Laporan Tugas Besar I

IF3170 Dasar Intelegensi Artifisial

Disusun oleh :

Jacob Reinhard M. Siagian - 18223026
Stevan Einer Bonagabe - 18223028
Hans Joseph B. W. Silitonga - 18223072

SISTEM DAN TEKNOLOGI INFORMASI
SEKOLAH TEKNIK ELEKTRO DAN INFORMATIKA
INSTITUT TEKNOLOGI BANDUNG

Daftar Isi

BAB I
Deskripsi Masalah

1.1 Bin Packing Problem

Permasalahan yang diselesaikan dalam tugas ini adalah *Bin Packing Problem* (BPP) klasik, sebuah masalah optimasi yang fundamental. Inti dari masalah ini adalah bagaimana cara menempatkan sekumpulan barang, yang masing-masing memiliki ukuran berbeda, ke dalam sejumlah kontainer. Setiap kontainer yang tersedia memiliki kapasitas yang seragam dan identik. Tujuan utamanya adalah menemukan konfigurasi pengepakan yang paling efisien, yaitu yang menggunakan jumlah kontainer sesedikit mungkin.

Setiap barang diidentifikasi melalui ID Barang yang unik dan memiliki atribut ukuran. Di sisi lain, setiap kontainer hanya didefinisikan oleh satu atribut, yaitu Kapasitas maksimumnya. Untuk merepresentasikan state dalam masalah ini,kami mengguakan struktur data berupa list of lists. Dalam representasi ini, setiap list internal melambangkan satu buah kontainer, dan elemen di dalam list tersebut adalah ID dari barang-barang yang ditempatkan di dalamnya.

Pencarian solusi dilakukan dengan menjelajahi state space menggunakan algoritma local search. Proses ini bergantung pada dua operator 'move' yang diizinkan untuk bertransisi dari satu state ke state tetangga:

1. Memindahkan satu barang dari kontainernya saat ini ke kontainer lain
2. Menukar dua barang yang berada di dua kontainer yang berbeda.

Kualitas dari setiap 'state' dievaluasi menggunakan sebuah objective function. Fungsi ini melakukan pencarian menuju solusi paling optimal dengan dua parameter, yaitu memberikan penalti yang besar untuk setiap solusi yang tidak valid, yaitu ketika total ukuran barang dalam sebuah kontainer melebihi kapasitasnya dan setelah validitas dipastikan, fungsi ini memprioritaskan solusi yang menggunakan lebih sedikit kontainer dan memiliki kepadatan yang lebih tinggi.

BAB II
Pembahasan

2.1 Pemilihan Objective Function

Pada tugas ini, algoritma *local search* yang digunakan bertujuan untuk **meminimalkan** nilai dari *objective function*. Nilai yang semakin kecil menunjukkan solusi yang semakin baik (biaya lebih rendah). Nilai *objective function* ini dihitung berdasarkan tiga komponen utama, yang masing-masing diberi bobot untuk merepresentasikan prioritasnya:

1. **Penalti Kapasitas Berlebih (Overload Penalty)**
2. **Skor Jumlah Kontainer**
3. **Skor Kepadatan (Total Ruang Kosong)**

Pada tugas ini, algoritma *local search* yang digunakan bertujuan untuk **meminimalkan** nilai dari *objective function*. Nilai yang semakin kecil menunjukkan solusi yang semakin baik (biaya lebih rendah). Nilai *objective function* ini dihitung berdasarkan tiga komponen utama, yang masing-masing diberi bobot untuk merepresentasikan prioritasnya:

Tujuan algoritma yang digunakan adalah untuk meminimalkan nilai fungsi objektif ini, nilai yang semakin kecil menunjukkan solusi yang semakin baik. Nilai 0 tidak mungkin dicapai dalam kasus ini karena minimal selalu ada kontainer yang digunakan, namun semakin kecil nilai objective function menunjukkan packing yang semakin efisien.

1. Penalti Kapasitas Berlebih (Overload Penalty)

   Komponen ini adalah prioritas utama untuk memastikan solusi yang dihasilkan valid. Komponen ini memberikan penalti yang sangat besar untuk setiap unit ukuran yang melebihi kapasitas.Bobot untuk penalti ini:

   (W_PENALTI_OVERLOAD) = 10

   Nilai ini secara signifikan paling tinggi daripada komponen lainnya. Tujuannya adalah agar algoritma pencarian selalu memprioritaskan perbaikan solusi yang overload di atas optimasi lainnya. Skor penalti untuk satu kontainer dihitung dengan rumus:

   (TotalUkuran - Kapasitas) x 10

   Misalnya, jika sebuah kontainer berkapasitas 100 diisi dengan barang berukuran total 105, ia akan mendapat penalti sebesar

   (105 - 100) x 10 = 50 poin.

2. Skor Jumlah Kontainer

   Komponen ini secara langsung menilai tujuan utama dari Bin Packing Problem, yaitu meminimalkan jumlah kontainer yang digunakan. Ini adalah prioritas kedua setelah validitas solusi. Setiap kontainer yang digunakan dalam state akan menambah biaya pada skor total.Bobot untuk komponen ini:

   (W_JUMLAH_KONTAINER) = 2

   Ini berarti bahwa menemukan cara untuk menghemat satu kontainer (mengurangi skor sebanyak 2 poin) dianggap setara dengan mengurangi 2 unit ruang kosong. Rumus perhitungannya:

   (Total Kontainer yang Digunakan) x 2

3. Skor Kepadatan (Total Ruang Kosong)

   Komponen ini bertindak sebagai pembeda untuk solusi-solusi yang sudah valid dan memiliki jumlah kontainer yang sama. Di antara dua solusi dengan 3 kontainer, solusi yang memiliki total ruang kosong lebih sedikit (lebih padat) akan dianggap lebih baik. Parameter ini menghitung bobot sisa ruang yang tersedia pada kontainer. Bobot untuk komponen ini:

   (W_KEPADATAN) = 1.

   Ini menjadikannya prioritas terendah. Setiap unit ruang yang tidak terpakai di dalam setiap kontainer yang valid akan dijumlahkan dan menambah skor total.

   Σ((Kapasitas - Total Ukuran)) x 1

   Dengan menggabungkan ketiga komponen ini, objective function secara efektif memandu algoritma untuk terlebih dahulu mencari solusi yang valid (tidak overload), kemudian mencari solusi yang menggunakan kontainer paling sedikit, dan akhirnya, mencari solusi yang paling padat.

2.2 Implementation

Pada bagian ini, akan dijelaskan implementasi dari setiap kelas dan fungsi utama yang digunakan untuk menyelesaikan *Bin Packing Problem*. Implementasi ini didasarkan pada spesifikasi masalah yang telah diuraikan pada Bab I.

2.2.1 Solusi Packing (Container.py)

Kelas SolusiPacking adalah semacam “blueprint” untuk setiap solusi dalam state space. Kelas ini menyimpan representasi state, menghitung nilai objective function nya dan juga menyediakan metode untuk memodifikasi state tersebut.

```
W_JUMLAH_KONTAINER = 2
W_PENALTI_OVERLOAD = 10
W_KEPADATAN = 1
```

Variabel class ini mendefinisikan bobot (weights) untuk tiga komponen objective function.

```
def __init__(self, kapasitas_kontainer, daftar_barang):
        self.kapasitas = kapasitas_kontainer
        self.barang = daftar_barang
        self.ukuran_barang = {}
        for item in self.barang:
            self.ukuran_barang[item['id']] = item['ukuran']
        self.state = []
```

Fungsi constructor ini dieksekusi setiap kali sebuah objek SolusiPacking dibuat. Fungsi ini menerima kapasitas maksimum kontainer dan daftar barang sebagai input. Kapasitas disimpan dalam self.kapasitas.

```
def hitung_total_ukuran(self, kontainer):
        total = 0
        for item_id in kontainer:
            total += self.ukuran_barang[item_id]
        return total
```

Fungsi helper ini bertugas untuk menghitung total ukuran barang yang ada di dalam satu kontainer. Fungsi ini menerima satu kontainer (berupa *list* berisi ID barang) sebagai input, kemudian melakukan iterasi dan menjumlahkan ukuran setiap barang menggunakan dictionary self.ukuran_barang.

```
def objective_function(self):
        # hitung skor total
        skor = 0
        ruang_kosong_total = 0

        # jumlah kontainer
        skor = len(self.state) * self.W_JUMLAH_KONTAINER

        # cek setiap kontainer
        for kontainer in self.state:
            ukuran = self.hitung_total_ukuran(kontainer)
```

```
            if ukuran > self.kapasitas:
                # kena penalty kalo overload
                overload = ukuran - self.kapasitas
                skor += overload * self.W_PENALTI_OVERLOAD
            else:
                # hitung ruang kosong
                ruang_kosong = self.kapasitas - ukuran
                ruang_kosong_total += ruang_kosong

        skor += ruang_kosong_total * self.W_KEPADATAN
        return skor
```

Fungsi ini merupakan implementasi dari objective function yang telah dirancang sebelumnya. Tujuannya adalah menghitung skor total dari self.state saat ini, di mana skor (nilai objective function) yang semakin kecil menunjukkan solusi yang semakin baik. Perhitungan skor dibagi menjadi tiga bagian seperti yang sudah dijelaskan pada bagian pemilihan objective function

```
def inisialisasi_random(self):
        # bikin state awal random (agak jelek biar bisa di-improve)
        self.state = []
        barang_list = self.barang.copy()
        random.shuffle(barang_list)

        for brg in barang_list:
            item_id = brg['id']
            ukuran = brg['ukuran']

            if not self.state:
                # kontainer pertama
                self.state.append([item_id])
            else:
                # random placement - sengaja ga optimal
                if random.random() < 0.5:
                    # masukin ke kontainer random (bisa overload)
                    idx = random.randint(0, len(self.state) - 1)
                    self.state[idx].append(item_id)
                elif random.random() < 0.4:
                    # bikin kontainer baru (boros)
```

```
                    self.state.append([item_id])
                else:
                    # coba cari yang muat
                    cek_muat = []
                    for i in range(len(self.state)):
                        if self.hitung_total_ukuran(self.state[i]) +
ukuran <= self.kapasitas:
                            cek_muat.append(i)

                    if len(cek_muat) > 0:
                        idx = random.choice(cek_muat)
                        self.state[idx].append(item_id)
                    else:
                        # ga ada yang muat, paksa aja
                        idx = random.randint(0, len(self.state) - 1)
                        self.state[idx].append(item_id)

        # bikin beberapa kontainer overload buat state awal jelek
        jumlah_overload = max(1, int(len(self.state) * 0.3))
        for i in range(jumlah_overload):
            if len(self.state) >= 2:
                idx1 = random.randint(0, len(self.state) - 1)
                idx2 = random.randint(0, len(self.state) - 1)
                if idx1 != idx2 and len(self.state[idx2]) > 0:
                    # pindahin barang biar overload
                    item = self.state[idx2].pop(0)
                    self.state[idx1].append(item)

        # hapus kontainer kosong
        self.state = [k for k in self.state if len(k) > 0]
```

Fungsi ini bertugas untuk membangun state awal (solusi acak) yang akan digunakan sebagai titik mula pencarian. Implementasi ini sengaja dirancang untuk menghasilkan solusi yang tidak optimal (cenderung boros kontainer dan mungkin overload). Ini dicapai dengan menempatkan barang secara acak, terkadang sengaja membuat kontainer baru, dan memindahkan beberapa barang di akhir untuk memastikan ada overload. Tujuannya adalah untuk memberikan ruang yang luas bagi algoritma local search untuk melakukan perbaikan. Fungsi ini juga memastikan tidak ada kontainer kosong yang tersisa.

```
def pindah_barang(self, id_barang, idx_tujuan):
        # cari barang di kontainer mana
        idx_asal = -1
        for i in range(len(self.state)):
            if id_barang in self.state[i]:
                idx_asal = i
                break

        if idx_asal == -1:
            return

        # pindahin barangnya
        self.state[idx_asal].remove(id_barang)

        if idx_tujuan >= len(self.state):
            # bikin kontainer baru
            self.state.append([id_barang])
        else:
            self.state[idx_tujuan].append(id_barang)

        # hapus kontainer kosong
        if len(self.state[idx_asal]) == 0:
            del self.state[idx_asal]
```

Fungsi ini mencari lokasi kontainer asal dari id_barang, menghapus barang tersebut dari kontainer asalnya, dan menambahkannya ke kontainer tujuan. Jika idx_tujuan adalah indeks baru (di luar jangkauan list), fungsi ini secara otomatis akan membuat kontainer baru. Fungsi ini juga menangani pembersihan (menghapus kontainer asal) jika kontainer tersebut menjadi kosong setelah barang dipindahkan.

```
def tukar_barang(self, id1, id2):
        # cari posisi dua barang
        pos1 = None
        pos2 = None

        for i in range(len(self.state)):
            if id1 in self.state[i]:
                pos1 = (i, self.state[i].index(id1))
```

```
            if id2 in self.state[i]:
                pos2 = (i, self.state[i].index(id2))

        if pos1 is None or pos2 is None:
            return

        # kalo di kontainer yang sama, ga usah tuker
        if pos1[0] == pos2[0]:
            return

        # tuker posisi barang
        idx_k1, idx_i1 = pos1
        idx_k2, idx_i2 = pos2

        self.state[idx_k1][idx_i1] = id2
        self.state[idx_k2][idx_i2] = id1
```

Fungsi ini mencari lokasi (pos1 dan pos2) dari kedua barang. Jika kedua barang berada di kontainer yang berbeda, fungsi ini akan menukar posisi mereka secara langsung di dalam self.state. Jika barang berada di kontainer yang sama, tidak ada aksi yang dilakukan.

```
def tampilkan_solusi(self, judul="Solusi Pengepakan"):
        print("\n" + "="*40)
        print(f"{judul}")
        print("="*40)
        print(f"Total Kontainer: {len(self.state)}")
        for i, kontainer in enumerate(self.state):
            total_ukuran = self.hitung_total_ukuran(kontainer)
            print(f"\nKontainer {i+1} (Total:
{total_ukuran}/{self.kapasitas}):")
            for item_id in kontainer:
                print(f"   - {item_id} ({self.ukuran_barang[item_id]})")

        skor = self.objective_function()
        print("-" * 40)
        print(f"Skor: {skor}")
        print("="*40)
```

Fungsi helper ini digunakan untuk menampilkan representasi self.state dengan format yang mudah dibaca. Fungsi ini akan mencetak total kontainer, dan untuk setiap kontainer, akan menampilkan daftar barang di dalamnya (beserta ukurannya), total ukuran terisi, dan kapasitas maksimum. Fungsi ini juga akan mencetak nilai objective function (skor) dari solusi tersebut.

2.2.2 Hill-Climbing

```
import copy
import time

class HillClimbAlgoritma:
    def __init__(self, solusi_awal, max_sideways_moves=100):
        self.solusi_awal = solusi_awal
        self.max_sideways = max_sideways_moves
    # cari tetangga terbaik
    def cari_tetangga(self, solusi_skrg):
        best_tetangga = None
        best_skor = solusi_skrg.objective_function()

        # kumpulin semua barang dulu biar gampang
        semua_barang = []
        for kont in solusi_skrg.state:
            for brg in kont:
                semua_barang.append(brg)

        # Operator: pindah barang ke semua posisi (termasuk bikin
kontainer baru)
        for brg in semua_barang:
            for pos in range(len(solusi_skrg.state) + 1):
                tetangga = copy.deepcopy(solusi_skrg)
                tetangga.pindah_barang(brg, pos)

                skor = tetangga.objective_function()
                if skor <= best_skor:
                    best_skor = skor
                    best_tetangga = tetangga

        # Operator: tukar setiap pasangan barang
```

```
        for a in range(len(semua_barang)):
            for b in range(a + 1, len(semua_barang)):
                tet = copy.deepcopy(solusi_skrg)
                tet.tukar_barang(semua_barang[a], semua_barang[b])

                skor = tet.objective_function()
                if skor <= best_skor:
                    best_skor = skor
                    best_tetangga = tet

        return best_tetangga, best_skor

    def run(self):
        start_time = time.time()
        sol_awal = copy.deepcopy(self.solusi_awal)
        score_awal = sol_awal.objective_function()
        # sekarang adalah solusi yang sedang diproses
        current = copy.deepcopy(self.solusi_awal)
        now_skor = score_awal

        hist_score = [score_awal]
        hist_iter = [0]
        sideways = 0
        iteration = 0

        sideways_log = []
        baik_moves = 0

        print("\n" + "="*70)
        print("HILL CLIMBING")
        print("="*70)
        print("\nState Awal:")
        print("  Nilai Objective Function:", score_awal)
        print("  Jumlah Kontainer:", len(sol_awal.state))
        print("  Max Sideways:", self.max_sideways)
        print("\nSedang mencari solusi...\n")

        while True:
            iteration += 1
```

```
            # cari tetangga terbaik sekarang
            best_neighbor, best_score = self.cari_tetangga(current)

            if best_neighbor is None:
                break

            if best_score < now_skor:
                # move lebih baik
                current = best_neighbor
                now_skor = best_score
                sideways = 0
                baik_moves += 1
                hist_score.append(now_skor)
                hist_iter.append(iteration)

            elif best_score == now_skor and sideways <
self.max_sideways:
                # sideways, masih boleh
                current = best_neighbor
                sideways += 1
                hist_score.append(now_skor)
                hist_iter.append(iteration)
                sideways_log.append((iteration, now_skor, sideways))

            else:
                # nggak ada yang bisa dilakukan lagi
                break

        end_time = time.time()
        durasi = end_time - start_time
        score_akhir = current.objective_function()

        if len(sideways_log) > 0:
            print("\n" + "="*70)
            print("SIDEWAYS MOVES")
            print("="*70)
            print(f"{'SIDEWAYS KE':<15} {'ITERASI':<15} {'NILAI
OBJECTIVE FUNCTION':<20}")
            print("-"*70)
```

```
            for iter_num, scr, sw_num in sideways_log:
                print(f"{sw_num:<15} {iter_num:<15} {scr:<20}")
            print("="*70)

        print("\n" + "="*70)
        print("HASIL AKHIR")
        print("="*70)
        print("\nState Akhir:")
        print("  Nilai Objective Function:", score_akhir)
        print("  Jumlah Kontainer:", len(current.state))
        print("\nStatistik:")
        print("  Nilai Objective Function Awal:", score_awal)
        print("  Nilai Objective Function Akhir:", score_akhir)
        improvement = score_awal - score_akhir
        pct = (improvement / score_awal * 100) if score_awal != 0 else 0
        print("  Peningkatan:", improvement, f"({pct:.2f}% )")
        print("  Total Iterasi:", iteration)
        print("  Better Moves:", baik_moves)
        print("  Sideways Moves:", len(sideways_log))
        print("  Durasi:", f"{durasi:.4f} detik")
        print("  Sideways Terakhir:", f"{sideways}/{self.max_sideways}")
        print("="*70 + "\n")

        stats = {
            'solusi_awal': sol_awal,
            'solusi_akhir': current,
            'skor_awal': score_awal,
            'skor_akhir': score_akhir,
            'peningkatan': improvement,
            'persentase_peningkatan': pct,
            'total_iterasi': iteration,
            'total_better_moves': baik_moves,
            'total_sideways': len(sideways_log),
            'sideways_terakhir': sideways,
            'durasi': durasi,
            'max_sideways_moves': self.max_sideways,
            'history_skor': hist_score,
            'history_iterasi': hist_iter,
            'sideways_log': sideways_log,
```

```
            'kapasitas_kontainer': sol_awal.kapasitas
        }

        return current, stats
```

Penjelasan Implementasi Algoritma Local Search: Hill Climbing

- “import copy”: Digunakan untuk fungsi copy.deepcopy(). Ini digunakan untuk membuat salinan *independen* dari *state* solusi. Sehingga, saat kita mengecek "tetangga", kita tidak membuat *state* asli kita ikut berubah.
- “import time”: Digunakan untuk menghitung durasi eksekusi algoritma dengan mencatat waktu mulai dan waktu selesai.
- “def __init__(self, solusi_awal, max_sideways_moves=100)”: Fungsi ini adalah constructor kelas.Untuk menginisialisasi atau "menyiapkan" algoritma dengan parameter yang dibutuhkan.
- def cari_tetangga(self, solusi_skrg): Fungsi ini bertugas untuk mengeksplorasi semua kemungkinan "tetangga" dari *state* saat ini dan menemukan satu yang memiliki skor terbaik (terendah). Inisiasi yang dilakukan: best_skor diatur ke skor *state* saat ini. Algoritma akan mencari tetangga yang skornya kurang dari atau sama dengan nilai ini. Metode yang digunakan untuk mencari skor terbaik adalah
    1. Melakukan loop yang mencoba memindahkan setiap barang ke setiap kontainer yang ada, dan juga ke satu kontainer baru. Lalu bagian “copy.deepcopy(solusi_skrg)” membuat salinan baru agar solusi_skrg tidak berubah. Tetangga.pindah_barang(brg, pos) menjalankan *move*.
       Jika skor tetangga baru ini lebih baik (lebih kecil) atau sama dengan skor terbaik yang pernah ditemukan, ia akan disimpan sebagai best_tetangga.
    2. Melakukan Loop yang mencoba menukar setiap pasangan barang yang unik di dalam solusi. Sama seperti sebelumnya, ia membuat salinan, menjalankan *move* (tukar_barang), dan membandingkan skornya.

  Setelah mencoba semua kemungkinan pindah dan tukar, fungsi ini mengembalikan best_tetangga (objek solusi terbaik yang ditemukan) dan best_skor (skor dari tetangga tersebut).
- def run(self): berfungsi untuk menjalankan algoritma dari awal sampai berhenti. Untuk inisiasi akan mencatat start_time, menyalin solusi_awal ke current (yang akan jadi *state* aktif), dan menyiapkan list hist_score & hist_iter untuk melacak progres yang akan digambar di grafik. Loop ini akan terus berjalan (while True) sampai ada perintah break. Di setiap iterasi, ia memanggil cari_tetangga() untuk menemukan *move* terbaik dari *state* current. Terdapat 3 kondisi yang membuat algoritma ini melakukan 1 kali iterasi:
    1. Jika skor tetangga lebih baik (lebih kecil) dari skor sekarang. current diperbarui ke best_neighbor.sideways di-reset ke 0, karena kita menemukan "bukit" baru untuk didaki.
    2. Jika skor tetangga sama dengan skor sekarang, dan jatah sideways kita masih ada. current tetap diperbarui. sideways ditambah 1.

3. Kondisi else yang ada akan aktif jika: best_score > now_skor (semua tetangga lebih buruk). best_score == now_skor tapi sideways >= self.max_sideways (jatah *sideways move* habis). Hal ini menandakan algoritma telah "terjebak" di *local optimum*, sehingga pencarian dihentikan.

- Return: Terakhir, fungsi ini mengembalikan dua hal:
  1. current: Objek SolusiPacking yang berisi solusi akhir terbaik yang ditemukan.
  2. stats: Sebuah dictionary yang berisi semua data statistik dari proses pencarian

2.2.3 Simulated Annealing

```
import copy
import time
import math
import random
from Container import SolusiPacking

class SimulatedAnnealingAlgoritma:

    def __init__(self, solusi_awal, temperatur_awal, temperatur_akhir,
cooling_rate):
        self.solusi_awal = solusi_awal
        self.temp_awal = temperatur_awal
        self.temp_akhir = temperatur_akhir
        self.cooling_rate = cooling_rate

    def _dapatkan_tetangga_acak(self, solusi_sekarang):
        tetangga = copy.deepcopy(solusi_sekarang)

        all_items = []
        for container in tetangga.state:
            for item in container:
                all_items.append(item)

        if not all_items:
            return tetangga

        if random.random() < 0.5:
```

```
            try:
                item_to_move = random.choice(all_items)
                idx_tujuan = random.randint(0, len(tetangga.state))
                tetangga.pindah_barang(item_to_move, idx_tujuan)
            except:
                pass

        else:
            if len(all_items) >= 2:
                try:
                    item1, item2 = random.sample(all_items, 2)
                    tetangga.tukar_barang(item1, item2)
                except:
                    pass

        return tetangga

    def run(self):
        start_time = time.time()

        current = copy.deepcopy(self.solusi_awal)
        current_score = current.objective_function()

        best = copy.deepcopy(current)
        best_score = current_score

        T = self.temp_awal
        iteration = 0

        hist_score = [current_score]
        hist_iter = [0]
        hist_exp_delta = [0.0]
        hist_temp = [T]

        stuck_count = 0
        threshold = 50
        last_best = best_score
        no_improve_cnt = 0
```

```
        print("\n" + "="*70)
        print(" "*24 + "SIMULATED ANNEALING")
        print("="*70)
        print(f"\nState Awal: Nilai Objective Function =
{current_score:.2f}, T = {T:.2f}")
        print("\nSedang mencari solusi...\n")

        while T > self.temp_akhir:
            iteration += 1

            neighbor = self._dapatkan_tetangga_acak(current)
            neighbor_score = neighbor.objective_function()

            delta = neighbor_score - current_score

            exp_val = 0.0

            if delta < 0:
                current = neighbor
                current_score = neighbor_score

                if current_score < best_score:
                    best = copy.deepcopy(current)
                    best_score = current_score
                    no_improve_cnt = 0
                else:
                    no_improve_cnt += 1

                try:
                    exp_val = math.exp(delta / T)
                except (OverflowError, ZeroDivisionError):
                    exp_val = 0.0

            else:
                try:
                    exp_val = math.exp(delta / T)
                except (OverflowError, ZeroDivisionError):
                    exp_val = float('inf') if delta > 0 else 0.0
```

```
                try:
                    prob = math.exp(-delta / T)
                except (OverflowError, ZeroDivisionError):
                    prob = 0.0

                if random.random() < prob:
                    current = neighbor
                    current_score = neighbor_score
                else:
                    pass

                no_improve_cnt += 1

            if no_improve_cnt >= threshold:
                stuck_count += 1
                no_improve_cnt = 0

            hist_score.append(current_score)
            hist_iter.append(iteration)
            hist_exp_delta.append(exp_val)
            hist_temp.append(T)

            T *= self.cooling_rate

            if iteration % 1000 == 0:
                print(f"Iter: {iteration}, T: {T:.2f}, Nilai Objective
Function: {current_score:.2f}, Best: {best_score:.2f}")

        end_time = time.time()
        durasi = end_time - start_time

        print("\n" + "="*70)
        print("HASIL AKHIR")
        print("="*70)
        print(f"\nState Akhir (Terbaik):")
        print(f"  Nilai Objective Function: {best_score}")
        print(f"  Jumlah Kontainer: {len(best.state)}")
        print(f"\nStatistik:")
        print(f"  Nilai Objective Function Awal:
```

```
{self.solusi_awal.objective_function()}")
        print(f"  Nilai Objective Function Akhir: {best_score}")
        print(f"  Total Iterasi: {iteration}")
        print(f"  Frekuensi Stuck di Local Optima: {stuck_count} kali")
        print(f"  Durasi: {durasi:.4f} detik")
        print("="*70 + "\n")

        stats = {
            'solusi_awal': self.solusi_awal,
            'solusi_akhir': best,
            'skor_awal': self.solusi_awal.objective_function(),
            'skor_akhir': best_score,
            'total_iterasi': iteration,
            'durasi': durasi,
            'history_skor': hist_score,
            'history_iterasi': hist_iter,
            'history_exp_delta': hist_exp_delta,
            'history_temperature': hist_temp,
            'stuck_count': stuck_count,
            'kapasitas_kontainer': self.solusi_awal.kapasitas,
            'temp_awal': self.temp_awal,
            'temp_akhir': self.temp_akhir,
            'cooling_rate': self.cooling_rate
        }

        return best, stats
```

Ada beberapa fungsi yang terdapat pada Algoritma ini, yakni :

1. Fungsi init(self, solusi_awal, temperatur_awal, temperatur_akhir, cooling_rate)

Fungsi init ini adalah constructor untuk kelas. Fungsi ini menerima solusi awal (sebuah objek solusipacking) dan tiga parameter utama untuk proses simulated annealing, yaitu: temperatur awal, temperatur akhir, dan cooling rate. Semua nilai ini disimpan ke dalam variabel internal kelas (seperti self.temp awal) untuk digunakan oleh fungsi run nanti.

2. Fungsi _dapatkan_tetangga_acak(self, solusi_sekarang)

Fungsi ini bertugas untuk menghasilkan satu tetangga acak dari solusi yang diberikan. Berbeda dengan hill climbing yang mencoba semua tetangga, fungsi ini hanya membuat satu perubahan kecil. Pertama, ia membuat salinan dari solusi sekarang. Lalu, ia secara acak (dengan peluang 50/50) memilih antara memanggil fungsi pindah barang atau tukar barang. Barang yang dipindah

atau ditukar juga dipilih secara acak dari semua barang yang ada. Hasilnya adalah satu solusi tetangga baru yang sedikit berbeda dari aslinya.

3. Fungsi run(self)

Fungsi run adalah inti dari algoritma simulated annealing. Saat dipanggil, fungsi ini akan menjalankan keseluruhan proses pencarian.
Pertama, ia mencatat waktu mulai, lalu menginisialisasi solusi. Ada dua variabel untuk solusi: current (solusi yang sedang dieksplorasi) dan best (solusi terbaik yang pernah ditemukan). Awalnya, keduanya diisi dengan solusi awal.
Fungsi ini juga menyiapkan beberapa list untuk menyimpan histori pencarian, seperti hist score, hist iter, hist exp delta, dan hist temp. Data ini akan digunakan untuk membuat grafik visualisasi hasil eksperimen. Fungsi ini juga melacak frekuensi 'stuck' (terjebak di lokal optima) menggunakan variabel stuck count.

Fungsi pencarian utama ada di dalam loop while t > self.temp akhir. Selama temperatur (t) masih lebih tinggi dari temperatur akhir, algoritma akan terus berjalan. Setelah loop selesai (karena t sudah mencapai batas akhir), fungsi ini akan menghentikan pencatatan waktu, mencetak hasil akhir ke konsol, mengemas semua data statistik (histori, durasi, skor akhir, dll) ke dalam sebuah dictionary 'stats', dan mengembalikan solusi 'best' beserta 'stats'-nya

2.2.3 Genetic

```
import random
import copy
import time
from Container import SolusiPacking

class GeneticAlgoritma:
    def __init__(self, kapasitas_kontainer, daftar_barang,
                 pop_size=100, mutation_rate=0.1,
                 max_generasi=500, elitism_count=1):

        self.kapasitas_kontainer = kapasitas_kontainer
        self.daftar_barang = daftar_barang
        self.pop_size = pop_size
        self.mutation_rate = mutation_rate
        self.max_generasi = max_generasi
        self.elitism_count = elitism_count  # Jumlah individu terbaik
yang dipertahankan
```

```
        self.populasi = []
        self.hist_skor_terbaik = []
        self.hist_skor_rata2 = []
        self.hist_generasi = []

        # Simpan ID barang untuk validasi
        self.semua_barang_id = set(b['id'] for b in self.daftar_barang)
        self.ukuran_barang = {b['id']: b['ukuran'] for b in
self.daftar_barang}



    def _inisialisasi_populasi(self):
        self.populasi = []

        print(f"  Membuat {self.pop_size} individu dengan inisialisasi
random...")

        for _ in range(self.pop_size):
            individu = SolusiPacking(self.kapasitas_kontainer,
self.daftar_barang)
            individu.inisialisasi_random()
            self.populasi.append(individu)

    def _hitung_fitness(self, individu):
        skor = individu.objective_function()
        if skor == 0:
            return float('inf')
        return 1 / skor

    def _seleksi_roulette(self):
        fitness_list = [self._hitung_fitness(ind) for ind in
self.populasi]
        total_fitness = sum(fitness_list)

        if total_fitness == 0 or total_fitness == float('inf'):
            return random.choice(self.populasi)

        prob_kumulatif = []
```

```
        kumulatif = 0
        for fitness in fitness_list:
            kumulatif += fitness / total_fitness
            prob_kumulatif.append(kumulatif)

        r = random.random()

        for i, prob in enumerate(prob_kumulatif):
            if r <= prob:
                return self.populasi[i]

        return self.populasi[-1]

    def _crossover(self, parent1, parent2):
        child = SolusiPacking(self.kapasitas_kontainer,
self.daftar_barang)

        state1 = parent1.state
        start, end = sorted(random.sample(range(len(state1)), 2))

        child_partial = [copy.deepcopy(k) for k in state1[start:end]]
        items_exist = set(item for k in child_partial for item in k)

        items_missing = []
        for kontainer in parent2.state:
            for item_id in kontainer:
                if item_id not in items_exist:
                    items_missing.append(item_id)

        state_final = child_partial

        for item_id in items_missing:
            item_size = self.ukuran_barang[item_id]
            placed = False

            for kontainer in state_final:
                total = child.hitung_total_ukuran(kontainer)
                if total + item_size <= self.kapasitas_kontainer:
                    kontainer.append(item_id)
```

```
                    placed = True
                    break

            if not placed:
                state_final.append([item_id])

        child.state = [k for k in state_final if k]

        items_in_child = set(item for k in child.state for item in k)
        if items_in_child != self.semua_barang_id:
             return copy.deepcopy(parent1)

        return child


    def _mutasi(self, individu, num_mutations=1):
        for _ in range(num_mutations):
            if random.random() > self.mutation_rate:
                continue

            move_type = random.choice(['pindah', 'tukar'])

            if move_type == 'pindah' and len(individu.state) > 0:
                try:
                    idx_asal = random.randint(0, len(individu.state) -
1)
                    if not individu.state[idx_asal]:
                        continue

                    item = random.choice(individu.state[idx_asal])
                    idx_tujuan = random.randint(0, len(individu.state))

                    individu.pindah_barang(item, idx_tujuan)
                except:
                    continue

            elif move_type == 'tukar' and len(individu.state) >= 2:
                try:
                    idx1 = random.randint(0, len(individu.state) - 1)
```

```
                    idx2 = random.randint(0, len(individu.state) - 1)

                    if idx1 == idx2 or not individu.state[idx1] or not
individu.state[idx2]:
                        continue

                    item1 = random.choice(individu.state[idx1])
                    item2 = random.choice(individu.state[idx2])

                    individu.tukar_barang(item1, item2)
                except:
                    continue


    def run(self):
        start_time = time.time()

        print("\n" + "="*70)
        print("GENETIC ALGORITHM")
        print("="*70)
        print(f"\nParameter:")
        print(f"  Jumlah Populasi: {self.pop_size}")
        print(f"  Max Generasi: {self.max_generasi}")
        print(f"  Mutation Rate: {self.mutation_rate}")
        print(f"  Elitism: {self.elitism_count} individu" if
self.elitism_count > 0 else "  Elitism: Tidak aktif")
        print(f"  Seleksi: Roulette Wheel")
        print("\nMembuat populasi awal...")

        self._inisialisasi_populasi()

        best_init = min(self.populasi, key=lambda x:
x.objective_function())
        score_init = best_init.objective_function()

        scores = [ind.objective_function() for ind in self.populasi]
        best_gen = min(scores)
        avg_gen = sum(scores) / len(scores)
        worst_gen = max(scores)
```

```
        self.hist_skor_terbaik.append(best_gen)
        self.hist_skor_rata2.append(avg_gen)
        self.hist_generasi.append(0)

        print(f"\nStatistik Populasi Awal:")
        print(f"  Nilai Objective Function Terbaik: {best_gen:.2f}")
        print(f"  Nilai Objective Function Rata-rata: {avg_gen:.2f}")
        print(f"  Nilai Objective Function Terburuk: {worst_gen:.2f}")
        print(f"  Range: {worst_gen - best_gen:.2f}")
        print("\nMemulai evolusi...\n")

        for gen in range(1, self.max_generasi + 1):
            new_pop = []

            if self.elitism_count > 0:
                sorted_pop = sorted(self.populasi, key=lambda x:
x.objective_function())
                elite = [copy.deepcopy(ind) for ind in
sorted_pop[:self.elitism_count]]
                new_pop.extend(elite)

            while len(new_pop) < self.pop_size:
                p1 = self._seleksi_roulette()
                p2 = self._seleksi_roulette()

                child = self._crossover(p1, p2)
                self._mutasi(child)

                new_pop.append(child)

            self.populasi = new_pop

            scores = [ind.objective_function() for ind in self.populasi]
            best_gen = min(scores)
            avg_gen = sum(scores) / len(scores)

            self.hist_skor_terbaik.append(best_gen)
            self.hist_skor_rata2.append(avg_gen)
```

```
            self.hist_generasi.append(gen)

            if gen % 100 == 0 or gen == self.max_generasi:
                print(f"Generasi {gen}/{self.max_generasi} | Nilai
Objective Function: {best_gen:.2f} | Rata-rata: {avg_gen:.2f}")

        end_time = time.time()
        durasi = end_time - start_time

        best_final = min(self.populasi, key=lambda x:
x.objective_function())
        score_final = best_final.objective_function()

        improvement = score_init - score_final
        pct = (improvement / score_init * 100) if score_init != 0 else 0

        print("\n" + "="*70)
        print("HASIL AKHIR")
        print("="*70)
        print(f"\nState Akhir (Terbaik):")
        print(f"  Nilai Objective Function: {score_final}")
        print(f"  Jumlah Kontainer: {len(best_final.state)}")
        print(f"\nStatistik:")
        print(f"  Nilai Objective Function Awal: {score_init}")
        print(f"  Nilai Objective Function Akhir: {score_final}")
        print(f"  Peningkatan: {improvement} ({pct:.2f}%)")
        print(f"  Total Generasi: {self.max_generasi}")
        print(f"  Durasi: {durasi:.4f} detik")
        print("="*70 + "\n")

        stats = {
            'solusi_awal': best_init,
            'solusi_akhir': best_final,
            'skor_awal': score_init,
            'skor_akhir': score_final,
            'peningkatan': improvement,
            'persentase_peningkatan': pct,
            'total_generasi': self.max_generasi,
            'durasi': durasi,
```

```
            'pop_size': self.pop_size,
            'mutation_rate': self.mutation_rate,
            'elitism_count': self.elitism_count,
            'history_skor_terbaik': self.hist_skor_terbaik,
            'history_skor_rata2': self.hist_skor_rata2,
            'history_generasi': self.hist_generasi,
            'kapasitas_kontainer': self.kapasitas_kontainer
        }

        return best_final, stats
```

Kelas GeneticAlgorithm adalah inti algoritma genetika (GA) yang berfungsi mencari solusi optimal dalam menempatkan barang ke dalam kontainer dengan meminimalkan nilai *objective function* (misalnya jumlah kontainer yang digunakan atau ruang kosong total).

Kelas ini dinisialisasi dengan fungsi __init__(). Ada beberapa parameter seperti kapasitas_konteiner sebagai batas ukuran tiap kontainer, daftar barang yang merupakan list berisi dict tiap barang, pop_size sebagai jumlah individu (solusi) dalam populasi, mutation_rate sebagai probabilitas mutasi, max_generasi sebagai jumlah iterasi evolusi maksimum, dan elitisim_count sebagai jumlah individu terbaik yang akan dipertahankan tiap generasi yang saya tetapkan sebagai 1. Ada beberapa variabel seperti populasi sebagai kumpulan individu solusi, hist_skor_terbaik, hist_skor_rata2, dan hist_generasi sebagai histori statistik tiap generasi, semua_barang_id sebagai himpunan ID semua barang, dan ukuran_barang sebagai peta id barnag ke ukuran barang.

Kemudian ada fungsi inisialisasi_populasi() yang akan membuat populasi awal secara acak dengan sebanyak self.pop_size, diamana tiap individu adalah objek dari SolusiPacking yang menggunakan inisialisasi_random() untuk menyusun barang acak ke dalam kontainer.

Kemudian fungsi hitung_fitness() yang mengubah objective function (yang ingin diminimalkan) menjadi fit**ness** (yang ingin dimaksimalkan). Semakin kecil nilai objective function, semakin besar fitness-nya.

Kemudian fungsi seleksi_roulette() yang akan melakukan seleksi orangtua dengan metode *Roulette Wheel Selection*. Dimana tiap individu memiliki peluang proporsional dengan nilai fitness-nya dan semakin baik (fitness tinggi), semakin besar peluang dipilih sebagai parent.

Kemudian fungsi crossover() untuk menghasilkan anak baru (child) dari dua parent. Caranya adalah dengan mengambil sebagian gen dari parent1 (subset kontainer secara acak) dan isi sisanya dengan barang dari parent2 yang belum dimasukkan. Pastikan kapasitas kontainer tidak dilanggar. Jika anak tidak valid (barang tidak lengkap), gunakan salinan parent1. Tapi pada

kasus ini child yang dihasilkan oleh 2 parent hanya 1 (tidak benar-benar pindah silang) karena saya ingin menyesuaikan dengan masalah packing.

Kemudian fungsi mutasi() untuk melakukan mutasi acak untuk menjaga keberagaman populasi. Dengan besar probabilitas kejadian mutasi sebesar mutation_rate, ada 2 hal yang dapat terjadi yaitu pindah, memindahkan barang dari satu kontainer ke kontainer lain, dan tukar, menukar dua barang antar kontainer, menggunakan fungsi pindah_barang() dan tukar_barang() dari kelas SolusiPacking.

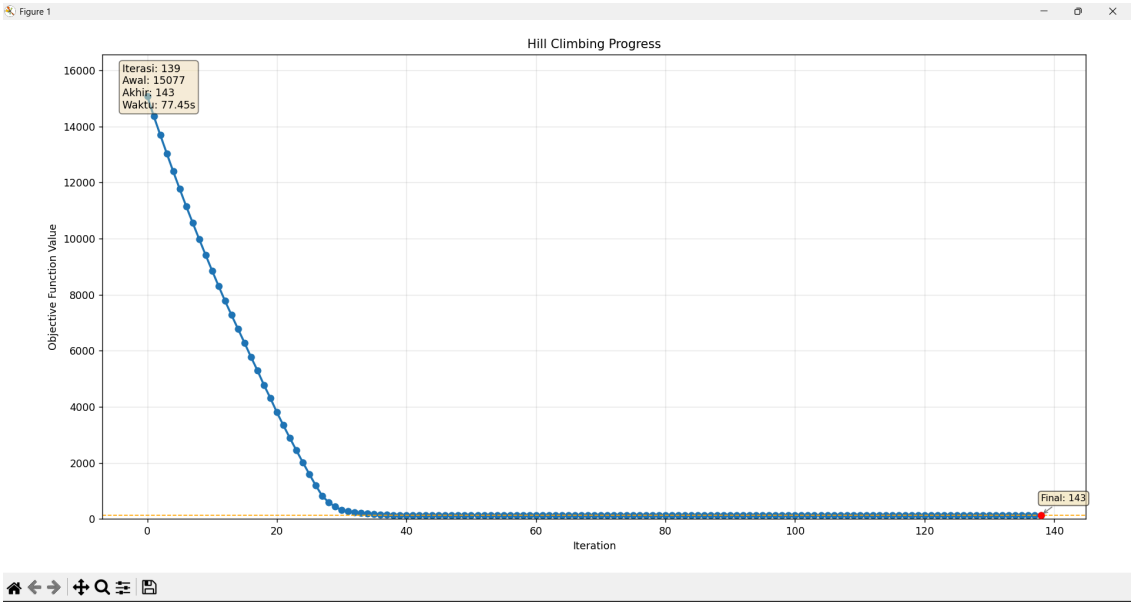
Terakhir ada fungsi run(). Fungsi ini akan menginisialisasi populasi awal dan evaluasi skor awal. Kemudian untuk setiap generasi akan dipertahankan individu elit, melakukan seleksi orang tua, pindah silang, mutasi, kemudian mencatat statistik tiap generasi. Setelahnya akan ditemukan individu terbaik (best_final), dibandingkan dengan solusi awal, dan melakukan perhitungan dan visualisasi.

BAB III
Hasil Eksperimen

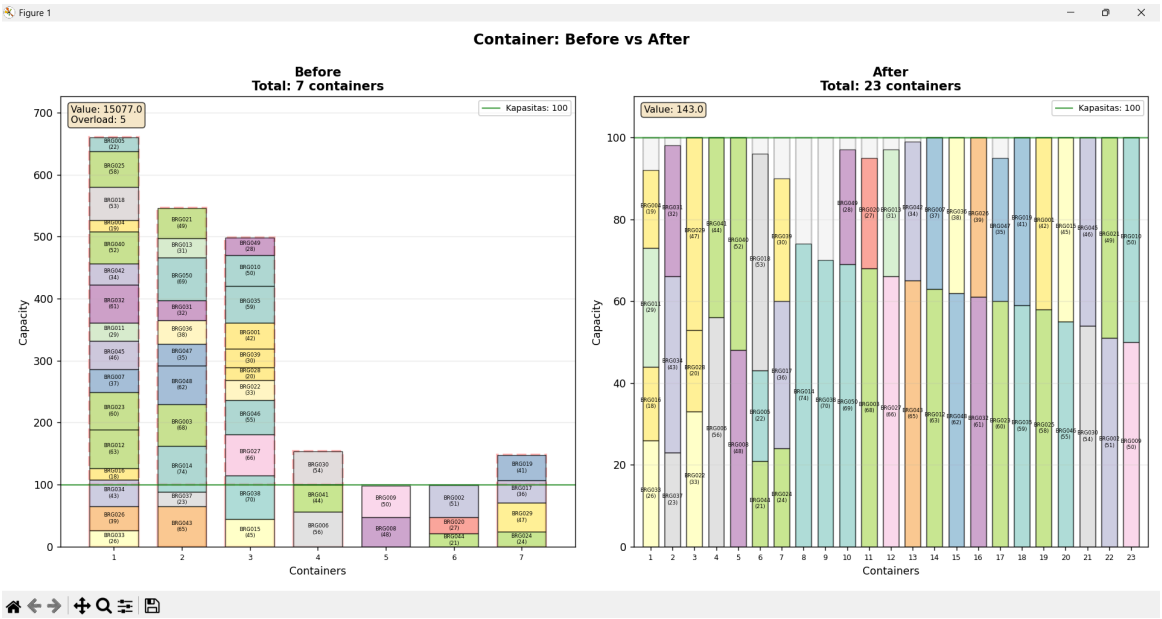
3.1 Hill-Climbing

3.1.1 Test Case 1

Berikut merupakan hasil dari penggunaan algoritma Hill-Climbing sideways pada test case 1:



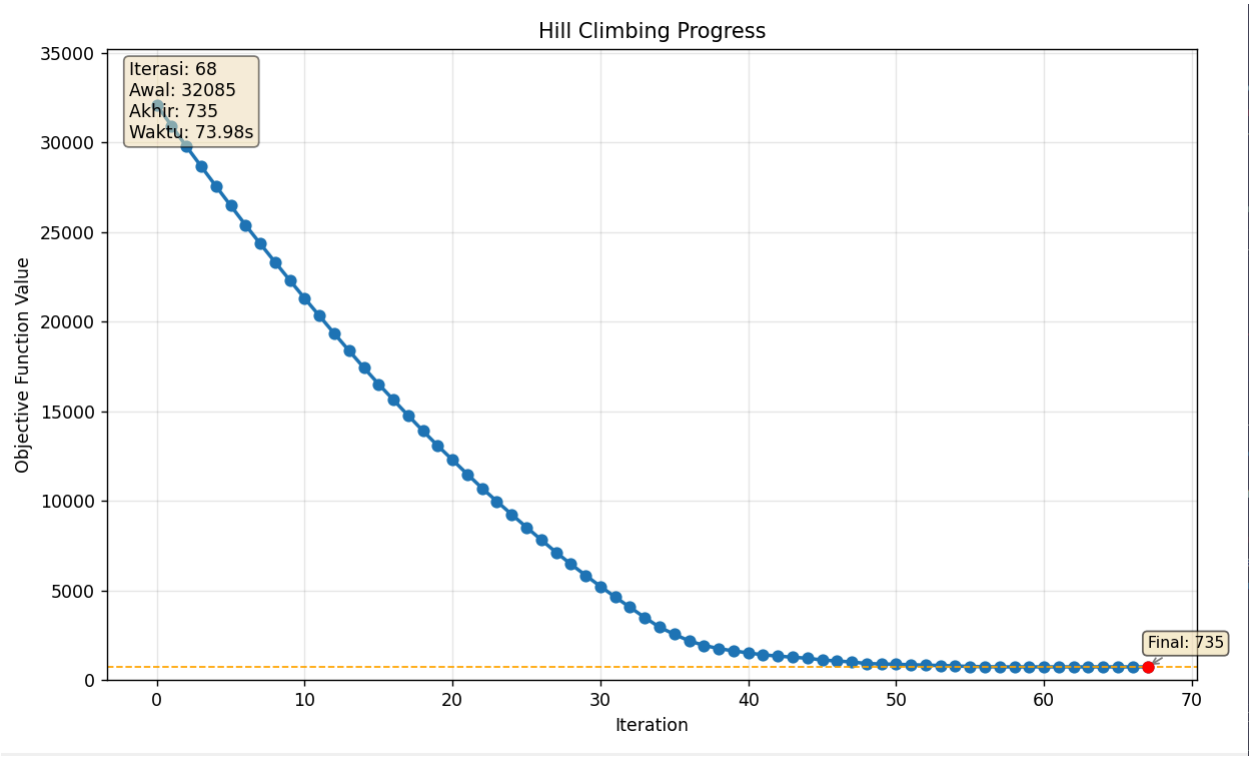
Gambar 3.1 Grafik test case 1 menggunakan Hill-Climbing with Sideways Move



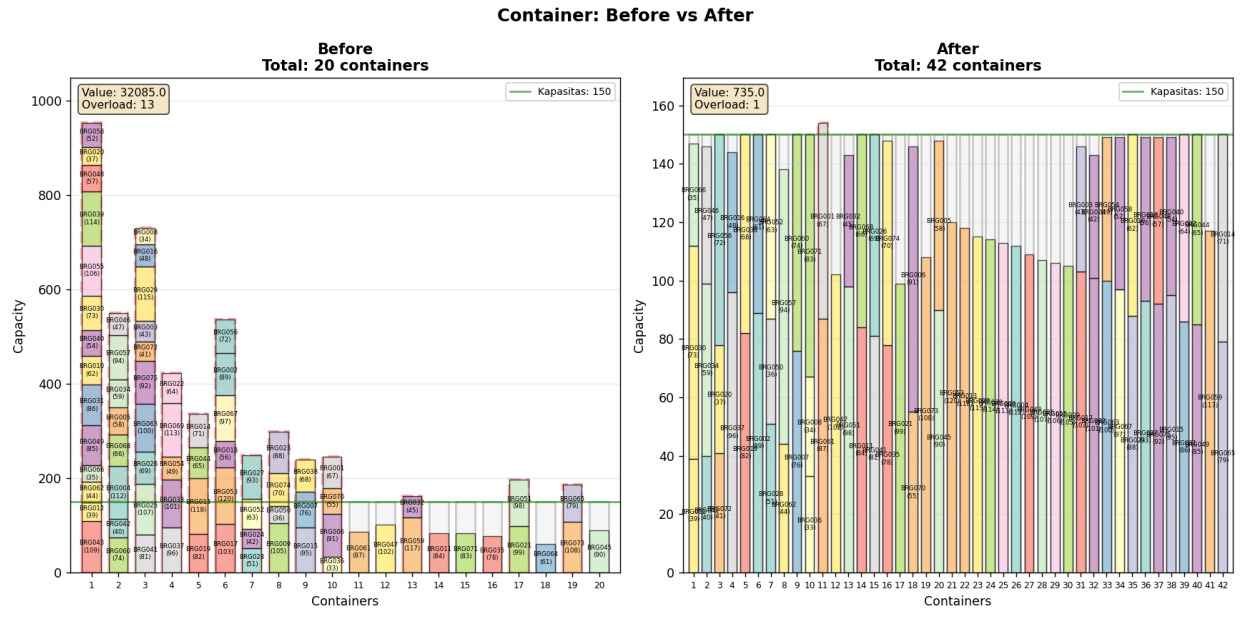
Gambar 3.2 Gambar kondisi before vs after test case 1 menggunakan Hill-Climbing with Sideways Move

3.1.2 Test Case 2

Berikut merupakan hasil dari penggunaan algoritma Hill-Climbing sideways pada test case 2:



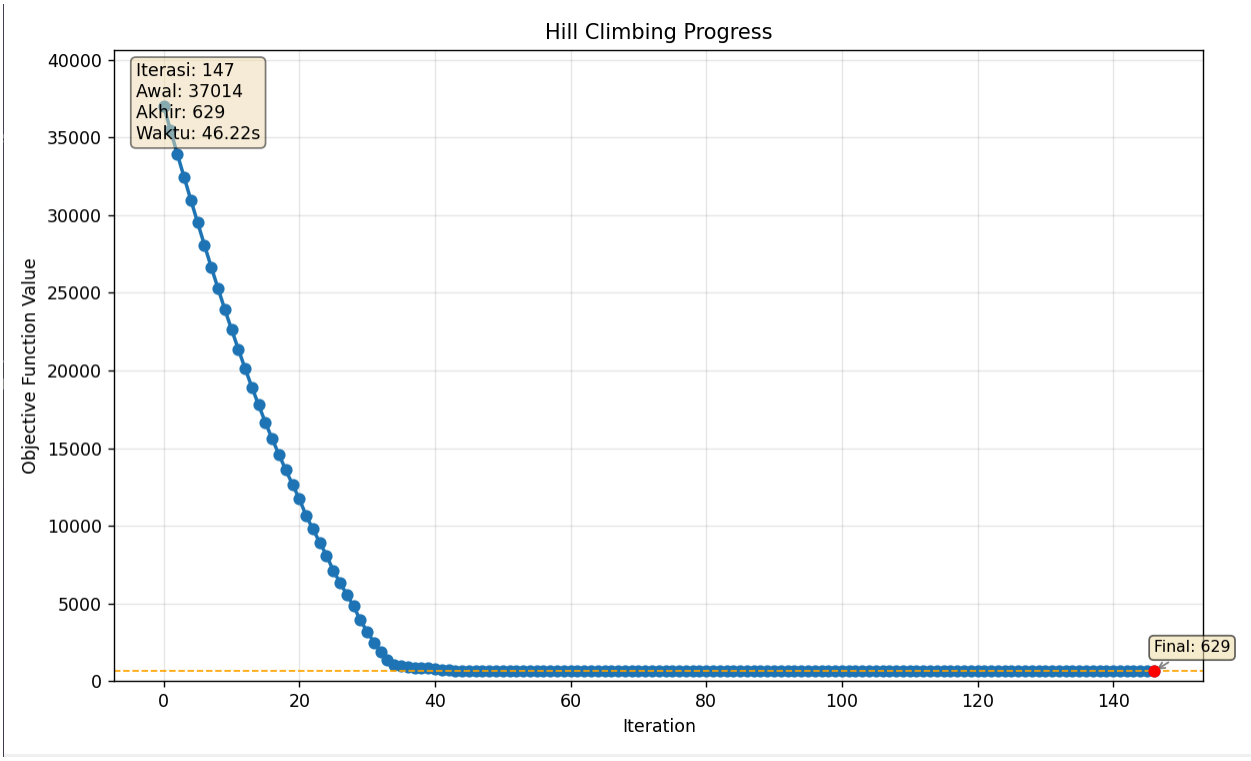
Gambar 3.3 Grafik test case 2 menggunakan Hill-Climbing with Sideways Move



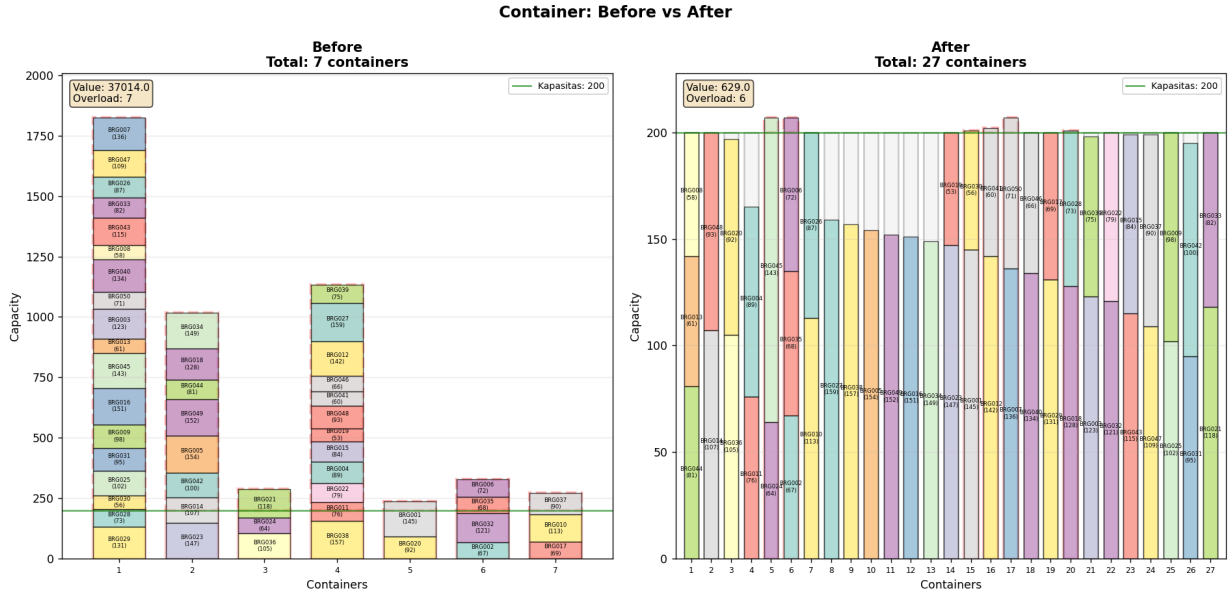
Gambar 3.4 Gambar kondisi before vs after test case 2 menggunakan Hill-Climbing with Sideways Move

3.1.3 Test Case 3

Berikut merupakan hasil dari penggunaan algoritma Hill-Climbing sideways pada test case 3:



Gambar 3.5 Grafik test case 3 menggunakan Hill-Climbing with Sideways Move

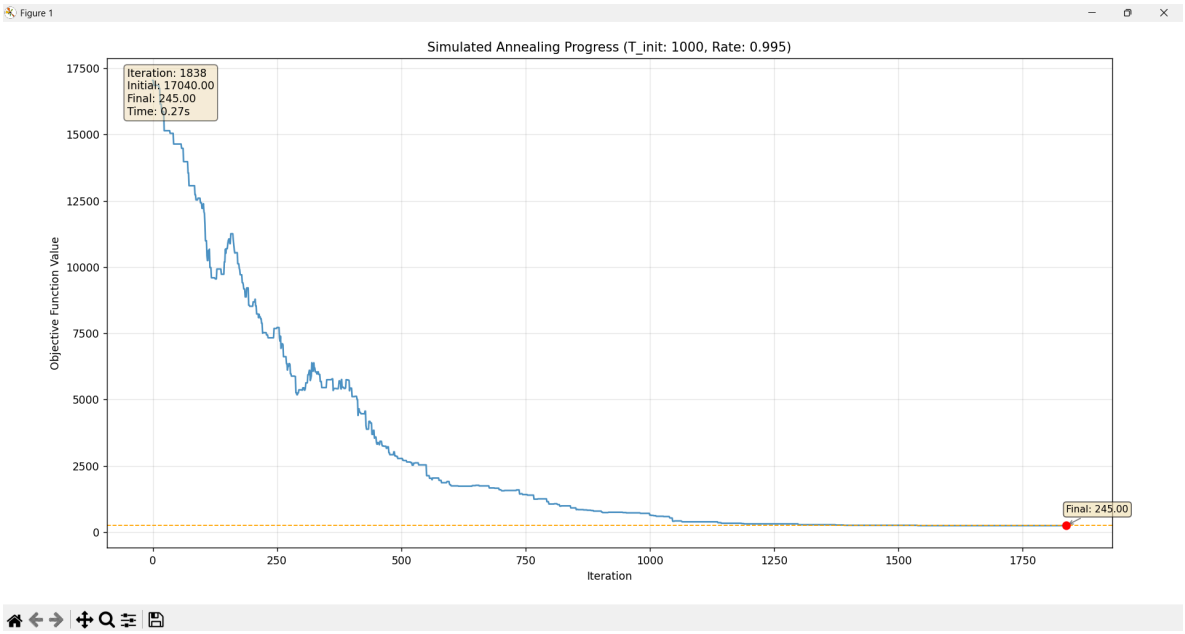


Gambar 3.6 Gambar kondisi before vs after test case 3 menggunakan Hill-Climbing with Sideways Move

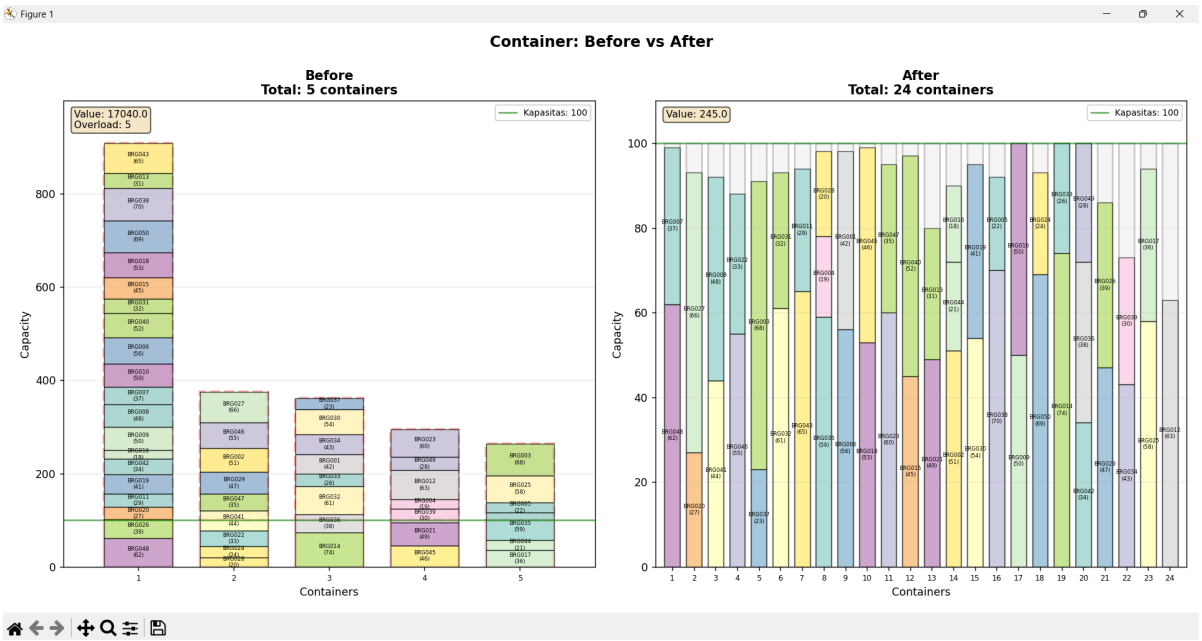
3.2 Simulated Annealing

3.2.1 Test Case 1

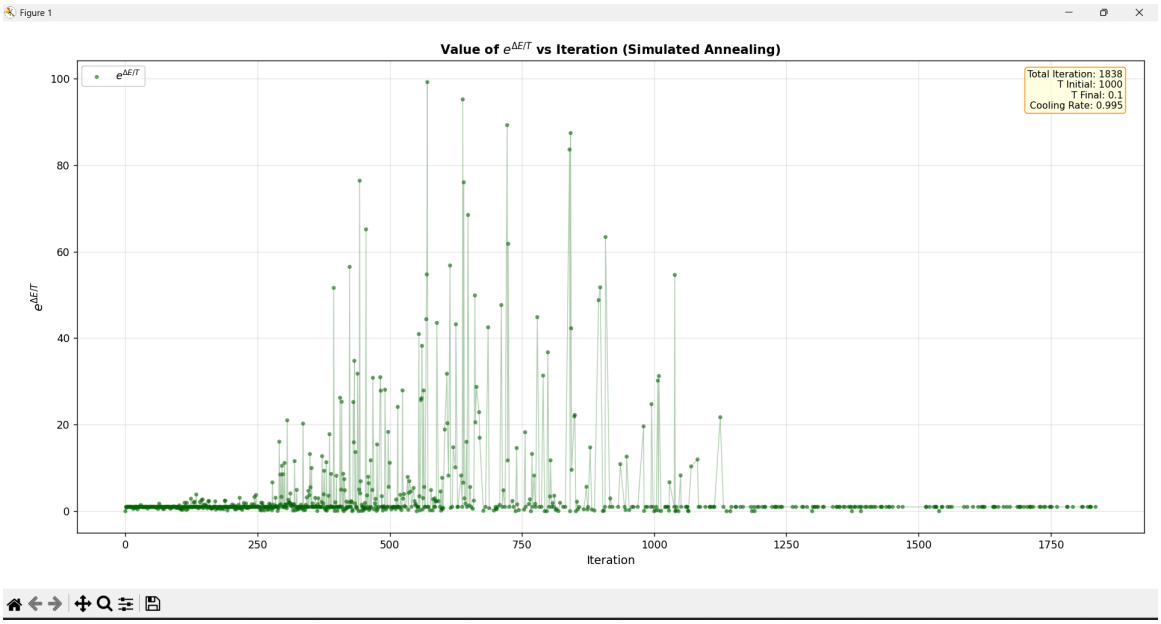
Test case dengan 50 barang dan kapasitas container = 100
Temperatur Awal = 1000 (default dari code yang dibuat)
Temperatur Akhir = 0.1 (default dari code yang dibuat)
Cooling Rate = 0.9995 (default dari code yang dibuat)



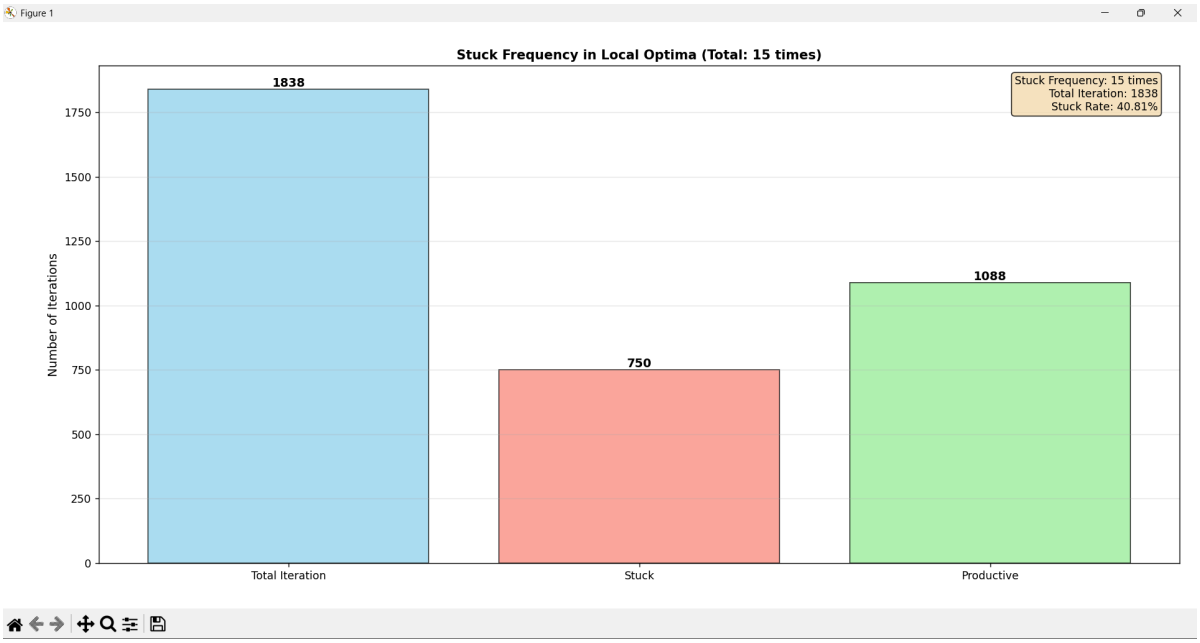
Gambar 3.7 Grafik Test Case 1 dengna Simulated Annealing



Gambar 3.8 Pemetaan Gambaran Kondisi State Awal dan State Akhir untuk Test Case 1 dengan Simulated Annealing



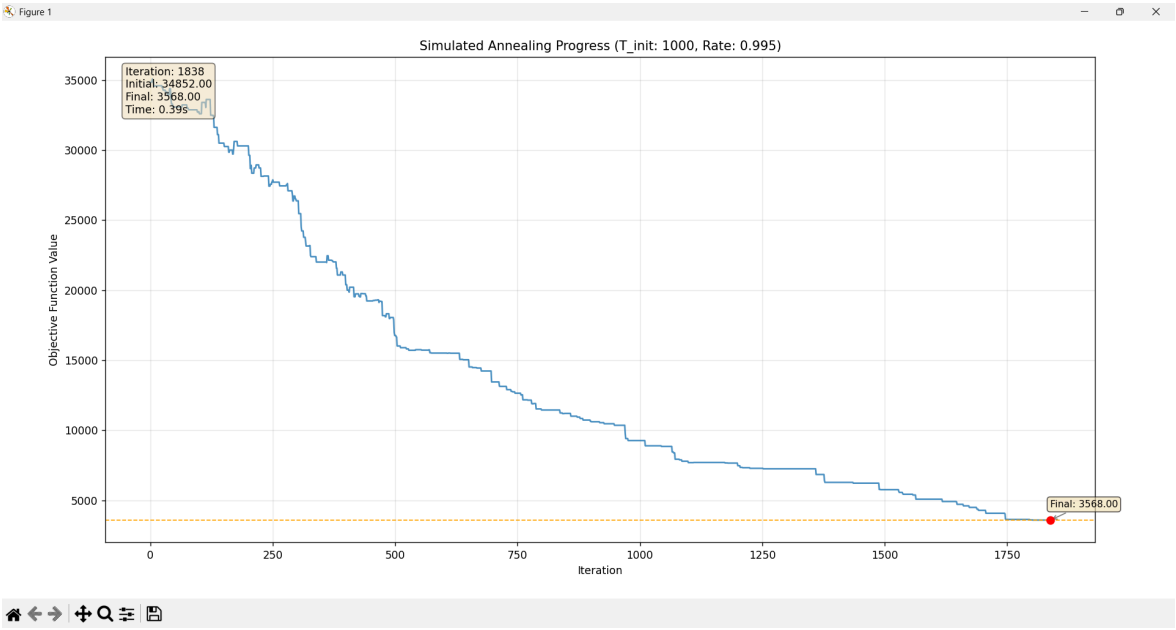
Gambar 3.9 Pemetaan value Faktor Boltzmann terhadap banyaknya iterasi untuk Test Case 1 dengan Simulated Annealing



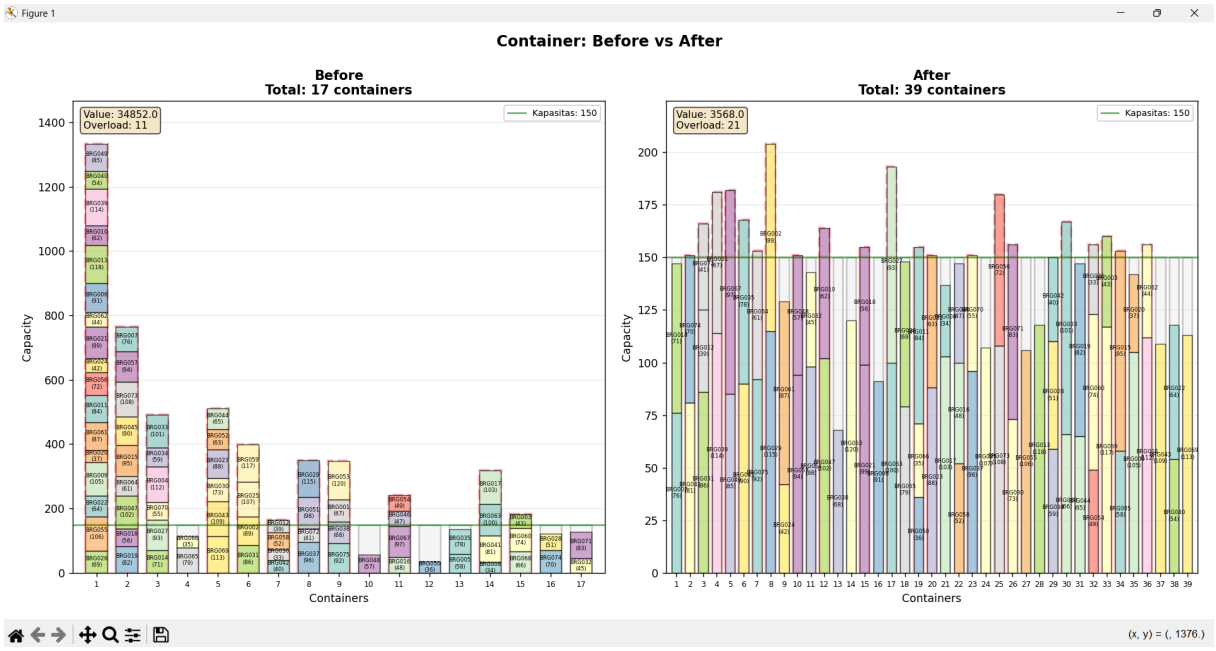
Gambar 3.10 Frekuensi Stuck di local optima untuk Test Case 1 dengan Simulated Annealing

3.2.2 Test Case 2

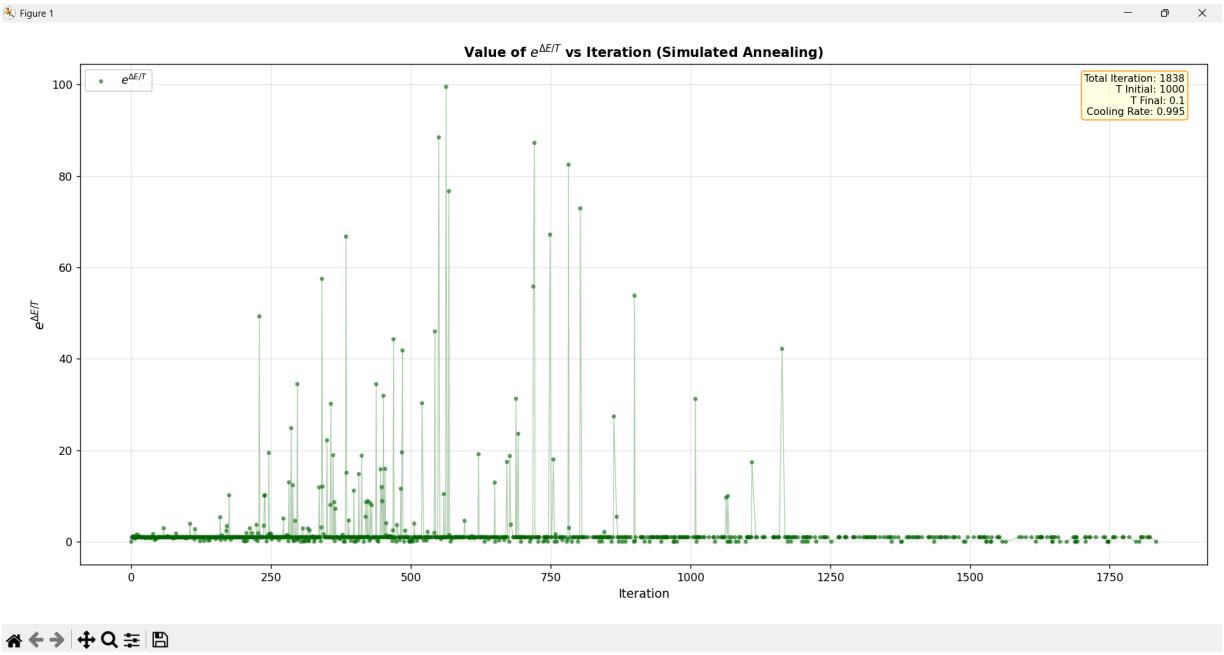
Test case dengan 75 barang dan kapasitas container = 150
Temperatur Awal = 1000 (default dari code yang dibuat)
Temperatur Akhir = 0.1 (default dari code yang dibuat)
Cooling Rate = 0.9995 (default dari code yang dibuat)



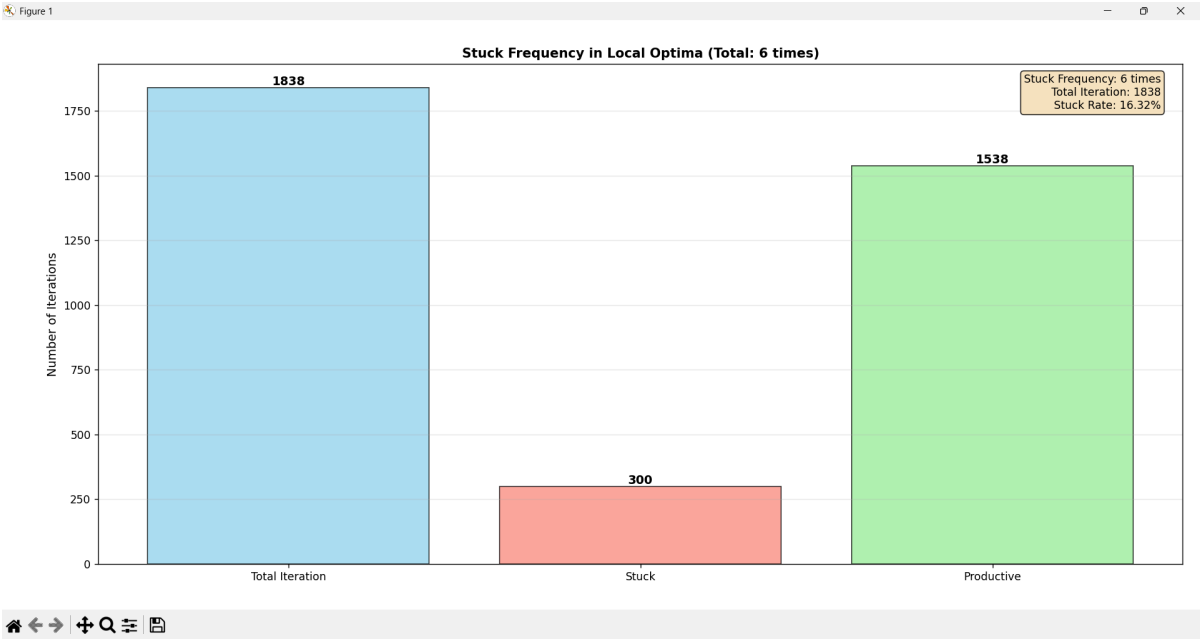
Gambar 3.11 Grafik Test Case 2 dengna Simulated Annealing



Gambar 3.12 Pemetaan Gambaran Kondisi State Awal dan State Akhir untuk Test Case 2 dengan Simulated Annealing



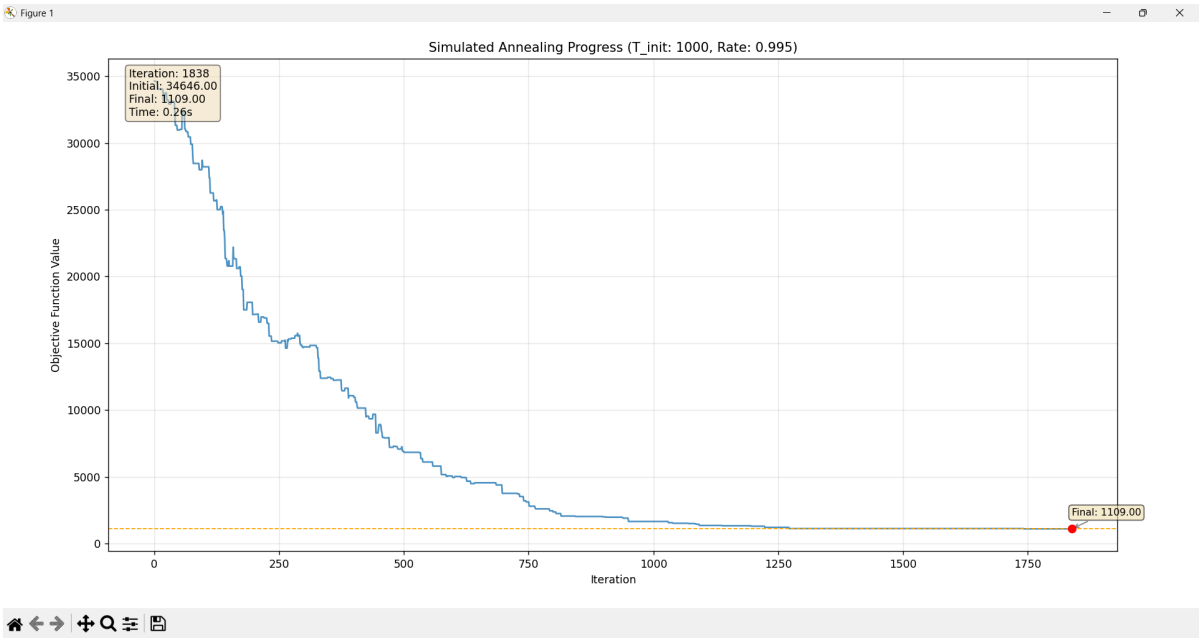
Gambar 3.13 Pemetaan value Faktor Boltzmann terhadap banyaknya iterasi untuk Test Case 2 dengan Simulated Annealing



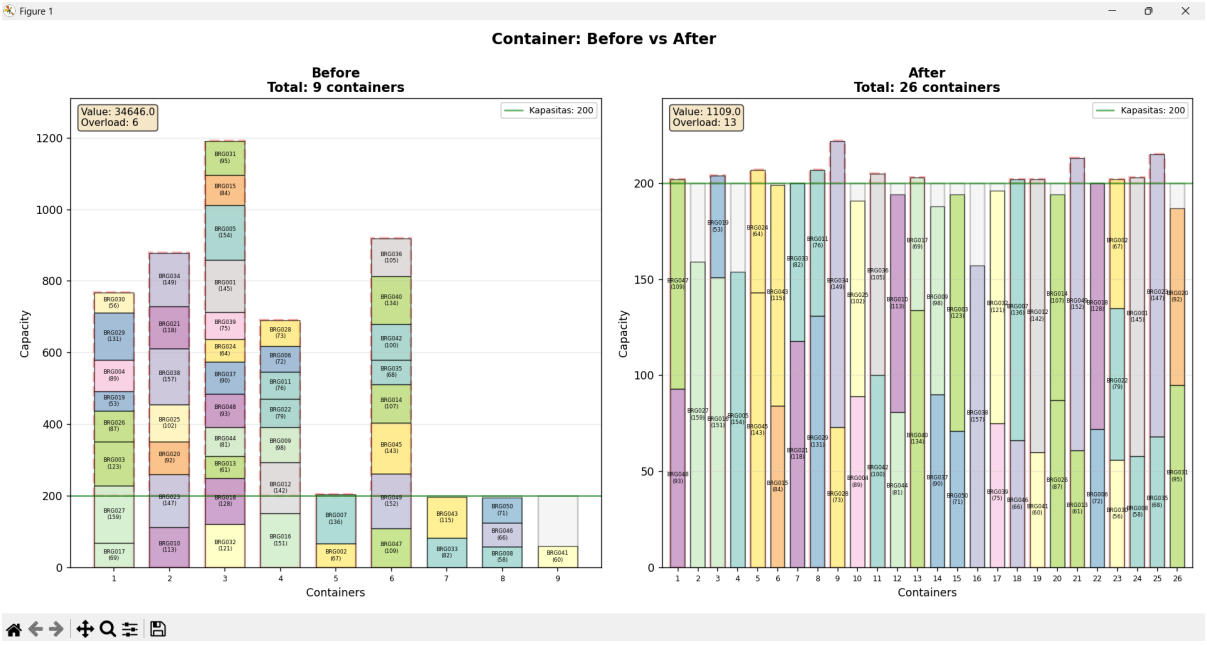
Gambar 3.14 Frekuensi Stuck di local optima untuk Test Case 2 dengan Simulated Annealing

3.2.3 Test Case 3

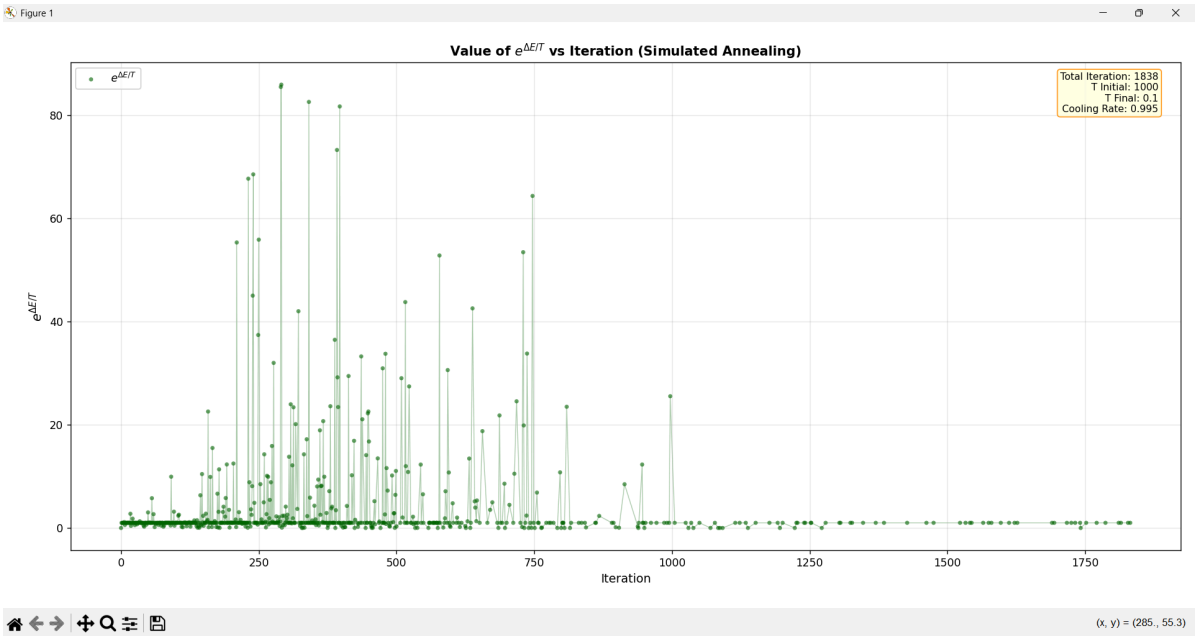
Test case dengan 50 barang dan kapasitas container = 200
Temperatur Awal = 1000 (default dari code yang dibuat)
Temperatur Akhir = 0.1 (default dari code yang dibuat)
Cooling Rate = 0.9995 (default dari code yang dibuat)



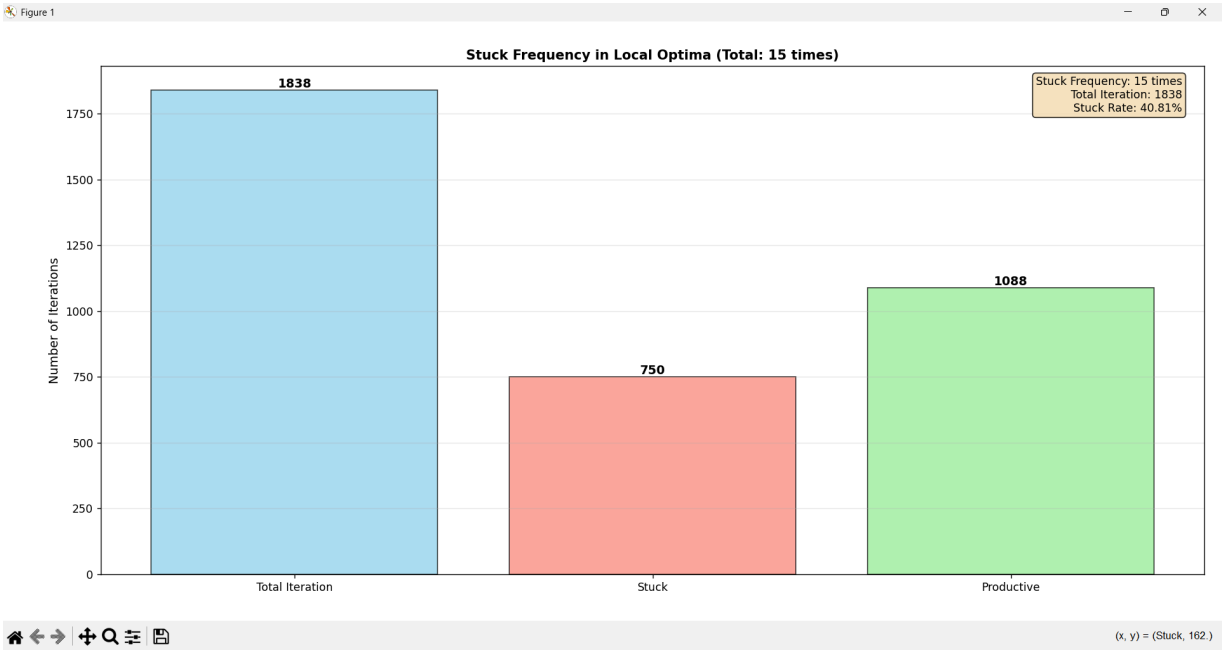
Gambar 3.15 Grafik Test Case 3 dengna Simulated Annealing



Gambar 3.16 Pemetaan Gambaran Kondisi State Awal dan State Akhir untuk Test Case 3 dengan Simulated Annealing



Gambar 3.17 Pemetaan value Faktor Boltzmann terhadap banyaknya iterasi untuk Test Case 3 dengan Simulated Annealing

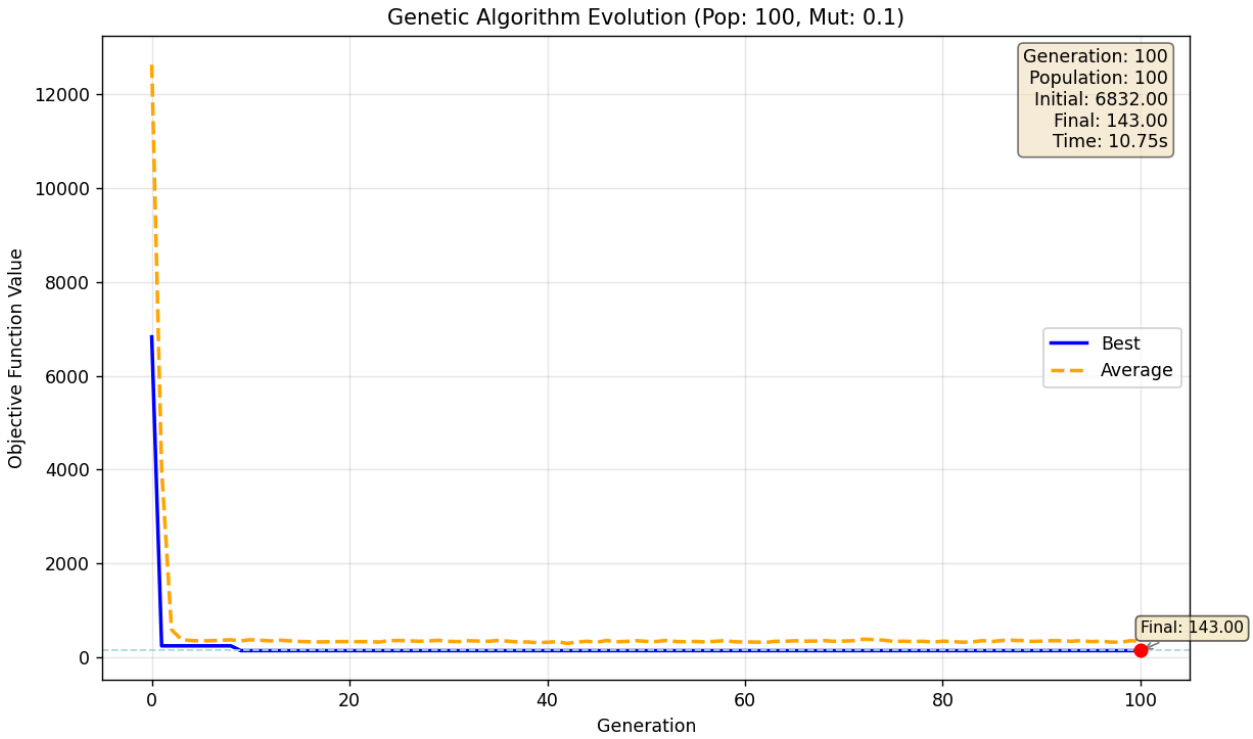


Gambar 3.18 Frekuensi Stuck di local optima untuk Test Case 2 dengan Simulated Annealing

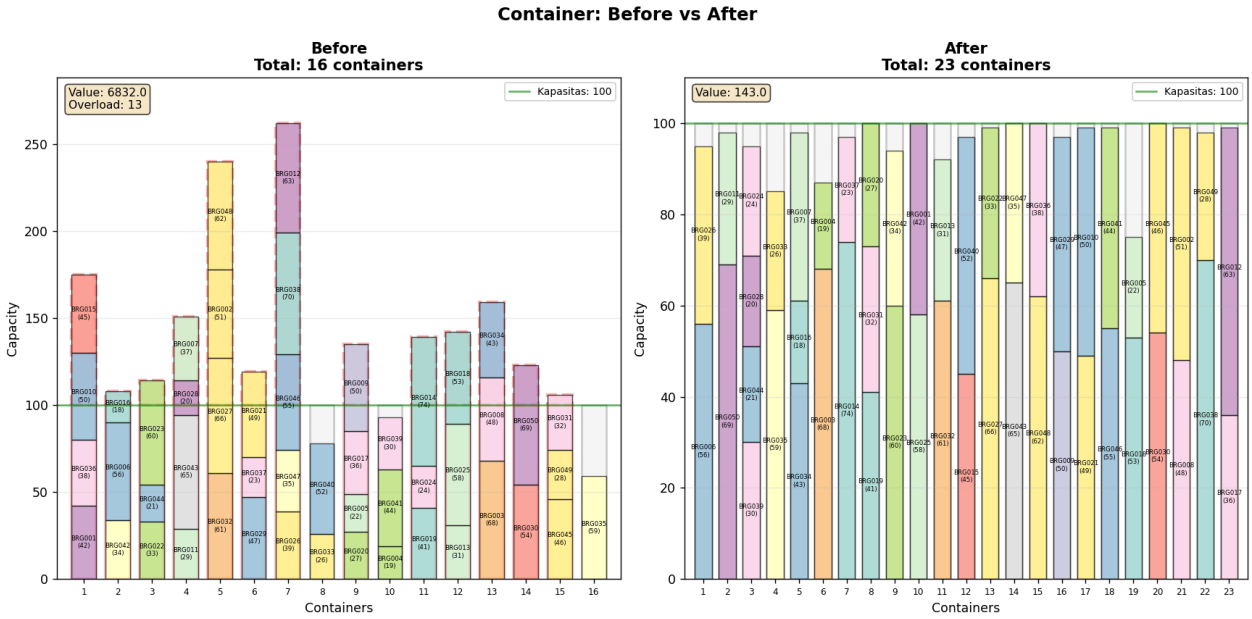
3.3 Genetic

3.2.1 Variabel Kontrol: Populasi

Test case 1
Populasi: 100
Iterasi: 100
Nilai objektif akhir: 143.00
Waktu: 10.75s

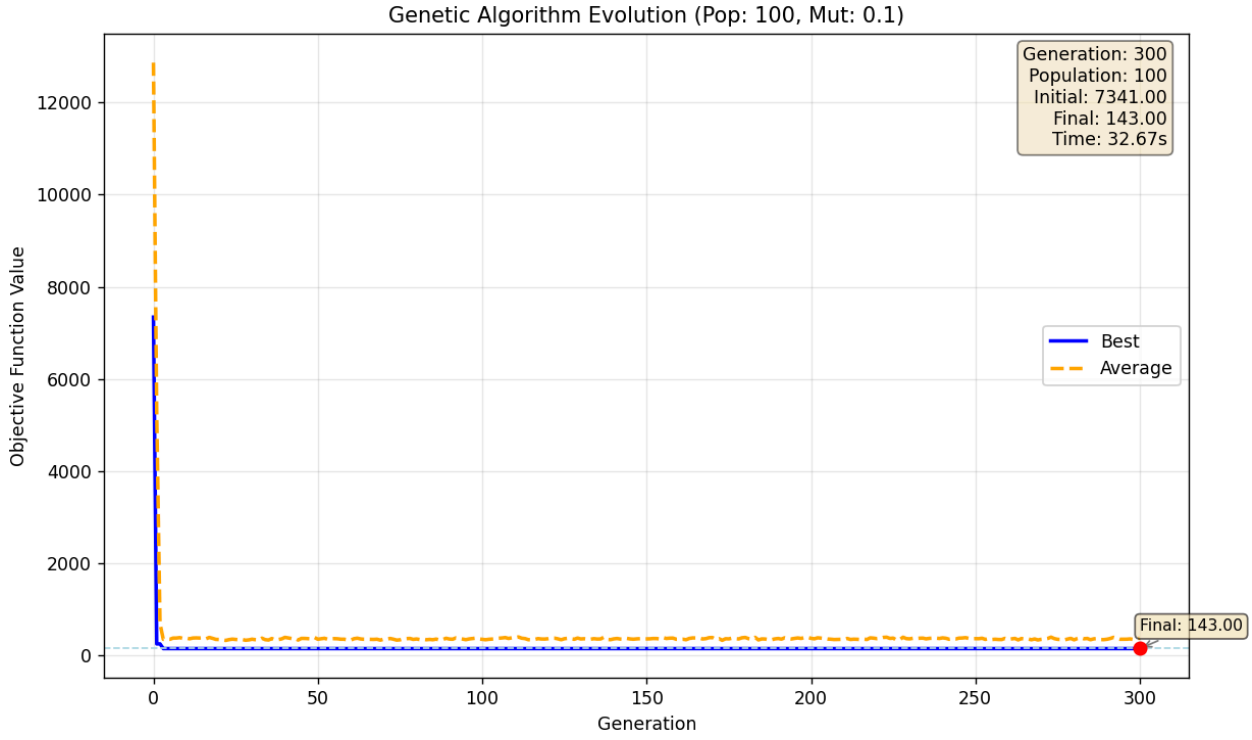


Gambar 3.19 Plot skor terbaik dan rata-rata tiap generasi terhadap iterasinya dengan populasi 100 dan iterasi 100

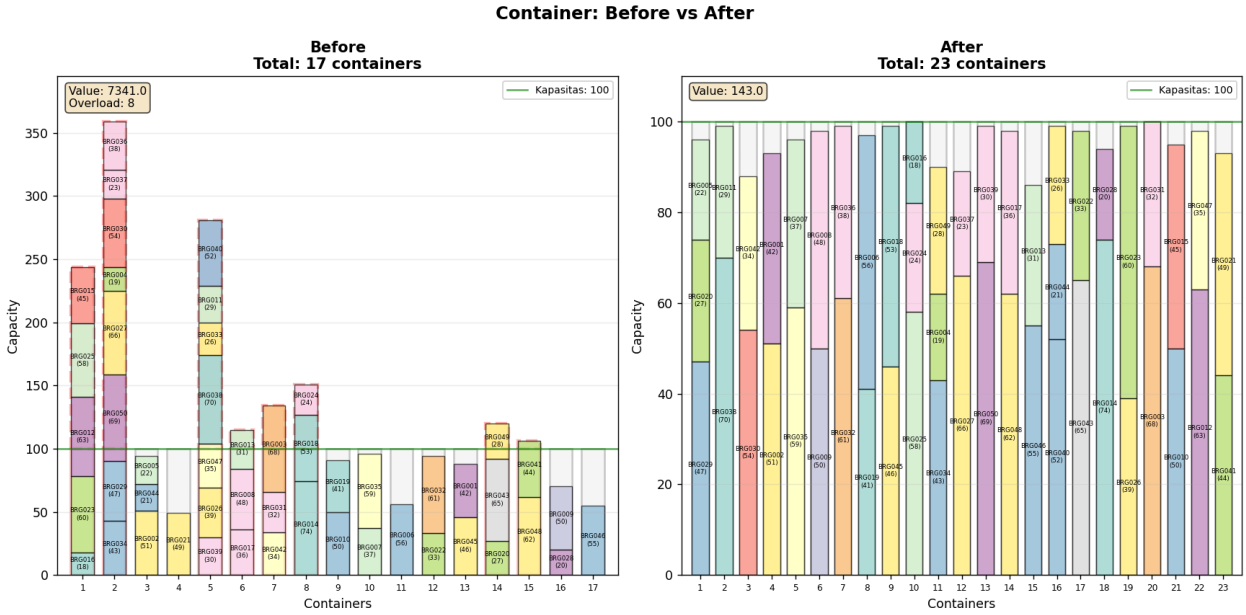


Gambar 3.20 Visualisasi perbardingan *state* awal dengan *state* akhir untuk populasi 100 dan iterasi 100

Populasi: 100
Iterasi: 300
Nilai objektif akhir: 143.00
Waktu: 32.67s

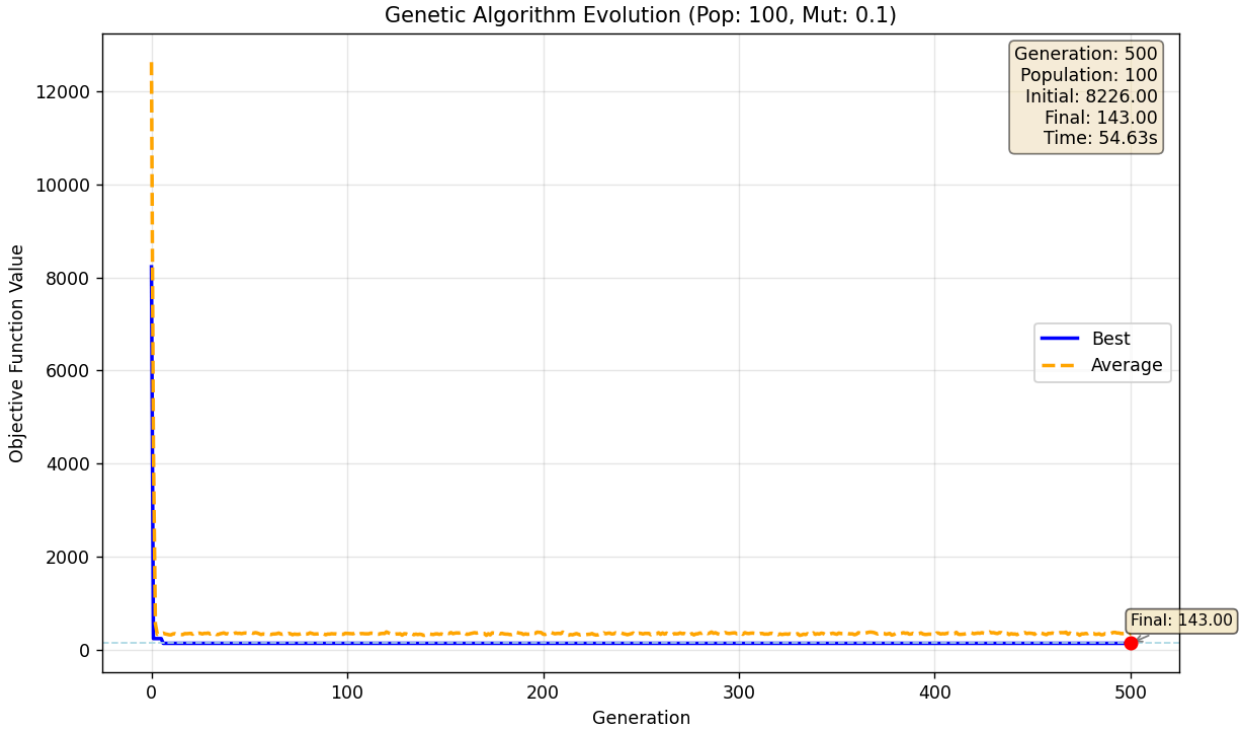


Gambar 3.21 Plot skor terbaik dan rata-rata tiap generasi terhadap iterasinya dengan populasi 100 dan iterasi 300

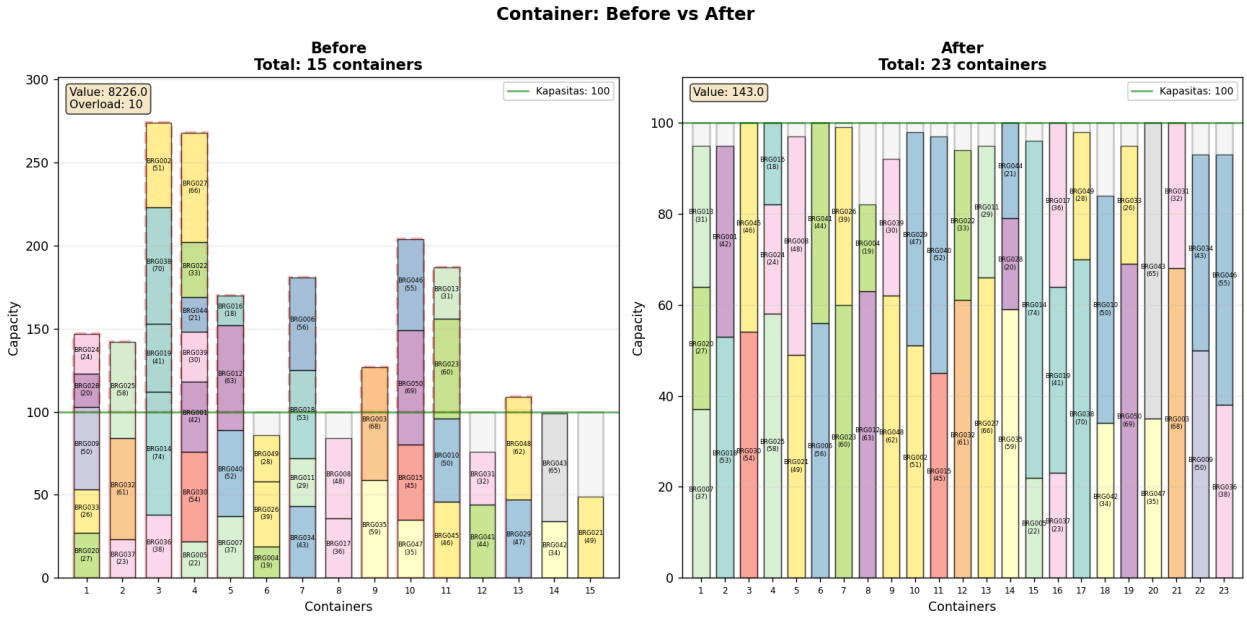


Gambar 3.22 Visualisasi perbardingan *state* awal dengan *state* akhir untuk populasi 100 dan iterasi 300

Populasi: 100
Iterasi: 500
Nilai objektif akhir: 143.00
Waktu: 54.63s



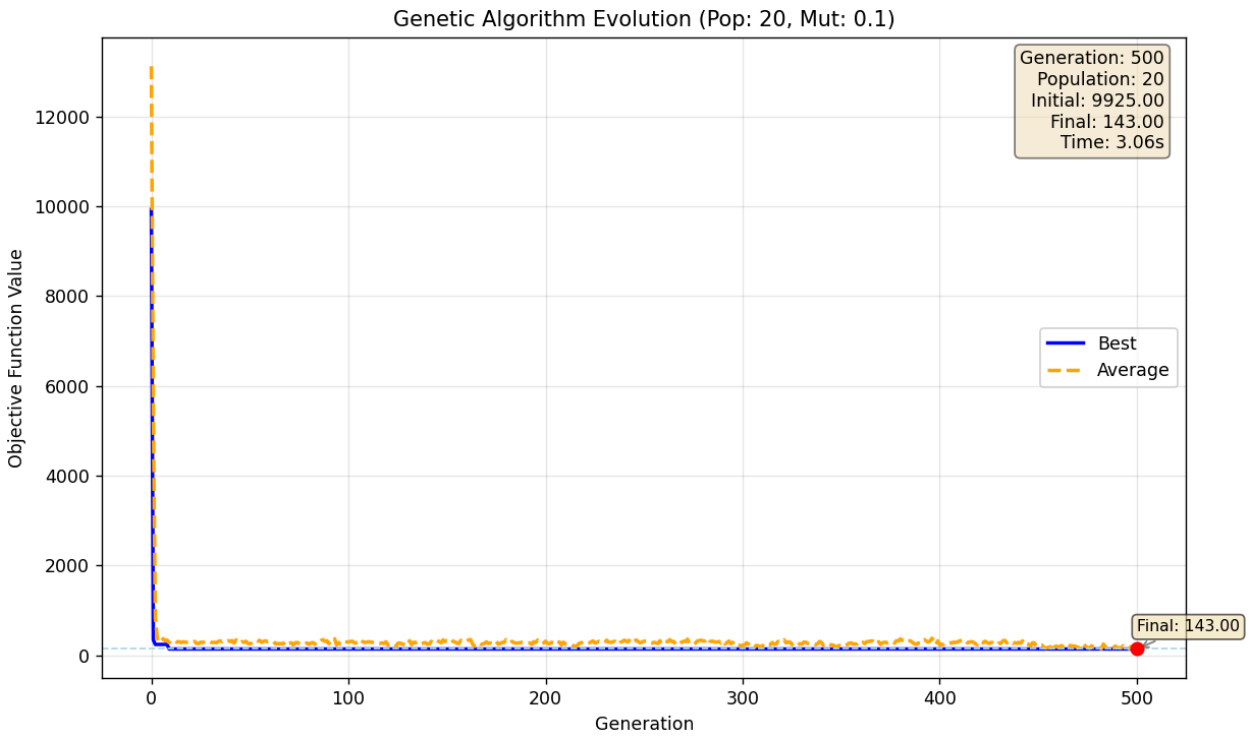
Gambar 3.21 Plot skor terbaik dan rata-rata tiap generasi terhadap iterasinya dengan populasi 100 dan iterasi 500



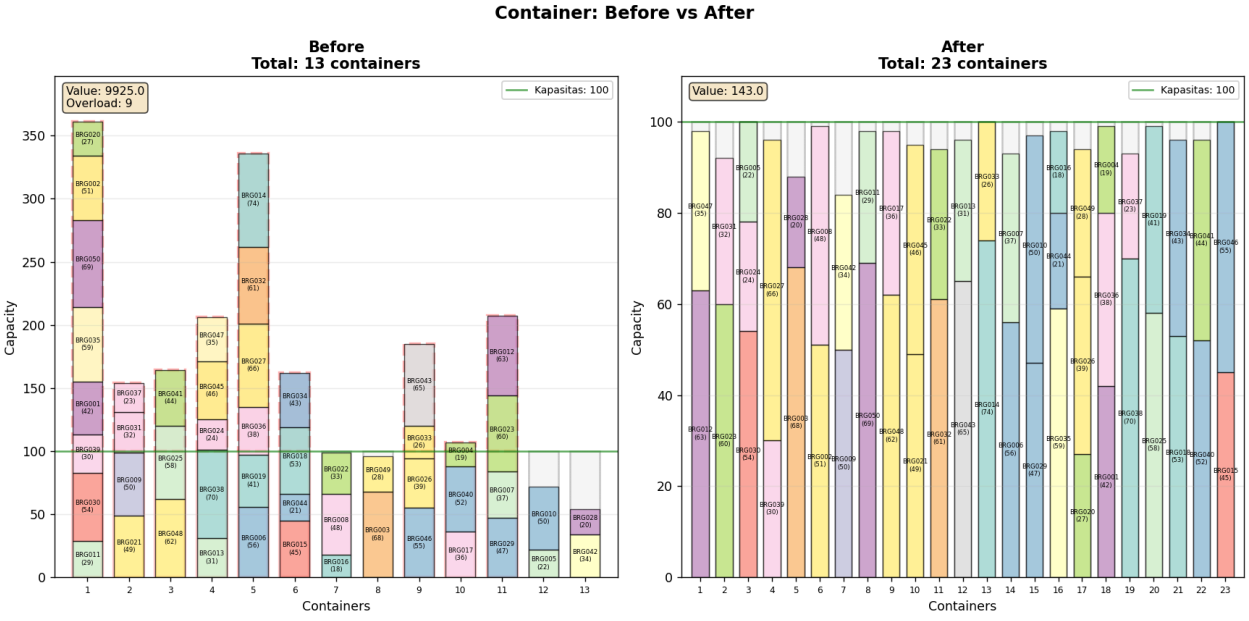
Gambar 3.24 Visualisasi perbardingan *state* awal dengan *state* akhir untuk populasi 100 dan iterasi 500

3.2.2 Variabel Kontrol: Iterasi

Test case 1
Populasi: 20
Iterasi: 500
Nilai objektif akhir: 143.00
Waktu: 3.06s



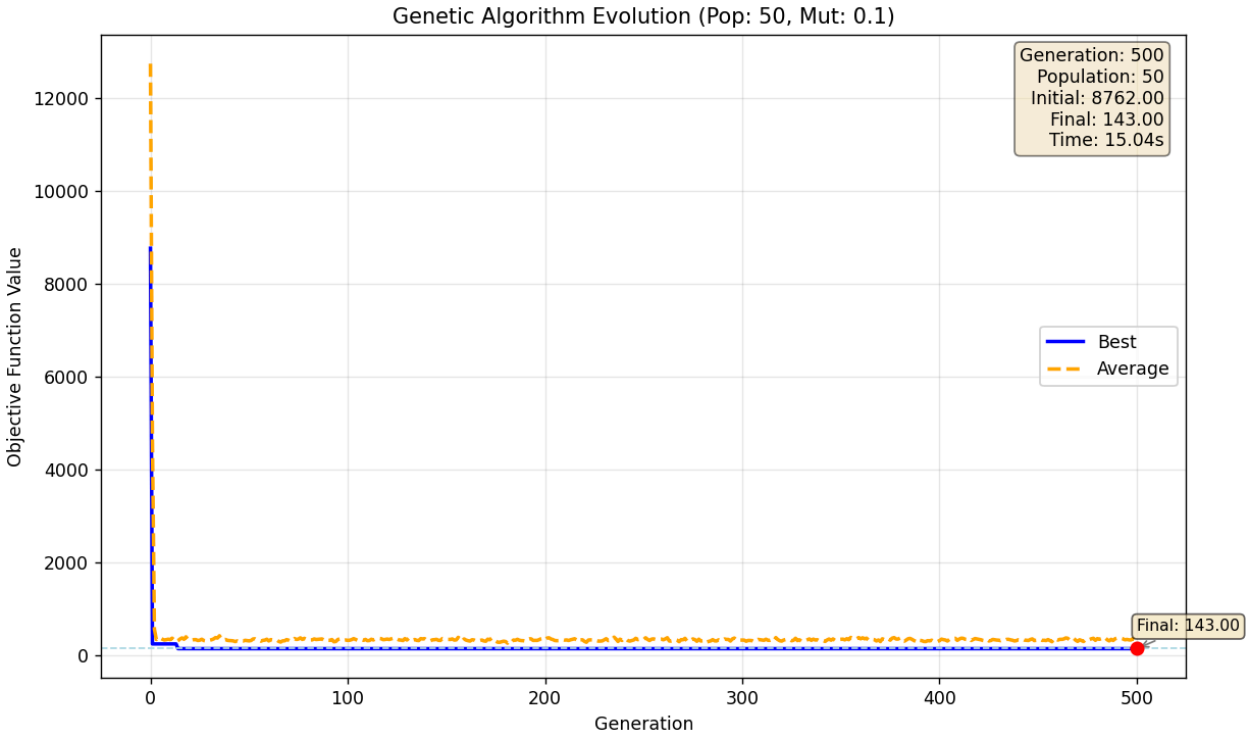
Gambar 3.25 Plot skor terbaik dan rata-rata tiap generasi terhadap iterasinya dengan populasi 20 dan iterasi 500



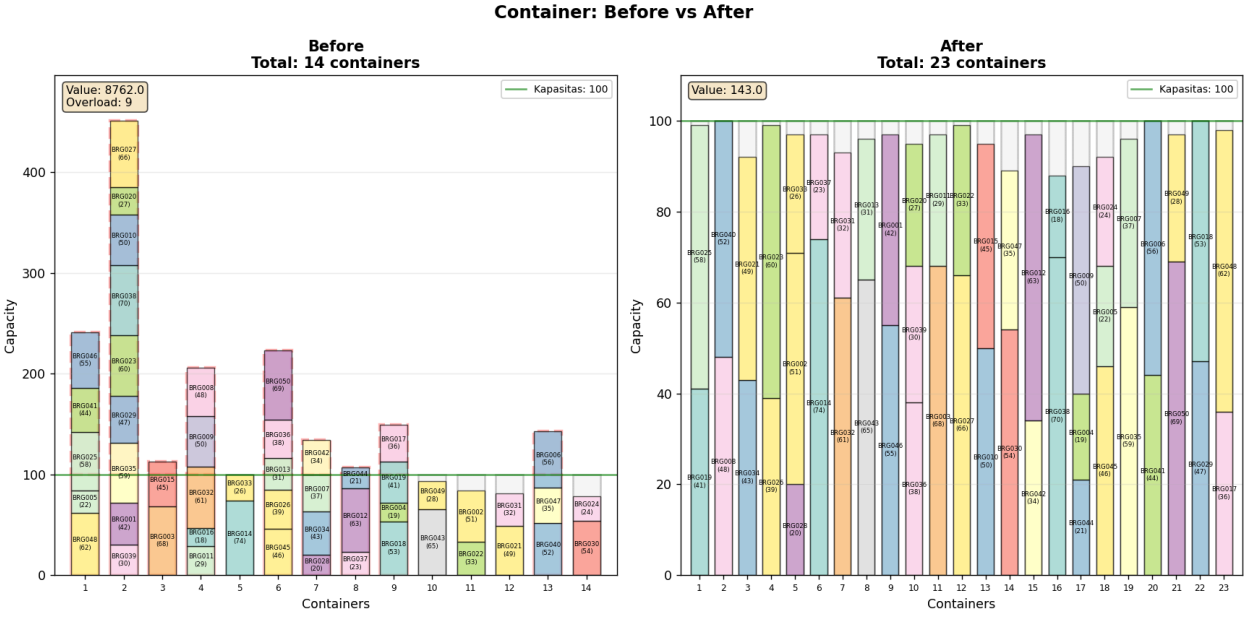
Gambar 3.24 Visualisasi perbardingan *state* awal dengan *state* akhir untuk populasi 20 dan iterasi 500

Populasi: 50
Iterasi: 500
Nilai objektif akhir: 143.00

Waktu: 15.04s



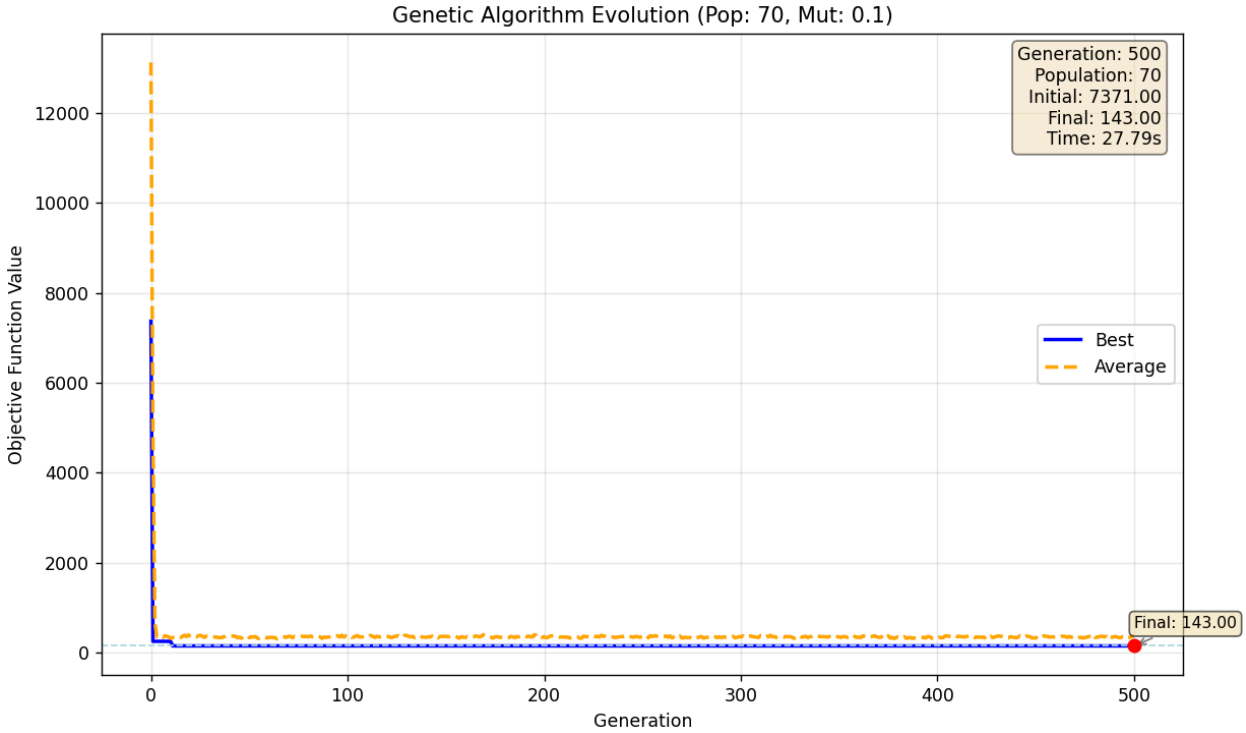
Gambar 3.21 Plot skor terbaik dan rata-rata tiap generasi terhadap iterasinya dengan populasi 50 dan iterasi 500



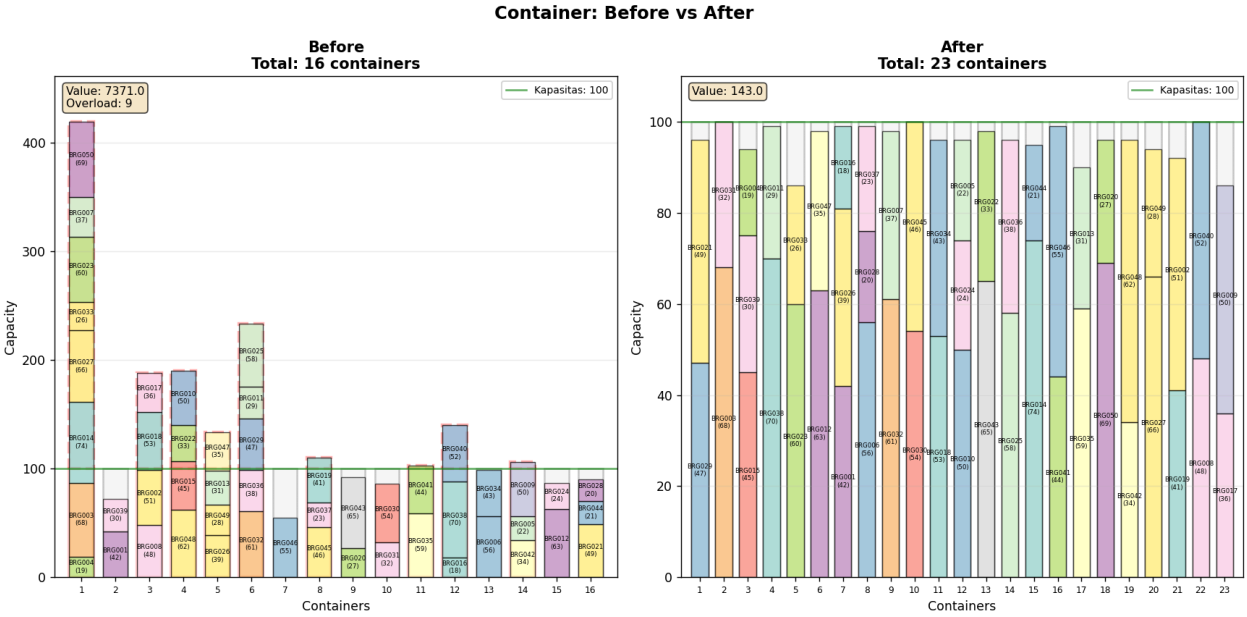
Gambar 3.24 Visualisasi perbardingan *state* awal dengan *state* akhir untuk populasi 500 dan iterasi 500

Populasi: 70
Iterasi: 500

Nilai objektif akhir: 143.00
Waktu: 27.79s



Gambar 3.27 Plot skor terbaik dan rata-rata tiap generasi terhadap iterasinya dengan populasi 70 dan iterasi 500



Gambar 3.24 Visualisasi perbardingan *state* awal dengan *state* akhir untuk populasi 70 dan iterasi 500

BAB IV
Analisis

Setelah melakukan eksperimen pada tiga algoritma local search menggunakan tiga test case yang berbeda, kita dapat menganalisis performa dari masing-masing algoritma. Analisis ini akan berfokus pada kualitas solusi akhir (nilai objective function), durasi eksekusi, dan konsistensi hasil.

Dari data hasil eksperimen, terlihat perbedaan kualitas solusi yang signifikan antara ketiga algoritma.

1. Hill Climbing dan Genetic Algorithm secara konsisten menghasilkan solusi dengan kualitas terbaik (skor akhir terendah). Pada Test Case 1, kedua algoritma ini bahkan mampu konvergen ke skor akhir yang identik, yaitu 143.00. Ini menunjukkan bahwa untuk masalah ini, keduanya sangat efektif dalam menemukan solusi yang sangat baik, yang kemungkinan besar adalah global optimum.
2. Simulated Annealing , di sisi lain, menunjukkan hasil yang jauh di bawah kedua algoritma lainnya. Pada Test Case 1, Simulated Annealing hanya mampu mencapai skor 245.00. Kesenjangan ini semakin terlihat pada Test Case 2 (Simulated Annealing: 3568.00, sementara Hill Climb: 735) dan Test Case 3 (Simulated Annealing: 1109.00, sementara Hill Climb: 629). Ini menunjukkan bahwa dengan parameter default yang kami gunakan, mekanisme probabilistik Simulated Annealing belum cukup tuning untuk bersaing dengan pendekatan Genetic Algorithm atau Hill Climb.

Semua algoritma terbukti mampu memperbaiki state awal yang sangat buruk (memiliki skor belasan ribu akibat overload) dan menghasilkan solusi akhir yang valid (atau setidaknya jauh lebih baik).

Perbandingan durasi eksekusi menunjukkan adanya perbedaan yang jelas dengan kualitas solusi:

1. Simulated Annealing adalah yang tercepat. Di semua test case, Simulated Annealing menyelesaikan pencarian dalam waktu kurang dari 1 detik. Ini wajar karena Simulated Annealing hanya mengevaluasi satu tetangga acak per iterasi.
2. Hill Climbing adalah yang paling lambat. Algoritma ini membutuhkan waktu puluhan detik (berkisar 46 hingga 77 detik). Penyebabnya adalah sifat exhaustive-nya, di mana ia harus mencari dan mengevaluasi semua kemungkinan tetangga (pindah dan tukar) di setiap iterasi sebelum memutuskan langkah.
3. Genetic Algorithm berada di posisi tengah. Waktunya sangat bergantung pada parameter yang dipilih, berkisar dari 3.06 detik (Populasi 20) hingga 54.63 detik (Populasi 100, Iterasi 500).

Ada beberapa hal yang didapat dari hasil eksperimen Genetic Algorithm pada Test Case 1.

- Konsistensi Genetic Algorithm: Algoritma ini terbukti sangat konsisten. Dari 6 eksperimen yang dilakukan (3 dengan variasi populasi dan 3 dengan variasi iterasi), semuanya berhasil menemukan skor akhir yang identik (143.00). Ini menunjukkan Genetic Algorithm sangat andal untuk masalah ini.

- Pengaruh Iterasi (Generasi): Saat populasi dijaga konstan (100), menambah jumlah iterasi dari 100 ke 300 atau 500 tidak memberikan peningkatan kualitas solusi (skor tetap 143.00). Ini menyiratkan bahwa algoritma sudah konvergen (menemukan solusi terbaik) hanya dalam 100 generasi. Menambah iterasi lebih lanjut hanya menambah waktu komputasi (dari 10.75 detik menjadi 54.63 detik) tanpa ada manfaat pada hasil akhir.
- Pengaruh Ukuran Populasi: Saat iterasi dijaga konstan (500), populasi yang lebih kecil (20 individu) ternyata sudah cukup untuk menemukan solusi optimal (skor 143.00) dengan waktu yang sangat cepat, yaitu 3.06 detik. Menambah populasi menjadi 50, 70, atau 100 juga menemukan skor yang sama , tetapi dengan waktu eksekusi yang jauh lebih lama (misalnya, 54.63 detik untuk 100 populasi) karena lebih banyak individu yang harus dievaluasi di setiap generasi

BAB V
Kesimpulan & Saran

5.1 Kesimpulan

Eksperimen ini telah berhasil mengimplementasikan dan membandingkan tiga algoritma *local search* dalam menyelesaikan *Bin Packing Problem*. Dari hasil pengujian pada berbagai *test case*, dapat disimpulkan bahwa terdapat perbedaan yang jelas antara kecepatan komputasi dan kualitas solusi akhir, yaitu:

1. Genetic Algorithm terbukti menjadi algoritma yang paling unggul secara keseluruhan. GA secara konsisten menghasilkan solusi dengan kualitas terbaik (skor akhir terendah, misal 143.00 pada Test Case 1), setara dengan Hill Climbing, namun dengan durasi eksekusi yang jauh lebih cepat (misal, 10.75 detik berbanding 77.45 detik). Pendekatan berbasis populasi dan operator *crossover* terbukti sangat efektif untuk menjelajahi *state space* secara efisien, menjadikannya pilihan terbaik untuk "hasil tepat dan waktu cepat".
2. Hill Climbing with Sideways Move juga terbukti sangat andal dalam menemukan solusi berkualitas tinggi ("hasil tepat"). Kemampuannya untuk mengeksplorasi semua tetangga di setiap iterasi dan fitur *sideways move* memastikannya tidak mudah terjebak dan mampu memperbaiki *state* awal yang sangat buruk (skor tinggi akibat *overload*) menjadi solusi akhir yang valid dan optimal secara lokal. Namun, keandalannya ini dibayar dengan waktu komputasi yang cukup lama, karena sifat pencariannya yang *exhaustive*.
3. Simulated Annealing, berdasarkan analisis, SA memberikan hasil yang cepat namun kurang efektif. Meskipun prosesnya secara teoritis cepat karena hanya mengevaluasi satu tetangga acak per iterasi, kelebihannya untuk menerima solusi yang lebih buruk secara probabilistik membuatnya kurang konsisten dan seringkali tidak konvergen ke solusi seoptimal Hill Climbing atau GA.

Secara keseluruhan, Genetic Algorithm memberikan keseimbangan terbaik antara kecepatan dan efektivitas untuk permasalahan *Bin Packing Problem* ini.

5.2 Saran

Berdasarkan pengerjaan tugas dan analisis yang telah dilakukan, terdapat saran:

1. Pengembangan Batasan Masalah: Sesuai dengan spesifikasi bonus, *objective function* dapat dikembangkan lebih lanjut untuk menangani batasan yang lebih kompleks, seperti barang yang tidak kompatibel (misal, 'makanan' dan 'kimia' tidak boleh dalam satu kontainer) atau barang rapuh.

Pembagian Tugas

NIM	Tugas
18223026	Membuat bagian Genetic, menulis laporan, membuat main dan visualisasi
18223028	Membuat format laporan, membuat laporan, membuat alur utama main, membuat laporan, Membuat algoritma hill-climbing, membuat setup awal system, Membuat repository github
18223072	Membuat code bagian algoritma Simulated Annealing (SA), menyusun dan mengisi laporan, membuat main dan visualisasi