Algoritma Genetika

Teori Dasar Kuliah

Panduan komprehensif tentang Algoritma Genetika

Nugroho Adi Susanto : 23/520312/PA/22367

30 Nopember 2025

Daftar Isi

Daftar Gambar

Daftar Tabel

Bab 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 [20].

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:

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

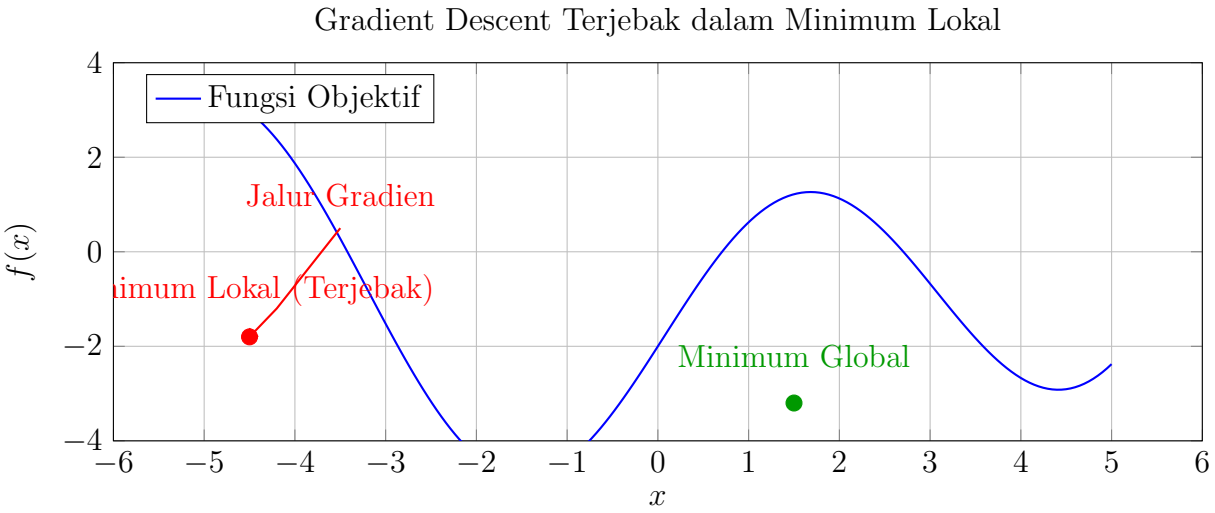
di mana:

- $f(x)$ 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

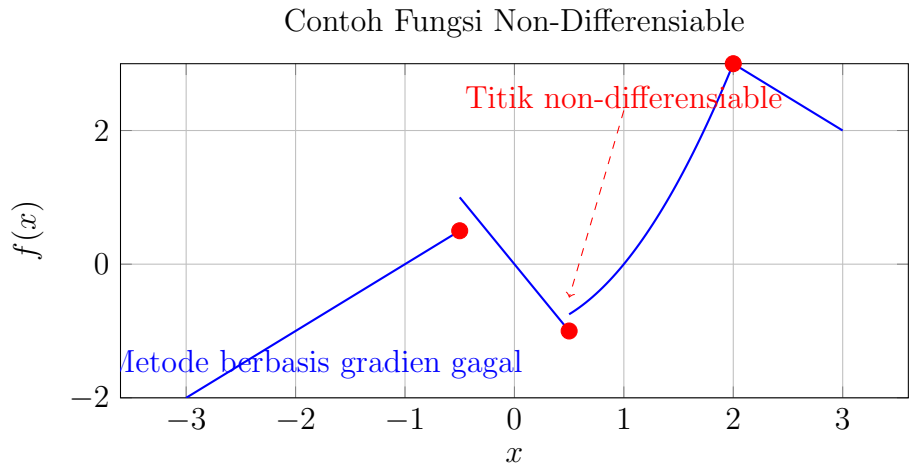
Simplex dan interior-point) untuk model linear, dan metode diskrit (branch-and-bound, pemrograman dinamis) untuk masalah kombinatorial. Namun, metode tradisional ini memiliki keterbatasan ketika diterapkan pada banyak masalah dunia nyata yang kompleks. Dalam bagian berikut kami menyoroti tiga tantangan 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.



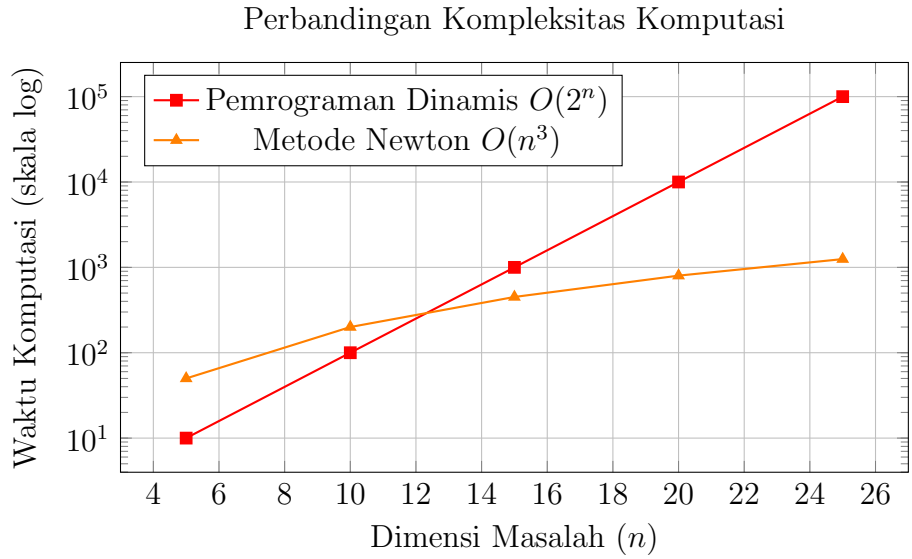
Gambar 1.1: Metode berbasis gradien tradisional mengikuti gradien lokal dan menjadi terjebak dalam optimum lokal, tidak mampu melarikan diri untuk menemukan optimum global.

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.



Gambar 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.



Gambar 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

Gambar 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 [24, 12]. 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.

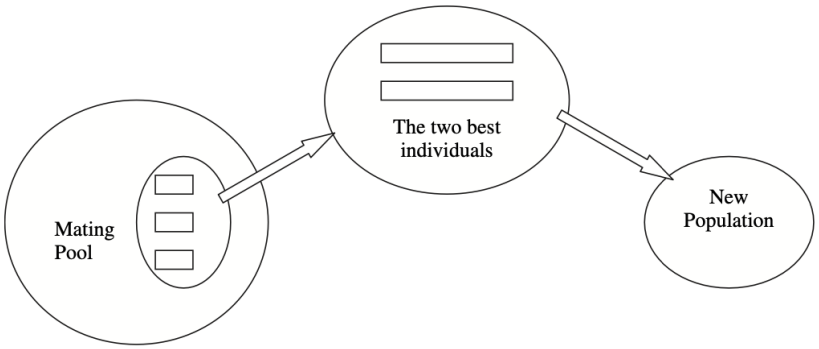
Bidang komputasi evolusioner mencakup beberapa keluarga yang mapan, masing-masing menekankan pilihan desain yang berbeda. Algoritma Genetika (GA) adalah salah satu formulasi paling awal dan paling berpengaruh, menekankan pengkodean panjang tetap dan operator crossover dan mutasi yang terinspirasi secara biologis [24, 20]. 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 [16, 17]. Pemrograman Genetika (GP) memperluas kerangka kerja ke struktur panjang variabel seperti program komputer dan pohon ekspresi, memungkinkan sintesis program otomatis [26].

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 [46]. 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 [31, 32]. Penjadwalan, pembuatan jadwal waktu, perutean (termasuk Travelling Salesman Problem), dan masalah kombinatorial la-

innya adalah aplikasi alami karena pengkodean yang fleksibel dan operator rekombinasi khusus yang menghormati struktur masalah [22, 15]. Di luar domain ini, komputasi evolusioner telah menemukan peran dalam bioinformatika, pemodelan keuangan, dan desain strategi permainan otomatis, menunjukkan utilitas praktis yang luas [19, 37, 12].

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.



Gambar 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 [28, 32].

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 [36, 40].

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 [12].

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 [28, 32].
- **GA Adaptif:** Secara dinamis menyesuaikan parameter seperti probabilitas crossover dan mutasi berdasarkan umpan balik populasi untuk menyeimbangkan eksplorasi dan eksploitasi [36, 40].
- **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.

Bab 2
Apa itu Algoritma Genetika?

2.1 Pendahuluan

Algoritma Genetika (GA) adalah metode pencarian dan optimasi acak yang mengambil inspirasi dari evolusi alami [24, 20]. 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

Tabel 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 [24, 30].

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 [12].

2.3 Terminologi Dasar

2.3.1 Istilah Algoritma Genetika

Memahami terminologi umum membantu menjembatani metafora biologis dan implementasi algoritmiknya [30, 20]. 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 [12].

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 [24, 30, 12].

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 [20, 12].

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 [30, 20].

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 [20]. 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 [12].

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 [37].

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 [30].

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 [23]. 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 [30].

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 [12].

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 [20].

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 [12, 23].

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 [30, 37].

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 [11, 12].

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.

Bab 3
Siklus GA dan Teori Skema Holland

3.1 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:

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

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}.

3.1.1 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:

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

3.1.2 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:

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

3.1.3 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 Implikasi Praktis

3.3.1 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.

3.3.2 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.

3.4 Keterbatasan Teori Skema

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

3.4.1 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.

3.4.2 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.

3.4.3 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.

3.4.4 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.

3.4.5 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.

3.5 Teorema No Free Lunch

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

Bab 4
Encoding pada Algoritma Genetika

4.1 Pendahuluan terhadap Encoding

Encoding (atau representasi) merumuskan bagaimana solusi calon dinyatakan untuk sebuah algoritma genetik (AG). Misalkan G adalah himpunan diskrit genotipe (ruang representasi) dan P adalah himpunan fenotipe (ruang solusi). Encoding didefinisikan oleh pemetaan

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

yang menetapkan untuk setiap genotipe sebuah fenotipe yang dapat dievaluasi oleh fungsi kecocokan (fitness). Dalam praktiknya G biasanya merupakan ruang kombinatorial hingga atau terhitung (misalnya rangkaian bit, vektor bilangan bulat, permutasi, tree, atau vektor bernilai riil) dan P adalah domain solusi masalah (misalnya vektor riil, jadwal, tur, atau program).

Ada dua aspek encoding yang perlu dibedakan: (i) bahasa representasi yang dipakai untuk membentuk genotipe (bit, integer, bilangan riil, node pada tree, dll.), dan (ii) pemetaan genotipe–fenotipe ϕ. Pencarian yang dilakukan AG berlangsung pada ruang G, sedangkan fungsi kecocokan dan kendala masalah didefinisikan pada P; dengan demikian sifat–sifat ϕ sangat menentukan bagaimana variasi pada ruang genotipe diterjemahkan menjadi perubahan bermakna pada kualitas solusi.

Encoding yang dipilih dengan baik memperlihatkan struktur yang dapat dimanfaatkan prosedur pencarian, mengurangi kemunculan solusi tak layak, serta mengendalikan redundansi representasi dan epistasis. Encoding yang buruk dapat membuat perbaikan lokal tidak terlihat oleh variasi, menghasilkan lanskap kecocokan patologis, atau mengharuskan prosedur perbaikan yang mahal. Pada bagian selanjutnya kita akan membahas keluarga encoding konkret (biner, Gray, bernilai riil, permutasi, tree) serta implikasi praktisnya terhadap desain representasi dan performa algoritma.

4.2 Persyaratan untuk Encoding yang Baik

Saat merancang encoding dan pemetaan $\phi : G \to P$, berguna untuk merumuskan kriteria secara tegas. Properti-properti berikut menangkap kebutuhan representasional inti dan trade-off; mereka membantu memilih atau menyusun encoding untuk suatu masalah tertentu.

4.2.1 Kelengkapan

Kelengkapan mensyaratkan agar encoding mampu menyatakan setiap fenotipe layak yang relevan: secara formal citra ϕ harus mencakup daerah layak $F \subseteq P$, yakni $\phi(G) \supseteq F$. Jika kelengkapan gagal maka beberapa solusi valid tidak dapat dijangkau oleh AG, yang memperkenalkan bias representasional dan dapat mencegah algoritma menemukan solusi optimal di luar $\phi(G)$.

Dalam praktiknya kelengkapan diseimbangkan dengan kepadatan representasi: encoding yang sepenuhnya lengkap mungkin besar atau tidak efisien, sedangkan encoding terbatas dapat menyederhanakan pencarian jika mengecualikan bagian–bagian P yang tidak menarik. Perancang harus menyatakan secara eksplisit subset P mana yang harus dapat dicapai dan memastikan $\phi(G)$ memuatnya.

4.2.2 Kebenaran (Soundness)

Kebenaran atau validitas menyatakan bahwa setiap genotipe harus dipetakan ke fenotipe yang terdefinisi dengan baik dan memenuhi kendala: $\forall g \in G,\ \phi(g) \in P_{valid}$. Encoding yang benar menghindari atau meminimalkan produksi solusi tak layak sehingga evaluasi kecocokan bermakna tanpa perbaikan mahal. Jika kebenaran ketat tidak mungkin, perancang boleh mengizinkan genotipe tak layak tetapi harus menyediakan dekoding dan strategi perbaikan yang efisien dan terdokumentasi sehingga pencarian tetap dapat berlangsung.

Kebenaran dan kelengkapan bersifat ortogonal: sebuah encoding bisa benar tetapi tidak lengkap (setiap genotipe valid, tetapi tidak semua fenotipe dapat direpresentasikan), atau lengkap tetapi tidak benar (semua fenotipe representable tetapi banyak genotipe tidak valid), tergantung pada G dan ϕ.

4.2.3 Non-redundansi

Non-redundansi berarti mengurangi (atau menghilangkan) banyak genotipe berbeda yang dipetakan pada fenotipe yang sama. Secara formal, diinginkan agar ϕ bersifat injektif pada himpunan genotipe yang relevan. Redundansi (pemetaan many-to-one) meningkatkan volume pencarian efektif dan dapat mem-bias sampling: beberapa fenotipe mungkin over-represented di G, sehingga lebih sering terjadi sampling meskipun tidak superior.

Namun, redundansi kadang sengaja diperkenalkan demi robustitas (misalnya jaringan netral yang memungkinkan drift netral) atau untuk menyederhanakan representasi. Jika redundansi ada, kuantifikasilah derajatnya dan pertimbangkan interaksinya dengan dinamika pencarian dan operator variasi.

4.2.4 Lokalitas

Lokalitas merumuskan intuisi bahwa perubahan genotipe kecil seharusnya menghasilkan perubahan fenotipe kecil. Misalkan d_G dan d_P adalah ukuran jarak pada G dan P masing-masing (mis. jarak Hamming pada bit-string, jarak Euclidean pada vektor riil). Lokalitas tinggi berarti

$$d_G(g_1, g_2) \text{ kecil} \Rightarrow d_P(\phi(g_1), \phi(g_2)) \text{ juga kecil.}$$

Lokalitas penting karena prosedur variasi umum membuat perubahan kecil di G; bila perubahan ini tidak berkorelasi dengan perubahan kecil di P, pencarian menjadi pada dasarnya acak dan rekombinasi building-block gagal. Pilihan encoding seperti Gray coding untuk bilangan bulat atau representasi bernilai riil bertujuan meningkatkan lokalitas.

Lokalitas tidak selalu bisa dicapai bersamaan dengan properti lain yang diinginkan; misalnya encoding injektif dan padat dengan lokalitas sempurna mungkin tidak ada untuk beberapa domain kombinatorial. Perancang harus memprioritaskan properti yang paling penting untuk masalah dan prosedur yang akan dipakai.

4.2.5 Persyaratan Praktis Tambahan

Selain empat properti formal di atas, encoding yang berguna juga harus memenuhi beberapa kendala pragmatis:

- **Operator Closure:** Prosedur variasi sedapat mungkin menghasilkan genotipe di kawasan G yang didekode menjadi fenotipe layak atau mudah diperbaiki.
- **Efisiensi Komputasi:** Dekoding ϕ dan prosedur perbaikan harus murah secara komputasi relatif terhadap evaluasi kecocokan.
- **Skalabilitas:** Encoding harus skala secara wajar dengan ukuran masalah; panjang representasi tidak boleh tumbuh supra-linear tanpa alasan.
- **Epistasis Rendah:** Representasi sebaiknya meminimalkan interaksi destruktif antar gen (epistasis) sehingga building-block yang menguntungkan dapat direkombinasikan secara andal.
- **Interpretabilitas dan Pengetahuan Awal:** Bila tersedia, masukkan struktur spesifik domain (simetri, invarian, kendala) untuk menyederhanakan pencarian dan mengurangi derajat kebebasan yang tidak perlu.

Merancang encoding adalah soal memenuhi persyaratan formal, kompatibilitas prosedur, dan validasi empiris. Pada bagian berikut akan dibahas keluarga encoding umum dan trade-off praktis yang harus dipertimbangkan.

4.3 Encoding Biner

Encoding biner merepresentasikan genotipe sebagai vektor berdimensi tetap atas alfabet biner: $G = \{0, 1\}^l$. Genotipe $g = (b_{l-1}, \ldots, b_0)$ sering diinterpretasikan sebagai bilangan unsigned

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

yang kemudian dipetakan ke fenotipe melalui dekoding afine bila fenotipe berskala numerik. Untuk variabel bernilai riil $x \in [x_{\min}, x_{\max}]$ dekoding yang umum adalah

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

Rumus ini menjelaskan resolusi representasi: langkah kuantisasi adalah

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

sehingga memilih l mempertukarkan presisi dengan dimensionalitas pencarian dan perilaku prosedur variasi.

Encoding biner menarik karena ringkas dan mudah dimanipulasi dengan operasi bitwise. Schema theory dan banyak hasil teoretis awal dikembangkan untuk representasi biner, yang membantu penalaran teoretis tentang konvergensi dan propagasi building-block [24, 20].

Namun, encoding biner juga memperkenalkan masalah khusus yang perlu ditangani:

- **Hamming cliffs dan lokalitas:** Nilai numerik berdekatan dapat berbeda di banyak bit pada encoding posisi biner standar, sehingga merusak lokalitas. Kode Gray adalah obat umum bila menjaga kedekatan penting.
- **Presisi versus panjang:** Presisi tinggi membutuhkan bit-string panjang, yang memperluas ruang pencarian secara eksponensial dan dapat membuat rekombinasi posisional menjadi sangat disruptif.
- **Epistasis:** Posisi bit dapat berinteraksi secara non-linier terhadap kualitas fenotipe; bit yang berkorelasi mengurangi efektivitas rekombinasi sederhana.

Rekomendasi praktis untuk encoding biner:

- Pilih panjang l dari resolusi yang diinginkan Δ dan rentang $[x_{\min}, x_{\max}]$ menggunakan aturan $2^l - 1 \geq (x_{\max} - x_{\min})/\Delta$.
- Jika kedekatan nilai penting, pertimbangkan Gray coding untuk variabel integer dan konversi ke biner hanya untuk prosedur yang memang bekerja pada bitstring.
- Gunakan prosedur yang menghormati batas gen saat menggabungkan beberapa variabel (mis. sesuaikan titik rekombinasi ke batas variabel bila relevan).
- Stemper laju modifikasi per-bit sebagai heuristik awal; kurangi bila menggunakan pencarian lokal atau dinamika seleksi yang kuat.

4.4 Ikhtisar Jenis Encoding

Encoding dapat dikelompokkan menurut struktur G dan domain fenotipe yang dimaksud P. Di bawah ini kami merangkum keluarga utama dan kasus penggunaan kanoniknya, disertai panduan praktis dan jebakan umum.

Taksonomi dan pemetaan ke kelas masalah

- **Biner (bit-string):** $G = \{0, 1\}^l$. Baik untuk pilihan kombinatorial dan bila analisis schema diinginkan. Gunakan Gray code atau penyusunan bit spesifik masalah untuk meningkatkan lokalitas pada fenotipe numerik.
- **Integer:** Vektor bilangan bulat; alami untuk masalah alokasi dan penghitungan. Gunakan prosedur variasi yang sadar-integer (random-reset, creep) dan metode rekombinasi diskrit.
- **Bernilai riil (real-valued):** Vektor kontinu $\mathbb{R}^n$. Direkomendasikan untuk optimisasi kontinu; mendukung kombinasi aritmetik, rekombinasi BLX-α, gangguan stokastik, dan hibrida berbasis gradien.
- **Permutasi:** Merepresentasikan pengurutan (TSP, penjadwalan). Memerlukan prosedur variasi khusus yang mempertahankan validitas permutasi.
- **Pohon dan graf:** Struktur berukuran variabel untuk pemrograman genetik, evolusi ekspresi, atau topologi rangkaian. Gunakan pertukaran subtree dan kontrol pertumbuhan untuk menghindari bloat.

- **Indirek / perkembangan (developmental):** Genotipe menetapkan aturan konstruksi atau tata bahasa yang menghasilkan fenotipe; berguna bila genotipe ringkas harus menghasilkan fenotipe terstruktur (mis. arsitektur neural, L-systems).

Memilih encoding

Pilih encoding dengan mencocokkan struktur kombinatorial masalah, himpunan kendala, dan toolkit prosedur yang diinginkan. Pertanyaan kunci:

- Apakah masalah membutuhkan pengurutan, multiset, atau parameter bernilai riil? Pilih permutasi, integer/multiset, atau encoding riil secara berturut-turut.
- Apakah kendala kelayakan bersifat keras (harus dipenuhi) atau lunak (pelanggaran diberi penalti)? Untuk kendala keras, utamakan encoding benar atau decoder konstruktif; untuk kendala lunak penalti mungkin dapat diterima.
- Apakah lokalitas penting untuk rekombinasi efektif? Jika ya, pilih encoding atau transformasi (mis. Gray) yang meningkatkan korelasi antara perubahan genotipe kecil dan perubahan fenotipe.
- Apakah prosedur akan kustom atau standar? Gunakan encoding yang menyederhanakan implementasi prosedur kecuali struktur domain mengharuskan sebaliknya.

Kompatibilitas operator dan validasi empiris

Sebuah encoding berguna hanya jika dipasangkan dengan prosedur yang mempertahankan struktur yang berguna. Setelah memilih encoding, rancang atau pilih prosedur variasi yang menjaga kelayakan, membatasi epistasis destruktif, dan menghormati batas gen yang bermakna. Akhirnya, validasi pilihan encoding secara empiris: bandingkan performa pada benchmark kecil (berbagai encoding, set prosedur, dan laju variasi) dan pilih kombinasi yang menunjukkan kemajuan robust pada instance representatif.

Bagian berikut memberikan contoh representasional konkret dan referensi untuk keluarga encoding utama yang dibahas di sini.

4.5 Encoding Bernilai Riil

Encoding bernilai riil merepresentasikan individu sebagai vektor di $\mathbb{R}^n$, yaitu $\mathbf{x} = (x_1, \ldots, x_n)$ dengan setiap koordinat mengambil nilai pada domain kontinu. Representasi langsung ini merupakan pilihan alami untuk masalah optimisasi kontinu dan tugas penyetelan parameter dimana fenotipe secara inheren numerik. Dengan beroperasi dalam ruang kontinu, encoding riil menghindari artefak kuantisasi dari encoding biner berdimensi tetap dan memungkinkan operator variasi mengekspresikan penyesuaian yang sangat kecil pada solusi kandidat (tergantung batas presisi floating-point) [6, 29].

Keuntungan praktis utama encoding riil adalah kompatibilitas operator: rekombinasi aritmetik (rata-rata berbobot), rekombinasi interval BLX-α, simulated binary crossover (SBX), serta mutasi dengan gangguan Gaussian atau Cauchy bekerja alami pada vektor riil dan dapat dirancang untuk menghormati batas atau struktur yang diketahui. Operator ini menghasilkan keturunan yang berada dalam kerangka cembung (atau perpanjangan terkendali) dari orang tua, yang biasanya memberikan lintasan pencarian lebih mulus dan

eksploitasi gradien lokal yang lebih baik pada lanskap kecocokan. Evolution Strategies (ES) dan banyak optimiser kontinu modern memanfaatkan properti ini dengan menggabungkan kontrol langkah adaptif mandiri dengan rekombinasi untuk menavigasi lanskap yang kasar namun dapat diturunkan secara efisien [6].

Namun demikian ada trade-off teoretis dan praktis. Argumen schema klasik untuk representasi biner tidak langsung berlaku untuk encoding kontinu: konsep building-block harus diformulasikan ulang dalam istilah daerah di $\mathbb{R}^n$ dan korelasi yang diinduksi operator antar koordinat. Encoding riil juga menempatkan penekanan lebih pada pilihan algoritmik untuk kontrol ukuran langkah dan penanganan kendala — skala mutasi yang buruk atau rekombinasi tak terbatas dapat menyebabkan kemajuan lambat atau ketidakstabilan numerik. Oleh karena itu praktisi harus menyetel atau mengadaptasi magnitudo mutasi (jadwal tetap, adaptasi mandiri, atau adaptasi matriks kovarians) dan memilih parameter rekombinasi yang cocok dengan kelancaran dan skala masalah.

Dari sisi implementasi, beberapa rekomendasi pragmatis meningkatkan robustitas dan performa. Selalu normalisasi atau skala variabel ke rentang yang sebanding sebelum menerapkan operator generik; ini mencegah koordinat tunggal mendominasi statistik rekombinasi dan menyederhanakan transfer parameter antar masalah. Gunakan operator terbatas atau skema proyeksi bila ada kendala, dan pilih strategi mutasi adaptif (mis. adaptasi ukuran langkah log-normal atau pembaruan kovarians ala CMA) bila lanskap pencarian menunjukkan anisotropi. Bila gradien lokal tersedia atau dapat didekati murah, menggabungkan pembaruan evolusioner dengan penyempurnaan berbasis gradien sering mempercepat konvergensi sambil mempertahankan eksplorasi global.

Seperti pilihan encoding lainnya, representasi riil harus divalidasi secara empiris terhadap alternatif. Untuk banyak masalah kontinu yang mulus dan berdimensi rendah hingga sedang, encoding riil secara substansial mengungguli encoding biner baik dari segi kecepatan konvergensi maupun kualitas solusi akhir; untuk masalah sangat multimodal atau kombinatorial, parameterisasi riil mungkin tidak sesuai. Oleh karena itu disarankan memulai dengan set operator riil sederhana (rekombinasi aritmetik/BLX dan mutasi Gaussian dengan deviasi baku yang disetel), lakukan eksperimen faktorial kecil untuk memilih mekanisme adaptasi, dan gunakan adaptasi lebih canggih (ukuran langkah adaptif, CMA) bila skala masalah atau perilaku pencarian memerlukannya.

4.6 Encoding Integer

Encoding integer merepresentasikan solusi yang variabelnya mengambil nilai bilangan bulat diskrit. Secara formal individu adalah vektor $\mathbf{x} = (x_1, \ldots, x_n)$ dengan setiap koordinat $x_i \in \mathbb{Z}$ dan, dalam praktik, dibatasi ke domain hingga $[a_i, b_i] \cap \mathbb{Z}$. Representasi ini sesuai untuk masalah alokasi, jumlah, dan banyak substruktur kombinatorial (mis. kuantitas pada knapsack, alokasi sumber daya, dan parameter kontrol yang didiskretisasi). Sifat diskrit variabel mengubah karakter pencarian: lingkungan (neighbourhood) didefinisikan oleh langkah-integer, dan lanskap pencarian bersifat tidak kontinu dan sering non-konveks.

Operator untuk encoding integer harus menghormati integritas dan batas domain atau kendala kelayakan. Strategi mutasi umum meliputi random-reset (mengganti koordinat dengan integer uniform di domainnya) dan "creep" (menaikkan atau menurunkan dengan langkah kecil yang diambil dari distribusi berekor pendek). Rekombinasi dapat dilakukan langsung di domain integer (mis. discrete uniform crossover atau seleksi per-koordinat), atau dengan mengangkat sementara nilai ke surrogate kontinu (rekombinasi aritmetik diikuti pembulatan) bila operator yang memanfaatkan rata-rata diinginkan. Saat meng-

gunakan rekombinasi kontinu surrogate, pembulatan stokastik atau pembulatan koreksi bias membantu mengurangi artefak pembulatan sistematis.

Encoding integer memiliki trade-off dibandingkan representasi riil. Karena domain diskrit, banyak asumsi analitik (mis. gradien mulus atau konveksitas kontinu) tidak berlaku, dan mekanisme adaptasi ukuran langkah kontinu perlu disesuaikan ke skala langkah diskrit. Di sisi lain, representasi integer dapat menyandikan kelayakan secara langsung, menghindari prosedur perbaikan mahal: mis. merepresentasikan kuantitas dengan integritas memastikan kendala alami, dan operator rekombinasi/mutasi diskrit khusus dapat dirancang untuk mempertahankan kelayakan atau hampir-kelayakan secara konstruktif.

Dari perspektif desain algoritma dan implementasi, beberapa rekomendasi pragmatis meningkatkan robustitas. Pertama, manfaatkan struktur masalah: bila variabel memiliki rentang integer kecil gunakan gerakan neighbourhood enumeratif dan hibrida pencarian lokal langkah-kecil; bila rentang besar, pilih operator yang mengeksplor luas (random-reset, proposal langkah besar) dikombinasikan dengan pengurangan adaptif magnitudo langkah. Kedua, tegakkan batas dan invarian dalam decoder atau melalui proyeksi setelah variasi daripada mengandalkan pemangkasan implisit; operator sadar-kendala eksplisit biasanya lebih jelas dan kurang rawan kesalahan. Ketiga, saat mencampur variabel integer dan kontinu gunakan operator campuran-integer atau jadwal terpisah sehingga setiap tipe variabel menerima variasi dengan skala yang sesuai.

Terakhir, validasi pilihan encoding secara empiris. Bandingkan encoding integer langsung dengan alternatif (integer yang dikodekan biner, surrogate bernilai riil dengan pembulatan) pada instance representatif kecil untuk mengukur kecepatan konvergensi, robustitas, dan biaya penanganan kendala. Pada banyak tugas alokasi atau penjadwalan, representasi integer yang dipilih dengan baik ditambah operator diskrit khusus mengungguli surrogate kontinu umum; namun untuk masalah yang membutuhkan perilaku pencarian halus, strategi kontinuu surrogate dengan pembulatan dan adaptasi ukuran langkah yang cermat dapat bersaing. Gunakan hasil empiris ini untuk memilih skala mutasi/rekombinasi dan memutuskan apakah akan menggabungkan loop evolusioner dengan pencarian lokal deterministik pada neighbourhood integer.

4.7 Encoding Permutasi

Encoding permutasi merepresentasikan solusi calon sebagai permutasi dari himpunan hingga elemen, yaitu ruang genotipe adalah himpunan bijeksi pada $\{1, \ldots, n\}$. Pemetaan genotipe–fenotipe ϕ biasanya merupakan peta identitas: sebuah permutasi langsung menentukan pengurutan yang diinterpretasikan oleh evaluator spesifik masalah (mis. tur pada travelling salesman problem, urutan job pada penjadwalan satu mesin, atau daftar tugas terurut untuk lini aliran). Karena permutasi secara inheren menegakkan kendala pengurutan, encoding permutasi bersifat benar (sound) untuk masalah pengurutan dan menghindari banyak perbaikan kelayakan yang diperlukan oleh encoding naif.

Walau secara formal permutasi bersifat satu-ke-satu terhadap pengurutan, representasi praktis sering memperkenalkan kelas ekuivalen dan redundansi yang harus dikenali. Tur melingkar (seperti pada TSP simetris) memiliki simetri rotasi: pergeseran siklik permutasi mewakili tur yang sama dan refleksi juga mungkin ekuivalen. Simetri semacam ini tidak mengubah validitas tetapi memengaruhi sampling dan probabilitas seleksi; perancang harus memilih representatif kanonik (memasang kota pertama) atau menggunakan operator dan perbandingan fitness yang menyadari kelas ekuivalen untuk menghindari bias representasional.

Jarak dan lokalitas di ruang permutasi sangat berbeda dari ruang vektor. Jarak Hamming atau metrik posisi sederhana tidak menangkap struktur neighbourhood yang bermakna untuk masalah berbasis urutan. Jarak seperti Kendall tau (jumlah ketidaksepakatan berpasangan), inversi, atau metrik berbasis edge (jumlah perbedaan hubungan kedekatan) lebih mencerminkan perubahan kecil yang dapat diinterpretasikan dan mempertahankan struktur masalah. Desain operator harus dipandu oleh aspek permutasi mana yang merupakan building-block berguna untuk masalah — blok berbasis posisi, blok adjacency/edge, atau relasi precedence — karena operator berbeda melestarikan struktur yang berbeda.

Operator variasi untuk encoding permutasi harus menjaga kelayakan (menghasilkan permutasi valid) dan idealnya menghormati notion lokalitas yang dipilih. Mutasi tipikal meliputi swap, insert (mengambil satu elemen dan memasukkannya ke posisi lain), dan inversi/ reversal subsekuens; ini memiliki interpretasi jelas sebagai reorder lokal kecil. Operator rekombinasi dirancang untuk menggabungkan urutan orang tua sambil mempertahankan validitas permutasi: contoh termasuk partially mapped crossover (PMX), order crossover (OX), cycle crossover (CX), dan edge recombination. Masing-masing menekankan struktur yang dilestarikan berbeda (posisi, urutan, atau adjacency) dan pilihan harus sesuai dengan building-block spesifik masalah (mis. edge-based recombiners alami untuk TSP di mana edge lebih penting daripada posisi absolut).

Alternatifnya adalah menggunakan encoding tidak langsung dan decoder konstruktif bila kendala atau heuristik konstruktif penting. Encoding prioritas atau random-key memetakan kunci bernilai riil ke permutasi melalui decoder sorting stabil; decoder konstruktif membangun jadwal atau tur layak secara greedy dari genotipe yang menyandikan preferensi. Encoding tidak langsung dapat sangat mengurangi kompleksitas desain dengan memisahkan pencarian genetik dari penegakan kelayakan dan dapat memasukkan heuristik domain ke dalam decoder, tetapi mereka memindahkan beban desain ke decoder dan dapat mengaburkan sifat lokalitas operator genetik.

Rekomendasi praktis: inisialisasi populasi menggunakan campuran permutasi acak dan heuristik spesifik masalah untuk menabur struktur berguna; ukur keberagaman dengan metrik yang menyadari permutasi (Kendall tau atau overlap edge) daripada jarak Hamming; utamakan operator yang melestarikan notion building-block relevan; dan gabungkan pencarian global berbasis permutasi dengan optimisasi lokal (mis. 2-opt atau 3-opt untuk TSP, atau pencarian neighbourhood khusus untuk penjadwalan) untuk memanfaatkan perbaikan halus. Saat simetri ada, gunakan kanonisasi atau evaluasi yang menyadari ekuivalensi untuk menghindari bias. Akhirnya, validasi pilihan secara empiris pada instance representatif karena efektivitas operator sangat bergantung pada masalah di ruang permutasi.

4.8 Tree Encoding

Tree Encoding merepresentasikan genotipe sebagai tree berakar berlabel dimana node membawa simbol yang diambil dari satu atau lebih alfabet (mis. simbol fungsi/operator untuk node internal dan simbol terminal untuk daun). Fenotipe diperoleh dengan menginterpretasikan tree sesuai semantik masalah: dalam pemrograman genetik tree menyatakan ekspresi atau program, dalam optimisasi sintaksis ia merepresentasikan tree parse, dan dalam desain hierarkis ia merepresentasikan komposisi komponen. Secara formal ruang genotipe adalah himpunan tree terurut hingga atas alfabet berperingkat, dan decoder ϕ adalah fungsi evaluasi atau instansiasi yang memetakan tree ke objek spesifik masalah di

P.

Tree Encoding memperkenalkan pilihan representasional yang sangat mempengaruhi perilaku operator dan dinamika pencarian. Perancang harus menentukan alfabet node (bertipe atau tidak), kendala aritas (tetap atau variabel), dan linearisation untuk penyimpanan (struktur pointer, string berpagar, notasi prefix/postfix, atau daftar anak eksplisit). Pohon bertipe (strongly-typed) menegakkan kendala sintaksis pada tingkat representasi, mencegah banyak keturunan tidak valid dan mengurangi kebutuhan perbaikan; tree tidak bertipe lebih fleksibel tetapi sering memerlukan pemeriksaan kelayakan tambahan atau decoder. Representasi memengaruhi lokalitas: penggantian subtree kecil dapat menimbulkan perubahan semantik besar ketika semantik node bersifat non-linier atau konteks-sensitif.

Kekhawatiran sentral pada tree encoding adalah bloat — pertumbuhan ukuran tree yang tidak terkendali tanpa peningkatan kecocokan yang sepadan. Bloat muncul dari daerah netral atau seleksi lemah dimana tree yang lebih besar tidak diberi penalti, dan menurunkan kinerja dengan meningkatkan biaya evaluasi serta mengurangi variabilitas efektif populasi. Tindakan pencegahan umum meliputi batas statis pada kedalaman/ukuran, tekanan parsimoni (penalti eksplisit ukuran atau kompleksitas dalam fitness), dan kontrol sadar-operator (membatasi ukuran keturunan pada crossover dan mutasi). Saat menggunakan kontrol pertumbuhan, diperlukan keseimbangan: pemangkasan terlalu agresif dapat menghilangkan variasi struktural berguna, sementara pengaturan longgar menyebabkan kehabisan sumber daya.

Operator variasi untuk tree harus menjaga keterbentukan tree. Operator standar termasuk subtree crossover (menukar subtree antar orang tua), point mutation (mengganti node atau subtree kecil dengan subtree yang dihasilkan acak), dan hoist mutation (mengganti sebuah tree dengan salah satu subtree-nya untuk mengurangi ukuran). Ada pula operator yang mempertahankan konteks dirancang untuk bahasa bertipe atau tata bahasa (pertukaran subtree terbatas, mutasi pandu-tata-bahasa) yang menjaga kebenaran sintaksis secara konstuktif. Desain operator harus selaras dengan semantik alfabet node: untuk tree ekspresi, mendorong rekombinasi yang menyadari komutatif/assosiatif atau penyederhanaan aljabar dapat meningkatkan produksi keturunan bermakna.

Encoding indirek dan berbasis tata bahasa sangat berguna bila fenotipe harus memenuhi kendala sintaksis atau semantik yang kaya. Dalam grammatical evolution dan grammar-guided GP, genotipe biasanya menyandikan derivasi atau urutan pilihan produksi, dan decoder deterministik memetakannya ke tree yang dijamin valid secara sintaksis. Encoding indirek ini dapat menghasilkan genotipe ringkas dan memungkinkan memasukkan pengetahuan domain, tetapi dapat mengaburkan sifat lokalitas operator dan membutuhkan desain decoder yang hati-hati untuk menghindari sampling fenotipe yang bias.

Rekomendasi praktis: tegakkan kelayakan sejak dini menggunakan pengetipan atau kendala tata bahasa bila domain memerlukan kebenaran sintaksis; gabungkan pencarian tree global dengan lintasan penyederhanaan lokal (constant folding, reduksi aljabar) untuk meningkatkan efisiensi evaluasi; gunakan strategi inisialisasi campuran (ramped half-and-half, grow/full) untuk menabur variasi ukuran dan bentuk tree; dan terapkan kontrol kompleksitas eksplisit (tekanan parsimoni, batas kedalaman, atau bias operator adaptif) untuk mengelola bloat. Akhirnya, validasi operator secara empiris pada instance representatif dan pantau ukuran tree, kedalaman, dan biaya evaluasi selama eksperimen untuk mendeteksi bloat atau perilaku patologis.

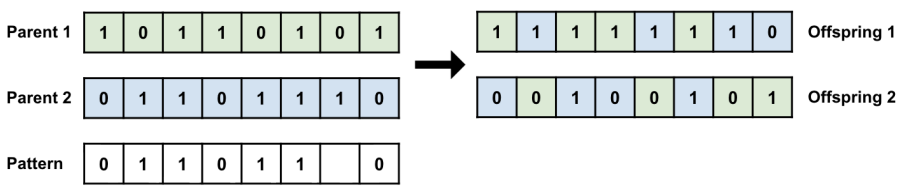
Bab 5
Metode Seleksi dalam Algoritma Genetika

5.1 Pendahuluan tentang Seleksi

Seleksi adalah mekanisme dalam algoritma genetika yang menentukan individu mana dari populasi yang dipilih untuk menyumbangkan materi genetik ke generasi berikutnya. Pada inti operasinya, seleksi mengubah informasi kebugaran (fitness) menjadi peluang reproduksi: individu dengan nilai kebugaran relatif lebih tinggi diberi peluang lebih besar untuk menghasilkan keturunan, sehingga memfokuskan pencarian ke wilayah solusi yang menjanjikan. Bias ini harus dikelola dengan hati-hati agar algoritma dapat mengeksploitasi solusi berkualitas tinggi sekaligus tetap mengeksplorasi alternatif yang beragam.

Konsep penting yang terkait dengan seleksi adalah tekanan seleksi (selection pressure), yang mengukur seberapa kuat mekanisme seleksi memfavoritkan individu yang lebih baik. Tekanan seleksi tinggi mempercepat konvergensi dengan memperbesar keuntungan reproduktif individu terbaik, namun meningkatkan risiko konvergensi prematur saat populasi kehilangan keragaman dan terjebak pada solusi suboptimal. Tekanan seleksi rendah mempertahankan keragaman dan mendorong eksplorasi, tetapi dapat memperlambat kemajuan menuju solusi berkualitas. Oleh karena itu, desain praktis algoritma membutuhkan penyeimbangan efek-efek ini, misalnya dengan pengaturan parameter seleksi atau dengan menggabungkan skema seleksi bersama mekanisme yang mempertahankan keragaman.

Operator seleksi terbagi ke dalam beberapa keluarga yang memberikan kompromi berbeda antara kesederhanaan, kontrol tekanan seleksi, dan sensitivitas terhadap skala kebugaran. Pendekatan umum meliputi metode proporsional terhadap kebugaran (mis. roulette wheel dan stochastic universal sampling), skema berbasis peringkat yang memberikan tekanan terkontrol dan independen skala, seleksi turnamen yang efisien dan dapat diatur tekanannya, serta strategi tronkasi atau elitisme yang secara deterministik mempertahankan individu terbaik. Bagian-bagian selanjutnya menguraikan metode tersebut secara detail, termasuk algoritma, sifat statistik, serta kelebihan dan kekurangan praktisnya.



Gambar 5.1: Proses seleksi dasar dalam Algoritma Genetika

5.2 Tekanan Seleksi

Tekanan seleksi mengkuantifikasi seberapa kuat suatu mekanisme seleksi memfavoritkan individu dengan kebugaran lebih tinggi saat menghasilkan generasi berikutnya. Secara intuitif, ia mengukur keuntungan reproduktif yang diharapkan dari solusi baik relatif terhadap rata-rata populasi. Tekanan seleksi dapat diformalkan dengan berbagai cara; ukuran operasional yang umum antara lain intensitas seleksi (perbedaan terstandarisasi antara rata-rata orang tua dan populasi) dan waktu takeover (jumlah generasi yang dibutuhkan agar individu terbaik mendominasi populasi bila seleksi diulang). Ukuran-ukuran ini memungkinkan perbandingan terukur antara operator seleksi dan parametrisasinya.

Pengendalian praktis terhadap tekanan seleksi meliputi pilihan algoritmik (mis. ukuran turnamen, kemiringan peringkat, fraksi tronkasi), teknik penskalaan kebugaran (mis. linear atau sigma scaling, seleksi Boltzmann), serta strategi hibrida yang menyesuaikan tekanan selama run (mis. mulai dengan tekanan rendah untuk eksplorasi lalu tingkatkan untuk eksploitasi). Memantau statistik terkait seleksi — seperti rata-rata dan varians kebugaran, ukuran keragaman (mis. jarak Hamming rata-rata pada enkoding biner), dan perkiraan takeover time — memberi umpan balik berguna untuk tuning.

5.3 Seleksi Proporsional terhadap Kebugaran (FPS)

Algoritma Genetika yang dikembangkan oleh Holland menggunakan Fitness Proportionate Selection (FPS) [24, 20], dimana nilai harapan suatu individu dihitung sebagai rasio kebugaran individu terhadap kebugaran rata-rata populasi.

Dalam metode ini, setiap individu dipilih sebagai orang tua dengan probabilitas proporsional terhadap nilai kebugarannya. Dengan demikian, individu yang lebih fit memiliki peluang lebih besar untuk mereproduksi dan menyebarkan sifatnya ke generasi berikutnya.

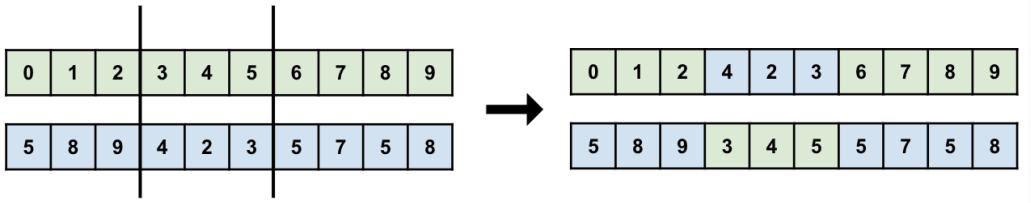
5.3.1 Seleksi Roulette Wheel

Dikenal juga sebagai seleksi proporsional kebugaran, di mana individu dipilih dengan probabilitas proporsional terhadap kebugarannya [20, 3, 4].

Skema seleksi paling sederhana adalah roulette wheel, atau pengambilan sampel stokastik dengan penggantian. Individu dipetakan ke segmen-segmen pada garis sesuai nilai kebugaran; bilangan acak kemudian menentukan segmen (individu) yang terpilih. Proses diulang hingga jumlah individu yang diinginkan terpenuhi.

Nomor Individu	Nilai Kebugaran	Probabilitas Seleksi	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	–

Tabel 5.1: Probabilitas seleksi dan nilai kebugaran (dari Buku Ajar)



Gambar 5.2: Proses seleksi roulette-wheel dengan contoh pengundian

Algoritma

Algorithm 1 Roulette Wheel Selection

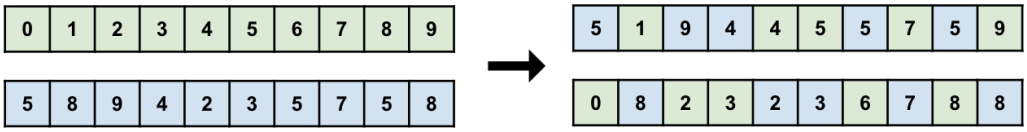
Hitung total kebugaran: $F = \sum_{i=1}^{N} f_i$
Hasilkan bilangan acak: $r \sim U[0, F]$
Set kumulatif kebugaran: $sum = 0$
for $i = 1$ to N **do**
  $sum = sum + f_i$
  if $sum \geq r$ **then**
    Pilih individu i
    break
  end if
end for

Probabilitas Seleksi

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

5.3.2 Stochastic Universal Sampling (SUS)

Versi yang ditingkatkan dari roulette-wheel yang mengurangi variansi sampling [7].



Gambar 5.3: Stochastic universal sampling dengan penunjuk berjarak sama

Algoritma

Algorithm 2 Stochastic Universal Sampling

Hitung total kebugaran: $F = \sum_{i=1}^{N} f_i$
Hitung jarak pointer: $distance = F/N$
Hasilkan start acak: $start \sim U[0, distance]$
Buat pointer: $pointer_i = start + i \times distance$ untuk $i = 0, 1, \ldots, N-1$
for setiap pointer **do**
  Pilih individu menggunakan logika roulette wheel
end for

5.4 Seleksi Berbasis Peringkat

Seleksi berbasis peringkat menetapkan probabilitas seleksi berdasarkan peringkat kebugaran, bukan nilai kebugaran mentah [21, 5].

5.4.1 Perankingan Linear

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

di mana $rank_i$ adalah peringkat individu i ($1 =$ terburuk, $N =$ terbaik) dan $\eta^+ + \eta^- = 2$.

5.5 Seleksi Turnamen

Seleksi turnamen memilih secara acak k individu dan memilih yang terbaik di antara mereka [20].

5.5.1 Mekanisme

1. Tentukan ukuran turnamen (k).
2. Pilih secara acak k individu.
3. Pilih individu dengan kebugaran tertinggi di antara mereka.
4. Tambahkan pemenang ke mating pool.

5. Ulangi hingga jumlah yang diinginkan tercapai.

Algorithm 3 Binary Tournament Selection

Pilih acak individu i
Pilih acak individu j (dengan $j \neq i$)
if $f_i > f_j$ **then**
  Pilih individu i
else
  Pilih individu j
end if

5.6 Truncation Selection

Truncation selection mempertahankan hanya fraksi teratas populasi untuk reproduksi. Parameter utama adalah rasio seleksi

$$\rho = \frac{\mu}{\lambda}, \tag{5.3}$$

yang mengontrol tekanan seleksi: ρ kecil berarti tekanan kuat.

5.7 Boltzmann Selection

Seleksi Boltzmann memetakan kebugaran ke probabilitas menggunakan distribusi Gibbs:

$$P_i = \frac{e^{f_i/T}}{\sum_{j=1}^{N} e^{f_j/T}}, \tag{5.4}$$

di mana T adalah parameter temperatur yang dapat dijadwalkan selama run.

5.8 Elitist Selection

Elitisme menjamin kelangsungan hidup sejumlah kecil individu terbaik antar generasi (mis. $e = 1$).

5.9 Seleksi yang Mempertahankan Keragaman

Teknik seperti fitness sharing, crowding, spesiasi, dan model pulau bertujuan mempertahankan variasi genetik. Contoh rumus fitness sharing:

$$f_i' = \frac{f_i}{\sum_{j=1}^{N} sh(d_{ij})}, \tag{5.5}$$

dengan fungsi sharing tipikal:

$$sh(d) = \begin{cases} 1 - \left(\dfrac{d}{\sigma_{share}}\right)^{\alpha} & \text{jika } d < \sigma_{share}, \\ 0 & \text{lainnya.} \end{cases} \tag{5.6}$$

5.10 Seleksi Multi-objektif

Untuk masalah multi-objektif, dominasi Pareto dan non-dominated sorting (mis. NSGA-II) digunakan untuk menyeimbangkan konvergensi dan penyebaran solusi.

5.11 Perbandingan Metode Seleksi

Metode	Tekanan	Keragaman	Kompleksitas	Skalabilitas	Parameter
Roulette Wheel	Variabel	Lemah	$O(N)$	Lemah	—
SUS	Variabel	Baik	$O(N)$	Lemah	—
Rank Linear	Konstan	Baik	$O(N \log N)$	Baik	η^+, η^-
Tournament	Dapat Diatur	Baik	$O(1)$	Sangat Baik	k
Truncation	Tinggi	Lemah	$O(N \log N)$	Baik	μ/λ
Boltzmann	Adaptif	Sangat Baik	$O(N)$	Baik	$T(t)$

Tabel 5.2: Perbandingan Metode Seleksi

Bab 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

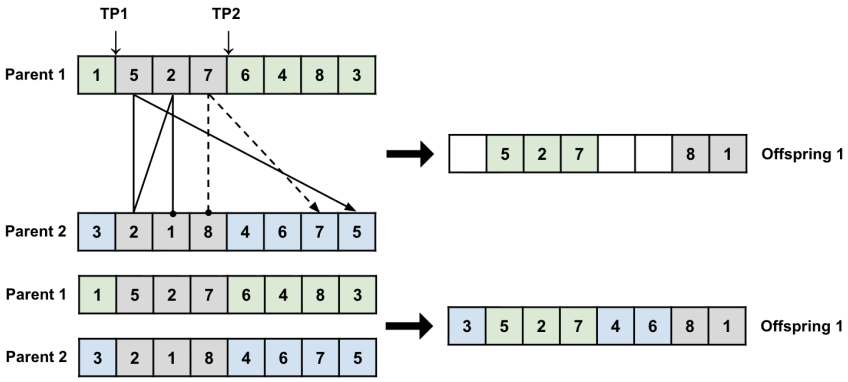
Crossover is a genetic operator used to vary the arrangement of chromosomes from one generation to the next [18, 41, 1]. The crossover method used depends on the encoding method applied.

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.



Gambar 6.1: Single Point Crossover for binary chromosomes

6.2.3 One-Point Crossover

Single crossover point divides chromosomes into two segments.

Algorithm

Algorithm 4 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.

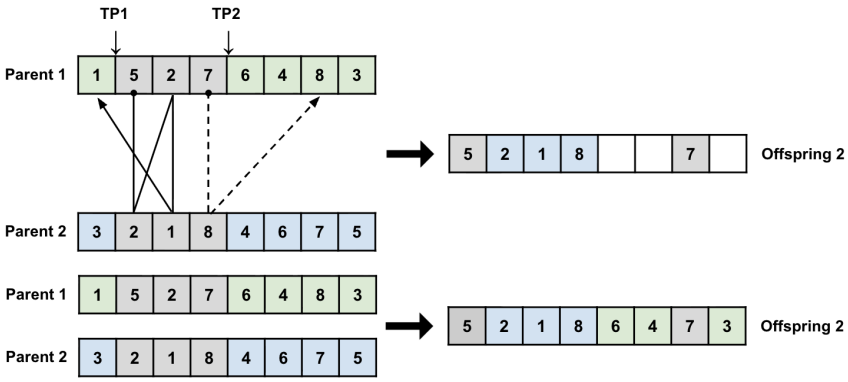
Multi-point crossover has a mechanism like single-point crossover; the difference is the number of randomly selected positions. In Multi-point crossover, a number N of positions are randomly selected along the length of the chromosome. These positions are exchanged to form two new offspring.

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 5 Two-Point Crossover

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



Gambar 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

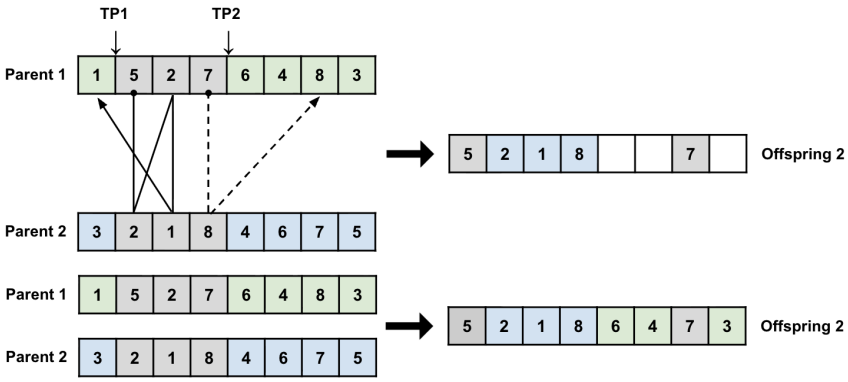
Two-point crossover reduces positional bias and can preserve building blocks at chromosome ends, although it is generally more disruptive than one-point crossover.

6.2.5 Uniform Crossover

Each gene is independently chosen from either parent [39, 14].

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.



Gambar 6.3: Uniform Crossover for binary chromosomes

Algorithm

Algorithm 6 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

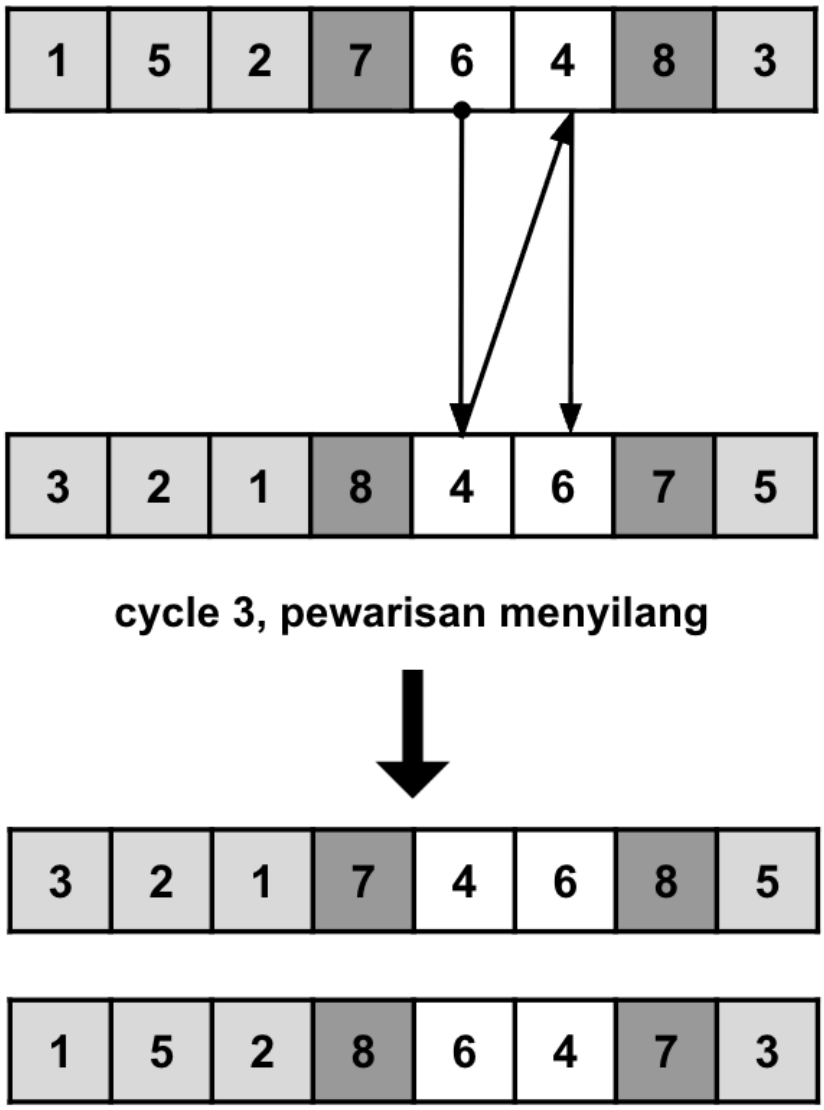
In uniform crossover, each gene is randomly selected from one of the corresponding genes on the parent chromosome. This selection process is done independently for each gene position. This method can be analogized to tossing a coin. If the result is "head", the gene is taken from parent 1; if the result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

Unlike crossover on binary or integer chromosomes, the crossover mechanism for real chromosomes involves arithmetic operations on gene values between parents. This method



Gambar 6.4: Single-Point Crossover for integer chromosomes

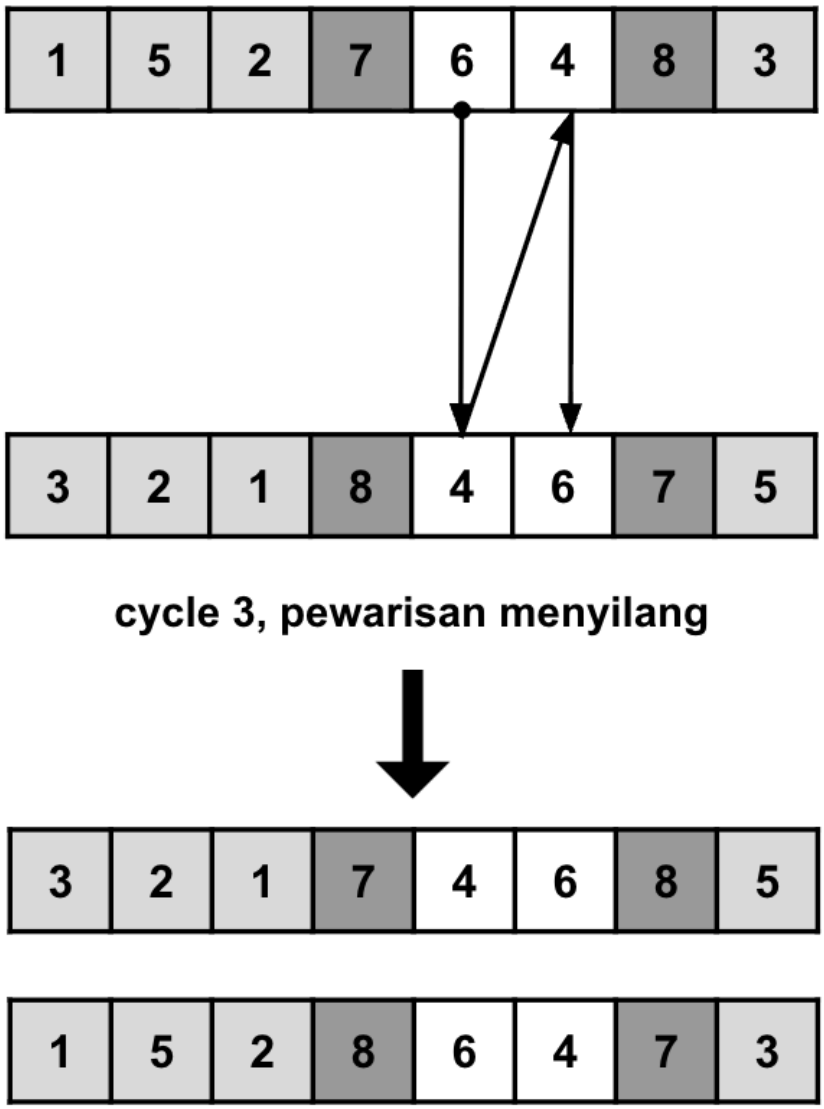
allows the creation of offspring with gene values that are between or around the parent gene values, thus maintaining the continuity and stability of the evolution process.

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
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 $0 \leq \alpha \leq 1$:

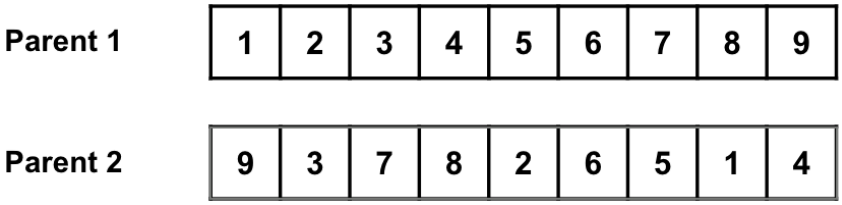


Gambar 6.5: Multi-point Crossover for integer chromosomes

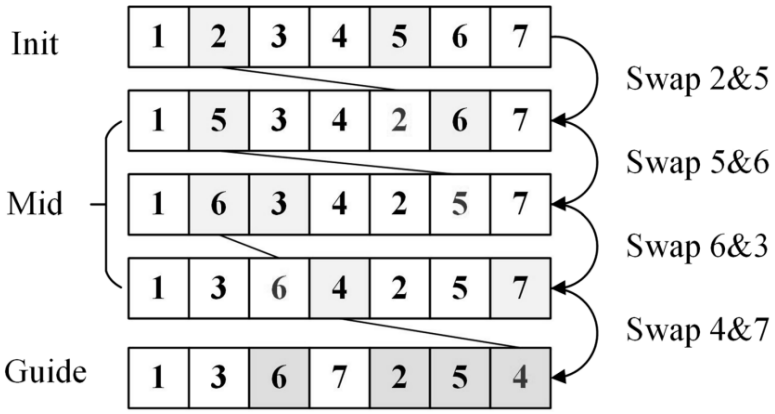
- Offspring 1: $\langle x_1, \ldots, x_{k-1}, \alpha \cdot y_k + (1-\alpha) \cdot x_k, x_{k+1}, \ldots, x_n \rangle$
- Offspring 2: $\langle y_1, \ldots, y_{k-1}, \alpha \cdot x_k + (1-\alpha) \cdot y_k, y_{k+1}, \ldots, y_n \rangle$

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 $0 \leq \alpha \leq 1$:
   - 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: $\langle y_1, \ldots, y_k, \alpha \cdot x_{k+1} + (1-\alpha) \cdot y_{k+1}, \ldots, \alpha \cdot x_n + (1-\alpha) \cdot y_n \rangle$



Gambar 6.6: Uniform Crossover for integer chromosomes



Gambar 6.7: Single 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: $z_i^2 = \alpha \cdot x_i + (1 - \alpha) \cdot y_i$

Whole Arithmetic Crossover

$$\mathbf{child_1} = \alpha \mathbf{parent_1} + (1 - \alpha)\mathbf{parent_2} \tag{6.14}$$
$$\mathbf{child_2} = (1 - \alpha)\mathbf{parent_1} + \alpha \mathbf{parent_2} \tag{6.15}$$

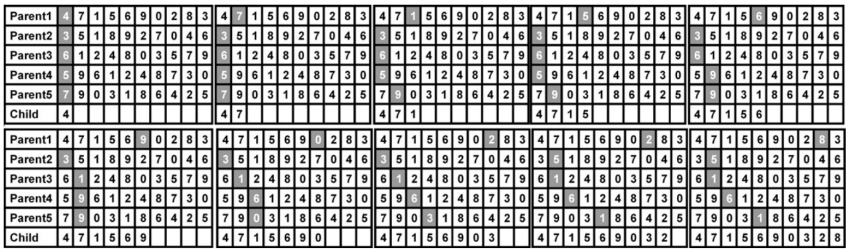
where $\alpha \in [0, 1]$ is a random weight.

Simple Arithmetic Crossover

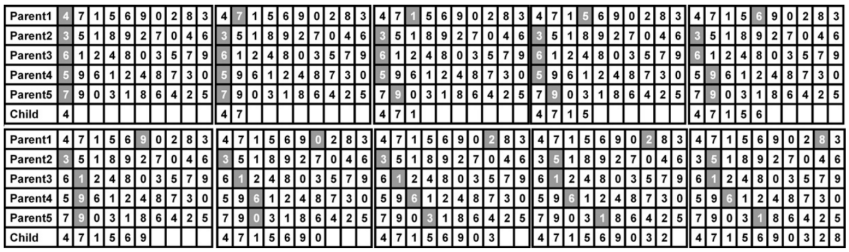
Apply arithmetic crossover to a random subset of genes.

Single Arithmetic Crossover

Apply arithmetic crossover to one randomly selected gene.



Gambar 6.8: Simple Arithmetic Crossover for real chromosomes



Gambar 6.9: Whole Arithmetic Crossover for real chromosomes

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 7 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 [33, 27].

Algorithm

Algorithm 8 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 [20, 27].

Algorithm

Algorithm 9 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 [33, 34].

Algorithm

Algorithm 10 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 11 Edge Recombination Crossover

```
Build edge table: for each element list its neighbors from both parents (no duplicates)
current ← element with fewest edges (break ties randomly)
for pos = 1 to n do
  child[pos] ← current
  Remove current from all edge lists
  if edge table of current has any neighbors then
    next ← neighbor of current with fewest edges (break ties randomly)
  else
    next ← random unused element
  end if
  current ← 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 [13, 45, 15, 8]); 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.

Bab 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, 42]. 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 [20, 30, 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 [29, 6, 23]. 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 12 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 13 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 [24, 20], 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 [9, 43]. 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 14 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 [43, 38], 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 [21, 35, 9]. 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 [44, 21]. 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:

Tabel 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

Bab 8
Aplikasi Algoritma Genetika

Bab ini menampilkan aplikasi dunia nyata dari algoritma genetika yang diambil dari materi perkuliahan dan mendemonstrasikan bagaimana konsep GA diterapkan dalam praktik [19, 37, 12].

Salah satu demonstrasi yang paling menarik adalah penerapan algoritma genetika pada AI game. Perkuliahan ini memuat contoh agen berbasis GA yang mengalahkan level pertama Super Mario Bros. dengan kecepatan 4× [31, 26].

Proyek "Towers of Reus" mendemonstrasikan bagaimana GA dapat digunakan untuk penyeimbangan gameplay. Pengguna membuat peta dengan parameter yang dapat disesuaikan sementara GA mencari konfigurasi yang menghasilkan karakteristik menang/kalah yang diinginkan. Sistem kemudian dapat melaporkan apakah menara terlalu kuat atau terlalu lemah dan menyajikan level yang dapat dikalahkan untuk diuji oleh pemain.

Contoh lain dalam perkuliahan adalah pencarian jalur: masalahnya adalah menemukan jalur terpendek melalui labirin yang kompleks. Pengkodean berupa urutan arah gerakan (atas, bawah, kiri, kanan) [27]. Fungsi fitness yang sesuai adalah kebalikan dari panjang jalur dengan penalti untuk menabrak dinding. Crossover digunakan untuk menggabungkan segmen jalur yang berhasil dan mutasi mengeksplorasi gerakan baru.

Navigasi robot fisik menunjukkan bagaimana GA ditransfer ke aplikasi perangkat keras. Kasus penggunaan meliputi perencanaan jalur real-time dalam lingkungan dinamis, mengintegrasikan data sensor untuk penghindaran rintangan, dan mengevolusi perilaku adaptif berdasarkan umpan balik lingkungan.

Perkuliahan ini juga merujuk contoh-contoh evolusi tersimulasi yang tersedia online di http://www.wreck.devisland.net/ga/. Contoh-contoh ini mengilustrasikan fitur seperti evolusi morfologi (perubahan pada struktur tubuh), optimasi pola pergerakan, adaptasi lingkungan, dan kriteria fitness multi-objektif seperti kecepatan, stabilitas, dan efisiensi [10, 11, 25].

Analogi intuitif yang digunakan dalam materi membandingkan populasi individu dengan kemampuan fisik yang bervariasi: seleksi menguntungkan mereka yang melompat lebih tinggi, pewarisan menurunkan sifat ke generasi berikutnya, dan mutasi memperkenalkan variasi teknik acak.

Contoh optimasi praktis adalah perencanaan perjalanan harian. Dalam analogi ini, pilihan rute bertindak sebagai gen, kondisi lalu lintas bertindak sebagai faktor lingkungan, dan waktu serta konsumsi bahan bakar berfungsi sebagai kriteria fitness. Selama generasi berulang sistem dapat belajar untuk memilih rute yang lebih efisien.

Bab ini juga menyoroti hubungan GA dengan paradigma lain: pembelajaran mesin (metode kernel, SVM, model Hidden Markov, metode Bayesian), komputasi lunak (jaringan saraf dan sistem fuzzy), dan pendekatan hibrid seperti pembelajaran penguatan.

Dari sudut pandang konseptual, perkuliahan ini mengontraskan gagasan Lamarck tentang karakteristik yang diperoleh dengan pandangan Darwin-Wallace tentang seleksi. Algoritma genetika mengikuti prinsip Darwin dengan menerapkan variasi acak dan seleksi untuk mencari solusi yang baik. Contoh-contoh visual ini mendemonstrasikan penerapan GA yang luas di berbagai bidang hiburan (AI game dan pembuatan konten prosedural), robotika (perencanaan jalur dan perilaku adaptif), simulasi (kehidupan buatan dan studi

evolusi), optimasi (perencanaan rute dan alokasi sumber daya), dan penelitian (mempelajari proses evolusioner). Wawasan kuncinya adalah bahwa GA menyediakan kerangka kerja terpadu untuk menyelesaikan masalah optimasi kompleks di berbagai domain, menjadikannya alat yang serbaguna dalam kecerdasan komputasional.

Lampiran 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):
```

```
110                     distance = (objectives[front[sorted_indices[i
                            + 1]], obj_idx] -
111                                 objectives[front[sorted_indices[i -
                                        1]], obj_idx])
112                     distances[sorted_indices[i]] += distance /
                            obj_range
113
114         return distances
115
116     def _tournament_selection(self, ranks: np.ndarray,
117                               crowding_distances: np.ndarray,
118                               population_size: int) -> List[int]:
119         """Seleksi turnamen biner berdasarkan rank dan crowding
                distance"""
120         selected = []
121
122         for _ in range(population_size):
123             # Select two random individuals
124             candidates = np.random.choice(len(ranks), 2, replace=
                    False)
125             i, j = candidates[0], candidates[1]
126
127             # Compare based on rank first, then crowding distance
128             if ranks[i] < ranks[j]:
129                 selected.append(i)
130             elif ranks[i] > ranks[j]:
131                 selected.append(j)
132             else:  # Same rank, compare crowding distance
133                 if crowding_distances[i] > crowding_distances[j]:
134                     selected.append(i)
135                 else:
136                     selected.append(j)
137
138         return selected
139
140     def _sbx_crossover(self, parent1: np.ndarray, parent2: np.
            ndarray,
141                        eta: float = 20.0) -> Tuple[np.ndarray, np.
                               ndarray]:
142         """Simulated Binary Crossover (SBX)"""
143         if np.random.random() > self.crossover_rate:
144             return parent1.copy(), parent2.copy()
145
146         child1 = np.zeros_like(parent1)
147         child2 = np.zeros_like(parent2)
148
149         for i in range(len(parent1)):
150             if np.random.random() <= 0.5:
151                 if abs(parent1[i] - parent2[i]) > 1e-14:
152                     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()
```

Lampiran 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

Bibliografi

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

[8] 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.

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

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

[11] 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.

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

[13] 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.

[14] 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.

[15] 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.

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

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

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

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

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

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

[22] 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.

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

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

[25] 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.

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

[27] 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.

[28] 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.

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

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

[31] 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.

[32] 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.

[33] 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.

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

[35] 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.

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

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

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

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

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

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

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

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

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

[45] 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.

[46] 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.