TUGAS BESAR 1 IF3070 DASAR INTELIGENSI ARTIFISIAL

PENCARIAN SOLUSI DIAGONAL *MAGIC CUBE* DENGAN *LOCAL SEARCH*

Disusun oleh

Kelompok 9

Daffa Ramadhan Elengi (18222009)

Givari Al Fachri (18222045)

Kayla Dyara (18222074)

Monica Angela Hartono (18222078)

PROGRAM STUDI SISTEM DAN TEKNOLOGI INFORMASI

SEKOLAH TEKNIK ELEKTRO DAN INFORMATIKA

JLN. GANESHA 10, BANDUNG 40132

2024

DAFTAR ISI

BAB I
DEKSRIPSI PERSOALAN

Diagonal magic cube merupakan kubus yang tersusun dari angka 1 hingga n^3 tanpa pengulangan dengan n merujuk pada panjang sisi pada kubus tersebut. Angka-angka pada kubus tersusun sedemikian rupa sehingga memenuhi beberapa kondisi sebagai berikut:

- Terdapat satu angka yang merupakan magic number dari kubus tersebut (*magic number* tidak harus termasuk dalam rentang 1 hingga n^3, *magic number* juga bukan termasuk ke dalam angka yang harus dimasukkan ke dalam kubus).
- Jumlah angka-angka untuk setiap baris sama dengan *magic number.*
- Jumlah angka-angka untuk setiap kolom sama dengan *magic number.*
- Jumlah angka-angka untuk setiap tiang sama dengan *magic number.*
- Jumlah angka-angka untuk seluruh diagonal ruang pada kubus sama dengan *magic number.*
- Jumlah angka-angka untuk seluruh diagonal pada suatu potongan bidang dari kubus sama dengan *magic number.*

Berdasarkan ketentuan di atas, akan dilakukan penyelesaian permasalahan *Diagonal Magic Cube* berukuran 5x5x5, dengan *initial state* dari kubus merupakan susunan angka 1 hingga 5^3 secara acak. Setiap iterasi pada algoritma *local search,* langkah yang boleh dilakukan adalah menukar posisi dari 2 angka pada kubus tersebut (2 angka yang ditukar tidak harus bersebelahan). Kami diminta untuk mengimplementasikan 3 algoritma *local search*, yaitu salah satu algoritma *hill climbing, simulated annealing,* dan *genetic algorithm* untuk menyelesaikan persoalan *diagonal magic cube*. Setelah itu, kami juga diminta untuk melakukan analisis terhadap hasil eksperimen. Penyelesaian masalah yang kami lakukan adalah dengan mengimplementasikan salah satu jenis *hill climbing algorithm*, yaitu *stochastic hill climbing, simulated annealing,* dan *genetic algorithm.*

BAB II
PEMBAHASAN

Program pencarian solusi *diagonal magic cube* dengan metode *local search* ini diimplementasikan menggunakan bahasa pemrograman Go. Pemilihan Go dalam tugas ini didasari oleh keunggulannya dalam hal kecepatan, efisiensi memori, dan kesederhanaan. Go menawarkan waktu kompilasi yang cepat dan *garbage collection* yang efektif dan sangat berguna dalam eksperimen dengan banyak iterasi. Selain itu, fitur *concurrency* berpotensi mempercepat pencarian solusi optimal. Sintaks yang sederhana membuat Go sangat cocok untuk mengimplementasikan algoritma *local search* yang membutuhkan evaluasi cepat di setiap tahap iterasi.

2.1 Kubus

Untuk memahami cara kerja program ini dengan baik, penting untuk melihatnya secara keseluruhan terlebih dahulu. Program ini terdiri dari berbagai komponen utama, yang masing-masing memiliki peran penting dalam mendefinisikan, memodifikasi, dan mengevaluasi kondisi kubus.

Pada constant.go, beberapa konstanta yang akan digunakan di seluruh bagian program. CUBE_ORDER mengacu pada dimensi kubus, yang dalam hal ini adalah 5. MAGIC_NUMBER dan MAGIC_NUMBER_AMOUNT merupakan angka kunci yang digunakan dalam perhitungan terkait dengan *magic cube*, sementara SEQUENCE_SIZE mendefinisikan jumlah total elemen dalam kubus (125 elemen).

Pada cube.go, didefinisikan struktur data *Cube*, yang menyimpan urutan elemen dalam kubus serta skor dan informasi mengenai solusi berikutnya (*successor*) dalam pencarian. Fungsi-fungsi yang ada di sini memungkinkan untuk membuat kubus baru dengan urutan acak dan menghitung skor berdasarkan seberapa dekat urutan tersebut dengan *magic number* yang ditargetkan. Fungsi RandomNeighbor memungkinkan untuk menghasilkan *neighbor* acak dari kubus yang sedang dianalisis.

Struktur dan konstruktor kubus dapat dilihat sebagai berikut sebagai berikut

Struktur Kubus

```
type Cube struct {
	sequence  [SEQUENCE_SIZE]int
	score     int
	successor *Cube
}
```

Konstruktor Kubus

```
func NewCube() *Cube {
	source := rand.NewSource(time.Now().UnixNano())
	r := rand.New(source)

	cube := &Cube{successor: nil}

	for i := 0; i < SEQUENCE_SIZE; i++ {
		cube.sequence[i] = i + 1
	}

	r.Shuffle(SEQUENCE_SIZE, func(i, j int) {
		cube.sequence[i],    cube.sequence[j]    =    cube.sequence[j],
cube.sequence[i]
	})

	cube.score = cube.ObjectiveFunction()

	return cube
}
```

Pengaksesan Kubus

```
func (c *Cube) GetSequence() [SEQUENCE_SIZE]int {
	return c.sequence
}

func (c *Cube) GetScore() int {
```

```
	return c.score
}

func (c *Cube) GetSuccessor() *Cube {
	return c.successor
}

func (c *Cube) SetSuccessor(successor *Cube) {
	c.successor = successor
}

func (c *Cube) get(x, y, z int) int {
	return c.sequence[x*CUBE_ORDER*CUBE_ORDER+y*CUBE_ORDER+z]
}
```

Algoritma *Random Neighbor*

```
func (c *Cube) RandomNeighbor() *Cube {
	neighbor := CopyCube(c)
	var idx1 int = rand.Intn(SEQUENCE_SIZE)
	var idx2 int = rand.Intn(SEQUENCE_SIZE)
	neighbor.sequence[idx1] = c.sequence[idx2]
	neighbor.sequence[idx2] = c.sequence[idx1]
	neighbor.score = neighbor.ObjectiveFunction()
	return neighbor
}
```

Di dalam detail.go, program memperkenalkan fungsi yang bertugas menghitung dan mencetak berapa banyak baris, kolom, pilar, serta diagonal pada kubus yang jumlahnya sama dengan MAGIC_NUMBER. Fungsi-fungsi ini memeriksa setiap aspek dari kubus, seperti baris dalam setiap lapisan XY, kolom dalam setiap lapisan XY, dan bahkan diagonal di dalam berbagai bidang, baik itu bidang XY, YZ, maupun XZ.

2.2 Pemilihan *Objective Function*

Dalam permasalahan diagonal *magic cube*, tujuan utamanya adalah untuk mengatur angka-angka dalam kubus sedemikian rupa sehingga setiap baris, kolom, tiang, diagonal

ruang, dan diagonal bidang memiliki jumlah yang sama dengan jumlah sebesar *magic number*. Oleh karena itu, *objective function* yang dipilih harus dapat mengukur seberapa jauh kondisi susunan angka saat ini dari kondisi ideal yang diinginkan. Dalam kasus Tugas Besar 1 IF3070 Dasar Intelegensi Artifisial, panjang sisi kubus adalah 5.

Pada suatu kubus orde 5 dapat diketahui informasi-informasi berikut:

- Terdapat 25 baris
- Terdapat 25 kolom
- Terdapat 25 tiang
- Terdapat 30 diagonal bidang
- Terdapat 4 diagonal ruang

Perhitungan *magic number* dalam diagonal *magic cube* dilakukan menggunakan rumus:

$$M = \frac{n(n^3+1)}{2}$$

(2.1.1)

Dengan M merupakan *magic number* dan n merupakan panjang sisi pada kubus. Menggunakan rumus (2.1.1), dapat diketahui *magic number* untuk kubus 5x5x5 adalah 315, dengan perhitungan sebagai berikut:

$$M = \frac{n(n^3+1)}{2} = \frac{5(5^3+1)}{2} = \frac{5(125+1)}{2} = \frac{5\,x\,126}{2} = 315$$

Objective function yang dipilih untuk menyelesaikan permasalahan diagonal *magic cube* ini akan menghitung deviasi total dari jumlah angka pada setiap baris, kolom, tiang, diagonal ruang, dan diagonal bidang dalam kubus terhadap *magic number*. Deviasi total adalah ukuran seberapa jauh konfigurasi angka dalam kubus saat ini terhadap *magic number*. Tujuan dari *objective function* ini adalah untuk meminimalkan deviasi total, semakin kecil deviasi total, semakin mendekati konfigurasi diagonal *magic cube* yang diinginkan.

Perhitungan deviasi total dalam diagonal *magic cube* dilakukan menggunakan rumus:

$$Deviasi\ Total = \sum_{i=1}^{n} (|\ Jumlah\ Baris_i - M\ |) + \sum_{j=1}^{n} (|\ Jumlah\ Kolom_j - M\ |) + \sum_{k=1}^{n} (|\ Jumlah\ Tiang_k - M\ |) + \sum_{l=1}^{d} (|\ Jumlah\ Diagonal\ Bidang_l - M\ |) + \sum_{m=1}^{r} (|\ Jumlah\ Diagonal\ Ruang_m - M\ |)$$

(2.1.2)

Dengan n merupakan panjang sisi kubus = 5, d merupakan jumlah diagonal bidang kubus = 30, r merupakan jumlah diagonal ruang kubus = 4, dan M merupakan *magic number* = 315. Setiap iterasi algoritma *local search* berjalan, *objective function* akan mengevaluasi konfigurasi baru tersebut dengan menghitung deviasi absolut untuk setiap baris, kolom, tiang, diagonal ruang, dan diagonal bidang, lalu menjumlahkan semua deviasi absolut tersebut untuk mendapatkan deviasi total.

Penggunaan *objective function* deviasi total untuk menyelesaikan permasalahan diagonal *magic cube* memberikan informasi yang jelas terkait seberapa jauh konfigurasi saat ini dari *magic number*. *Objective function* ini juga dapat diterapkan dengan mudah untuk berbagai ukuran kubus dengan menyesuaikan jumlah baris, kolom, tiang, diagonal ruang, dan diagonal bidang. Perhitungan deviasi total setiap kali terjadi perubahan konfigurasi kubus akan mempermudah penemuan solusi yang lebih mendekati konfigurasi *magic cube* pada setiap iterasi. Namun, *objective function* deviasi total masih memiliki beberapa kekurangan. Pada setiap iterasi, algoritma harus menghitung deviasi untuk setiap baris, kolom, tiang, diagonal ruang, dan diagonal bidang sehingga memakan waktu dan sumber daya yang lebih banyak. Deviasi total sangat efektif saat solusi masih jauh dari optimal karena memberikan informasi terkait seberapa besar kesalahannya. Namun, ketika solusi sudah mendekati sempurna, deviasi total mungkin tidak cukup memberikan peringatan kepada algoritma untuk memperbaiki elemen-elemen yang ada. Dengan menggunakan deviasi total, algoritma tidak secara eksplisit diarahkan untuk memperbaiki kesalahan terbesar yang ada. Semua kesalahan dianggap sama rata, baik yang memiliki kesalahan besar maupun yang kecil.

Kita dapat melihat implementasi *objective function* pada objective.go. Pada objective.go, perhitungan seberapa jauh suatu kubus dari konfigurasi yang ideal dapat dilakukan. Fungsi ObjectiveFunction ini melakukan penilaian dengan menghitung jumlah perbedaan antara

jumlah yang ada dengan MAGIC_NUMBER untuk setiap baris, kolom, pilar, dan diagonal di kubus. Semakin kecil nilai skor yang dihitung oleh fungsi ini, semakin baik konfigurasi kubus tersebut.

Implementasinya adalah sebagai berikut.

Parameter	
Nama	**Deskripsi**
`c *Cube`	Representasi dari kubus 3D yang digunakan dalam program
Variabel	
Nama	**Deskripsi**
`score int`	Menyimpan total skor dari *objective function*
`x int`	Merepresentasikan koordinat x dalam kubus 3D
`y int`	Merepresentasikan koordinat y dalam kubus 3D
`z int`	Merepresentasikan koordinat z dalam kubus 3D
`sum int`	Menyimpan jumlah nilai elemen dalam satu baris, kolom, pilar, atau diagonal pada setiap iterasi
Method	
Nama	**Deskripsi**

`ObjectiveFunction(c *Cube)`	Menghitung total skor dari *objective function* yang akan melakukan iterasi pada setiap baris, kolom, pilar, dan diagonal
`absoluteInt(n int)`	Menerima nilai integer n sebagai *input* dan mengembalikan nilai absolut dari n

Algoritma

```
func (c *Cube) ObjectiveFunction() int {
	score := 0
	for z := 0; z < CUBE_ORDER; z++ {
		for y := 0; y < CUBE_ORDER; y++ {
			sum := 0
			for x := 0; x < CUBE_ORDER; x++ {
				sum += c.get(x, y, z)
			}
			score += absoluteInt(MAGIC_NUMBER - sum)
		}
	}
	for z := 0; z < CUBE_ORDER; z++ {
		for x := 0; x < CUBE_ORDER; x++ {
			sum := 0
			for y := 0; y < CUBE_ORDER; y++ {
				sum += c.get(x, y, z)
			}
			score += absoluteInt(MAGIC_NUMBER - sum)
		}
	}
	for x := 0; x < CUBE_ORDER; x++ {
		for y := 0; y < CUBE_ORDER; y++ {
			sum := 0
			for z := 0; z < CUBE_ORDER; z++ {
				sum += c.get(x, y, z)
			}
			score += absoluteInt(MAGIC_NUMBER - sum)
		}
```

```
	}

	for z := 0; z < CUBE_ORDER; z++ {
		sum1, sum2 := 0, 0
		for i := 0; i < CUBE_ORDER; i++ {
			sum1 += c.get(i, i, z)
			sum2 += c.get(i, CUBE_ORDER-1-i, z)
		}
		score += absoluteInt(MAGIC_NUMBER - sum1)
		score += absoluteInt(MAGIC_NUMBER - sum2)
	}
	for x := 0; x < CUBE_ORDER; x++ {
		sum1, sum2 := 0, 0
		for i := 0; i < CUBE_ORDER; i++ {
			sum1 += c.get(x, i, i)
			sum2 += c.get(x, i, CUBE_ORDER-1-i)
		}
		score += absoluteInt(MAGIC_NUMBER - sum1)
		score += absoluteInt(MAGIC_NUMBER - sum2)
	}
	for y := 0; y < CUBE_ORDER; y++ {
		sum1, sum2 := 0, 0
		for i := 0; i < CUBE_ORDER; i++ {
			sum1 += c.get(i, y, i)
			sum2 += c.get(CUBE_ORDER-1-i, y, i)
		}
		score += absoluteInt(MAGIC_NUMBER - sum1)
		score += absoluteInt(MAGIC_NUMBER - sum2)
	}

	sum1, sum2, sum3, sum4 := 0, 0, 0, 0
	for i := 0; i < CUBE_ORDER; i++ {
		sum1 += c.get(i, i, i)
		sum2 += c.get(i, i, CUBE_ORDER-1-i)
		sum3 += c.get(i, CUBE_ORDER-1-i, i)
		sum4 += c.get(CUBE_ORDER-1-i, i, i)
	}
```

```
        score += absoluteInt(MAGIC_NUMBER - sum1)
        score += absoluteInt(MAGIC_NUMBER - sum2)
        score += absoluteInt(MAGIC_NUMBER - sum3)
        score += absoluteInt(MAGIC_NUMBER - sum4)
        return score
}
```

2.3 Penjelasan Implementasi Algoritma *Local Search*

2.3.1 Stochastic Hill Climbing

Stochastic Hill Climbing merupakan salah satu algoritma *hill climbing*. Dalam algoritma ini, dipilih satu solusi *neighbor* secara acak pada setiap iterasi. Jika *neighbor* memiliki skor yang lebih baik, maka algoritma akan berpindah ke solusi tersebut. Algoritma ini berulang hingga mencapai batas iterasi maksimum atau menemukan solusi yang diinginkan.

Program stochastic_hill_climbing.go menyelesaikan masalah *Diagonal Magic Cube* berukuran 5x5x5. Di dalam program ini, proses pencarian solusi dimulai dengan mendefinisikan sebuah kubus acak *current* sebagai keadaan awal. Algoritma kemudian melakukan iterasi hingga batas maksimal yang ditentukan oleh variabel maxIteration. Dalam setiap iterasi, algoritma mencoba menemukan solusi acak *neighbor* dari *current* dengan menggunakan metode RandomNeighbor(), yang secara acak menukar posisi dua angka pada kubus.

Pada setiap iterasi, algoritma mengevaluasi skor kubus *neighbor* menggunakan metode GetScore(), yang menunjukkan seberapa dekat atau jauh kubus tersebut dari kondisi optimal, yaitu nilai skor yang rendah. Jika skor *neighbor* lebih rendah dari *current*, algoritma menganggap *neighbor* sebagai solusi yang lebih baik dan menggantikan *current* dengan tetangga tersebut melalui metode SetSuccessor(). Setelah mencapai batas maksimal iterasi atau menemukan kubus dengan skor terbaik, program mengembalikan *current* sebagai hasil akhir, yang merupakan keadaan kubus yang paling dekat dengan solusi optimal.

Implementasinya adalah sebagai berikut.

Parameter	
Nama	**Deskripsi**
`c *cube.Cube`	Merupakan *state* awal yang diinisiasi sebagai *current.*
`maxIterations int`	Jumlah maksimum iterasi yang digunakan untuk proses pencarian solusi optimal.
Variabel	
Nama	**Deskripsi**
`current *cube.Cube`	Menyimpan konfigurasi saat ini yang sedang dievaluasi
`neighbor *cube.Cube`	Merepresentasikan kubus tetangga yang dihasilkan secara acak dari kubus saat ini
Method	
Nama	**Deskripsi**
`StochasticHillClimbing(c *cube.Cube, maxIteration int) *cube.Cube`	Konstruktor kelas StochasticHillClimbing, parameternya adalah objek *cube* yang dihasilkan di awal.
`RandomNeighbor()`	Menghasilkan *neighbor* acak dari kubus saat ini (*current*)
`GetScore()`	Mengembalikan skor atau nilai dari konfigurasi kubus saat ini untuk membandingkan solusi.

`SetSuccessor(neighbor *cube.Cube)`	Mengatur *neighbor* sebagai pergantian kubus saat ini

Algoritma

```
package localsearch

import (
	"diagonalmagiccube/cube"
)

func StochasticHillClimbing(c *cube.Cube, maxIteration int) *cube.Cube {
	current := c
	var neighbor *cube.Cube
	for i := 0; i < maxIteration; i++ {
		neighbor = current.RandomNeighbor()
		if neighbor.GetScore() < current.GetScore() {
			current.SetSuccessor(neighbor)
			current = neighbor
		}
	}
	return current
}
```

2.3.2 Simulated Annealing

Algoritma *Simulated Annealing* melakukan pencarian solusi optimal yang menggunakan konsep proses pendinginan logam. Algoritma ini dimulai dengan inisialisasi solusi awal (*current*), sebuah objek dari kelas cube.Cube. Solusi ini kemudian dievaluasi menggunakan atribut GetScore untuk mendapatkan skor solusi. Penggunaan konsep *cooling schedule* digunakan pada setiap iterasi, dengan temperature dihitung melalui fungsi schedule(t) yang menurun seiring bertambahnya iterasi. Konsep ini lah yang mengatur probabilitas penerimaan solusi yang lebih buruk. Solusi *neighbor* (*next*) dihasilkan melalui *method* RandomNeighbor() pada *current* yang menciptakan konfigurasi acak baru dari kubus.

deltaE menyimpan perubahan skor antara *current* dan *next*. Apabila deltaE bernilai negatif, *next* lebih baik, maka *method* SetSuccessor menerima *next* sebagai solusi baru. Namun, apabila deltaE bernilai positif atau sama dengan nol, *next* lebih buruk, solusi tersebut hanya bisa diterima dengan probabilitas math.Exp(-deltaE/temperature) yang dihitung berdasarkan suhu saat itu. Algoritma semakin selektif memilih solusi terbaik seiring dengan penurunan suhu. Hal ini memungkinkan algoritma untuk mendekati solusi optimal saat proses pencarian selesai setelah mencapai jumlah iterasi maksimum oleh maxIterations.

Implementasinya adalah sebagai berikut

Parameter	
Nama	**Deskripsi**
`c *cube.Cube`	Merupakan *state* awal yang diinisiasi sebagai *current.*
`maxIterations int`	Jumlah maksimum iterasi yang digunakan untuk proses pencarian solusi optimal.
Variabel	
Nama	**Deskripsi**
`current *cube.Cube`	Menyimpan konfigurasi saat ini yang sedang dievaluasi
`next *cube.Cube`	Menyimpan konfigurasi baru yang merupakan solusi *neighbor* dari *current*.
`t int`	Menyimpan nilai iterasi saat ini.
`temperature float64`	Menyimpan nilai suhu pada setiap iterasi.
`deltaE float64`	Menyimpan perbedaan skor antara solusi

	tetangga (*next*) dan solusi saat ini (*current*).
Method	
Nama	**Deskripsi**
`schedule (t int) float64`	Menghitung suhu pada setiap iterasi t, yang menurun seiring bertambahnya iterasi.
`SimulatedAnnealing(c *cube.Cube, maxIterations int) *cube.Cube`	Konstruktor kelas SimulatedAnnealing, parameternya adalah objek *cube* yang dihasilkan di awal.
`RandomNeighbor()`	Menghasilkan *neighbor* acak dari kubus saat ini (*current*)
`GetScore()`	Mengembalikan skor atau nilai dari konfigurasi kubus saat ini untuk membandingkan solusi.
`SetSuccessor(next *cube.Cube)`	Method untuk memperbarui kubus (*current)* menjadi *neighbor* baru (*next)*, jika *next* diterima sebagai solusi baru.
Algoritma	

```
package localsearch

import (
	"diagonalmagiccube/cube"
	"math"
	"math/rand"
)
```

```
func schedule(t int) float64 {
	return float64(100) / (0.001*float64(t) + 1)
}

// Algoritma Simulated Annealing untuk mencari solusi
func SimulatedAnnealing(c *cube.Cube, maxIterations int) *cube.Cube {
	current := c

	for t := 0; t < maxIterations; t++ {
		temperature := schedule(t)
		next := current.RandomNeighbor()
		deltaE := float64(next.GetScore() - current.GetScore())

		if deltaE < 0 {
			current.SetSuccessor(next)
			current = next
		} else if math.Exp(-deltaE/temperature) > rand.Float64() {
			current.SetSuccessor(next)
			current = next
		}
	}
	return current
}
```

2.3.3 Genetic Algorithm

Genetic algorithm merupakan algoritma *search* yang berbasis alam. Dalam algoritma ini, solusi untuk suatu permasalahan dianggap sebagai individu yang berevolusi untuk menghasilkan solusi yang lebih baik. Melalui mekanisme seleksi, *crossover*, dan mutasi, algoritma ini mencoba menemukan solusi optimal dengan memodifikasi populasi solusi secara iteratif. Setiap individu dalam populasi diukur dengan fungsi *fitness* untuk menilai seberapa baik individu tersebut memenuhi tujuan dari permasalahan. Algoritma ini akan menghasilkan generasi baru hingga menemukan solusi yang optimal atau mencapai batas iterasi tertentu.

Implementasi *genetic algorithm* untuk *diagonal magic cube* terbagi dalam dua file, yaitu generation.go dan genetic_algorithm.go. File generation.go mengelola struktur dasar dari generasi dan individu dalam populasi serta mendefinisikan berbagai fungsi evolusi. Struktur generation di dalamnya mencakup populasi solusi, nilai total *fitness*, dan referensi ke generasi berikutnya, sementara individual mewakili karakteristik genetik dari kubus. Di dalam file ini terdapat beberapa fungsi utama seperti NewGeneration yang membentuk generasi pertama dengan populasi acak, fitness yang menghitung nilai *fitness* tiap individu, selection yang memilih individu berdasarkan nilai *fitness*, serta crossOver yang menghasilkan keturunan dari kombinasi gen kedua orang tua. Selain itu, terdapat fungsi mutation untuk memperkenalkan variasi acak, Evolution yang melakukan seluruh proses evolusi, dan BestIndividual untuk mencari individu dengan nilai *fitness* tertinggi dalam generasi. Sementara itu, file genetic_algorithm.go bertindak sebagai penggerak utama *genetic algorithm*, dengan fungsi GeneticAlgorithm yang menjalankan iterasi dan meng-*update* generasi dengan memanfaatkan fungsi Evolution, sehingga menghasilkan solusi terbaik setelah batas iterasi atau solusi optimal tercapai.

Implementasi generation.go adalah sebagai berikut.

Struktur Generasi dan Individu

```
type Generation struct {
      population     [POPULATION_SIZE]*Individual
      nextGeneration *Generation
      totalFitness   float64
}

type Individual struct {
      cube     *Cube
      parentX  *Individual
      parentY  *Individual
      mutation bool
}
```

Parameter

Nama	Deskripsi
`population [POPULATION_SIZE]*Individual`	Menyetel populasi pada generasi tertentu dengan sejumlah individu yang dihasilkan.
Variabel	
Nama	**Deskripsi**
`generation *Generation`	Merepresentasikan satu generasi berisi populasi solusi
`individual *Individual`	Merepresentasikan satu solusi tunggal dalam generasi
`randValue float64`	Menentukan titik acak pada *fitness wheel*, berfungsi untuk memilih individu berdasarkan *fitness* mereka.
`offspiring *Individual`	Menyimpan individu baru yang dihasilkan dari dua orang tua (hasil *crossover*)
`crossoverPoint1 int` dan `crossoverPoint2 int`	Menentukan titik-titik acak dalam gen untuk menggabungkan gen orang tua dalam menghasilkan individu baru.
`searchIndex int`	Membantu mencari elemen dalam urutan gen orang tua kedua saat mengisi gen individu baru.
`nextGeneration *Generation`	Menyimpan referensi ke generasi berikutnya dalam proses evolusi
`parentX *Individual` dan `parentY *Individual`	Menyimpan referensi ke dua orang tua

	dari individu yang diperlukan dalam proses *crossover*
`child *Individual`	Menyimpan individu hasil *crossover* dan mutasi sebelum dimasukkan ke dalam generasi berikutnya
`abortion int`	Menghasilkan individu keturunan yang lebih baik dari orang tua.
`best *Individual`	Menyimpan referensi ke individu terbaik yang ditemukan dalam satu generasi berdasarkan nilai *fitness*
`bestFitness float64`	Menyimpan nilai *fitness* tertinggi saat ini saat mengidentifikasi individu terbaik dalam generasi
Method	
Nama	**Deskripsi**
`NewGeneration() *Generation`	Membuat dan menginisialisasi generasi baru dengan populasi yang diisi individu-individu secara acak
`NewIndividual() *Individual`	Menghasilkan individu baru yang diinisialisasi dengan kubus yang diacak
`fitness(individual *Individual) float64`	Menghitung nilai *fitness* dari satu individu, yang menunjukkan seberapa baik solusi tersebut mendekati solusi optimal
`selection(generation *Generation)`	Memilih satu individu dari generasi

`*Individual`	berdasarkan probabilitas proporsional terhadap nilai *fitness*
`crossOver(parentX *Individual, parentY *Individual) *Individual`	Menghasilkan individu keturunan baru melalui *crossover*
`mutation(individual *Individual) *Individual`	Variasi acak pada satu individu dengan probabilitas tertentu (tingkat mutasi)
`Evolution(generation *Generation) *Generation`	Mengelola seluruh proses evolusi dalam satu generasi
`BestIndividual(generation *Generation) *Individual`	Menemukan dan mengembalikan individu dengan nilai *fitness* tertinggi dalam satu generasi

Algoritma

```
func NewGeneration() *Generation {
	generation := &Generation{nextGeneration: nil}
	for i := 0; i < POPULATION_SIZE; i++ {
		generation.population[i] = NewIndividual()
		generation.totalFitness += fitness(generation.population[i])
	}
	return generation
}

func NewIndividual() *Individual {
	individual := &Individual{cube: NewCube(), parentX: nil, parentY: nil, mutation: false}
	return individual
}

func fitness(individual *Individual) float64 {
	return 10000 / (float64(individual.cube.score) + 1)
```

```
}

func selection(generation *Generation) *Individual {
	randValue := rand.Float64()
	var cumulativeFitness float64 = 0

	for i := 0; i < POPULATION_SIZE; i++ {
		cumulativeFitness += fitness(generation.population[i]) / generation.totalFitness
		if randValue < cumulativeFitness {
			return generation.population[i]
		}
	}

	return generation.population[POPULATION_SIZE-1]
}

func crossOver(parentX *Individual, parentY *Individual) *Individual {
	offspring := &Individual{cube: &Cube{
		sequence: [SEQUENCE_SIZE]int{}, // default values of int are zeros
		successor: nil,
	}, parentX: parentX, parentY: parentY, mutation: false}

	crossoverPoint1 := rand.Intn(SEQUENCE_SIZE / 2)
	crossoverPoint2 := rand.Intn(SEQUENCE_SIZE/2) + SEQUENCE_SIZE/2
	if crossoverPoint1 > crossoverPoint2 {
		crossoverPoint1, crossoverPoint2 = crossoverPoint2, crossoverPoint1
	}

	for i := crossoverPoint1; i < crossoverPoint2; i++ {
		offspring.cube.sequence[i] = parentX.cube.sequence[i]
	}

	for i := 0; i < SEQUENCE_SIZE; i++ {
		if i < crossoverPoint1 || i >= crossoverPoint2 {
```

```
                  searchIndex := i

                  for                  contains(offspring.cube.sequence,
parentY.cube.sequence[searchIndex]) {
                        searchIndex++

                        if searchIndex >= SEQUENCE_SIZE {
                              searchIndex = 0
                        }
                  }
                  offspring.cube.sequence[i]                              =
parentY.cube.sequence[searchIndex]
            }
      }
      offspring.cube.score = offspring.cube.ObjectiveFunction()
      return offspring
}

func mutation(individual *Individual) *Individual {
      if rand.Float64() < MUTATION_RATE {
            individual.mutation = true
            individual.cube = individual.cube.RandomNeighbor()
      }
      return individual
}

func Evolution(generation *Generation) *Generation {
      nextGeneration := &Generation{totalFitness: 0}
      var population [POPULATION_SIZE]*Individual
      for i := 0; i < POPULATION_SIZE; i++ {
            parentX := selection(generation)
            parentY := selection(generation)
            child := crossOver(parentX, parentY)
            mutation(child)
            var abortion int = 0
            for    ((child.cube.score    >    parentX.cube.score)    ||
(child.cube.score > parentY.cube.score)) && abortion < 1000 {
```

```
                  if child.cube.score == parentX.cube.score {
                        mutation(child)
                  } else {
                        child = crossOver(parentX, parentY)
                  }
                  abortion++
            }
            population[i] = child
            nextGeneration.totalFitness += fitness(child)
      }
      nextGeneration.SetPopulation(population)
      generation.SetNextGeneration(nextGeneration)
      return nextGeneration
}

func BestIndividual(generation *Generation) *Individual {
      var best *Individual
      var bestFitness float64

      for _, individual := range generation.population {
            // Calculate the fitness for the current individual
            individualFitness := fitness(individual)

            if best == nil || individualFitness > bestFitness {
                  best = individual
                  bestFitness = individualFitness
            }
      }
      return best
}
```

Implementasi genetic_algorithm.go adalah sebagai berikut.

Parameter	
Nama	**Deskripsi**

`generation *cube.Generation`	Generasi awal yang berisi populasi individu
`maxIterations int`	Jumlah maksimum iterasi yang digunakan untuk proses pencarian solusi optimal.
Variabel	
Nama	**Deskripsi**
`currentGeneration *cube.Generation`	Menyimpan generasi aktif yang sedang mengalami evolusi dalam setiap iterasi *genetic algorithm*
Method	
Nama	**Deskripsi**
`GeneticAlgorithm(generation *cube.Generation, maxIterations int) *cube.Generation`	Menjalankan proses evolusi untuk setiap generasi hingga batas iterasi tercapai atau solusi optimal ditemukan
Algoritma	

```
package localsearch

import (
	"diagonalmagiccube/cube"
)

func GeneticAlgorithm(generation *cube.Generation, maxIterations int)
*cube.Generation {
	currentGeneration := generation
	cube.SearchBestAVGScore(currentGeneration)
	for i := 0; i < maxIterations; i++ {
		currentGeneration = cube.Evolution(currentGeneration)
		cube.SearchBestAVGScore(currentGeneration)
```

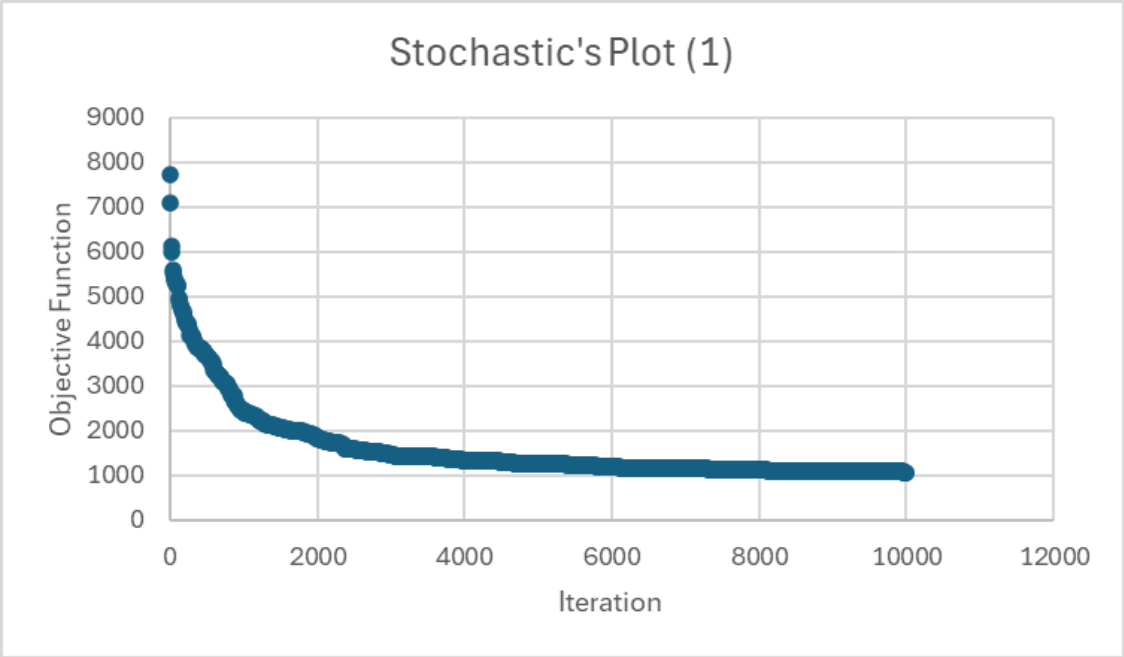
```
        }
        return currentGeneration
}
```

2.4 Hasil Eksperimen

2.4.1 Stochastic Hill Climbing

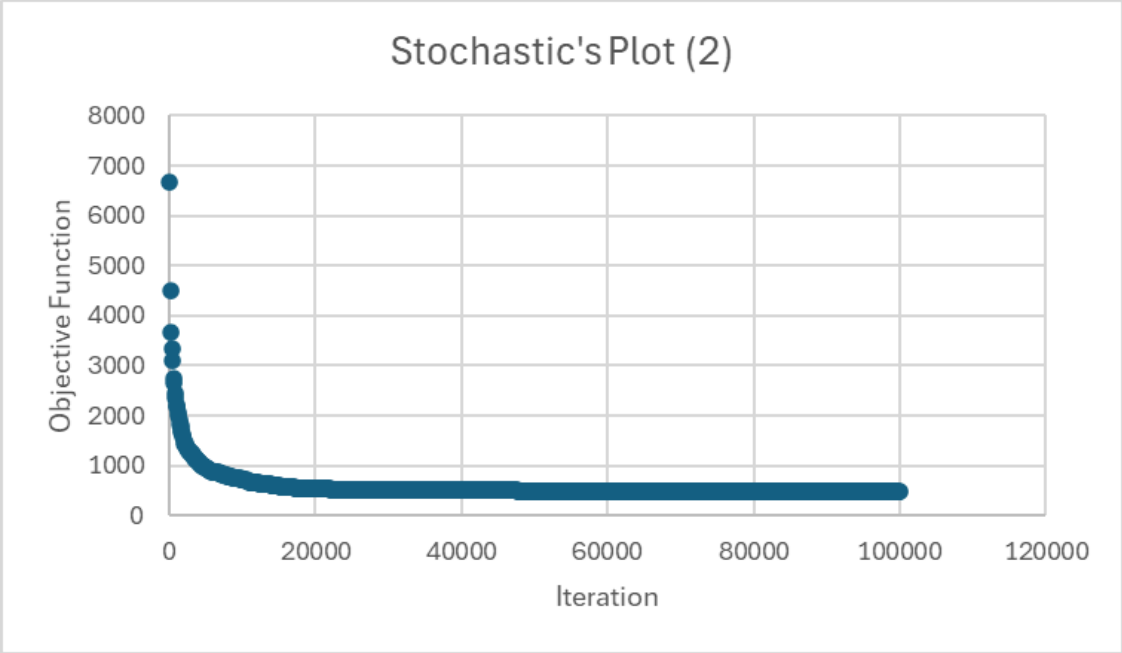
Hasil Eksperimen	
Percobaan 1 dengan 10.000 iterasi	
Initial State	**Cube Sequence:** [124 107 35 84 54 19 18 82 20 79 1 75 51 73 119 85 76 91 122 64 50 105 96 92 100 88 37 40 121 111 63 61 3 99 72 68 104 25 42 26 22 114 57 118 49 41 115 39 8 83 97 103 112 125 81 43 117 102 98 67 116 101 12 65 90 87 9 113 17 66 36 108 123 120 78 53 7 27 4 29 80 58 34 77 56 6 2 38 11 55 70 33 46 15 71 106 21 14 52 69 10 48 86 93 74 28 24 45 44 13 89 5 47 30 60 59 62 94 23 95 31 109 16 32 110]
	Objective Function Score: 7739
	Jumlah Elemen yang Memenuhi *Magic Number*: Rows: 0 Columns: 1 Pillars: 0 Plane Diagonals: 0 Space Diagonals: 0
Final State	**Cube Sequence:** [34 113 62 96 13 101 23 76 4 112 42 47 106 49 71 87 110 2 81 39 51 25 68 102 73 120 16 17 55 104

	6 33 90 103 98 88 125 36 22 24 43 20 69 100 80 56 121 105 32 10 28 26 118 21 109 54 115 18 82 46 108 41 53 57 58 117 64 99 31 14 8 65 27 124 89 92 97 52 70 5 75 78 15 119 29 1 91 60 50 114 59 38 67 40 111 94 11 122 37 45 35 61 66 85 77 86 74 116 7 30 79 9 48 123 44 12 84 83 63 72 107 93 3 19 95]
	Objective Function Score: 1075
	Jumlah Elemen yang Memenuhi *Magic Number*: Rows: 6 Columns: 6 Pillars: 3 Plane Diagonals: 4 Space Diagonals: 0
	Durasi Proses Pencarian: 27.5589ms
Plot Nilai *Objective Function* terhadap Banyak Iterasi	



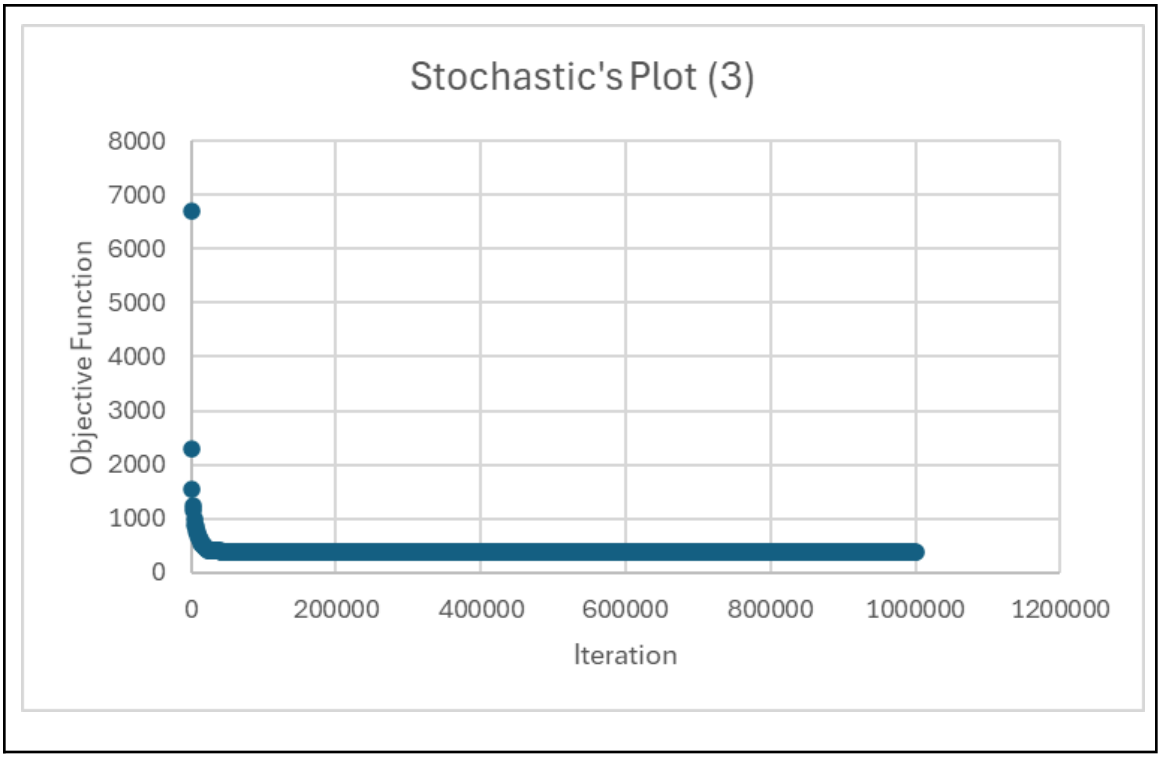
Percobaan 2 dengan 100.000 iterasi	
Initial State	**Cube Sequence:** [94 104 66 53 74 82 7 27 50 60 113 49 24 119 45 64 108 125 70 22 34 97 81 32 69 71 1 67 101 95 107 15 115 106 6 87 41 96 17 28 26 63 35 99 73 5 47 40 44 90 14 37 42 109 33 13 93 19 52 46 110 98 116 88 84 51 83 103 78 92 30 77 111 18 75 57 121 76 117 38 39 62 20 36 61 79 72 54 23 100 89 55 11 118 112 29 80 68 4 10 122 31 102 8 123 2 86 124 9 21 3 65 43 105 56 25 48 16 120 12 58 85 91 59 114]
	Objective Function Score: 6669
	Jumlah Elemen yang Memenuhi *Magic Number*: Rows: 0 Columns: 1 Pillars: 0

	Plane Diagonals: 0 Space Diagonals: 0
Final State	**Cube Sequence:** [103 107 2 47 56 92 8 82 46 87 67 23 74 81 83 34 115 116 40 11 18 62 55 101 79 36 58 97 96 28 17 39 118 108 32 90 105 21 31 70 77 64 61 53 60 95 49 19 27 124 100 4 3 98 109 120 123 9 50 13 52 75 76 71 41 6 43 125 10 122 37 69 93 86 30 16 112 119 48 20 66 89 24 63 73 38 42 114 25 91 88 15 12 85 117 106 57 44 94 14 51 35 99 26 102 22 54 84 45 110 72 68 29 113 33 111 78 1 121 5 59 80 104 7 65]
	Objective Function Score: 467
	Jumlah Elemen yang Memenuhi *Magic Number*: Rows: 9 Columns: 14 Pillars: 14 Plane Diagonals: 5 Space Diagonals: 0
	Durasi Proses Pencarian: 259.8962ms
Plot Nilai *Objective Function* terhadap Banyak Iterasi	



Percobaan 3 dengan 1.000.000 iterasi	
Initial State	**Cube Sequence:** [118 119 88 54 7 102 87 108 18 123 64 4 56 74 81 72 96 114 104 2 39 40 71 91 15 49 58 101 68 57 20 48 35 107 59 69 41 42 85 95 21 52 117 31 89 92 33 116 93 61 28 1 26 45 36 98 106 44 53 75 60 105 62 22 103 25 113 90 83 99 12 34 124 32 112 97 8 5 79 47 23 110 55 111 14 51 16 30 67 50 6 17 13 3 70 82 38 77 37 9 100 115 121 109 46 120 76 27 94 65 122 73 78 10 29 125 66 11 43 80 63 24 19 84 86]
	Objective Function Score: 6698
	Jumlah Elemen yang Memenuhi *Magic Number*: Rows: 0 Columns: 0 Pillars: 0

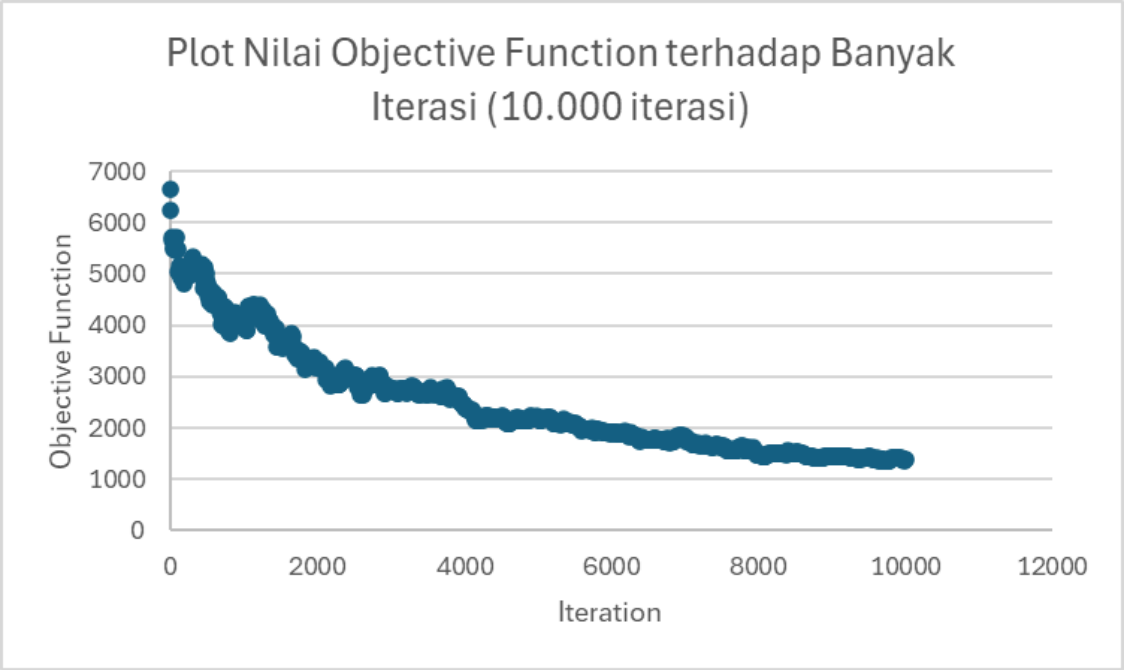
	Plane Diagonals: 0 Space Diagonals: 0
Final State	**Cube Sequence:** [51 108 42 30 95 101 4 117 22 70 9 88 93 87 41 76 19 39 116 65 84 99 25 61 44 80 100 106 18 11 27 48 71 109 56 50 79 26 81 82 37 45 112 58 62 121 43 2 49 104 14 5 40 123 118 124 111 17 23 38 120 66 59 24 46 28 105 86 60 35 10 29 113 85 78 103 89 20 69 34 21 91 12 94 97 6 7 73 115 114 102 36 90 33 54 83 92 122 3 15 67 13 107 75 57 31 63 98 68 55 125 77 64 8 32 72 110 1 47 96 16 52 53 119 74]
	Objective Function Score: 372
	Jumlah Elemen yang Memenuhi *Magic Number*: Rows: 14 Columns: 11 Pillars: 10 Plane Diagonals: 8 Space Diagonals: 1
	Durasi Proses Pencarian: 2.4178801s
Plot Nilai *Objective Function* terhadap Banyak Iterasi	



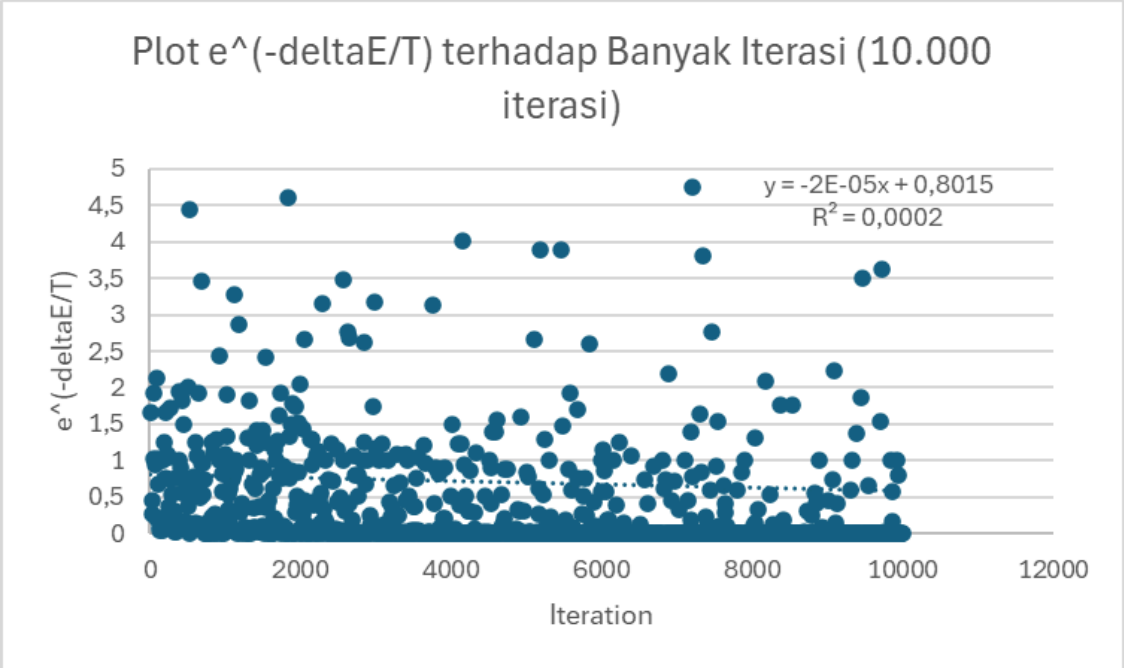
2.4.2 Simulated Annealing

Hasil Eksperimen	
Percobaan 1 dengan 10.000 iterasi	
Initial State	**Cube Sequence:** [73 86 16 58 27 67 33 35 123 105 111 48 12 62 99 54 101 70 41 6 11 13 31 51 14 120 57 103 122 87 89 109 106 38 65 3 8 10 66 102 47 125 64 19 59 100 69 88 23 74 121 93 29 92 80 68 45 79 116 26 50 5 119 55 24 124 108 61 1 83 28 56 2 77 115 25 117 90 37 107 44 97 43 21 9 114 60 110 81 94 4 98 30 78 95 46 22 52 82 84 18 72 96 85 17 20 71 53 7 112 118 113 15 40 42 75 49 76 104 91 34 36 39 32 63]
	Objective Function Score: 6650

	Jumlah Elemen yang Memenuhi *Magic Number*: Rows: 0 Columns: 0 Pillars: 0 Plane Diagonals: 0 Space Diagonals: 0
Final State	**Cube Sequence:** [116 60 62 6 63 38 18 108 98 50 35 94 58 105 13 90 43 79 11 78 41 68 3 99 109 26 91 47 115 42 81 37 93 59 49 9 51 31 80 114 106 102 101 27 1 89 40 53 23 110 87 2 14 103 124 73 96 76 46 15 82 48 75 57 44 25 45 28 83 125 52 123 118 21 10 70 122 95 12 7 55 61 19 85 92 97 64 29 66 65 4 88 56 69 111 104 5 121 71 67 20 39 113 54 86 77 74 16 34 112 84 36 120 8 72 100 32 33 107 22 30 117 24 119 17]
	Objective Function Score: 1374
	Jumlah Elemen yang Memenuhi *Magic Number*: Rows: 0 Columns: 0 Pillars: 1 Plane Diagonals: 0 Space Diagonals: 1
	Durasi Proses Pencarian: 45.027ms
	Frekuensi "Stuck" di *Local Optima*: 1241
Plot Nilai *Objective Function* terhadap Banyak Iterasi	



Plot Nilai $e^{(\frac{-\Delta E}{T})}$ terhadap Banyak Iterasi



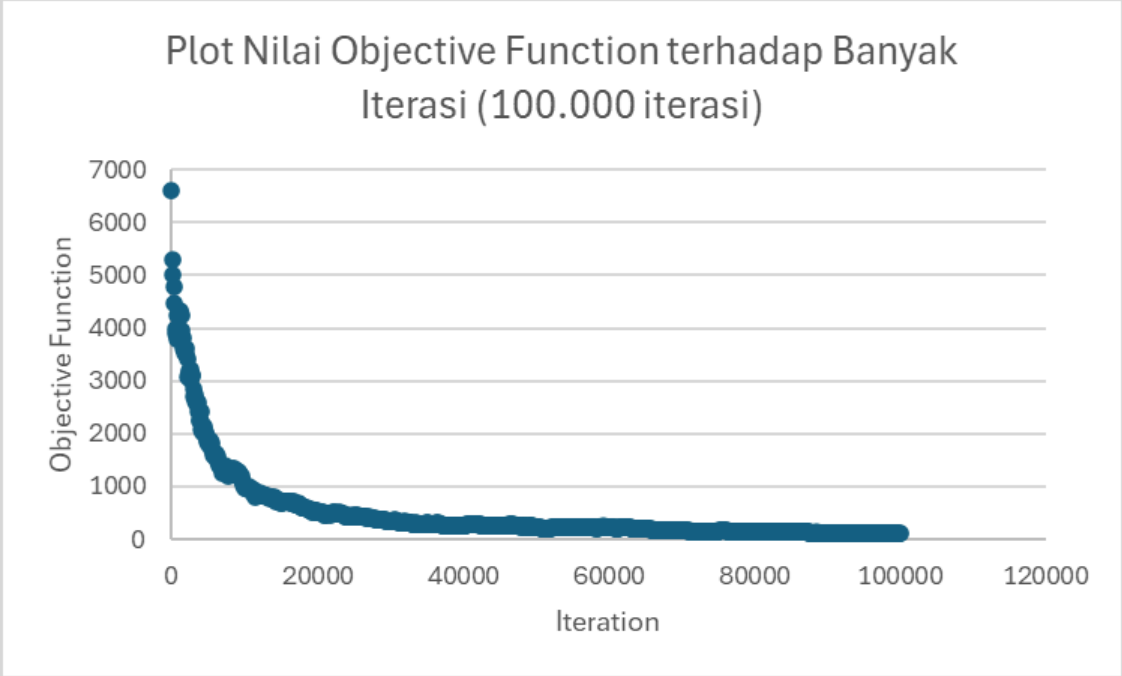
Percobaan 2 dengan 100.000 iterasi

Initial State	**Cube Sequence:**

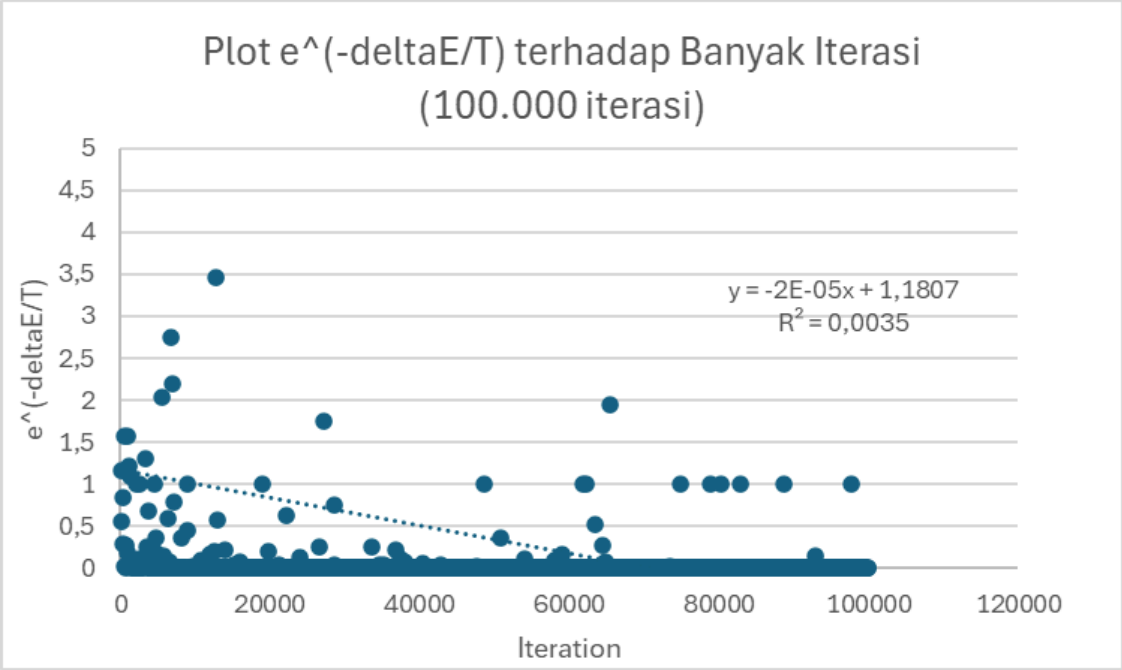
	[79 18 66 14 94 19 122 62 117 95 113 34 53 68 44 88 63 61 41 90 112 75 45 29 125 49 26 60 22 85 54 6 108 69 40 78 15 58 30 52 76 118 11 43 100 36 120 87 74 13 57 119 10 73 35 51 16 83 105 42 106 65 55 89 110 4 48 9 31 24 114 28 3 70 96 81 12 20 103 101 2 59 46 124 92 37 23 17 32 102 50 5 99 111 107 93 56 77 38 115 33 104 71 72 27 67 91 39 47 116 7 98 97 84 109 123 8 25 64 80 121 82 86 1 21]
	Objective Function Score: 6598
	Jumlah Elemen yang Memenuhi *Magic Number*: Rows: 0 Columns: 0 Pillars: 0 Plane Diagonals: 0 Space Diagonals: 1
Final State	**Cube Sequence:** [83 5 39 121 68 125 71 4 2 113 10 57 90 42 114 48 107 108 52 1 47 77 72 100 19 36 123 93 17 46 29 25 101 84 76 115 80 69 37 14 73 56 6 94 85 63 32 45 82 91 24 55 7 118 111 116 97 59 26 16 67 122 64 9 53 3 8 75 112 117 105 33 110 50 18 74 120 51 41 30 31 44 49 95 96 40 21 65 109 79 104 60 88 43 22 66 70 62 28 89 99 13 124 20 61 15 78 103 106 11 81 35 27 119 54 86 87 38 12 92 34 102 23 58 98]
	Objective Function Score: 119
	Jumlah Elemen yang Memenuhi *Magic Number*:

	Rows: 6 Columns: 13 Pillars: 12 Plane Diagonals: 13 Space Diagonals: 0
	Durasi Proses Pencarian: 216.1312ms
	Frekuensi "Stuck" di *Local Optima*: 2795

Plot Nilai *Objective Function* terhadap Banyak Iterasi

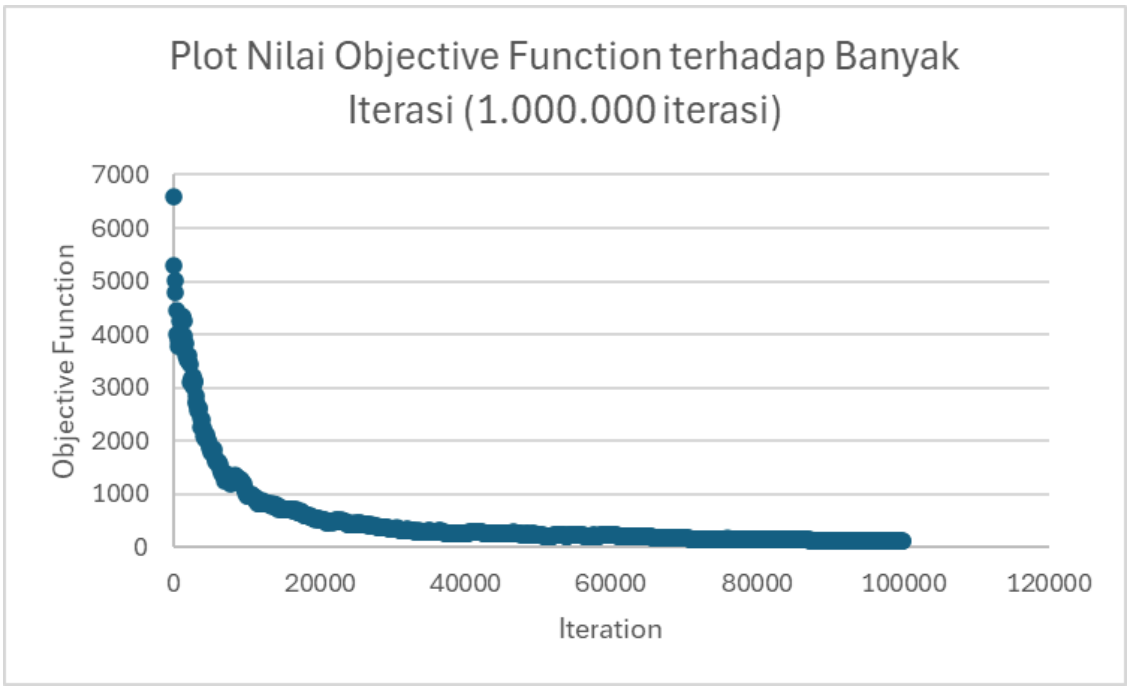


Plot Nilai $e^{(\frac{-\Delta E}{T})}$ terhadap Banyak Iterasi

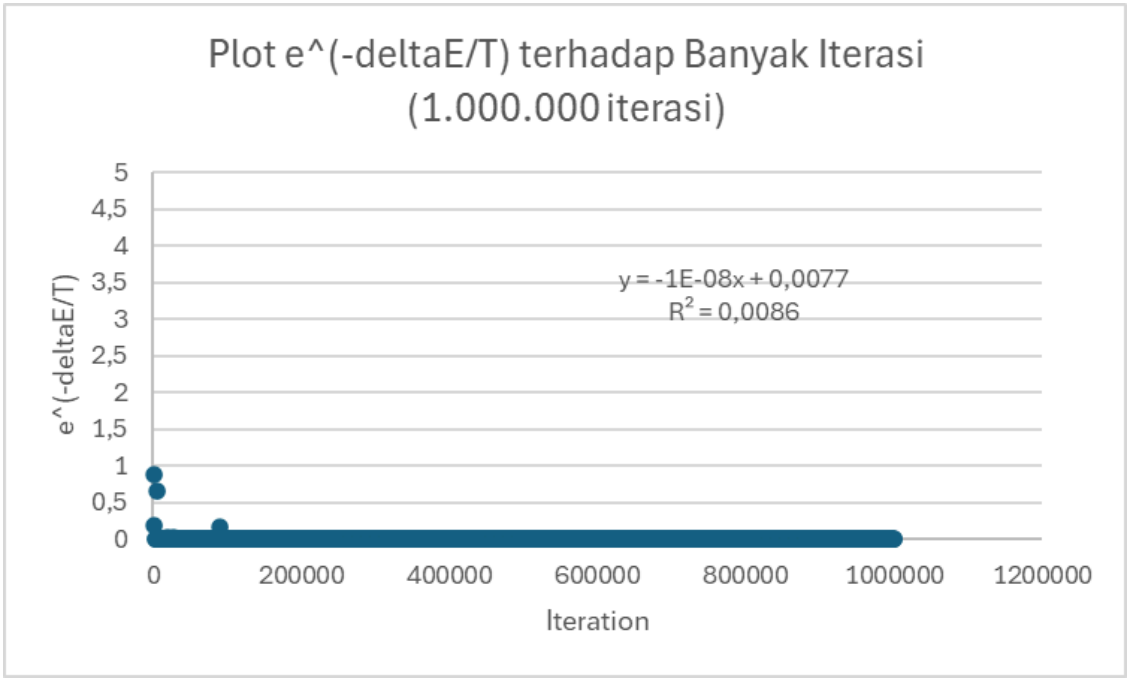


Percobaan 3 dengan 1.000.000 iterasi	
Initial State	**Cube Sequence:** [113 85 69 25 52 16 75 28 36 24 119 39 2 86 76 117 95 13 107 50 109 96 1 116 83 87 11 77 12 62 48 49 45 44 21 122 18 93 102 33 47 101 15 106 27 112 71 115 53 6 99 38 72 54 78 123 9 108 17 74 20 63 98 66 10 43 7 124 57 22 92 14 41 59 118 84 70 35 81 80 8 103 5 67 34 31 79 68 4 111 26 88 97 51 30 55 110 89 120 42 61 104 29 60 32 94 100 3 90 73 23 65 125 19 82 64 105 114 91 37 56 40 46 58 121]
	Objective Function Score: 6938
	Jumlah Elemen yang Memenuhi *Magic Number*: Rows: 1 Columns: 0

	Pillars: 0 Plane Diagonals: 0 Space Diagonals: 0
Final State	**Cube Sequence:** [17 75 106 63 54 86 12 65 114 34 90 97 56 3 71 53 24 81 115 43 67 107 7 22 113 88 84 45 29 69 50 104 41 87 33 60 28 27 74 124 83 94 91 31 16 38 5 111 95 66 76 21 92 46 80 49 121 8 85 52 112 11 62 110 20 30 40 116 10 119 48 122 37 64 44 102 109 14 51 39 23 77 101 18 96 6 61 98 93 58 79 55 25 36 120 105 13 78 117 2 32 26 59 125 73 108 1 100 9 99 47 118 72 35 42 70 103 4 123 15 57 68 82 19 89]
	Objective Function Score: 100
	Jumlah Elemen yang Memenuhi *Magic Number*: Rows: 15 Columns: 14 Pillars: 17 Plane Diagonals: 14 Space Diagonals: 2
	Durasi Proses Pencarian: 2.0062245s
	Frekuensi "Stuck" di *Local Optima*: 10229
Plot Nilai *Objective Function* terhadap Banyak Iterasi	



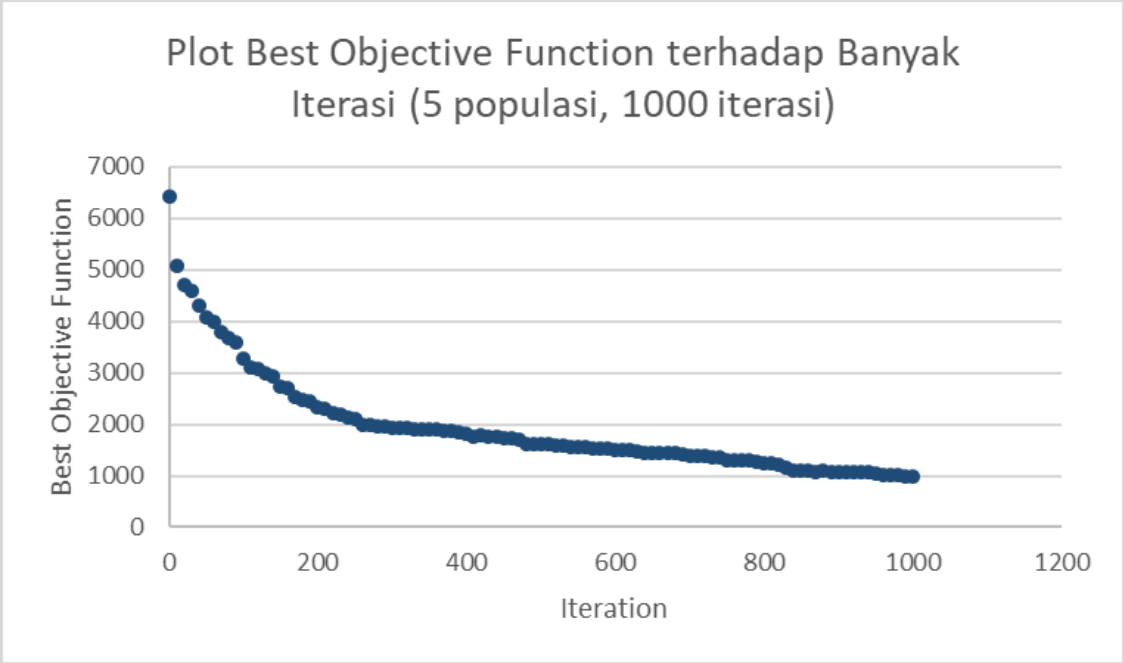
Plot Nilai $e^{(\frac{-\Delta E}{T})}$ terhadap Banyak Iterasi



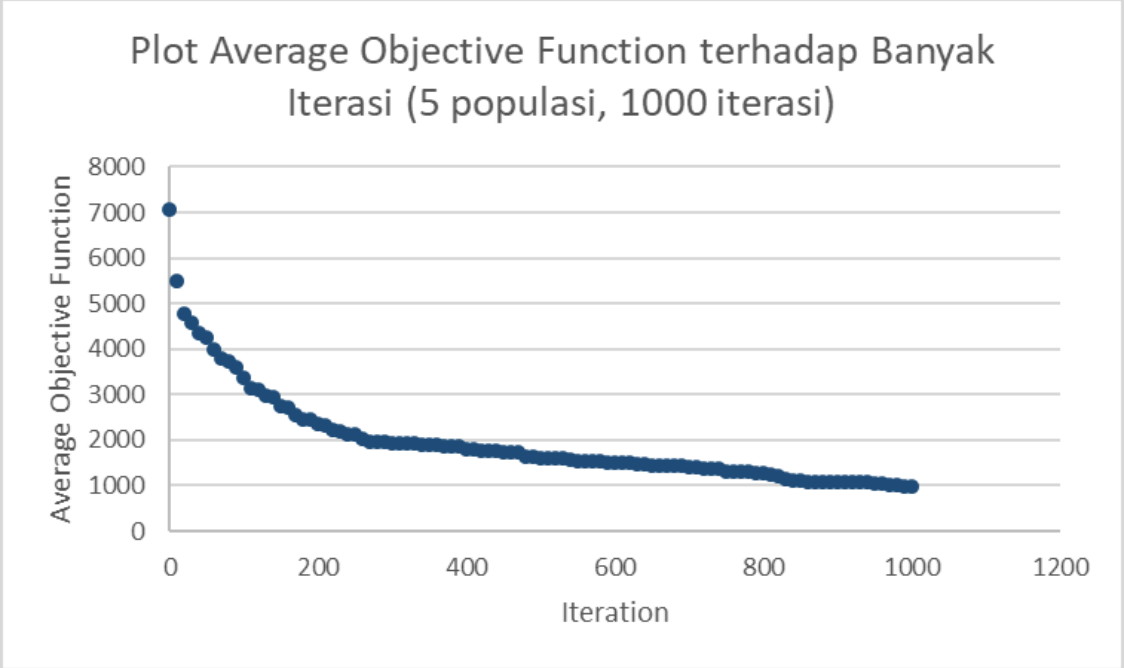
2.4.3 Genetic Algorithm

2.4.3.1 Eksperimen dengan Populasi Sebagai Variabel Kontrol

<table>
<tr><th colspan="2">Hasil Eksperimen</th></tr>
<tr><td colspan="2">Jumlah Populasi : 5</td></tr>
<tr><td colspan="2">Percobaan 1 dengan 1.000 iterasi</td></tr>
<tr><td rowspan="2">Initial State</td><td>Total Fitness: 7.09</td></tr>
<tr><td>Objective Function Score:
Population Size: 5
Individuals 1 Score: 6422
Individuals 2 Score: 7339
Individuals 3 Score: 7349
Individuals 4 Score: 7205
Individuals 5 Score: 7050</td></tr>
<tr><td rowspan="3">Final State</td><td>Total Fitness: 50.81</td></tr>
<tr><td>Objective Function Score:
Population Size: 5
Individuals 1 Score: 983
Individuals 2 Score: 983
Individuals 3 Score: 983
Individuals 4 Score: 983</td></tr>
<tr><td>Durasi Proses Pencarian: 1.4461969s</td></tr>
<tr><td colspan="2">Plot Nilai Objective Function Terbaik Terhadap Banyak Iterasi</td></tr>
</table>



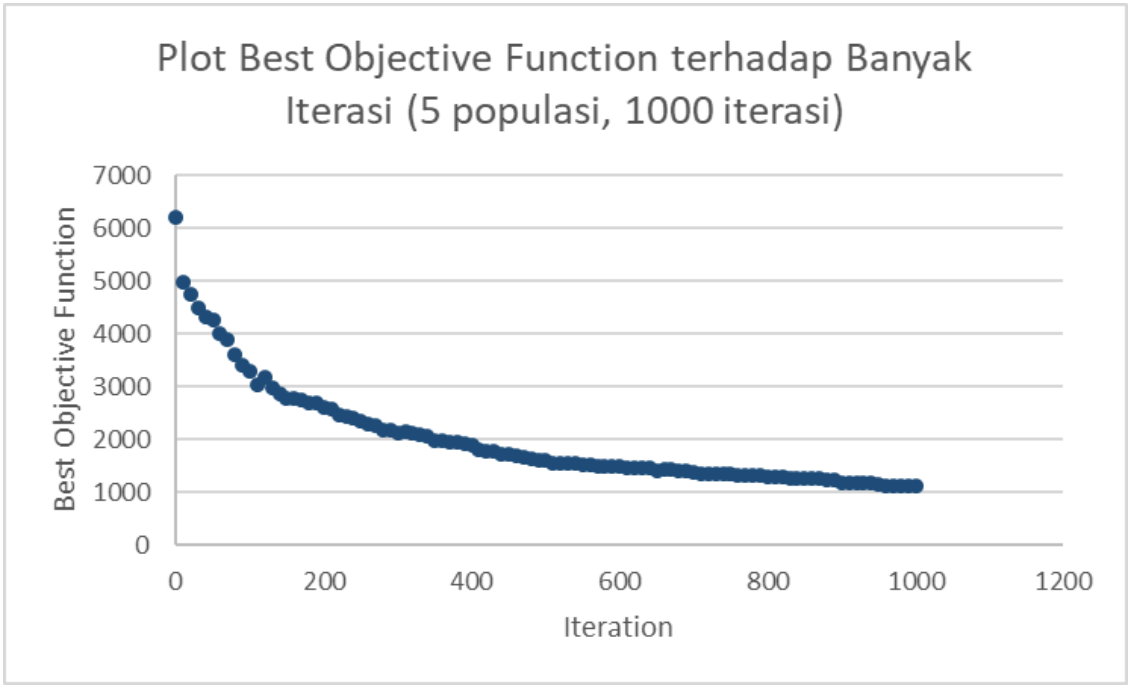
Plot Nilai *Objective Function* Rata-Rata Terhadap Banyak Iterasi



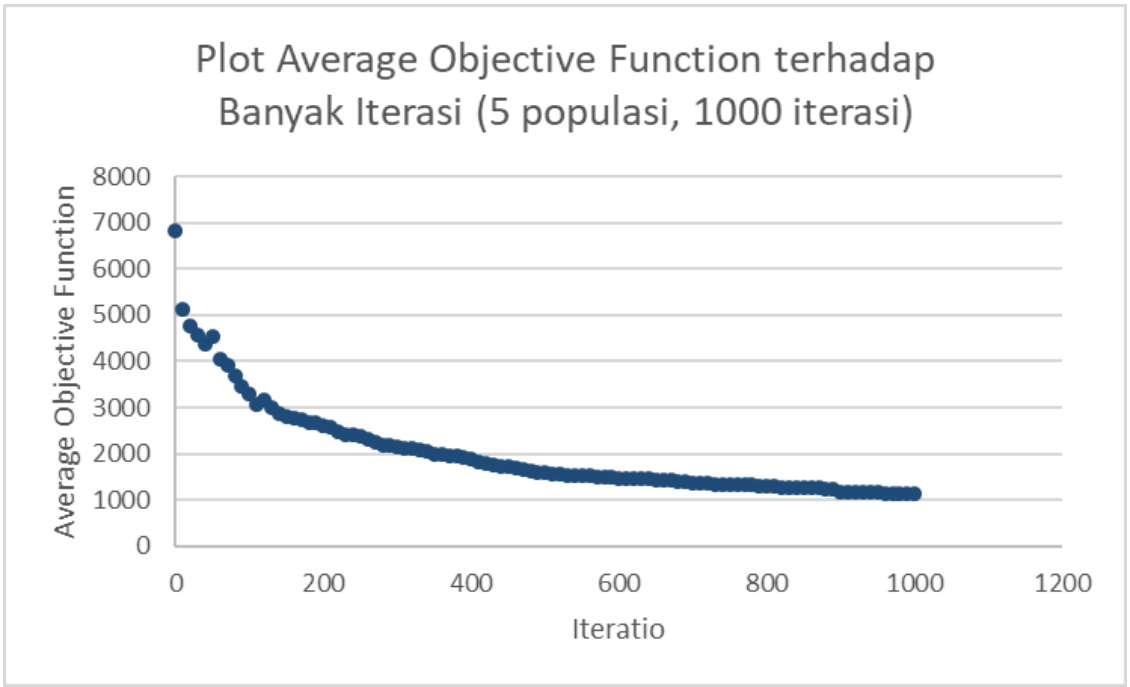
Percobaan 2 dengan 1.000 iterasi

Initial State	**Total Fitness:** 7.35

	Objective Function Score: Population Size: 5 Individuals 1 Score: 6924 Individuals 2 Score: 6885 Individuals 3 Score: 6883 Individuals 4 Score: 7195 Individuals 5 Score: 6209
Final State	**Total Fitness:** 44.48
	Objective Function Score: Population Size: 5 Individuals 1 Score: 1123 Individuals 2 Score: 1123 Individuals 3 Score: 1123 Individuals 4 Score: 1123
	Durasi Proses Pencarian: 1.6932733s
Plot Nilai Objective Function Terbaik Terhadap Banyak Iterasi	

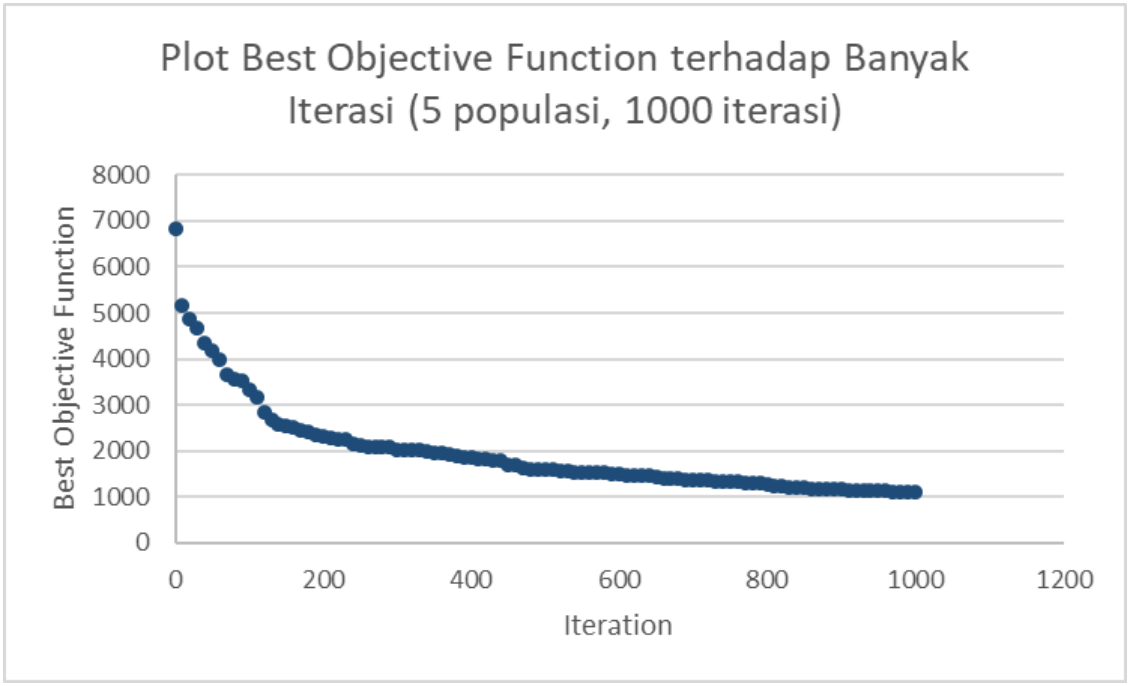


Plot Nilai *Objective Function* Rata-Rata Terhadap Banyak Iterasi

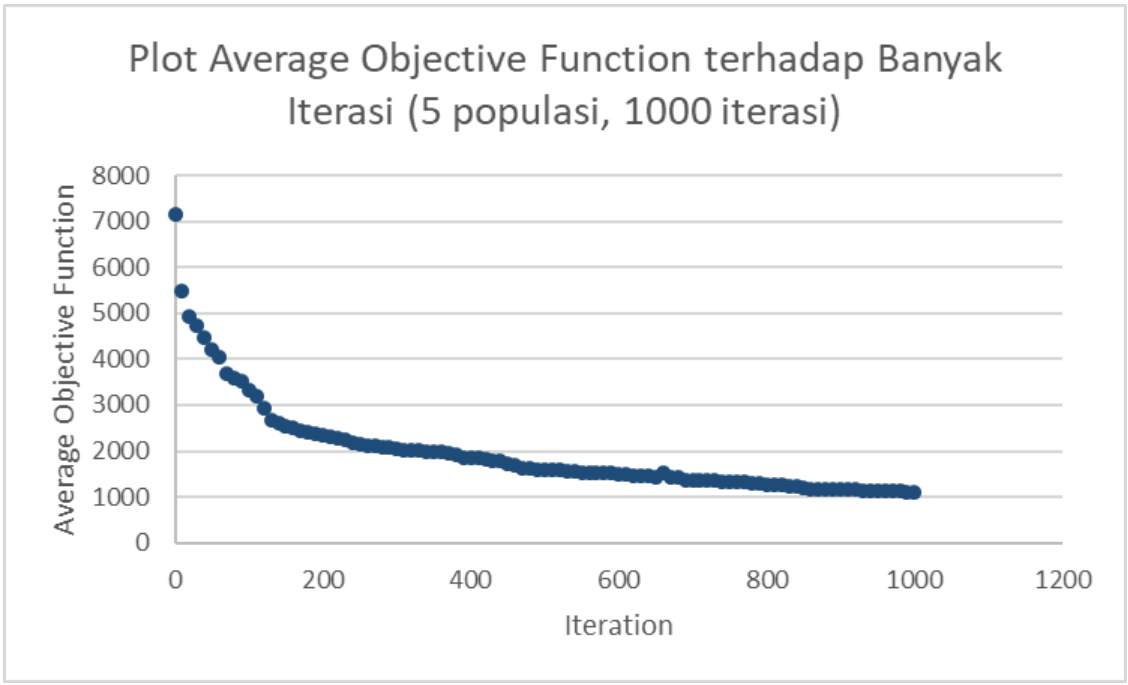


Percobaan 3 dengan 1.000 iterasi

Initial State	**Total Fitness:** 6.99
	Objective Function Score: Population Size: 5 Individuals 1 Score: 7240 Individuals 2 Score: 7537 Individuals 3 Score: 7299 Individuals 4 Score: 6926 Individuals 5 Score: 6820
Final State	**Total Fitness:** 45.05
	Objective Function Score: Population Size: 5 Individuals 1 Score: 1109 Individuals 2 Score: 1109 Individuals 3 Score: 1109
	Durasi Proses Pencarian: 1.5866177s
Plot Nilai Objective Function Terbaik Terhadap Banyak Iterasi	



Plot Nilai *Objective Function* Rata-Rata Terhadap Banyak Iterasi



Hasil Eksperimen

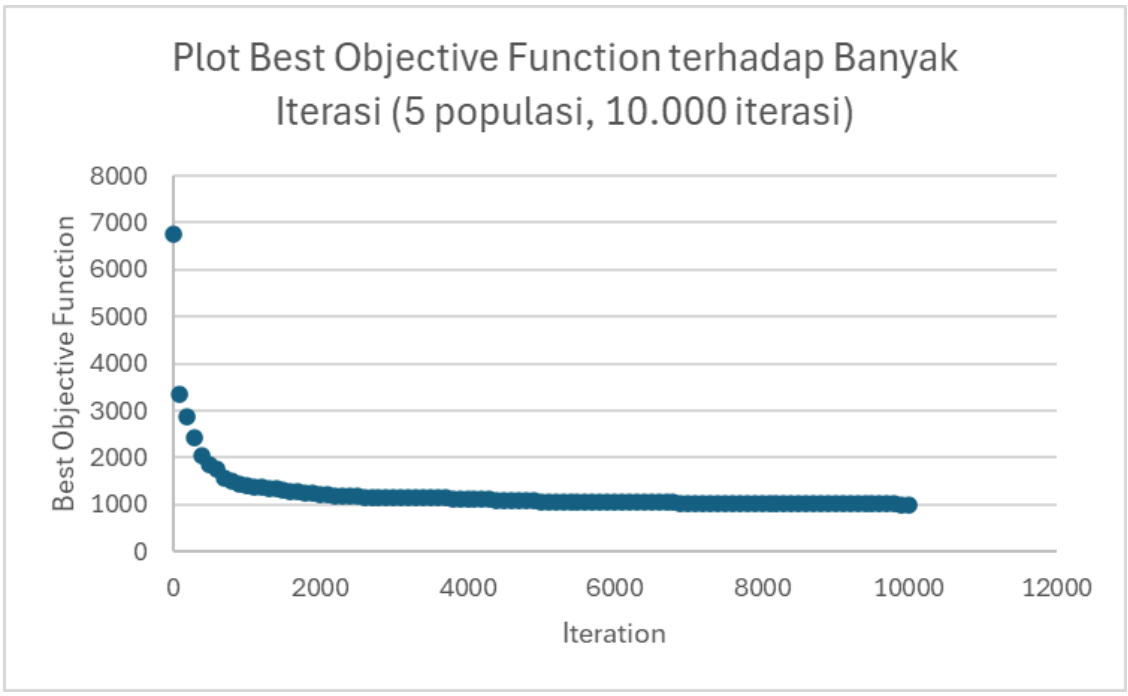
Jumlah Populasi : 5	
Percobaan 1 dengan 10.000 iterasi	
Initial State	**Total Fitness:** `6.95`

	Objective Function Score:
	`Population Size: 5`
	`Individuals 1 Score: 7368`
	`Individuals 2 Score: 7728`
	`Individuals 3 Score: 6745`
	`Individuals 4 Score: 7089`
	`Individuals 5 Score: 7124`

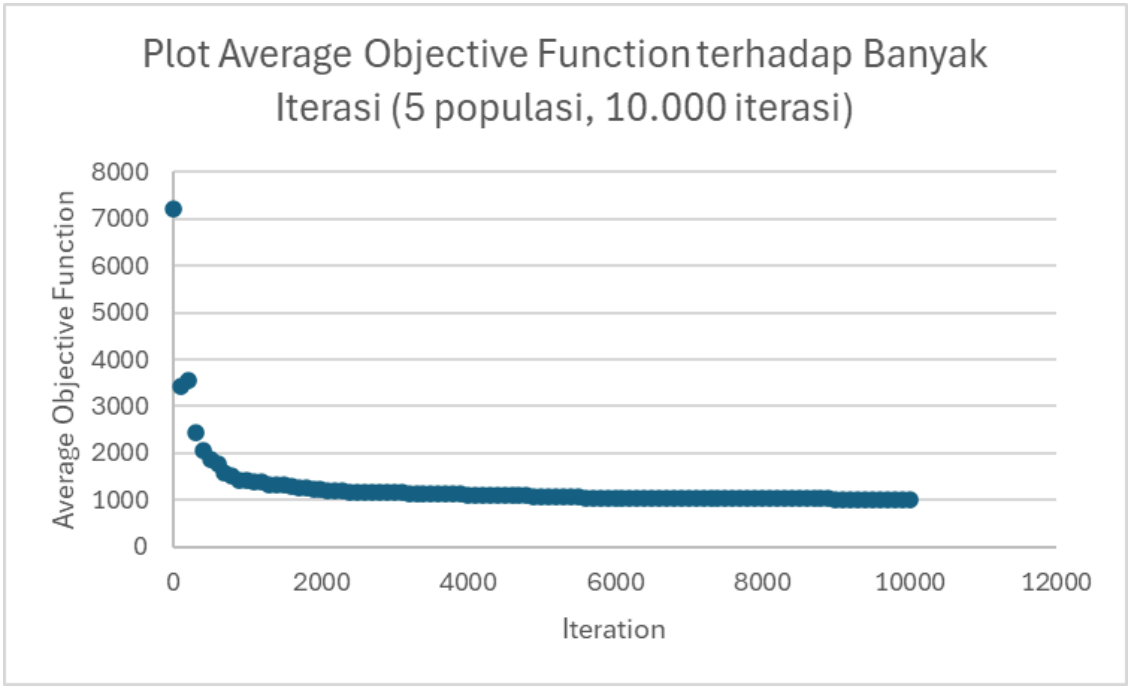
Final State	**Total Fitness:** `49.36`

	Objective Function Score:
	`Population Size: 5`
	`Individuals 1 Score: 1012`
	`Individuals 2 Score: 1012`
	`Individuals 3 Score: 1012`
	`Individuals 4 Score: 1012`
	`Individuals 5 Score: 1012`

	Durasi Proses Pencarian: `1.6834443s`
Plot Nilai Objective Function Terbaik Terhadap Banyak Iterasi	

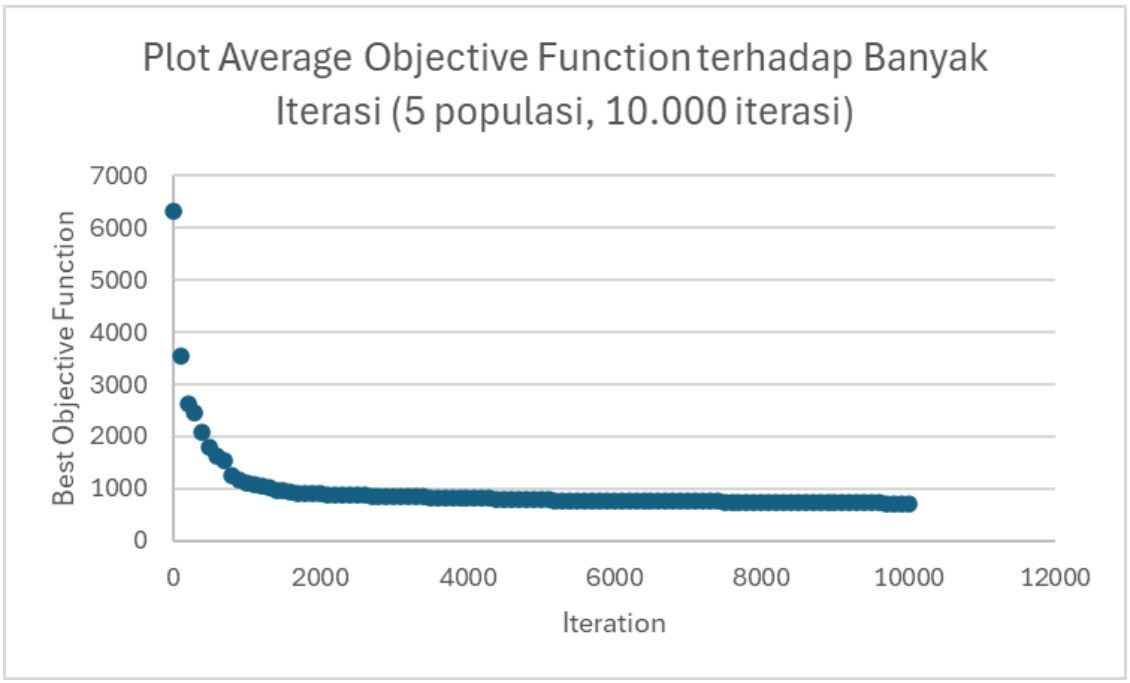


Plot Nilai *Objective Function* Rata-Rata Terhadap Banyak Iterasi

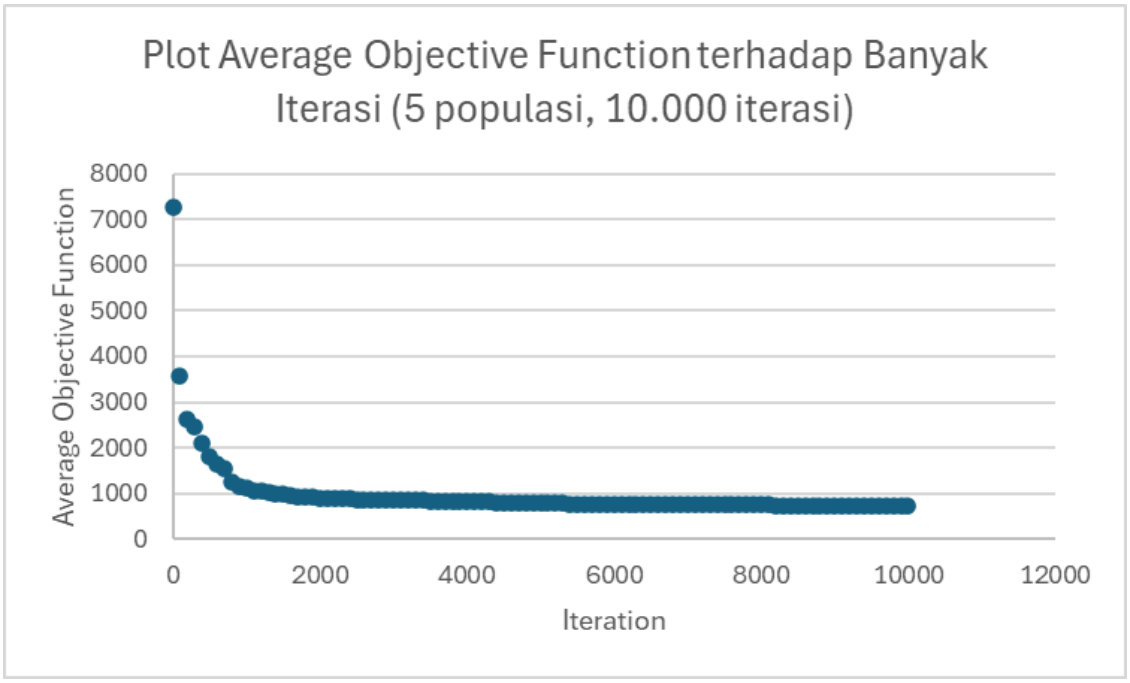


Percobaan 2 dengan 10.000 iterasi

Initial State	**Total Fitness:** 6.96
	Objective Function Score: Population Size: 5 Individuals 1 Score: 7824 Individuals 2 Score: 6676 Individuals 3 Score: 6307 Individuals 4 Score: 7217 Individuals 5 Score: 8238
Final State	**Total Fitness:** 69.35
	Objective Function Score: Population Size: 5 Individuals 1 Score: 720 Individuals 2 Score: 720 Individuals 3 Score: 720 Individuals 4 Score: 720 Individuals 5 Score: 720
	Durasi Proses Pencarian: 1.3952765s
Plot Nilai Objective Function Terbaik Terhadap Banyak Iterasi	

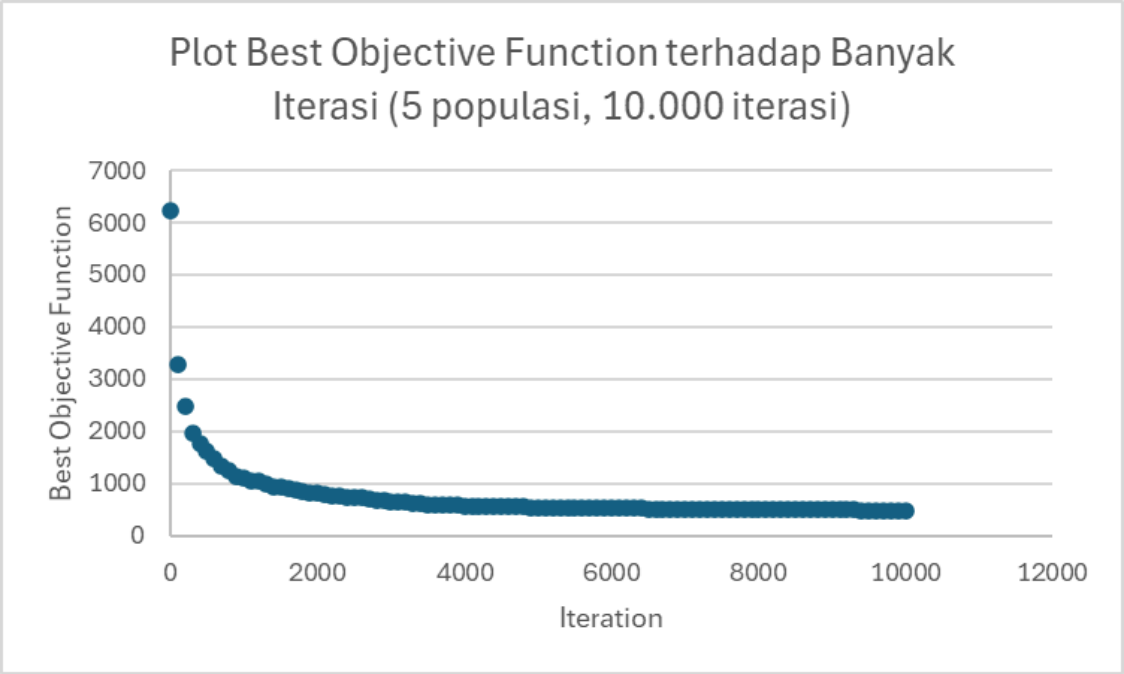


Plot Nilai *Objective Function* Rata-Rata Terhadap Banyak Iterasi

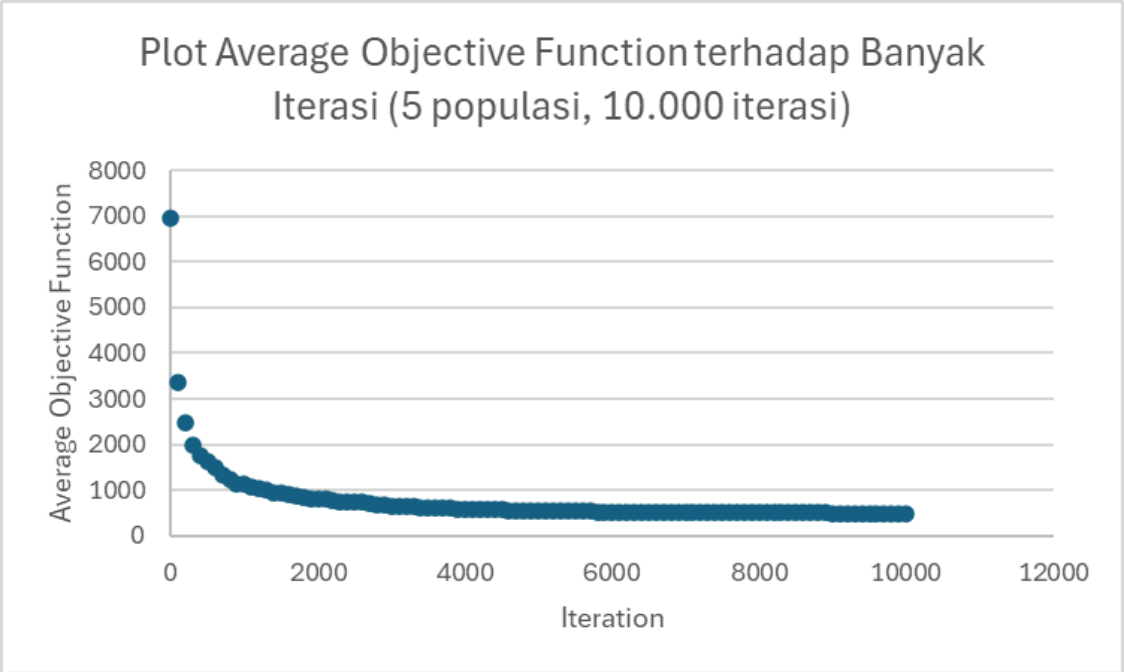


Percobaan 3 dengan 10.000 iterasi

Initial State	**Total Fitness:** 7.24
	Objective Function Score: Population Size: 5 Individuals 1 Score: 6229 Individuals 2 Score: 7652 Individuals 3 Score: 6509 Individuals 4 Score: 6839 Individuals 5 Score: 7508
Final State	**Total Fitness:** 101.63
	Objective Function Score: Population Size: 5 Individuals 1 Score: 491 Individuals 2 Score: 491 Individuals 3 Score: 491 Individuals 4 Score: 491 Individuals 5 Score: 491
	Durasi Proses Pencarian: 1.5241829s
Plot Nilai Objective Function Terbaik Terhadap Banyak Iterasi	

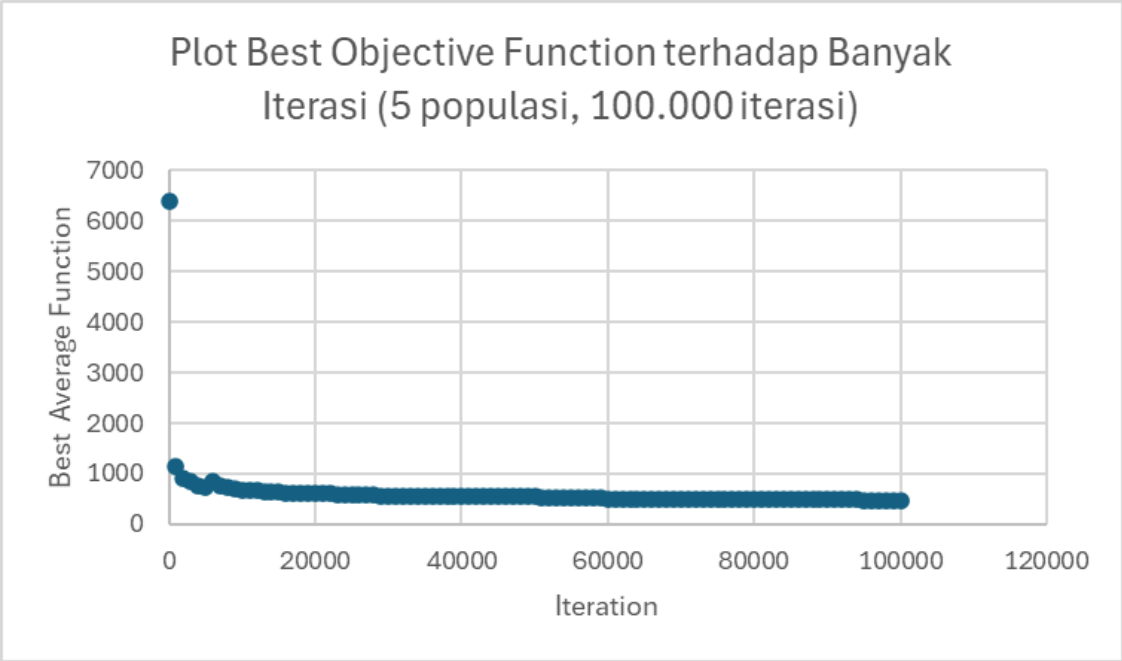


Plot Nilai *Objective Function* Rata-Rata Terhadap Banyak Iterasi

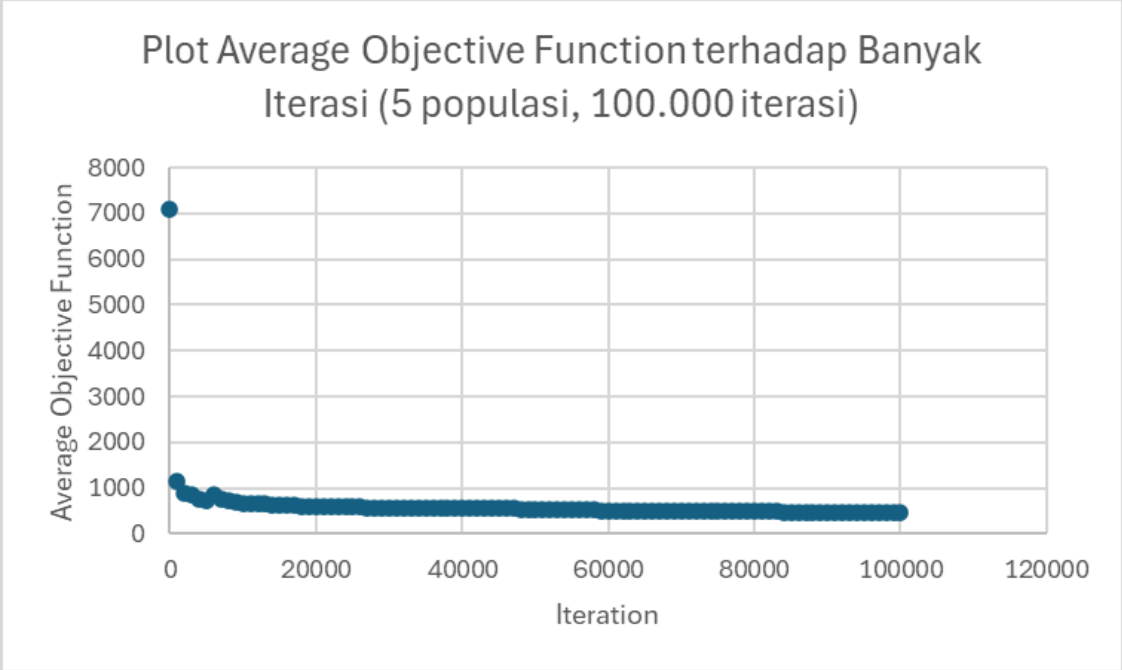


Hasil Eksperimen

Jumlah Populasi : 5	
Percobaan 1 dengan 100.000 iterasi	
Initial State	**Total Fitness:** 7.07
	Objective Function Score: Population Size: 5 Individuals 1 Score: 7770 Individuals 2 Score: 6397 Individuals 3 Score: 7172 Individuals 4 Score: 6879 Individuals 5 Score: 7285
Final State	**Total Fitness:** 103.73
	Objective Function Score: Population Size: 5 Individuals 1 Score: 481 Individuals 2 Score: 481 Individuals 3 Score: 481 Individuals 4 Score: 481 Individuals 5 Score: 481
	Durasi Proses Pencarian: 18.7813475s
Plot Nilai Objective Function Terbaik Terhadap Banyak Iterasi	



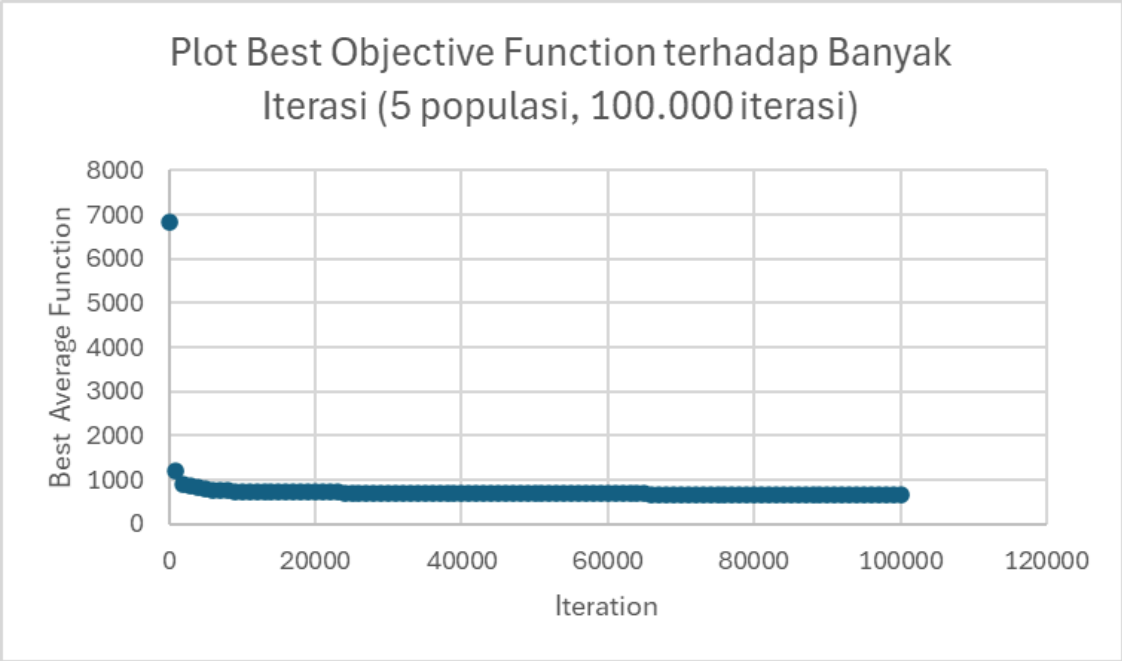
Plot Nilai *Objective Function* Rata-Rata Terhadap Banyak Iterasi



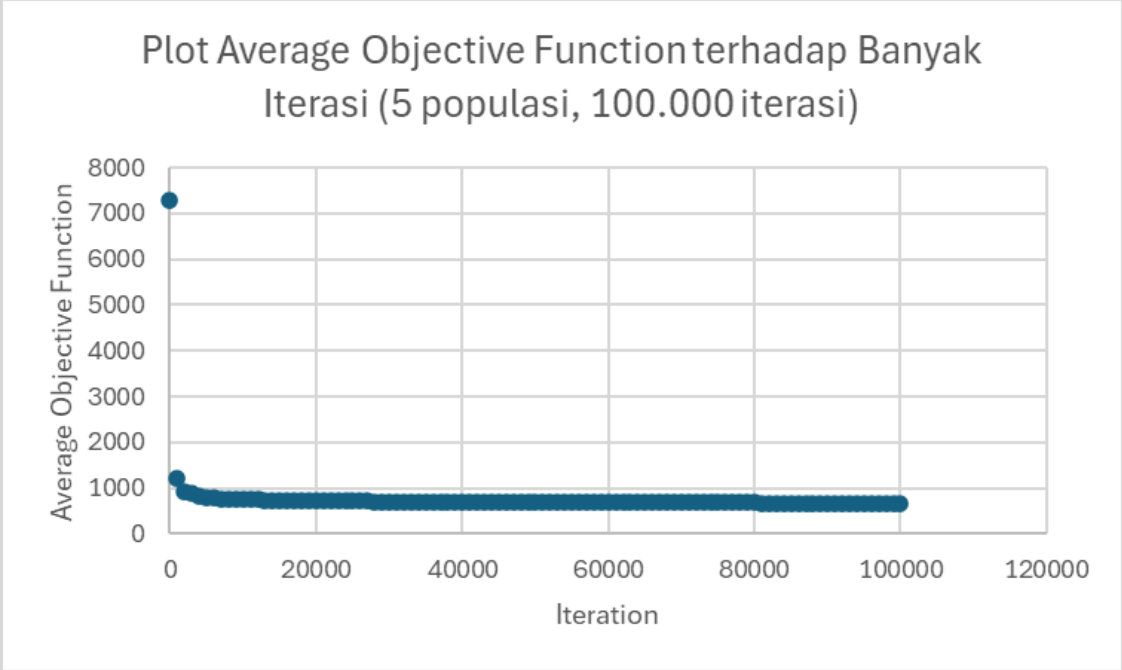
Percobaan 2 dengan 100.000 iterasi

Initial State	**Total Fitness:** `6.88`

	Objective Function Score: Population Size: 5 Individuals 1 Score: 7182 Individuals 2 Score: 7373 Individuals 3 Score: 7281 Individuals 4 Score: 6825 Individuals 5 Score: 7728
Final State	**Total Fitness:** 73.75
	Objective Function Score: Population Size: 5 Individuals 1 Score: 677 Individuals 2 Score: 677 Individuals 3 Score: 677 Individuals 4 Score: 677 Individuals 5 Score: 677
	Durasi Proses Pencarian: 20.1440206s
Plot Nilai Objective Function Terbaik Terhadap Banyak Iterasi	



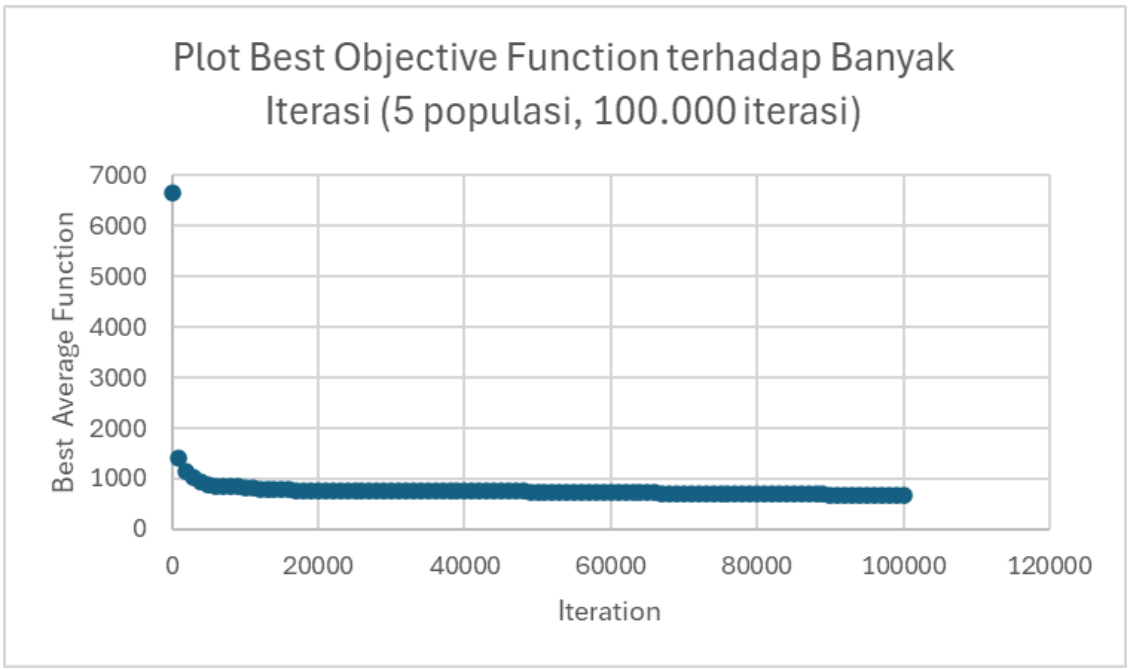
Plot Nilai *Objective Function* Rata-Rata Terhadap Banyak Iterasi



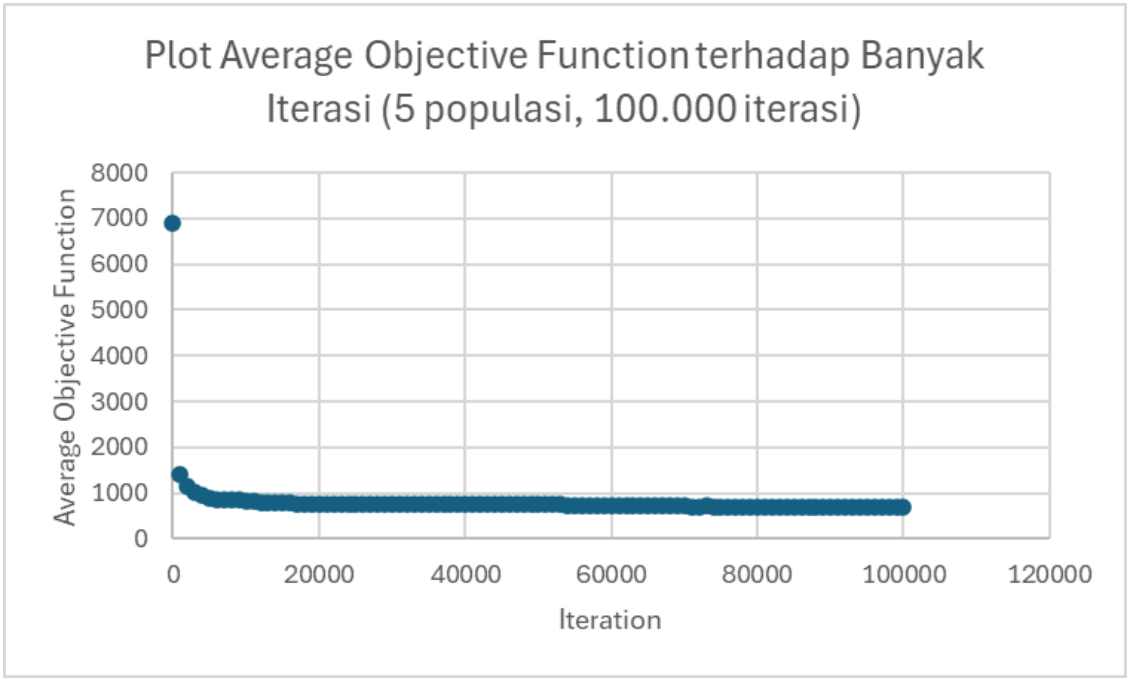
Percobaan 3 dengan 100.000 iterasi

Initial State	**Total Fitness:** 7.27

	Objective Function Score: Population Size: 5 Individuals 1 Score: 6936 Individuals 2 Score: 7049 Individuals 3 Score: 6658 Individuals 4 Score: 6806 Individuals 5 Score: 6968
Final State	**Total Fitness:** 73.31
	Objective Function Score: Population Size: 5 Individuals 1 Score: 681 Individuals 2 Score: 681 Individuals 3 Score: 681 Individuals 4 Score: 681 Individuals 5 Score: 681
	Durasi Proses Pencarian: 14.8432068s
Plot Nilai Objective Function Terbaik Terhadap Banyak Iterasi	

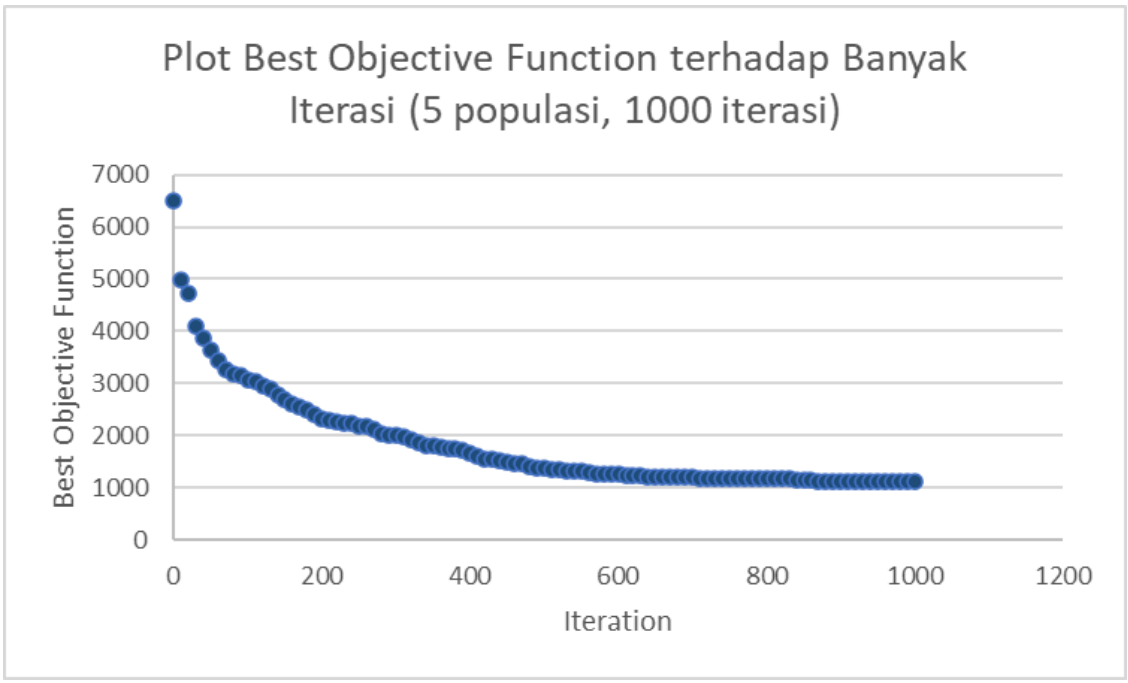


Plot Nilai *Objective Function* Rata-Rata Terhadap Banyak Iterasi

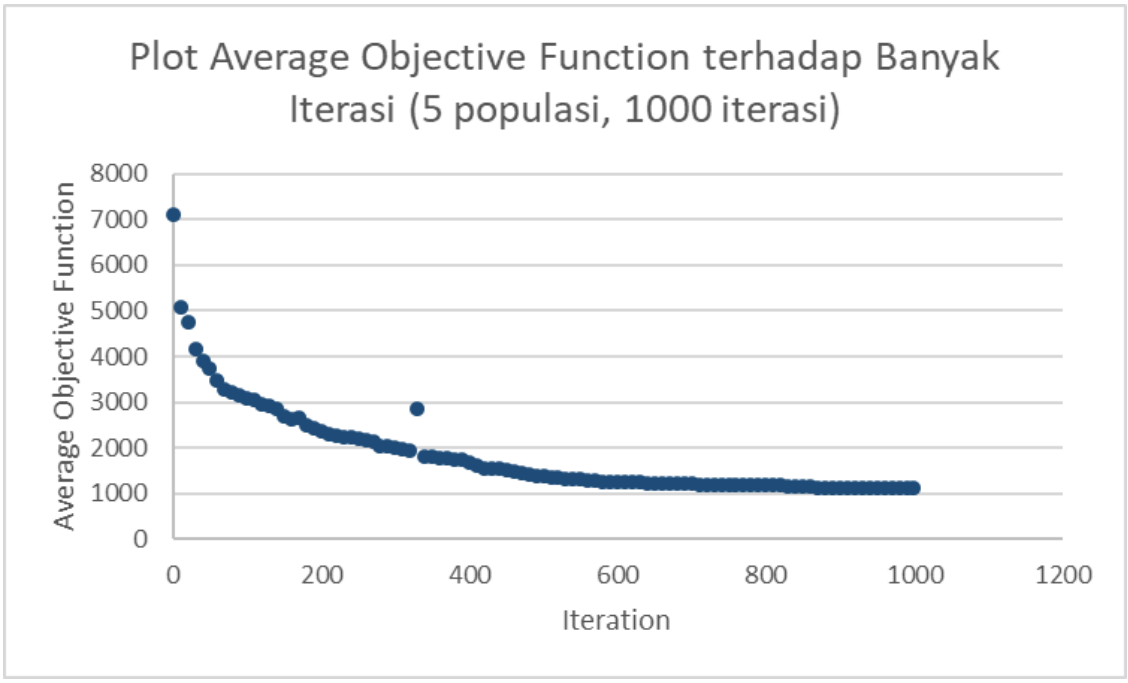


2.4.3.2 Eksperimen dengan Iterasi Sebagai Variabel Kontrol

<table>
<tr><th colspan="2">Hasil Eksperimen</th></tr>
<tr><td colspan="2">Jumlah Iterasi: 1.000</td></tr>
<tr><td colspan="2">Percobaan 1 dengan 5 Populasi</td></tr>
<tr><td rowspan="2">Initial State</td><td>Total Fitness: 7.07</td></tr>
<tr><td>Objective Function Score:
Population Size: 5
Individuals 1 Score: 6514
Individuals 2 Score: 6528
Individuals 3 Score: 7195
Individuals 4 Score: 7473
Individuals 5 Score: 7834</td></tr>
<tr><td rowspan="3">Final State</td><td>Total Fitness: 44.88</td></tr>
<tr><td>Objective Function Score:
Population Size: 5
Individuals 1 Score: 1113
Individuals 2 Score: 1113
Individuals 3 Score: 1113
Individuals 4 Score: 1113
Individuals 5 Score: 1113</td></tr>
<tr><td>Durasi Proses Pencarian: 2.5683471s</td></tr>
<tr><td colspan="2">Plot Nilai Objective Function Terbaik Terhadap Banyak Iterasi</td></tr>
</table>

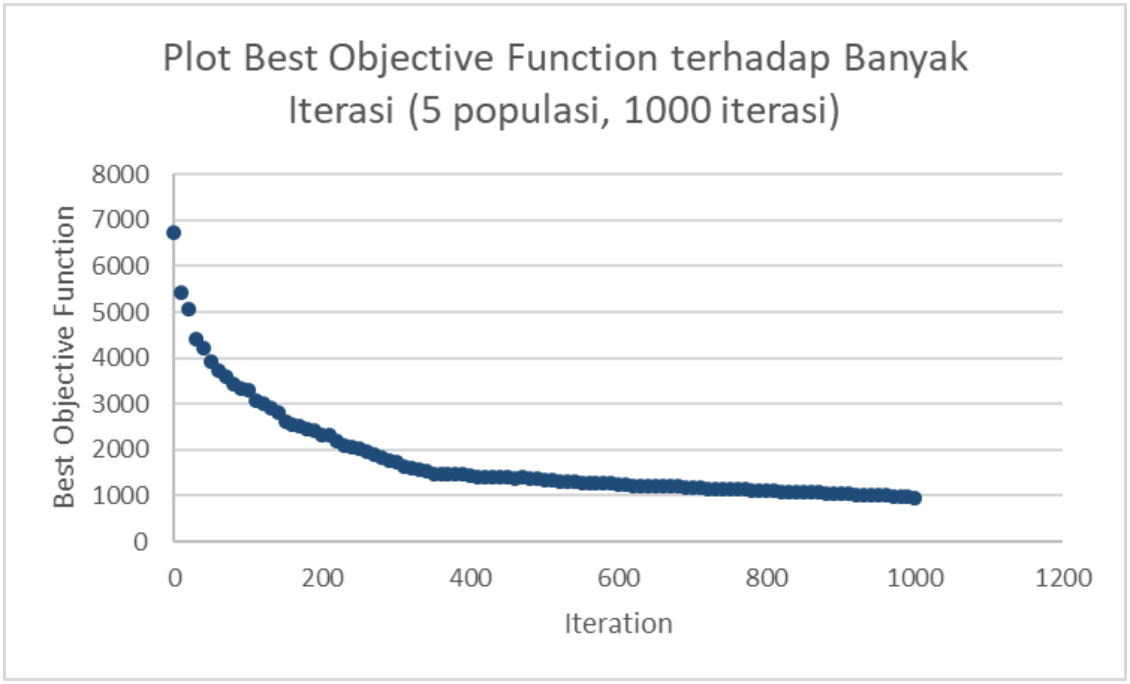


Plot Nilai *Objective Function* Rata-Rata Terhadap Banyak Iterasi

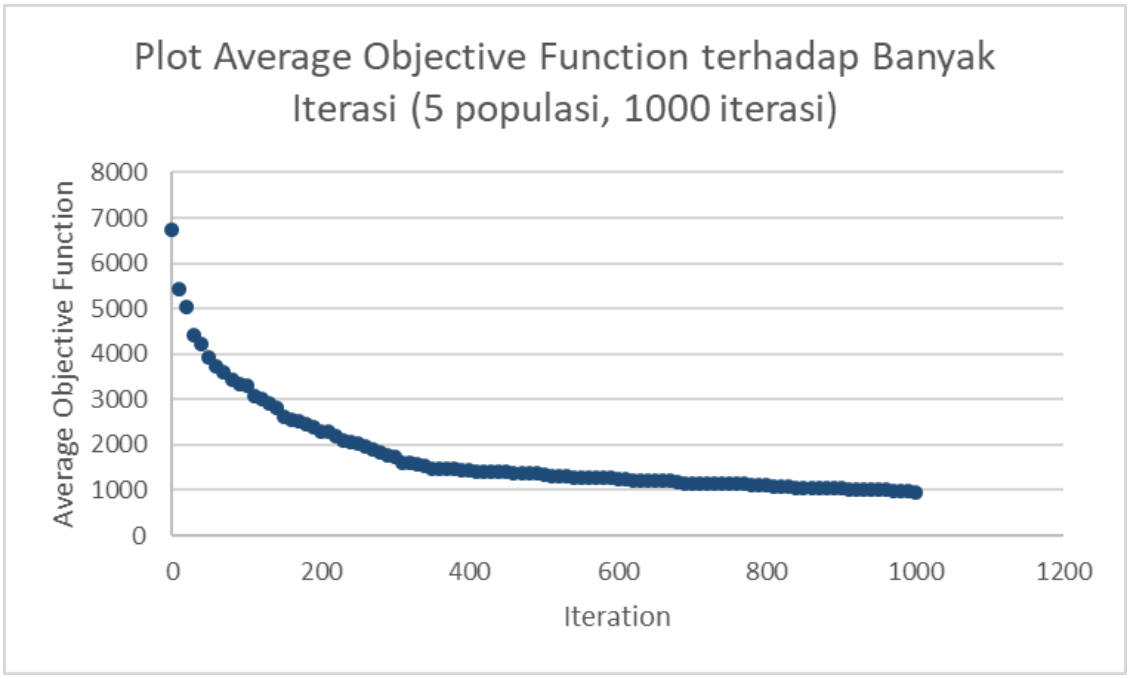


Percobaan 2 dengan 5 Populasi

Initial State	**Total Fitness:** 7.17
	Objective Function Score: Population Size: 5 Individuals 1 Score: 7118 Individuals 2 Score: 6772 Individuals 3 Score: 7344 Individuals 4 Score: 6728 Individuals 5 Score: 6954
Final State	**Total Fitness:** 52.32
	Objective Function Score: Population Size: 5 Individuals 1 Score: 956 Individuals 2 Score: 956 Individuals 3 Score: 956 Individuals 4 Score: 949 Individuals 5 Score: 956
	Durasi Proses Pencarian: 1.2268546s
Plot Nilai Objective Function Terbaik Terhadap Banyak Iterasi	

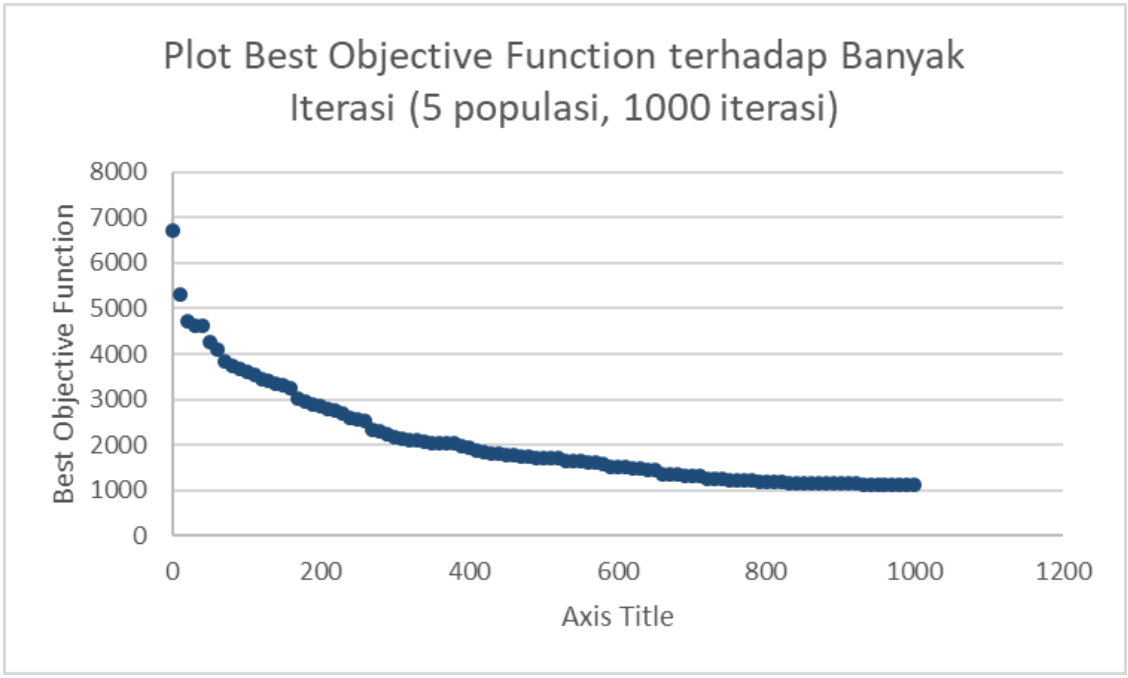


Plot Nilai *Objective Function* Rata-Rata Terhadap Banyak Iterasi

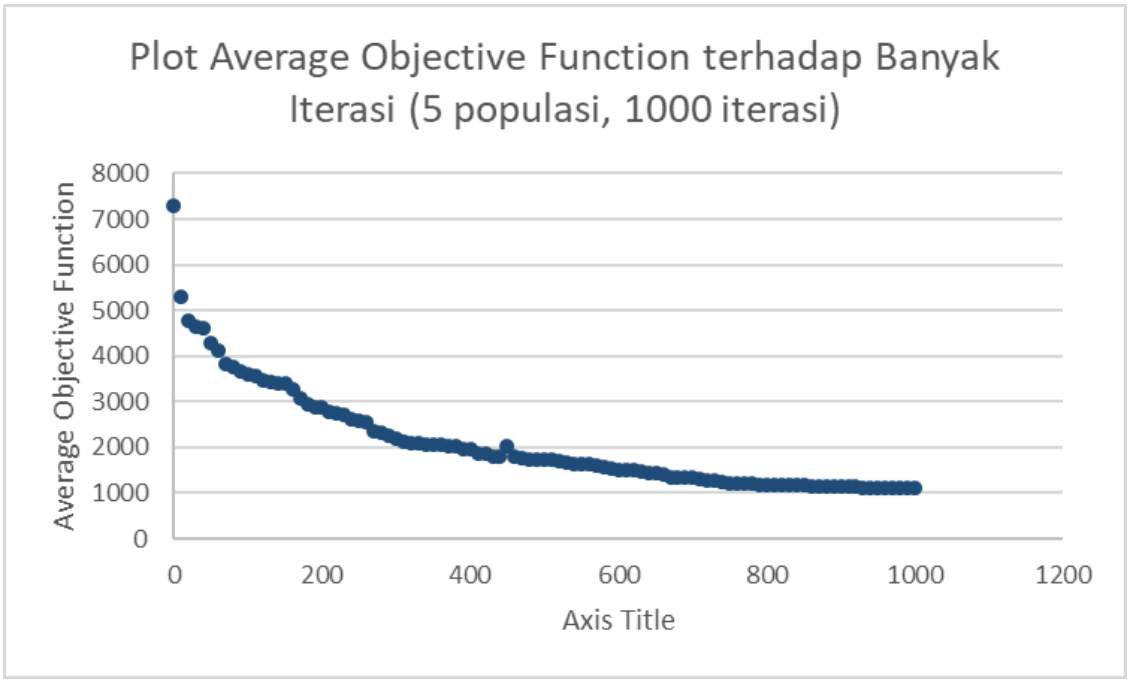


Percobaan 3 dengan 5 Populasi

Initial State	**Total Fitness:** 6.88
	Objective Function Score: Population Size: 5 Individuals 1 Score: 6712 Individuals 2 Score: 6963 Individuals 3 Score: 7871 Individuals 4 Score: 7512 Individuals 5 Score: 7384
Final State	**Total Fitness:** 45.00
	Objective Function Score: Population Size: 5 Individuals 1 Score: 1111 Individuals 2 Score: 1111 Individuals 3 Score: 1109 Individuals 4 Score: 1109 Individuals 5 Score: 1111
	Durasi Proses Pencarian: 3.0804168s
Plot Nilai Objective Function Terbaik Terhadap Banyak Iterasi	



Plot Nilai *Objective Function* Rata-Rata Terhadap Banyak Iterasi

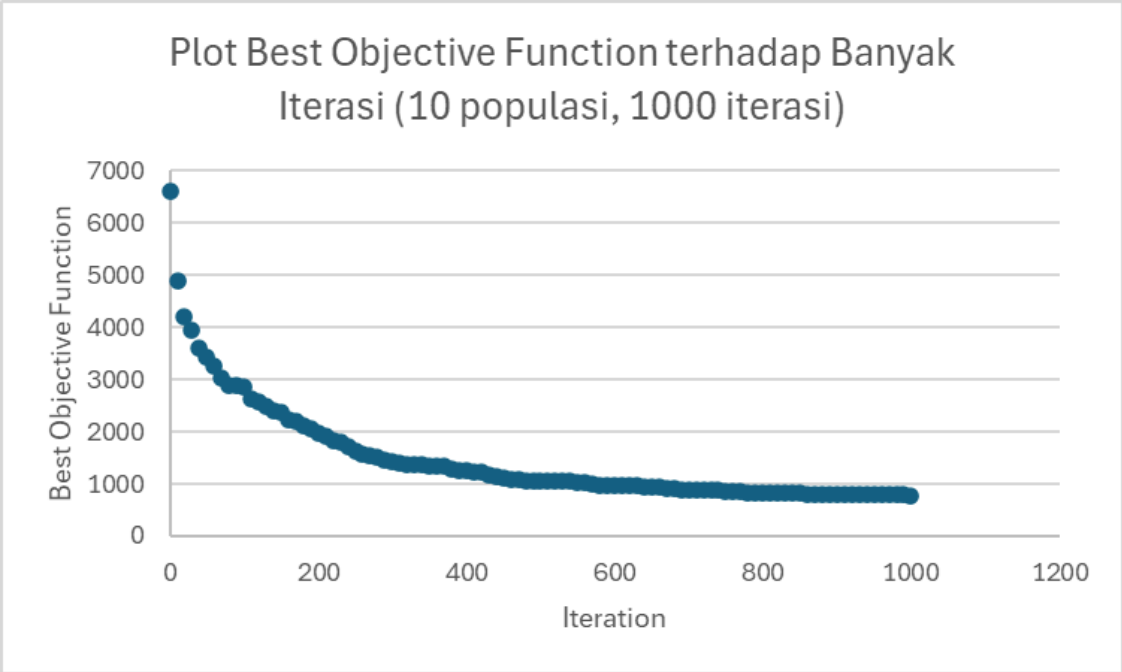


Hasil Eksperimen

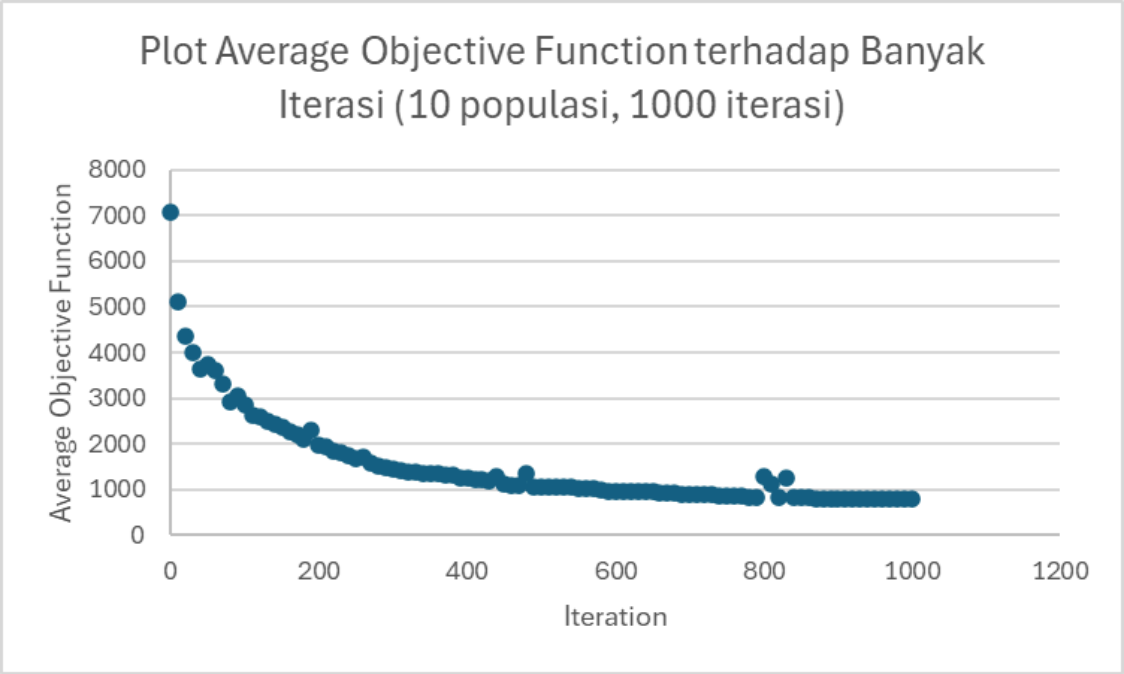
Jumlah Iterasi: 1.000	
Percobaan 1 dengan 10 Populasi	
Initial State	**Total Fitness:** 14.17
	Objective Function Score: Population Size: 10 Individuals 1 Score: 6672 Individuals 2 Score: 6601 Individuals 3 Score: 6951 Individuals 4 Score: 7466 Individuals 5 Score: 7434 Individuals 6 Score: 7046 Individuals 7 Score: 6782 Individuals 8 Score: 6877 Individuals 9 Score: 7630 Individuals 10 Score: 7267
Final State	**Total Fitness:** 127.39
	Objective Function Score: Population Size: 10 Individuals 1 Score: 784 Individuals 2 Score: 784 Individuals 3 Score: 784 Individuals 4 Score: 784 Individuals 5 Score: 784 Individuals 6 Score: 784 Individuals 7 Score: 784 Individuals 8 Score: 784 Individuals 9 Score: 784 Individuals 10 Score: 784

	Durasi Proses Pencarian: `5.1667029s`

Plot Nilai Objective Function Terbaik Terhadap Banyak Iterasi



Plot Nilai *Objective Function* Rata-Rata Terhadap Banyak Iterasi



Percobaan 2 dengan 10 Populasi	
Initial State	**Total Fitness:** `14.69`

	Objective Function Score:
	`Population Size: 10`
	`Individuals 1 Score: 6906`
	`Individuals 2 Score: 7005`
	`Individuals 3 Score: 7257`
	`Individuals 4 Score: 7007`
	`Individuals 5 Score: 6732`
	`Individuals 6 Score: 6586`
	`Individuals 7 Score: 7068`
	`Individuals 8 Score: 6541`
	`Individuals 9 Score: 6850`
	`Individuals 10 Score: 6237`

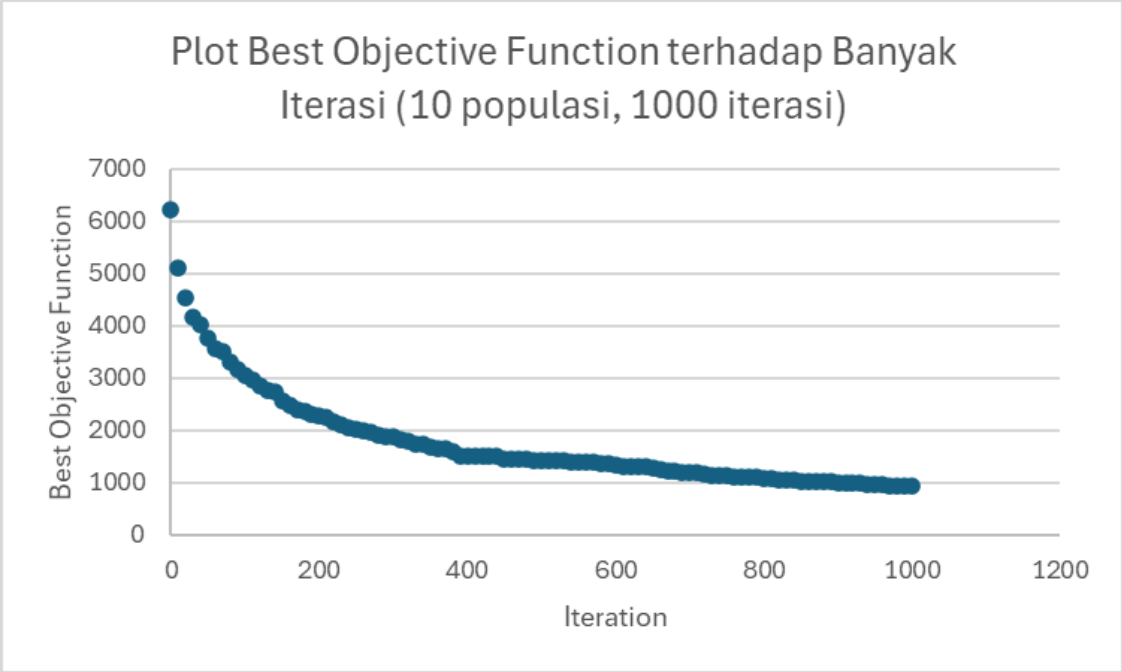
Final State	**Total Fitness:** `106.07`

Objective Function Score:

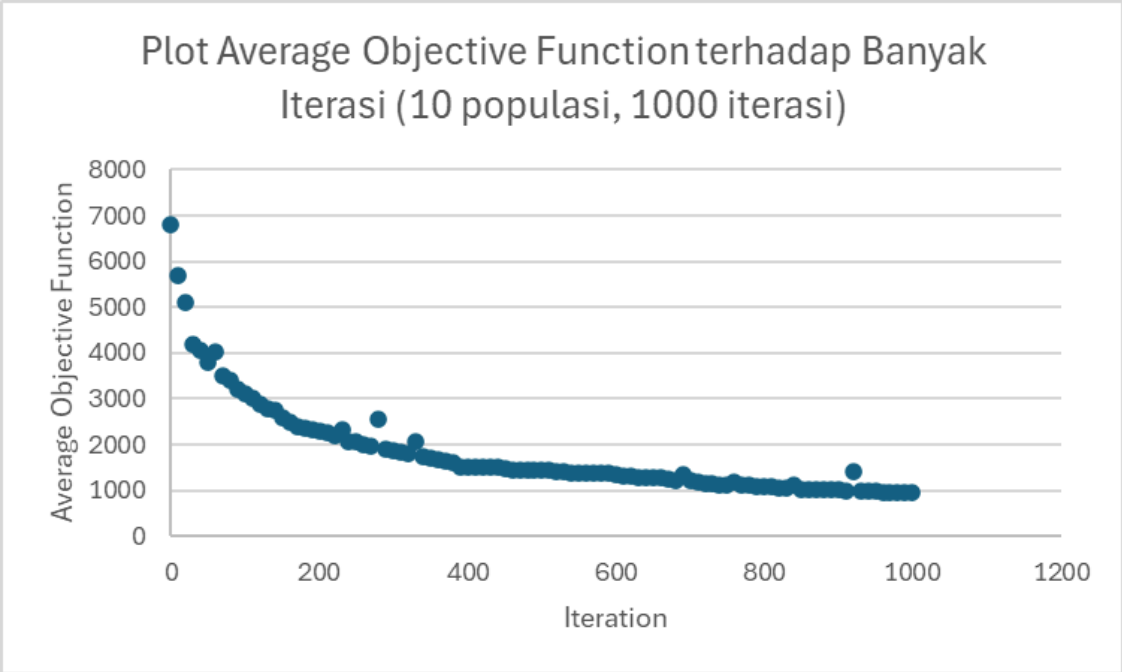
```
Total Fitness: 106.07
Population Size: 10
Individuals 1 Score: 937
Individuals 2 Score: 943
Individuals 3 Score: 943
Individuals 4 Score: 943
Individuals 5 Score: 937
Individuals 6 Score: 943
Individuals 7 Score: 943
Individuals 8 Score: 943
Individuals 9 Score: 943
Individuals 10 Score: 943
```

Durasi Proses Pencarian: `4.5703538s`

Plot Nilai Objective Function Terbaik Terhadap Banyak Iterasi



Plot Nilai *Objective Function* Rata-Rata Terhadap Banyak Iterasi



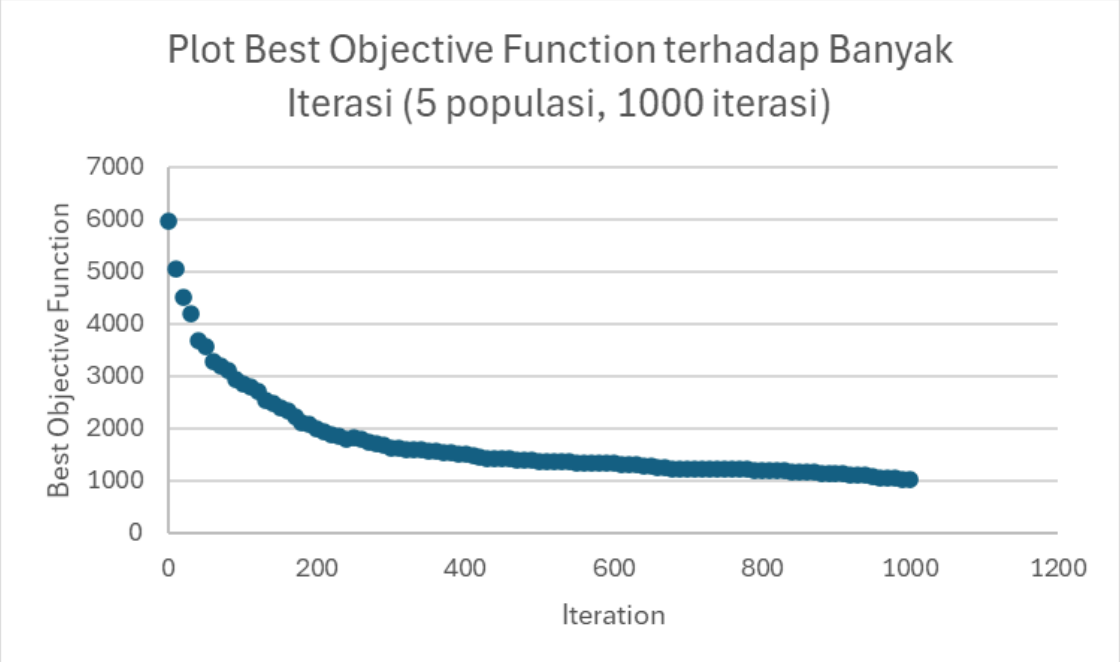
Percobaan 3 dengan 10 Populasi

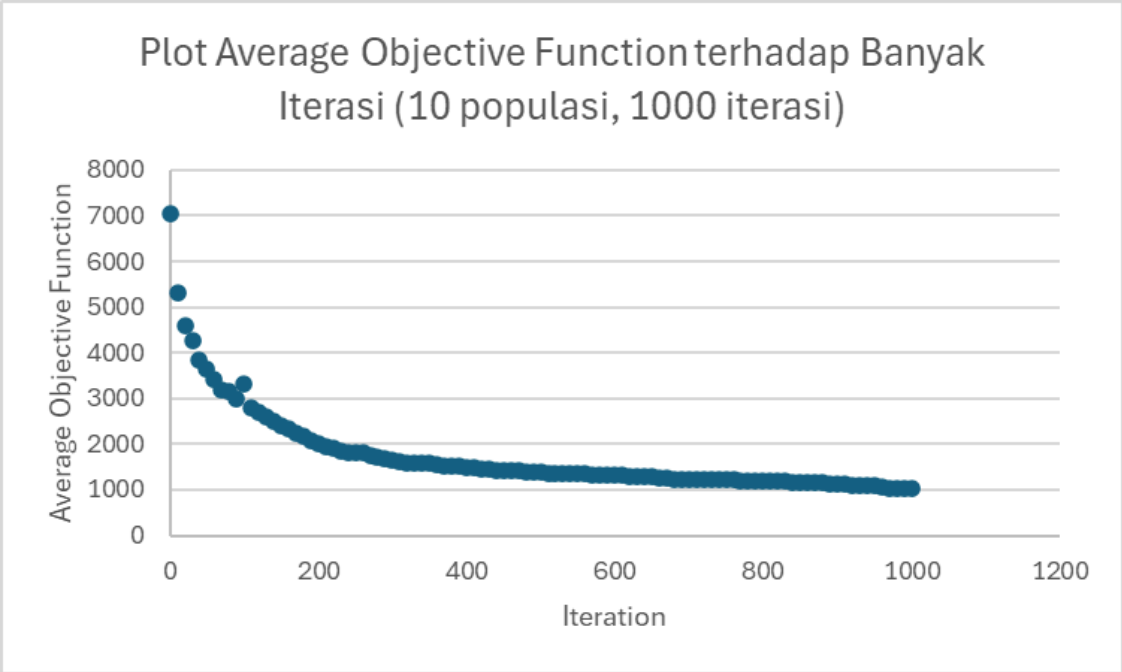
Initial State	**Total Fitness:** `14.29`

	Objective Function Score:
	`Population Size: 10`
	`Individuals 1 Score: 7154`
	`Individuals 2 Score: 6379`
	`Individuals 3 Score: 6447`
	`Individuals 4 Score: 7682`
	`Individuals 5 Score: 7440`
	`Individuals 6 Score: 7616`
	`Individuals 7 Score: 5955`
	`Individuals 8 Score: 7129`
	`Individuals 9 Score: 7389`
	`Individuals 10 Score: 7223`

Final State	**Total Fitness:** `97.75`

	Objective Function Score:
	`Population Size: 10`
	`Individuals 1 Score: 1022`
	`Individuals 2 Score: 1022`
	`Individuals 3 Score: 1022`
	`Individuals 4 Score: 1022`
	`Individuals 5 Score: 1022`
	`Individuals 6 Score: 1022`
	`Individuals 7 Score: 1022`
	`Individuals 8 Score: 1022`
	`Individuals 9 Score: 1022`
	`Individuals 10 Score: 1022`

	Durasi Proses Pencarian: `3.0094707s`
Plot Nilai Objective Function Terbaik Terhadap Banyak Iterasi	
	

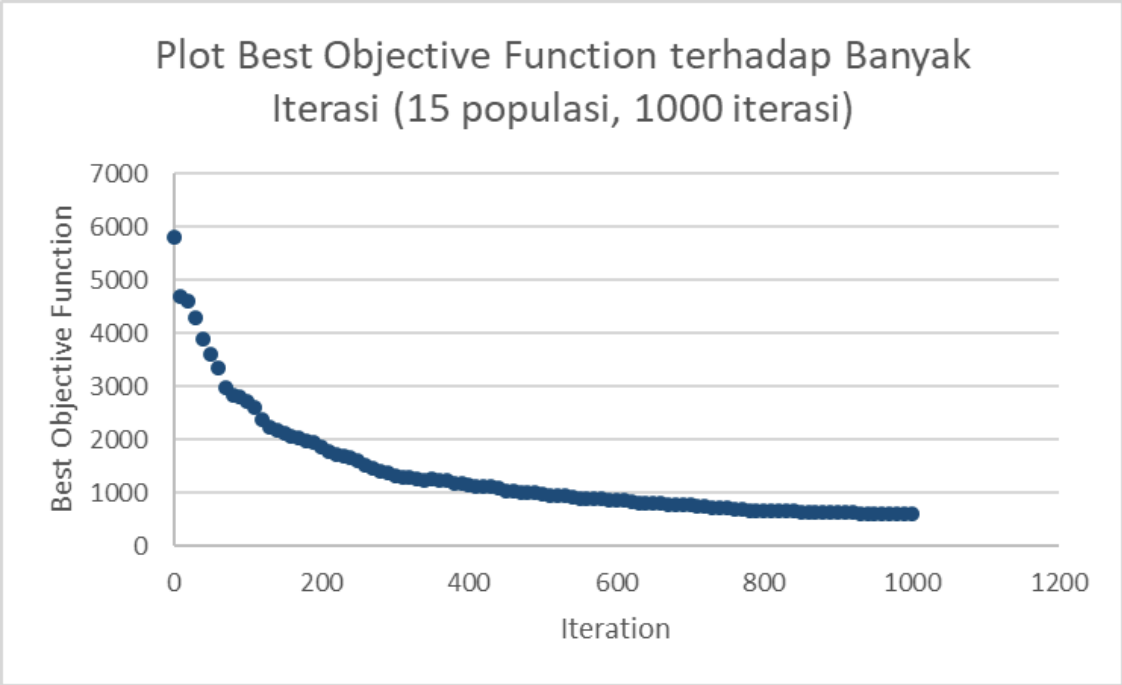
Plot Nilai *Objective Function* Rata-Rata Terhadap Banyak Iterasi


Hasil Eksperimen	
Jumlah Iterasi: 1.000	
Percobaan 1 dengan 15 Populasi	
Initial State	**Total Fitness:** `22.28`

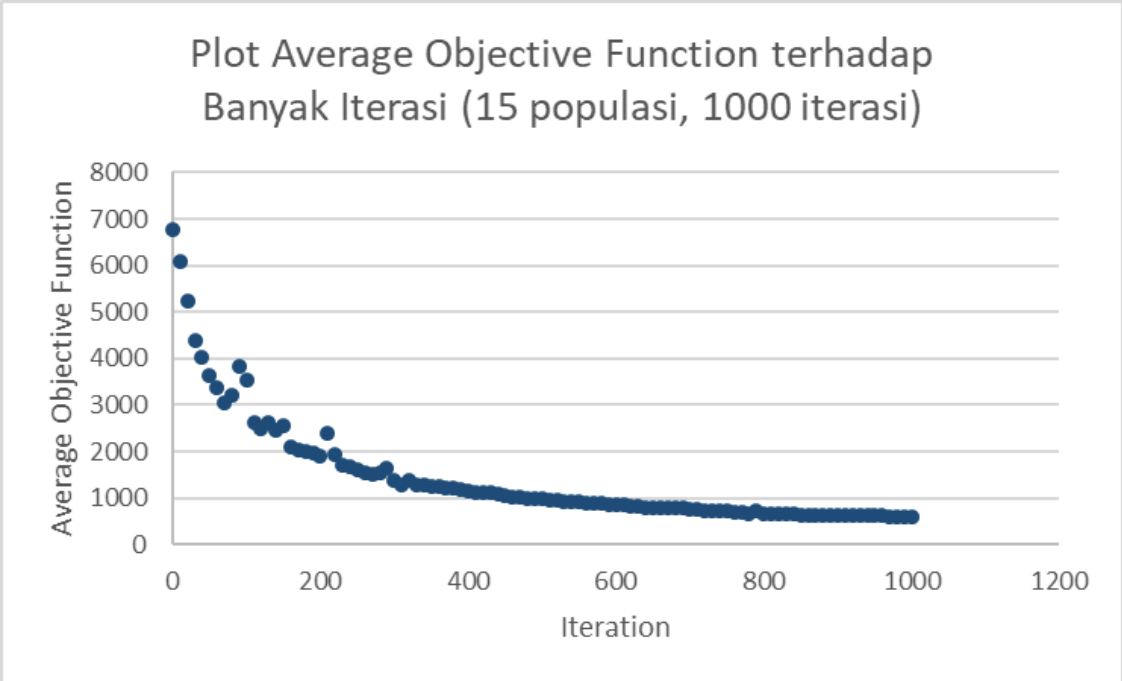
	Objective Function Score:
	`Population Size: 15`
	`Individuals 1 Score: 6521`
	`Individuals 2 Score: 7576`
	`Individuals 3 Score: 6412`
	`Individuals 4 Score: 6790`
	`Individuals 5 Score: 6169`
	`Individuals 6 Score: 6834`

	Individuals 7 Score: 6511 Individuals 8 Score: 7344 Individuals 9 Score: 6251 Individuals 10 Score: 6702 Individuals 11 Score: 7213 Individuals 12 Score: 7742 Individuals 13 Score: 5797 Individuals 14 Score: 7461 Individuals 15 Score: 6346
Final State	**Total Fitness:** 247.52
	Objective Function Score: Population Size: 15 Individuals 1 Score: 605 Individuals 2 Score: 605 Individuals 3 Score: 605 Individuals 4 Score: 605 Individuals 5 Score: 605 Individuals 6 Score: 605 Individuals 7 Score: 605 Individuals 8 Score: 605 Individuals 9 Score: 605 Individuals 10 Score: 605 Individuals 11 Score: 605 Individuals 12 Score: 605 Individuals 13 Score: 605 Individuals 14 Score: 605 Individuals 15 Score: 605
	Durasi Proses Pencarian: 24.8011435s

Plot Nilai Objective Function Terbaik Terhadap Banyak Iterasi



Plot Nilai *Objective Function* Rata-Rata Terhadap Banyak Iterasi

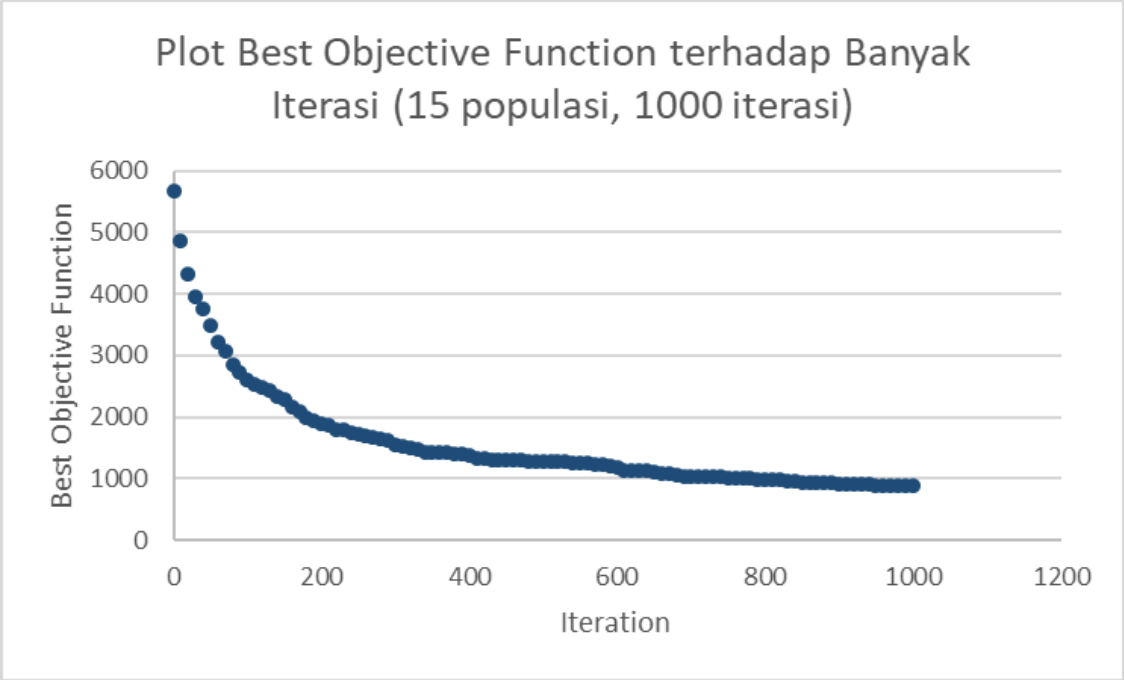


Percobaan 2 dengan 15 Populasi	
Initial State	**Total Fitness:** 22.00
	Objective Function Score: Population Size: 15 Individuals 1 Score: 6366 Individuals 2 Score: 7225 Individuals 3 Score: 7578 Individuals 4 Score: 6843 Individuals 5 Score: 6970 Individuals 6 Score: 6875 Individuals 7 Score: 6320 Individuals 8 Score: 7587 Individuals 9 Score: 6615 Individuals 10 Score: 6938 Individuals 11 Score: 7245 Individuals 12 Score: 6640 Individuals 13 Score: 5683 Individuals 14 Score: 6922 Individuals 15 Score: 6969
Final State	**Total Fitness:** 170.26
	Objective Function Score: Population Size: 15 Individuals 1 Score: 880 Individuals 2 Score: 880 Individuals 3 Score: 880 Individuals 4 Score: 880 Individuals 5 Score: 880 Individuals 6 Score: 880

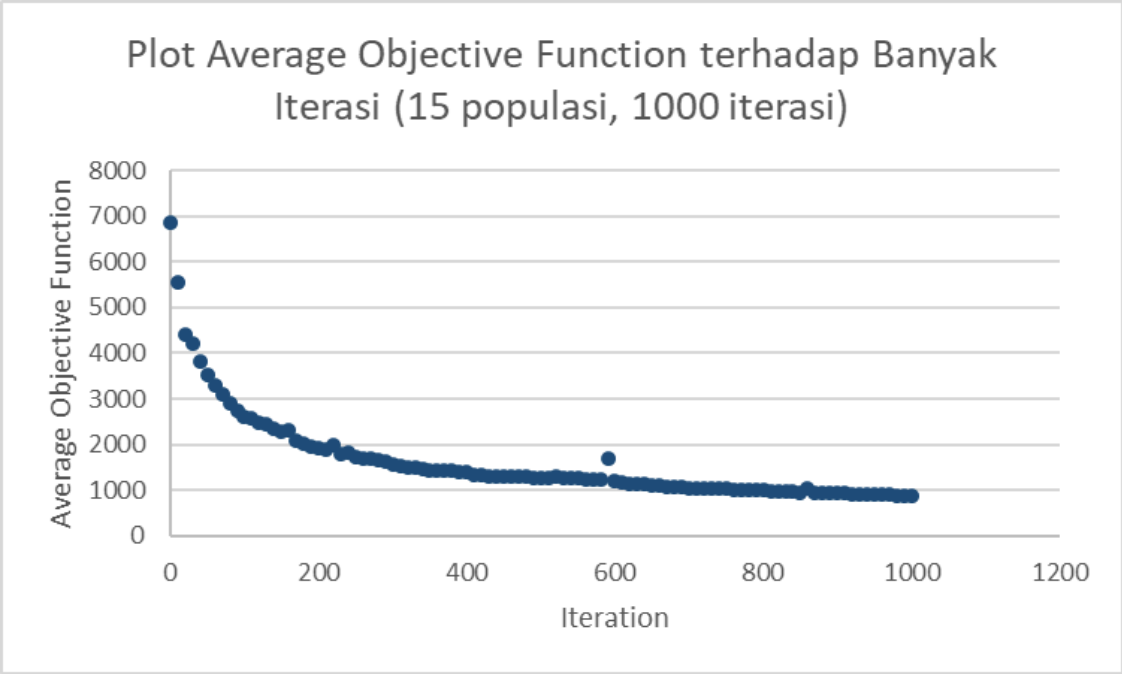
```
Individuals 7 Score: 880
Individuals 8 Score: 880
Individuals 9 Score: 880
Individuals 10 Score: 880
Individuals 11 Score: 880
Individuals 12 Score: 880
Individuals 13 Score: 880
Individuals 14 Score: 880
Individuals 15 Score: 880
```

Durasi Proses Pencarian: `16.128304s`

Plot Nilai Objective Function Terbaik Terhadap Banyak Iterasi

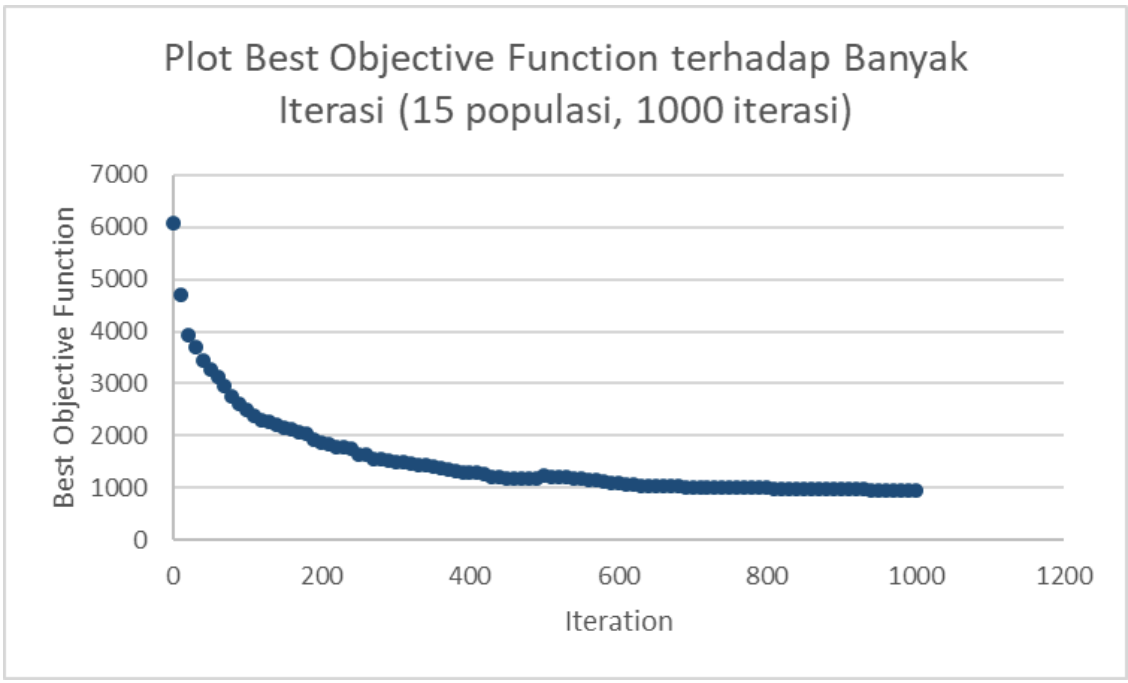


Plot Nilai *Objective Function* Rata-Rata Terhadap Banyak Iterasi

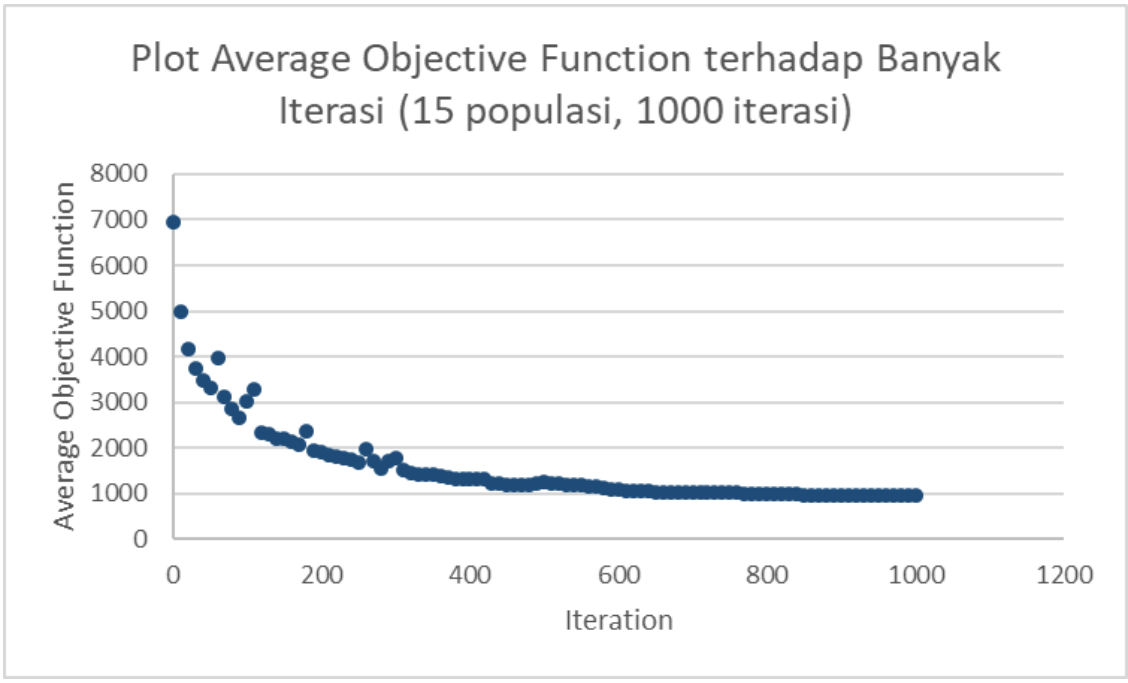


Percobaan 3 dengan 15 Populasi	
Initial State	**Total Fitness:** 21.73
	Objective Function Score: Population Size: 15 Individuals 1 Score: 7383 Individuals 2 Score: 7396 Individuals 3 Score: 6663 Individuals 4 Score: 6460 Individuals 5 Score: 6874 Individuals 6 Score: 6831 Individuals 7 Score: 6073 Individuals 8 Score: 7292 Individuals 9 Score: 7110 Individuals 10 Score: 7673 Individuals 11 Score: 7567 Individuals 12 Score: 6627

	Individuals 13 Score: 6973 Individuals 14 Score: 6259 Individuals 15 Score: 6797
Final State	**Total Fitness:** 157.56
	Objective Function Score: Population Size: 15 Individuals 1 Score: 951 Individuals 2 Score: 951 Individuals 3 Score: 951 Individuals 4 Score: 951 Individuals 5 Score: 951 Individuals 6 Score: 951 Individuals 7 Score: 951 Individuals 8 Score: 951 Individuals 9 Score: 951 Individuals 10 Score: 951 Individuals 11 Score: 951 Individuals 12 Score: 951 Individuals 13 Score: 951 Individuals 14 Score: 951 Individuals 15 Score: 951
	Durasi Proses Pencarian: 13.5754838s
Plot Nilai Objective Function Terbaik Terhadap Banyak Iterasi	



Plot Nilai *Objective Function* Rata-Rata Terhadap Banyak Iterasi



2.5 Analisis

Dalam menyelesaikan permasalahan *Diagonal Magic Cube* berukuran 5x5x5, kami melakukan implementasi terhadap 3 jenis algoritma *local search*, yaitu *Stochastic Hill Climbing, Simulated Annealing,* dan *Genetic Algorithm.*

Pada implementasi algoritma *Stochastic Hill Climbing*, terlihat bahwa dengan peningkatan jumlah iterasi, *objective function* menurun. Hal ini menunjukkan bahwa adanya peningkatan kualitas solusi. Pada percobaan 3 iterasi (10.000 kali, 100.000 kali, dan 1.000.000 kali), menunjukkan lebih banyak iterasi akan menghasilkan solusi dengan kualitas yang baik, namun dengan peningkatan waktu komputasi yang cukup signifikan. Pada 10000 iterasi, solusi yang dihasilkan masih jauh dari global optima, menunjukkan bahwa jumlah iterasi rendah membuat pencarian kurang mendalam, sehingga tidak tercapainya global optima. Pada 100000 iterasi, pencarian cenderung mendekati solusi yang lebih optimal meskipun belum mencapai global optima. Pada 1000000 iterasi, pencarian mendekati solusi optimal. Berdasarkan eksperimen sebelumnya, *Stochastic Hill Climbing* hanya mencapai solusi *local maximum*, karena algoritma hanya mencari solusi yang lebih baik secara langsung dan tidak mempertimbangan solusi yang lebih buruk sementara waktu untuk mencapai eksplorasi lebih lanjut. *Stochastic Hill Climbing* memiliki akhir konsisten tetapi sulit mendekati *global maximum*.

Pada implementasi algoritma *Simulated Annealing*, kami melakukan 3 kali percobaan dengan 3 iterasi (10.000 kali, 100.000 kali, dan 1.000.000 kali). Dari ketiga iterasi, terlihat bahwa semakin banyak iterasi yang dilakukan mempengaruhi kualitas solusi yang dihasilkan (*objective function* semakin berkurang). Hal ini terjadi karena pada algoritma *Simulated Annealing,* di awal iterasi program diperbolehkan untuk memilih *bad move* dengan jumlah yang lebih banyak, pemilihan *bad move* ini terus berkurang seiring bertambahnya iterasi. Pernyataan ini juga dapat dibuktikan dari hasil plot nilai *objective function* terhadap banyak iterasi dan plot nilai $e^{(\frac{-\Delta E}{T})}$ terhadap banyak iterasi yang telah dilakukan. Pada *Simulated Annealing* dengan 10.000 iterasi, penurunan nilai *objective function* yang terjadi seiring bertambahnya iterasi tidak terlalu signifikan, sementara pada *Simulated Annealing* dengan 100.000 dan 1.000.000 iterasi, penurunan nilai *objective function* yang terjadi terlihat sangat signifikan. Pada plot nilai $e^{(\frac{-\Delta E}{T})}$ terhadap banyak iterasi yang dilakukan, saat 10.000 iterasi *scatter plot* yang dihasilkan masih sangat acak yang

menunjukkan masih tingginya kemungkinan untuk melakukan *bad move*. Sementara pada *Simulated Annealing* dengan 100.000 dan 1.000.000 iterasi, *scatter plot* yang dihasilkan semakin turun bahkan mendekati 0. Pada *Simulated Annealing* dengan 10.000 iterasi, *objective function* yang dihasilkan adalah 1374, pada 100.000 iterasi adalah 119, dan pada 1.000.000 iterasi adalah 100. Dengan semakin menurunnya *objective function* ini menunjukkan bahwa *Simulated Annealing* semakin mendekati *global optima* seiring dengan bertambahnya iterasi. Konsistensi hasil akhir pada *Simulated Annealing* meningkat seiring bertambahnya jumlah iterasi. Pada 10.000 iterasi, hasil yang didapatkan tidak konsisten dan cenderung jauh dari optimal. Pada 100.000 iterasi, hasil menjadi lebih konsisten dan mendekati optimal. Sedangkan pada 1.000.000 iterasi, hasilnya hampir mencapai *global optimal* dengan konsistensi yang tinggi, sehingga pengulangan percobaan pada iterasi ini kemungkinan akan menghasilkan *objective function* yang sangat mirip atau sama.

Pada percobaan *genetic algorithm*, kami melakukan 18 kali percobaan dengan mengelompokkannya menjadi dua bagian, yaitu melakukan eksperimen dengan populasi sebagai variabel kontrol dan eksperimen dengan iterasi sebagai variabel kontrol.

Eksperimen pertama, adalah melakukan percobaan dengan parameter populasi konstan sebanyak 5 individu dan iterasi sebanyak 1.000 kali. Pada state awal, skor individu bervariasi, yaitu antara 622 hingga 7349. Namun, setelah algoritma dijalankan, semua individu memiliki skor yang seragam, yaitu 983. Hal ini menunjukkan bahwa konvergensi populasi ke solusi optimal atau mendekati optimal. Proses pencarian algoritma untuk menyelesaikan 1.000 iterasi dengan jumlah populasi 5 dilakukan selama 1,446s. Proses pencarian ini dilakukan sebanyak 3 kali, dan di dapatkan perbandingan total fitness sebagai berikut :

Total Fitness	Percobaan 1	Percobaan 2	Percobaan 3
Initial State	7.09	7.35	6.99
Final State	50.81	44.48	45.05

Dari tabel diatas, dapat dilihat bahwa hasil dari total fitness (kualitas populasi) konsisten dan menunjukkan bahwa algoritma berhasil meningkatkan kualitas solusi selama proses iterasi.

Eksperimen kedua, adalah melakukan percobaan dengan parameter populasi konstan sebanyak 5 individu dan iterasi sebanyak 10.000 kali. Pada state awal, skor individu bervariasi, yaitu antara 6775 hingga 7728. Namun, setelah algoritma dijalankan, semua individu memiliki skor yang seragam, yaitu 1012. Hal ini menunjukkan bahwa konvergensi populasi ke solusi optimal atau mendekati optimal. Proses pencarian algoritma untuk menyelesaikan 10.000 iterasi dengan jumlah populasi 5 dilakukan selama 1,683s. Proses pencarian ini dilakukan sebanyak 3 kali, dan di dapatkan perbandingan total fitness sebagai berikut :

Total Fitness	Percobaan 1	Percobaan 2	Percobaan 3
Initial State	6.95	6.96	7.24
Final State	49.36	69.35	101.63

Dari tabel diatas, dapat dilihat bahwa penambahan jumlah iterasi menyebabkan ketidakkonsistenan. Hal ini dapat dilihat dari perbedaan nilai *total fitness*-nya yang cukup besar. Namun, algoritma tetap berhasil meningkatkan kualitas solusi selama proses iterasi.

Eksperimen ketiga, adalah melakukan percobaan dengan parameter populasi konstan sebanyak 5 individu dan iterasi sebanyak 100.000 kali. Pada state awal, skor individu bervariasi, yaitu antara 6397 hingga 7770. Namun, setelah algoritma dijalankan, semua individu memiliki skor yang seragam, yaitu 481. Hal ini menunjukkan bahwa konvergensi populasi ke solusi optimal atau mendekati optimal. Proses pencarian algoritma untuk menyelesaikan 100.000 iterasi dengan jumlah populasi 5 dilakukan selama 18,781s. Proses pencarian ini dilakukan sebanyak 3 kali, dan di dapatkan perbandingan total fitness sebagai berikut :

Total Fitness	Percobaan 1	Percobaan 2	Percobaan 3
Initial State	7.07	6.88	7.27
Final State	103.73	73.75	73.31

Dari tabel diatas, dapat dilihat bahwa penambahan jumlah iterasi menyebabkan ketidakkonsistenan. Hal ini dapat dilihat dari perbedaan nilai *total fitness*-nya yang cukup besar. Namun, algoritma tetap berhasil meningkatkan kualitas solusi selama proses iterasi.

Setelah melakukan tiga kali eksperimen dengan parameter populasi konstan sebanyak 5 individu, dapat dilihat bahwa banyaknya iterasi mempengaruhi hasil akhir dari pencarian Genetic Algorithm. Banyaknya iterasi memungkinkan algoritma untuk melakukan lebih banyak pencarian dan proses seleksi, mutasi, dan *crossover*. Hal ini akan memberikan peluang bagi algoritma untuk memperbaiki nilai objective functionnya. Peningkanam fitness dari state awal ke state akhir menunjukkan bahwa lebih banyak iterasi memberikan peluang yang lebih baik untuk mendekati solusi.

Eksperimen keempat, adalah melakukan percobaan dengan parameter iterasi konstan sebanyak 1.000 kali iterasi dan populasi sebanyak 5 individu. Pada state awal, skor individu bervariasi, yaitu antara 6514 hingga 7834. Namun, setelah algoritma dijalankan, semua individu memiliki skor yang seragam, yaitu 1113. Hal ini menunjukkan bahwa konvergensi populasi ke solusi optimal atau mendekati optimal. Proses pencarian algoritma untuk menyelesaikan 1.000 iterasi dengan jumlah populasi 5 dilakukan selama 2,568s. Proses pencarian ini dilakukan sebanyak 3 kali, dan di dapatkan perbandingan total fitness sebagai berikut :

Total Fitness	Percobaan 1	Percobaan 2	Percobaan 3
Initial State	7.07	7.17	6.88
Final State	44.88	52.32	45.00

Dari tabel diatas, dapat dilihat bahwa hasil dari total fitness (kualitas populasi) konsisten dan menunjukkan bahwa algoritma berhasil meningkatkan kualitas solusi selama proses iterasi.

Eksperimen kelima, adalah melakukan percobaan dengan parameter iterasi konstan sebanyak 1.000 kali iterasi dan populasi sebanyak 10 individu. Pada state awal, skor individu bervariasi, yaitu antara 6601 hingga 7630. Namun, setelah algoritma dijalankan, semua individu memiliki

skor yang seragam, yaitu 1113. Hal ini menunjukkan bahwa konvergensi populasi ke solusi optimal atau mendekati optimal. Proses pencarian algoritma untuk menyelesaikan 1.000 iterasi dengan jumlah populasi 10 dilakukan selama 5.166s. Proses pencarian ini dilakukan sebanyak 3 kali, dan di dapatkan perbandingan total fitness sebagai berikut :

Total Fitness	Percobaan 1	Percobaan 2	Percobaan 3
Initial State	14.17	14.69	14.29
Final State	127.39	106.07	97.75

Dari tabel diatas, dapat dilihat bahwa penambahan jumlah populasi menyebabkan ketidakkonsistenan. Hal ini dapat dilihat dari perbedaan nilai *total fitness*-nya yang cukup besar. Namun, algoritma tetap berhasil meningkatkan kualitas solusi selama proses iterasi.

Eksperimen keenam, adalah melakukan percobaan dengan parameter iterasi konstan sebanyak 1.000 kali iterasi dan populasi sebanyak 15 individu. Pada state awal, skor individu bervariasi, yaitu antara 5797 hingga 7742. Namun, setelah algoritma dijalankan, semua individu memiliki skor yang seragam, yaitu 605. Hal ini menunjukkan bahwa konvergensi populasi ke solusi optimal atau mendekati optimal. Proses pencarian algoritma untuk menyelesaikan 1.000 iterasi dengan jumlah populasi 15 dilakukan selama 24.801s. Proses pencarian ini dilakukan sebanyak 3 kali, dan di dapatkan perbandingan total fitness sebagai berikut :

Total Fitness	Percobaan 1	Percobaan 2	Percobaan 3
Initial State	22.28	22.00	21.73
Final State	247.52	170.26	157.56

Dari tabel diatas, dapat dilihat bahwa penambahan jumlah populasi menyebabkan ketidakkonsistenan. Hal ini dapat dilihat dari perbedaan nilai *total fitness*-nya yang cukup besar. Namun, algoritma tetap berhasil meningkatkan kualitas solusi selama proses iterasi.

Jumlah populasi yang besar mempengaruhi hasil akhir pencarian. Dengan populasi yang lebih besar, algoritma memiliki lebih banyak variasi genetik dan memungkinkan untuk memperbesar ruang lingkup eksplorasi solusi dan menghindari algoritma menemukan solusi optimal yang bukan merupakan solusi terbaik secara global.

Berikut merupakan tabel durasi pencarian solusi oleh algoritma :

Durasi	Durasi 5 Pop	Durasi 10 Pop	Durasi 15 Pop
Percobaan 1	2.5683471s	5.1667029s	24.8011435s
Percobaan 2	1.2268546s	4.5703538s	16.128304s
Percobaan 3	3.0804168s	3.0094707s	13.5754838s

Penambahan populasi juga mempengaruhi durasi pencarian. Hal ini terlihat dari durasi pencarian untuk populasi sebanyak 5 individu sekitar 1 hingga 3 detik, sedangkan untuk populasi 15 individu, diperlukan durasi sekitar 13 hingga 24 detik. Semakin banyak populasi, maka semakin besar juga durasi yang dibutuhkan.

Berdasarkan hasil eksperimen yang dilakukan, genetic algorithm dapat mendekati solusi global optimum dengan proses eksplorasi ruang solusi dengan mengandalkan crossover dan mutasi untuk menciptakan solusi. Ukuran populasi membantu meningkatkan keragaman genetik sehingga memperbesar ruang pencarian solusi. *Crossover* yang dilakukan berfungsi untuk menggabungkan solusi potensial untuk menemukan solusi yang lebih baik. Semakin efektif *crossover* yang dilakukan, solusi semakin mendekati optimum.

Berdasarkan hasil percobaan, durasi proses pencarian dari ketiga algoritma S*tochastic, Simulated Annealing,* dan *Genetic Algorithm* menunjukkan perbedaan signifikan terkait efisiensi waktu, yang dipengaruhi oleh jumlah iterasi serta variabel kontrol populasi dalam *Genetic Algorithm*. Algoritma *Stochastic* memiliki durasi tercepat pada setiap percobaan, sedangkan *Simulated Annealing* memiliki durasi yang sedikit lebih tinggi dibanding *Stochastic* untuk jumlah iterasi yang sama. Hal ini terjadi karena *Simulated Annealing* menambah kompleksitas perhitungan

dalam proses pencarian. Semesntara itu, *genetic algorithm* memiliki durasi proses yang lebih lama secara signifikan dibanding kedua algoritma lainnya. Dalam percobaan dengan kenaikan variabel kontrol populasi mengakibatkan durasi pencarian bertambah. Hal ini disebabkan oleh langkah-langkah seleksi, mutasi, dan crossover dalam *Genetic Algorithm* yang membutuhkan lebih banyak waktu komputasi, terutama seiring dengan bertambahnya ukuran populasi.

Berdasarkan percobaan yang telah dilakukan, algoritma *Simulated Annealing* memberikan hasil *objective function* yang paling mendekati 0. Pada iterasi-iterasi awal *Simulated Annealing*, *objective function* yang dihasilkan bervariasi. Hal ini terjadi karena algoritma ini memberikan kesempatan untuk melakukan *bad move*. *Stochastic Hill Climbing* memiliki nilai *objective function* yang lebih baik daripada *Genetic Algorithm*, namun masih lebih buruk jika dibandingkan dengan *Simulated Annealing*. *Stochastic Hill Climbing* membangkitkan *neighbor* secara acak, namun masih mungkin untuk terjebak pada *state* yang tidak memiliki *successor* lebih baik lagi. *Genetic Algorithm* merupakan algoritma dengan *objective function* yang paling buruk dari ketiga algoritma tersebut. Hal ini terjadi karena pada *Genetic Algorithm,* kecenderungan anak tidak mewarisi *fitness function* dari orang tuanya.

BAB III
KESIMPULAN DAN SARAN

Berdasarkan eksperimen penyelesaian persoalan *diagonal magic cube* berukuran 5 × 5 × 5 dengan berbagai algoritma *local search*, algoritma *local search* cenderung sulit mencapai global optima pada masalah dengan banyak puncak lokal, seperti pada persoalan *diagonal magic cube*. Hal ini mengakibatkan algoritma lebih mudah terjebak dalam solusi suboptimal. Dalam hal efektivitas pencarian, algoritma *simulated annealing* memberikan hasil pencarian terbaik, karena mendekati solusi optimal. Di sisi lain, dalam aspek efisiensi waktu, *stochastic hill-climbing* unggul dengan waktu pencarian yang lebih cepat, sedangkan *genetic algorithm* membutuhkan waktu yang paling lama untuk mencapai solusi.

BAB IV
PEMBAGIAN TUGAS

NIM	Nama	Tugas
18222009	Daffa Ramadhan Elengi	- Mengerjakan bagian genetic algorithm - Membuat struktur program - Membuat visualisasi dari kubus - Finalisasi dokumen
18222045	Givari Al Fachri	- Mengerjakan bagian *stochastic hill climbing* - Finalisasi dokumen
18222074	Kayla Dyara	- Mengerjakan bagian *simulated annealing* - Finalisasi dokumen
18222078	Monica Angela Hartono	- Mengerjakan bagian *simulated annealing* - Finalisasi dokumen

REFERENSI

Baeldung. (n.d.). *Partially Mapped Crossover (PMX)*. Retrieved November 10, 2024, from https://www.baeldung.com/cs/ga-pmx-operator

Russell, S., & Norvig, P. (2020). Artificial Intelligence: A Modern Approach (4th ed.). Pearson.