

**Laporan Tugas 1**  
**IF3070 Dasar Inteligensi Artifisial**  
**Pencarian Solusi Pengepakan Barang (Bin Packing Problem)**  
**dengan Local Search**



Disusun Oleh:  
Kelompok 43

Fawwaz Aydin Mustofa	18222109
Daffa Athalla Rajasa	18223053
Adam Joaquin Girsang	18223089

**Sekolah Teknik Elektro dan Informatika**  
**Institut Teknologi Bandung**  
**2024**

## **DAFTAR ISI**

<b>DAFTAR ISI.....</b>	<b>2</b>
<b>I. DESKRIPSI PERSOALAN.....</b>	<b>3</b>
<b>II. PEMBAHASAN.....</b>	<b>3</b>
<b>III. KESIMPULAN DAN SARAN.....</b>	<b>3</b>
<b>IV. PEMBAGIAN TUGAS.....</b>	<b>3</b>
<b>V. REFERENSI.....</b>	<b>3</b>

## I. DESKRIPSI PERSOALAN

Pada tugas besar ini, kami diminta untuk menyelesaikan permasalahan *Bin Packing Problem* menggunakan pendekatan *local search*. Masalah ini secara sederhana bisa dijelaskan sebagai proses menempatkan sejumlah barang dengan ukuran berbeda ke dalam beberapa kontainer (atau wadah) yang kapasitasnya sama. Tujuannya adalah agar semua barang bisa masuk tanpa melebihi kapasitas, dan jumlah kontainer yang digunakan sesedikit mungkin.

Setiap barang memiliki atribut berupa ID barang dan ukuran (misalnya dalam satuan kilogram atau meter kubik). Sedangkan setiap kontainer memiliki kapasitas total yang seragam. Dalam representasi program, satu solusi (atau state) digambarkan sebagai daftar berisi beberapa daftar lain, di mana setiap daftar internal mewakili satu kontainer dan berisi ID barang yang dimasukkan ke dalamnya, seperti `[['BRG001', 'BRG005'], ['BRG002', 'BRG003'], ['BRG004']]` yang berarti ada tiga kontainer digunakan, masing-masing berisi kombinasi barang yang berbeda.

Selama proses pencarian solusi, algoritma akan mencoba melakukan perubahan kecil (moves) pada susunan barang, misalnya dengan:

1. Memindahkan satu barang dari satu kontainer ke kontainer lain.
2. Menukar dua barang dari dua kontainer yang berbeda.

Dengan langkah-langkah tersebut, algoritma berusaha memperbaiki solusi dari waktu ke waktu hingga menemukan hasil yang paling efisien, yaitu jumlah kontainer paling sedikit tanpa ada pelanggaran kapasitas. Untuk menilai seberapa baik suatu solusi, digunakan fungsi objektif (*objective function*) yang mempertimbangkan tiga hal utama:

- Memberi penalti besar jika ada kontainer yang kelebihan kapasitas (agar solusi tetap valid),
- Menghitung jumlah kontainer yang digunakan (semakin sedikit semakin baik),
- Mempertimbangkan kepadatan isi tiap kontainer supaya pemakaian ruang lebih efisien.

## II. PEMBAHASAN

### A. Pemilihan Objective Function

Fungsi objektif utama proyek ini diimplementasikan dalam file objective\_function.py melalui fungsi calculate\_objective(state, kapasitas, barang), yang mengembalikan nilai numerik untuk menilai seberapa baik sebuah solusi (semakin kecil nilainya, semakin baik solusinya).

```
def calculate_objective(state: List[List[str]], kapasitas: int, barang: Dict[str, int]) -> float:
    """
    Menghitung nilai fungsi objektif (SEMAKIN RENDAH SEMAKIN BAIK)

    Komponen:
    1. Penalti untuk kontainer yang melebihi kapasitas (KRITIS - harus valid)
    2. Jumlah kontainer yang digunakan (objektif utama)
    3. Penalti ruang terbuang (mendorong kontainer lebih penuh)

    Args:
        state: State saat ini (daftar kontainer)
        kapasitas: Kapasitas kontainer
        barang: Dictionary dari id_barang -> ukuran

    Returns:
        Nilai objektif (semakin rendah semakin baik)
    """

    if len(state) == 0:
        return float('inf')

    score = 0.0

    # 1. PENALTI OVERFLOW (sangat besar)
    # Ini memastikan solusi akhir VALID
    overflow_penalty = 0
    for bin_items in state:
        bin_size = sum(barang[item] for item in bin_items)
        if bin_size > kapasitas:
            overflow = bin_size - kapasitas
            overflow_penalty += 10000 * overflow # PENALTI SANGAT BESAR

    score += overflow_penalty

    # 2. JUMLAH KONTAINER (objektif utama)
    # Semakin sedikit kontainer, semakin baik
    num_bins = len(state)
    score += num_bins * 100

    # 3. PENALTI RUANG TERBUANG
    # Mendorong kontainer yang lebih penuh
    wasted_space = 0
    for bin_items in state:
        bin_size = sum(barang[item] for item in bin_items)
        if bin_size <= kapasitas:
            wasted = kapasitas - bin_size
            wasted_space += wasted

    # Penalti kecil untuk ruang terbuang
    score += wasted_space * 0.1

    return score
```

Fungsi ini menilai solusi berdasarkan tiga komponen utama:

- **Penalti overflow** – Jika total ukuran barang dalam suatu bin melebihi kapasitas, maka diberikan penalti sebesar  $10000 \times (\text{kelebihan})$  untuk setiap bin. Komponen ini memiliki bobot paling besar agar solusi yang melanggar kapasitas langsung dianggap buruk.

- **Jumlah bin yang digunakan** – Setiap bin aktif menambah penalti  $100 \times \text{jumlah\_bin}$ , mendorong algoritma untuk mencari solusi dengan jumlah kontainer sesedikit mungkin.
- **Ruang terbuang (wasted space)** – Untuk setiap bin yang valid (tidak overflow), dihitung sisa kapasitasnya dan dijumlahkan sebagai penalti ringan  $0.1 \times \text{total\_ruang\_terbuang}$ , agar solusi yang lebih efisien diutamakan.

Selain itu, terdapat fungsi calculate\_fitness untuk digunakan oleh Genetic Algorithm, yang mengubah nilai objektif menjadi nilai kebugaran (fitness).

```
def calculate_fitness(state: List[List[str]], kapasitas: int, barang: Dict[str, int]) -> float:
    """
    Menghitung fitness untuk Algoritma Genetika (SEMAKIN TINGGI SEMAKIN BAIK)
    Hanya kebalikan dari fungsi objektif
    """
    obj_value = calculate_objective(state, kapasitas, barang)

    # Hindari pembagian dengan nol
    if obj_value == 0:
        return float('inf')

    return 1.0 / obj_value
```

Semakin kecil nilai objektif, semakin besar nilai *fitness*-nya, sehingga solusi lebih baik memiliki peluang seleksi yang lebih tinggi.

## B. Implementasi Algoritma Local Search

### A. Hill Climbing

Implementasi terdiri atas beberapa varian:

- Steepest Ascent Hill Climbing
- Stochastic Hill Climbing
- Sideways Move Hill Climbing
- Random Restart Hill Climbing

## A.1. Steepest Ascent Hill Climbing

```
def steepest_ascent_hill_climbing(bp: BinPacking, initial_state: List[List[str]], max_iterations: int = 1000) -> Tuple[List[List[str]], float, List[float], int]:
    """
    Steepest Ascent Hill Climbing
    Selalu pilih tetangga TERBAIK

    Returns:
        best_state, best_score, history, iterations
    """
    current_state = copy.deepcopy(initial_state)
    current_score = calculate_objective(current_state, bp.kapasitas, bp.barang)

    history = [current_score]
    iteration = 0

    while iteration < max_iterations:
        # dapatin semua tetangga
        neighbors = bp.get_neighbors(current_state)

        if len(neighbors) == 0:
            # ga ada tetangga, catat skor dan berhenti
            history.append(current_score)
            break

        # cari tetangga terbaik
        best_neighbor = None
        best_neighbor_score = current_score

        for neighbor in neighbors:
            score = calculate_objective(neighbor, bp.kapasitas, bp.barang)
            if score < best_neighbor_score: # Minimisasi
                best_neighbor_score = score
                best_neighbor = neighbor

        # kalo ga ada improvement, catat skor dan berhenti
        if best_neighbor is None:
            history.append(current_score)
            break

        current_state = best_neighbor
        current_score = best_neighbor_score
        history.append(current_score)
        iteration += 1

    return current_state, current_score, history, iteration
```

**Tujuan:** selalu memilih **tetangga terbaik** (nilai objektif minimum) setiap iterasi.

**Langkah utama:**

1. Salin initial\_state ke current state; hitung current\_score.
2. Dapatkan semua tetangga dengan neighbors = bp.get\_neighbors(current\_state)
- 3.

```
for neighbor in neighbors:
    score = calculate_objective(neighbor, bp.kapasitas, bp.barang)
    if score < best_neighbor_score: # Minimisasi
        best_neighbor_score = score
        best_neighbor = neighbor
```

Hitung nilai tiap tetangga dan pilih yg nilai yg paling rendah

- 4.

```
if best_neighbor is None:
    history.append(current_score)
    break
```

Jika tidak ada tetangga yg memperbaiki maka berhenti

5.

```
    current_state = best_neighbor
    current_score = best_neighbor_score
    history.append(current_score)
    iteration += 1

return current_state, current_score, history, iteration
```

Jika ada perbaikan maka update current\_state & current\_score lalu lanjut. Mengembalikan nilai current\_state yg merupakan state terbaik, score terbaik, history (jejak skor setiap langkah), dan iteration (jumlah iterasi)

## A.2. Stochastic Hill Climbing

```
def stochastic_hill_climbing(bp: BinPacking, initial_state: List[List[str]], max_iterations: int = 1000) -> Tuple[List[List[str]], float, List[float], int]:
    """
    Stochastic Hill Climbing
    Pilih tetangga yang lebih baik secara RANDOM

    Returns:
        best_state, best_score, history, iterations
    """
    current_state = copy.deepcopy(initial_state)
    current_score = calculate_objective(current_state, bp.kapasitas, bp.barang)

    history = [current_score]
    iteration = 0

    while iteration < max_iterations:
        # dapatin semua tetangga
        neighbors = bp.get_neighbors(current_state)

        if len(neighbors) == 0:
            # gaada tetangga, catat skor dan berhenti
            history.append(current_score)
            break

        # cari semua tetangga yang Lebih baik
        better_neighbors = []
        for neighbor in neighbors:
            score = calculate_objective(neighbor, bp.kapasitas, bp.barang)
            if score < current_score:
                better_neighbors.append((neighbor, score))

        # kalo ga ada tetangga yang Lebih baik, catat skor dan berhenti
        if len(better_neighbors) == 0:
            history.append(current_score)
            break

        # pilih tetangga yang Lebih baik secara random
        current_state, current_score = random.choice(better_neighbors)
        history.append(current_score)
        iteration += 1

    return current_state, current_score, history, iteration
```

**Tujuan:** dari semua tetangga yang **lebih baik** dari current, **pilih satu secara acak**.

**Langkah utama:**

1.

```
# cari semua tetangga yang Lebih baik
better_neighbors = []
for neighbor in neighbors:
    score = calculate_objective(neighbor, bp.kapasitas, bp.barang)
    if score < current_score:
        better_neighbors.append((neighbor, score))
```

Kumpulkan semua tetangga yang lebih baik.

2.

```
# kalo ga ada tetangga yang Lebih baik, catat skor dan berhenti
if len(better_neighbors) == 0:
    history.append(current_score)
    break
```

Jika tidak ada yang lebih baik, berhenti

3.

```
# pilih tetangga yang lebih baik secara random
current_state, current_score = random.choice(better_neighbors)
history.append(current_score)
iteration += 1

return current_state, current_score, history, iteration
```

Jika ada yang lebih baik, pilih secara *random* dari tetangga - tetangga yang sudah dikumpulkan

### A.3. Sideways Move Hill Climbing

```
def sideways_move_hill_climbing(bp: BinPacking, initial_state: List[List[str]], max_iterations: int = 1000, max_sideways: int = 100) -> Tuple[List[List[str]], float, List[float], int]:
"""
Hill Climbing with Sideways Move
Izinin perpindahan ke tetangga dengan skor SAMA (maksimal max_sideways kali)

Returns:
    best_state, best_score, history, iterations
"""
current_state = copy.deepcopy(initial_state)
current_score = calculate_objective(current_state, bp.kapasitas, bp.barang)

history = [current_score]
iteration = 0
sideways_count = 0

while iteration < max_iterations and sideways_count < max_sideways:
    neighbors = bp.get_neighbors(current_state)

    if len(neighbors) == 0:
        # ga ada tetangga, catat skor dan berhenti
        history.append(current_score)
        break

    best_neighbor = None
    best_neighbor_score = current_score

    for neighbor in neighbors:
        score = calculate_objective(neighbor, bp.kapasitas, bp.barang)
        if score < best_neighbor_score:
            best_neighbor_score = score
            best_neighbor = neighbor

    # kalo ga ada improvement
    if best_neighbor is None:  daffathall, 4 hours ago * feat(hill_climbing): implementasi empat varian ...
        # coba sideways move
        sideways_neighbors = [n for n in neighbors
                             if calculate_objective(n, bp.kapasitas, bp.barang) == current_score]

        if len(sideways_neighbors) > 0:
            best_neighbor = random.choice(sideways_neighbors)
            best_neighbor_score = current_score
            sideways_count += 1
        else:
            # ga ada improvement dan sideways move, catat dan berhenti
            history.append(current_score)
            break
    else:
        sideways_count = 0 # reset kalo ada improvement

    current_state = best_neighbor
    current_score = best_neighbor_score
    history.append(current_score)
    iteration += 1

return current_state, current_score, history, iteration
```

**Tujuan:** mengizinkan perpindahan datar (sideways) ke tetangga dengan skor sama saat tidak ada perbaikan, maksimal max\_sideways kali berturut-turut.

**Langkah utama:**

1. Coba cari perbaikan, jika ada maka gunakan itu.
- 2.

```
# kalo ga ada improvement
if best_neighbor is None:
    # coba sideways move
    sideways_neighbors = [n for n in neighbors
                           if calculate_objective(n, bp.kapasitas, bp.barang) == current_score]

    if len(sideways_neighbors) > 0:
        best_neighbor = random.choice(sideways_neighbors)
        best_neighbor_score = current_score
        sideways_count += 1
```

Jika tidak ada perbaikan, kumpulkan tetangga dengan skor sama; bila ada, pilih acak dan sideways\_count tambah 1.

- 3.

```
else:
    # ga ada improvement dan sideways move, catat dan berhenti
    history.append(current_score)
    break
```

Jika tidak ada perbaikan dan tidak ada sideways move; berhenti.

4. Kembalikan nilai state, skor, history (jejak skor), iterasi.

#### A.4. Random Restart Hill Climbing

```
def random_restart_hill_climbing(bp: BinPacking, max_restarts: int = 10, max_iterations_per_restart: int = 100) -> Tuple[List[List[str]], float, List[float], int, List[int]]:
    """
    Random Restart Hill Climbing
    Jalankan steepest ascent berkali-kali dengan initial state berbeda

    Returns:
        best_state, best_score, history, total_iterations, iterations_per_restart
    """
    global_best_state = None
    global_best_score = float('inf')
    global_history = []
    total_iterations = 0
    iterations_per_restart = []

    for restart in range(max_restarts):
        # generate state awal random baru
        initial_state = bp.initial_state_random()

        # jalankan steepest ascent
        state, score, history, iterations = steepest_ascent_hill_climbing(bp, initial_state, max_iterations_per_restart)

        iterations_per_restart.append(iterations)
        total_iterations += iterations
        global_history.extend(history)

        # update global best
        if score < global_best_score:
            global_best_score = score
            global_best_state = state

    return global_best_state, global_best_score, global_history, total_iterations, iterations_per_restart
```

**Tujuan:** menjalankan steepest-ascent berulang kali dari *initial state* acak yang berbeda untuk menghindari local optimum.

**Langkah utama:**

1.

```
for restart in range(max_restarts):
    # generate state awal random baru
    initial_state = bp.initial_state_random()
```

Loop sebanyak max\_restarts dan generate initial state awal random

2.

```
# jalankan steepest ascent
state, score, history, iterations = steepest_ascent_hill_climbing(bp, initial_state, max_iterations_per_restart)
```

Jalankan steepest ascent hill climbing selama max\_iterations\_per\_restart

3.

```
# update global best
if score < global_best_score:
    global_best_score = score
    global_best_state = state

return global_best_state, global_best_score, global_history, total_iterations, iterations_per_restart
```

Update global best di seluruh restart dan kembalikan nilainya.

## B. Simulated Annealing

Algoritma ini terdiri dari 2 varian yaitu: Simulated Annealing standar, dan Simulated Annealing dengan reheating.

### 1. Simulated Annealing standar

```
def simulated_annealing(
    bp: BinPacking,
    initial_state: List[List[str]],
    T_initial: float = 1000.0,
    T_min: float = 0.1,
    alpha: float = 0.95,
    max_iterations: int = 1000
) -> Tuple[List[List[str]], float, List[float], List[float], int]:
    """
    Algoritma: Simulated Annealing

    Args:
        bp: Instance BinPacking
        initial_state: State awal
        T_initial: Temperatur awal
        T_min: Temperatur minimum (kondisi berhenti)
        alpha: Laju pendinginan ( $0 < \alpha < 1$ )
        max_iterations: Iterasi maksimum

    Returns:
        best_state, best_score, score_history, probability_history, stuck_count
    """
    current_state = copy.deepcopy(initial_state)
    current_score = calculate_objective(current_state, bp.kapasitas, bp.barang)

    best_state = copy.deepcopy(current_state)
    best_score = current_score

    T = T_initial
    score_history = [current_score]
    probability_history = [] # Untuk plotting e^(-ΔE/T)
    stuck_count = 0
    iteration = 0

    while T > T_min and iteration < max_iterations:
        # Dapatkan tetangga random
        neighbor = bp.get_random_neighbor(current_state)
        neighbor_score = calculate_objective(neighbor, bp.kapasitas, bp.barang)

        # Hitung delta E nya
        delta_E = neighbor_score - current_score

        # Putuskan apakah menerima tetangga
        if delta_E < 0:
            # Solusi lebih baik - selalu terima
            current_state = neighbor
            current_score = neighbor_score

            # Perbarui solusi terbaik
            if current_score < best_score:
                best_state = copy.deepcopy(current_state)
                best_score = current_score

        probability_history.append(1.0)
        else:
            # Solusi lebih buruk - terima dengan probabilitas
            probability = math.exp(-delta_E / T)
            probability_history.append(probability)

            if random.random() < probability:
                current_state = neighbor
                current_score = neighbor_score
            else:
                stuck_count += 1

        # Catat skor saat ini setelah keputusan
        score_history.append(current_score)

        # Turunkan temperatur
        T *= alpha
        iteration += 1

    return best_state, best_score, score_history, probability_history, stuck_count
```

**Tujuan:** menyeimbangkan eksplorasi (suhu tinggi) dan eksploitasi (suhu rendah) agar bisa keluar dari local optimum.

**Langkah utama:**

- A. Inisialisasi state, skor, dan T
- B.

```
# Dapatkan tetangga random
neighbor = bp.get_random_neighbor(current_state)
neighbor_score = calculate_objective(neighbor, bp.kapasitas, bp.barang)

# Hitung delta_E nya
delta_E = neighbor_score - current_score
```

Ambil tetangga acak dan hitung delta\_E nya.

- C.

```
# Putuskan apakah menerima tetangga
if delta_E < 0:
    # Solusi lebih baik - selalu terima
    current_state = neighbor
    current_score = neighbor_score

    # Perbarui solusi terbaik
    if current_score < best_score:
        best_state = copy.deepcopy(current_state)
        best_score = current_score

    probability_history.append(1.0)
```

Jika  $\Delta E < 0$  maka selalu terima, perbarui best bila perlu, catat prob 1.0.

- D.

```
else:
    # Solusi lebih buruk - terima dengan probabilitas
    probability = math.exp(-delta_E / T)
    probability_history.append(probability)

    if random.random() < probability:
        current_state = neighbor
        current_score = neighbor_score
    else:
        stuck_count += 1
```

Jika  $\Delta E \geq 0$ , terima dengan probabilitas  $e^{-\Delta E/T}$ ; jika tidak diterima naikkan stuck\_count.

- E. Simpan current\_score ke score\_history, turunkan T.
- F. Ulangi selama  $T > T_{min}$  dan  $iteration < max\_iterations$
- G. Kembalikan nilai terbaik yg pernah dijumpai

## 2. Simulated Annealing dengan reheating

**Tujuan:** menambahkan **pemanasan ulang** (reheating) saat *stuck* terlalu lama, agar SA bisa “naik” lagi lalu eksplorasi ulang.

**Tambahan:**

- Menghitung jika tidak ada perbaikan
- 

```
# Pemanasan ulang kalau stuck
if no_improvement_count >= reheat_threshold:
    T = min(T * reheat_factor, T_initial)
    no_improvement_count = 0
```

Jika jumlah tidak ada perbaikan  $\geq$  reheat\_threshold maka T dinaikkan dan hitung ulang jumlah tidak ada perbaikan

## C. Genetic Algorithm

Mencari solusi global via evolusi populasi. Algoritma ini memiliki beberapa fungsi yang digunakan untuk reproduksi/membuat keturunan. Fungsi-fungsinya adalah sebagai berikut:



```
def tournament_selection(population: List[List[str]], fitness_scores: List[float], tournament_size: int = 3) -> List[List[str]]:
    """
    Seleksi Turnamen
    Pilih individu terbaik dari subset acak
    """
    tournament_indices = random.sample(range(len(population)), tournament_size)
    best_idx = max(tournament_indices, key=lambda i: fitness_scores[i])
    return copy.deepcopy(population[best_idx])
```

Seleksi orang tua: memilih fitness tertinggi diantara 3 individu acak



```
def crossover(parent1: List[List[str]], parent2: List[List[str]], bp: BinPacking) -> Tuple[List[List[str]], List[List[str]]]:
    """
    One-Point Crossover untuk Bin Packing
    Pisahkan kontainer dan gabungkan
    """
    if len(parent1) == 0 or len(parent2) == 0:
        return copy.deepcopy(parent1), copy.deepcopy(parent2)

    # One-point crossover
    cut1 = random.randint(0, len(parent1))
    cut2 = random.randint(0, len(parent2))

    # Buat anak
    child1_bins = parent1[:cut1] + parent2[cut2:]
    child2_bins = parent2[:cut2] + parent1[cut1:]

    # Perbaiki: pastikan semua item ada tepat satu kali
    child1 = repair_solution(child1_bins, bp)
    child2 = repair_solution(child2_bins, bp)

    return child1, child2
```

Crossover: gabungkan bagian depan parent1 dengan bagian belakang parent2 (dan sebaliknya) untuk membentuk child1 dan child2. Lalu pastikan semua item hanya muncul satu kali dan tidak ada duplikat.

```

def mutate(individual: List[List[str]], bp: BinPacking) -> List[List[str]]:
    """
    Mutasi: pindahkan atau tukar item secara acak
    """
    mutated = copy.deepcopy(individual)

    if len(mutated) == 0:
        return mutated

    # Pilih tipe mutasi
    mutation_type = random.choice(['move', 'swap'])

    if mutation_type == 'move' and len(mutated) > 0:
        # Pindahkan item acak ke kontainer acak
        source_bin_idx = random.randint(0, len(mutated) - 1)
        if len(mutated[source_bin_idx]) > 0:
            item = random.choice(mutated[source_bin_idx])
            mutated[source_bin_idx].remove(item)

            # Tambahkan ke kontainer acak atau kontainer baru
            if random.random() < 0.5 and len(mutated) > 1:
                dest_bin_idx = random.randint(0, len(mutated) - 1)
                mutated[dest_bin_idx].append(item)
            else:
                mutated.append([item])

    elif mutation_type == 'swap' and len(mutated) >= 2:
        # Tukar item antara dua kontainer acak
        bin1_idx = random.randint(0, len(mutated) - 1)
        bin2_idx = random.randint(0, len(mutated) - 1)

        if bin1_idx != bin2_idx and len(mutated[bin1_idx]) > 0 and len(mutated[bin2_idx]) > 0:
            item1 = random.choice(mutated[bin1_idx])
            item2 = random.choice(mutated[bin2_idx])

            mutated[bin1_idx].remove(item1)
            mutated[bin1_idx].append(item2)
            mutated[bin2_idx].remove(item2)
            mutated[bin2_idx].append(item1)

        # Hapus kontainer kosong
        mutated = [b for b in mutated if len(b) > 0]

    # Pastikan valid
    if not bp.is_valid(mutated):
        mutated = repair_solution(mutated, bp)

    return mutated

```

Mutasi: memilih secara acak tipe mutasinya. Jika tipe move, pilih sumber acak, ambil item acak, lalu pindahkan ke bin lain acak (atau buat bin baru). Jika tipe swap, tukar dua item acak dari dua bin berbeda. Setelah mutasi, bersihkan bin kosong dan pastikan apakah valid atau tidak.

## Alur per generasi:

1.

```
for individual in population:
    obj_score = calculate_objective(individual, bp.kapasitas, bp.barang)
    fit_score = calculate_fitness(individual, bp.kapasitas, bp.barang)
    objective_scores.append(obj_score)
    fitness_scores.append(fit_score)

# Lacak statistik
best_history.append(min(objective_scores))
avg_history.append(sum(objective_scores) / len(objective_scores))
```

Hitung nilai objektif dan fitness pada tiap individu di populasi. Simpan juga best\_history (min objective) dan avg\_history (rata-rata objective).

2.

```
# Elitisme: pertahankan individu terbaik
if elitism > 0:
    elite_indices = sorted(range(len(objective_scores)),
                           key=lambda i: objective_scores[i])[:elitism]
    for idx in elite_indices:
        new_population.append(copy.deepcopy(population[idx]))
```

Elitisme: ambil indeks elitism individu dengan objective terendah, deepcopy masuk new\_population untuk mempertahankan individu terbaik.

3.

Reproduksi dengan menggunakan seleksi, crossover, dan mutasi. Lakukan sampai ukuran populasi terpenuhi dan update populasi.

4.

```
# Kembalikan individu terbaik
final_scores = [calculate_objective(ind, bp.kapasitas, bp.barang) for ind in population]
best_idx = final_scores.index(min(final_scores))
best_state = population[best_idx]
best_score = final_scores[best_idx]

return best_state, best_score, best_history, avg_history
```

Kembalikan individu terbaik dari populasi terakhir berdasarkan objective terendah.

## C. Hasil eksperimen dan analisis

Eksperimen dilakukan untuk membandingkan performa tiga algoritma pencarian lokal yaitu Hill Climbing, Simulated Annealing (SA), dan Genetic Algorithm (GA). Pada kasus Bin Packing Problem dengan delapan item (BRG001–BRG008) dan kapasitas setiap kontainer sebesar 100. Semua algoritma diuji menggunakan fungsi objektif yang sama, yaitu calculate\_objective() yang menimbang penalti overflow, jumlah bin, dan ruang terbuang. Setiap algoritma diuji 3 kali

### 1. Hill Climbing

Empat varian diuji: Steepest Ascent, Stochastic, Sideways Move, dan Random Restart.

```
[1] HILL CLIMBING ALGORITHMS
[1a] Running Steepest Ascent Hill Climbing...
Initial bins: 8, Score: 847.50
Final bins: 4, Score: 407.50
Iterations: 4, Time: 0.006s

[1b] Running Stochastic Hill Climbing...
Initial bins: 8, Score: 847.50
Final bins: 4, Score: 407.50
Iterations: 4, Time: 0.007s

[1c] Running Sideways Move Hill Climbing...
Initial bins: 8, Score: 847.50
Final bins: 4, Score: 407.50
Iterations: 104, Time: 0.100s

[1d] Running Random Restart Hill Climbing...
Jumlah restart: 10
Final bins: 4, Score: 407.50
Total iterations: 0, Time: 0.008s

>> Best Hill Climbing: Steepest Ascent
Bins: 4, Score: 407.50

[1] HILL CLIMBING ALGORITHMS
[1a] Running Steepest Ascent Hill Climbing...
Initial bins: 8, Score: 847.50
Final bins: 4, Score: 407.50
Iterations: 4, Time: 0.006s

[1b] Running Stochastic Hill Climbing...
Initial bins: 8, Score: 847.50
Final bins: 5, Score: 517.50
Iterations: 3, Time: 0.007s

[1c] Running Sideways Move Hill Climbing...
Initial bins: 8, Score: 847.50
Final bins: 4, Score: 407.50
Iterations: 104, Time: 0.091s

[1d] Running Random Restart Hill Climbing...
Jumlah restart: 10
Final bins: 4, Score: 407.50
Total iterations: 0, Time: 0.009s

>> Best Hill Climbing: Steepest Ascent
Bins: 4, Score: 407.50

[1] HILL CLIMBING ALGORITHMS
[1a] Running Steepest Ascent Hill Climbing...
Initial bins: 8, Score: 847.50
Final bins: 4, Score: 407.50
Iterations: 4, Time: 0.007s

[1b] Running Stochastic Hill Climbing...
Initial bins: 8, Score: 847.50
Final bins: 4, Score: 407.50
Iterations: 4, Time: 0.007s

[1c] Running Sideways Move Hill Climbing...
Initial bins: 8, Score: 847.50
Final bins: 4, Score: 407.50
Iterations: 105, Time: 0.097s

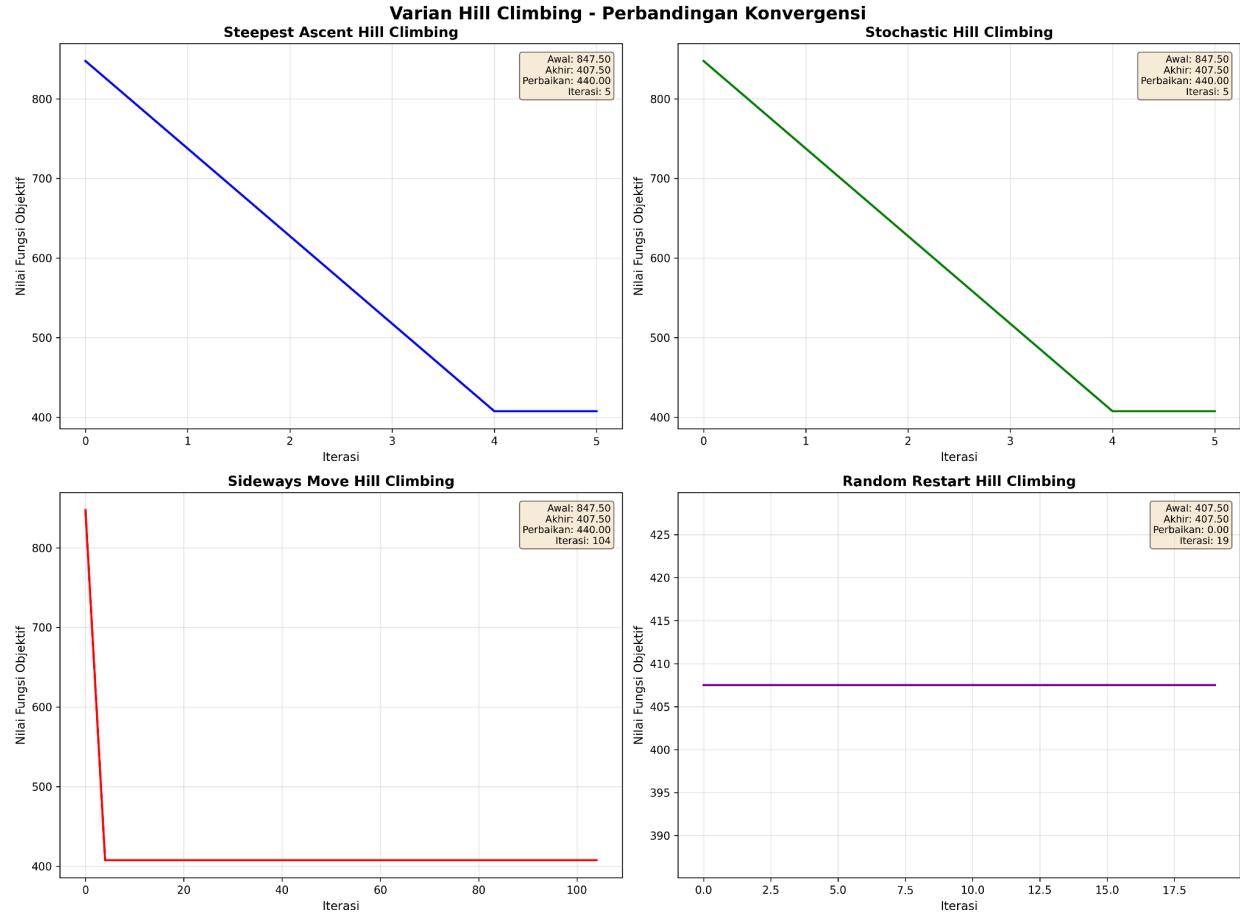
[1d] Running Random Restart Hill Climbing...
Jumlah restart: 10
Final bins: 4, Score: 407.50
Total iterations: 0, Time: 0.007s

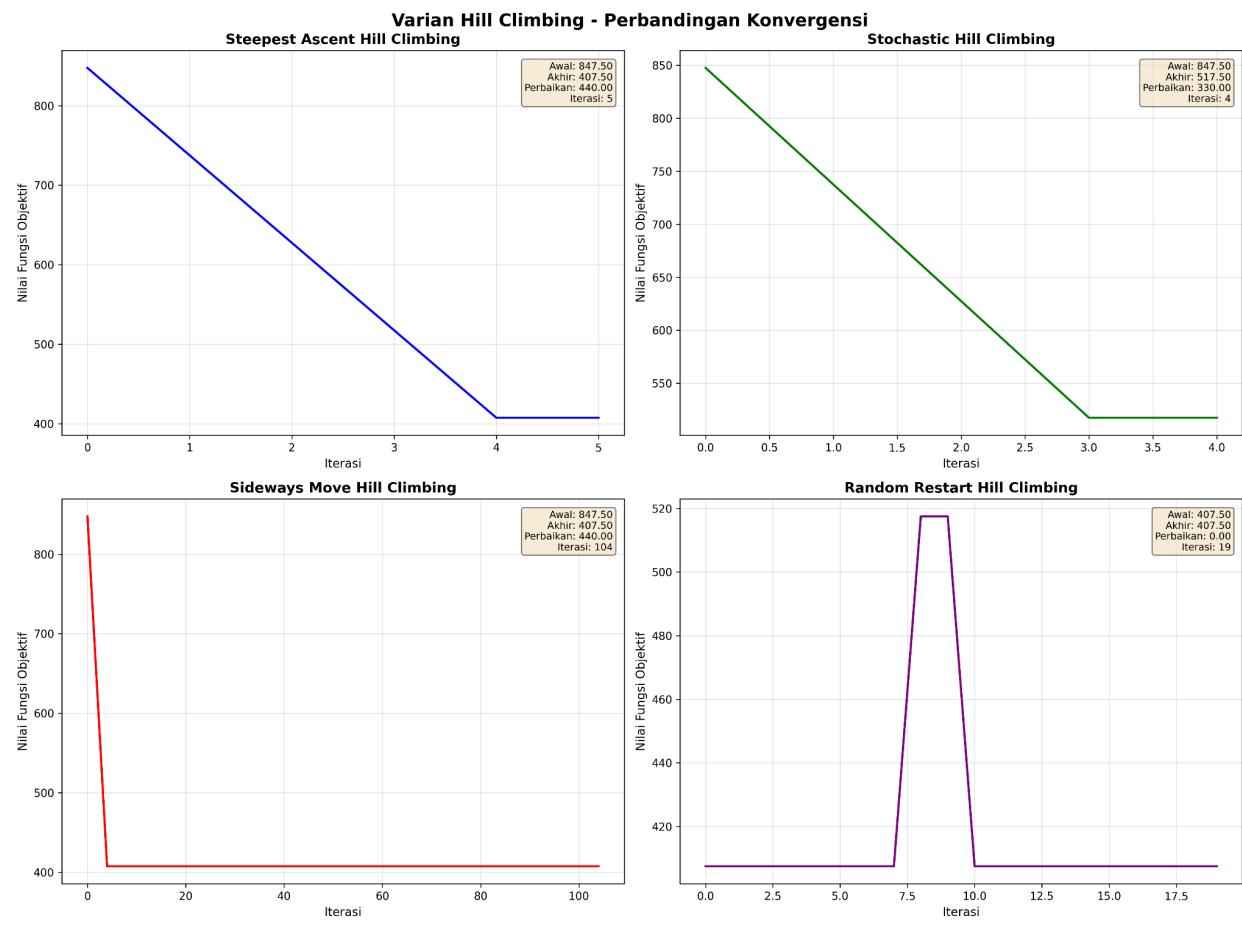
>> Best Hill Climbing: Steepest Ascent
Bins: 4, Score: 407.50
```

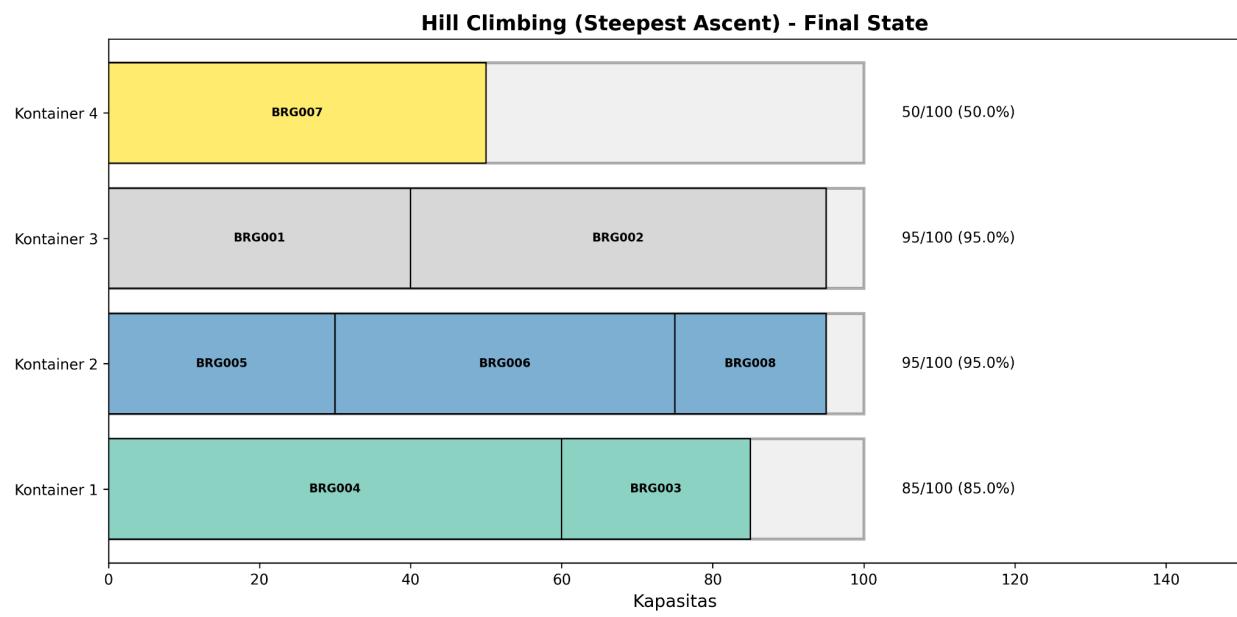
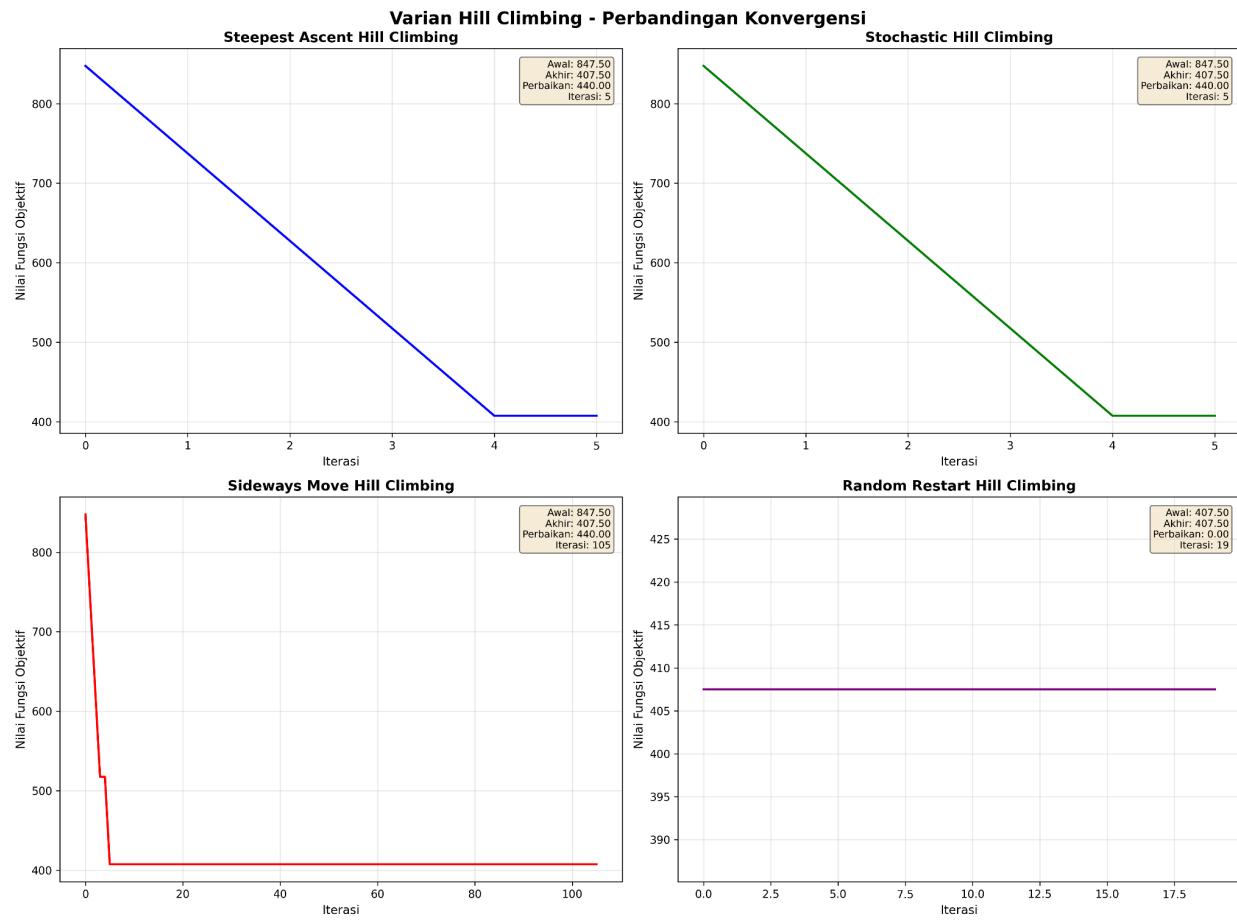
Berdasarkan eksperimen diatas, steepest ascent konsisten paling cepat, konvergen ke optimum lokal. Stochastic mengeluarkan nilai 407.50–517.50. Sifat acak memberi kemungkinan

menemukan solusi berbeda, meski tidak selalu lebih baik.. Sideways move memiliki iterasi dan waktu paling lama diantara varian lainnya. Random restart meskipun mencoba bnyk titik awal acak, tidak ada peningkatan hasil. berarti *search space* tidak memiliki alternatif global optimum lebih baik.

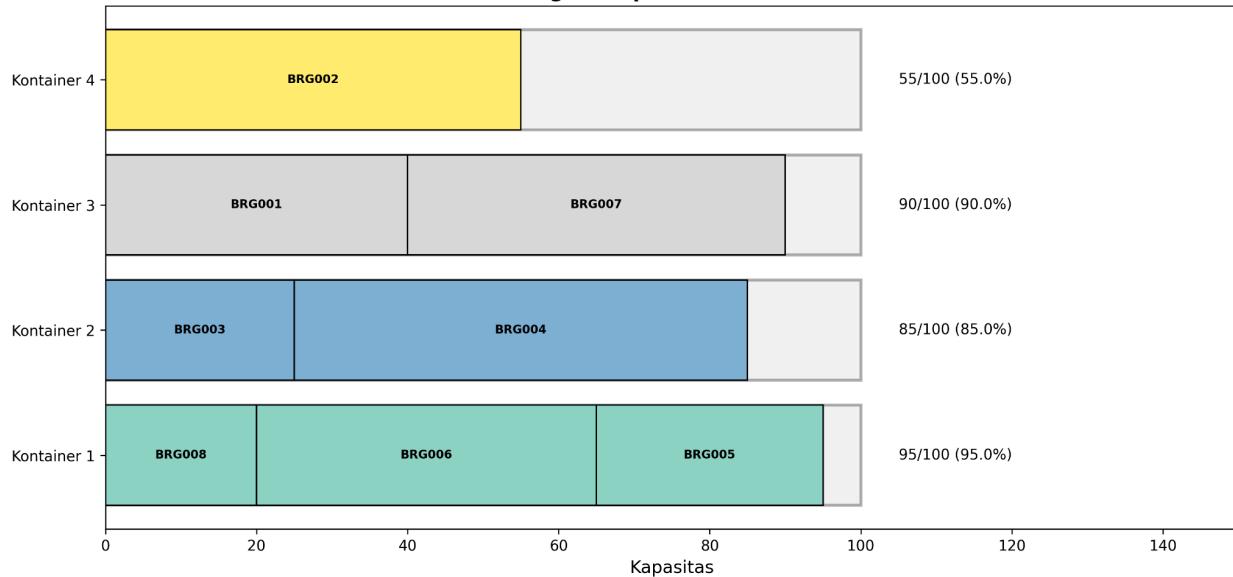
Visualisasi:



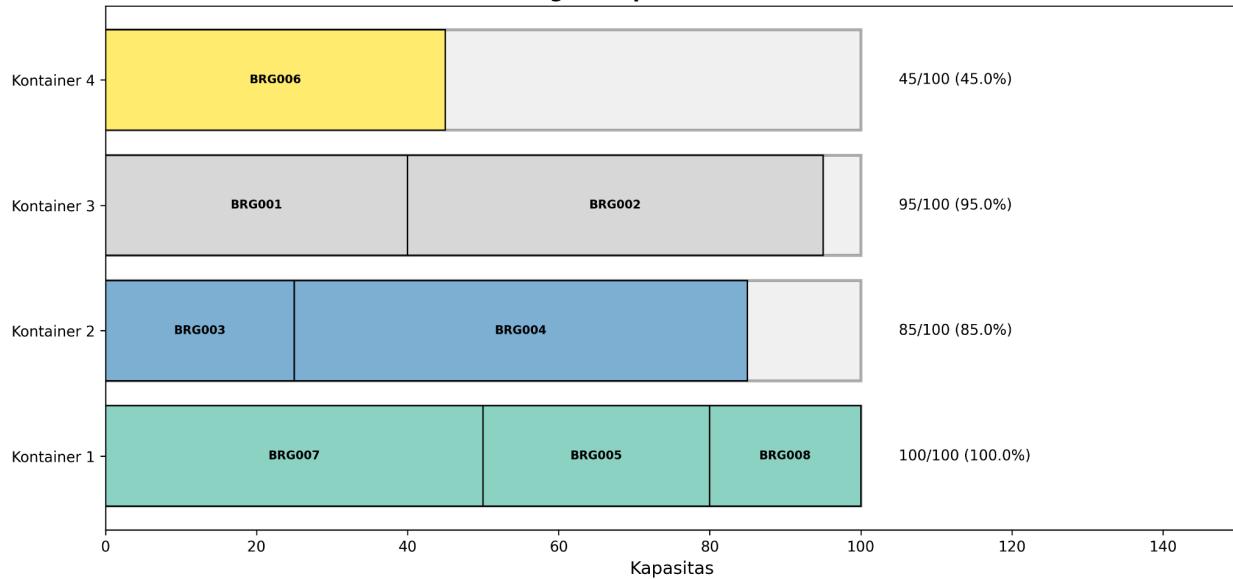




**Hill Climbing (Steepest Ascent) - Final State**



**Hill Climbing (Steepest Ascent) - Final State**



## 2. Simulated Annealing

SA diuji dengan parameter: T\_initial = 1000, alpha = 0.95, T\_min = 0.1, dan max\_iterations = 1000.

```
[2] Running Simulated Annealing...
Initial bins: 8, Score: 847.50
Final bins: 4
Objective score: 407.50
Stuck count: 31
Time: 0.005 seconds
=====
Simulated Annealing - Final State
=====
Total Kontainer yang Digunakan: 4
Total Barang: 8
Total Ukuran: 325
Total Kapasitas: 400
Utilisasi: 81.25%
-----
Kontainer 1: 95/100 (95.0%)
- BRG001: 40
- BRG002: 55

Kontainer 2: 60/100 (60.0%)
- BRG004: 60

Kontainer 3: 100/100 (100.0%)
- BRG007: 50
- BRG008: 20
- BRG005: 30

Kontainer 4: 70/100 (70.0%)
- BRG003: 25
- BRG006: 45
=====

[3] Running Genetic Algorithm...
Final bins: 4
Objective score: 407.50
Time: 0.117 seconds
```

```
[2] Running Simulated Annealing...
Initial bins: 8, Score: 847.50
Final bins: 4
Objective score: 407.50
Stuck count: 26
Time: 0.004 seconds
=====
Simulated Annealing - Final State
=====
Total Kontainer yang Digunakan: 4
Total Barang: 8
Total Ukuran: 325
Total Kapasitas: 400
Utilisasi: 81.25%
-----
Kontainer 1: 85/100 (85.0%)
- BRG002: 55
- BRG005: 30

Kontainer 2: 70/100 (70.0%)
- BRG007: 50
- BRG008: 20

Kontainer 3: 85/100 (85.0%)
- BRG006: 45
- BRG001: 40

Kontainer 4: 85/100 (85.0%)
- BRG003: 25
- BRG004: 60
=====
```

```
[2] Running Simulated Annealing...
Initial bins: 8, Score: 847.50
Final bins: 4
Objective score: 407.50
Stuck count: 30
Time: 0.004 seconds
=====
Simulated Annealing - Final State
=====
Total Kontainer yang Digunakan: 4
Total Barang: 8
Total Ukuran: 325
Total Kapasitas: 400
Utilisasi: 81.25%
-----
Kontainer 1: 85/100 (85.0%)
- BRG004: 60
- BRG003: 25

Kontainer 2: 70/100 (70.0%)
- BRG007: 50
- BRG008: 20

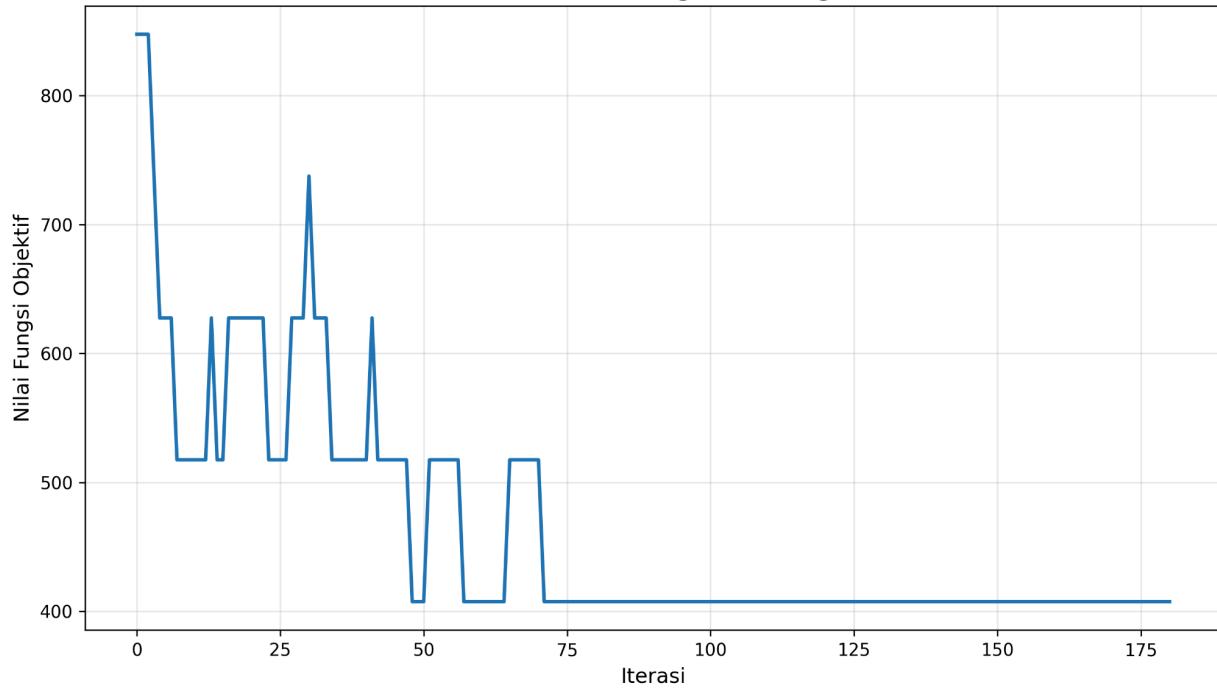
Kontainer 3: 75/100 (75.0%)
- BRG005: 30
- BRG006: 45

Kontainer 4: 95/100 (95.0%)
- BRG001: 40
- BRG002: 55
=====
```

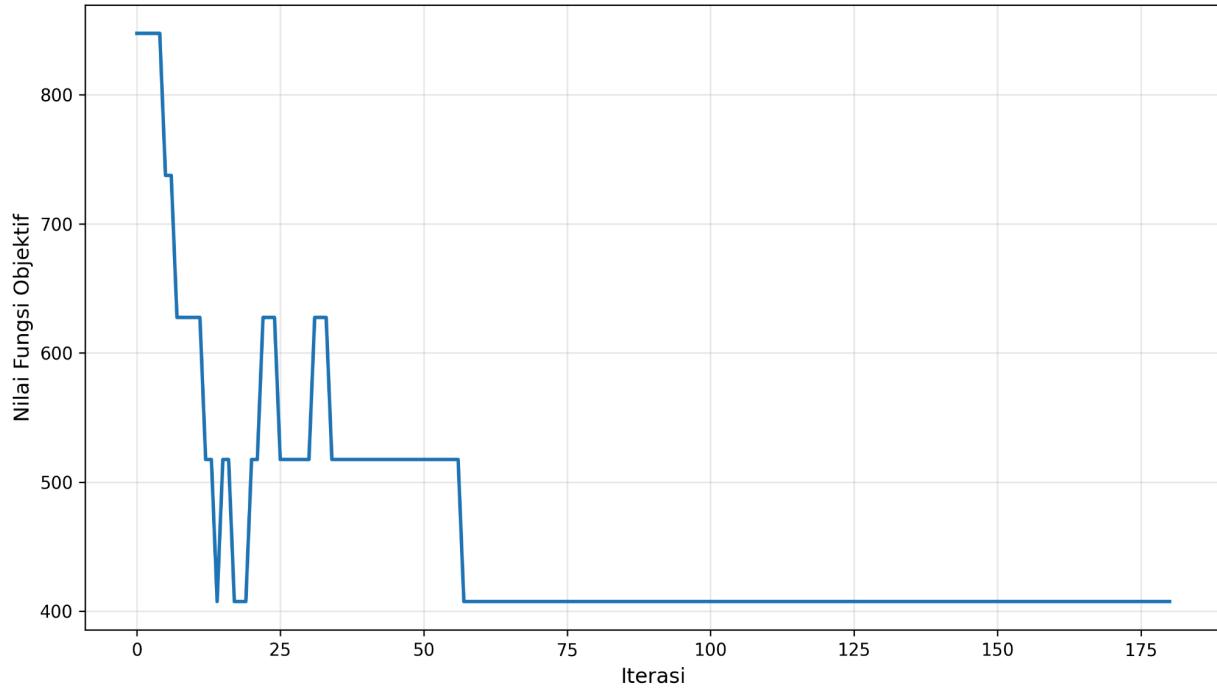
Berdasarkan eksperimen diatas, simulated annealing berjalan lebih cepat dibanding hill climbing dan mencapai hasil optilam secara konsisten. Distribusi beban antar kontainer juga sangat seimbang.

Visualisasi:

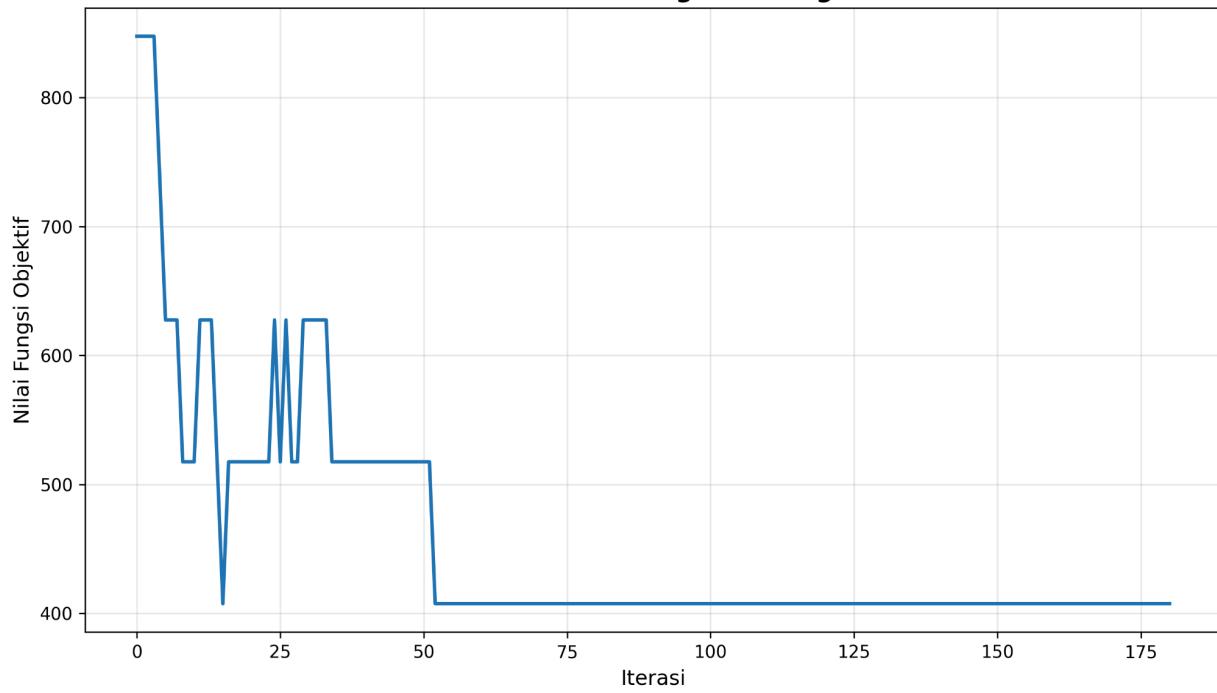
**Simulated Annealing - Convergence**



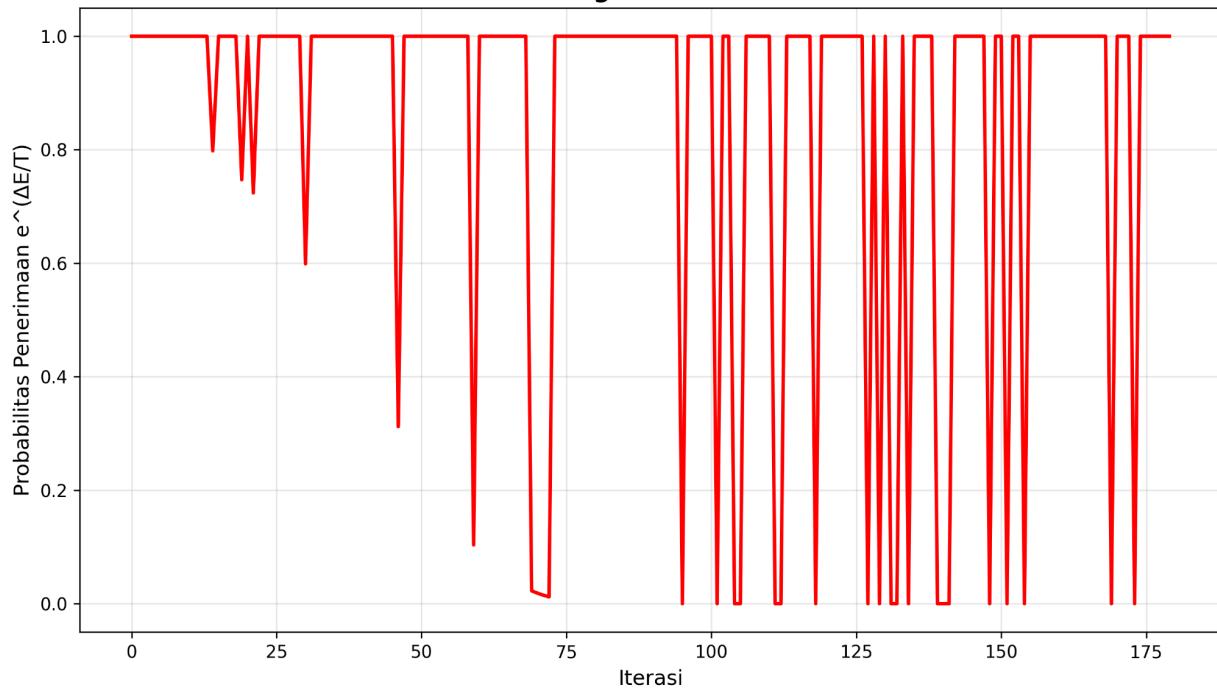
**Simulated Annealing - Convergence**



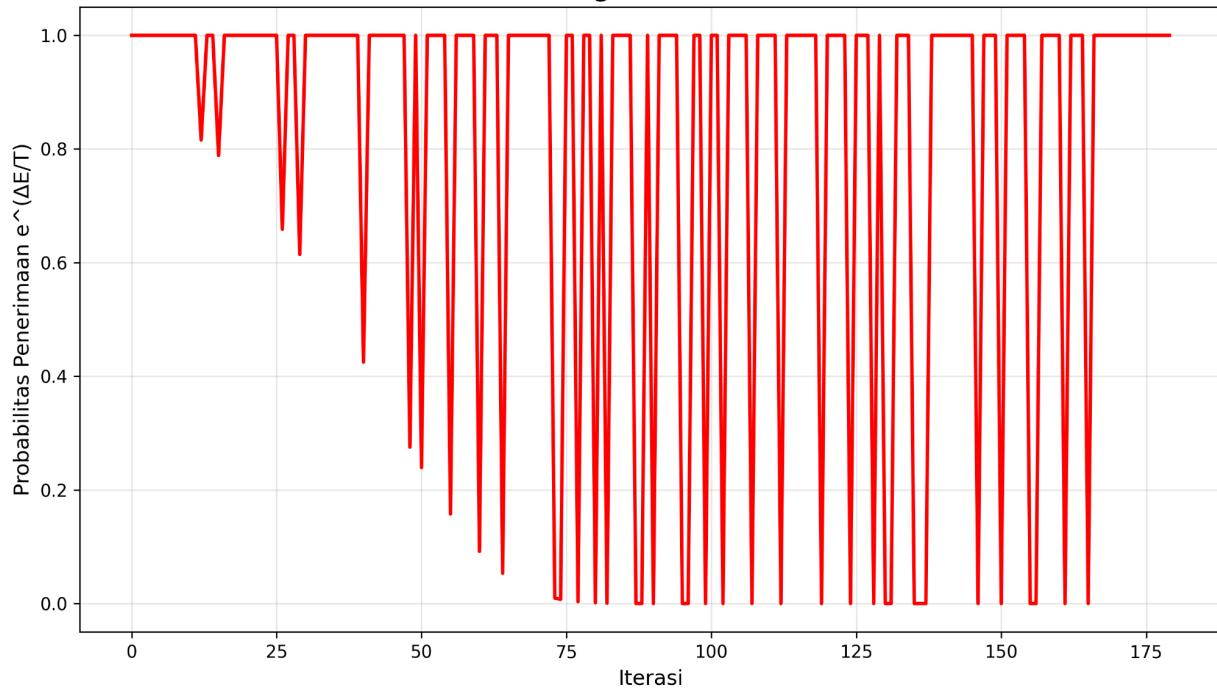
**Simulated Annealing - Convergence**



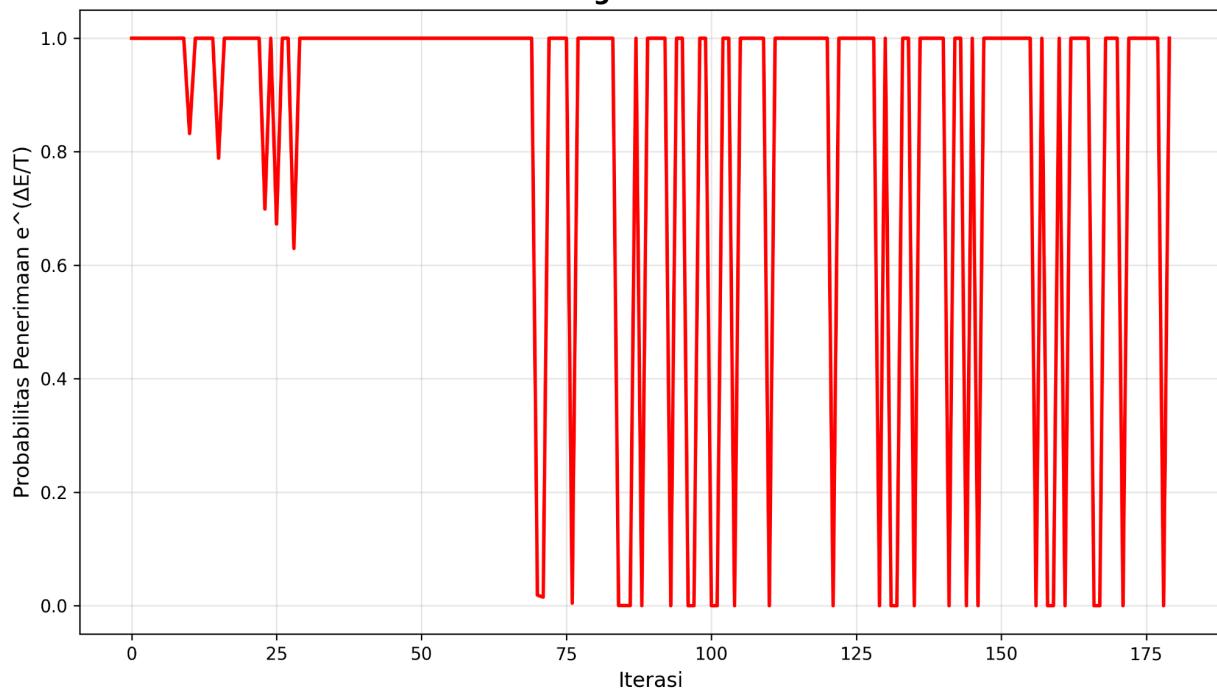
**Simulated Annealing - Probabilitas Penerimaan**



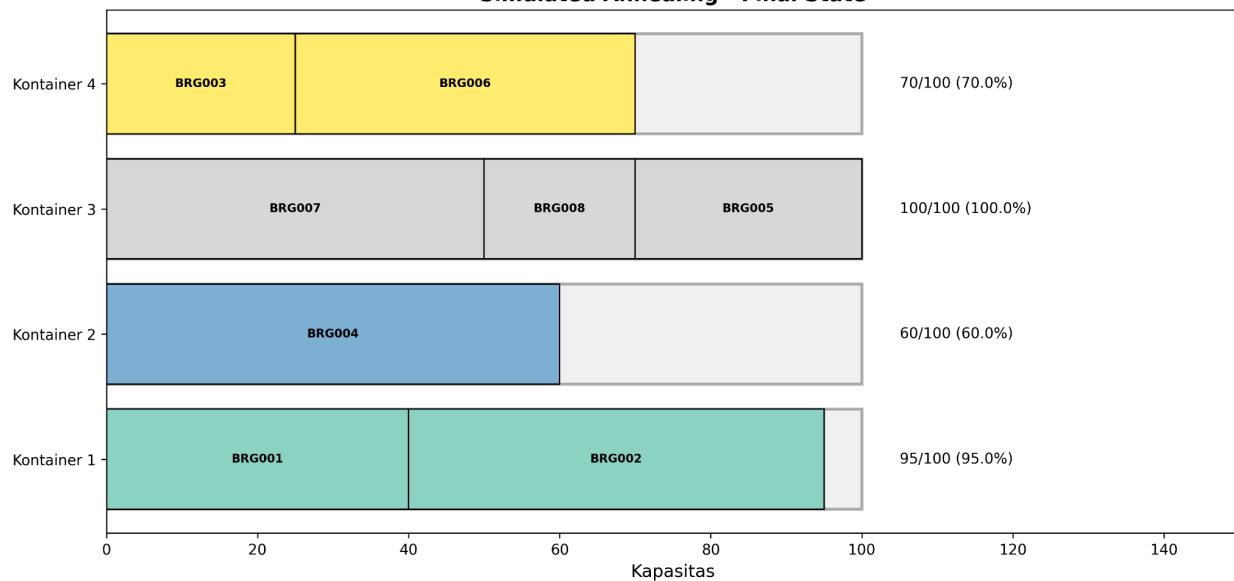
**Simulated Annealing - Probabilitas Penerimaan**



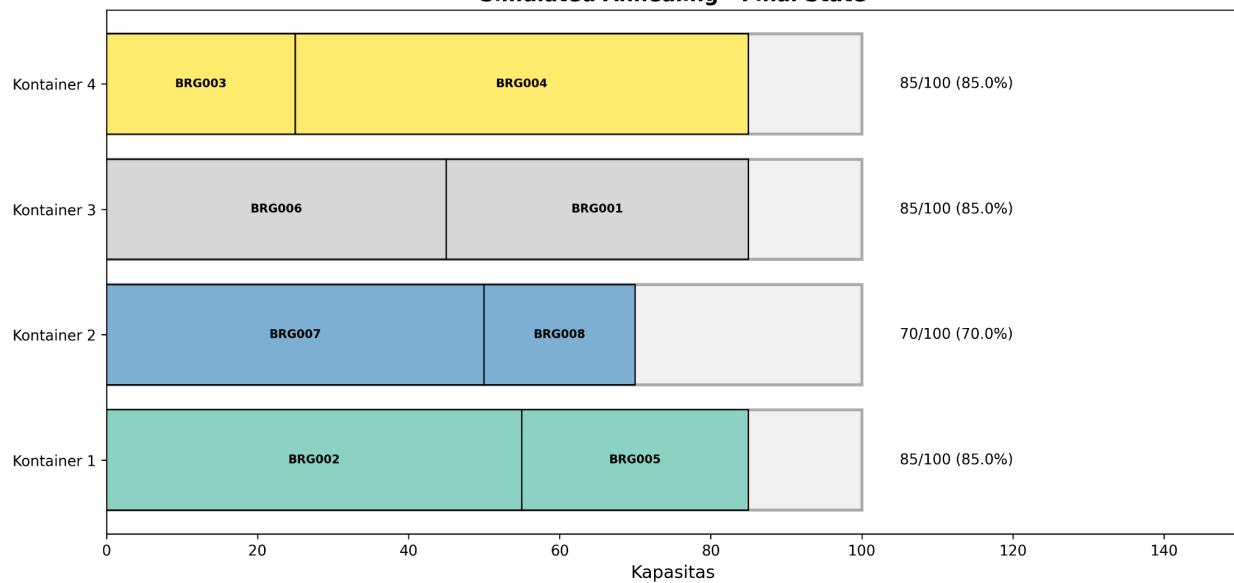
### Simulated Annealing - Probabilitas Penerimaan

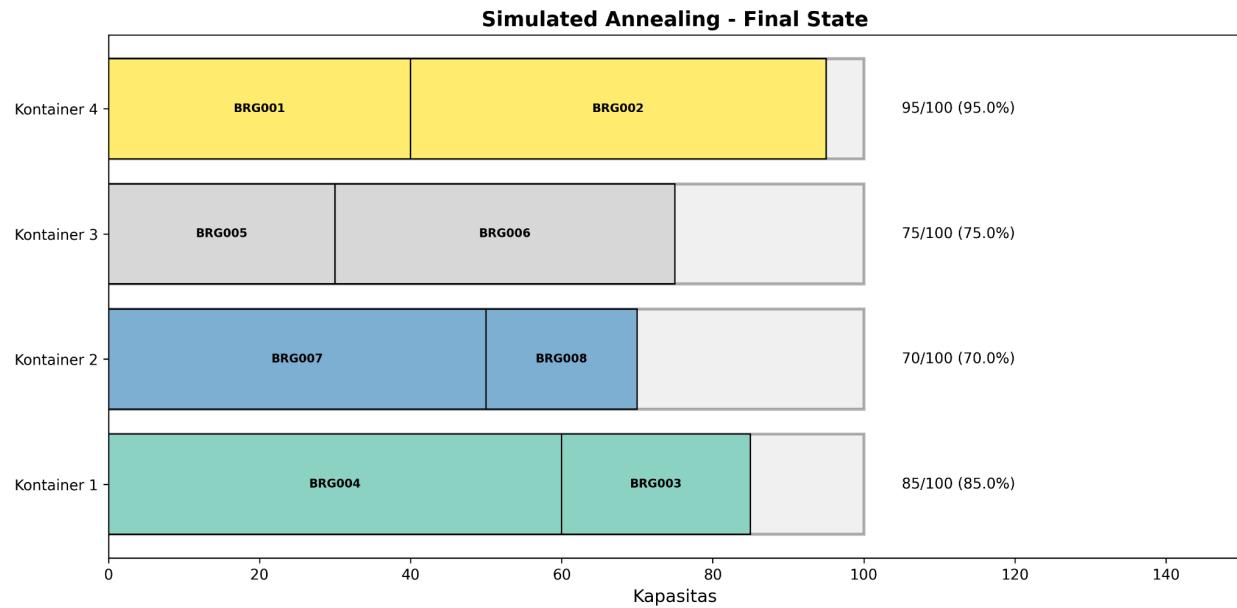


**Simulated Annealing - Final State**



Simulated Annealing - Final State





### 3. Genetic Algorithm

GA dijalankan dengan parameter: population\_size = 50, generations = 100, mutation\_rate = 0.1, crossover\_rate = 0.8, dan elitism = 2.

```
[3] Running Genetic Algorithm...
Final bins: 4
Objective score: 407.50
Time: 0.117 seconds
=====
Genetic Algorithm - Final State
=====
Total Kontainer yang Digunakan: 4
Total Barang: 8
Total Ukuran: 325
Total Kapasitas: 400
Utilisasi: 81.25%
=====
Kontainer 1: 100/100 (100.0%)
- BRG008: 20
- BRG002: 55
- BRG003: 25

Kontainer 2: 70/100 (70.0%)
- BRG001: 40
- BRG005: 30

Kontainer 3: 95/100 (95.0%)
- BRG006: 45
- BRG007: 50

Kontainer 4: 60/100 (60.0%)
- BRG004: 60
=====
```

```
[3] Running Genetic Algorithm...
Final bins: 4
Objective score: 407.50
Time: 0.140 seconds
=====
Genetic Algorithm - Final State
=====
Total Kontainer yang Digunakan: 4
Total Barang: 8
Total Ukuran: 325
Total Kapasitas: 400
Utilisasi: 81.25%
=====
Kontainer 1: 100/100 (100.0%)
- BRG002: 55
- BRG003: 25
- BRG008: 20

Kontainer 2: 80/100 (80.0%)
- BRG007: 50
- BRG005: 30

Kontainer 3: 100/100 (100.0%)
- BRG004: 60
- BRG001: 40

Kontainer 4: 45/100 (45.0%)
- BRG006: 45
=====
```

```
[3] Running Genetic Algorithm...
Final bins: 4
Objective score: 407.50
Time: 0.128 seconds
=====
Genetic Algorithm - Final State
=====
Total Kontainer yang Digunakan: 4
Total Barang: 8
Total Ukuran: 325
Total Kapasitas: 400
Utilisasi: 81.25%
=====
Kontainer 1: 100/100 (100.0%)
- BRG007: 50
- BRG005: 30
- BRG008: 20

Kontainer 2: 85/100 (85.0%)
- BRG006: 45
- BRG001: 40

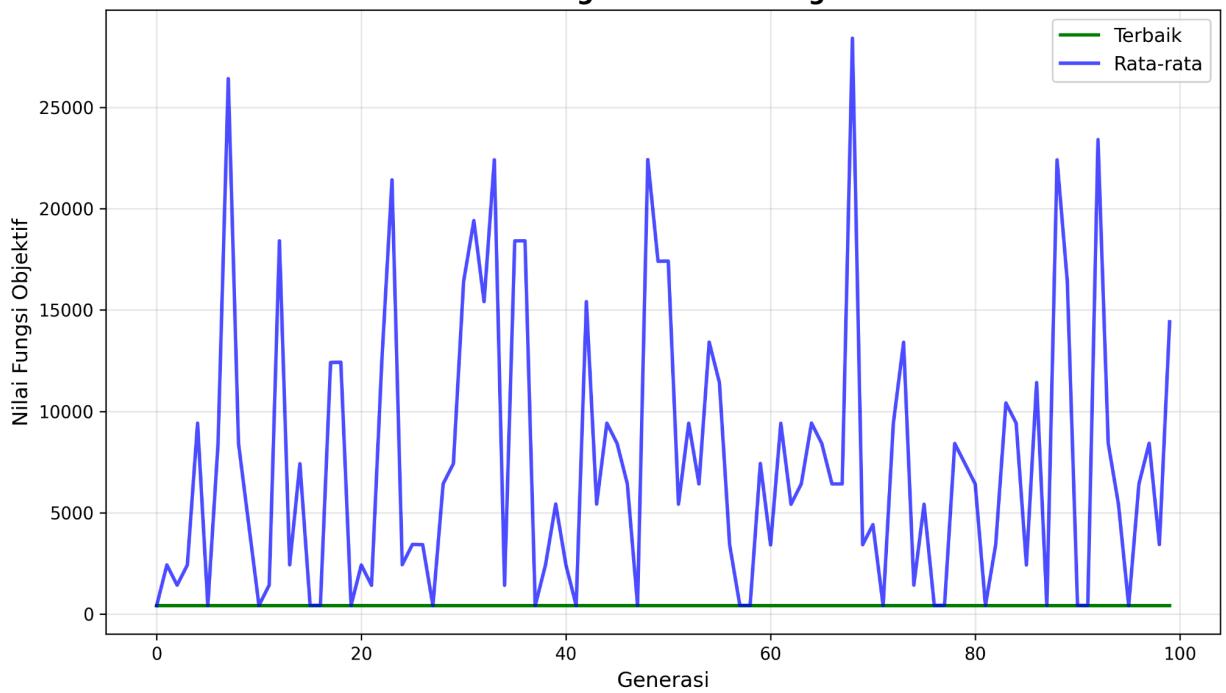
Kontainer 3: 85/100 (85.0%)
- BRG004: 60
- BRG003: 25

Kontainer 4: 55/100 (55.0%)
- BRG002: 55
=====
```

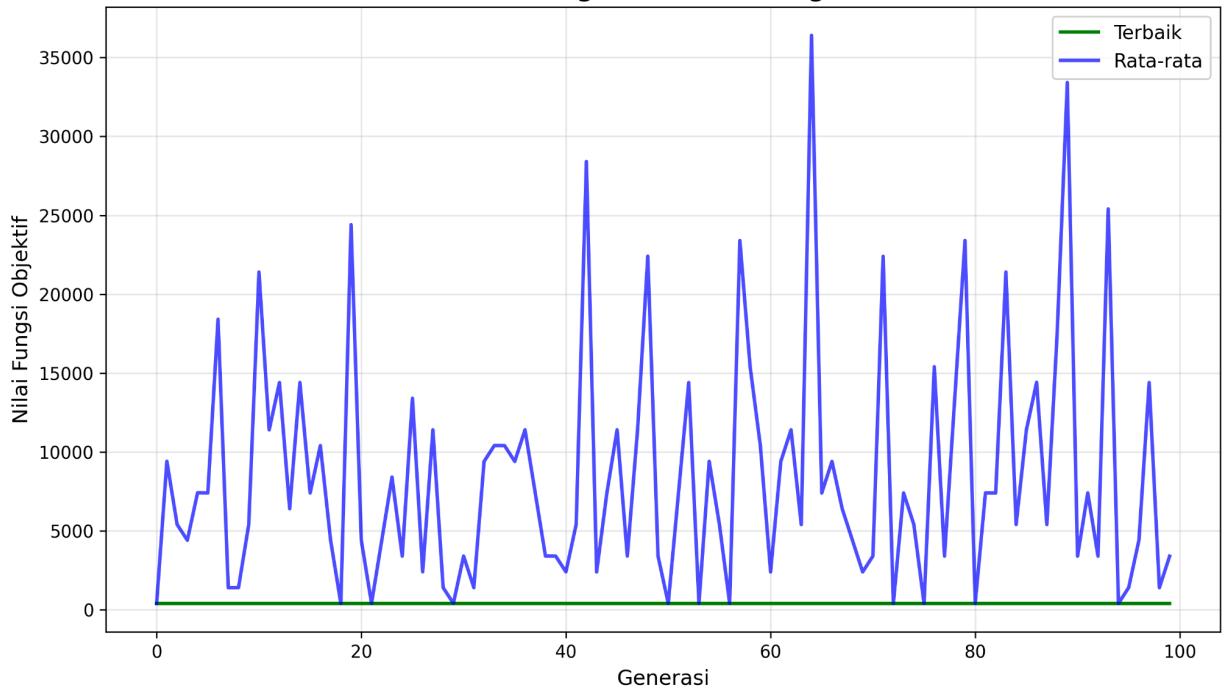
Berdasarkan eksperimen diatas, algoritma ini konsisten mendapatkan skor optimal, utilisasi yang stabil, namun waktu eksekusi yang dtinggi dibanding SA dan hill climbing. Variasi struktur solusi di tiap run menunjukkan efek dari elemen acak pada crossover dan mutasi, yang membantu eksplorasi *solution space*.

Visualisasi:

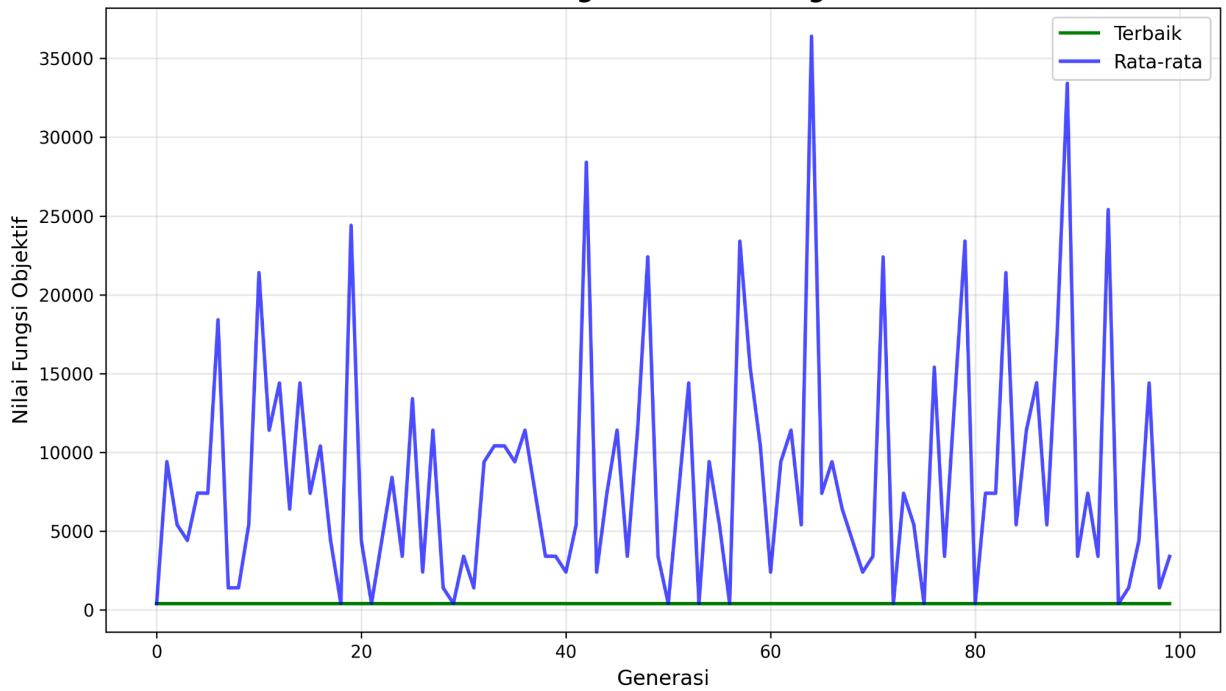
**Genetic Algorithm - Convergence**



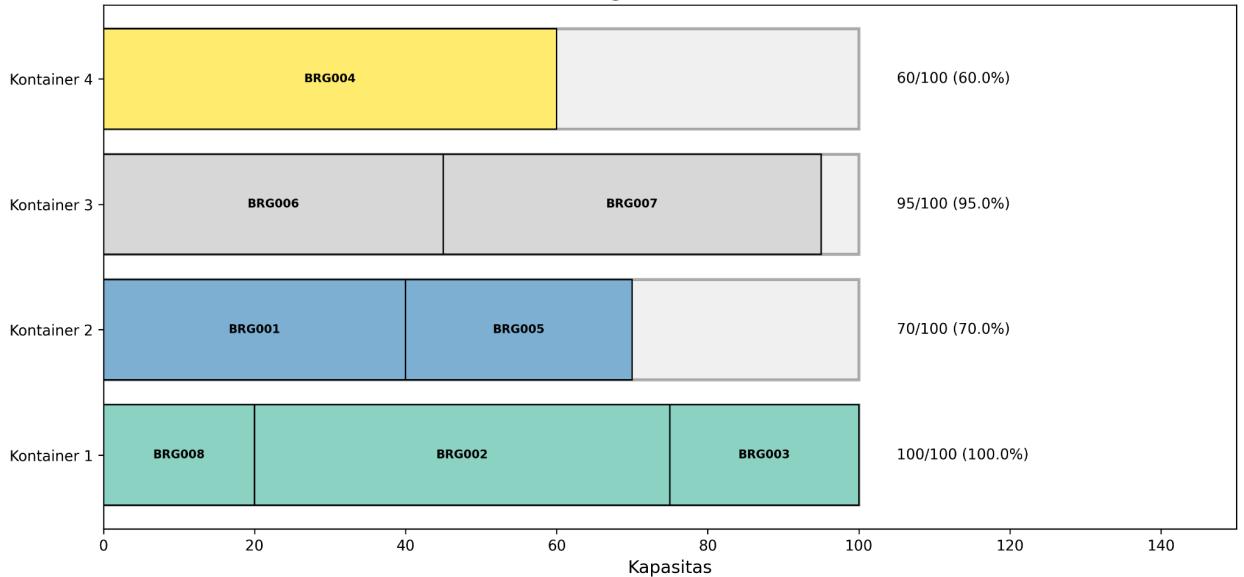
**Genetic Algorithm - Convergence**



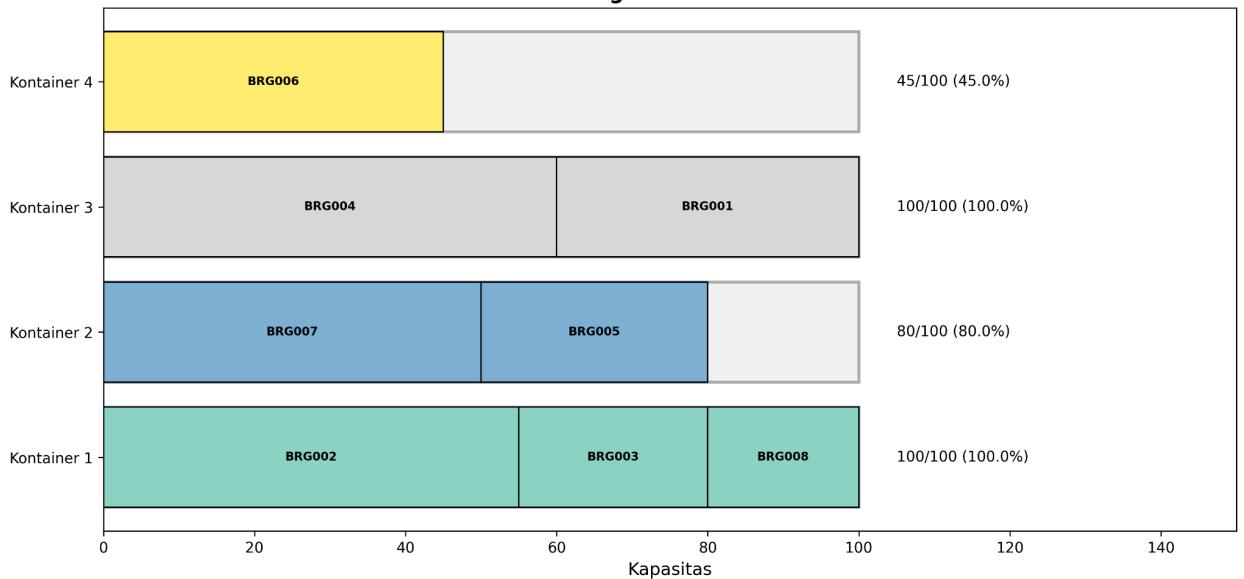
### Genetic Algorithm - Convergence



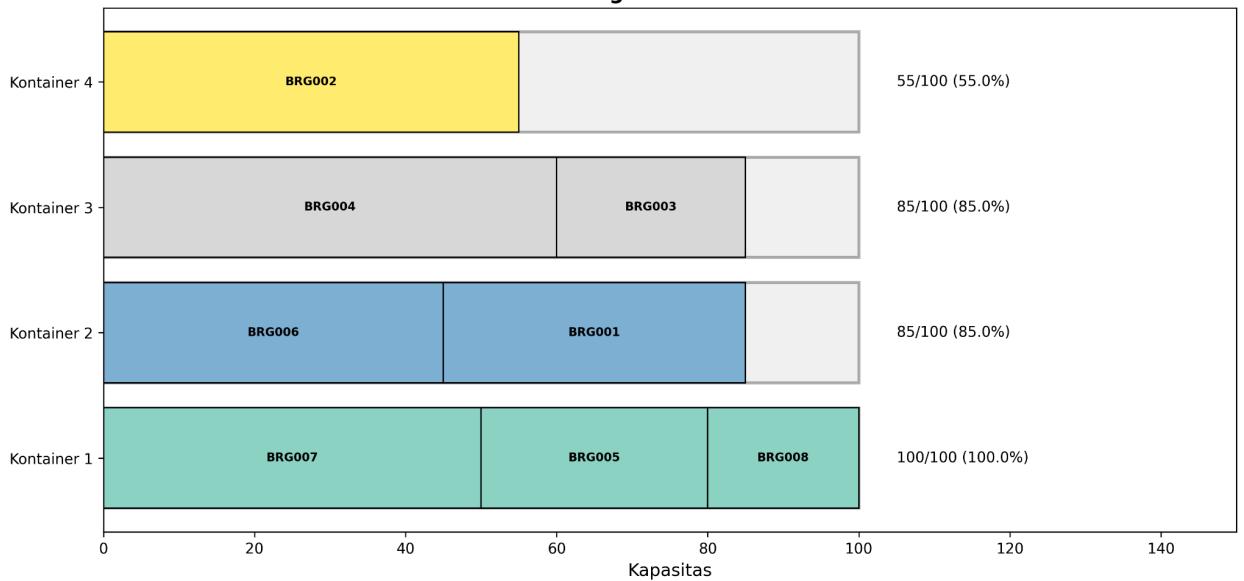
### Genetic Algorithm - Final State



### Genetic Algorithm - Final State



### Genetic Algorithm - Final State



### III. KESIMPULAN DAN SARAN

#### Kesimpulan

Berdasarkan hasil implementasi dan eksperimen terhadap enam algoritma pencarian lokal empat varian Hill Climbing (Steepest Ascent, Stochastic, Sideways Move, dan Random Restart), Simulated Annealing (SA), dan Genetic Algorithm (GA), dapat diambil beberapa kesimpulan sebagai berikut:

1. **Hill Climbing** secara umum memiliki waktu eksekusi paling cepat, terutama varian **Steepest Ascent**, namun algoritma ini cenderung cepat terjebak pada *local optimum*.
  - a. *Stochastic Hill Climbing* memperkenalkan elemen acak yang meningkatkan eksplorasi, tetapi hasilnya tidak selalu lebih baik.
  - b. *Sideways Move* dapat menembus *plateau* namun memerlukan iterasi lebih panjang.
  - c. *Random Restart* mencoba keluar dari *local optimum* melalui banyak titik awal acak, tetapi hasilnya tidak meningkat signifikan karena seluruh ruang pencarian mengarah pada solusi yang sama.
2. **Simulated Annealing (SA)** memberikan keseimbangan terbaik antara kecepatan, stabilitas, dan kualitas solusi. Dengan mekanisme penerimaan solusi lebih buruk di suhu awal, SA mampu keluar dari *local optimum* dan mencapai solusi yang konsisten optimal dengan efisiensi ruang sebesar  $\pm 81\%$ .
3. **Genetic Algorithm (GA)** menghasilkan solusi dengan kualitas setara SA (skor objektif 407.50) namun membutuhkan waktu komputasi lebih lama karena proses evolusi berbasis populasi, crossover, dan mutasi. Meskipun demikian, GA memiliki kemampuan eksplorasi yang tinggi dan berpotensi menemukan solusi global yang lebih baik pada ruang solusi yang lebih kompleks.
4. Secara keseluruhan, **Simulated Annealing** terbukti sebagai **algoritma paling efektif** untuk permasalahan *Bin Packing Problem* dalam eksperimen ini, diikuti oleh Genetic Algorithm dan Hill Climbing sebagai alternatif yang lebih sederhana namun kurang eksploratif.

## Saran

- Uji algoritma pada data yang lebih besar dan kompleks untuk mengukur ketahanan performa.
- Lakukan tuning parameter (misal *cooling rate*, *mutation rate*) agar hasil lebih optimal.
- Pertimbangkan gabungan algoritma (hybrid) seperti SA–GA untuk eksplorasi dan konvergensi lebih baik.
- Tambahkan visualisasi dan analisis hasil agar perilaku algoritma lebih mudah diamati.
- Kembangkan constraint tambahan seperti prioritas barang atau waktu pemrosesan untuk aplikasi nyata

## IV. PEMBAGIAN TUGAS

Nama	Tugas
Fawwaz Aydin Mustofa (18222109)	<ul style="list-style-type: none"><li>• Membuat README.md</li><li>• Melakukan testing dan running code</li><li>• Mengisi laporan bagian pembahasan</li><li>• Mengisi laporan bagian kesimpulan dan saran</li></ul>
Daffa Athalla Rajasa (182223053)	<ul style="list-style-type: none"><li>• Membuat representasi state dengan file bin_packing.py.</li><li>• Mengimplementasikan seluruh varian Hill Climbing: steepest ascent, stochastic, sideways, dan random restart dengan file hill_climbing.py.</li><li>• Menangani I/O data (load JSON, save hasil) dan utilitas pendukung (timer, logger) dengan file utils.py.</li><li>• Menyusun script untuk menjalankan seluruh algoritma secara terpusat dengan main.py.</li><li>• Membuat cover dan kerangka laporan</li></ul>
Adam Joaquin Girsang (182223089)	<ul style="list-style-type: none"><li>• Merancang fungsi objektif dan fitness untuk mengukur kualitas solusi dengan objective_function.py.</li><li>• Mengembangkan algoritma Simulated Annealing (cooling schedule,</li></ul>

	<p>acceptance probability) dengan simulated_annealing.py.</p> <ul style="list-style-type: none"> <li>• Mengimplementasikan Genetic Algorithm dengan seleksi turnamen, crossover OX, mutasi move/swap, serta repair function dengan genetic_algorithm.py.</li> <li>• Mengembangkan fungsi visualisasi konvergensi, probabilitas SA, perbandingan GA, dan isi kontainer dengan visualizer.py.</li> <li>• Membuat laporan bagian desain persoalan</li> </ul>
--	---

## V. REFERENSI

*File Spesifikasi Tugas Besar 1 IF3070 Dasar Inteligensi Artifisial 2025/2026.* Tim Asisten Lab AI '22. Diakses pada 30 Oktober, 2025, dari <https://docs.google.com/document/d/1jrx2B5QMU9Hiev93qYbcHCE-XrGdFK13t3bLWvIAZk/edit?pli=1&tab=t.0>

*Module in Edunex IF3070 Foundations of Artificial Intelligence [Parent Class].* IF3070. Diakses pada 30 Oktober, 2025, dari <https://edunex.itb.ac.id/courses/79119/preview>

*Hill Climbing in Artificial Intelligence.* Geeksforgeeks. Diakses pada 30 Oktober, 2025, dari <https://www.geeksforgeeks.org/artificial-intelligence/introduction-hill-climbing-artificial-intelligence/>

*Simulated Annealing.* Geeksforgeeks. Diakses pada 30 Oktober, 2025, dari <https://www.geeksforgeeks.org/dsa/simulated-annealing/>

*Genetic Algorithm.* Geeksforgeeks. Diakses pada 30 Oktober, 2025, dari <https://www.geeksforgeeks.org/dsa/genetic-algorithms/>