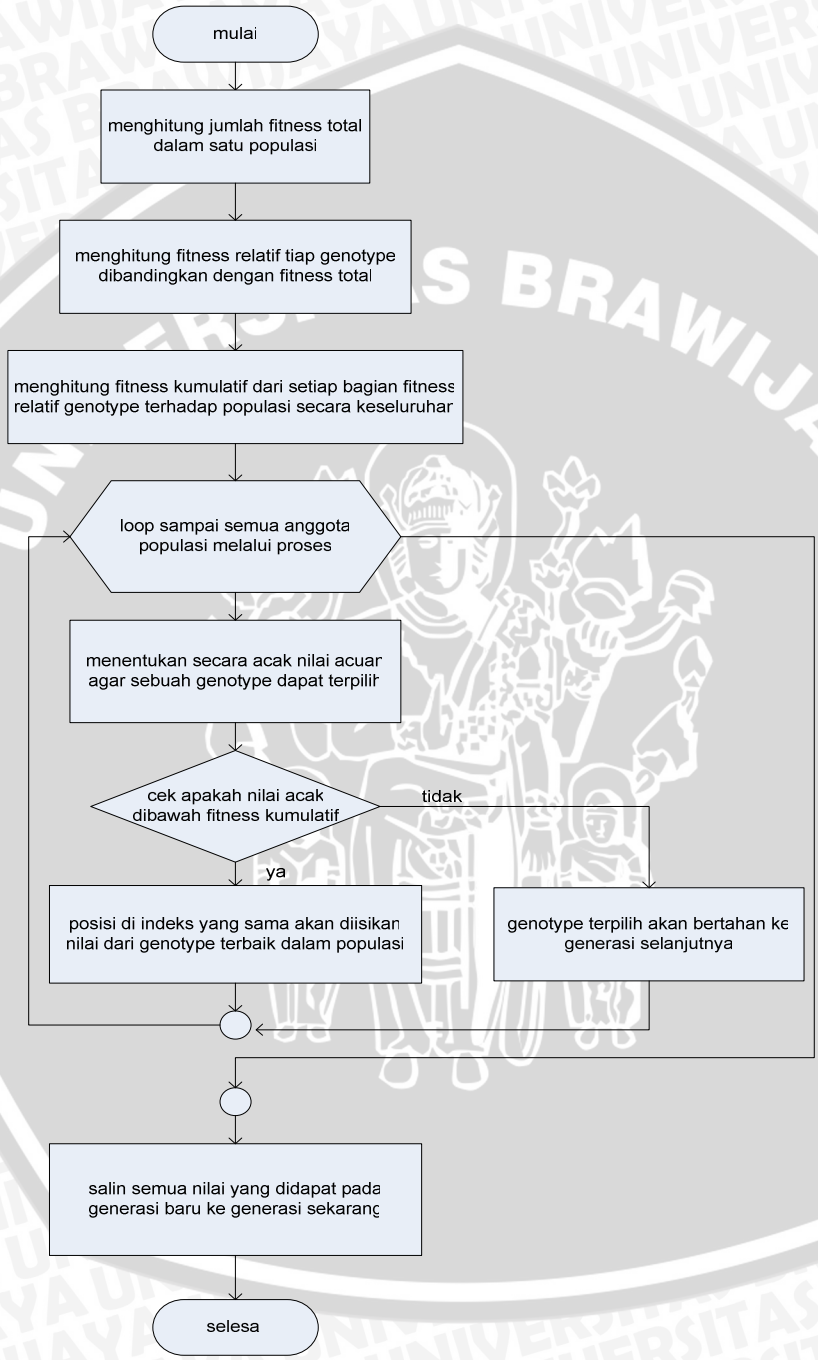
Setelah diketahui kompleksitas algoritma untuk setiap bagian, bisa diketahui kompleksitas algoritma pada prosedur ini dengan cara:

$O(1) + O(n) + O(n) + O(1) = O(n)$

Setelah dilakukan perhitungan diketahui bahwa kompleksitas algoritma untuk prosedur *elitist* adalah O(n).

5.2.4 Algoritma pada Prosedur Seleksi

Prosedur ini merupakan bagian dari algoritma genetika yang menangani proses seleksi alam. Berikut ini *flowchart* secara detail dari hasil implementasi prosedur seleksi:



Gambar 5.13 *Flowchart* prosedur seleksi

Berikut merupakan analisis kompleksitas algoritma pada prosedur ini:

1. Pada bagian menghitung *fitness* total dalam satu populasi didapatkan kompleksitas algoritma O(n) karena bagian ini dilakukan sebanyak n kali jumlah anggota populasi.
2. Pada bagian menghitung perbandingan *fitness* relatif didapatkan kompleksitas algoritma O(n) karena bagian ini dilakukan sebanyak n kali jumlah anggota populasi.
3. Pada bagian menghitung *fitness* kumulatif dari setiap bagian *fitness* relatif terhadap populasi secara keseluruhan didapatkan kompleksitas algoritma O(n) karena bagian ini dilakukan sebanyak n kali jumlah anggota populasi.
4. Pada bagian menentukan nilai acuan untuk proses seleksi didapatkan kompleksitas algoritma O(n) karena bagian ini dilakukan sebanyak n kali jumlah anggota populasi.
5. Pada bagian pemilihan *genotype* didapatkan kompleksitas algoritma $O(n^2)$ karena bagian ini dilakukan sebanyak n kali jumlah anggota populasi dengan 2 indeks bersarang.
6. Pada bagian menyalin nilai ke generasi terbaru didapatkan kompleksitas algoritma O(n) karena bagian ini dilakukan sebanyak n kali jumlah anggota populasi.

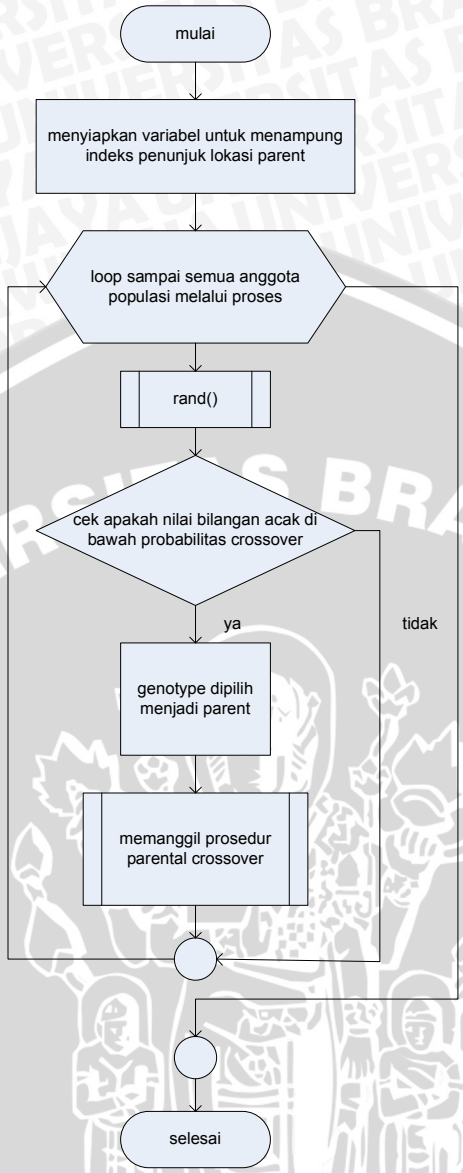
Setelah diketahui kompleksitas algoritma untuk setiap bagian, bisa diketahui kompleksitas algoritma pada prosedur ini dengan cara:

$O(n) + O(n) + O(n) + O(n) + O(n^2) + O(n) = O(n^2)$

Setelah dilakukan perhitungan diketahui bahwa kompleksitas algoritma untuk prosedur seleksi adalah $O(n^2)$.

5.2.5 Algoritma pada Prosedur *Crossover*

Prosedur ini merupakan bagian dari algoritma genetika yang menangani proses persilangan. Berikut ini *flowchart* secara detail dari hasil implementasi prosedur *crossover*:



Gambar 5.14 *Flowchart* prosedur *crossover*

Berikut merupakan analisis kompleksitas algoritma pada prosedur ini:

1. Pada bagian deklarasi variabel lokal didapatkan kompleksitas algoritma O(1) karena bagian ini hanya dilakukan sekali dalam satu siklus kerja *crossover*.
2. Pada bagian pemanggilan fungsi `rand()` dari *library* standart `stdlib.h` didapatkan kompleksitas algoritma O(n) karena bagian ini dilakukan sebanyak n kali ukuran populasi.
3. Pada bagian seleksi kondisi berdasarkan nilai acak didapatkan kompleksitas algoritma O(n) karena bagian ini dilakukan sebanyak n kali ukuran populasi.

4. Pada bagian pemilihan *parent* didapatkan kompleksitas algoritma O(n) karena bagian ini dilakukan sebanyak n kali ukuran populasi.
5. Pada bagian pemanggilan prosedur persilangan *parent* yang terpilih didapatkan kompleksitas algoritma O(n) karena bagian ini dilakukan sebanyak n kali ukuran populasi.

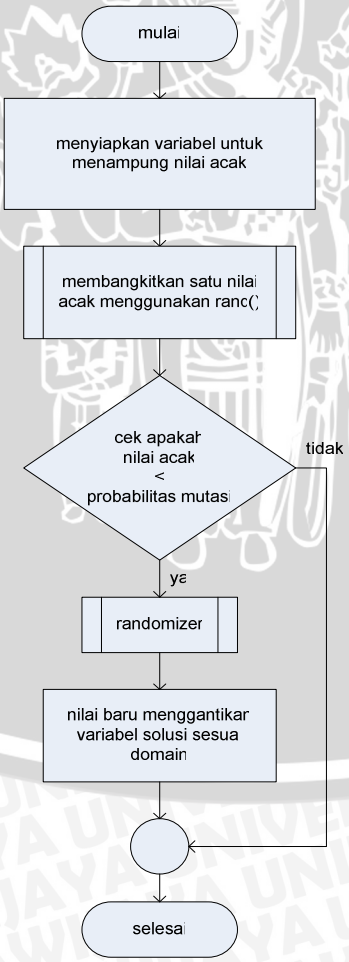
Setelah diketahui kompleksitas algoritma untuk setiap bagian, bisa diketahui kompleksitas algoritma pada prosedur ini dengan cara:

O(1) + O(n) + O(n) + O(n) + O(n) = O(n)

Setelah dilakukan perhitungan diketahui bahwa kompleksitas algoritma untuk prosedur *crossover* adalah O(n).

5.2.6 Algoritma pada Prosedur Mutasi

Prosedur ini merupakan bagian dari algoritma genetika yang menangani proses mutasi. Berikut ini *flowchart* secara detail dari hasil implementasi prosedur mutasi:



Gambar 5.15 *Flowchart* prosedur mutasi

Berikut merupakan analisis kompleksitas algoritma pada prosedur ini:

1. Pada bagian deklarasi variabel lokal didapatkan kompleksitas algoritma O(1) karena bagian ini hanya dilakukan sekali dalam satu siklus kerja mutasi.
2. Pada bagian pemanggilan fungsi `rand()` dari *library* standart `stdlib.h` didapatkan kompleksitas algoritma $O(n^2)$ karena bagian ini dilakukan sebanyak n kali ukuran populasi dan n kali jumlah variabel solusi.
3. Pada bagian seleksi kondisi berdasarkan probabilitas mutasi didapatkan kompleksitas algoritma $O(n^2)$ karena bagian ini dilakukan sebanyak n kali ukuran populasi dan n kali jumlah variabel solusi.
4. Pada bagian memanggil prosedur pembangkit nilai acak didapatkan kompleksitas algoritma $O(n^2)$ karena bagian ini dilakukan sebanyak n kali ukuran populasi dan n kali jumlah variabel solusi.
5. Pada bagian mengganti nilai gen yang baru didapatkan kompleksitas algoritma $O(n^2)$ karena bagian ini dilakukan sebanyak n kali ukuran populasi dan n kali jumlah variabel solusi.

Setelah diketahui kompleksitas algoritma untuk setiap bagian, bisa diketahui kompleksitas algoritma pada prosedur ini dengan cara:

$O(1) + O(n^2) + O(n^2) + O(n^2) + O(n^2) = O(n^2)$

Setelah dilakukan perhitungan diketahui bahwa kompleksitas algoritma untuk prosedur mutasi adalah $O(n^2)$.

5.3 Pengujian *Execution Time*

Pengujian ini bertujuan untuk mengetahui waktu eksekusi yang dibutuhkan saat bagian program memasuki proses genetik menggunakan prosedur yang dimiliki *library* GASLib. Pengujian ini menggunakan media program perantara untuk kasus yang sama dengan bagian validasi *library* yaitu kasus uji 1, dengan beberapa baris tambahan untuk proses menghitung waktu eksekusi. Tipe data `clock_t` dan fungsi `clock()` didefinisikan dalam *library* standart `time.h`.

```
clock_t startm, stopm;
#define STARTTIME if((startm = clock())==-
1){printf("\nError calling clock");exit(1);}
#define STOPTIME if((stopm = clock())==-
1){printf("\nError calling clock");exit(1);}
```

```
#define PRINTTIME printf("\n\nproses %6.3f
detik\n\n",((double)stopm-startm)/CLOCKS_PER_SEC)
```

Pada pengujian ini, waktu akan mulai dihitung saat proses memasuki bagian inisialisasi, yaitu bagian awal dari algoritma genetika setelah semua parameter diberikan oleh pengguna. Berdasarkan cara kerja dari *library genetic algorithm* yang dirancang, dapat disimpulkan bahwa faktor yang paling mempengaruhi waktu eksekusi dari program yang menggunakan *library* ini untuk satu kasus yang sama adalah parameter ukuran populasi, jumlah generasi, probabilitas *crossover*, dan probabilitas mutasi.

Parameter yang akan ditentukan untuk pengujian ini ditunjukkan pada tabel 5.2 berikut ini.

Tabel 5.2 Nilai parameter yang digunakan untuk pengujian *execution time*

Pengujian ke-	Parameter			
	Ukuran Populasi	Jumlah Generasi	Prob. Crossover	Prob. Mutasi
1	10	10000	25%	1%
2	20	10000	25%	1%
3	50	10000	25%	1%
4	10	50000	25%	1%
5	10	100000	25%	1%
6	10	10000	50%	1%
7	10	10000	75%	1%
8	10	10000	25%	10%
9	10	10000	25%	50%

Pengujian dilakukan dengan mengubah-ubah nilai parameter-parameter tersebut. Nilai-nilai dari parameter yang sedang tidak diuji akan menggunakan nilai pada pengujian pertama sehingga dapat terlihat perbedaan yang diakibatkan dari perubahan tiap-tiap nilai parameter. Pengujian ini menggunakan perangkat keras komputer dan perangkat lunak dengan spesifikasi yang tunjukkan pada tabel 5.3.

Tabel 5.3 Spesifikasi perangkat keras komputer dan perangkat lunak pada pengujian *execution time*

Prosesor	Intel Core 2 Duo T6500 2,10 GHz
Memori (RAM)	2,00 GB
Sistem Operasi	Windows Vista Home Premium Service Pack 2

Hasil dari pengujian pada masing-masing perubahan nilai parameter ditunjukkan pada tabel 5.4.

Tabel 5.4 Hasil pengujian *execution time*

Pengujian ke-	Waktu eksekusi (dalam milidetik)
1	113
2	210
3	543
4	693
5	1183
6	114
7	161
8	152
9	149

Dari pengujian waktu eksekusi didapatkan kesimpulan bahwa semakin besar ukuran populasi dan jumlah generasi, maka waktu eksekusi akan semakin bertambah lama secara signifikan. Sedangkan untuk nilai probabilitas *crossover* dan mutasi yang semakin besar akan memberikan kemungkinan penambahan waktu eksekusi karena peluang untuk proses *crossover* dan mutasi dijalankan semakin besar pula. Tetapi karena kedua proses ini bisa dijalankan atau tidak tergantung angka yang acak, maka akan ada kemungkinan waktu eksekusi lebih singkat seperti pada pengujian dengan perubahan nilai parameter probabilitas mutasi pada pengujian ke-9 berdasarkan tabel 5.2.

BAB VI
PENUTUP

Setelah melakukan perancangan, implementasi, dan pengujian, maka tugas akhir ini mencapai tahap pengambilan kesimpulan disertai pula dengan saran bagi para peminat yang ingin menyempurnakan perangkat lunak ini

6.1 Kesimpulan

Kesimpulan yang dapat ditarik dari rangkaian tugas akhir ini adalah sebagai berikut:

1. *Library* yang dirancang terdiri dari kumpulan prosedur atau fungsi yang mendukung proses pencarian solusi secara *stochastic* berdasarkan seleksi alam dan proses evolusi, juga beberapa fungsi tambahan dan *file pointer* untuk proses *input-output*, dan fungsi pembantu lain untuk *value* maupun *variable-handling*, dengan konfigurasi yang diberikan oleh *user library* melalui program perantara.
2. *Library genetic algorithm* diimplementasikan dengan pendekatan *procedural programming* sebagai sebuah *archive* yang berisi *header file* dan C *source library* yang dibangun menggunakan GCC *compiler*.
3. Pengujian *library genetic algorithm* dilakukan melalui proses validasi dengan membandingkan hasil perhitungan aplikasi yang menggunakan GASLib dengan hasil yang diberikan oleh referensi [FAD-05] dan dengan metode konvensional. Dari kasus uji 1 diperoleh nilai *fitness* terbaik yang konsisten untuk jumlah generasi sebanyak 50.000 generasi. Pada kasus uji 2 didapatkan perbedaan nilai variabel solusi sebesar 0,2% untuk x_1 dan 2,5% untuk x_2, dibandingkan dengan metode konvensional.
4. Performa waktu eksekusi dari algoritma, yang ditunjukkan melalui pengujian program yang menggunakan *library* GASLib, dipengaruhi secara signifikan oleh parameter berupa jumlah populasi dan jumlah generasi. Semakin tinggi nilai parameternya, maka proses eksekusi akan berlangsung lebih lama.

6.2 Saran

Terdapat beberapa hal yang perlu dibenahi dalam *library* yang dibuat oleh penulis. Untuk mempermudah dalam mempelajari dan berguna sebagai bahan pertimbangan bagi mereka yang berminat pada perangkat lunak ini, penulis memberikan saran-saran umum sebagai berikut:

1. *Library* ini dapat dikembangkan fiturnya agar mendukung representasi *genotype* secara *binary*.
2. *Library* dapat dilengkapi tambahan prosedur untuk mendukung tipe data selain *double*.
3. *Library* dapat dikembangkan untuk model evolusi yang mendukung persebaran populasi majemuk.

DAFTAR PUSTAKA

Anonymous. 2010. *Algoritma Genetika.* http://lecturer.eepisits.edu/~kangedi/materikuliah/Kecerdasan Buatan/Bab 7 Algoritma Genetika.pdf, diakses tanggal 22 Juli 2010.

Aribowo, A., Lukas, S. & Gunawan, Martin. 2008. Penerapan Algoritma Genetika pada Penentuan Komposisi Pakan Ayam Petelur. *Seminar Nasional Aplikasi Teknologi Informasi 2008*, (Online), (http://journal.uii.ac.id/index.php/Snati/article/view/384/299, diakses 3 Agustus 2010).

Basuki, Achmad. 2003**.** *Algoritma Genetika Suatu Alternatif Penyelesaian Permasalahan Searching, Optimasi dan Machine Learning***.** Surabaya **:** PENS-ITS.

Burgess, Mark. 1999. *C Programming Tutorial, 4th Edition.* Boston : Free Software Foundation.

Chambers, Lance. 2001. *The Practical Handbook of Genetic Algorithms Applications, Second Edition.* Boca Raton: Chapman & Hall / CRC.

Fadlisyah, Arnawan, Faisal. 2009. *Algoritma Genetik*. Yogyakarta : Graha Ilmu.

Gough, Brian. 2004. *An Introduction to GCC*. Bristol: Network Theory Ltd.

Hagen, William von. 2006. *The Definitive Guide to GCC, Second Edition.* New York : Springer-Verlag New York, Inc.

Kadir, Abdul. 1994. *Pemrograman Dasar Turbo C untuk IBM PC*. Yogyakarta : Andi Offset.

Mitchell, Melanie. 1996. *An Inroduction to Genetic Algorithms.* London : MIT Press.

Munir, Rinaldi. 2005. *Matematika Diskrit Edisi 3*. Bandung : Informatika.

Obitko, Marek. 1998. *Introduction to Genetic Algorithms.* http://www.obitko.com/tutorials/genetic-algorithms/, diakses tanggal 22 Juli 2010.

Sinaga, Edison. 2009. *Implementasi Algoritma Genetika Dalam Penyusunan Teka Teki Silang*. Skripsi. Program Studi Ilmu Komputer, Fakultas Matematika dan Ilmu Pengetahuan Alam, Universitas Sumatera Utara, Medan.

Sivanandam, S.N dan Deepa, S.N. 2008. *Introduction to Genetic Algorithms.* New York : Springer-Verlag Berlin Heidelberg.

Sommerville, Ian. 2004. *Software Engineering, Eighth Edition*. Harlow: Addison-Wesley.