Genetic Algorithms

Theory and Practice

A Comprehensive Guide to Evolutionary Optimization

Course Materials Collection

November 30, 2025

Contents

List of Figures

List of Tables

Chapter 1
Pengantar Optimasi dan Komputasi Evolusioner

Algoritma genetika (GA) adalah metode pencarian berbasis populasi yang terinspirasi dari seleksi alam dan genetika. Mereka mempertahankan populasi solusi kandidat yang dikodekan sebagai string, secara iteratif menghasilkan generasi baru dengan memilih individu terbaik, menggabungkan informasi mereka, dan terkadang memperkenalkan variasi acak. Meskipun stokastik dalam operatornya, GA bukan random walk buta: mereka mempertahankan dan mengeksploitasi informasi historis tentang solusi yang baik untuk menghasilkan titik pencarian baru yang menjanjikan dan dengan demikian mendorong eksplorasi dan eksploitasi ruang kompleks yang efisien.

Keluarga metode ini dikembangkan dari karya fundamental Holland dan rekan-rekannya untuk memodelkan proses adaptif yang diamati di alam dan merancang sistem buatan yang mewujudkan mekanisme tersebut. Tujuan utamanya adalah ketahanan—kemampuan untuk menyeimbangkan efisiensi dengan keandalan di berbagai lingkungan masalah—yang membuat GA menarik ketika biaya perancangan ulang tinggi atau ketika struktur masalah melanggar asumsi umum (misalnya, kontinuitas, diferensiabilitas, atau unimodalitas). Karena mereka secara konseptual sederhana, dapat diterapkan secara luas, dan efektif secara empiris dalam optimasi dan kontrol, algoritma genetika telah menjadi alat praktis di berbagai domain teknik, sains, dan bisnis [21].

Untuk menempatkan algoritma genetika dalam konteks, pertama-tama kami memberikan definisi ringkas tentang optimasi — kelas masalah yang biasanya diselesaikan menggunakan GA. Kami mendefinisikan optimasi sebagai proses menemukan solusi terbaik dari sekumpulan alternatif yang tersedia. Dalam istilah matematis, masalah optimasi dapat diformulasikan sebagai:

$$\begin{aligned} \text{minimumkan (atau maksimumkan)} \quad & f(x) \\ \text{dengan batasan} \quad & g_i(x) \leq 0, \quad i = 1, 2, \ldots, m \\ & h_j(x) = 0, \quad j = 1, 2, \ldots, p \\ & x \in X \end{aligned} \tag{1.1}$$

di mana:

- $f(x)$ adalah fungsi objektif yang akan dioptimasi
- $g_i(x)$ adalah batasan ketidaksetaraan
- $h_j(x)$ adalah batasan persamaan
- X adalah wilayah yang layak

Masalah optimasi berbeda berdasarkan jenis variabel (diskrit, kontinu, atau mixed-integer) dan berdasarkan properti struktural seperti linearitas, konveksitas, dan jumlah objektif. Metode solusi tradisional mencakup teknik berbasis gradien (misalnya, metode Newton dan quasi-Newton) untuk masalah kontinu halus, pemrograman linear (metode

Simplex dan interior-point) untuk model linear, dan metode diskrit (branch-and-bound, pemrograman dinamis) untuk masalah kombinatorial. Namun, metode tradisional ini memiliki keterbatasan ketika diterapkan pada banyak masalah dunia nyata yang kompleks. Dalam bagian berikut kami menyoroti tiga tantangan utama di mana pendekatan konvensional sering mengalami kesulitan, dan menjelaskan bagaimana algoritma genetika dapat membantu mengatasinya.

Masalah pertama dengan metode optimasi tradisional adalah kecenderungan mereka untuk terjebak dalam optimum lokal. Dalam lanskap multi-modal dengan banyak puncak dan lembah, pencarian berbasis gradien dapat konvergen ke optimum lokal daripada optimum global. Ini terjadi karena metode-metode ini bergantung pada informasi gradien lokal untuk memandu proses pencarian. Ketika pencarian mencapai optimum lokal, gradien menjadi nol, menyebabkan algoritma berhenti berkembang. Keterbatasan ini terutama bermasalah dalam ruang berdimensi tinggi di mana jumlah optimum lokal dapat tumbuh secara eksponensial.

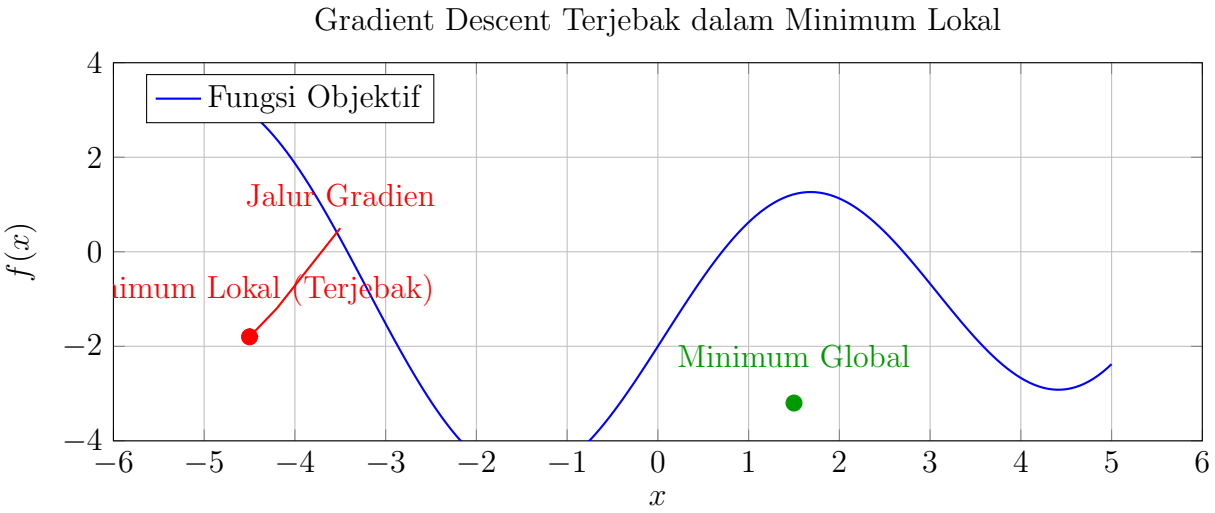

Figure 1.1: Metode berbasis gradien tradisional mengikuti gradien lokal dan menjadi terjebak dalam optimum lokal, tidak mampu melarikan diri untuk menemukan optimum global.

Masalah kedua dengan metode berbasis gradien adalah bahwa mereka memerlukan fungsi objektif yang dapat didiferensiasikan. Ini menjadi keterbatasan signifikan ketika menangani masalah dunia nyata yang melibatkan diskontinuitas, sudut tajam, atau lompatan diskrit. Masalah seperti itu umum dalam desain teknik, penjadwalan, dan optimasi kombinatorial.

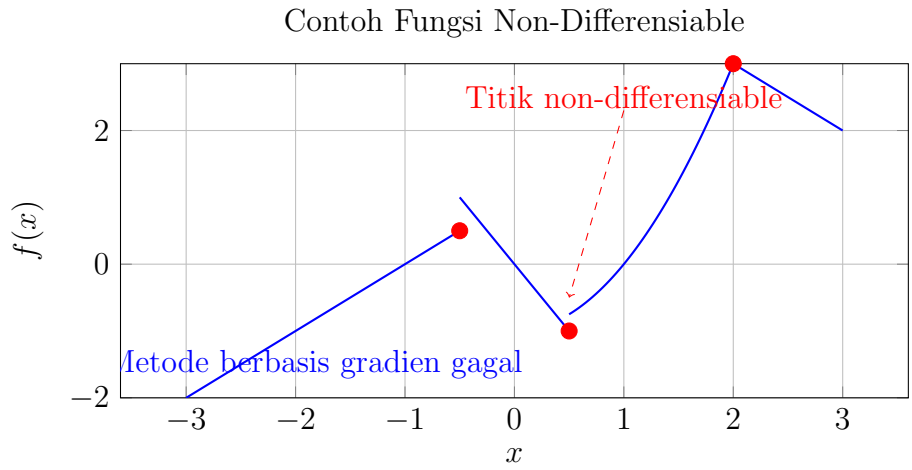

Figure 1.2: Fungsi dengan diskontinuitas, sudut tajam, atau lompatan diskrit tidak dapat dioptimalkan menggunakan metode berbasis gradien.

Sementara metode diskrit seperti pemrograman dinamis dapat menangani diskontinuitas dan struktur kombinatorial, baik algoritma berbasis gradien maupun algoritma diskrit eksak mengalami kutukan dimensionalitas: komputasi biasanya menjadi tidak dapat dilacak seiring dimensi masalah tumbuh.

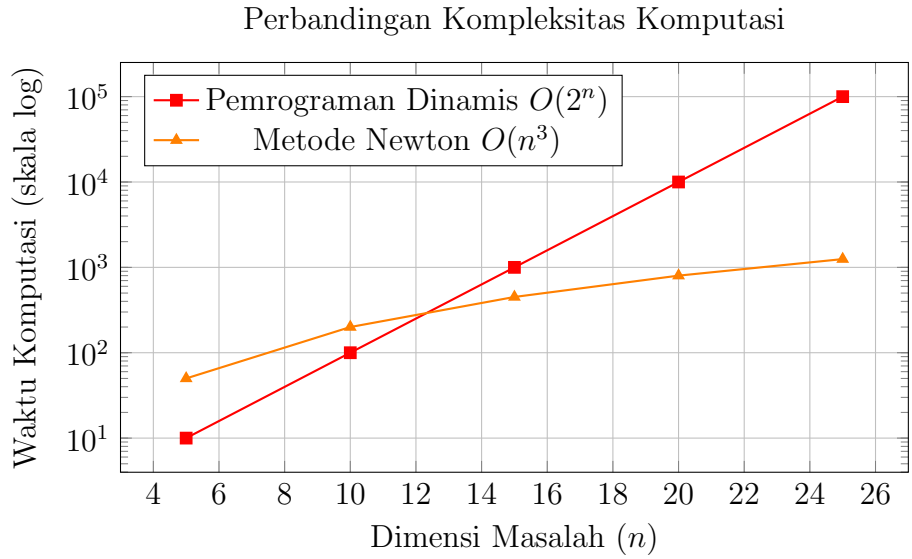

Figure 1.3: Metode optimasi tradisional sering menunjukkan pertumbuhan eksponensial atau polinomial tinggi dalam waktu komputasi seiring dimensi masalah meningkat, membuat mereka tidak praktis untuk masalah skala besar.

Oleh karena itu, kita dapat merangkum bagaimana algoritma genetika secara efektif mengatasi keterbatasan fundamental metode optimasi tradisional dalam tabel berikut:

Fitur	Tradisional	GA
Strategi Pencarian	Lokal	Global
Paralelisasi	Sulit	Alami
Multi-objektif	Kompleks	Bawaan
Penanganan Batasan	Sedang	Fleksibel

Figure 1.4: Perbandingan komprehensif menunjukkan bagaimana GA mengatasi keterbatasan fundamental metode optimasi tradisional.

1.1 Pengantar Komputasi Evolusioner

Komputasi evolusioner adalah keluarga algoritma pencarian stokastik berbasis populasi yang terinspirasi oleh prinsip-prinsip evolusi biologis [25, 13]. Anggota keluarga ini beroperasi pada populasi solusi kandidat dan berulang kali menerapkan seleksi (prinsip survival of the fittest), rekombinasi atau crossover (untuk bertukar informasi antar solusi), dan mutasi (untuk memperkenalkan variasi baru). Mekanisme ini memungkinkan algoritma evolusioner untuk mempertahankan dan menggabungkan kembali komponen solusi yang berguna sambil terus mengeksplorasi wilayah baru dari ruang pencarian.

Properti ini memberikan pendekatan evolusioner sejumlah keuntungan praktis untuk masalah optimasi yang sulit. Karena mereka bebas turunan dan hanya memerlukan evaluasi fungsi, metode evolusioner secara alami cocok untuk masalah optimasi black-box, termasuk fungsi objektif yang diskontinu, bising, tidak dapat didiferensiasikan, atau mengalami lompatan diskrit. Pencarian berbasis populasi mereka memberikan ketahanan terhadap optimum lokal dan memungkinkan evaluasi paralel kandidat solusi secara langsung, yang berharga untuk evaluasi fitness yang mahal. Sementara algoritma evolusioner umumnya tidak memberikan jaminan optimalitas formal, dengan representasi dan operator yang tepat mereka menawarkan kinerja heuristik yang andal di berbagai domain variabel kontinu, diskrit, dan campuran.

Bidang komputasi evolusioner mencakup beberapa keluarga yang mapan, masing-masing menekankan pilihan desain yang berbeda. Algoritma Genetika (GA) adalah salah satu formulasi paling awal dan paling berpengaruh, menekankan pengkodean panjang tetap dan operator crossover dan mutasi yang terinspirasi secara biologis [25, 21]. Strategi Evolusi (ES) berkonsentrasi pada strategi mutasi yang dapat beradaptasi sendiri dan sangat efektif untuk optimasi kontinu bernilai nyata [6]. Pemrograman Evolusioner (EP) secara historis fokus pada evolusi model perilaku dan skema mutasi stokastik daripada rekombinasi eksplisit [17, 18]. Pemrograman Genetika (GP) memperluas kerangka kerja ke struktur panjang variabel seperti program komputer dan pohon ekspresi, memungkinkan sintesis program otomatis [27].

Algoritma evolusioner telah berhasil diterapkan di berbagai bidang teknik, sains, dan bisnis. Dalam desain teknik mereka digunakan untuk mengoptimalkan konfigurasi sistem kompleks di mana evaluasi objektif mungkin mahal atau tidak dapat didiferensiasikan [47]. Dalam pembelajaran mesin, metode evolusioner telah digunakan baik untuk mengembangkan arsitektur jaringan saraf dan untuk mengoptimalkan bobot ketika informasi gradien tidak tersedia atau tidak dapat diandalkan [32, 33]. Penjadwalan, pembuatan jadwal waktu, perutean (termasuk Travelling Salesman Problem), dan masalah kombinatorial

lainnya adalah aplikasi alami karena pengkodean yang fleksibel dan operator rekombinasi khusus yang menghormati struktur masalah [23, 16]. Di luar domain ini, komputasi evolusioner telah menemukan peran dalam bioinformatika, pemodelan keuangan, dan desain strategi permainan otomatis, menunjukkan utilitas praktis yang luas [20, 38, 13].

Untuk membuat ide-ide ini konkret, pertimbangkan dua contoh ringkas yang biasa digunakan untuk ilustrasi pedagogis. Masalah mainan sederhana adalah memaksimalkan fungsi kuadrat $f(x) = x^2$ pada domain diskrit $0 \leq x \leq 31$. Dengan pengkodean biner (kromosom 5-bit), siklus berulang seleksi, crossover, dan mutasi mengkonsentrasikan blok bangunan yang mewakili nilai x yang lebih tinggi sampai, biasanya dalam beberapa generasi, individu yang mengkodekan optimum ($x = 31$) mendominasi populasi. Contoh ini menyoroti bagaimana rekombinasi mengakumulasi solusi parsial yang berguna (blok bangunan) bahkan ketika populasi awal acak.

Dalam contoh kombinatorial yang lebih menantang, Travelling Salesman Problem (TSP) meminta tur terpendek yang mengunjungi sekumpulan kota. Algoritma Genetika untuk TSP menggunakan representasi dan operator crossover yang mempertahankan properti urutan kota (misalnya, crossover berbasis urutan atau posisi) dan operator mutasi yang melakukan gangguan lokal tur. Sementara GA tidak menjamin optimalitas untuk masalah NP-hard seperti TSP, mereka sering menghasilkan solusi perkiraan berkualitas tinggi dengan cepat dan dapat dikombinasikan dengan pencarian lokal (pendekatan hibrid) untuk peningkatan lebih lanjut.

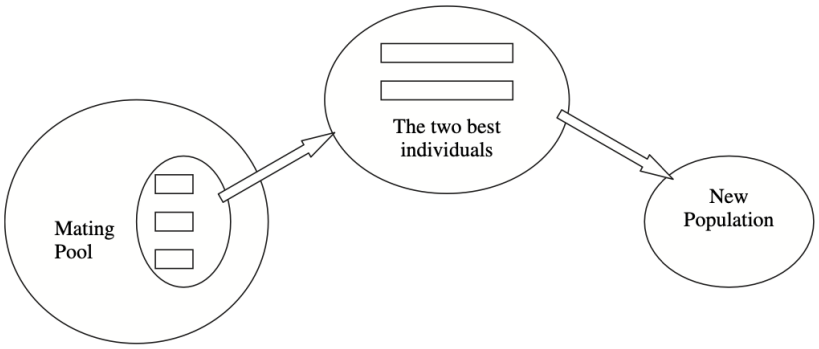

Figure 1.5: Ilustrasi siklus GA dan variasinya

Meskipun Algoritma Genetika Generasi Sederhana berfungsi sebagai kerangka dasar, berbagai modifikasi telah dikembangkan untuk meningkatkan kinerja dalam menangani kompleksitas masalah. Beberapa modifikasi ini termasuk varian yang telah dipelajari dengan baik berikut.

Algoritma Genetika Hibrid menggabungkan pencarian evolusioner global dengan prosedur peningkatan lokal yang ditargetkan untuk mengeksploitasi kekuatan komplementer dari kedua paradigma. Desain hibrid tipikal menggunakan GA untuk eksplorasi luas sambil menerapkan pencarian lokal deterministik atau stokastik (misalnya, hill-climbing, pencarian tabu, atau heuristik khusus masalah) untuk menyempurnakan individu terpilih atau solusi terbaik yang ditemukan sejauh ini. Dengan menggabungkan diversifikasi dan intensifikasi, metode hibrid sering mencapai konvergensi lebih cepat dan hasil berkualitas lebih tinggi pada tugas optimasi praktis [29, 33].

Algoritma Genetika Adaptif memodifikasi parameter kontrol secara online menggunakan umpan balik dari statistik populasi seperti tingkat peningkatan fitness, keberhasilan operator, atau ukuran keragaman genetik. Adaptasi dapat diimplementasikan

secara eksternal melalui aturan kontrol atau secara internal dengan mengkodekan parameter dalam individu sehingga evolusi itu sendiri memilih pengaturan yang efektif. Mekanisme ini mengurangi penyetelan manual dan membantu mempertahankan keseimbangan eksplorasi-eksploitasi yang produktif di berbagai fase pencarian [37, 41].

Algoritma Genetika Paralel mengeksploitasi perangkat keras paralel modern dengan mendistribusikan komputasi dan/atau struktur populasi. Model berbutir kasar (pulau) memegang sub-populasi pada prosesor yang berbeda dengan migrasi sesekali untuk berbagi informasi; model berbutir halus atau master-slave memparalelkan evaluasi fitness untuk mengurangi waktu wall-clock. Paralelisme tidak hanya mempercepat komputasi tetapi juga dapat meningkatkan ketahanan pencarian dengan melestarikan banyak relung dan mengurangi konvergensi prematur [13].

1.2 Variasi Algoritma Genetika

Meskipun Algoritma Genetika generasi sederhana berfungsi sebagai kerangka dasar, berbagai modifikasi telah dikembangkan untuk meningkatkan kinerja pada masalah kompleks. Variasi umum meliputi:

1.2.1 Tinjauan

- **GA Hibrid:** Menggabungkan Algoritma Genetika dengan pencarian lokal atau metode optimasi lain untuk menyempurnakan solusi yang menjanjikan setelah eksplorasi global [29, 33].
- **GA Adaptif:** Secara dinamis menyesuaikan parameter seperti probabilitas crossover dan mutasi berdasarkan umpan balik populasi untuk menyeimbangkan eksplorasi dan eksploitasi [37, 41].
- **GA Paralel:** Membagi populasi menjadi sub-populasi di seluruh prosesor (model pulau, master-slave, dll.) dan secara berkala bertukar individu untuk mempercepat pencarian dan mempertahankan keragaman.

1.3 Bacaan Lebih Lanjut

- Deb, K. (2001). Multi-objective optimization using evolutionary algorithms.
- Eiben, A. E., & Smith, J. E. (2015). Introduction to evolutionary computing.
- Goldberg, D. E. (1989). Genetic algorithms in search, optimization, and machine learning.

Chapter 2
Apa itu Algoritma Genetika?

2.1 Pendahuluan

Algoritma Genetika (GA) adalah metode pencarian dan optimasi acak yang mengambil inspirasi dari evolusi alami [25, 21]. Daripada meningkatkan satu solusi kandidat, GA mempertahankan populasi solusi potensial dan menerapkan operator yang terinspirasi secara biologis—seleksi, rekombinasi (crossover), dan mutasi—untuk menciptakan generasi solusi berturut-turut. Pendekatan berbasis populasi ini memungkinkan eksplorasi beberapa wilayah ruang pencarian secara paralel dan, bersama dengan variasi stokastik, sering membantu algoritma menghindari terjebak dalam optimum lokal.

GA sangat berguna untuk masalah dengan ruang pencarian yang besar, kompleks, atau kurang dipahami di mana informasi gradien tidak tersedia atau tidak dapat diandalkan. Fleksibilitas mereka dalam representasi dan operator membuat mereka dapat diterapkan pada berbagai domain, dari optimasi kombinatorial hingga penyetelan parameter kontinu dan regresi simbolik.

Individu	Biner	Desimal	Fitness
1	01101	13	169
2	11000	24	576
3	01000	8	64
4	10011	19	361

Table 2.1: Contoh Populasi Awal

Tabel di atas menunjukkan populasi awal kecil yang dikodekan dalam biner, bersama dengan nilai desimal masing-masing individu yang didekodekan dan fitnessnya. Contoh sederhana ini menggambarkan bagaimana solusi kandidat direpresentasikan dan dievaluasi, yang merupakan langkah pertama dalam implementasi algoritma genetika apa pun.

Setelah inisialisasi, GA secara iteratif mengevaluasi individu, memilih orang tua berdasarkan fitness, menerapkan crossover dan mutasi untuk menghasilkan keturunan, dan kemudian membentuk generasi berikutnya. Melalui siklus berulang ini, populasi cenderung membaik dan algoritma konvergen ke solusi berkualitas tinggi, tunduk pada pengkodean yang dipilih, fungsi fitness, dan pengaturan operator.

2.2 Inspirasi Biologis

Algoritma genetika meminjam ide-ide inti mereka dari teori seleksi alam dan adaptasi. Dalam populasi biologis, variasi muncul melalui rekombinasi dan mutasi, individu bersaing untuk sumber daya yang terbatas, dan mereka yang memiliki sifat turun-temurun yang memberikan keberhasilan reproduktif yang lebih tinggi cenderung meninggalkan lebih banyak keturunan. Selama banyak generasi proses ini mengarah pada populasi yang

lebih beradaptasi dengan lingkungan mereka; kerangka GA mengabstraksi mekanisme ini untuk mendorong peningkatan solusi kandidat dalam proses pencarian [25, 31].

Dalam metafora GA, seleksi mendukung individu dengan fitness lebih tinggi sebagai orang tua untuk generasi berikutnya, crossover menggabungkan materi genetik dari orang tua untuk mengeksplorasi wilayah baru dari ruang pencarian, dan mutasi memperkenalkan perubahan acak yang mempertahankan keragaman genetik dan memungkinkan solusi yang belum pernah terlihat sebelumnya muncul. Mekanisme ini—seleksi, rekombinasi, dan mutasi—bekerja bersama untuk menyeimbangkan eksplorasi dan eksploitasi selama pencarian, memungkinkan adaptasi bertahap populasi menuju solusi berkualitas lebih tinggi [13].

2.3 Terminologi Dasar

2.3.1 Istilah Algoritma Genetika

Memahami terminologi umum membantu menjembatani metafora biologis dan implementasi algoritmiknya [31, 21]. Seorang **individu** atau **kromosom** menunjukkan solusi kandidat tunggal; itu terdiri dari satu atau lebih **gen**, di mana setiap gen mewakili komponen solusi dan **alel** adalah nilai spesifik yang dipegang oleh gen. **Populasi** adalah kumpulan individu yang dipertahankan algoritma pada waktu tertentu, dan **generasi** mengacu pada satu iterasi siklus evolusi di mana seleksi, rekombinasi, dan mutasi menghasilkan populasi berikutnya. **Fitness** dari seorang individu mengukur kualitasnya sehubungan dengan tujuan optimasi dan digunakan untuk membiaskan seleksi ke arah solusi yang lebih baik. **Genotipe** menggambarkan representasi yang dikodekan yang digunakan oleh algoritma (misalnya, string biner atau vektor nilai nyata), sementara **fenotipe** adalah bentuk yang didekodekan atau ditafsirkan dari genotipe tersebut (instance solusi sebenarnya yang dievaluasi oleh fungsi fitness). Mengklarifikasi istilah-istilah ini berguna ketika merancang representasi dan operator, karena pilihan implementasi pada tingkat genotipe menentukan fenotipe apa yang dapat diekspresikan dan oleh karena itu mempengaruhi perilaku pencarian dan efektivitas GA [13].

2.4 Struktur Dasar Algoritma Genetika

Algoritma genetika (GA) adalah prosedur pencarian stokastik berbasis populasi yang mengubah seperangkat solusi kandidat melalui aplikasi berulang operator variasi dan seleksi. Secara formal, GA dapat digambarkan oleh tupel (X, Φ, f, S, C, M, R) di mana X adalah ruang pencarian (fenotipe), Φ adalah pengkodean yang memetakan genotipe ke fenotipe, $f\colon X \to \mathbb{R}$ adalah fungsi fitness, S adalah operator seleksi, C operator rekombinasi (crossover), M operator mutasi, dan R operator penggantian (seleksi survivor). Pada generasi t algoritma mempertahankan populasi $P_t \subseteq \Gamma$ dari genotipe (di mana Γ menunjukkan set pengkodean); operator bertindak untuk menghasilkan populasi baru P_{t+1} sesuai dengan skema

$$P_{t+1} = R\big(P_t,\ \{\, C \circ M(\pi) : \pi \in \Pi(S(P_t)) \,\}\big),$$

di mana $S(P_t)$ menunjukkan multiset seleksi orang tua yang ditarik dari P_t, $C \circ M$ menunjukkan bahwa keturunan diproduksi dengan menerapkan mutasi dan crossover pada orang tua yang dipilih, dan R menentukan individu mana yang bertahan ke generasi berikutnya.

Deskripsi abstrak ini menangkap loop kanonik dari inisialisasi, evaluasi, seleksi, variasi, dan penggantian yang berulang sampai kondisi terminasi (misalnya, anggaran komputasi tetap, fitness target, atau kurangnya peningkatan) terpenuhi [25, 31, 13].

Dalam praktiknya desain setiap komponen sangat mempengaruhi perilaku pencarian. Pengkodean Φ menentukan solusi apa yang dapat direpresentasikan dan bagaimana operator variasi mengeksplorasi ruang fenotipe; fungsi fitness f mendefinisikan tujuan optimasi dan memberikan sinyal seleksi; operator seleksi S mengontrol tekanan selektif terhadap individu dengan fitness lebih tinggi (contoh termasuk skema proporsional fitness, turnamen, dan berbasis peringkat); rekombinasi C mencampur informasi antara orang tua untuk mengeksplorasi wilayah baru dari ruang pencarian; mutasi M memperkenalkan gangguan acak untuk melestarikan keragaman dan memungkinkan eksplorasi lokal; dan kebijakan penggantian R menyeimbangkan retensi solusi yang baik dengan pengenalan keturunan segar. Pilihan desain ini mewujudkan trade-off eksplorasi-eksploitasi yang dibahas dalam Bagian **??** dan merupakan tuas utama untuk mengadaptasi GA ke domain masalah tertentu [21, 13].

Dilihat secara algoritmik, GA melakukan langkah-langkah tingkat tinggi berikut setiap generasi: evaluasi f pada P_t, pilih orang tua menggunakan S, produksi keturunan melalui C dan M, dan bentuk P_{t+1} melalui R. Meskipun banyak varian ada (pembaruan steady-state, model pulau, skema hibrid yang menggabungkan pencarian lokal), struktur kanonik ini menjelaskan baik fleksibilitas empiris GA dan alasan untuk biaya komputasi mereka: evaluasi fitness berulang atas populasi dapat mahal, tetapi pendekatan berbasis populasi memungkinkan eksplorasi paralel dan ketahanan terhadap multimodalitas dan kebisingan [31, 21].

2.5 Keunggulan Algoritma Genetika

Karena GA memanipulasi populasi solusi kandidat menggunakan seleksi, rekombinasi, dan mutasi, ia membawa beberapa keunggulan praktis yang mengikuti langsung dari desain berbasis populasi yang didorong variasi tersebut.

Pertama, algoritma genetika menyediakan mekanisme pencarian global yang efektif: dengan mengeksplorasi banyak titik di ruang pencarian secara bersamaan dan menggabungkan informasi dari banyak orang tua, GA dapat melarikan diri dari optimum lokal dan menemukan beragam cekungan atraksi dalam lanskap multimodal [21]. Rekombinasi memungkinkan pencampuran blok bangunan yang berguna dari individu yang berbeda, sementara mutasi menyuntikkan variasi baru yang dapat mengarahkan pencarian ke wilayah yang sebelumnya belum dijelajahi.

Kedua, sifat berbasis populasi dari GA membuat mereka secara alami dapat diparalelkan. Evaluasi fitness untuk individu yang berbeda independen dan dapat didistribusikan di seluruh prosesor atau mesin, yang mengurangi biaya komputasi evaluasi populasi besar dan memungkinkan penggunaan efisien perangkat keras paralel modern [13].

Ketiga, GA fleksibel dalam desain representasi dan operator. Pengkodean (genotipe) dapat dipilih untuk sesuai dengan ruang pencarian kombinatorial, kontinu, atau terstruktur, dan operator dapat disesuaikan untuk melestarikan batasan khusus masalah atau mengeksploitasi pengetahuan domain. Fleksibilitas representasi ini berarti GA dapat diterapkan pada berbagai jenis masalah di mana pengoptimal yang lebih khusus akan memerlukan pengerjaan ulang yang substansial [38].

Keempat, karena GA tidak bergantung pada informasi gradien, mereka bekerja dengan baik dengan fungsi objektif yang diskontinu, bising, atau tidak dapat didiferensiasikan. Ini

membuat mereka pilihan yang baik ketika metode berbasis turunan tidak dapat diterapkan atau tidak dapat diandalkan [31].

Akhirnya, GA cenderung kuat dalam menghadapi kebisingan dan ketidakpastian: keragaman populasi dan variasi stokastik membantu mencegah konvergensi prematur ke solusi palsu ketika evaluasi fitness bising atau tidak tepat [24]. Secara bersama-sama, keunggulan ini menjelaskan mengapa algoritma genetika banyak digunakan sebagai alat optimasi tujuan umum, sambil juga menyoroti bahwa kesesuaian mereka tergantung pada struktur masalah dan sumber daya komputasi yang tersedia.

2.6 Kerugian Algoritma Genetika

Terlepas dari kekuatan mereka, algoritma genetika juga memiliki keterbatasan praktis yang mengikuti dari fitur desain yang sama yang disorot di bagian sebelumnya. Yang paling menonjol, ketergantungan pada populasi dan evaluasi fitness berulang membuat GA mahal secara komputasi untuk masalah di mana evaluasi fitness tunggal mahal. Menjalankan populasi besar selama banyak generasi dapat memerlukan waktu CPU atau waktu wall-clock yang substansial kecuali evaluasi diparalelkan atau dipercepat dengan cara lain [31].

Kerugian penting lainnya adalah sensitivitas terhadap pengaturan parameter. GA mengekspos banyak parameter yang dapat disetel—ukuran populasi, tingkat crossover dan mutasi, tekanan seleksi, strategi penggantian, dan kriteria terminasi—dan pilihan parameter ini sangat mempengaruhi kinerja. Menemukan konfigurasi parameter yang baik sering memerlukan eksperimen, penyetelan otomatis, atau keahlian domain; pengaturan yang buruk dapat menyebabkan pencarian yang tidak efisien atau kegagalan untuk konvergen ke solusi yang memuaskan [13].

Selain itu, tidak ada jaminan formal bahwa GA akan menemukan optimum global dalam waktu terbatas. Seperti kebanyakan metode pencarian heuristik, GA adalah stokastik dan memberikan jaminan probabilistik daripada deterministik; mereka paling baik dipandang sebagai heuristik pencarian yang kuat daripada pengoptimal eksak. Keterbatasan ini terutama relevan ketika sertifikat optimalitas diperlukan oleh aplikasi atau ketika ruang pencarian memiliki fitur patologis yang menyesatkan eksplorasi berbasis populasi [21].

Masalah terkait erat adalah konvergensi prematur: populasi dapat kehilangan keragaman dan menjadi didominasi oleh individu yang mirip, yang mengurangi kemampuan algoritma untuk mengeksplorasi wilayah baru dari ruang pencarian. Konvergensi prematur sering disebabkan oleh tekanan seleksi yang berlebihan, rekombinasi yang terlalu mengganggu, atau populasi yang terlalu kecil, dan dapat dikurangi melalui strategi seperti mempertahankan keragaman (niching, crowding), kontrol parameter adaptif, hibridisasi dengan pencarian lokal, atau menggunakan model pulau yang melestarikan sub-populasi terpisah [13, 24].

Mengenali kerugian ini mengklarifikasi trade-off yang dibahas dalam Bagian 2.5: mekanisme yang sama yang memberikan GA ketahanan dan fleksibilitas mereka juga menciptakan biaya yang harus dikelola melalui desain algoritma yang hati-hati, penyetelan parameter, dan sumber daya komputasi. Untuk banyak masalah praktis, manfaatnya melebihi biayanya, tetapi mengevaluasi keseimbangan itu adalah langkah penting ketika memilih apakah akan menerapkan algoritma genetika pada tugas yang diberikan.

2.7 Kapan Menggunakan Algoritma Genetika

Memilih untuk menggunakan algoritma genetika tergantung pada penilaian struktur masalah, sumber daya komputasi yang tersedia, dan tujuan pencarian. GA paling menarik ketika ruang pencarian besar, kompleks, atau kurang dipahami: eksplorasi berbasis populasi mereka dan fleksibilitas representasi memungkinkan mereka menemukan solusi di mana informasi turunan tidak tersedia atau pengoptimal konvensional kesulitan.

Ketika sedikit yang diketahui tentang struktur masalah atau ketika fungsi objektif diskontinu, bising, atau multimodal, GA memberikan alternatif praktis untuk metode berbasis gradien atau khusus masalah. Tidak adanya persyaratan untuk diferensiabilitas dan kemampuan untuk beroperasi pada pengkodean kombinatorial dan terstruktur membuat GA berguna dalam desain teknik, penjadwalan, regresi simbolik, dan domain serupa di mana pengoptimal klasik tidak dapat diterapkan [31, 38].

GA juga merupakan pilihan alami ketika beberapa tujuan yang sering bertentangan harus dieksplorasi secara bersamaan. Varian multi-objektif menghasilkan set solusi perkiraan Pareto yang beragam, memungkinkan pembuat keputusan untuk memeriksa trade-off daripada memaksa tujuan yang diskalarisasi tunggal [12, 13].

Namun, keunggulan yang tercantum dalam Bagian 2.5 harus ditimbang terhadap kerugian yang dibahas dalam Bagian 2.6. Jika evaluasi fitness sangat mahal dan sumber daya paralel tidak tersedia, beban komputasi mempertahankan dan mengembangkan populasi dapat melebihi manfaatnya. Demikian pula, jika jaminan optimalitas yang ketat diperlukan, sifat heuristik dan stokastik GA mungkin tidak tepat. Dalam kasus seperti itu, pendekatan hibrid (menggabungkan GA dengan pencarian lokal atau model surrogate), penyetelan parameter yang hati-hati, atau penggunaan pengoptimal khusus dapat menawarkan trade-off yang lebih baik.

Dalam praktiknya, aturan keputusan yang berguna adalah lebih memilih algoritma genetika ketika ketahanan, fleksibilitas, dan kemampuan untuk menangani ruang pencarian yang kompleks atau berperilaku buruk lebih penting daripada efisiensi mentah atau jaminan optimalitas formal. Di mana kondisi ini berlaku, menerapkan variasi dan strategi mitigasi yang dijelaskan sebelumnya (evaluasi paralel, model pulau, hibrid, dan kontrol adaptif) sering menghasilkan solusi praktis berkualitas tinggi.

Chapter 3
Siklus GA dan Teori Skema Holland

3.1 Siklus Algoritma Genetika

Algoritma genetika mengikuti proses siklik yang meniru evolusi alami. Memahami siklus ini penting untuk mengimplementasikan dan menganalisis kinerja GA.

3.1.1 Siklus GA secara Rinci

Algoritma genetika dirancang untuk secara iteratif meningkatkan sekumpulan kandidat solusi. Proses dimulai dengan populasi solusi potensial dan secara berulang menerapkan operasi evaluasi dan variasi untuk bergerak menuju solusi yang lebih baik. Siklus umum ini bersifat modular: inisialisasi menyiapkan pencarian, evaluasi mengukur kualitas saat ini, dan seleksi membentuk generasi berikutnya. Langkah-langkah ini membentuk sebuah loop yang berlanjut hingga terpenuhi kondisi berhenti.

Alur operasi digambarkan pada diagram di bawah ini yang menunjukkan urutan aksi tipikal dalam GA klasik: inisialisasi, evaluasi, pengecekan terminasi, pemilihan orang tua, penerapan crossover dan mutasi, seleksi, dan kembali ke evaluasi. Setiap langkah memiliki banyak implementasi dan parameter yang memengaruhi eksplorasi dan eksploitasi; misalnya ukuran populasi, tekanan seleksi, tipe crossover, dan laju mutasi memengaruhi bagaimana pencarian menavigasi ruang solusi. Meskipun diagram memperlihatkan urutan sederhana, algoritma praktis sering menyertakan peningkatan seperti laju adaptif, atau pembaruan steady-state yang mengubah detail bagaimana keturunan dan induk digabungkan.

Inisialisasi adalah langkah praktis pertama dan menentukan titik awal pencarian. Populasi awal berukuran N dapat dihasilkan secara acak untuk memberikan cakupan luas ruang solusi, atau diisi dengan heuristik spesifik masalah untuk memberi awal yang lebih baik di wilayah yang menjanjikan. Menjaga keberagaman pada populasi awal penting karena mengurangi risiko konvergensi prematur dan meningkatkan kemungkinan bahwa blok bangunan berguna hadir sejak awal. Selain kandidat solusi itu sendiri, inisialisasi biasanya mencakup penetapan penghitung atau parameter tingkat-algoritma, seperti indeks generasi $t = 0$, dan pencatatan status yang diperlukan untuk operator adaptif.

Evaluasi mengkuantifikasi seberapa baik setiap individu menyelesaikan masalah dan mengubah solusi mentah menjadi nilai fitness yang digunakan oleh GA. Langkah ini umumnya menghitung fungsi objektif atau fitness untuk setiap anggota populasi, dan dapat juga mengumpulkan statistik ringkasan seperti rata-rata populasi, varians, serta fitness terbaik dan terburuk. Statistik tersebut berguna untuk memantau kemajuan, mendiagnosis stagnasi, dan menggerakkan mekanisme adaptif. Karena evaluasi sering menjadi bagian paling mahal dari GA—terutama jika setiap komputasi fitness melibatkan simulasi atau perhitungan kompleks—praktisi memperhatikan efisiensi evaluasi dan penggunaan kembali komputasi bila memungkinkan.

Terminasi adalah keputusan kendali yang diperiksa setelah evaluasi untuk memutuskan apakah algoritma harus berhenti atau dilanjutkan. Kondisi berhenti umum termasuk mencapai jumlah generasi maksimum yang ditetapkan, mencapai ambang fitness

yang memadai, mengamati konvergensi populasi atau keberagaman yang sangat rendah, tidak ada perbaikan selama sejumlah generasi tertentu, atau habisnya anggaran evaluasi fungsi. Memilih kriteria terminasi merupakan trade-off antara biaya komputasi dan kualitas solusi: berhenti terlalu awal berisiko kehilangan solusi lebih baik, sementara berjalan terlalu lama membuang sumber daya dengan pengembalian yang menurun. Dalam praktiknya seringkali digabungkan beberapa kondisi (misalnya berhenti apabila target fitness tercapai atau batas generasi dilewati).

3.1.2 Apa itu Skema?

Skema adalah template formal yang didefinisikan di atas alfabet $\{0, 1, *\}$. Untuk panjang string tetap l, sebuah skema

$$H \in \{0, 1, *\}^l$$

menentukan nilai yang dibutuhkan pada beberapa lokus dan membiarkan lokus lain tidak ditentukan (simbol "don't care"). Misalkan $\Sigma = \{0, 1\}$ dan Σ^l himpunan semua string biner panjang-l. Kita mengatakan sebuah string konkret $s \in \Sigma^l$ memenuhi skema H (tulis $s \in [H]$) jika setiap posisi yang terdefinisi pada H cocok dengan s:

$$s \in [H] \quad \Leftrightarrow \quad \forall i \in \{1, \ldots, l\},\ H_i \neq * \Rightarrow s_i = H_i. \tag{3.1}$$

Himpunan $[H] = \{s \in \Sigma^l : s \text{ matches } H\}$ adalah kelas ekivalen genom konkret yang direpresentasikan oleh H. Jika $k(H)$ menyatakan jumlah simbol don't-care pada H (jadi $k(H) = |\{i :\ H_i = *\}|$), maka kardinalitas kelas ini adalah

$$|[H]| = 2^{k(H)}. \tag{3.2}$$

Dua kuantitas lain yang sering dipakai adalah orde dan panjang pendefinisian. Orde $o(H)$ sama dengan jumlah posisi yang tetap (simbol bukan-*), jadi $o(H) = l - k(H)$. Jika $i_{\min}$ dan $i_{\max}$ adalah indeks posisi tetap pertama dan terakhir pada H, panjang pendefinisian didefinisikan sebagai

$$\delta(H) = i_{\max} - i_{\min}. \tag{3.3}$$

Contoh: untuk $H = 1 * 0 * 1$ dengan $l = 5$ kita memiliki posisi tetap di indeks $1, 3, 5$, $k(H) = 2$, $o(H) = 3$, dan

$$|[H]| = 2^2 = 4, \qquad \delta(H) = 5 - 1 = 4,$$

dengan string yang cocok {10001,10011,11001,11011}.

3.1.3 Properti Skema

Orde Skema

Orde $o(H)$ adalah jumlah posisi tetap (simbol bukan-*):

$$o(H) = \text{jumlah bit terdefinisi pada } H \tag{3.4}$$

Untuk $H = 1 * 0 * 1$: $o(H) = 3$

Panjang Pendefinisian

Panjang pendefinisian $\delta(H)$ mengukur rentang antara posisi tetap pertama dan terakhir pada pola. Secara intuitif, ini menunjukkan seberapa tersebar bit-bit penting skema sepanjang kromosom dan oleh karena itu seberapa terekspos skema tersebut terhadap kejadian rekombinasi. Secara formal dihitung sebagai selisih indeks antara posisi tetap terakhir dan pertama:

$$\delta(H) = \text{posisi tetap terakhir} - \text{posisi tetap pertama} \tag{3.5}$$

Panjang pendefinisian kecil berarti bit tetap skema berkumpul rapat. Signifikansi praktis panjang pendefinisian terkait erat dengan pandangan blok-bangunan dalam GA: blok gen pendek yang saling terhubung yang memberi fitness di atas rata-rata.

Untuk $H = 1 * 0 * 1$: $\delta(H) = 5 - 1 = 4$

Contoh dari Buku Ajar:

- $S_1 = (* * *001 * 110)$: $\delta(S_1) = 10 - 4 = 6$
- $S_2 = (* * * * 00 * *0*)$: $\delta(S_2) = 9 - 5 = 4$
- $S_3 = (11101 * *001)$: $\delta(S_3) = 10 - 1 = 9$

3.1.4 Teorema Skema (Teorema Fundamental)

Teorema skema menjelaskan bagaimana jumlah harapan string yang memenuhi sebuah skema berubah dari satu generasi ke generasi berikutnya.

Teorema Skema Gabungan

Menggabungkan semua pengaruh:

$$E[m(H, t+1)] \geq m(H, t) \cdot \frac{f(H)}{\bar{f}} \cdot \left(1 - p_c \cdot \frac{\delta(H)}{l-1}\right) \cdot (1 - p_m)^{o(H)} \tag{3.6}$$

3.1.5 Hipotesis Blok-Bangunan

Hipotesis blok-bangunan dapat dinyatakan secara tepat menggunakan formalismo skema Holland: sebuah blok-bangunan adalah skema H dengan panjang pendefinisian kecil $\delta(H)$, orde rendah $o(H)$, dan fitness di atas rata-rata $f(H) > \bar{f}$. Teorema skema memberi kriteria kuantitatif agar skema tersebut tumbuh dalam ekspektasi dari generasi t ke $t+1$:

$$E[m(H, t+1)] \geq m(H, t) \cdot \frac{f(H)}{\bar{f}} \cdot \left(1 - p_c \frac{\delta(H)}{l-1}\right) \cdot (1 - p_m)^{o(H)}. \tag{3.7}$$

Dengan demikian, kondisi yang diperlukan (tetapi tidak cukup) agar H tumbuh secara harapan adalah faktor multiplikatif di ruas kanan melebihi satu. Menyusun ulang memberi kondisi ambang informal

$$\frac{f(H)}{\bar{f}} > \frac{1}{\left(1 - p_c \frac{\delta(H)}{l-1}\right)(1 - p_m)^{o(H)}}. \tag{3.8}$$

Ketidaksamaan ini memperjelas trade-off: fitness relatif lebih tinggi, panjang pendefinisian lebih kecil, dan orde lebih rendah meningkatkan peluang sebuah skema berkembang.

3.2 Paralelisme Implisit

GA bekerja pada populasi genom konkret, tetapi setiap genom konkret secara simultan menginstansiasi keluarga skema yang jumlahnya eksponensial. Secara konkret, untuk string biner panjang l ada 3^l skema yang mungkin (setiap posisi bisa 0, 1, atau "don't care"). Satu string panjang-l cocok dengan tepat 2^l skema berbeda karena setiap lokus bisa dibiarkan tetap atau diganti simbol don't-care. Akibatnya, populasi berukuran n menyediakan contoh langsung paling banyak $n2^l$ skema (menghitung multiplikitas), dan—mengabaikan tumpang tindih antar individu—angka ini bisa eksponensial besar terhadap l. Fakta kombinatorial ini mendasari gagasan intuitif bahwa GA mengevaluasi banyak skema secara paralel.

Perhitungan yang lebih rinci mengelompokkan skema menurut orde mereka (jumlah lokus tetap). Jumlah skema orde r adalah

$$\binom{l}{r}2^r, \tag{3.9}$$

karena dipilih r lokus yang tetap lalu diberi bit (0 atau 1) pada setiap lokus tetap. Jumlah skema yang orde-nya tidak melebihi k adalah

$$S_k = \sum_{r=0}^{k} \binom{l}{r}2^r, \tag{3.10}$$

yang, untuk k kecil tetap, tumbuh secara polinomial terhadap l (derajat k) bukan eksponensial.

Ungkapan terkenal Holland "paralelisme implisit" merangkum heuristik bahwa populasi berukuran sedang dapat secara kolektif mengevaluasi dan memproses sejumlah besar skema pendek dan orde-rendah pada setiap generasi. Dalam eksposisinya, ia menawarkan aturan praktis bahwa populasi n string dapat efektif memproses urutan skema sebesar n^3; pernyataan ini harus dibaca sebagai heuristik, bukan identitas kombinatorial ketat. Angka $O(n^3)$ muncul dari asumsi tentang orde-orde tipikal dan panjang pendefinisian skema yang memengaruhi fitness, bersama dengan estimasi plausibel tentang berapa banyak skema pendek berbeda yang disampling populasi dan bagaimana seleksi memperkuat instance di atas rata-rata. Pilihan berbeda untuk n, l, representasi, dan pengaturan operator mengubah faktor konstanta dan jangkauan praktis paralelisme ini.

Implikasi praktisnya dua arah. Pertama, dengan bekerja pada populasi alih-alih satu lintasan pencarian tunggal, GA dapat menjelajah dan menyebarkan banyak blok-bangunan kandidat secara bersamaan, memungkinkan rekombinasi konstruktif pola-pola pendek yang berguna. Kedua, cakupan implisit ini bersifat selektif: algoritma menyalurkan daya pemrosesan efektif ke skema yang cukup sering disampling (muncul cukup sering) dan cukup tangguh (memiliki panjang pendefinisian kecil dan orde rendah). Akibatnya, paralelisme implisit bukanlah solusi ajaib yang memeriksa setiap skema sama rata; melainkan mengarahkan upaya komputasi kepada subset terstruktur besar dari skema yang paling relevan berdasarkan enkoding dan operator yang dipilih.

3.3 Deception dan Teori Skema

3.3.1 Masalah Menipu (Deceptive)

Deception terjadi ketika seleksi, yang bekerja pada sinyal fitness jangka pendek, secara sistematis mendorong populasi menjauh dari genotip yang menuju optimum global. Secara informal, sebuah fungsi fitness bersifat menipu terhadap sekumpulan skema orde-rendah jika skema yang tampak "terbaik" secara lokal (mis. memiliki fitness di atas rata-rata di antara instance-nya) adalah skema yang rekombinasinya tidak merakit solusi global melainkan mempromosikan genotip yang sulit diperbaiki. Dengan kata lain, seleksi memberi penghargaan pada blok-bangunan yang mengarahkan pencarian ke optimum lokal daripada ke optimum global.

Fenomena ini dapat dinyatakan dengan istilah skema. Pertimbangkan dua kelas skema terpisah H_1 dan H_2 yang didefinisikan pada set lokus terpisah. Jika instance paling fit dari H_1 dan H_2 cenderung menghasilkan keturunan yang gabungan fitnesnya lebih rendah daripada kombinasi alternatif (atau jika penggabungan mereka tidak mungkin di bawah operator yang dipilih), maka seleksi yang bekerja pada H_1 dan H_2 dapat meningkatkan frekuensi keduanya meskipun keberadaan bersama mereka merugikan tercapainya optimum global. Ketidaksesuaian antara fitness skema jangka pendek dan nilai konstruktif jangka panjang inilah inti deception.

Contoh benchmark klasik yang menggambarkan deception adalah masalah trap-deceptive yang dikonkatenasi. Genom dibagi menjadi m blok terpisah masing-masing berukuran k. Setiap blok memberikan kontribusi fitness blok-wise yang maksimum pada konfigurasi tertentu (optimum global blok) tetapi selain itu memberikan fitness lebih tinggi pada konfigurasi yang menarik secara lokal namun tidak mengarah ke optimum blok. Jika blok-blok sederhana dan independen, rekombinasi dapat merakit optimum blok; jika blok menipu, seleksi memperkuat konfigurasi lokal yang salah dan rekombinasi saja mungkin tidak mampu menyelamatkan solusi global.

3.3.2 Mengapa Deception Penting bagi Teori Skema

Teori skema memprediksi bahwa skema orde-rendah dan panjang-pendefinisian pendek dengan fitness di atas rata-rata akan meningkat dalam ekspektasi di bawah seleksi. Deception merusak penalaran ini dengan membuat skema-skema lokal yang tampak di atas rata-rata justru menjauhkan populasi dari genotip yang mengandung kombinasi blok-bangunan optimal global. Jadi, walaupun pernyataan aljabar teorema skema tetap benar dalam ekspektasi, interpretasi konstruktifnya (bahwa seleksi akan merakit blok-bangunan baik menjadi solusi lebih baik) dapat gagal ketika skema pendek yang disampling menyesatkan.

3.3.3 Mendeteksi dan Mengukur Deception

Secara praktis, deception dinilai dengan menganalisis bagaimana perbaikan lokal berkorelasi dengan kemajuan global. Diagnostik umum meliputi:

- Korelasi jarak-fitness (Fitness-Distance Correlation, FDC): korelasi antara fitness dan jarak ke optimum global yang diketahui. Korelasi negatif kuat menunjukkan pencarian lebih mudah; korelasi lemah atau positif dapat menandakan deception.

- Analisis blok empiris: untuk masalah yang dapat didekomposisi (mis. trap yang dikonkatenasi), mempelajari lanskap fitness blok-wise (plot unitation) mengungkap apakah optimum lokal menarik pencarian dalam blok.

- Sensitivitas performa: mengukur probabilitas keberhasilan sebagai fungsi ukuran populasi dan pengaturan operator dapat menunjukkan apakah perubahan kecil menghilangkan atau menimbulkan perilaku menipu.

3.3.4 Strategi Mengatasi Deception

Karena deception muncul dari ketidakcocokan antara representasi, operator, dan struktur modular masalah, langkah-langkah penanggulangannya umumnya jatuh pada tiga kategorial: meningkatkan sampling, mempertahankan diversitas yang berguna, dan meningkatkan kemampuan algoritma menghormati atau mempelajari linkage.

- **Tingkatkan sampling efektif:** Populasi lebih besar dan tekanan seleksi konservatif mengurangi kemungkinan skema menipu cepat fix.

- **Metode pelestarian diversitas:** Niching (fitness sharing, crowding), model pulau, dan restricted tournament selection mempertahankan alel atau subpopulasi bersaing sehingga kombinasi blok alternatif tidak hilang prematur.

- **Pembelajaran linkage dan metode estimasi:** Algoritma yang mempelajari ketergantungan—messy GAs, linkage-tree GAs, hierarchical BOA, dan pemodelan struktur dependensi—secara eksplisit mendeteksi dan melindungi gen yang saling berinteraksi alih-alih bergantung pada rekombinasi buta.

- **Hibridisasi dan pencarian lokal:** Menggabungkan GA dengan pencarian lokal spesifik masalah (algoritma memetik) atau heuristik konstruktif membantu memperbaiki atau menyelesaikan solusi parsial yang rekombinasi murni tidak dapat rakit.

3.4 Implikasi Praktis

3.4.1 Pemilihan Operator dan Parameter

Pertimbangan berdasarkan skema menunjukkan trade-off tertentu saat memilih operator dan parameternya:

- **Ukuran populasi (n):** Populasi lebih besar mengurangi noise sampling dan meningkatkan probabilitas keberadaan skema orde-rendah yang berguna dalam multiplicity yang cukup. Gunakan aturan ukuran populasi atau eksperimen untuk memastikan sampling andal.

3.4.2 Pemantauan dan Diagnostik Praktis

Untuk menerapkan desain yang diinformasikan skema dalam praktik, pantau statistik populasi secara berkala:

- Lacak keberagaman genotip (mis. frekuensi alel per-lokus) dan varians fenotip untuk mendeteksi konvergensi prematur.

- Hitung hitungan skema sederhana atau sampling skema kandidat untuk memverifikasi apakah blok-bangunan yang diharapkan ditemukan dan dipertahankan.
- Ukur metrik kemajuan (fitness terbaik, median, rata-rata) bersamaan dengan indikator keberagaman; perbaikan lambat dengan diversitas yang runtuh sering menandakan kegagalan rekombinasi skema.

3.5 Keterbatasan Teori Skema

Teori skema memberikan kerangka konseptual dan analitik yang bernilai, tetapi asumsi dan cakupannya menimbulkan keterbatasan penting yang harus dikenali praktisi.

3.5.1 Ekspektasi vs dinamika populasi terbatas

Pernyataan aljabar inti dari teori skema adalah pernyataan tentang ekspektasi. Pada populasi terbatas, noise sampling stokastik, genetic drift, dan galat sampling dapat menyebabkan dinamika realisasi menyimpang jauh dari ekspektasi. Oleh karena itu prediksi berbasis skema harus ditafsirkan sebagai kecenderungan, bukan hasil deterministik.

3.5.2 Penanganan epistasis dan interaksi kompleks terbatas

Teori skema paling informatif untuk pola orde-rendah dan panjang-pendefinisian pendek. Ketika fitness muncul dari interaksi orde-tinggi (epistasis kuat) atau dari dependensi terdistribusi yang kompleks antar lokus, hitungan skema mengaburkan struktur relevan dan menawarkan daya prediksi yang terbatas.

3.5.3 Pembatasan pada alfabet sederhana dan enkoding panjang tetap

Analisis skema klasik mengasumsikan alfabet biner dan string panjang tetap. Perlu reformulasi hati-hati untuk alfabet yang lebih kaya, genom ber- panjang variabel, atau enkoding tidak langsung; penerapan intuisi berbasis biner secara naif dapat menyesatkan.

3.5.4 Kurangnya spesifikasi preskriptif

Walaupun teori skema menjelaskan mengapa blok-bangunan pendek yang fit berguna, ia tidak memberikan algoritma preskriptif umum untuk menemukan representasi atau operator terbaik untuk masalah sewenang-wenang. Metode modern—pembelajaran linkage, EDAs, dan pendekatan pemodelan probabilistik—secara eksplisit mencoba mempelajari dan memanfaatkan struktur masalah yang tidak dapat diungkap hanya oleh hitungan skema.

3.5.5 Protokol empiris yang direkomendasikan

Saat menggunakan penalaran berbasis skema untuk membimbing desain algoritma, ikuti protokol empiris yang mengurangi risiko kesimpulan yang keliru:

- Jalankan beberapa percobaan independen dan laporkan varians, bukan hanya rata-rata performa.

- Gunakan studi ablation terkontrol untuk mengukur efek pilihan representasi dan operator terhadap sampling dan pelestarian skema kandidat.
- Jika memungkinkan, visualisasikan lintasan frekuensi alel dan plot unitation per-blok untuk mendiagnosis kapan dan di mana skema berguna hilang atau dipertahankan.
- Bandingkan dengan baseline pemodelan (EDAs sederhana, GA peka-linkage) untuk menilai apakah rekombinasi buta cukup untuk masalah target.

3.6 Teorema No Free Lunch

Menyatakan bahwa tidak ada algoritma yang unggul di seluruh kemungkinan masalah.

Chapter 4
Genetic Algorithm Encoding

4.1 Introduction to Encoding

Encoding (also called representation) formalises how candidate solutions are described for a genetic algorithm (GA). Let G denote the discrete set of genotypes (the representation space) and P denote the set of phenotypes (the solution space). Encoding is the specification of a mapping

$$\phi : G \to P,$$

that assigns to each genotype a phenotype that can be evaluated by a fitness function. In practice G is usually a finite or countable combinatorial space (for example, bit-strings, vectors of integers, permutations, trees or real-valued vectors) and P is the domain of problem solutions (for example, real vectors, schedules, tours, or programs).

Two aspects of encoding must be distinguished: (i) the representational language used to construct genotypes (bits, integers, reals, nodes in a tree, etc.), and (ii) the genotype–phenotype mapping ϕ. The search performed by a GA operates in G, while fitness and problem constraints are defined on P; therefore the properties of ϕ crucially determine how variation in genotype space translates to meaningful changes in solution quality.

Well-chosen encodings expose structure that the search procedures can exploit, reduce the incidence of infeasible solutions, and control representational redundancy and epistasis. Poor encodings can render local improvements invisible to variation, produce pathological fitness landscapes, or require expensive repair procedures. In later sections we discuss concrete encoding families (binary, gray, real-valued, permutation, tree) and the practical implications they have for representation design and algorithm performance.

4.2 Requirements for Good Encoding

When designing an encoding and its associated mapping $\phi : G \to P$, it is useful to state desiderata precisely. The following properties capture core representational requirements and trade-offs; they guide the selection or construction of encodings for a given problem.

4.2.1 Completeness

Completeness requires that the encoding be able to express every feasible phenotype of interest: formally, the image of ϕ should cover the feasible region $F \subseteq P$ of solutions, i.e. $\phi(G) \supseteq F$. If completeness fails then some valid solutions are unreachable by the GA, which introduces representational bias and can prevent the algorithm from finding optimal solutions that lie outside $\phi(G)$.

In practice completeness is balanced against representational compactness: a fully complete encoding may be large or inefficient, whereas a restricted encoding can greatly simplify search if it excludes uninteresting parts of P. Designers should explicitly state which subset of P must be reachable and ensure $\phi(G)$ contains it.

4.2.2 Soundness

Soundness (also called validity) stipulates that every genotype should map to a well-defined, constraint-satisfying phenotype: $\forall g \in G,\ \phi(g) \in P_{valid}$. Sound encodings avoid or minimise the production of infeasible solutions so that fitness evaluations are meaningful without costly repair. When strict soundness is impossible or impractical, designers may allow infeasible genotypes but must provide an efficient, well-defined decoding and repair strategy and ensure the search can still progress.

Soundness and completeness are orthogonal: an encoding can be sound but incomplete (every genotype valid, but not all phenotypes representable), or complete but unsound (all phenotypes representable but many genotypes invalid) depending on G and ϕ.

4.2.3 Non-redundancy

Non-redundancy means reducing (or eliminating) multiple distinct genotypes that map to the same phenotype. Formally, one prefers ϕ to be injective on the set of representationally relevant genotypes. Redundancy (many-to-one mapping) increases the effective search volume and can bias sampling: some phenotypes may be over-represented in G, making them more likely to be sampled even if they are not superior.

However, redundancy is sometimes deliberately introduced for robustness (e.g. neutral networks that allow neutral drift) or to simplify representation. When redundancy is present, it should be understood and controlled: quantify the degree of redundancy and consider its interaction with search dynamics and variation.

4.2.4 Locality

Locality formalises the intuition that small genotypic changes should produce small phenotypic changes. Let d_G and d_P be distance measures on G and P respectively (e.g. Hamming distance on bit-strings, Euclidean distance on real vectors). High locality means that

$$d_G(g_1, g_2) \text{ is small } \Rightarrow\ d_P(\phi(g_1), \phi(g_2)) \text{ is small.}$$

Locality is important because common variation procedures make small changes in G; if these do not correspond to small, correlated changes in P the search becomes effectively random and building-block recombination fails. Encoding choices such as Gray coding for integers or real-valued representations aim to improve locality.

Locality cannot always be achieved with other desirable properties; for example, injective, compact encodings with perfect locality may not exist for some combinatorial domains. Designers should therefore prioritise which properties matter most for the problem and for the procedures they plan to use.

4.2.5 Additional Practical Requirements

Beyond the four formal properties above, useful encodings should also satisfy several pragmatic constraints:

- **Operator Closure:** Variation procedures should, as much as possible, produce genotypes within a region of G that decodes to feasible or easily repaired phenotypes.
- **Computational Efficiency:** Decoding ϕ and any repair procedures should be computationally inexpensive relative to fitness evaluation.

- **Scalability:** The encoding should scale gracefully with problem size; representation length should not grow superlinearly without justification.
- **Low Epistasis:** The representation should aim to minimise destructive interactions between genes (epistasis) so that beneficial building blocks can be recombined reliably.
- **Interpretability and Prior Knowledge:** When available, incorporate problem-specific structure (symmetries, invariants, constraints) to simplify search and reduce unnecessary degrees of freedom.

Designing an encoding is therefore a matter of formal requirements, procedure compatibility, and empirical validation. Later sections examine common encoding families and trade-offs, and discuss practical choices that respect the desiderata above.

4.3 Binary Encoding

Binary encoding represents genotypes as fixed-length vectors over the binary alphabet: $G = \{0, 1\}^l$. A genotype $g = (b_{l-1}, \ldots, b_0)$ is commonly interpreted as an unsigned integer

$$\text{bin}(g) = \sum_{i=0}^{l-1} b_i 2^i,$$

which is then mapped to a phenotype by an affine decoding when the phenotype is numeric. For a real-valued variable $x \in [x_{\min}, x_{\max}]$ the usual decoding is

$$x = x_{\min} + \frac{\text{bin}(g)}{2^l - 1}(x_{\max} - x_{\min}). \tag{4.1}$$

This formula makes the representational resolution explicit: the quantisation step is

$$\Delta = \frac{x_{\max} - x_{\min}}{2^l - 1},$$

so choosing l trades off precision against search dimensionality and behavioural properties of variation procedures.

Binary encodings are attractive because they are compact and simple to manipulate with bitwise operators. Schema theory and many early theoretical results were developed for binary representations, which aids theoretical reasoning about convergence and building-block propagation [25, 21].

However, binary encodings also introduce specific problems that must be addressed in practice:

- **Hamming cliffs and locality:** Adjacent numeric values can differ in many bits under standard binary positional encodings, breaking locality. Gray codes are a common remedy when preserving adjacency is important.
- **Precision versus length:** High precision requires long bit-strings, which increases the search space exponentially and can make positional recombination disruptive.
- **Epistasis:** Bit positions may interact non-linearly with respect to phenotype quality; correlated bits reduce the effectiveness of simple recombination.

Practical recommendations for binary encodings:

- Choose length l from the desired resolution Δ and range $[x_{\min}, x_{\max}]$ using $2^l - 1 \geq (x_{\max} - x_{\min})/\Delta$.
- If adjacency matters, consider Gray coding for integer variables and convert to binary only for procedures that work on bitstrings.
- Use procedure choices that respect gene boundaries for multi-variable concatenations (e.g. align recombination points to variable boundaries when appropriate).
- Tune per-bit modification rates as a starting heuristic; decrease when using local search or strong selective dynamics.

4.4 Overview of Encoding Types

Encodings can be organised according to the structure of G and the intended phenotype domain P. Below we summarise principal families and their canonical use-cases, with practical guidance for representation design and common pitfalls.

Taxonomy and mapping to problem classes

- **Binary (bit-strings):** $G = \{0,1\}^l$. Good for combinatorial choices and when schema analysis is desired. Use Gray code or problem-specific bit ordering to improve locality for numeric phenotypes.
- **Integer (value) encodings:** Vectors of integers; natural for count and allocation problems. Use integer-aware variation procedures (random reset, creep) and discrete recombination methods.
- **Real-valued encodings:** Continuous vectors $\mathbb{R}^n$. Preferable for continuous optimisation; supports arithmetic combination and BLX-α style recombination, stochastic perturbations, and gradient-informed hybrids.
- **Permutation encodings:** Represent orderings (TSP, scheduling). Require specialised variation procedures (e.g. order-preserving transforms, mapping-based procedures, local reordering moves) that preserve permutation feasibility.
- **Tree and graph encodings:** Variable-size structures used in genetic programming, evolving expressions, or circuit topologies. Use subtree exchange and constrained growth controls to avoid bloat.
- **Indirect / developmental encodings:** Genotypes specify construction rules or grammars that generate phenotypes; useful when compact genotypes should produce structured phenotypes (e.g. neural architectures, L-systems).

Choosing an encoding

Select an encoding by matching the problem's combinatorial structure, constraint set, and desired procedure toolkit. Key questions:

- Does the problem require an ordering, a multiset, or real-valued parameters? Choose permutation, integer/multiset, or real encodings respectively.
- Are feasibility constraints hard (must be satisfied) or soft (violations penalised)? For hard constraints prefer sound encodings or constructive decoders; for soft constraints penalisation may be acceptable.
- Is locality important for effective recombination? If so, prefer encodings (or transformations, e.g. Gray) that increase correlation between small genotypic and phenotypic changes.
- Will procedures be custom or standard? Use encodings that keep procedure implementation simple unless domain structure mandates otherwise.

Operator compatibility and empirical validation

An encoding is only useful if paired with procedures that preserve useful structure. After selecting an encoding, design or choose variation procedures that maintain feasibility, limit destructive epistasis, and respect meaningful gene boundaries. Finally, validate encoding choices empirically: compare performance across a small benchmark (different encodings, procedure sets, and variation rates) and select the combination that gives robust progress on representative instances.

The following sections give concrete representational examples and references for the main encoding families discussed here.

4.5 Real-valued Encoding

Real-valued encoding represents individuals as vectors in $\mathbb{R}^n$, i.e. $\mathbf{x} = (x_1, \ldots, x_n)$ with each coordinate taking values on a continuous domain. This direct representation is the natural choice for continuous optimisation problems and for parameter tuning tasks where the phenotype is inherently numeric. By operating in a continuous space, real-valued encodings avoid the quantisation artefacts of fixed-length binary encodings and allow variation operators to express arbitrarily small adjustments to candidate solutions (subject to floating-point precision limits) [6, 30].

The principal practical advantage of real-valued encodings is operator compatibility: arithmetic recombination (weighted averages), BLX-α style interval recombination, simulated binary crossover (SBX) and Gaussian or Cauchy perturbation mutations all act naturally on real vectors and can be designed to respect bounds or known structure. These operators produce offspring that lie in the convex hull (or a controlled extension) of the parents, which typically yields smoother search trajectories and better exploitation of local gradients in the fitness landscape. Evolution Strategies (ES) and many modern continuous optimisers exploit these properties by combining self-adaptive step-size control with recombination to navigate rugged but differentiable landscapes efficiently [6].

There are, however, theoretical and practical trade-offs. Classical schema arguments developed for binary representations do not carry over directly to continuous encodings:

building-block notions must be reformulated in terms of regions of $\mathbb{R}^n$ and operator-induced correlations between coordinates. Real encodings also place greater emphasis on algorithmic choices for step-size control and constraint handling — poor mutation scales or unbounded recombination can lead to slow progress or numerical instability. Consequently, practitioners must tune or adapt mutation magnitudes (fixed schedules, self-adaptation, or covariance matrix adaptation) and choose recombination parameters that match problem smoothness and scale.

From an implementation viewpoint, several pragmatic recommendations improve robustness and performance. Always normalise or scale variables to comparable ranges before applying generic operators; this prevents single coordinates from dominating recombination statistics and simplifies parameter transfer between problems. Use bounded operators or projection schemes when constraints are present, and prefer adaptive mutation strategies (e.g. log-normal step-size adaptation or CMA-style covariance updates) when the search landscape exhibits anisotropy. When local gradients are available or can be approximated cheaply, hybridising evolutionary updates with gradient-based refinement often accelerates convergence while preserving global exploration.

Finally, like any encoding choice, real-valued representations should be validated empirically against alternatives. For many smooth, low-to-moderate dimensional continuous problems they substantially outperform binary encodings in both convergence speed and final solution quality; for highly multimodal or combinatorial problems a real-valued parameterisation may be inappropriate. We therefore recommend starting with a simple real-valued operator set (arithmetic/BLX recombination and Gaussian mutation with a tuned standard deviation), run small factorial experiments to select adaptation mechanisms, and escalate to more sophisticated adaptations (self-adaptive step sizes, CMA) when warranted by problem scale or observed search behaviour.

4.6 Integer Encoding

Integer encoding represents solutions whose variables take discrete integer values. Formally an individual is a vector $\mathbf{x} = (x_1, \ldots, x_n)$ with each coordinate $x_i \in \mathbb{Z}$ and, in practice, bounded to a finite domain $[a_i, b_i] \cap \mathbb{Z}$. This representation is appropriate for allocation problems, counts, and many combinatorial substructures (for example quantities in knapsack-like models, resource allocations, and discretised control parameters). The discrete nature of the variables changes the character of the search: neighbourhoods are naturally defined by integer steps, and the search landscape is inherently non-continuous and often non-convex.

Operators for integer encodings must respect integrality and any problem-specific bounds or feasibility constraints. Common mutation strategies include random-reset mutation (replace a coordinate with a uniformly sampled integer in its domain) and small-step or "creep" mutation (increment or decrement by a small integer drawn from a short-tailed distribution). Recombination can be performed directly in the integer domain (for example, discrete uniform crossover or coordinate-wise selection), or by temporarily lifting values to a continuous surrogate (arithmetic recombination followed by rounding) when an operator that benefits from averaging is desirable. When using surrogate continuous recombination, stochastic rounding or bias-corrected rounding helps reduce systematic rounding artefacts.

Integer encodings present trade-offs compared to real-valued representations. Because the domain is discrete, many analytical assumptions (e.g. smooth gradients or continuous

convexity) do not apply, and standard continuous step-size adaptation mechanisms require adaptation to discrete step scales. On the other hand, integer representations can encode feasibility directly, avoiding expensive repair procedures: e.g. representing quantities with integrality enforces natural constraints, and specialised discrete crossover/mutation operators can be designed to preserve feasibility or near-feasibility by construction.

From an algorithm design and implementation perspective several pragmatic recommendations improve robustness. First, exploit problem structure: if variables have small integer ranges prefer enumerative neighbourhood moves and small-step local search hybrids; if ranges are large, favour operators that explore broadly (random-reset, large-step proposals) combined with adaptive reduction in step magnitude. Second, enforce bounds and invariants in the decoder or by projection after variation rather than relying on implicit truncation; explicit constraint-aware operators are usually clearer and less error prone. Third, when mixing integer and continuous variables use mixed-integer operators or decoupled schedules so that each variable type receives appropriately scaled variation.

Finally, validate encoding choices empirically. Compare a direct integer encoding to alternatives (binary-encoded integers, real-valued surrogate with rounding) on small representative instances to measure convergence speed, robustness, and the cost of constraint handling. In many allocation or scheduling tasks a well-chosen integer representation plus tailored discrete operators outperforms generic continuous surrogates; however, for problems where fine-grained search behaviour is important, surrogate continuous strategies with careful rounding and step-size adaptation can be competitive. Use these empirical results to select mutation/recombination scales and to decide whether to hybridise the evolutionary loop with deterministic local search on integer neighbourhoods.

4.7 Permutation Encoding

Permutation encoding represents candidate solutions as permutations of a finite set of elements, i.e. the genotype space is the set of bijections on {1,. . . ,n}. The genotype–phenotype mapping ϕ is usually the identity map: a permutation directly specifies an ordering that is interpreted by the problem-specific evaluator (for example a tour in the travelling salesman problem, a job sequence in single-machine scheduling, or an ordered list of tasks for a flow line). Because permutations inherently enforce ordering constraints, permutation encodings are sound for ordering problems and avoid many feasibility repairs required by naive encodings.

Although permutations are formally one-to-one with respect to orderings, practical representations often introduce equivalence classes and redundancies that must be recognised. A circular tour (as in symmetric TSP) admits rotational symmetry: cyclic shifts of a permutation represent the same tour and reflections may also be equivalent. Such symmetries do not change validity but affect sampling and selection probabilities; designers should either choose a canonical representative (fixing the first city) or use operators and fitness comparisons that account for the equivalence class to avoid representational bias.

Distance and locality in permutation spaces differ markedly from vector spaces. Hamming distance or simple positional metrics do not capture meaningful neighbourhood structure for order-based problems. Distances such as Kendall tau (number of pairwise disagreements), inversion distance, or edge-based metrics (number of differing adjacency relationships) better reflect the kinds of small, interpretable changes that preserve problem structure. Operator design should therefore be guided by which aspects of a permutation

constitute useful building blocks for the problem — position-based blocks, adjacency/edge blocks, or precedence relations — because different operators preserve different structures.

Variation operators for permutation encodings must preserve feasibility (i.e. produce valid permutations) and ideally respect the chosen notion of locality. Typical mutation moves are swap, insert (take-one-and-insert-at-another-position), and inversion/reversal of a subsequence; these have clear interpretations as small, local reorders. Recombination operators are designed to combine parent orderings while maintaining permutation validity: examples include partially mapped crossover (PMX), order crossover (OX), cycle crossover (CX), and edge recombination. Each emphasises different preserved structures (position, order, or adjacency) and the choice should match problem-specific building blocks (for instance, edge-based recombiners are natural for TSP where edges matter more than absolute positions).

An alternative is to use indirect encodings and constructive decoders when constraints or constructive heuristics are important. Priority or random-key encodings map real-valued keys to a permutation via a stable sorting decoder; constructive decoders build feasible schedules or tours greedily from a genotype that encodes preferences. Indirect encodings can drastically reduce design complexity by separating the genetic search from feasibility enforcement and can incorporate domain heuristics directly into the decoder, but they shift the design burden to the decoder and may obscure locality properties of the genetic operators.

Practical recommendations: initialise populations using a mixture of random permutations and problem-specific heuristics to seed useful structure; measure diversity with permutation-aware metrics (Kendall tau or edge overlap) rather than Hamming distance; prefer operators that preserve the notion of building blocks relevant to your problem; and combine global permutation-based search with local optimisation (e.g. 2-opt or 3-opt for TSP, or specialised neighbourhood search for scheduling) to exploit fine-grained improvements. When symmetries exist, use canonicalisation or equivalence-aware evaluation to avoid bias. Finally, validate choices empirically on representative instances, since operator effectiveness is strongly problem-dependent in permutation spaces.

4.8 Tree Encoding

Tree encoding represents genotypes as labelled, rooted trees whose nodes carry symbols drawn from one or more alphabets (for example function/operator symbols for internal nodes and terminal symbols for leaves). The phenotype is obtained by interpreting the tree according to problem semantics: in genetic programming the tree denotes an expression or program, in syntactic optimisation it denotes a parse tree, and in hierarchical design it denotes a composition of components. Formally the genotype space is the set of finite ordered trees over a ranked alphabet, and the decoder ϕ is the evaluation or instantiation function that maps a tree to the problem-specific object in P.

Tree encodings introduce representational choices that strongly affect operator behaviour and search dynamics. One must decide on the node alphabets (typed or untyped), arity constraints (fixed or variable arity), and linearisation for storage (pointer structures, bracketed strings, prefix/postfix notations, or explicit child lists). Typed (strongly-typed) trees enforce syntactic constraints at the representation level, preventing many invalid offspring and reducing the need for repair; untyped trees are more flexible but often require additional feasibility checks or decoders. Representation impacts locality: small subtree

replacements may induce large semantic changes when node semantics are non-linear or context-sensitive.

A central concern with tree encodings is bloat — unbounded growth in tree size without commensurate fitness improvement. Bloat arises from neutral or weakly selective regions where larger trees are not penalised, and it degrades performance by increasing evaluation cost and reducing effective population variability. Common countermeasures include static limits on depth/size, parsimony pressure (explicit size or complexity penalties in the fitness), and operator-aware controls (limiting offspring size in crossover and mutation). When using growth controls, balance is required: overly aggressive pruning can eliminate useful structural variation, while permissive settings lead to resource exhaustion.

Variation operators for trees must preserve tree well-formedness. Standard operators include subtree crossover (swap subtrees between parents), point mutation (replace a node or small subtree with a randomly generated subtree), and hoist mutation (replace a tree by one of its subtrees to reduce size). There are also context-preserving operators designed for typed languages or grammars (constrained subtree exchange, grammar-guided mutation) that maintain syntactic correctness by construction. Operator design should align with the semantics of the node alphabets: for expression trees, promoting commutative/associative-aware recombination or algebraic simplifications can increase meaningful offspring generation.

Indirect and grammar-based encodings are especially useful when the phenotype must satisfy rich syntactic or semantic constraints. In grammatical evolution and grammar-guided GP the genotype typically encodes a derivation or a sequence of production choices, and a deterministic decoder maps this to a tree that is guaranteed syntactically valid. These indirect encodings can produce compact genotypes and enable incorporation of domain knowledge, but they may obscure the locality properties of operators and require careful decoder design to avoid biased sampling of the phenotype space.

Practical recommendations: enforce feasibility early using typing or grammar constraints when the domain requires syntactic correctness; combine global tree-based search with local simplification passes (constant folding, algebraic reductions) to improve evaluation efficiency; use mixed initialisation strategies (ramped half-and-half, grow/full) to seed diverse tree sizes and shapes; and adopt explicit complexity control (parsimony pressure, depth limits, or adaptive operator bias) to manage bloat. Finally, validate operator choices empirically on representative instances and instrument tree-size, depth, and evaluation cost during experiments to detect emergent bloat or pathological behaviours.

Chapter 5
Selection Methods in Genetic Algorithms

5.1 Introduction to Selection

Selection is the mechanism within a genetic algorithm that determines which individuals from a population are chosen to contribute genetic material to the next generation. At its core, selection converts fitness information into reproductive opportunities: individuals with relatively higher fitness are given greater chances to produce offspring, thereby biasing the search process toward promising regions of the solution space. This bias must be managed carefully so that the algorithm both exploits high-quality solutions and continues to explore diverse alternatives.

An essential concept associated with selection is selection pressure, which quantifies the degree to which better individuals are favored. High selection pressure accelerates convergence by amplifying the reproductive advantage of the fittest individuals, but it increases the risk of premature convergence where the population loses diversity and becomes trapped in suboptimal regions. Low selection pressure preserves diversity and encourages exploration, yet may slow the algorithm's progress toward improved solutions. Practical algorithm design therefore requires balancing these competing effects, often by tuning selection parameters or combining selection schemes with diversity-preserving mechanisms.

Selection operators come in several families, each offering different trade-offs between simplicity, selection pressure control, and sensitivity to fitness scaling. Common approaches include fitness-proportionate methods (e.g., roulette wheel and stochastic universal sampling), rank-based schemes that ensure controlled and scale-independent pressure, tournament selection which provides an adjustable and efficient means of imposing pressure, and truncation or elitist strategies that deterministically preserve top individuals. Later sections of this chapter examine these methods in detail, including their algorithms, statistical properties, and practical advantages or drawbacks.

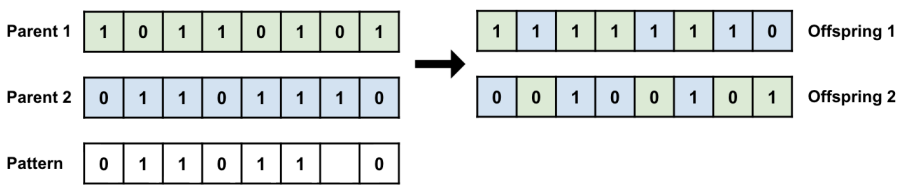

Figure 5.1: Basic selection process in Genetic Algorithms

5.2 Selection Pressure

Selection pressure quantifies how strongly a selection mechanism favors individuals with higher fitness when producing the next generation. Intuitively, it measures the expected reproductive advantage of good solutions relative to the population average. Selection pressure can be formalized in several ways; common operational measures include selection intensity (the standardized difference between parent and population means) and takeover time (the number of generations required for the best individual to dominate under repeated selection). These measures allow practitioners to compare different selection operators and parameterizations in a principled manner.

The magnitude of selection pressure has direct and predictable effects on search dynamics. Strong pressure accelerates the propagation of beneficial alleles and shortens the time to apparent convergence, which is useful when the fitness landscape is smooth and unimodal. However, excessive pressure reduces genetic diversity and increases the risk of premature convergence to local optima. Conversely, weak pressure preserves diversity and supports broader exploration of the search space but slows progress toward high-fitness regions. Therefore, the choice of selection operator and its parameters should be informed by the problem's modality, population size, and the available number of generations.

Practical controls for selection pressure include algorithmic choices (e.g., tournament size, ranking slope, truncation fraction), fitness scaling techniques (e.g., linear or sigma scaling, Boltzmann selection), and hybrid strategies that adapt pressure during the run (e.g., start with low pressure for exploration and increase pressure for exploitation). Monitoring selection-related statistics — such as mean and variance of fitness, diversity measures (e.g., average Hamming distance in binary encodings), and takeover time estimates — provides actionable feedback for tuning. In applied settings, a common heuristic is to begin with moderate pressure (e.g., small tournament sizes, conservative ranking parameters) and adjust based on empirical performance and observed loss of diversity.

5.3 Fitness Proportionate Selection (FPS)

The Genetic Algorithm developed by Holland uses Fitness Proportionate Selection (FPS) [25, 21], where the expected value of an individual (i.e., the expected number of times that individual will be selected for reproduction) is calculated as that individual's fitness divided by the population's average fitness.

In this method, each individual can be selected as a parent with a probability proportional to its fitness value. Therefore, individuals with higher fitness have greater opportunities to reproduce and spread their characteristics to the next generation. Thus, this method provides selection pressure on fitter individuals in the population, thus driving evolution toward better individuals over time.

5.3.1 Roulette Wheel Selection

Also known as fitness proportionate selection, individuals are selected with probability proportional to their fitness [21, 3, 4].

The simplest selection schema is roulette-wheel selection, also called stochastic sampling with replacement. This is a stochastic algorithm and involves the following technique:

Individuals are mapped to contiguous segments on a line, where the size of each segment is equal to that individual's fitness value. A random number is generated, and the individual whose segment spans that random number is selected. This process is repeated until the desired number of individuals is reached, called the mating population. This technique is analogous to a roulette wheel, where each slice is proportional in size to the fitness value.

Number of Individual	Fitness Value	Selection Probability	Interval
1	2.0	0.18	[0.00, 0.18]
2	1.8	0.16	[0.18, 0.34]
3	1.6	0.15	[0.34, 0.49]
4	1.4	0.13	[0.49, 0.62]
5	1.2	0.11	[0.62, 0.73]
6	1.0	0.09	[0.73, 0.82]
7	0.8	0.07	[0.82, 0.89]
8	0.6	0.06	[0.89, 0.95]
9	0.4	0.03	[0.95, 0.98]
10	0.2	0.02	[0.98, 1.00]
11	0.0	0.0	–

Table 5.1: Selection probability and fitness value (from Buku Ajar)

Table 5.1 shows the selection probabilities for 11 individuals, with linear ranking with selective pressure of 2, along with their fitness values. Individual 1 is the individual with the highest fitness and occupies the largest interval, while individual 10 as the individual with the second lowest fitness has the smallest interval on the line. Individual 11, with the lowest fitness, has fitness value = 0 and gets no chance for reproduction.

To select the mating population, a number of uniformly distributed random numbers (uniformly distributed between 0.0 and 1.0) are generated independently.

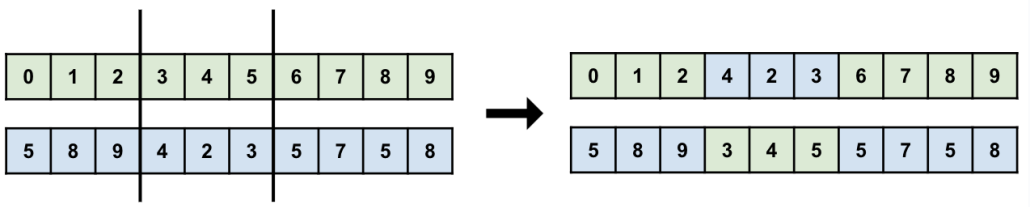

Figure 5.2: Roulette-wheel selection process with sample trials

The disadvantage of roulette-wheel selection is that although it provides zero bias, it does not guarantee minimum spread.

Algorithm

Selection Probability

The probability of selecting individual i is:

$$P_i = \frac{f_i}{\sum_{j=1}^{N} f_j} \tag{5.1}$$

Algorithm 1 Roulette Wheel Selection

Calculate total fitness: $F = \sum_{i=1}^{N} f_i$
Generate random number: $r \sim U[0, F]$
Set cumulative fitness: $sum = 0$
for $i = 1$ to N **do**
  $sum = sum + f_i$
  if $sum \geq r$ **then**
    Select individual i
    break
  end if
end for

Example

Individual	Fitness	Probability	Cumulative
1	10	0.25	0.25
2	20	0.50	0.75
3	5	0.125	0.875
4	5	0.125	1.0
Total	40	1.0	

Table 5.2: Roulette Wheel Selection Example

If random number $r = 0.6$, individual 2 is selected.

Advantages

Roulette-wheel selection is straightforward to implement and directly realizes fitness-proportionate sampling, so higher-fitness individuals are naturally more likely to contribute genetic material. Its probabilistic nature ensures that every individual retains a nonzero chance of selection, which helps maintain some exploration even when fitter individuals dominate.

Disadvantages

Roulette-wheel selection can suffer when fitness values are highly skewed: individuals with very large fitness may dominate and drive premature convergence, while very similar fitness values lead to weak selection pressure. Practical use therefore often requires fitness scaling or normalization, and care must be taken with negative or non-comparable fitness measures.

5.3.2 Stochastic Universal Sampling (SUS)

Improved version of roulette wheel selection that reduces variance [8].

Baker's SUS

Stochastic Universal Sampling (SUS) provides zero bias and minimum spread [8]. Individuals are mapped to contiguous segments on a line, where the size of each segment equals its fitness value, exactly as in roulette-wheel selection. In this method, equally spaced pointers are placed on the line equal to the number of individuals to be selected.

Let NPointer be the number of individuals to be selected, then the distance between pointers is $1/N_{Pointer}$, and the position of the first pointer is determined by a random number generated in the range $[0, 1/N_{Pointer}]$.

For example, to select 6 individuals, the distance between pointers is $1/6 = 0.167$.

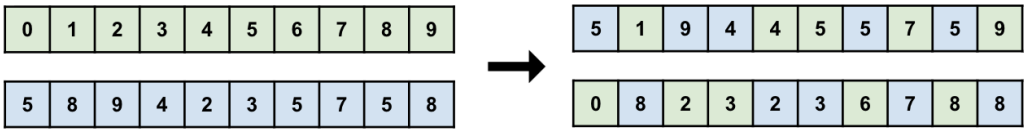

Figure 5.3: Stochastic universal sampling with equally spaced pointers

Stochastic universal sampling ensures offspring selection that is closer to the expected values compared to roulette-wheel selection.

Algorithm

Algorithm 2 Stochastic Universal Sampling

Calculate total fitness: $F = \sum_{i=1}^{N} f_i$
Calculate pointer distance: $distance = F/N$
Generate random start: $start \sim U[0, distance]$
Create pointers: $pointer_i = start + i \times distance$ for $i = 0, 1, \ldots, N-1$
for each pointer **do**
  Select individual using roulette wheel logic
end for

Advantages over Roulette Wheel

Stochastic Universal Sampling reduces sampling variance relative to independent roulette draws by using evenly spaced pointers; this produces selection counts that are closer to their expected values and yields a more uniform coverage of the population. As a consequence, SUS better preserves the expected representation of individuals across repeated samplings, improving stability of the selection operator.

5.4 Rank-based Selection

Rank-based selection assigns selection probabilities based on fitness rank rather than raw fitness values [22, 5].

5.4.1 Overview

Ranked-Based Selection introduces a different approach to selection in Genetic Algorithms. Instead of directly using fitness values to determine selection probability, individuals in the population are first sorted (ranked) based on their fitness values, then each individual is assigned a rank. The selection probability is then calculated based on that rank, not the actual fitness value.

This rank-based approach helps reduce problems associated with direct fitness-based selection, such as premature convergence and domination by a few very fit individuals in the early stages of the optimization process. By assigning ranks and using them for selection, Ranked-Based Selection provides more balanced and controlled selection pressure, allowing better exploration of the search space and maintaining diversity in the population.

Rankings are typically assigned linearly or exponentially, where the best individual receives the highest rank and the worst individual receives the lowest rank. Selection probability is then calculated based on that ranking using a predetermined formula or mapping function. This mapping function can be adjusted to control selection pressure, where higher pressure will favor individuals with the highest ranks, while lower pressure provides a more even distribution of selection probabilities.

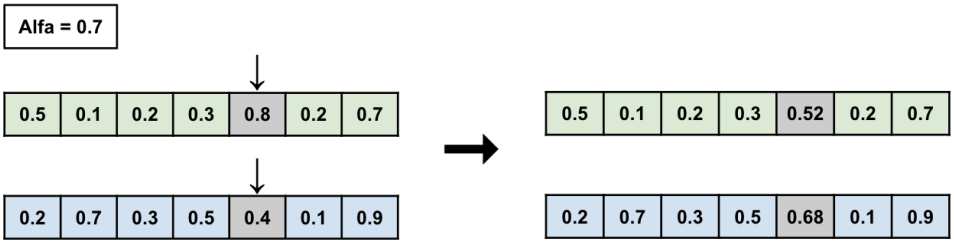

Figure 5.4: How the situation changes after converting fitness to order number (rank)

5.4.2 Linear Ranking

$$P_i = \frac{1}{N}\left[\eta^- + (\eta^+ - \eta^-)\frac{rank_i - 1}{N - 1}\right] \tag{5.2}$$

where:

- $rank_i$ is the rank of individual i (1 = worst, N = best)
- η^+ is the expected number of copies for best individual
- η^- is the expected number of copies for worst individual
- $\eta^+ + \eta^- = 2$ (to maintain population size)
- Typically: $\eta^+ = 2.0$, $\eta^- = 0.0$

5.4.3 Exponential Ranking

$$P_i = \frac{1 - e^{-rank_i}}{c} \tag{5.3}$$

where c is a normalization constant ensuring $\sum P_i = 1$.

5.4.4 Advantages of Rank Selection

By basing probabilities on rank rather than raw fitness, rank-based selection enforces a predictable and bounded selection pressure that is insensitive to the scale or distribution of fitness values. This makes it robust to outliers and negative fitness values and helps prevent a few extreme individuals from dominating the population.

5.4.5 Disadvantages

Rank-based schemes discard the magnitude information contained in fitness differences, which can slow convergence when those magnitudes are informative; additionally, they require sorting the population each generation, introducing an O(N log N) cost and some implementation complexity compared to simpler, linear-time samplers.

5.5 Tournament Selection

Tournament selection randomly selects k individuals and chooses the best among them [21, 7].

5.5.1 Overview

Tournament selection is a strong and widely used selection mechanism in Genetic Algorithms because it can maintain a balance between diversity maintenance and selective pressure [7]. Unlike roulette-wheel selection, which directly depends on an individual's fitness relative to the total population fitness, tournament selection works by holding "tournaments" among subsets of individuals, and the winner of each tournament is selected for reproduction.

The main concept of tournament selection is quite simple: instead of considering the entire population at once, a subset of individuals is randomly selected to compete with each other. The individual with the highest fitness in that "tournament" is then selected. This process is repeated until the desired number of individuals for reproduction is reached.

This method has several advantages: tournament selection maintains diversity because individuals with low fitness still have the opportunity to participate in tournaments. Additionally, this method allows selective pressure to be adjusted by setting the tournament size.

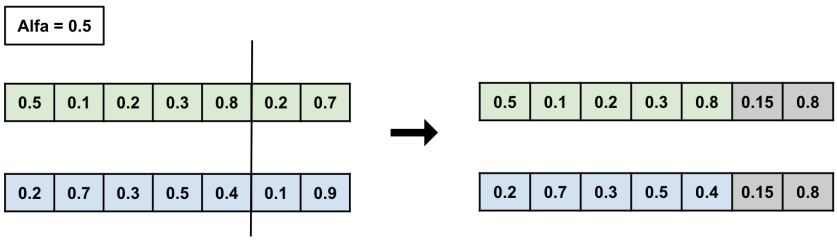

Figure 5.5: Tournament selection mechanism

5.5.2 Tournament Selection Mechanism

1. Determine tournament size (k), i.e., the number of individuals participating in each tournament.
2. Randomly select k individuals from the population.
3. Compare the fitness values of these individuals and select the individual with the highest fitness as the winner.
4. Add the winner to the mating pool.
5. Repeat steps 2–4 until the desired number of individuals is reached.

5.5.3 Binary Tournament

Most common form with $k = 2$.

Algorithm 3 Binary Tournament Selection

Randomly select individual i
Randomly select individual j (where $j \neq i$)
if $f_i > f_j$ **then**
  Select individual i
else
  Select individual j
end if

5.5.4 k-Tournament Selection

Algorithm 4 k-Tournament Selection

Create empty tournament set T
for $i = 1$ to k **do**
  Randomly select individual and add to T
end for
Select best individual from T

5.5.5 Tournament Size Effects

- $k = 1$: Random selection (no pressure)
- Small k: Low selection pressure
- Large k: High selection pressure
- $k = N$: Always selects best individual

5.5.6 Selection Probability

For individual with rank r out of N (1 = worst, N = best):

$$P_i = \frac{1}{N}\binom{N}{k}\sum_{j=0}^{r-1}\binom{j}{k-1}\binom{N-j-1}{0} \tag{5.4}$$

For binary tournament ($k = 2$):

$$P_i = \frac{2r-1}{N^2} \tag{5.5}$$

5.5.7 Advantages

Tournament selection is easy to implement, requires only local comparisons (no global fitness normalization), and provides a direct knob for selection pressure via the tournament size. Its independence from global statistics also makes it naturally parallelizable and tolerant of arbitrary fitness scales, including negative values.

5.5.8 Disadvantages

Tournament selection's behavior depends strongly on the chosen tournament size: large tournaments can impose very high pressure and reduce diversity, while very small tournaments approach random selection. The method can also repeatedly choose the same individual in multiple tournaments, which—without complementary diversity mechanisms—may accelerate loss of genetic variety.

5.6 Truncation Selection

Truncation selection is a deterministic policy that retains only the top fraction of the population for reproduction: given a population of size λ, the best μ individuals (by fitness) are selected and used to produce the next generation. The central parameter is the selection ratio

$$\rho = \frac{\mu}{\lambda}, \tag{5.6}$$

which directly controls selection pressure — smaller values of ρ correspond to stronger pressure because fewer individuals are permitted to reproduce. Truncation is typically implemented by sorting individuals by fitness (complexity $O(\lambda \log \lambda)$) and slicing the top μ entries; the deterministic nature makes its effect on the population easy to analyze and predict.

The principal effect of truncation selection is strong and immediate directional pressure: good solutions rapidly dominate the gene pool, which can greatly speed convergence in smooth, unimodal landscapes. This same property is its main drawback in multimodal or deceptive landscapes, where aggressive truncation reduces genetic diversity and increases the likelihood of premature convergence to suboptimal peaks. To mitigate these risks, practitioners often choose moderate values of ρ (common heuristics place ρ between 0.1 and 0.5 depending on problem scale) or combine truncation with diversity-preserving measures such as fitness sharing, crowding, or occasional stochastic replacement.

In practice, truncation is well suited for exploitation phases of an evolutionary run or for algorithms that require deterministic selection behavior (for reproducibility or theoretical analysis). When using truncation, monitor diversity statistics (e.g., genotype variance or average pairwise distance) and consider adaptive schedules that relax truncation early in the run and tighten it later as the search focuses on refinement.

5.7 Boltzmann Selection

Boltzmann selection adapts the familiar Boltzmann (Gibbs) distribution from statistical mechanics to map fitness values to selection probabilities. Under this scheme each individual i is assigned selection probability

$$P_i = \frac{e^{f_i/T}}{\sum_{j=1}^{N} e^{f_j/T}}, \tag{5.7}$$

where f_i is the fitness of individual i and $T > 0$ is the temperature parameter that controls the degree of randomness in selection. When T is large the distribution approaches uniform sampling (promoting exploration); as $T \to 0$ the distribution concentrates on the best individuals (promoting exploitation). This explicit temperature control makes Boltzmann selection a natural mechanism for smoothly interpolating between exploration and exploitation during an evolutionary run.

Practical use of Boltzmann selection requires a temperature schedule $T(t)$ that varies with generation t. A common choice is exponential cooling, for example $T(t) = T_0\,\alpha^t$ with $0 < \alpha < 1$, but linear or problem-specific schedules may be preferable depending on landscape characteristics. The choice of initial temperature T_0 and the annealing rate determine how quickly selection pressure increases; poor choices can either leave the search unfocused for too long or force premature convergence.

Compared with simpler selection schemes, Boltzmann selection has two notable trade-offs. Its benefits are principled control of pressure and the ability to schedule a gradual transition from exploration to exploitation. Its costs are additional parameter tuning (temperature schedule) and slightly higher computational overhead due to exponentials and normalization. In practice, Boltzmann selection is most valuable when a controlled annealing strategy is desirable—e.g., when combining global exploration early with focused refinement later—or when fitness magnitudes vary widely and a temperature parameter offers robust scaling. When using Boltzmann selection, monitor fitness variance and population diversity, and be prepared to adjust the temperature schedule empirically for best results.

5.8 Elitist Selection

Elitist selection refers to selection policies that guarantee the survival of one or more top-ranked individuals from one generation to the next. The rationale is simple: stochastic variation in reproduction and replacement can accidentally discard the best solutions; preserving a small set of elites ensures that high-quality genetic material is never lost, which in turn makes measured, monotonic progress possible in many implementations.

There are two common variants. Pure elitism explicitly copies the best e individuals unchanged into the next generation; this is the most conservative approach and is typically

used with very small e (often $e = 1$). Elitist replacement is a softer variant in which after the usual selection and reproduction steps the worst individuals in the new population are replaced by the best individuals from the previous generation if those elites are strictly better. Both variants preserve improved solutions but differ in determinism and how aggressively they constrain the population.

¿The principal benefit of elitism is reliability: it prevents the accidental loss of the best-so-far solutions and often accelerates practical convergence by retaining proven building blocks. However, elitism also affects population diversity and exploration. If too many elites are preserved or elites are preserved for too long, the search can become overly exploitative, reducing the population's capacity to discover alternate peaks. Good practice is to keep the elite count small relative to population size (for instance, $e/\lambda \ll 0.1$) and, when necessary, combine elitism with explicit diversity-preserving techniques (e.g., occasional random immigrants, fitness sharing, or controlled mutation rates).

When using elitist strategies, monitor both the best fitness and diversity indicators (e.g., genotype variance, number of unique individuals). Consider adaptive policies that reduce elitism early to promote exploration or temporarily increase elitism late in a run for final refinement. These pragmatic controls help preserve the safety that elitism provides while mitigating its tendency to narrow the search prematurely.

5.9 Diversity-Preserving Selection

Diversity-preserving selection encompasses techniques intended to maintain useful genetic variation in the population so that evolution can continue to explore multiple promising regions of the search space. Maintaining diversity is particularly important for multimodal and deceptive problems where premature loss of variation can cause the run to converge to suboptimal peaks. The following paragraphs summarize commonly used mechanisms and practical guidance for their use.

One widespread approach is fitness sharing, which reduces the effective fitness of individuals that are close to many others in genotype or phenotype space. The shared fitness of individual i is computed as

$$f_i' = \frac{f_i}{\sum_{j=1}^{N} sh(d_{ij})}, \tag{5.8}$$

where d_{ij} is a distance between individuals i and j (Hamming distance for binary encodings or Euclidean distance for real-valued representations) and the sharing function is often defined as

$$sh(d) = \begin{cases} 1 - \left(\dfrac{d}{\sigma_{share}}\right)^{\alpha} & \text{if } d < \sigma_{share}, \\ 0 & \text{otherwise.} \end{cases} \tag{5.9}$$

The parameter σ_{share} defines the niche radius and α controls the decline of sharing; typical practice uses $\alpha = 1$ and selects σ_{share} by testing or domain knowledge. Fitness sharing encourages the population to occupy multiple niches and reduces the advantage of densely populated regions.

Crowding methods provide an alternative that directly controls replacement: offspring are preferentially compared and possibly replace similar individuals rather than random or worst ones. Deterministic crowding and probabilistic crowding are common variants; both aim to preserve local subpopulations by ensuring that newly created individuals

compete with genetically similar members, thereby preventing a single genotype from quickly sweeping the population.

Speciation, niching, and island models explicitly partition the population into subpopulations (species or islands) that evolve semi-independently, with occasional migration of individuals between groups. These structures preserve diversity by allowing different regions of the search space to be explored in parallel and are especially useful for problems with many well-separated optima. Practical design choices include migration rate, topology (ring, fully connected, etc.), and migration policy (best individuals, random migrants, or fitness-proportionate migrants).

When choosing and tuning diversity-preserving mechanisms, consider computational cost and measurement: fitness sharing requires $\mathrm{O}(N^2)$ pairwise distance computations unless approximations or clustering are used; crowding typically runs in $\mathrm{O}(N)$ per generation with careful bookkeeping; speciation and island models scale linearly per island but require configuration of migration parameters. Monitor population statistics (e.g., average pairwise distance, number of distinct genotypes, fitness variance) to detect excessive loss of diversity and adjust parameters dynamically (for example, increase mutation rate or relax selection pressure when diversity drops). Combining modest diversity-preserving methods with a well-calibrated selection pressure often yields the best practical results.

5.10 Multi-objective Selection

Multi-objective selection treats optimization problems where the quality of a solution is described by a vector of objectives rather than a single scalar. Let $\mathbf{f}(\mathbf{x}) = (f_1(\mathbf{x}), \ldots, f_m(\mathbf{x}))$ denote the objective vector to be maximized; the concepts below are straightforwardly adapted to minimization by sign reversal. Central to multi-objective selection is the notion of Pareto dominance: a solution $\mathbf{x}$ dominates $\mathbf{y}$ (written $\mathbf{x} \prec \mathbf{y}$) if

$$\forall k \in \{1, \ldots, m\} : \; f_k(\mathbf{x}) \geq f_k(\mathbf{y}), \quad \text{and} \quad \exists k : \; f_k(\mathbf{x}) > f_k(\mathbf{y}). \tag{5.10}$$

This partial order induces a front structure on the population: nondominated solutions form the Pareto front (rank 1), the nondominated set of the remainder forms rank 2, and so on. Non-dominated sorting partitions the population by repeatedly extracting the current nondominated set; the naive procedure has worst-case cost $\mathrm{O}(mN^2)$ for N individuals and m objectives, while optimized algorithms and data structures can offer empirical improvements for many practical sizes.

Selection in multi-objective evolutionary algorithms (MOEAs) must both promote convergence toward the Pareto front and preserve diversity along it. A widely used strategy, popularized by NSGA-II, performs selection in two stages: (1) apply non-dominated sorting to assign a rank to each individual, and (2) within the same rank prefer individuals that increase population spread using a crowding measure. The crowding distance for an individual is computed by summing normalized gaps between neighboring solutions for each objective after sorting by that objective; larger crowding distance indicates a less crowded region and is therefore preferred when ranks tie. Practically, parent or survivor selection can be implemented as a binary tournament that compares first by rank (lower is better) and then by crowding distance (higher is better), which yields a simple, effective rule that balances convergence and diversity.

Several practical considerations arise when applying multi-objective selection. For many objectives (the many-objective case, $m \gtrsim 5$), dominance relations become less dis-

criminating and alternative approaches—indicator-based methods (e.g., IBEA), decomposition techniques (e.g., MOEA/D), or reference-point strategies—often perform better. Computational cost is also important: while NSGA-II is efficient and robust for moderate N and m, indicator-based selection (hypervolume-based) can be costly for large fronts. Finally, parameter choices (population size, replacement policy, mating selection) affect both the algorithm's ability to approximate the Pareto front and the distribution of solutions; monitor convergence (e.g., IGD, hypervolume) and spread metrics, and consider hybridizing selection with archiving strategies or adaptive population sizing when sustained exploration of multiple trade-offs is required.

5.11 Selection Comparison

Method	Pressure	Diversity	Complexity	Scalability	Parameters
Roulette Wheel	Variable	Poor	O(N)	Poor	None
SUS	Variable	Good	O(N)	Poor	None
Rank Linear	Constant	Good	O(N log N)	Good	η^+, η^-
Tournament	Adjustable	Good	O(1)	Excellent	k
Truncation	High	Poor	O(N log N)	Good	μ/λ
Boltzmann	Adaptive	Excellent	O(N)	Good	$T(t)$

Table 5.3: Comparison of Selection Methods

The table above summarizes key practical properties of commonly used selection methods. It condenses four dimensions that guide method choice: the effective selection pressure (how strongly the operator favors high-fitness individuals), the operator's tendency to preserve or erode population diversity, the asymptotic computational complexity for a single generation, and how well the method scales with population size or parallel implementations. The final column lists the principal tuning parameters that practitioners must set or schedule.

Interpreting the table requires combining its quantitative entries with problem-specific considerations. Methods labelled "variable" pressure (roulette wheel and SUS) depend directly on the raw fitness distribution and so are sensitive to scaling and outliers; when fitness values are skewed these methods either collapse diversity (large gaps) or provide negligible pressure (small differences). Stochastic Universal Sampling reduces the sampling variance of roulette draws and therefore yields selection counts closer to expectation, but it does not by itself remove sensitivity to fitness scaling.

Rank-based linear selection deliberately discards raw magnitude information in favor of ordinal information, producing predictable and bounded pressure that is robust to arbitrary fitness scales and outliers. The trade-off is loss of useful signal when fitness magnitudes are meaningful, plus the O(N log N) cost of sorting each generation. Tournament selection provides an efficient, local-comparison mechanism whose pressure is adjusted directly by the tournament size k; it is simple, parallel-friendly, and insensitive to global normalization, which explains its widespread use in large-scale and distributed implementations.

Truncation selection is the most aggressive deterministic policy: keeping only the top fraction (μ/λ) applies very high pressure and rapidly concentrates the population, which can be desirable in unimodal problems or late-stage exploitation but harmful in multimodal or deceptive landscapes absent strong diversity-preservation. Boltzmann selection

offers an explicit, continuous control knob via the temperature schedule $T(t)$: when tuned well, it interpolates smoothly between exploration and exploitation and handles widely varying fitness magnitudes, at the cost of an additional scheduling parameter and the need to monitor annealing behavior.

From a practical standpoint, selection should rarely be chosen on a single criterion. For problems where fitness scaling is unreliable or unknown, prefer rank-based or tournament methods. For applications that demand reproducible, deterministic behavior or very fast convergence on a known unimodal problem, truncation (with small μ/λ) or elitist augmentation can be appropriate. When maintaining a diverse Pareto of solutions or exploring rugged landscapes, combine selection with explicit diversity mechanisms (fitness sharing, crowding, speciation) and keep selection pressure moderate. Finally, parameter choices (tournament size, ranking slope, truncation fraction, temperature schedule) should be validated empirically and monitored with diversity statistics (e.g., average pairwise distance, takeover time) to avoid premature convergence.

The remainder of the chapter provides guidelines and hybrid strategies that implement these recommendations in practice.

5.12 Selection Guidelines

Choice of selection operator should be informed first by the problem's modality and deception. For problems that are essentially unimodal or where a single peak is the objective, stronger selection pressure (for example, truncation or larger tournament sizes) accelerates convergence and is often appropriate. For multimodal problems, where multiple peaks may contain useful solutions, moderate pressure such as binary tournament or rank-based selection helps preserve alternative lineages and reduces the risk of premature loss of useful niches. In deceptive landscapes—where locally attractive solutions mislead the search—favor lower pressure and couple selection with explicit diversity-preserving mechanisms (fitness sharing, crowding, speciation or island models) so that the algorithm can continue exploring promising but initially low-frequency regions.

Population size interacts with selection pressure in important ways. Small populations are more susceptible to sampling error and genetic drift, so conservative pressure settings (smaller tournament sizes, gentler ranking parameters, and limited elitism) help maintain useful variation. Larger populations can sustain stronger pressure without as much risk of accidental loss of rare but valuable genotypes, and they are better suited to schemes that rely on sampling stability (e.g., rank-based selection or modest truncation). In all cases, monitor diversity statistics (average pairwise distance, number of unique genotypes, fitness variance) and adjust selection parameters if diversity falls faster than expected.

Selection intensity should also vary over the run rather than remain fixed. Early generations benefit from lower effective pressure—larger temperature in Boltzmann schedules, smaller tournament sizes, or milder ranking—so that the search emphasizes exploration and discovers multiple basins of attraction. As the run progresses and the population accumulates evidence about promising regions, gradually increase pressure (reduce temperature, enlarge tournaments, or tighten truncation) to focus effort on exploitation and refinement. Adaptive schedules, occasional re-introduction of random immigrants, or multi-level selection (different operators for parent selection and survivor selection) are practical ways to implement this temporal modulation while guarding against premature convergence.

5.13 Hybrid Selection Strategies

Hybrid selection strategies combine multiple selection ideas to obtain better practical performance than any single method in isolation. One common approach is adaptive selection, where operators or their parameters are adjusted automatically during the run in response to population statistics. Examples include annealing a Boltzmann temperature $T(t)$, increasing tournament size as diversity drops, or switching from rank-based to truncation selection during late exploitation. Adaptive rules can be simple (predefined schedules) or feedback-driven (triggered by measured diversity, fitness improvement rate, or takeover time). The central advantage of adaptation is that it permits different phases of the search—exploration and exploitation—to use different pressure regimes without manual retuning for each problem instance.

Multi-level selection splits selection responsibilities across different stages or hierarchies. For instance, parent selection (which chooses who mates) can use a low-pressure method to preserve variety of mating combinations, while survivor selection (which decides who remains in the population) can be more aggressive to retain progress. Similarly, island or hierarchical population models run semi-independent subpopulations with occasional migration; within each island different selection policies may be used to encourage complementary search behavior. Multi-level designs are particularly effective when search must balance global exploration with local refinement or when computational resources are distributed across nodes.

Combined methods explicitly mix selection operators to exploit complementary strengths. Practical examples include performing tournament selection for most parents but reserving a fraction of survivors for rank-based or fitness-shared selection to preserve niches, or applying stochastic universal sampling with elitist replacement to reduce sampling variance while guaranteeing the best-so-far individuals survive. When combining operators, carefully manage interactions — for example, ensure that deterministic components (elitism, truncation) do not negate the diversity benefits of stochastic components. Empirical validation, monitoring of diversity metrics, and conservative parameterization (small elite fractions, limited truncation windows) help achieve robust gains.

In practice, hybrid strategies are most useful when the problem exhibits multiple regimes of difficulty (early exploration, mid-run discovery of promising basins, late-stage refinement) or when robustness across problem instances is required. Implement hybrids incrementally, instrument population statistics to guide choices, and prefer simple, interpretable combinations over elaborate schemes unless justified by experimental results.

Chapter 6
Crossover (Recombination) in Genetic Algorithms

6.1 Introduction to Crossover

Crossover, or recombination, is the primary genetic operator in genetic algorithms: it constructs new candidate solutions by combining genetic material from two (or more) parents. By recombining existing solutions, crossover leverages useful partial solutions—so-called building blocks—to produce offspring that may inherit and amplify beneficial traits while exploring nearby regions of the search space. In practice, crossover works together with selection and mutation to drive population-level search toward better solutions.

Biologically, crossover is inspired by sexual reproduction where chromosomes exchange segments during meiosis and genes assort independently into gametes. These natural mechanisms generate variation: offspring differ from their parents, which increases diversity and the raw material upon which selection can act. The analogy emphasizes two useful ideas for algorithms: mixing parental material to preserve and combine advantageous substructures, and introducing variation to avoid premature convergence.

Key phenomena from biology map directly to algorithm design. Crossing over swaps contiguous genetic segments (preserving local structure), independent assortment randomizes combinations of chromosomes (promoting novel mixes), and the resulting genetic diversity helps populations escape local optima. The notion of building blocks suggests that compact, high-quality gene combinations should be protected and recombined rather than destroyed by overly disruptive operators.

Crossover typically proceeds in three conceptual steps: select parents for mating (usually biased by fitness), choose one or more crossover points or a recombination scheme, and exchange genetic material to form offspring. Implementation details (where the cut points are placed or whether genes are blended arithmetically) determine how much structure is preserved versus how much novelty is introduced; these choices critically shape the search dynamics.

A practical control over crossover is its application probability p_c: the fraction of selected parent pairs that actually undergo recombination. Common practice places p_c in the range 0.6–0.9. High p_c increases exploration by producing many new combinations each generation and can speed convergence when useful building blocks exist, while low p_c places more emphasis on exploiting existing individuals and relies more on mutation and selection to introduce variation.

Crossover mediates the trade-off between exploration and exploitation. Exploitation arises when the operator successfully combines and preserves good substructures from parents, accelerating progress toward high-quality solutions. Exploration occurs when recombination produces novel configurations not present in either parent, increasing the chance of discovering better regions of the search space. Effective algorithm design tunes the operator so that both behaviors occur in balance.

The risk of schema disruption—breaking apart useful gene combinations—depends on

how crossover is implemented and where cuts occur. Less disruptive schemes preserve adjacent relationships and short schemas, while more disruptive schemes (or many cut points) can destroy long, coadapted gene patterns even as they increase diversity. Awareness of these effects guides the choice of crossover style and its parameters for a given problem domain.

In practice, designers choose crossover type and rate based on representation, problem structure, and empirical testing: start with standard p_c values (around 0.8), monitor diversity and progress, and adjust to favor either preservation of building blocks or increased exploration as needed. Because crossover interacts with selection, population size, and mutation, tuning should be done holistically and validated by experiments on representative problem instances.

6.2 Binary Crossover Operators

6.2.1 Definition and Function of Crossover Operator

Crossover is a genetic operator used to vary the arrangement of chromosomes from one generation to the next [19, 42, 1]. The crossover method used depends on the encoding method applied.

In practice, crossover occurs in three stages: the reproduction operator selects a pair of individuals for mating, a crossover site is chosen along the length of the string, and the values after that site are exchanged between the two parents to form new offspring.

In binary chromosome representation, each individual in the population is represented as a sequence of bits (0 and 1) that express a potential solution to a problem.

6.2.2 One-Point Crossover

Single point crossover is the simplest form of crossover. One crossover position is randomly selected, then the part of the two parents after that position is exchanged to form two new offspring. The strings resulting from this process have Positional Bias characteristics.

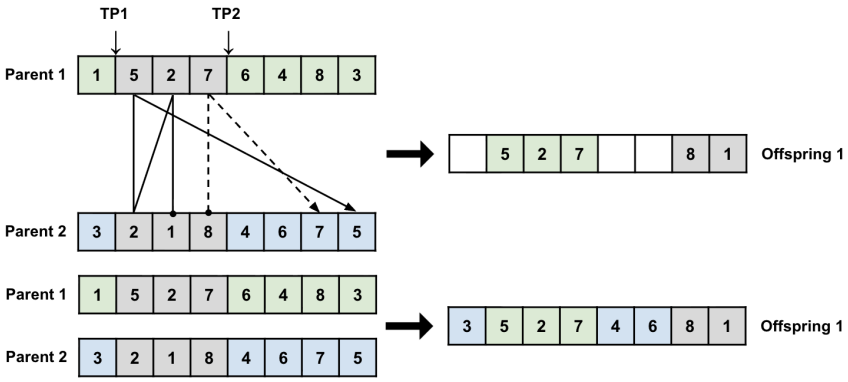

Figure 6.1: Single Point Crossover for binary chromosomes

6.2.3 One-Point Crossover

Single crossover point divides chromosomes into two segments.

Algorithm

Algorithm 5 One-Point Crossover

$k \leftarrow \text{RandomInteger}(1,\, l-1)$
$child1 \leftarrow parent1[1{:}k] + parent2[k+1{:}l]$
$child2 \leftarrow parent2[1{:}k] + parent1[k+1{:}l]$

Example

$$\text{Parent 1:} \quad 1|1010011 \tag{6.1}$$
$$\text{Parent 2:} \quad 0|0111100 \tag{6.2}$$
$$\text{Child 1:} \quad 1|0111100 \tag{6.3}$$
$$\text{Child 2:} \quad 0|1010011 \tag{6.4}$$

Crossover point at position 1.

Characteristics

Single-point crossover is simple and efficient; it tends to preserve long building blocks near chromosome ends but may disrupt blocks that cross the crossover point, producing a positional bias where end positions are less likely to be separated.

6.2.4 Two-Point Crossover

Two crossover points create three segments.

Multi-point crossover has a mechanism like single-point crossover; the difference is the number of randomly selected positions. In Multi-point crossover, a number N of positions are randomly selected along the length of the chromosome. These positions are exchanged to form two new offspring.

The objectives of multi-point crossover include increasing genetic diversity in the population, reducing positional bias, and increasing the chances of recombination from various schemas or different solution blocks. However, using too many cut points can become too disruptive, as it can damage already good gene combinations (coadapted alleles).

Algorithm

Algorithm 6 Two-Point Crossover

$(k_1, k_2) \leftarrow$ Two distinct random integers with $1 \le k_1 < k_2 \le l-1$
$child1 \leftarrow parent1[1{:}k_1] + parent2[k_1+1{:}k_2] + parent1[k_2+1{:}l]$
$child2 \leftarrow parent2[1{:}k_1] + parent1[k_1+1{:}k_2] + parent2[k_2+1{:}l]$

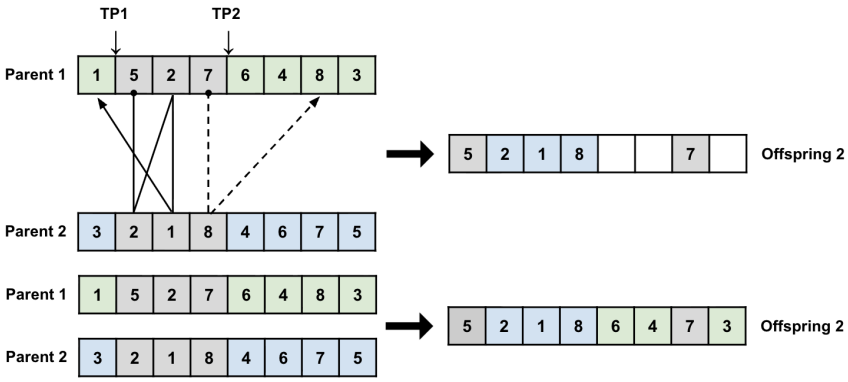

Figure 6.2: Multi-point Crossover for binary chromosomes

Example

$$\begin{aligned} \text{Parent 1:} \quad & 11|010|011 && (6.5)\\ \text{Parent 2:} \quad & 00|111|100 && (6.6)\\ \text{Child 1:} \quad & 11|111|011 && (6.7)\\ \text{Child 2:} \quad & 00|010|100 && (6.8) \end{aligned}$$

Crossover points at positions 2 and 5.

Advantages

Two-point crossover reduces positional bias and can preserve building blocks at chromosome ends, although it is generally more disruptive than one-point crossover.

6.2.5 Uniform Crossover

Each gene is independently chosen from either parent [40, 15].

In uniform crossover, each gene (bit) is randomly selected from one of the corresponding genes on the parent chromosome. This selection process is done independently for each gene position. This method can be analogized to tossing a coin. If the result is "head", the gene is taken from parent 1; if the result is "tail", the gene is taken from parent 2.

This method provides equal opportunity for each gene to be inherited from one of the parents, thereby increasing genetic diversity and eliminating positional bias that typically appears in single-point crossover or multi-point crossover.

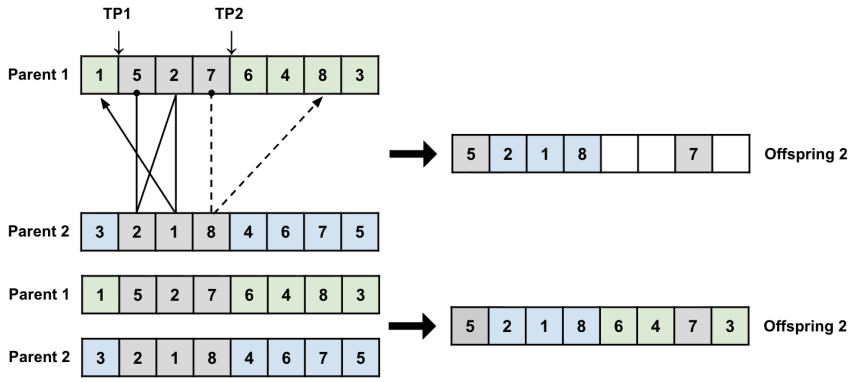

Figure 6.3: Uniform Crossover for binary chromosomes

Algorithm

Algorithm 7 Uniform Crossover

for $i = 1$ to l **do**
  $r \leftarrow$ UniformRandom(0,1)
  if $r < 0.5$ **then**
    $child1[i] \leftarrow parent1[i]$; $child2[i] \leftarrow parent2[i]$
  else
    $child1[i] \leftarrow parent2[i]$; $child2[i] \leftarrow parent1[i]$
  end if
end for

Example with Mask

$$\text{Parent 1:} \quad 11010011 \tag{6.9}$$
$$\text{Parent 2:} \quad 00111100 \tag{6.10}$$
$$\text{Mask:} \quad 10110100 \tag{6.11}$$
$$\text{Child 1:} \quad 10111011 \tag{6.12}$$
$$\text{Child 2:} \quad 01010100 \tag{6.13}$$

Mask bit 1: take from Parent 1, Mask bit 0: take from Parent 2.

Properties

Uniform crossover has a high disruption potential and eliminates positional bias; it is useful when gene positions are independent but can destroy long building blocks.

6.2.6 Multi-Point Crossover

Generalization with k crossover points.

Characteristics

Multi-point crossover generalizes from no crossover ($k = 0$, copy parents) through one-point ($k = 1$) up to the limit $k = l-1$ which approaches uniform crossover in expectation; increasing k moves the operator closer to uniform recombination.

6.3 Integer Chromosome Crossover

Unlike binary chromosomes that use bits 0 and 1, integer representation is more suitable for problems involving discrete parameters or numerical values that have quantitative meaning, such as scheduling, sequencing, or combinatorial optimization.

6.3.1 Single-Point Crossover for Integer

Single-Point Crossover is the simplest form of crossover. One crossover position is randomly selected, then the part of the two parents after that position is exchanged to form two new offspring. The strings resulting from this process have Positional Bias characteristics.

6.3.2 Multi-point Crossover for Integer

Multi-point crossover has a mechanism like single-point crossover; the difference is the number of randomly selected positions. In Multi-point crossover, a number N of positions are randomly selected along the length of the chromosome. These positions are exchanged to form two new offspring.

The objectives of multi-point crossover include increasing genetic diversity in the population, reducing positional bias, and increasing the chances of recombination from various schemas or different solution blocks. However, using too many cut points can become too disruptive, as it can damage already good gene combinations (coadapted alleles).

6.3.3 Uniform Crossover for Integer

In uniform crossover, each gene is randomly selected from one of the corresponding genes on the parent chromosome. This selection process is done independently for each gene position. This method can be analogized to tossing a coin. If the result is "head", the gene is taken from parent 1; if the result is "tail", the gene is taken from parent 2.

This method provides equal opportunity for each gene to be inherited from one of the parents, thereby increasing genetic diversity and eliminating positional bias that typically appears in single-point crossover or multi-point crossover.

6.4 Real-Valued Crossover Operators

Crossover on real chromosomes is a genetic recombination process in Genetic Algorithms applied to chromosomes represented in real number form (floating-point representation). This representation is commonly used to solve continuous optimization problems, where decision variables have values within a certain range and are not limited to integers or binary.

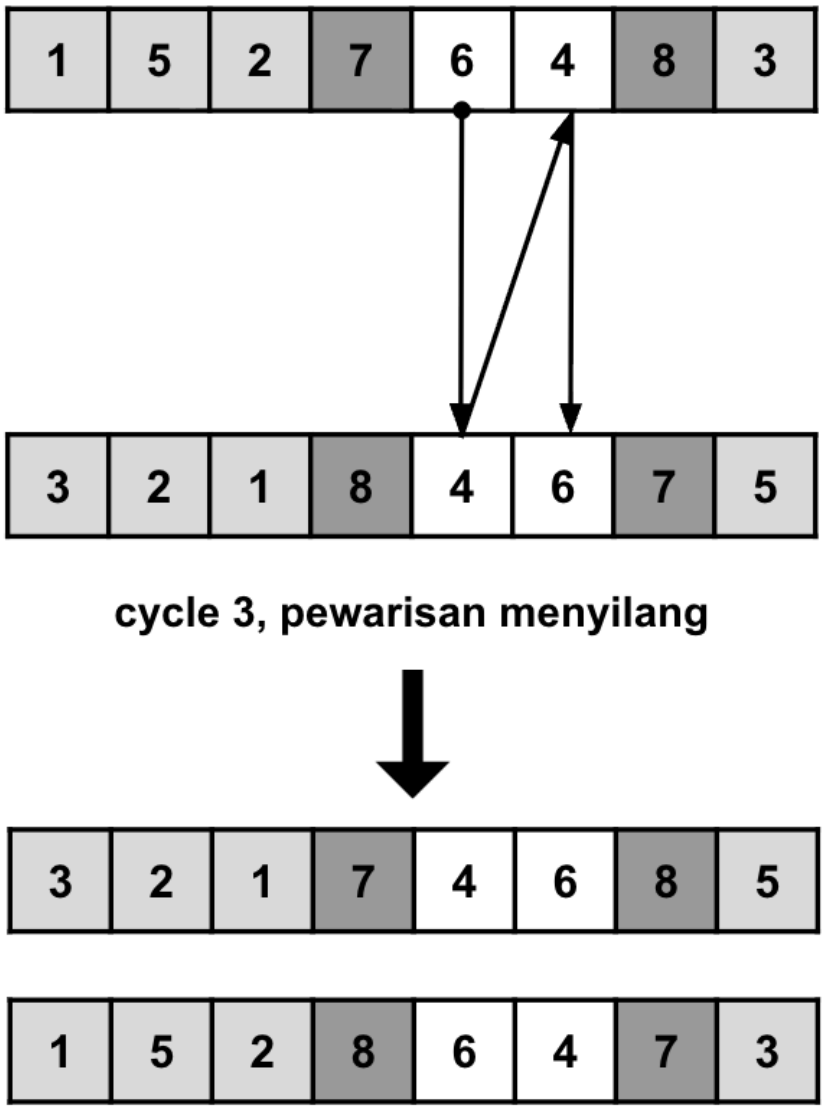

Figure 6.4: Single-Point Crossover for integer chromosomes

Unlike crossover on binary or integer chromosomes, the crossover mechanism for real chromosomes involves arithmetic operations on gene values between parents. This method allows the creation of offspring with gene values that are between or around the parent gene values, thus maintaining the continuity and stability of the evolution process.

6.4.1 Arithmetic Crossover

Linear combination of parent vectors.

Single Arithmetic Crossover

1. Two parents are represented as:
   - Parent 1: $\langle x_1, \ldots, x_n \rangle$
   - Parent 2: $\langle y_1, \ldots, y_n \rangle$
2. Randomly select one gene (k) to undergo crossover operation

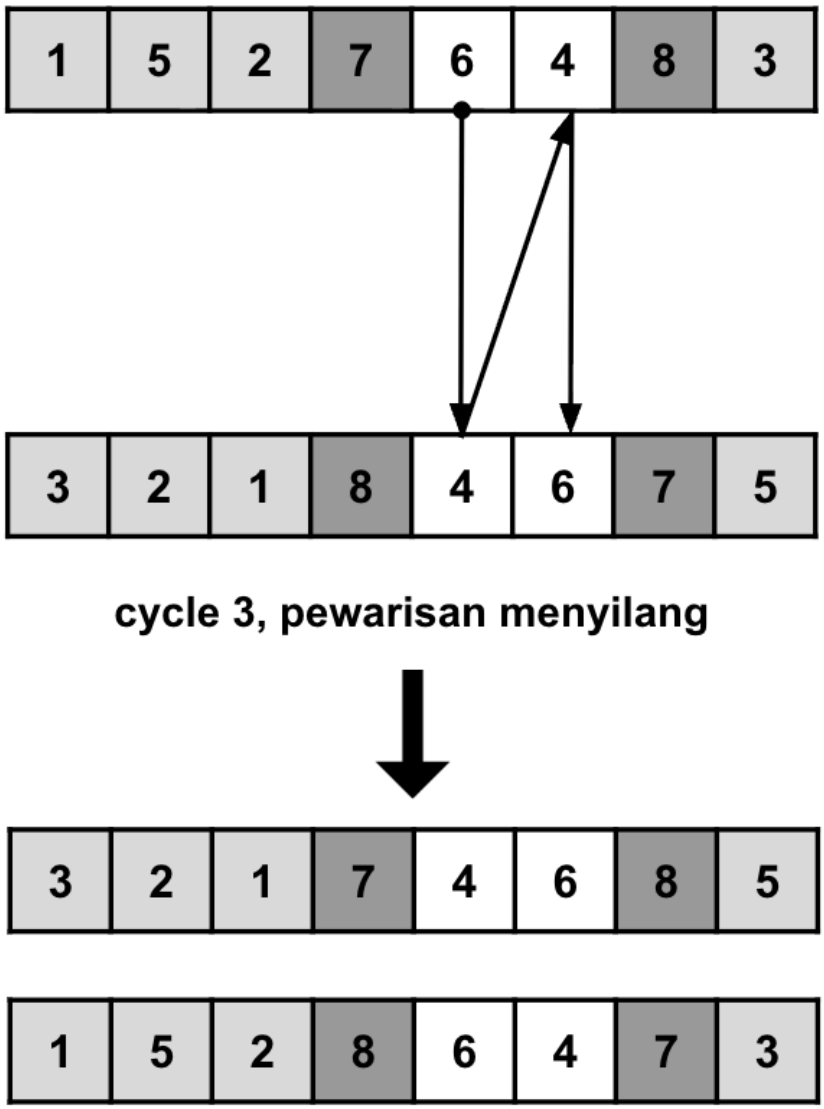

Figure 6.5: Multi-point Crossover for integer chromosomes

3. The result is two offspring formed based on a linear combination of the k-th gene of those parents with control parameter α, where $0 \leq \alpha \leq 1$:
   - Offspring 1: $\langle x_1, \ldots, x_{k-1}, \alpha \cdot y_k + (1-\alpha) \cdot x_k, x_{k+1}, \ldots, x_n \rangle$
   - Offspring 2: $\langle y_1, \ldots, y_{k-1}, \alpha \cdot x_k + (1-\alpha) \cdot y_k, y_{k+1}, \ldots, y_n \rangle$

Simple Arithmetic Crossover

1. Two parents are represented as:
   - Parent 1: $\langle x_1, \ldots, x_n \rangle$
   - Parent 2: $\langle y_1, \ldots, y_n \rangle$
2. Randomly select one gene (k) to become the crossover boundary point
3. The result is two offspring formed based on a linear combination from gene $k+1$ to gene n with control parameter α, where $0 \leq \alpha \leq 1$:

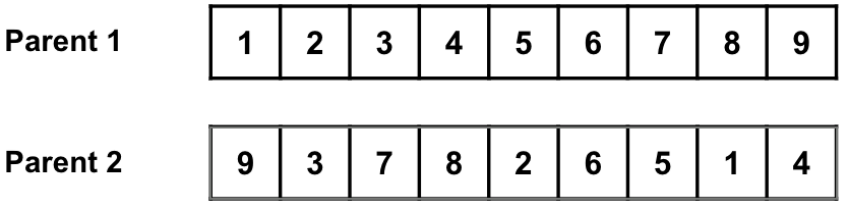

Figure 6.6: Uniform Crossover for integer chromosomes

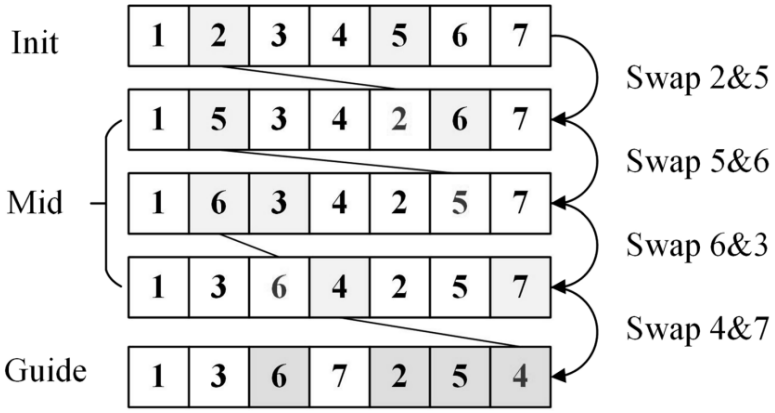

Figure 6.7: Single Arithmetic Crossover for real chromosomes

- Offspring 1: $\langle x_1, \ldots, x_k, \alpha \cdot y_{k+1} + (1-\alpha) \cdot x_{k+1}, \ldots, \alpha \cdot y_n + (1-\alpha) \cdot x_n \rangle$
- Offspring 2: $\langle y_1, \ldots, y_k, \alpha \cdot x_{k+1} + (1-\alpha) \cdot y_{k+1}, \ldots, \alpha \cdot x_n + (1-\alpha) \cdot y_n \rangle$

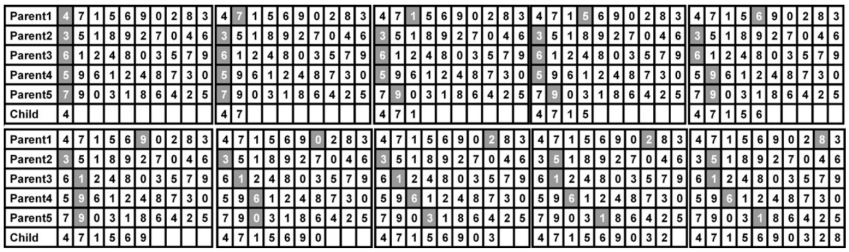

Figure 6.8: Simple Arithmetic Crossover for real chromosomes

Whole Arithmetic Crossover

1. Two parents are represented as:
   - Parent 1: $\langle x_1, \ldots, x_n \rangle$
   - Parent 2: $\langle y_1, \ldots, y_n \rangle$
2. For each gene i ($i = 1, 2, \ldots, n$), offspring are formed with a linear combination of genes from both parents with control parameter α, where $0 \leq \alpha \leq 1$:
   - Offspring 1: $z_i^1 = \alpha \cdot y_i + (1-\alpha) \cdot x_i$
   - Offspring 2: $z_i^2 = \alpha \cdot x_i + (1-\alpha) \cdot y_i$

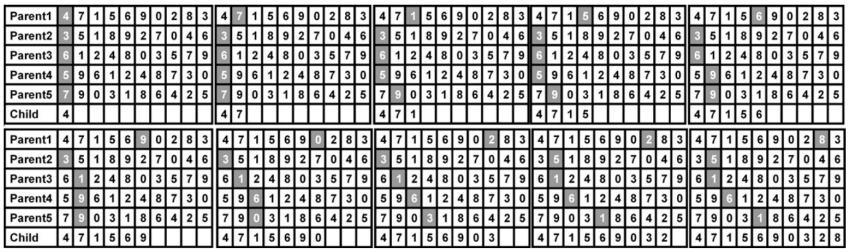

Figure 6.9: Whole Arithmetic Crossover for real chromosomes

Whole Arithmetic Crossover

$$\mathbf{child_1} = \alpha\mathbf{parent_1} + (1-\alpha)\mathbf{parent_2} \tag{6.14}$$

$$\mathbf{child_2} = (1-\alpha)\mathbf{parent_1} + \alpha\mathbf{parent_2} \tag{6.15}$$

where $\alpha \in [0, 1]$ is a random weight.

Simple Arithmetic Crossover

Apply arithmetic crossover to a random subset of genes.

Single Arithmetic Crossover

Apply arithmetic crossover to one randomly selected gene.

Example

$$\text{Parent 1:} \quad (2.1, 5.7, 1.3, 8.9) \tag{6.16}$$

$$\text{Parent 2:} \quad (4.2, 3.1, 6.8, 2.4) \tag{6.17}$$

$$\text{Child 1 } (\alpha = 0.3): \quad (3.57, 4.49, 4.98, 4.17) \tag{6.18}$$

$$\text{Child 2 } (\alpha = 0.3): \quad (2.73, 4.32, 2.98, 6.17) \tag{6.19}$$

6.4.2 BLX-α Crossover (Blend Crossover)

Creates offspring in an interval around the parents.

Algorithm

Algorithm 8 BLX-α Crossover

for $i = 1$ to n **do**
  $c_{min} \leftarrow \min(parent1[i], parent2[i])$
  $c_{max} \leftarrow \max(parent1[i], parent2[i])$
  $I \leftarrow c_{max} - c_{min}$
  Sample $child[i]$ uniformly from $[c_{min} - \alpha I,\ c_{max} + \alpha I]$
end for

Parameters

- $\alpha = 0$: Offspring between parents
- $\alpha = 0.5$: Standard BLX-0.5
- Larger α: More exploration beyond parents

6.4.3 SBX (Simulated Binary Crossover)

Simulates the behavior of one-point crossover for real-valued genes.

Formula

$$child_{1i} = 0.5[(1 + \beta_i)parent_{1i} + (1 - \beta_i)parent_{2i}] \tag{6.20}$$
$$child_{2i} = 0.5[(1 - \beta_i)parent_{1i} + (1 + \beta_i)parent_{2i}] \tag{6.21}$$

where β_i is calculated from:

$$\beta_i = \begin{cases} (2u_i)^{1/(\eta_c+1)} & \text{if } u_i \leq 0.5 \\ \left(\frac{1}{2(1-u_i)}\right)^{1/(\eta_c+1)} & \text{if } u_i > 0.5 \end{cases} \tag{6.22}$$

$u_i \sim U[0, 1]$ and η_c is the distribution index.

6.5 Permutation Crossover Operators

6.5.1 Order Crossover (OX)

Preserves relative order of elements from one parent [34, 28].

Algorithm

Algorithm 9 Order Crossover (OX)

Choose two crossover points a, b with $1 \leq a < b \leq n$
Copy segment $parent1[a{:}b]$ into $child[a{:}b]$
$pos \leftarrow b + 1$ (wrap to 1 if $> n$)
for each element x in $parent2$ in order **do**
  if x not in $child$ **then**
    $child[pos] \leftarrow x$
    $pos \leftarrow pos + 1$ (wrap to 1 if $> n$)
  end if
end for

Example

$$\text{Parent 1:} \quad (1,2,3,4,5,6,7,8,9) \tag{6.23}$$
$$\text{Parent 2:} \quad (9,3,7,8,2,6,5,1,4) \tag{6.24}$$
$$\text{Copy segment:} \quad (\_,\_,3,4,5,6,\_,\_,\_) \tag{6.25}$$
$$\text{Fill from P2:} \quad (7,8,3,4,5,6,2,1,9) \tag{6.26}$$

6.5.2 Partially Mapped Crossover (PMX)

Creates mapping between elements in the crossover segment [21, 28].

Algorithm

Algorithm 10 Partially Mapped Crossover (PMX)

Choose two crossover points a, b with $1 \leq a < b \leq n$
Copy $parent1[a{:}b]$ into $child1[a{:}b]$ and $parent2[a{:}b]$ into $child2[a{:}b]$
Create mapping pairs from exchanged segments
for each position i outside $[a, b]$ **do**
  $val \leftarrow parent1[i]$
  while val is already in $child1[a{:}b]$ **do**
    $val \leftarrow$ mapping of val (follow mapping until an unused value found)
  end while
  $child1[i] \leftarrow val$ {Repeat symmetrically for $child2$}
end for

Example

$$\text{Parent 1:} \quad (1,2,3,4,5,6,7,8,9) \tag{6.27}$$
$$\text{Parent 2:} \quad (5,4,6,9,2,3,7,1,8) \tag{6.28}$$
$$\text{Mapping:} \quad 3 \leftrightarrow 6, 4 \leftrightarrow 9, 5 \leftrightarrow 2, 6 \leftrightarrow 3 \tag{6.29}$$
$$\text{Child 1:} \quad (1,5,6,9,2,3,7,8,4) \tag{6.30}$$

6.5.3 Cycle Crossover (CX)

Preserves absolute positions of elements from both parents [34, 35].

Algorithm

Algorithm 11 Cycle Crossover (CX)

Initialize all positions as unassigned
$cycleStart \leftarrow 1$
while there are unassigned positions **do**
  $i \leftarrow cycleStart$
  Build cycle: add i to cycle; set $i \leftarrow$ position of $parent1[i]$ in $parent2$; repeat until returning to $cycleStart$
  For indices in cycle assign values from $parent1$ to $child1$ and from $parent2$ to $child2$
  Choose next unassigned position as new $cycleStart$
end while

6.5.4 Edge Recombination Crossover

Preserves edge information from both parents (useful for TSP).

Algorithm

Algorithm 12 Edge Recombination Crossover

Build edge table: for each element list its neighbors from both parents (no duplicates)
$current \leftarrow$ element with fewest edges (break ties randomly)
for $pos = 1$ to n **do**
  $child[pos] \leftarrow current$
  Remove $current$ from all edge lists
  if edge table of $current$ has any neighbors **then**
    $next \leftarrow$ neighbor of $current$ with fewest edges (break ties randomly)
  else
    $next \leftarrow$ random unused element
  end if
  $current \leftarrow next$
end for

6.6 Crossover Analysis

6.6.1 Schema Disruption

The probability that a schema H is disrupted by crossover:

One-Point Crossover

$$P_{disruption} = p_c \cdot \frac{\delta(H)}{l-1} \tag{6.31}$$

Two-Point Crossover

$$P_{disruption} = p_c \cdot \left(\frac{2\delta(H)}{l-1} - \frac{\delta(H)(\delta(H)-1)}{(l-1)(l-2)} \right) \tag{6.32}$$

Uniform Crossover

$$P_{disruption} = p_c \cdot \left(1 - \left(\frac{1}{2} \right)^{o(H)-1} \right) \tag{6.33}$$

6.6.2 Building Block Preservation

Short schemas are generally better preserved by crossover operators, while long schemas are more likely to be disrupted (particularly by uniform crossover); tightly linked genes benefit from permutation operators such as order crossover which preserve adjacency relationships.

6.7 Advanced Crossover Techniques

6.7.1 Adaptive Crossover

Adaptive crossover adjusts parameters dynamically based on signals such as population diversity, fitness improvement rate, generation number, and individual fitness levels.

6.7.2 Multiple Parent Crossover

Multiple-parent crossover combines material from more than two parents (see [14, 46, 16, 9]); examples include scanning crossovers that traverse parents sequentially, voting crossovers that use a majority rule, and averaging crossovers that compute averages of parent gene values.

6.7.3 Problem-Specific Crossover

Problem-specific crossover operators are designed to preserve domain constraints and useful structure: for graph problems preserve graph properties, for scheduling preserve temporal constraints, and for neural networks preserve network topology.

6.8 Crossover Guidelines

6.8.1 Choosing Crossover Type

Choose crossover type to match the representation: for binary use one-point, two-point, or uniform; for real-valued use arithmetic, BLX-α, or SBX; for permutations use OX, PMX, or CX depending on problem structure; variable-length representations require specialized operators.

6.8.2 Parameter Setting

Typical parameter suggestions: start with crossover rate $p_c \approx 0.8$–0.9; larger populations can tolerate higher crossover rates; harder problems may benefit from lower rates and more conservative recombination.

6.8.3 Empirical Testing

Empirical testing should compare multiple crossover operators, vary parameters, measure diversity and convergence, and include problem-specific metrics to validate choices.

Chapter 7
Mutasi dan Pembaruan Generasi

Pada bab-bab sebelumnya, kita telah membahas operasi-operasi fundamental Algoritma Genetika (AG) termasuk pengkodean, evaluasi kebugaran, seleksi, dan pindah silang. Bab ini melengkapi pembahasan operator AG dengan mengkaji **mutasi** dan **mekanisme pembaruan generasi** [2, 43]. Operasi-operasi ini sangat penting untuk mempertahankan keragaman genetik dan memastikan kemampuan algoritma untuk mengeksplorasi ruang pencarian secara efektif.

7.1 Pengantar Mutasi

Setelah tahap rekombinasi (pindah silang) diterapkan pada semua pasangan kromosom dalam kumpulan kawin, menghasilkan N kromosom (dengan N adalah ukuran populasi), AG mengeksekusi operator mutasi pada setiap kromosom tersebut. Mutasi adalah operator kritis yang mencegah konvergensi prematur ke optima lokal, mempertahankan keragaman genetik dalam populasi, memperkenalkan materi genetik baru yang mungkin tidak ada dalam populasi awal, dan menyediakan mekanisme untuk meloloskan diri dari optima lokal.

7.1.1 Apa itu Mutasi?

Mutasi adalah proses mengubah nilai satu atau lebih gen dalam genom [21, 31, 6]. Lebih spesifik, mutasi dapat mengubah alel dari gen pada lokus tertentu ke alel lain, membantu menghindari konvergensi prematur (yaitu mencapai hasil suboptimal yang bukan maksimum global), dan menciptakan keturunan yang belum tentu lebih baik dari induknya.

Secara konkret, mutasi mengubah satu atau lebih nilai gen (alel) pada lokus yang dipilih. Perubahan ini bisa acak atau mengikuti aturan stokastik sederhana; tujuan utamanya adalah mempertahankan variasi dalam populasi sehingga proses pencarian dapat terus mengeksplorasi wilayah-wilayah menjanjikan dan baru dari lanskap kebugaran. Mutasi kadang-kadang menghasilkan keturunan yang lebih rendah, tetapi sering kali merupakan satu-satunya mekanisme yang mampu memperkenalkan blok bangunan baru yang mengarah pada perbaikan di masa depan.

Catatan Penting: Populasi baru yang dihasilkan dari mutasi tidak dijamin lebih baik dari populasi sebelumnya. Namun, mutasi menyediakan mekanisme esensial untuk mempertahankan keragaman dan mengeksplorasi wilayah baru dari ruang pencarian.

7.1.2 Mutasi dalam Algoritma Evolusioner vs. Evolusi Biologis

Dalam evolusi biologis, mutasi biasanya dianggap berbahaya karena organisme kompleks memiliki sistem yang sangat saling bergantung. Namun, dalam Algoritma Evolusioner (AE):

Meskipun mutasi biologis sering merusak pada organisme kompleks, situasinya berbeda dalam Algoritma Evolusioner. Representasi yang digunakan dalam AE biasanya jauh

lebih sederhana dan lebih modular daripada genom biologis, sehingga perubahan kecil dan terlokalisasi dapat menghasilkan variasi konstruktif. Sebagai hasilnya, mutasi dalam AE sering dapat menghasilkan keragaman yang bermanfaat: memutasikan subset kecil dari gen dapat menghasilkan keturunan yang lebih baik tanpa mengganggu komponen fungsional lain dari solusi.

7.2 Mutasi untuk Representasi Berbeda

Banyak metode mutasi telah diusulkan dalam literatur [30, 6, 24]. Setiap metode memiliki karakteristik khusus dan mungkin hanya dapat diterapkan pada jenis representasi tertentu. Pemilihan operator mutasi harus kompatibel dengan skema pengkodean kromosom.

7.2.1 Mutasi untuk Representasi Biner

Representasi biner menggunakan bentuk mutasi paling sederhana: **mutasi pembalikan bit**.

Mutasi Pembalikan Bit

Dalam mutasi pembalikan bit, setiap bit dalam kromosom memiliki probabilitas P_m (probabilitas mutasi) untuk dibalik: bit dengan nilai 1 menjadi 0, dan bit dengan nilai 0 menjadi 1.

Dalam pengkodean biner, mutasi paling sederhana adalah pembalikan bit: setiap bit secara independen dibalik dengan probabilitas P_m, sehingga 1 menjadi 0 dan sebaliknya. Operator ini minimal dan tidak bias, dan ketika P_m kecil, operator ini memberikan perturbasi langka tetapi bermakna pada string bit yang stabil.

Contoh:

```
Induk:      1 0 1 1 0 1 0 0
                  ^     ^
Keturunan: 1 0 0 1 0 0 0 0
```

Dalam contoh ini, bit pada posisi 3 dan 6 dipilih untuk mutasi dan dibalik.

Algoritma:

Algorithm 13 Mutasi Pembalikan Bit

for setiap gen g_i dalam kromosom **do**
  $r \leftarrow$ bilangan acak dalam $[0, 1]$
  if $r < P_m$ **then**
    Balik g_i: jika $g_i = 1$ maka $g_i \leftarrow 0$, sebaliknya $g_i \leftarrow 1$
  end if
end for

7.2.2 Mutasi untuk Representasi Integer

Representasi integer memerlukan strategi mutasi yang berbeda. Pendekatan umum meliputi pembalikan nilai integer, pemilihan nilai acak, dan mutasi creep.

Pembalikan Nilai Integer

Menggunakan operasi matematika ($+$, $-$, $\times$, $\div$) untuk mengubah nilai gen yang dipilih.

Contoh:

```
Induk:      8  3  7  5  2  1  9  4  6
                  ^        ^
Keturunan: 8  3  2  5  2  8  9  4  6
```

Nilai pada posisi 3 dan 6 diubah menggunakan operasi matematika.

Pemilihan Nilai Acak

Gen yang dipilih diganti dengan nilai yang dipilih secara acak dari rentang yang valid.

Contoh: Jika rentang yang valid adalah $[1, 9]$:

```
Induk:      8  3  7  5  2  1  9  4  6
                     ^
Keturunan: 8  3  7  9  2  1  9  4  6
```

Mutasi Creep

Menambahkan atau mengurangkan nilai integer acak kecil (biasanya ± 1 atau ± 2) pada gen yang dipilih.

Contoh:

```
Induk:      8  3  7  5  2  1  9  4  6
               ^           ^
Keturunan: 8  4  7  5  2  2  9  4  6
```

Metode ini membuat perubahan kecil dan bertahap serta sangat berguna untuk penyetelan halus solusi.

7.2.3 Mutasi untuk Representasi Bernilai Riil

Representasi bernilai riil memiliki karakteristik berbeda dari representasi biner dan integer. Nilai gen dalam representasi riil bersifat kontinu, sedangkan representasi biner dan integer bersifat diskrit. Oleh karena itu, representasi riil memerlukan operator mutasi khusus.

Mutasi Seragam

Dalam mutasi seragam, gen yang dipilih diganti dengan nilai yang diambil dari distribusi acak seragam dalam rentang valid $[a, b]$:

$$x_i' = a + \text{rand}(0, 1) \times (b - a) \tag{7.1}$$

dengan:

- x_i' adalah nilai gen baru
- a dan b adalah batas bawah dan atas
- $\text{rand}(0, 1)$ menghasilkan bilangan acak dalam $[0, 1]$

Mutasi Tidak Seragam dengan Distribusi Tetap

Mutasi ini mirip dengan metode creep untuk representasi integer tetapi menggunakan penambahan bernilai riil. Nilai yang bermutasi dihitung sebagai:

$$x_i' = x_i + \mathcal{N}(0, \sigma^2) \tag{7.2}$$

dengan:

- x_i adalah nilai gen asli
- $\mathcal{N}(0, \sigma^2)$ adalah nilai acak dari distribusi normal (Gaussian) dengan mean 0 dan variansi σ^2
- σ mengontrol ukuran langkah mutasi

Contoh:

```
Induk:      2.45  7.89  3.12  9.01  5.67
                        ^
Keturunan: 2.45  7.89  3.45  9.01  5.67
```

7.2.4 Mutasi untuk Representasi Permutasi

Mutasi pada representasi permutasi harus memastikan bahwa kromosom yang dihasilkan tetap valid (semua elemen muncul tepat sekali). Metode khusus telah dikembangkan untuk menjaga validitas sambil memperkenalkan variasi.

Mutasi Tukar

Dua posisi gen dipilih secara acak, dan nilainya ditukar.

Contoh:

```
Induk:      3  1  5  2  7  6  8  4  9
               ^           ^
Keturunan: 3  1  8  2  7  6  5  4  9
```

Posisi 3 dan 7 dipilih, sehingga nilai 5 dan 8 ditukar.

Algoritma:

Algorithm 14 Mutasi Tukar

$i \leftarrow$ posisi acak dalam kromosom
$j \leftarrow$ posisi acak dalam kromosom (berbeda dari i)
Tukar nilai pada posisi i dan j

Mutasi Sisip

Gen pada satu posisi dihapus dan disisipkan pada posisi lain, menggeser gen-gen di antara keduanya.

Contoh:

```
Induk:      3  1  5  2  7  6  8  4  9
               ^           ^
Keturunan: 3  1  5  2  7  8  6  4  9
```

Gen pada posisi 7 (nilai 8) dihapus dan disisipkan setelah posisi 2 (nilai 5).

Mutasi Acak

Segmen kromosom dipilih, dan gen-gen dalam segmen tersebut diacak secara acak.

Contoh:

```
Induk:      3  1  5  2  7  6  8  4  9
               \_______/
Keturunan: 3  1  2  6  5  7  8  4  9
```

Segmen $\{5, 2, 7, 6\}$ dipilih dan diacak secara acak menjadi $\{2, 6, 5, 7\}$.

Mutasi Inversi

Segmen kromosom dipilih, dan urutan gen dalam segmen tersebut dibalik.

Contoh:

```
Induk:      3  1  5  2  7  6  8  4  9
               \_______/
Keturunan: 3  1  6  7  2  5  8  4  9
```

Segmen $\{5, 2, 7, 6\}$ dibalik menjadi $\{6, 7, 2, 5\}$.

7.3 Mekanisme Pembaruan Generasi

Setelah operasi seleksi, pindah silang, dan mutasi diterapkan pada populasi, mekanisme pembaruan generasi menentukan individu mana yang bertahan ke generasi berikutnya. Proses ini juga disebut **seleksi penyintas** atau **strategi penggantian**.

7.3.1 Model Asli Holland (Penggantian Generasional)

Dalam AG asli Holland [25, 21], semua keturunan menggantikan seluruh populasi induk. Induk dianggap "mati" dan dihapus, sehingga populasi baru sepenuhnya terdiri dari keturunan dan generasi bersifat berbeda dan tidak tumpang tindih.

Dalam model penggantian generasional asli Holland, populasi keturunan sepenuhnya menggantikan induk, menghasilkan generasi yang berbeda dan tidak tumpang tindih. Model ini sederhana dan mudah diimplementasikan serta memberikan pemisahan yang jelas antar generasi. Kelemahan praktis adalah potensi kehilangan induk berkualitas tinggi kecuali mekanisme seperti elitisme digunakan untuk melestarikannya.

7.3.2 Model Generasional dengan Elitisme

Dalam model generasional dengan elitisme, populasi berukuran N kromosom dalam satu generasi digantikan oleh N individu baru di generasi berikutnya [10, 44]. Namun, untuk melestarikan solusi terbaik, k kromosom terbaik (elit) dari generasi induk disalin langsung ke generasi berikutnya sementara $N - k$ posisi yang tersisa diisi dengan keturunan; ini memastikan bahwa solusi terbaik tidak pernah menjadi lebih buruk lintas generasi.

Dalam model generasional dengan elitisme, k individu teratas dari generasi induk dibawa maju tanpa perubahan dan $N - k$ posisi yang tersisa diisi oleh keturunan yang baru dihasilkan. Modifikasi sederhana ini menjamin bahwa solusi terbaik-sejauh-ini tidak

hilang, yang menstabilkan pencarian dan sering mempercepat konvergensi. Pilihan k yang khas adalah kecil (misalnya 1 atau 2), menyeimbangkan pelestarian dan eksplorasi.

Algoritma:

Algorithm 15 Model Generasional dengan Elitisme

Urutkan populasi induk berdasarkan kebugaran
Salin k individu teratas ke generasi berikutnya (elit)
Hasilkan $N - k$ keturunan melalui seleksi, pindah silang, dan mutasi
Tambahkan keturunan ke generasi berikutnya
Generasi berikutnya menjadi generasi saat ini

Nilai khas: $k = 1$ atau $k = 2$ (melestarikan 1-2 individu terbaik)

7.3.3 Pembaruan Steady-State

Dalam model steady-state [44, 39], tidak semua kromosom digantikan dalam setiap generasi; hanya M kromosom yang digantikan dengan $M < N$ (sering $M = 2$, ketika satu perkawinan menghasilkan dua keturunan yang menggantikan dua individu). Strategi penggantian meliputi: **ganti induk** (dua keturunan menggantikan dua induknya), **ganti terburuk** (dua keturunan menggantikan dua individu terburuk), dan **ganti tertua** (dua keturunan menggantikan dua individu tertua). Model ini memungkinkan individu baik untuk berpartisipasi dalam beberapa perkawinan, menghasilkan evolusi yang lebih bertahap, memungkinkan induk dan keturunan hidup berdampingan dalam populasi yang sama, dan dapat lebih efisien secara komputasional.

Dalam skema pembaruan steady-state, hanya sejumlah kecil M individu (dengan $M < N$) yang digantikan pada setiap langkah, yang memungkinkan induk dan keturunan hidup berdampingan dan memungkinkan individu berkualitas tinggi digunakan kembali dalam beberapa perkawinan. Strategi penggantian umum adalah menggantikan induk dari keturunan, menggantikan individu terburuk yang ditemukan dalam populasi, atau menggantikan individu tertua; setiap strategi menekankan trade-off berbeda antara melestarikan keragaman dan mengintensifkan seleksi. Pendekatan steady-state biasanya menghasilkan evolusi yang lebih bertahap dan dapat efisien secara komputasional ketika M kecil.

7.3.4 Pembaruan Kontinu

Dalam pembaruan kontinu, keturunan dan induk dapat hidup berdampingan dalam generasi yang sama; individu dipilih secara acak dari kedua kelompok untuk generasi berikutnya, memberikan tumpang tindih maksimum antar generasi. Metode ini kurang umum digunakan dibandingkan metode pembaruan lainnya.

Skema pembaruan kontinu memungkinkan koeksistensi penuh antara induk dan keturunan dan biasanya memilih individu untuk bertahan dari set gabungan. Ini menghasilkan tumpang tindih generasional maksimum dan populasi yang sangat bercampur, meskipun dalam praktiknya skema seperti itu kurang umum digunakan dibandingkan dengan penggantian generasional atau steady-state.

7.4 Parameter AG

Kinerja Algoritma Genetika sangat bergantung pada pengaturan parameter yang tepat [22, 36, 10]. Parameter utama yang perlu dikonfigurasi adalah:

7.4.1 Probabilitas Pindah Silang (P_c)

P_c adalah probabilitas bahwa dua induk akan mengalami pindah silang. Jika $P_c = 100\%$, semua keturunan dihasilkan melalui pindah silang; jika $P_c = 0\%$, tidak ada pindah silang yang terjadi dan keturunan adalah salinan persis dari induk. Nilai khas berada dalam rentang $P_c \in [0.65, 0.90]$. Nilai yang lebih tinggi (0.8–0.9) mendorong eksplorasi, sedangkan nilai yang lebih rendah melestarikan solusi baik tetapi mengurangi keragaman; pengaturan awal standar adalah $P_c = 0.8$.

Probabilitas pindah silang P_c mengontrol seberapa sering rekombinasi terjadi. Nilai mendekati 1 (misalnya 0.8–0.9) mendorong pencampuran agresif materi parental dan oleh karena itu eksplorasi ruang pencarian, sedangkan nilai yang lebih rendah mengkonservasi struktur parental dan memperlambat penciptaan kombinasi baru. Default yang umum digunakan adalah $P_c \approx 0.8$, tetapi pilihan akhir tergantung pada karakteristik masalah dan penyetelan empiris.

7.4.2 Probabilitas Mutasi (P_m)

P_m adalah probabilitas bahwa gen dalam kromosom keturunan akan mengalami mutasi. Ketika $P_m = 100\%$, semua gen bermutasi (menyebabkan kekacauan), dan ketika $P_m = 0\%$, tidak ada mutasi yang terjadi dan tidak ada materi genetik baru yang diperkenalkan. Nilai khas adalah kecil, misalnya $P_m \in [0.005, 0.01]$ (0.5

Probabilitas mutasi khas sangat kecil sehingga mutasi terjadi jarang; nilai dalam rentang 0.5% hingga 1% per gen adalah titik awal yang umum. Dua heuristik yang umum digunakan adalah $P_m = 1/L$ (satu mutasi per kromosom rata-rata) atau $P_m = 1/(N \times L)$ ketika menskalakan mutasi relatif terhadap total evaluasi. Mengatur P_m terlalu tinggi merusak struktur yang berguna, sedangkan mengaturnya terlalu rendah dapat memungkinkan konvergensi prematur melalui hilangnya keragaman.

$$P_m = \frac{1}{L} \tag{7.3}$$

atau

$$P_m = \frac{1}{N \times L} \tag{7.4}$$

dengan:

- L adalah panjang kromosom (jumlah gen)
- N adalah ukuran populasi

Alasan: Probabilitas mutasi sering diatur sehingga, rata-rata, satu mutasi terjadi per kromosom.

7.4.3 Ukuran Populasi (N)

Ukuran populasi harus proporsional dengan volume ruang pencarian. Jika populasi terlalu kecil, mungkin sulit mencapai optimum global dan pencarian dapat konvergen ke optima lokal; jika populasi terlalu besar, ini memberlakukan biaya komputasi yang berat dan dapat tidak perlu. Rentang khas adalah $N \in [50, 100]$, tetapi nilai yang tepat harus ditentukan melalui eksperimen dan dipilih sesuai dengan kompleksitas masalah dan sumber daya komputasi yang tersedia.

Ukuran populasi N memediasi trade-off antara cakupan eksplorasi ruang pencarian dan biaya komputasi. Populasi kecil dapat gagal mewakili keragaman yang cukup dan dapat konvergen ke optima lokal, sedangkan populasi yang terlalu besar meningkatkan waktu eksekusi tanpa keuntungan proporsional. Sebagai panduan praktis, banyak masalah dimulai dengan N antara 50 dan 100 dan kemudian menyesuaikan berdasarkan kinerja empiris dan sumber daya komputasi yang tersedia.

7.4.4 Jumlah Generasi (G)

Jumlah generasi harus proporsional dengan ukuran populasi dan ukuran ruang pencarian.

Jumlah generasi G harus dipilih dalam kaitannya dengan N dan kompleksitas ruang pencarian: masalah yang lebih besar atau lebih kompleks biasanya memerlukan lebih banyak generasi untuk konvergen. Kriteria penghentian umum meliputi jumlah generasi tetap, jumlah evaluasi kebugaran maksimum, tidak ada perbaikan selama k generasi berturut-turut, mencapai kebugaran target, atau kombinasi yang sesuai dari kondisi-kondisi ini.

7.4.5 Pedoman Umum Pengaturan Parameter

Catatan Penting: Tidak ada aturan universal untuk memilih parameter AG [45, 22]. Pengaturan yang baik biasanya ditemukan melalui kombinasi heuristik teoritis, pengalaman sebelumnya, dan eksperimen sistematis. Konfigurasi awal yang masuk akal adalah memilih representasi yang sesuai dengan masalah (biner, integer, riil atau permutasi), mengatur ukuran populasi N dalam puluhan hingga ratusan rendah (misalnya 50–100), menggunakan $P_c \approx 0.8$, dan mengatur heuristik mutasi seperti $P_m \approx 1/L$ (atau varian berskala seperti $1/(N \times L)$) dengan penyetelan selanjutnya berdasarkan hasil.

7.5 Studi Observasi Parameter

Untuk memahami efek dari parameter yang berbeda, kami menyajikan studi observasi sistematis.

7.5.1 Masalah Uji

Tujuan: Meminimalkan fungsi:

$$h(x_1, x_2) = x_1^2 + x_2^2 \tag{7.5}$$

dengan $x_1, x_2 \in [-10, 10]$

Fungsi kebugaran:

$$\text{Kebugaran} = \frac{1}{x_1^2 + x_2^2 + 0.001} \tag{7.6}$$

Konstanta 0.001 ditambahkan untuk menghindari pembagian dengan nol pada titik optimal $(0, 0)$.

7.5.2 Pengaturan Eksperimental

Pengaturan eksperimental: Studi memvariasikan ukuran populasi (50, 100, 200), presisi bit per variabel (10, 50, 90), probabilitas pindah silang ($P_c \in \{0.5, 0.7, 0.9\}$), dan probabilitas mutasi relatif terhadap panjang kromosom (misalnya $0.5/L$, $1/L$, $2/L$). Untuk memastikan perbandingan yang adil, setiap konfigurasi dibatasi oleh maksimum 20.000 individu yang dievaluasi dan diulang 30 kali untuk mendapatkan statistik yang andal.

7.5.3 Hasil Sampel

Tabel 7.1 menunjukkan hasil yang dipilih dari studi parameter:

Table 7.1: Hasil Observasi Parameter AG

Ukuran Pop	Bit	P_c	P_m	Rata-rata Kebugaran Terbaik	Rata-rata Evaluasi
50	10	0.5	0.0250	839.55	20000
50	50	0.5	0.0050	1000.00	8301.67
50	50	0.7	0.0100	1000.00	20000
50	90	0.7	0.0056	1000.00	8780.00
100	50	0.7	0.0050	1000.00	14416.67
100	90	0.5	0.0111	1000.00	20000
200	50	0.5	0.0050	1000.00	20000
200	90	0.7	0.0056	1000.00	20000
200	90	0.9	0.0028	1000.00	19866.67

Pengamatan kunci: Konfigurasi paling efisien dalam eksperimen ini adalah ukuran populasi 50 dengan 90 bit per variabel, $P_c = 0.7$ dan $P_m \approx 0.0056$, yang secara konsisten mencapai optimum (fitness 1000.00) sambil hanya memerlukan sekitar 8780 evaluasi rata-rata. Mengenai presisi, 10 bit sering kali tidak cukup untuk mencapai optimum, sedangkan 50–90 bit memberikan granularitas yang diperlukan untuk konvergensi yang andal. Populasi yang lebih kecil (misalnya 50) terbukti efisien dalam masalah uji ini, sedangkan populasi yang lebih besar (misalnya 200) menawarkan lebih banyak ketahanan dengan biaya komputasi yang lebih besar — sebuah trade-off klasik antara kecepatan dan keandalan. Probabilitas pindah silang sekitar 0.7 cenderung menyeimbangkan eksplorasi dan eksploitasi secara efektif. Akhirnya, tingkat mutasi rendah pada orde $1/L$ bekerja paling baik: tingkat yang terlalu tinggi memperkenalkan keacakan yang mengganggu, sementara tingkat yang terlalu rendah mengurangi keragaman dan meningkatkan risiko konvergensi prematur.

7.6 Latihan

1. Diberikan dua kromosom induk untuk masalah permutasi:

   - Induk 1: [1, 2, 7, 3, 4, 9, 8, 6, 5]

- Induk 2: [5, 4, 3, 9, 1, 2, 6, 8, 7]

(a) Lakukan Partial-Mapped Crossover (PMX) dengan titik potong pada posisi 2 dan 5

(b) Terapkan mutasi inversi pada keturunan dengan segmen mutasi dari lokus 2 hingga 5

2. Untuk GA dengan pengkodean biner dengan panjang kromosom $L = 50$ dan ukuran populasi $N = 100$:

   (a) Hitung probabilitas mutasi yang sesuai menggunakan $P_m = 1/L$

   (b) Hitung probabilitas mutasi alternatif menggunakan $P_m = 1/(N \times L)$

   (c) Diskusikan mana yang mungkin lebih sesuai dan mengapa

3. Rancang operator mutasi untuk kromosom bernilai riil yang mewakili koordinat (x, y) di mana $x, y \in [-100, 100]$:

   (a) Implementasikan mutasi seragam

   (b) Implementasikan mutasi Gaussian dengan $\sigma = 5$

   (c) Bandingkan perilaku yang diharapkan dari kedua operator

4. Implementasikan dan bandingkan tiga strategi pembaruan generasi:

   (a) Penggantian generasional dengan elitisme ($k = 2$)

   (b) Steady-state dengan penggantian individu terburuk

   (c) Steady-state dengan penggantian individu tertua

   Diskusikan skenario di mana masing-masing mungkin lebih disukai.

5. Untuk fungsi uji $f(x_1, x_2) = x_1^2 + x_2^2$ dengan $x_1, x_2 \in [-10, 10]$:

   (a) Rancang GA lengkap termasuk semua parameter

   (b) Jalankan eksperimen dengan kombinasi parameter yang berbeda

   (c) Analisis parameter mana yang memiliki dampak paling signifikan

   (d) Usulkan konfigurasi parameter optimal berdasarkan hasil Anda

Chapter 8
Real-World Applications and Visual Examples

This chapter showcases real-world applications of genetic algorithms drawn from the course materials and demonstrates how GA concepts are applied in practice [20, 38, 13].

One of the most compelling demonstrations is the application of genetic algorithms to game AI. The course contains an example of a GA-based agent that beats the first level of Super Mario Bros. at 4× speed [32, 27].

The "Towers of Reus" project demonstrates how a GA can be used for gameplay balancing. Users create maps with adjustable parameters while a GA searches for configurations that produce desirable win/lose characteristics. The system can then report whether towers are too strong or too weak and present beatable levels for players to test.

Another example in the course is path-finding: the problem is to find the shortest path through a complex maze. The encoding is a sequence of movement directions (up, down, left, right) [28]. A suitable fitness function is the inverse of the path length with penalties for hitting walls. Crossover is used to combine successful path segments and mutation explores new moves.

Physical robot navigation shows how GAs transfer to hardware applications. Use cases include real-time path planning in dynamic environments, integrating sensor data for obstacle avoidance, and evolving adaptive behaviors based on environmental feedback.

The course also references simulated-evolution examples available online at `http://www.wreck.devisland.net/ga/`. These examples illustrate features such as morphology evolution (changes to body structure), locomotion pattern optimization, environmental adaptation, and multi-objective fitness criteria like speed, stability, and efficiency [11, 12, 26].

An intuitive analogy used in the materials compares a population of individuals with varying physical abilities: selection favors those who jump higher, inheritance passes traits to subsequent generations, and mutation introduces random technique variations.

A practical optimisation example is daily-commute planning. In this analogy, route options act as genes, traffic conditions act as environmental factors, and time and fuel consumption serve as fitness criteria. Over repeated generations the system can learn to prefer more efficient routes.

The chapter also highlights GA's relationship with other paradigms: machine learning (kernel methods, SVMs, Hidden Markov models, Bayesian methods), soft computing (neural networks and fuzzy systems), and hybrid approaches such as reinforcement learning.

From a conceptual standpoint the course contrasts Lamarck's idea of acquired characteristics with the Darwin-Wallace view of selection. Genetic algorithms follow Darwinian principles by applying random variation and selection to search for good solutions. These visual examples demonstrate GA's wide applicability across entertainment (game AI and procedural content generation), robotics (path planning and adaptive behaviour), simulation (artificial life and evolution studies), optimization (route planning and resource allocation), and research (studying evolutionary processes). The key insight is that GAs

provide a unified framework for solving complex optimization problems across diverse domains, making them a versatile tool in computational intelligence.

Appendix A
Implementasi Algoritma

A.1 Implementasi Algoritma Genetika Dasar

A.1.1 Implementasi Python

Listing A.1: Algoritma Genetika Dasar dalam Python

```python
import numpy as np
import matplotlib.pyplot as plt
from typing import List, Tuple, Callable

class GeneticAlgorithm:
    def __init__(self,
                 fitness_func: Callable,
                 chromosome_length: int,
                 population_size: int = 100,
                 crossover_rate: float = 0.8,
                 mutation_rate: float = 0.01,
                 elitism: bool = True):

        self.fitness_func = fitness_func
        self.chromosome_length = chromosome_length
        self.population_size = population_size
        self.crossover_rate = crossover_rate
        self.mutation_rate = mutation_rate
        self.elitism = elitism

        # Initialize population
        self.population = self._initialize_population()
        self.fitness_history = []
        self.best_individual = None
        self.best_fitness = float('-inf')

    def _initialize_population(self) -> np.ndarray:
        """Inisialisasi populasi biner acak"""
        return np.random.randint(0, 2,
                                 (self.population_size, self.
                                    chromosome_length))

    def _evaluate_fitness(self, population: np.ndarray) -> np.
       ndarray:
        """Evaluasi fungsi kesesuaian untuk semua individu"""
        fitness_values = np.array([self.fitness_func(individual)
                                   for individual in population])
        return fitness_values
```

```
37
38      def _tournament_selection(self, population: np.ndarray,
39                                fitness_values: np.ndarray,
40                                tournament_size: int = 3) -> np.
                                      ndarray:
41          """Seleksi turnamen"""
42          selected = []
43          for _ in range(len(population)):
44              # Pilih individu acak untuk turnamen
45              tournament_indices = np.random.choice(len(population)
                    ,
46                                                    tournament_size,
47                                                    replace=False)
48              tournament_fitness = fitness_values[
                    tournament_indices]
49
50              # Pilih pemenang
51              winner_index = tournament_indices[np.argmax(
                    tournament_fitness)]
52              selected.append(population[winner_index])
53
54          return np.array(selected)
55
56      def _one_point_crossover(self, parent1: np.ndarray,
57                               parent2: np.ndarray) -> Tuple[np.
                                     ndarray, np.ndarray]:
58          """Persilangan satu titik (one-point crossover)"""
59          if np.random.random() > self.crossover_rate:
60              return parent1.copy(), parent2.copy()
61
62          crossover_point = np.random.randint(1, len(parent1))
63
64          child1 = np.concatenate([parent1[:crossover_point],
65                                   parent2[crossover_point:]])
66          child2 = np.concatenate([parent2[:crossover_point],
67                                   parent1[crossover_point:]])
68
69          return child1, child2
70
71      def _bit_flip_mutation(self, individual: np.ndarray) -> np.
            ndarray:
72          """Mutasi bit-flip"""
73          mutated = individual.copy()
74          for i in range(len(mutated)):
75              if np.random.random() < self.mutation_rate:
76                  mutated[i] = 1 - mutated[i]  # Membalik bit
77          return mutated
78
79      def _apply_elitism(self, old_population: np.ndarray,
80                         old_fitness: np.ndarray,
81                         new_population: np.ndarray) -> np.ndarray:
```

```
82          """Terapkan elitisme dengan mempertahankan individu
               terbaik"""
83          if not self.elitism:
84              return new_population
85
86          best_index = np.argmax(old_fitness)
87          best_individual = old_population[best_index]
88
89          # Gantikan individu terburuk di populasi baru dengan yang
               terbaik dari lama
90          new_fitness = self._evaluate_fitness(new_population)
91          worst_index = np.argmin(new_fitness)
92          new_population[worst_index] = best_individual
93
94          return new_population
95
96      def evolve(self, generations: int) -> dict:
97          """Loop evolusi utama"""
98          for generation in range(generations):
99              # Evaluasi kesesuaian
100             fitness_values = self._evaluate_fitness(self.
                   population)
101
102             # Lacak individu terbaik
103             max_fitness_idx = np.argmax(fitness_values)
104             if fitness_values[max_fitness_idx] > self.
                   best_fitness:
105                 self.best_fitness = fitness_values[
                       max_fitness_idx]
106                 self.best_individual = self.population[
                       max_fitness_idx].copy()
107
108             # Catat statistik
109             self.fitness_history.append({
110                 'generation': generation,
111                 'best_fitness': np.max(fitness_values),
112                 'avg_fitness': np.mean(fitness_values),
113                 'worst_fitness': np.min(fitness_values)
114             })
115
116             # Seleksi
117             selected = self._tournament_selection(self.population
                   , fitness_values)
118
119             # Persilangan dan mutasi
120             new_population = []
121             for i in range(0, len(selected), 2):
122                 parent1 = selected[i]
123                 parent2 = selected[(i + 1) % len(selected)]
124
125                 # Persilangan
```

```
126                 child1, child2 = self._one_point_crossover(
                        parent1, parent2)
127
128                 # Mutasi
129                 child1 = self._bit_flip_mutation(child1)
130                 child2 = self._bit_flip_mutation(child2)
131
132                 new_population.extend([child1, child2])
133
134             new_population = np.array(new_population[:self.
                    population_size])
135
136             # Terapkan elitisme
137             self.population = self._apply_elitism(self.population
                    ,
138                                                   fitness_values,
139                                                   new_population)
140
141         return {
142             'best_individual': self.best_individual,
143             'best_fitness': self.best_fitness,
144             'fitness_history': self.fitness_history
145         }
146
147     def plot_fitness_history(self):
148         """Plot evolusi kesesuaian sepanjang generasi"""
149         generations = [entry['generation'] for entry in self.
                fitness_history]
150         best_fitness = [entry['best_fitness'] for entry in self.
                fitness_history]
151         avg_fitness = [entry['avg_fitness'] for entry in self.
                fitness_history]
152
153         plt.figure(figsize=(10, 6))
154         plt.plot(generations, best_fitness, label='Kesesuaian␣
                Terbaik', linewidth=2)
155         plt.plot(generations, avg_fitness, label='Kesesuaian␣Rata
                -rata', linewidth=2)
156         plt.xlabel('Generasi')
157         plt.ylabel('Kesesuaian')
158         plt.title('Evolusi␣Kesesuaian')
159         plt.legend()
160         plt.grid(True, alpha=0.3)
161         plt.show()
162
163 # Example usage
164 def onemax_fitness(individual):
165     """Masalah OneMax: maksimalkan jumlah bit bernilai 1"""
166     return np.sum(individual)
167
168 def sphere_function_binary(individual, bounds=(-5.12, 5.12)):
```

```
    """Fungsi Sphere dengan encoding biner"""
    # Dekode biner ke nilai riil
    x = bounds[0] + (bounds[1] - bounds[0]) * np.sum(individual *
        2**np.arange(len(individual))[::-1]) / (2**len(individual)
        - 1)
    return -(x**2)  # Negatif karena kita ingin meminimalkan

# Run GA on OneMax problem
if __name__ == "__main__":
    ga = GeneticAlgorithm(
        fitness_func=onemax_fitness,
        chromosome_length=20,
        population_size=50,
        crossover_rate=0.8,
        mutation_rate=0.01
    )

    result = ga.evolve(generations=100)

    print(f"Individu terbaik: {result['best_individual']}")
    print(f"Kesesuaian terbaik: {result['best_fitness']}")

    ga.plot_fitness_history()
```

A.2 Algoritma Genetika Bernilai Riil

Listing A.2: Implementasi GA Bernilai Riil

```
import numpy as np
from typing import List, Tuple, Callable

class RealValuedGA:
    def __init__(self,
                 fitness_func: Callable,
                 dimensions: int,
                 bounds: List[Tuple[float, float]],
                 population_size: int = 100,
                 crossover_rate: float = 0.8,
                 mutation_rate: float = 0.1,
                 mutation_strength: float = 0.1):

        self.fitness_func = fitness_func
        self.dimensions = dimensions
        self.bounds = bounds
        self.population_size = population_size
        self.crossover_rate = crossover_rate
        self.mutation_rate = mutation_rate
        self.mutation_strength = mutation_strength

        self.population = self._initialize_population()
```

```
23          self.fitness_history = []
24
25      def _initialize_population(self) -> np.ndarray:
26          """Inisialisasi populasi bernilai riil acak"""
27          population = np.zeros((self.population_size, self.
               dimensions))
28          for i in range(self.dimensions):
29              low, high = self.bounds[i]
30              population[:, i] = np.random.uniform(low, high, self.
                   population_size)
31          return population
32
33      def _blx_alpha_crossover(self, parent1: np.ndarray,
34                               parent2: np.ndarray,
35                               alpha: float = 0.5) -> Tuple[np.
                                    ndarray, np.ndarray]:
36          """Persilangan BLX-alpha"""
37          if np.random.random() > self.crossover_rate:
38              return parent1.copy(), parent2.copy()
39
40          child1 = np.zeros_like(parent1)
41          child2 = np.zeros_like(parent2)
42
43          for i in range(len(parent1)):
44              min_val = min(parent1[i], parent2[i])
45              max_val = max(parent1[i], parent2[i])
46              interval = max_val - min_val
47
48              low_bound = max(min_val - alpha * interval, self.
                   bounds[i][0])
49              high_bound = min(max_val + alpha * interval, self.
                   bounds[i][1])
50
51              child1[i] = np.random.uniform(low_bound, high_bound)
52              child2[i] = np.random.uniform(low_bound, high_bound)
53
54          return child1, child2
55
56      def _gaussian_mutation(self, individual: np.ndarray) -> np.
           ndarray:
57          """Mutasi Gaussian"""
58          mutated = individual.copy()
59          for i in range(len(mutated)):
60              if np.random.random() < self.mutation_rate:
61                  noise = np.random.normal(0, self.
                       mutation_strength)
62                  mutated[i] += noise
63
64                  # Pastikan tetap dalam batas
65                  low, high = self.bounds[i]
66                  mutated[i] = np.clip(mutated[i], low, high)
```

```
67
68          return mutated
69
70      def evolve(self, generations: int) -> dict:
71          """Loop evolusi utama"""
72          for generation in range(generations):
73              # Evaluasi kesesuaian
74              fitness_values = np.array([self.fitness_func(ind)
75                                         for ind in self.population])
76
77              # Catat statistik
78              self.fitness_history.append({
79                  'generation': generation,
80                  'best_fitness': np.max(fitness_values),
81                  'avg_fitness': np.mean(fitness_values),
82                  'worst_fitness': np.min(fitness_values)
83              })
84
85              # Seleksi turnamen
86              new_population = []
87              for _ in range(self.population_size // 2):
88                  # Pilih orangtua
89                  parent1_idx = self._tournament_selection(
                        fitness_values)
90                  parent2_idx = self._tournament_selection(
                        fitness_values)
91
92                  parent1 = self.population[parent1_idx]
93                  parent2 = self.population[parent2_idx]
94
95                  # Persilangan
96                  child1, child2 = self._blx_alpha_crossover(
                        parent1, parent2)
97
98                  # Mutasi
99                  child1 = self._gaussian_mutation(child1)
100                 child2 = self._gaussian_mutation(child2)
101
102                 new_population.extend([child1, child2])
103
104             self.population = np.array(new_population)
105
106         # Evaluasi akhir
107         final_fitness = np.array([self.fitness_func(ind)
108                                   for ind in self.population])
109         best_idx = np.argmax(final_fitness)
110
111         return {
112             'best_individual': self.population[best_idx],
113             'best_fitness': final_fitness[best_idx],
114             'fitness_history': self.fitness_history
```

```
        }

    def _tournament_selection(self, fitness_values: np.ndarray,
                              tournament_size: int = 3) -> int:
        """Tournament selection returning index"""
        tournament_indices = np.random.choice(len(fitness_values),
                                              tournament_size,
                                              replace=False)
        tournament_fitness = fitness_values[tournament_indices]
        winner_idx = tournament_indices[np.argmax(tournament_fitness)]
        return winner_idx

# Example: Optimize Rastrigin function
def rastrigin_function(x):
    """Fungsi Rastrigin (masalah minimisasi)"""
    A = 10
    n = len(x)
    return -(A * n + np.sum(x**2 - A * np.cos(2 * np.pi * x)))

# Usage
bounds = [(-5.12, 5.12)] * 2  # 2D Rastrigin
ga = RealValuedGA(
    fitness_func=rastrigin_function,
    dimensions=2,
    bounds=bounds,
    population_size=100,
    mutation_strength=0.1
)

result = ga.evolve(generations=200)
print(f"Best solution: {result['best_individual']}")
print(f"Best fitness: {result['best_fitness']}")
```

A.3 Algoritma Genetika untuk Traveling Salesman Problem

Listing A.3: TSP dengan Algoritma Genetika

```
import numpy as np
import matplotlib.pyplot as plt
from typing import List, Tuple

class TSP_GA:
    def __init__(self,
                 cities: np.ndarray,
                 population_size: int = 100,
                 crossover_rate: float = 0.8,
```

```
                 mutation_rate: float = 0.02):

        self.cities = cities
        self.num_cities = len(cities)
        self.population_size = population_size
        self.crossover_rate = crossover_rate
        self.mutation_rate = mutation_rate

        # Create distance matrix
        self.distance_matrix = self._calculate_distance_matrix()

        # Initialize population
        self.population = self._initialize_population()

    def _calculate_distance_matrix(self) -> np.ndarray:
        """Hitung matriks jarak antar semua kota"""
        n = self.num_cities
        distances = np.zeros((n, n))

        for i in range(n):
            for j in range(n):
                if i != j:
                    distances[i][j] = np.sqrt(
                        (self.cities[i][0] - self.cities[j][0])
                            **2 +
                        (self.cities[i][1] - self.cities[j][1])
                            **2
                    )
        return distances

    def _initialize_population(self) -> List[List[int]]:
        """Inisialisasi populasi dengan permutasi acak"""
        population = []
        for _ in range(self.population_size):
            tour = list(range(self.num_cities))
            np.random.shuffle(tour)
            population.append(tour)
        return population

    def _calculate_tour_distance(self, tour: List[int]) -> float:
        """Hitung total jarak sebuah rute (tour)"""
        total_distance = 0
        for i in range(len(tour)):
            from_city = tour[i]
            to_city = tour[(i + 1) % len(tour)]
            total_distance += self.distance_matrix[from_city][
                to_city]
        return total_distance

    def _fitness(self, tour: List[int]) -> float:
        """Fungsi kesesuaian (invers jarak)"""
```

```
58          distance = self._calculate_tour_distance(tour)
59          return 1.0 / (1.0 + distance)
60
61      def _order_crossover(self, parent1: List[int],
62                           parent2: List[int]) -> Tuple[List[int],
                                 List[int]]:
63          """Order crossover (OX)"""
64          if np.random.random() > self.crossover_rate:
65              return parent1.copy(), parent2.copy()
66
67          size = len(parent1)
68          start, end = sorted(np.random.choice(size, 2, replace=
                False))
69
70          # Buat anak
71          child1 = [None] * size
72          child2 = [None] * size
73
74          # Salin segmen
75          child1[start:end] = parent1[start:end]
76          child2[start:end] = parent2[start:end]
77
78          # Isi posisi yang tersisa
79          self._fill_remaining_ox(child1, parent2, start, end)
80          self._fill_remaining_ox(child2, parent1, start, end)
81
82          return child1, child2
83
84      def _fill_remaining_ox(self, child: List[int], parent: List[
            int],
85                             start: int, end: int):
86          """Fungsi bantu untuk order crossover"""
87          child_set = set(child[start:end])
88          parent_filtered = [city for city in parent if city not in
                child_set]
89
90          # Isi posisi sebelum start
91          for i in range(start):
92              child[i] = parent_filtered.pop(0)
93
94          # Isi posisi setelah end
95          for i in range(end, len(child)):
96              child[i] = parent_filtered.pop(0)
97
98      def _swap_mutation(self, tour: List[int]) -> List[int]:
99          """Mutasi swap"""
100         mutated = tour.copy()
101         if np.random.random() < self.mutation_rate:
102             i, j = np.random.choice(len(tour), 2, replace=False)
103             mutated[i], mutated[j] = mutated[j], mutated[i]
104         return mutated
```

```
105
106     def _tournament_selection(self, fitness_values: List[float],
107                               tournament_size: int = 3) -> int:
108         """Seleksi turnamen"""
109         tournament_indices = np.random.choice(len(fitness_values)
                ,
110                                               tournament_size,
                                                    replace=False)
111         tournament_fitness = [fitness_values[i] for i in
                tournament_indices]
112         winner_idx = tournament_indices[np.argmax(
                tournament_fitness)]
113         return winner_idx
114
115     def evolve(self, generations: int) -> dict:
116         """Loop evolusi utama"""
117         fitness_history = []
118         best_tour = None
119         best_distance = float('inf')
120
121         for generation in range(generations):
122             # Evaluasi kesesuaian
123             fitness_values = [self._fitness(tour) for tour in
                    self.population]
124             distances = [self._calculate_tour_distance(tour)
125                          for tour in self.population]
126
127             # Lacak solusi terbaik
128             min_distance_idx = np.argmin(distances)
129             if distances[min_distance_idx] < best_distance:
130                 best_distance = distances[min_distance_idx]
131                 best_tour = self.population[min_distance_idx].
                        copy()
132
133             # Catat statistik
134             fitness_history.append({
135                 'generation': generation,
136                 'best_distance': np.min(distances),
137                 'avg_distance': np.mean(distances),
138                 'worst_distance': np.max(distances)
139             })
140
141             # Buat populasi baru
142             new_population = []
143
144             # Elitisme: simpan individu terbaik
145             new_population.append(best_tour.copy())
146
147             # Hasilkan sisa populasi
148             while len(new_population) < self.population_size:
149                 # Seleksi
```

```
150                 parent1_idx = self._tournament_selection(
                        fitness_values)
151                 parent2_idx = self._tournament_selection(
                        fitness_values)
152
153                 parent1 = self.population[parent1_idx]
154                 parent2 = self.population[parent2_idx]
155
156                 # Persilangan
157                 child1, child2 = self._order_crossover(parent1,
                        parent2)
158
159                 # Mutasi
160                 child1 = self._swap_mutation(child1)
161                 child2 = self._swap_mutation(child2)
162
163                 new_population.extend([child1, child2])
164
165             # Pangkas sesuai ukuran populasi
166             self.population = new_population[:self.
                    population_size]
167
168         return {
169             'best_tour': best_tour,
170             'best_distance': best_distance,
171             'fitness_history': fitness_history
172         }
173
174     def plot_tour(self, tour: List[int], title: str = "Rute␣
            Terbaik"):
175         """Plot rute (tour)"""
176         plt.figure(figsize=(10, 8))
177
178         # Plot kota-kota
179         plt.scatter(self.cities[:, 0], self.cities[:, 1],
180                     c='red', s=100, zorder=2)
181
182         # Plot rute
183         tour_cities = self.cities[tour + [tour[0]]]  # Tutup loop
184         plt.plot(tour_cities[:, 0], tour_cities[:, 1],
185                  'b-', linewidth=2, zorder=1)
186
187         # Tambahkan label kota
188         for i, city in enumerate(self.cities):
189             plt.annotate(str(i), (city[0], city[1]),
190                          xytext=(5, 5), textcoords='offset␣points'
                                 )
191
192         plt.title(f"{title}\nJarak:␣{self.
                _calculate_tour_distance(tour):.2f}")
193         plt.xlabel("Koordinat␣X")
```

```
        plt.ylabel("Koordinat Y")
        plt.grid(True, alpha=0.3)
        plt.show()

# Example usage
if __name__ == "__main__":
    # Buat kota acak
    np.random.seed(42)
    num_cities = 20
    cities = np.random.rand(num_cities, 2) * 100

    # Inisialisasi dan jalankan GA
    tsp_ga = TSP_GA(cities, population_size=100, mutation_rate
       =0.02)
    result = tsp_ga.evolve(generations=500)

    print(f"Jarak terbaik: {result['best_distance']:.2f}")
    print(f"Rute terbaik: {result['best_tour']}")

    # Plot rute terbaik
    tsp_ga.plot_tour(result['best_tour'])
```

A.4 NSGA-II untuk Optimisasi Multi-Objektif

Listing A.4: Implementasi NSGA-II

```
import numpy as np
from typing import List, Tuple

class NSGA2:
    def __init__(self,
                 objective_functions: List,
                 num_variables: int,
                 bounds: List[Tuple[float, float]],
                 population_size: int = 100,
                 crossover_rate: float = 0.9,
                 mutation_rate: float = 0.1):

        self.objective_functions = objective_functions
        self.num_objectives = len(objective_functions)
        self.num_variables = num_variables
        self.bounds = bounds
        self.population_size = population_size
        self.crossover_rate = crossover_rate
        self.mutation_rate = mutation_rate

        # Ensure even population size
        if self.population_size % 2 != 0:
            self.population_size += 1

```

```
25      def _initialize_population(self) -> np.ndarray:
26          """Inisialisasi populasi acak"""
27          population = np.zeros((self.population_size, self.
               num_variables))
28          for i in range(self.num_variables):
29              low, high = self.bounds[i]
30              population[:, i] = np.random.uniform(low, high, self.
                   population_size)
31          return population
32
33      def _evaluate_objectives(self, population: np.ndarray) -> np.
           ndarray:
34          """Evaluasi semua fungsi objektif untuk populasi"""
35          objectives = np.zeros((len(population), self.
               num_objectives))
36          for i, individual in enumerate(population):
37              for j, obj_func in enumerate(self.objective_functions
                   ):
38                  objectives[i, j] = obj_func(individual)
39          return objectives
40
41      def _dominates(self, obj1: np.ndarray, obj2: np.ndarray) ->
           bool:
42          """Periksa apakah obj1 mendominasi obj2 (mengasumsikan
               minimisasi)"""
43          return np.all(obj1 <= obj2) and np.any(obj1 < obj2)
44
45      def _fast_non_dominated_sort(self, objectives: np.ndarray) ->
           Tuple[List[List[int]], np.ndarray]:
46          """Penyortiran non-dominated cepat"""
47          population_size = len(objectives)
48          domination_count = np.zeros(population_size)
49          dominated_solutions = [[] for _ in range(population_size)
               ]
50          fronts = [[]]
51
52          # Find domination relationships
53          for i in range(population_size):
54              for j in range(population_size):
55                  if i != j:
56                      if self._dominates(objectives[i], objectives[
                           j]):
57                          dominated_solutions[i].append(j)
58                      elif self._dominates(objectives[j],
                           objectives[i]):
59                          domination_count[i] += 1
60
61              if domination_count[i] == 0:
62                  fronts[0].append(i)
63
64          # Build subsequent fronts
```

```
65          current_front = 0
66          while len(fronts[current_front]) > 0:
67              next_front = []
68              for i in fronts[current_front]:
69                  for j in dominated_solutions[i]:
70                      domination_count[j] -= 1
71                      if domination_count[j] == 0:
72                          next_front.append(j)
73              current_front += 1
74              fronts.append(next_front)
75
76          # Remove empty last front
77          fronts.pop()
78
79          # Assign ranks
80          ranks = np.zeros(population_size)
81          for rank, front in enumerate(fronts):
82              for individual in front:
83                  ranks[individual] = rank
84
85          return fronts, ranks
86
87      def _calculate_crowding_distance(self, objectives: np.ndarray
        ,
88                                       front: List[int]) -> np.
                                           ndarray:
89          """Hitung crowding distance untuk individu dalam sebuah
               front"""
90          if len(front) <= 2:
91              return np.full(len(front), float('inf'))
92
93          distances = np.zeros(len(front))
94
95          for obj_idx in range(self.num_objectives):
96              # Sort by objective value
97              sorted_indices = sorted(range(len(front)),
98                                      key=lambda x: objectives[front[
                                           x], obj_idx])
99
100             # Set boundary points to infinity
101             distances[sorted_indices[0]] = float('inf')
102             distances[sorted_indices[-1]] = float('inf')
103
104             # Calculate distances for middle points
105             obj_range = (objectives[front[sorted_indices[-1]],
                    obj_idx] -
106                          objectives[front[sorted_indices[0]],
                                obj_idx])
107
108             if obj_range > 0:
109                 for i in range(1, len(sorted_indices) - 1):
```

```
                    distance = (objectives[front[sorted_indices[i
                        + 1]], obj_idx] -
                                objectives[front[sorted_indices[i -
                                    1]], obj_idx])
                    distances[sorted_indices[i]] += distance /
                        obj_range

        return distances

    def _tournament_selection(self, ranks: np.ndarray,
                              crowding_distances: np.ndarray,
                              population_size: int) -> List[int]:
        """Seleksi turnamen biner berdasarkan rank dan crowding
            distance"""
        selected = []

        for _ in range(population_size):
            # Select two random individuals
            candidates = np.random.choice(len(ranks), 2, replace=
                False)
            i, j = candidates[0], candidates[1]

            # Compare based on rank first, then crowding distance
            if ranks[i] < ranks[j]:
                selected.append(i)
            elif ranks[i] > ranks[j]:
                selected.append(j)
            else:  # Same rank, compare crowding distance
                if crowding_distances[i] > crowding_distances[j]:
                    selected.append(i)
                else:
                    selected.append(j)

        return selected

    def _sbx_crossover(self, parent1: np.ndarray, parent2: np.
        ndarray,
                       eta: float = 20.0) -> Tuple[np.ndarray, np.
                           ndarray]:
        """Simulated Binary Crossover (SBX)"""
        if np.random.random() > self.crossover_rate:
            return parent1.copy(), parent2.copy()

        child1 = np.zeros_like(parent1)
        child2 = np.zeros_like(parent2)

        for i in range(len(parent1)):
            if np.random.random() <= 0.5:
                if abs(parent1[i] - parent2[i]) > 1e-14:
                    y1, y2 = min(parent1[i], parent2[i]), max(
                        parent1[i], parent2[i])
```

```
153
154                     # Calculate beta
155                     rand = np.random.random()
156                     if rand <= 0.5:
157                         beta = (2 * rand) ** (1.0 / (eta + 1))
158                     else:
159                         beta = (1.0 / (2 * (1 - rand))) ** (1.0 /
                                (eta + 1))
160
161                     child1[i] = 0.5 * ((y1 + y2) - beta * (y2 -
                        y1))
162                     child2[i] = 0.5 * ((y1 + y2) + beta * (y2 -
                        y1))
163
164                     # Ensure bounds
165                     low, high = self.bounds[i]
166                     child1[i] = np.clip(child1[i], low, high)
167                     child2[i] = np.clip(child2[i], low, high)
168                 else:
169                     child1[i] = parent1[i]
170                     child2[i] = parent2[i]
171             else:
172                 child1[i] = parent1[i]
173                 child2[i] = parent2[i]
174
175         return child1, child2
176
177     def _polynomial_mutation(self, individual: np.ndarray,
178                              eta: float = 20.0) -> np.ndarray:
179         """Mutasi polinomial"""
180         mutated = individual.copy()
181
182         for i in range(len(mutated)):
183             if np.random.random() < self.mutation_rate:
184                 low, high = self.bounds[i]
185                 delta1 = (mutated[i] - low) / (high - low)
186                 delta2 = (high - mutated[i]) / (high - low)
187
188                 rand = np.random.random()
189                 mut_pow = 1.0 / (eta + 1.0)
190
191                 if rand <= 0.5:
192                     xy = 1.0 - delta1
193                     val = 2.0 * rand + (1.0 - 2.0 * rand) * (xy
                            ** (eta + 1.0))
194                     deltaq = val ** mut_pow - 1.0
195                 else:
196                     xy = 1.0 - delta2
197                     val = 2.0 * (1.0 - rand) + 2.0 * (rand - 0.5)
                            * (xy ** (eta + 1.0))
198                     deltaq = 1.0 - val ** mut_pow
```

```
199
200                 mutated[i] += deltaq * (high - low)
201                 mutated[i] = np.clip(mutated[i], low, high)
202
203         return mutated
204
205     def evolve(self, generations: int) -> dict:
206         """Loop evolusi NSGA-II utama"""
207         # Inisialisasi populasi
208         population = self._initialize_population()
209
210         for generation in range(generations):
211             # Evaluasi objektif
212             objectives = self._evaluate_objectives(population)
213
214             # Penyortiran non-dominated
215             fronts, ranks = self._fast_non_dominated_sort(
                    objectives)
216
217             # Hitung crowding distances
218             crowding_distances = np.zeros(len(population))
219             for front in fronts:
220                 if len(front) > 0:
221                     distances = self._calculate_crowding_distance
                            (objectives, front)
222                     for i, individual_idx in enumerate(front):
223                         crowding_distances[individual_idx] =
                                distances[i]
224
225             # Seleksi untuk mating pool
226             mating_pool_indices = self._tournament_selection(
                    ranks, crowding_distances,
227                                                              self.
                                                                 population_size
                                                                 )
228             mating_pool = population[mating_pool_indices]
229
230             # Buat keturunan melalui persilangan dan mutasi
231             offspring = []
232             for i in range(0, self.population_size, 2):
233                 parent1 = mating_pool[i]
234                 parent2 = mating_pool[i + 1]
235
236                 child1, child2 = self._sbx_crossover(parent1,
                        parent2)
237                 child1 = self._polynomial_mutation(child1)
238                 child2 = self._polynomial_mutation(child2)
239
240                 offspring.extend([child1, child2])
241
242             offspring = np.array(offspring)
```

```
243
244             # Gabungkan populasi orangtua dan keturunan
245             combined_population = np.vstack([population,
                    offspring])
246             combined_objectives = self._evaluate_objectives(
                    combined_population)
247
248             # Seleksi lingkungan
249             combined_fronts, combined_ranks = self.
                    _fast_non_dominated_sort(combined_objectives)
250
251             new_population = []
252             front_idx = 0
253
254             # Tambah front lengkap
255             while (len(new_population) + len(combined_fronts[
                    front_idx]) <= self.population_size):
256                 for individual_idx in combined_fronts[front_idx]:
257                     new_population.append(individual_idx)
258                 front_idx += 1
259
260                 if front_idx >= len(combined_fronts):
261                     break
262
263             # Tambah front parsial jika diperlukan
264             if len(new_population) < self.population_size and
                    front_idx < len(combined_fronts):
265                 last_front = combined_fronts[front_idx]
266                 crowding_distances = self.
                        _calculate_crowding_distance(
                        combined_objectives, last_front)
267
268                 # Urutkan berdasarkan crowding distance (menurun)
269                 sorted_indices = sorted(range(len(last_front)),
270                                         key=lambda x:
                                            crowding_distances[x],
                                            reverse=True)
271
272                 remaining_slots = self.population_size - len(
                        new_population)
273                 for i in range(remaining_slots):
274                     new_population.append(last_front[
                            sorted_indices[i]])
275
276             # Perbarui populasi
277             population = combined_population[new_population]
278
279         # Evaluasi akhir dan kembalikan front Pareto
280         final_objectives = self._evaluate_objectives(population)
281         fronts, _ = self._fast_non_dominated_sort(
                final_objectives)
```

```
282
283         pareto_front_indices = fronts[0]
284         pareto_front_solutions = population[pareto_front_indices]
285         pareto_front_objectives = final_objectives[
               pareto_front_indices]
286
287         return {
288             'pareto_front_solutions': pareto_front_solutions,
289             'pareto_front_objectives': pareto_front_objectives,
290             'final_population': population,
291             'final_objectives': final_objectives
292         }
293
294 # Example: Minimize two objectives (ZDT1 problem)
295 def objective1(x):
296     return x[0]
297
298 def objective2(x):
299     g = 1 + 9 * np.sum(x[1:]) / (len(x) - 1)
300     h = 1 - np.sqrt(x[0] / g)
301     return g * h
302
303 # Usage
304 if __name__ == "__main__":
305     objectives = [objective1, objective2]
306     bounds = [(0, 1)] * 10  # 10-dimensi
307
308     nsga2 = NSGA2(objectives, 10, bounds, population_size=100)
309     result = nsga2.evolve(generations=250)
310
311     # Plot front Pareto
312     pareto_objectives = result['pareto_front_objectives']
313     plt.figure(figsize=(10, 6))
314     plt.scatter(pareto_objectives[:, 0], pareto_objectives[:, 1],
315                 c='red', alpha=0.7)
316     plt.xlabel('Objektif␣1')
317     plt.ylabel('Objektif␣2')
318     plt.title('Front␣Pareto')
319     plt.grid(True, alpha=0.3)
320     plt.show()
```

Appendix B
Contoh Praktis dan Studi Kasus

B.1 Masalah Optimasi Fungsi

B.1.1 OneMax

OneMax adalah contoh sederhana untuk algoritma genetika biner. Tujuan: maksimalkan jumlah bit 1 dalam kromosom.

$$f(x) = \sum_{i=1}^{n} x_i \tag{B.1}$$

B.1.2 Beberapa Fungsi Benchmark

- **Sphere**: $f(\mathbf{x}) = \sum_{i=1}^{n} x_i^2$, domain $[-5.12, 5.12]^n$, minimum global di $\mathbf{0}$.
- **Rastrigin**: multimodal, $f(\mathbf{x}) = An + \sum[x_i^2 - A\cos(2\pi x_i)]$, $A = 10$.
- **Rosenbrock**: lembah sempit, $f(\mathbf{x}) = \sum[100(x_{i+1} - x_i^2)^2 + (1 - x_i)^2]$.

B.2 Masalah Optimasi Kombinatorial

B.2.1 TSP

Traveling Salesman Problem: temukan rute terpendek yang melewati semua kota sekali dan kembali ke awal. Pada GA digunakan encoding permutasi dan operator khusus (PMX, OX), serta langkah perbaikan lokal.

B.2.2 Knapsack 0/1

Pilih item untuk memaksimalkan nilai dengan batas bobot W:

$$\max \sum_{i=1}^{n} v_i x_i \tag{B.2}$$

$$exts.t. \sum_{i=1}^{n} w_i x_i \leq W, \quad x_i \in \{0, 1\}. \tag{B.3}$$

B.3 Aplikasi Dunia Nyata

B.3.1 Pelatihan Jaringan Syaraf

GA dapat mengoptimalkan bobot atau arsitektur jaringan dengan encoding vektor real; contoh metrik: $fitness = 1/(1 + MSE)$.

B.3.2 Seleksi Fitur

Encoding biner memungkinkan GA memilih subset fitur. Tujuan sering kali multi-objektif: maksimalkan performa sambil minimalkan jumlah fitur.

B.3.3 Penjadwalan

Job shop scheduling dapat direpresentasikan dengan permutasi atau prioritas; GA sering digabungkan dengan pencarian lokal (2-opt, 3-opt).

B.4 Panduan Penyetelan Singkat

- Ukuran populasi: sederhana 50–100, sedang 100–500, kompleks 500–2000.
- Crossover: umum 0.7–0.9.
- Mutasi: binary $1/L$, real 0.01–0.1.
- Seleksi: turnamen (size 2–7) untuk mengatur tekanan seleksi.

B.5 Analisis Performa

Gunakan metrik: best/average fitness, keberagaman, success rate. Jalankan 20–30 trial independen, laporkan mean, std, best, worst, dan lakukan uji statistik bila perlu.

B.6 Kendala Umum dan Solusi

- Konvergensi dini: tingkatkan populasi, kurangi tekanan seleksi, tingkatkan mutasi, gunakan mekanisme pelestarian keberagaman.
- Konvergensi lambat: tingkatkan tekanan seleksi, tambahkan pencarian lokal.
- Kendala: gunakan penalti adaptif, repair, atau pendekatan multi-objektif.

B.7 Teknik Lanjutan

Hibridisasi (memetic algorithms), kontrol parameter adaptif/self-adaptive, dan paralelisasi (master-slave, island model, cellular GA) sering meningkatkan performa.

B.8 Ringkasan

Ringkasan singkat: desain representasi, tuning parameter, validasi statistik, dan penggunaan teknik hibrida adalah kunci keberhasilan penerapan GA.

- Sedikit perbaikan selama banyak generasi
- Keberagaman populasi tetap tinggi

- Perilaku seperti random walk

extbfSolusi:

- Tingkatkan tekanan seleksi
- Kurangi laju mutasi
- Terapkan pencarian lokal (GA hibrida)
- Gunakan inisialisasi yang lebih baik
- Sesuaikan operator crossover

B.8.1 Masalah Penanganan Kendala

extbfMasalah Umum:

- Semua individu melanggar kendala
- Wilayah feasibel terlalu kecil
- Koefisien penalti disetel buruk

extbfSolusi:

- Gunakan mekanisme perbaikan
- Terapkan operator khusus
- Implementasikan pelestarian feasibilitas
- Gunakan pendekatan multi-objektif
- Sesuaikan bobot penalti secara dinamis

B.9 Teknik Lanjutan

B.9.1 Genetic Algorithms Hibrida

Gabungkan GA dengan metode pencarian lokal:

- **Memetic algorithms**: GA + pencarian lokal
- **Evolusi Lamarckian**: Mewariskan solusi yang telah diperbaiki
- **Evolusi Baldwinian**: Gunakan pencarian lokal hanya untuk evaluasi fitness

B.9.2 Kontrol Parameter Adaptif

Sesuaikan parameter GA secara otomatis selama evolusi:

- **Deterministik**: Jadwal pra-definisi
- **Adaptif**: Berdasarkan keadaan populasi
- **Self-adaptive**: Parameter berevolusi bersama populasi

B.9.3 Parallel Genetic Algorithms

Distribusikan komputasi ke beberapa prosesor:

- **Master-slave**: Evaluasi fitness paralel
- **Island model**: Beberapa populasi dengan migrasi
- **Cellular GA**: Struktur populasi spasial

B.10 Praktik Terbaik Implementasi

B.10.1 Organisasi Kode

- Pisahkan representasi dari operator
- Gunakan desain modular untuk kemudahan pengujian
- Implementasikan generator bilangan acak yang baik
- Tambahkan logging dan kemampuan visualisasi

B.10.2 Pengujian dan Validasi

- Uji pada masalah benchmark yang dikenal
- Verifikasi operator menjaga validitas solusi
- Periksa kualitas generator bilangan acak
- Profil dulu hambatan performa

B.10.3 Dokumentasi

- Dokumentasikan pilihan parameter dan alasannya
- Catat detail pengaturan eksperimen
- Pertahankan kontrol versi
- Bagikan hasil yang dapat direproduksi

B.11 Ringkasan Bab

Bab ini menyajikan contoh praktis dan studi kasus yang menunjukkan penerapan algoritma genetika pada berbagai domain masalah. Pelajaran penting meliputi pentingnya desain representasi yang tepat, penyetelan parameter, dan analisis performa. Memahami kendala umum dan solusinya sangat krusial untuk implementasi GA yang berhasil.

B.12 Poin Penting

- Representasi masalah sangat menentukan keberhasilan GA
- Pengaturan parameter harus sesuai karakteristik masalah
- Validasi statistik memastikan hasil yang dapat dipercaya
- Pendekatan hibrida seringkali mengungguli GA murni
- Pengetahuan domain harus memandu desain operator
- Pengujian dan dokumentasi yang baik adalah esensial

Bibliography

[1] Course material week 4 - crossover. Course material.

[2] Course material week 9 - mutation and update generation. Course material.

[3] Selection - introduction to genetic algorithms - tutorial with interactive java applets. https://www.obitko.com/tutorials/genetic-algorithms/selection.php. Retrieved September 30, 2025.

[4] Algorithm Afternoon. Chapter 4 - selection strategies. https://algorithmafternoon.com/books/genetic_algorithm/chapter04/. Retrieved September 30, 2025.

[5] Algorithm Afternoon. Ranked selection genetic algorithm. https://algorithmafternoon.com/genetic/ranked_selection_genetic_algorithm/. Retrieved September 30, 2025.

[6] Thomas Bäck. *Evolutionary algorithms in theory and practice: evolution strategies, evolutionary programming, genetic algorithms*. Oxford University Press, Oxford, UK, 1996.

[7] Baeldung on Computer Science. Tournament selection in genetic algorithms. https://www.baeldung.com/cs/ga-tournament-selection. Retrieved September 30, 2025.

[8] James E Baker. Reducing bias and inefficiency in the selection algorithm. *Proceedings of the second international conference on genetic algorithms*, pages 14–21, 1987.

[9] Shih-Hsin Chen, Min-Chih Chen, Pei-Chann Chang, and V. Mani. Multiple parents crossover operators: A new approach removes the overlapping solutions for sequencing problems. *Applied Mathematical Modelling*, 37(5):2737–2746, 2013.

[10] Kenneth A De Jong. An analysis of the behavior of a class of genetic adaptive systems. 1975. PhD thesis.

[11] Kalyanmoy Deb. *Multi-objective optimization using evolutionary algorithms*. John Wiley & Sons, Chichester, UK, 2001.

[12] Kalyanmoy Deb, Amrit Pratap, Sameer Agarwal, and TAMT Meyarivan. A fast and elitist multiobjective genetic algorithm: Nsga-ii. *IEEE transactions on evolutionary computation*, 6(2):182–197, 2002.

[13] Agoston E Eiben and James E Smith. *Introduction to evolutionary computing*. Springer, 2nd edition, 2015.

[14] S. M. Elsayed, R. A. Sarker, and D. L. Essam. GA with a new multi-parent crossover for constrained optimization. In *2011 IEEE Congress of Evolutionary Computation (CEC)*, pages 857–864, New Orleans, LA, USA, 2011.

[15] Larry J Eshelman. The chc adaptive search algorithm: How to have safe search when engaging in nontraditional genetic recombination. *Foundations of genetic algorithms*, 1:265–283, 1991.

[16] A. M. Fajrin and C. Fatichah. Multi-parent order crossover mechanism of genetic algorithm for minimizing violation of soft constraint on course timetabling problem. *Register: Jurnal Ilmiah Teknologi Sistem Informasi*, 6(1):43–51, 2020.

[17] David B Fogel. Evolutionary programming: an introduction and some current directions. *Statistics and computing*, 5(2):103–109, 1995.

[18] David B Fogel. *Evolutionary computation: toward a new philosophy of machine intelligence*. John Wiley & Sons, 3rd edition, 2006.

[19] GeeksforGeeks. Crossover in genetic algorithm. https://www.geeksforgeeks.org/machine-learning/crossover-in-genetic-algorithm/. Retrieved November 3, 2025.

[20] Mitsuo Gen and Runwei Cheng. *Genetic algorithms and engineering optimization*. Wiley-Interscience, 2007.

[21] David E Goldberg. *Genetic algorithms in search, optimization, and machine learning*. Addison-Wesley, Reading, MA, 1989.

[22] John J Grefenstette. Optimization of control parameters for genetic algorithms. *IEEE Transactions on systems, man, and cybernetics*, 16(1):122–128, 1986.

[23] H. Gu, H. C. Lam, and Y. Zinder. A hybrid genetic algorithm for scheduling jobs sharing multiple resources under uncertainty. *EURO Journal on Computational Optimization*, 10:100050, 2022.

[24] Randy L Haupt and Sue Ellen Haupt. Practical genetic algorithms. 2004.

[25] John H Holland. *Adaptation in natural and artificial systems*. University of Michigan Press, Ann Arbor, MI, 1975.

[26] Jeffrey Horn, Nicholas Nafpliotis, and David E Goldberg. A niched pareto genetic algorithm for multiobjective optimization. *Proceedings of the first IEEE conference on evolutionary computation*, pages 82–87, 1994.

[27] John R Koza. *Genetic programming: on the programming of computers by means of natural selection*. MIT Press, Cambridge, MA, 1992.

[28] Pedro Larrañaga, Cindy MH Kuijpers, Roberto H Murga, Iñaki Inza, and Sejla Dizdarevic. Genetic algorithms for the travelling salesman problem: a review of representations and operators. *Artificial intelligence review*, 13(2):129–170, 1999.

[29] N. Majhi and R. Mishra. A novel hybrid genetic algorithm and nelder-mead approach and it's application for parameter estimation. *F1000Research*, 13:1073, 2025.

[30] Zbigniew Michalewicz. Genetic algorithms+ data structures= evolution programs. 1996.

[31] Melanie Mitchell. *An introduction to genetic algorithms*. MIT Press, Cambridge, MA, 1996.

[32] David J Montana and Lawrence Davis. Training feedforward neural networks using genetic algorithms. *Proceedings of the Eleventh international joint conference on Artificial intelligence*, 1:762–767, 1989.

[33] S. H. Murad, N. B. Tayfor, N. H. Mahmood, and L. Arman. Hybrid genetic algorithms-driven optimization of machine learning models for heart disease prediction. *MethodsX*, 15:103510, 2025.

[34] IM Oliver, DJ Smith, and John RC Holland. A study of permutation crossover operators on the traveling salesman problem. *Proceedings of the Second International Conference on Genetic Algorithms*, pages 224–230, 1987.

[35] Colin R Reeves. *Modern heuristic techniques for combinatorial problems*. John Wiley & Sons, New York, 1993.

[36] J David Schaffer, Rich A Caruana, Larry J Eshelman, and Rajarshi Das. A study of control parameters affecting online performance of genetic algorithms for function optimization. *Proceedings of the third international conference on genetic algorithms*, pages 51–60, 1989.

[37] E. Shams. Resolving the exploration-exploitation dilemma in evolutionary algorithms: A novel human-centered framework. *arXiv preprint*, 2025.

[38] S. N. Sivanandam and S. N. Deepa. *Introduction to genetic algorithms*. Springer, 2008.

[39] J. Smith and F. Vavak. Replacement strategies in steady state genetic algorithms: Static environments. pages 219–234, 1998.

[40] William M Spears. Crossover or mutation? *Foundations of genetic algorithms*, 2:221–237, 1993.

[41] Mandavilli Srinivas and Lalit M Patnaik. Genetic algorithms: a survey. *Computer*, 27(6):17–26, 1994.

[42] Tutorialspoint. Genetic algorithms - crossover. https://www.tutorialspoint.com/genetic_algorithms/genetic_algorithms_crossover.htm. Retrieved November 3, 2025.

[43] Tutorialspoint. Genetic algorithms - mutation. https://www.tutorialspoint.com/genetic_algorithms/genetic_algorithms_mutation.htm. Retrieved November 22, 2025.

[44] Darrell Whitley. A genetic algorithm tutorial. *Statistics and computing*, 4(2):65–85, 1994.

[45] David H Wolpert and William G Macready. No free lunch theorems for optimization. *IEEE transactions on evolutionary computation*, 1(1):67–82, 1997.

[46] E. T. Yassen, M. Ayob, M. Z. A. Nazri, and N. R. Sabar. Multi-parent insertion crossover for vehicle routing problem with time windows. In *2012 4th Conference on Data Mining and Optimization (DMO)*, pages 103–108, Langkawi, Malaysia, 2012.

[47] Betul Sultan Yıldız, S. Kumar, Natee Panagant, P. Mehta, S. M. Sait, Ali Riza Yıldız, Nantiwat Pholdee, Sujin Bureerat, and Seyedali Mirjalili. A novel hybrid arithmetic optimization algorithm for solving constrained optimization problems. *Knowledge-Based Systems*, 271:110554, 2023.