LAPORAN TUGAS BESAR 1
IF3170 - Inteligensi Artifisial
Pencarian Solusi Penjadwalan Kelas Mingguan dengan Local Search

Disusun oleh:

Kelompok AI_Hoshino

Muhammad Hazim Ramadan Prajoda 13523009
Faqih Muhammad Syuhada 13523057
Darrel Adinarya Sunanda 13523061

Dosen Pengajar:
Dr. Nur Ulfa Maulidevi, S.T., M.Sc

PROGRAM STUDI TEKNIK INFORMATIKA
SEKOLAH TEKNIK ELEKTRO DAN INFORMATIKA
INSTITUT TEKNOLOGI BANDUNG

Semester II Tahun 2024/2025

Daftar Isi

1 Deskripsi Persoalan **3**
- 1.1 Latar Belakang . . . 3
- 1.2 Definisi Permasalahan . . . 4
  - 1.2.1 Komponen Permasalahan . . . 4
  - 1.2.2 Representasi State . . . 5
  - 1.2.3 Operator Neighbor . . . 5
- 1.3 Tujuan dan Kendala . . . 6
  - 1.3.1 Tujuan . . . 6
  - 1.3.2 Kendala . . . 6

2 Pembahasan **8**
- 2.1 Pemilihan Objective Function . . . 8
  - 2.1.1 Komponen Objective Function . . . 8
  - 2.1.2 Total Objective Function . . . 9
- 2.2 Penjelasan Implementasi Algoritma Local Search . . . 10
  - 2.2.1 Struktur Data dan Representasi . . . 10
  - 2.2.2 Algoritma Hill-Climbing . . . 13
  - 2.2.3 Algoritma Simulated Annealing . . . 17
  - 2.2.4 Algoritma Genetic Algorithm . . . 20
- 2.3 Hasil Eksperimen dan Analisis . . . 25
  - 2.3.1 Setup Eksperimen . . . 25
  - 2.3.2 Hasil Eksperimen Steepest Ascent Hill-Climbing . . . 25
  - 2.3.3 Hasil Eksperimen Stochastic Hill-Climbing . . . 26
  - 2.3.4 Hasil Eksperimen Sideways Move Hill-Climbing . . . 27
  - 2.3.5 Hasil Eksperimen Random Restart Hill-Climbing . . . 28
  - 2.3.6 Hasil Eksperimen Simulated Annealing . . . 29
  - 2.3.7 Hasil Eksperimen Genetic Algorithm . . . 30
  - 2.3.8 Perbandingan Algoritma . . . 31
  - 2.3.9 Visualisasi State Akhir (Solusi Optimal) . . . 33

3 Kesimpulan dan Saran **34**
- 3.1 Kesimpulan . . . 34
- 3.2 Saran . . . 35

4 Pembagian Tugas **36**

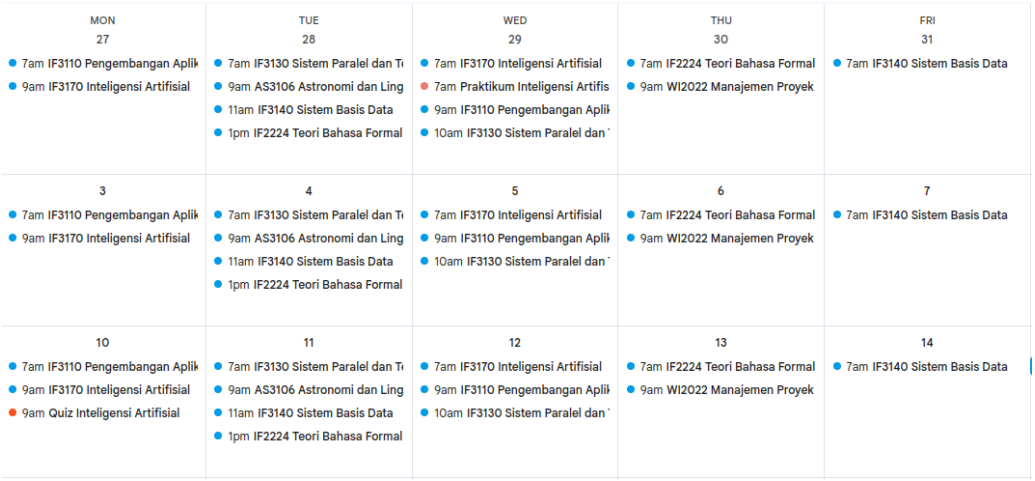
Referensi **37**

Lampiran **38**

BAB 1 Deskripsi Persoalan

1.1 Latar Belakang

Penjadwalan kelas mingguan merupakan permasalahan fundamental dalam manajemen institusi pendidikan yang melibatkan pengalokasian mata kuliah ke dalam slot waktu dan ruangan dengan mempertimbangkan berbagai kendala dan preferensi. Permasalahan ini tergolong dalam kategori *constraint satisfaction problems* (CSP) yang memiliki kompleksitas tinggi seiring dengan bertambahnya jumlah variabel seperti mata kuliah, ruangan, mahasiswa, dan preferensi masing-masing pihak [4]. Penjadwalan yang optimal sangat penting karena dapat meminimalkan konflik waktu mahasiswa, memaksimalkan penggunaan ruangan, dan mempertimbangkan prioritas mata kuliah untuk meningkatkan kepuasan dan kelancaran proses akademik.



Gambar 1.1: Ilustrasi kompleksitas penjadwalan kelas mingguan dengan berbagai kendala

Kompleksitas permasalahan penjadwalan meningkat secara eksponensial seiring bertambahnya jumlah mata kuliah, ruangan, dan mahasiswa, sehingga pencarian solusi optimal termasuk dalam kategori permasalahan NP-hard. Ruang pencarian solusi yang sangat besar membuat pendekatan *brute-force* atau *exhaustive search* menjadi tidak praktis untuk kasus-kasus nyata. Sebagai contoh, dengan 25 mata kuliah, 6 ruangan, dan 5 hari kerja dengan 11 slot waktu per hari, terdapat lebih dari 10^{30} kemungkinan kombinasi jadwal yang dapat dibentuk.

Mengingat kompleksitas tersebut, algoritma *local search* menjadi pilihan yang tepat untuk menyelesaikan permasalahan ini. Algoritma *local search* bekerja dengan cara mengeksplorasi ruang solusi secara iteratif, berpindah dari satu solusi ke solusi tetangga yang lebih baik berdasarkan fungsi objektif tertentu [1]. Beberapa varian algoritma yang digunakan antara lain Hill-Climbing (Steepest Ascent, Stochastic, Sideways Move, dan Random Restart), Simulated Annealing, dan Genetic Algorithm. Meskipun tidak menjamin solusi optimal global, algoritma-algoritma ini dapat menemukan solusi yang cukup baik (*near-optimal*) dalam waktu yang relatif singkat dan efisien secara komputasional.

1.2 Definisi Permasalahan

1.2.1 Komponen Permasalahan

Kelas Mata Kuliah

Kelas mata kuliah merepresentasikan suatu mata kuliah tertentu beserta kelasnya. Setiap kelas mata kuliah memiliki atribut sebagai berikut:

- **Kode**: Kode identifikasi yang menggabungkan kode mata kuliah dan kode kelas (contoh: `IF3071_K01`)
- **Jumlah Mahasiswa**: Jumlah mahasiswa yang terdaftar pada kelas tersebut
- **Jumlah SKS**: Bobot kredit mata kuliah yang menentukan durasi pertemuan per minggu (1 SKS = 1 jam pertemuan)

Ruangan

Ruangan adalah tempat berlangsungnya pertemuan mata kuliah. Setiap ruangan memiliki atribut:

- **Kode Ruangan**: Identifikasi unik ruangan (contoh: ***7609, multimedia***))
- **Kuota**: Kapasitas maksimum mahasiswa yang dapat ditampung oleh ruangan

Waktu

Waktu merepresentasikan slot waktu pertemuan dalam seminggu. Atribut yang dimiliki:

- **Hari**: Hari pertemuan dari Senin hingga Jumat (direpresentasikan dengan enumerasi 0-4)
- **Jam Mulai**: Waktu mulai pertemuan (dalam format jam, rentang 7-17)
- **Jam Selesai**: Waktu selesai pertemuan (dalam format jam, maksimal 18)

Setiap pertemuan dapat berlangsung selama 1 hingga 3 jam berturut-turut, dengan durasi total pertemuan dalam seminggu harus sesuai dengan jumlah SKS mata kuliah tersebut.

Mahasiswa

Mahasiswa merepresentasikan peserta yang mengambil mata kuliah. Setiap mahasiswa memiliki:

- **NIM**: Nomor Induk Mahasiswa sebagai identitas unik
- **Daftar Mata Kuliah**: Kumpulan kode kelas mata kuliah yang diambil oleh mahasiswa

- **Prioritas**: Urutan prioritas pengambilan mata kuliah (1 = prioritas tertinggi)

Prioritas mata kuliah digunakan untuk menentukan bobot dalam perhitungan *objective function*, di mana mata kuliah dengan prioritas lebih tinggi akan memiliki penalti yang lebih besar jika terjadi konflik.

1.2.2 Representasi State

State dalam permasalahan penjadwalan ini direpresentasikan sebagai pemetaan setiap pertemuan mata kuliah ke pasangan waktu dan ruangan. Sebuah *state* lengkap berisi seluruh alokasi pertemuan untuk semua mata kuliah yang harus dijadwalkan.

Dalam implementasi, *state* direpresentasikan menggunakan kelas `State` yang berisi daftar objek `Allocation`. Setiap objek `Allocation` merepresentasikan satu pertemuan mata kuliah dengan atribut:

- `course_class`: Objek kelas mata kuliah
- `time_slot`: Objek slot waktu yang berisi hari, jam mulai, dan jam selesai
- `room`: Objek ruangan tempat pertemuan berlangsung

State awal dibangkitkan secara acak dengan mengalokasikan setiap mata kuliah ke slot waktu dan ruangan secara random, dengan memastikan total durasi pertemuan sesuai dengan jumlah SKS.

1.2.3 Operator Neighbor

Untuk melakukan eksplorasi ruang solusi, terdapat dua jenis operasi yang dapat dilakukan untuk menghasilkan *state* tetangga (*neighbor*):

Operasi *Move*

Operasi *move* memindahkan satu pertemuan mata kuliah ke slot waktu dan/atau ruangan yang berbeda. Operasi ini dilakukan dengan cara:

1. Memilih satu pertemuan mata kuliah secara acak atau sistematis
2. Mengubah atribut waktu (hari, jam mulai, jam selesai) dan/atau ruangan dari pertemuan tersebut
3. Durasi pertemuan tetap dipertahankan

Operasi ini menghasilkan $N \times D \times H \times R$ tetangga untuk setiap pertemuan, di mana:

- N = jumlah pertemuan mata kuliah
- D = jumlah hari (5 hari: Senin-Jumat)
- H = jumlah slot jam yang mungkin (tergantung durasi pertemuan)
- R = jumlah ruangan

Operasi *Swap*

Operasi *swap* menukar posisi dua pertemuan mata kuliah, baik waktu maupun ruangannya. Operasi ini dilakukan dengan cara:

1. Memilih dua pertemuan mata kuliah yang berbeda
2. Menukar slot waktu dan ruangan dari kedua pertemuan tersebut

Operasi ini menghasilkan $\frac{N \times (N-1)}{2}$ tetangga yang mungkin untuk setiap *state*, di mana N adalah jumlah total pertemuan mata kuliah.

Kedua operasi ini memungkinkan algoritma untuk mengeksplorasi ruang solusi dengan cara yang berbeda, sehingga meningkatkan kemungkinan menemukan solusi yang lebih baik.

1.3 Tujuan dan Kendala

1.3.1 Tujuan

Tujuan utama dari permasalahan penjadwalan ini adalah meminimalkan nilai *objective function* yang merepresentasikan total penalti dari berbagai pelanggaran kendala dan preferensi. Semakin kecil nilai *objective function*, semakin baik kualitas jadwal yang dihasilkan. Solusi optimal adalah jadwal dengan nilai *objective function* sama dengan nol, yang berarti tidak ada pelanggaran kendala sama sekali.

1.3.2 Kendala

Terdapat beberapa kendala yang harus diperhatikan dalam penjadwalan kelas mingguan:

1. **Tidak ada konflik waktu untuk mahasiswa**: Setiap mahasiswa tidak boleh memiliki dua atau lebih mata kuliah yang berlangsung pada waktu yang sama atau tumpang tindih. Pelanggaran kendala ini akan memberikan penalti sebesar jumlah jam yang tumpang tindih.
2. **Tidak ada konflik ruangan**: Setiap ruangan hanya dapat digunakan oleh satu pertemuan mata kuliah pada waktu tertentu. Jika terdapat lebih dari satu pertemuan di ruangan yang sama pada waktu yang sama, maka akan diberikan penalti yang diperhitungkan berdasarkan jumlah mahasiswa dan prioritas mata kuliah mereka.
3. **Kapasitas ruangan mencukupi**: Jumlah mahasiswa yang mengikuti suatu pertemuan tidak boleh melebihi kapasitas ruangan yang tersedia. Jika terjadi kelebihan kapasitas, penalti dihitung berdasarkan selisih jumlah mahasiswa dengan kapasitas ruangan dikalikan dengan durasi pertemuan.
4. **Setiap mata kuliah terjadwalkan sesuai SKS**: Total durasi pertemuan setiap mata kuliah dalam seminggu harus sama dengan jumlah SKS yang dimiliki (1 SKS = 1 jam). Kendala ini dipastikan terpenuhi sejak inisialisasi *state* dan dipertahankan oleh operator *neighbor*.
5. **Jadwal hanya pada hari kerja**: Pertemuan hanya dapat dijadwalkan pada hari Senin hingga Jumat, dengan rentang waktu dari jam 7 pagi hingga jam 6 sore (jam 18).

Kendala-kendala ini dikuantifikasi dalam *objective function* yang akan digunakan sebagai panduan oleh algoritma *local search* untuk menemukan solusi yang optimal atau mendekati optimal.

BAB 2 Pembahasan

2.1 Pemilihan Objective Function

Pada implementasi ini, objective function dirancang untuk meminimalkan total penalty dari jadwal yang dihasilkan. Semakin kecil nilai penalty, semakin baik kualitas jadwal tersebut. Objective function terdiri dari tiga komponen utama yang dapat diaktifkan atau dinonaktifkan sesuai kebutuhan.

2.1.1 Komponen Objective Function

Komponen 1: Student Time Conflict Penalty

Komponen ini menghitung penalty dari bentrokan jadwal yang dialami mahasiswa. Jika seorang mahasiswa memiliki dua atau lebih pertemuan mata kuliah yang waktunya bertumpuk (overlap) pada hari yang sama, maka akan timbul penalty.

$$f_1(s) = \sum_{\text{student}} \sum_{i=1}^{n-1} \sum_{j=i+1}^{n} \text{overlap\_hours}(m_i, m_j) \tag{2.1}$$

dengan:

- n adalah jumlah pertemuan mata kuliah yang diambil oleh seorang mahasiswa
- m_i dan m_j adalah dua pertemuan mata kuliah yang berbeda
- overlap_hours(m_i, m_j) menghitung durasi overlap (dalam jam) antara dua pertemuan jika keduanya terjadi pada hari yang sama

Lebih detail, untuk dua pertemuan pada hari yang sama:

$$\text{overlap\_hours}(m_i, m_j) = \max(0, \min(\text{end}_i, \text{end}_j) - \max(\text{start}_i, \text{start}_j)) \tag{2.2}$$

Penalty dihitung dengan menjumlahkan semua jam yang bertumpuk untuk setiap mahasiswa di seluruh pertemuan mata kuliah mereka.

Komponen 2: Room Conflict Penalty dengan Bobot Prioritas

Komponen ini menghitung penalty ketika beberapa mata kuliah dijadwalkan di ruangan yang sama pada waktu yang sama. Penalty diperberat berdasarkan prioritas mata kuliah bagi setiap mahasiswa.

$$f_2(s) = \sum_{r,d,h} \sum_{m \in C_{r,d,h}} \sum_{\text{student} \in m} w_p \tag{2.3}$$

dengan:

- $C_{r,d,h}$ adalah himpunan pertemuan mata kuliah yang terjadwal di ruangan r pada hari d jam h

- w_p adalah bobot prioritas untuk mahasiswa tersebut pada mata kuliah m

Bobot prioritas didefinisikan sebagai:

$$w_p = \begin{cases} 1.75 & \text{jika prioritas} = 1 \\ 1.50 & \text{jika prioritas} = 2 \\ 1.25 & \text{jika prioritas} = 3 \\ 1.00 & \text{jika prioritas lainnya} \end{cases} \tag{2.4}$$

Penalty dihitung per jam. Jika terdapat $|C_{r,d,h}| > 1$ (lebih dari satu pertemuan di ruangan yang sama pada waktu yang sama), maka untuk setiap pertemuan dalam konflik, penalty ditambahkan sebesar jumlah mahasiswa dikalikan dengan bobot prioritas mereka masing-masing.

Komponen 3: Capacity Overflow Penalty

Komponen ini menghitung penalty ketika jumlah mahasiswa dalam suatu pertemuan mata kuliah melebihi kapasitas ruangan yang dialokasikan.

$$f_3(s) = \sum_{m \in M} \max(0, \text{studentCount}_m - \text{capacity}_r) \times \text{duration}_m \tag{2.5}$$

dengan:

- M adalah himpunan semua pertemuan mata kuliah yang dijadwalkan
- studentCount_m adalah jumlah mahasiswa yang mengambil mata kuliah pada pertemuan m
- capacity_r adalah kapasitas ruangan yang dialokasikan untuk pertemuan m
- duration_m adalah durasi pertemuan dalam jam

Penalty dihitung sebagai selisih antara jumlah mahasiswa dan kapasitas ruangan (jika melebihi) dikalikan dengan durasi pertemuan. Hal ini memastikan bahwa overflow pada pertemuan yang lebih lama mendapat penalty yang lebih besar.

2.1.2 Total Objective Function

Fungsi objektif total merupakan kombinasi linear dari ketiga komponen penalty di atas:

$$f(s) = f_1(s) + f_2(s) + f_3(s) \tag{2.6}$$

Dalam implementasi ini, semua komponen memiliki bobot yang sama (setiap koefisien bernilai 1) karena setiap kendala dianggap sama pentingnya. Namun, sistem dirancang dengan fleksibilitas untuk mengaktifkan atau menonaktifkan setiap komponen sesuai kebutuhan. Hal ini memungkinkan pengguna untuk fokus pada aspek tertentu dari penjadwalan, misalnya hanya meminimalkan konflik mahasiswa tanpa mempertimbangkan kapasitas ruangan.

Tujuan algoritma optimisasi adalah untuk menemukan state s^* yang meminimalkan fungsi objektif:

$$s^* = \arg\min_{s \in S} f(s) \tag{2.7}$$

di mana S adalah ruang solusi yang berisi semua kemungkinan jadwal yang valid. Nilai ideal adalah $f(s^*) = 0$, yang berarti tidak ada pelanggaran kendala sama sekali.

2.2 Penjelasan Implementasi Algoritma Local Search

2.2.1 Struktur Data dan Representasi

Class State

Class `State` merupakan representasi dari sebuah solusi jadwal lengkap. Implementasinya menggunakan struktur data yang efisien untuk menyimpan dan memanipulasi alokasi pertemuan.

```python
class State:
    class Allocation:
        def __init__(self, course_class: CourseClass,
                     time_slot: TimeSlot,
                     room: Room):
            self.course_class = course_class
            self.time_slot = time_slot
            self.room = room

    def __init__(self):
        self.meetings = []

    def add_meeting(self, meeting: Allocation):
        """Menambahkan satu pertemuan ke dalam jadwal"""
        self.meetings.append(meeting)

    def random_fill(self, classes: dict, rooms: dict):
        """Inisialisasi state dengan alokasi acak"""
        room_list = list(rooms.values())
        for cls in classes.values():
            # Alokasi random room dan time slot
            random_room = random.choice(room_list)
            random_time = TimeSlot.random_with_duration(
                cls.credits
            )
            meeting = State.Allocation(
                cls, random_time, random_room
            )
            self.add_meeting(meeting)
```

Listing 2.1: Implementasi Class State dan Allocation

Class `State` memiliki inner class `Allocation` yang merepresentasikan satu pertemuan mata kuliah dengan pasangan waktu dan ruangan. Setiap `State` menyimpan daftar `meetings` yang berisi seluruh alokasi pertemuan.

Method `random_fill` digunakan untuk membangkitkan state awal secara acak dengan mengalokasikan setiap mata kuliah ke ruangan dan slot waktu random. Durasi pertemuan disesuaikan dengan jumlah SKS mata kuliah tersebut.

Neighbor Generation

Untuk mengeksplorasi ruang solusi, class `State` menyediakan dua method untuk membangkitkan tetangga:

```
def get_random_neighbor(self, rooms: dict) -> State:
    """Generate satu random neighbor dengan operasi
       move atau swap"""
    new_state = copy.deepcopy(self)
    room_list = list(rooms.values())

    if random.random() < 0.7:  # 70% move, 30% swap
        # Move operation: ubah waktu/ruangan satu meeting
        idx = random.randint(0, len(new_state.meetings)-1)
        meeting = new_state.meetings[idx]

        if random.random() < 0.5:
            # Ubah ruangan
            meeting.room = random.choice(room_list)
        else:
            # Ubah waktu (hari dan jam)
            meeting.time_slot = TimeSlot.random_with_duration(
                meeting.course_class.credits
            )
    else:
        # Swap operation: tukar 2 meetings
        idx1, idx2 = random.sample(
            range(len(new_state.meetings)), 2
        )
        m1 = new_state.meetings[idx1]
        m2 = new_state.meetings[idx2]
        m1.room, m2.room = m2.room, m1.room
        m1.time_slot, m2.time_slot = (
            m2.time_slot, m1.time_slot
        )

    return new_state

def get_all_neighbors(self, rooms: dict) -> list:
    """Generate semua possible neighbors secara
       deterministik"""
    neighbors = []
    room_list = list(rooms.values())

    # Generate neighbors dengan move operation
    for i, meeting in enumerate(self.meetings):
        # Try different rooms
        for room in room_list:
            if room != meeting.room:
                # ... buat neighbor baru

        # Try different time slots
        slots = TimeSlot.all_slots_with_duration(
            meeting.course_class.credits
        )
        for slot in slots:
            if not slot.equals(meeting.time_slot):
                # ... buat neighbor baru

```

```
    # Generate neighbors dengan swap operation
    for i in range(len(self.meetings)):
        for j in range(i+1, len(self.meetings)):
            # ... buat neighbor dengan swap meeting i dan j

    return neighbors
```

Listing 2.2: Method untuk Generate Neighbors

Method `get_random_neighbor` digunakan oleh algoritma stochastic yang memilih tetangga secara random. Method ini melakukan operasi move (70% probabilitas) atau swap (30% probabilitas) pada meetings.

Method `get_all_neighbors` digunakan oleh algoritma deterministik seperti Steepest Ascent Hill-Climbing yang perlu mengevaluasi seluruh tetangga. Method ini membangkitkan semua kemungkinan tetangga dengan mengeksplorasi semua kombinasi move dan swap yang valid.

Model Data Lainnya

```
class TimeSlot:
    class Day(Enum):
        MONDAY = 0
        TUESDAY = 1
        WEDNESDAY = 2
        THURSDAY = 3
        FRIDAY = 4

    def __init__(self, day: Day, start_hour: int,
                 end_hour: int):
        self.day = day
        self.start_hour = start_hour
        self.end_hour = end_hour

    def duration(self) -> int:
        return self.end_hour - self.start_hour

    def overlaps_with(self, other: 'TimeSlot') -> bool:
        """Cek apakah dua time slot overlap"""
        if self.day != other.day:
            return False
        return not (self.end_hour <= other.start_hour or
                    self.start_hour >= other.end_hour)

class CourseClass:
    def __init__(self, code: str, studentCount: int,
                 credits: int):
        self.code = code
        self.studentCount = studentCount
        self.students = []
        self.credits = credits

class Room:
    def __init__(self, code: str, capacity: int):
        self.code = code
        self.capacity = capacity

class Student:
```

```
    def __init__(self, id: str, classes: list):
        self.id = id
        self.classes = classes

    def get_priority(self, class_code: str) -> int:
        """Return priority (1-based) dari mata kuliah"""
        try:
            return self.classes.index(class_code) + 1
        except ValueError:
            return len(self.classes) + 1
```

Listing 2.3: Class Model Pendukung

Model data lainnya dirancang untuk merepresentasikan entitas-entitas dalam permasalahan penjadwalan. `TimeSlot` menyimpan informasi waktu pertemuan dan menyediakan method untuk cek overlap. `CourseClass` menyimpan informasi mata kuliah beserta list mahasiswa yang mengambilnya. `Room` menyimpan kapasitas ruangan. `Student` menyimpan prioritas mata kuliah yang diambil.

2.2.2 Algoritma Hill-Climbing

Hill-Climbing adalah algoritma *local search* yang bekerja dengan cara bergerak dari state saat ini ke neighbor yang memiliki nilai objektif lebih baik. Algoritma ini terus melakukan iterasi hingga tidak ada neighbor yang lebih baik (mencapai local optimum) atau mencapai batas iterasi maksimum [1].

Steepest Ascent Hill-Climbing

Steepest Ascent Hill-Climbing (SAHC) adalah varian yang mengevaluasi seluruh neighbor dan memilih yang terbaik di setiap iterasi. Pendekatan ini bersifat deterministik dan greedy.

```
def _steepest_ascent_hc(self, initial_state: State,
                        max_iterations: int):
    current = initial_state
    current_penalty = self.objective_function.calculate(
        current
    )

    results = {
        "iteration_history": [],
        "penalty_history": [current_penalty],
        "best_state": current,
        "best_penalty": current_penalty
    }

    for iteration in range(max_iterations):
        # Generate semua neighbors
        neighbors = current.get_all_neighbors(self.rooms)

        if not neighbors:
            break

        # Evaluasi semua neighbors
        best_neighbor = None
        best_neighbor_penalty = current_penalty

        for neighbor in neighbors:
```

```
            penalty = self.objective_function.calculate(
                neighbor
            )
            if penalty < best_neighbor_penalty:
                best_neighbor = neighbor
                best_neighbor_penalty = penalty

        # Jika tidak ada neighbor yang lebih baik, stop
        if best_neighbor is None:
            break

        # Move ke best neighbor
        current = best_neighbor
        current_penalty = best_neighbor_penalty

        # Update tracking
        results["penalty_history"].append(current_penalty)
        if current_penalty < results["best_penalty"]:
            results["best_state"] = copy.deepcopy(current)
            results["best_penalty"] = current_penalty

    return results
```

Listing 2.4: Implementasi Steepest Ascent Hill-Climbing

Algoritma SAHC melakukan eksplorasi lengkap pada setiap iterasi dengan mengevaluasi seluruh neighbor. Karakteristik utama:

- **Deterministik**: Dengan state awal yang sama, akan menghasilkan hasil yang sama
- **Greedy**: Selalu memilih neighbor terbaik di setiap langkah
- **Komputasi intensif**: Harus mengevaluasi semua neighbor (bisa ribuan evaluasi per iterasi)
- **Mudah stuck**: Berhenti segera ketika mencapai local optimum

Stochastic Hill-Climbing

Stochastic Hill-Climbing memilih neighbor secara random dari neighbor-neighbor yang lebih baik dari state saat ini. Pendekatan ini lebih cepat karena tidak perlu mengevaluasi semua neighbor.

```
def _stochastic_hc(self, initial_state: State,
                   max_iterations: int):
    current = initial_state
    current_penalty = self.objective_function.calculate(
        current
    )

    results = {
        "penalty_history": [current_penalty],
        "best_state": current,
        "best_penalty": current_penalty
    }

    for iteration in range(max_iterations):
```

```
        # Generate random neighbor
        neighbor = current.get_random_neighbor(self.rooms)
        neighbor_penalty = self.objective_function.calculate(
            neighbor
        )

        # Jika neighbor lebih baik, pindah ke sana
        if neighbor_penalty < current_penalty:
            current = neighbor
            current_penalty = neighbor_penalty

            if current_penalty < results["best_penalty"]:
                results["best_state"] = copy.deepcopy(
                    current
                )
                results["best_penalty"] = current_penalty

        results["penalty_history"].append(current_penalty)

    return results
```

Listing 2.5: Implementasi Stochastic Hill-Climbing

Karakteristik Stochastic Hill-Climbing:

- **Non-deterministik**: Hasil bisa berbeda dengan state awal yang sama
- **Cepat**: Hanya perlu 1 evaluasi neighbor per iterasi
- **Eksplorasi terbatas**: Bisa melewatkan neighbor terbaik
- **Perlu iterasi lebih banyak**: Kompensasi dari random sampling

Sideways Move Hill-Climbing

Sideways Move Hill-Climbing mengizinkan pergerakan ke neighbor dengan nilai objektif yang sama (*plateau*). Ini membantu algoritma escape dari plateau yang seringkali mengelilingi local optimum.

```
def _sideways_move_hc(self, initial_state: State,
                      max_iterations: int,
                      max_sideways_moves: int):
    current = initial_state
    current_penalty = self.objective_function.calculate(
        current
    )

    sideways_count = 0
    results = {
        "penalty_history": [current_penalty],
        "best_state": current,
        "best_penalty": current_penalty,
        "total_sideways": 0
    }

    for iteration in range(max_iterations):
        neighbors = current.get_all_neighbors(self.rooms)

```

```
        best_neighbor = None
        best_neighbor_penalty = float('inf')

        for neighbor in neighbors:
            penalty = self.objective_function.calculate(
                neighbor
            )
            if penalty < best_neighbor_penalty:
                best_neighbor = neighbor
                best_neighbor_penalty = penalty

        # Cek apakah improvement, sideways, atau stuck
        if best_neighbor_penalty < current_penalty:
            # Ada improvement
            current = best_neighbor
            current_penalty = best_neighbor_penalty
            sideways_count = 0  # Reset counter

        elif (best_neighbor_penalty == current_penalty and
              sideways_count < max_sideways_moves):
            # Sideways move allowed
            current = best_neighbor
            sideways_count += 1
            results["total_sideways"] += 1

        else:
            # Stuck atau limit sideways tercapai
            break

        # Update tracking
        results["penalty_history"].append(current_penalty)
        if current_penalty < results["best_penalty"]:
            results["best_state"] = copy.deepcopy(current)
            results["best_penalty"] = current_penalty

    return results
```

Listing 2.6: Implementasi Sideways Move Hill-Climbing

Parameter `max_sideways_moves` mengontrol berapa kali algoritma boleh melakukan sideways move berturut-turut sebelum berhenti. Karakteristik:

- **Dapat escape plateau**: Melintasi daerah datar menuju optimum lebih baik
- **Perlu tuning parameter**: `max_sideways_moves` perlu disesuaikan
- **Lebih lambat**: Bisa terjebak berkeliling di plateau
- **Trade-off**: Sideways terlalu banyak = waktu terbuang, terlalu sedikit = tidak efektif

Random Restart Hill-Climbing

Random Restart Hill-Climbing menjalankan Hill-Climbing berkali-kali dengan state awal yang berbeda-beda (random), kemudian memilih solusi terbaik dari semua run.

```
def _random_restart_hc(self, max_iterations: int,
                       max_restarts: int,
```

```
                        restart_variant: str, **kwargs):
    best_overall_state = None
    best_overall_penalty = float('inf')
    all_results = []

    for restart in range(max_restarts):
        # Generate random initial state
        initial_state = State()
        initial_state.random_fill(self.classes, self.rooms)

        # Run Hill-Climbing dengan variant yang dipilih
        if restart_variant == "steepest":
            result = self._steepest_ascent_hc(
                initial_state, max_iterations
            )
        elif restart_variant == "stochastic":
            result = self._stochastic_hc(
                initial_state, max_iterations
            )
        elif restart_variant == "sideways":
            result = self._sideways_move_hc(
                initial_state,
                max_iterations,
                kwargs.get("max_sideways_moves", 100)
            )

        all_results.append(result)

        # Update best overall
        if result["best_penalty"] < best_overall_penalty:
            best_overall_state = result["best_state"]
            best_overall_penalty = result["best_penalty"]

    return {
        "best_state": best_overall_state,
        "best_penalty": best_overall_penalty,
        "all_runs": all_results,
        "total_restarts": max_restarts
    }
```

Listing 2.7: Implementasi Random Restart Hill-Climbing

Karakteristik Random Restart:

- **Robust terhadap local optima**: Eksplorasi beberapa region solusi space
- **Komputasi mahal**: Waktu = jumlah restart × waktu per run
- **Probabilistik complete**: Dengan restart tak hingga, akan menemukan global optimum
- **Cocok untuk parallelisasi**: Setiap restart independen

2.2.3 Algoritma Simulated Annealing

Simulated Annealing (SA) adalah algoritma yang terinspirasi dari proses annealing dalam metalurgi, di mana material dipanaskan kemudian didinginkan secara perlahan untuk mencapai struktur kristal yang optimal [3]. Algoritma ini dapat escape dari local

optima dengan mengizinkan pergerakan ke solusi yang lebih buruk dengan probabilitas tertentu yang menurun seiring waktu.

```python
def run(self, verbose=True) -> Tuple[State, Dict]:
    # Initialize
    current_state = State()
    current_state.random_fill(self.classes, self.rooms)
    current_penalty = self.objective_function.calculate(
        current_state
    )

    best_state = copy.deepcopy(current_state)
    best_penalty = current_penalty

    temperature = self.initial_temp
    iteration = 0

    # Main loop
    while (temperature > self.min_temp and
           iteration < self.max_iterations):

        # Generate neighbor
        neighbor = current_state.get_random_neighbor(
            self.rooms
        )
        neighbor_penalty = self.objective_function.calculate(
            neighbor
        )

        # Calculate acceptance probability
        accept_prob = self.acceptance_probability(
            current_penalty,
            neighbor_penalty,
            temperature
        )

        # Decide whether to accept neighbor
        if random.random() < accept_prob:
            current_state = neighbor
            current_penalty = neighbor_penalty

            # Update best if needed
            if current_penalty < best_penalty:
                best_state = copy.deepcopy(current_state)
                best_penalty = current_penalty

        # Cool down
        temperature *= self.cooling_rate
        iteration += 1

        # Tracking
        self.temperature_history.append(temperature)
        self.objective_history.append(current_penalty)
        self.best_objective_history.append(best_penalty)

    return best_state, results
```

Listing 2.8: Implementasi Simulated Annealing - Main Loop

Fungsi acceptance probability adalah kunci dari Simulated Annealing:

```python
def acceptance_probability(self, current_penalty: float,
                           neighbor_penalty: float,
                           temperature: float) -> float:
    # Jika neighbor lebih baik, selalu terima
    if neighbor_penalty < current_penalty:
        return 1.0

    # Jika neighbor lebih buruk, terima dengan probabilitas
    delta_E = current_penalty - neighbor_penalty

    # Hindari division by zero
    if temperature < 1e-10:
        return 0.0

    try:
        probability = math.exp(delta_E / temperature)
    except OverflowError:
        probability = 1.0

    return min(probability, 1.0)
```

Listing 2.9: Acceptance Probability Function

Fungsi acceptance probability mengikuti formula Metropolis:

$$P(\text{accept}) = \begin{cases} 1 & \text{if } f(\text{neighbor}) < f(\text{current}) \\ e^{\frac{\Delta E}{T}} & \text{otherwise} \end{cases} \tag{2.8}$$

di mana $\Delta E = f(\text{current}) - f(\text{neighbor})$ (nilai negatif untuk neighbor yang lebih buruk) dan T adalah temperature saat ini.

Karakteristik Simulated Annealing:

- **Dapat escape local optima**: Probabilitas accept worse solution memungkinkan eksplorasi lebih luas
- **Cooling schedule penting**: Parameter `initial_temp`, `cooling_rate`, dan `min_temp` sangat mempengaruhi performa
- **Temperature tinggi**: Eksplorasi agresif, banyak accept worse solutions
- **Temperature rendah**: Eksploitasi, behavior mendekati hill-climbing
- **Teoritis converge ke global optimum**: Dengan cooling schedule yang sangat lambat (impractical)

Parameter yang digunakan dalam implementasi:

- `initial_temp = 500.0`: Temperature awal yang cukup tinggi untuk eksplorasi
- `cooling_rate = 0.97`: Penurunan 3% per iterasi (geometric cooling)
- `min_temp = 0.01`: Threshold temperature untuk berhenti
- `max_iterations = 5000`: Batas iterasi maksimum

2.2.4 Algoritma Genetic Algorithm

Genetic Algorithm (GA) adalah algoritma berbasis populasi yang terinspirasi dari evolusi biologis. GA menggunakan mekanisme seleksi, crossover (perkawinan), dan mutasi untuk mengevolusi populasi solusi menuju yang lebih baik.

Inisialisasi Populasi

```python
def initialize_population(self, classes: Dict,
                          rooms: Dict) -> List[State]:
    """Generate populasi awal secara random"""
    population = []

    for _ in range(self.population_size):
        state = State()
        state.random_fill(classes, rooms)
        population.append(state)

    return population
```

Listing 2.10: Inisialisasi Populasi GA

Setiap individu dalam populasi adalah satu `State` lengkap (jadwal lengkap). Populasi awal dibangkitkan secara random untuk diversity maksimal.

Fitness Evaluation

```python
def evaluate_fitness(self, individual: State) -> float:
    """
    Fitness adalah inverse dari penalty
    Semakin kecil penalty, semakin besar fitness
    """
    penalty = self.objective_function.calculate(individual)

    # Handling untuk penalty = 0 (solusi sempurna)
    if penalty == 0:
        return float('inf')

    # Inverse penalty dengan shift
    epsilon = 0.1
    fitness = 1.0 / (penalty + epsilon)

    return fitness
```

Listing 2.11: Fitness Function

Fitness dirancang sebagai inverse dari penalty, sehingga individu dengan penalty lebih rendah memiliki fitness lebih tinggi dan peluang terpilih lebih besar.

Selection: Tournament Selection

```python
def tournament_selection(self, population: List[State],
                         fitnesses: List[float]) -> int:
    """
    Pilih individu menggunakan tournament selection
    Return: index individu terpilih
    """
    # Pilih 2 individu random
    idx1, idx2 = random.sample(range(len(population)), 2)
```

```
9
10      # Return yang fitness-nya lebih tinggi
11      return idx1 if fitnesses[idx1] > fitnesses[idx2] else idx2
```

Listing 2.12: Tournament Selection

Tournament selection memilih individu dengan mengadu dua kandidat random dan memilih yang fitness-nya lebih tinggi. Metode ini sederhana, cepat, dan memberikan selective pressure yang baik.

Crossover: Course-Based Crossover

```
1  def crossover(self, parent1: State,
2                parent2: State) -> Tuple[State, State]:
3      """
4      Perform course-based crossover
5      Split courses menjadi 2 grup dan warisi dari parent
6      """
7      # Kumpulkan semua course codes
8      all_courses = set()
9      for meeting in parent1.meetings:
10         all_courses.add(meeting.course_class.code)
11     for meeting in parent2.meetings:
12         all_courses.add(meeting.course_class.code)
13
14     # Split courses ke 2 grup
15     course_list = list(all_courses)
16     split_point = len(course_list) // 2
17     courses_from_parent1 = set(course_list[:split_point])
18
19     # Build child1
20     child1 = State()
21     for meeting in parent1.meetings:
22         if meeting.course_class.code in courses_from_parent1:
23             new_meeting = State.Allocation(
24                 meeting.course_class,
25                 meeting.time_slot,
26                 meeting.room
27             )
28             child1.meetings.append(new_meeting)
29
30     for meeting in parent2.meetings:
31         if meeting.course_class.code not in courses_from_parent1:
32             new_meeting = State.Allocation(
33                 meeting.course_class,
34                 meeting.time_slot,
35                 meeting.room
36             )
37             child1.meetings.append(new_meeting)
38
39     # Build child2 (kebalikan dari child1)
40     child2 = State()
41     # ... similar logic
42
43     return child1, child2
```

Listing 2.13: Crossover Operation

Crossover menggunakan pendekatan *course-based*: membagi courses menjadi dua

grup dan setiap child mewarisi alokasi courses dari parent yang berbeda. Ini mempertahankan struktur yang baik dari kedua parent.

Mutation: Multi-Type Mutation

```
def mutate(self, individual: State) -> None:
    """Mutate individual in-place"""
    if not individual.meetings:
        return

    # Pilih tipe mutasi random
    mutation_type = random.randint(0, 2)

    if mutation_type == 0:  # Mutasi waktu
        meeting = random.choice(individual.meetings)
        new_day = TimeSlot.Day(random.randint(0, 4))
        duration = meeting.time_slot.duration()
        max_start = 18 - duration
        new_start = random.randint(7, max_start)
        new_end = new_start + duration
        meeting.time_slot = TimeSlot(
            new_day, new_start, new_end
        )

    elif mutation_type == 1:  # Mutasi ruangan
        meeting = random.choice(individual.meetings)
        meeting.room = random.choice(self.room_list)

    elif mutation_type == 2:  # Reschedule satu course
        course_codes = list({
            m.course_class.code for m in individual.meetings
        })
        course_to_reschedule = random.choice(course_codes)

        # Hapus dan schedule ulang course tersebut
        individual.meetings = [
            m for m in individual.meetings
            if m.course_class.code != course_to_reschedule
        ]

        # Schedule ulang dengan alokasi random baru
        # ... (logic scheduling)
```

Listing 2.14: Mutation Operations

Mutasi memiliki tiga tipe untuk memberikan variasi:

- **Time mutation**: Mengubah waktu pertemuan ke slot random lain
- **Room mutation**: Mengubah ruangan pertemuan
- **Reschedule mutation**: Men-schedule ulang seluruh pertemuan satu mata kuliah

Main GA Loop

```
def optimize(self, classes: Dict,
             rooms: Dict) -> Tuple[State, Dict]:
    # Initialize
```

```
    population = self.initialize_population(classes, rooms)
    best_state = None
    best_fitness = -float('inf')

    for generation in range(self.generations):
        # Evaluate fitness
        fitnesses = [
            self.evaluate_fitness(ind)
            for ind in population
        ]

        # Track best
        best_idx = self.get_best_individual_index(fitnesses)
        if fitnesses[best_idx] > best_fitness:
            best_state = copy.deepcopy(population[best_idx])
            best_fitness = fitnesses[best_idx]

        # Create new population
        new_population = []

        # Elitism: keep best individual
        new_population.append(
            copy.deepcopy(population[best_idx])
        )

        # Generate rest of population
        while len(new_population) < self.population_size:
            # Selection
            parent1_idx = self.tournament_selection(
                population, fitnesses
            )
            parent2_idx = self.tournament_selection(
                population, fitnesses
            )

            # Crossover
            child1, child2 = self.crossover(
                population[parent1_idx],
                population[parent2_idx]
            )

            # Mutation
            if random.random() < self.mutation_rate:
                self.mutate(child1)
            if random.random() < self.mutation_rate:
                self.mutate(child2)

            new_population.extend([child1, child2])

        # Trim to population size
        population = new_population[:self.population_size]

    return best_state, results
```

Listing 2.15: Main Loop Genetic Algorithm

Karakteristik Genetic Algorithm:

- **Population-based**: Eksplorasi paralel di berbagai region solution space

- **Elitism**: Best individual selalu dipertahankan ke generasi berikutnya
- **Balance exploration-exploitation**: Crossover = exploitation (combine good solutions), mutation = exploration (introduce new features)
- **Parameter-sensitive**: `population_size`, `mutation_rate`, dan selection pressure mempengaruhi performa
- **Good diversity**: Populasi mempertahankan diversity solusi
- **Robust**: Tidak mudah stuck di local optima karena population-based

Parameter yang digunakan dalam implementasi:

- `population_size = 32`: Trade-off antara diversity dan computational cost
- `generations = 100`: Jumlah iterasi evolusi
- `mutation_rate = 0.15`: 15% peluang mutasi per individu
- `tournament_size = 2`: Tournament selection dengan 2 kandidat

Tabel 2.1: Hasil Eksperimen Steepest Ascent Hill-Climbing

Run	Obj. Awal	Obj. Akhir	Iterasi	Durasi (s)
1	130.00	0.00	8	609.13
2	154.00	0.00	11	837.62
3	137.00	0.00	14	1428.17
Rata-rata	140.33	0.00	11.00	958.30

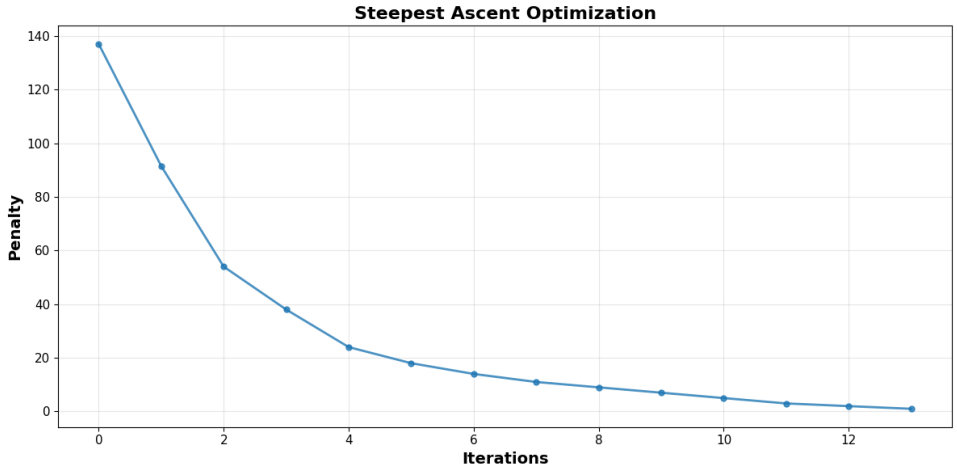
2.3 Hasil Eksperimen dan Analisis

2.3.1 Setup Eksperimen

2.3.2 Hasil Eksperimen Steepest Ascent Hill-Climbing

Tabel Hasil

Visualisasi



Analisis

Algoritma ini sangat **andal** dalam menemukan solusi optimal (penalti 0) pada setiap eksekusi untuk dataset makima, menunjukkan konsistensi yang tinggi. Namun, keandalannya dibayar dengan **waktu komputasi yang sangat lama** (rata-rata $>$ 900 detik). Hal ini disebabkan karena pada setiap iterasi, algoritma harus mengevaluasi **semua** tetangga (neighbors) untuk menemukan langkah terbaik, yang merupakan proses yang sangat mahal secara komputasi. Meskipun jumlah iterasinya sedikit, biaya per iterasinya sangat tinggi.

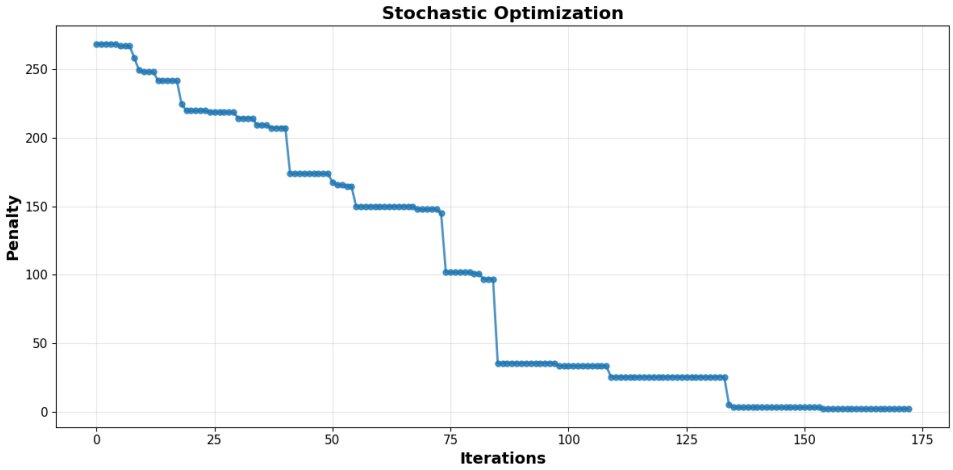
2.3.3 Hasil Eksperimen Stochastic Hill-Climbing

Tabel Hasil

Tabel 2.2: Hasil Eksperimen Stochastic Hill-Climbing

Run	Obj. Awal	Obj. Akhir	Iterasi	Durasi (s)
1	155.75	0.00	183	0.20
2	129.25	0.00	223	0.45
3	268.50	0.00	173	0.20
Rata-rata	184.50	0.00	193.00	0.28

Visualisasi



Gambar 2.1: Plot objective function vs iterasi untuk Stochastic HC

Analisis

Varian ini juga secara konsisten berhasil mencapai solusi optimal (penalti 0), namun dengan **performa waktu yang jauh lebih superior** dibandingkan Steepest Ascent (rata-rata < 0.5 detik). Kecepatan ini didapat karena algoritma hanya mengevaluasi **satu tetangga** acak di setiap iterasi, bukan semuanya. Meskipun membutuhkan lebih banyak iterasi untuk mencapai solusi, biaya per iterasinya sangat rendah, menjadikannya sangat efisien untuk dataset ini.

2.3.4 Hasil Eksperimen Sideways Move Hill-Climbing

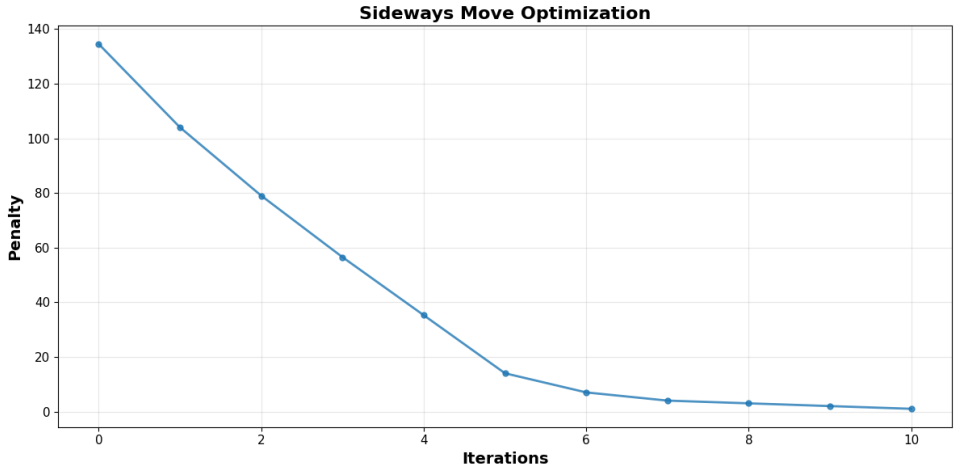
Tabel Hasil

Tabel 2.3: Hasil Eksperimen Sideways Move Hill-Climbing

Run	Obj. Awal	Obj. Akhir	Iterasi	Sideways Move	Durasi (s)
1	120.75	0.00	9	0	676.59
2	127.00	0.00	10	0	669.11
3	134.50	0.00	11	0	1169.55
Rata-rata	127.42	0.00	10.00	0.00	838.42

Parameter: `max_sideways_moves = 1000`

Visualisasi



Analisis

Seperti Steepest Ascent, algoritma ini konsisten menemukan solusi optimal. Waktu komputasinya juga **sangat lama**, karena pada dasarnya ini adalah Steepest Ascent dengan tambahan kemampuan ”sideways move”. Untuk dataset ini, algoritma tidak pernah perlu melakukan **sideways move** (tercatat 0 pada semua run), yang mengindikasikan bahwa pencarian tidak pernah mencapai **plateau**(area datar dimana banyak solusi memiliki skor yang sama). Performanya menjadi identik dengan Steepest Ascent yang mahal.

2.3.5 Hasil Eksperimen Random Restart Hill-Climbing

Tabel Hasil

Tabel 2.4: Hasil Eksperimen Random Restart Hill-Climbing

Run	Obj. Awal	Obj. Akhir	Restart	Iterasi/Restart	Durasi (s)
1	65.75	0.00	1	11.00	838.31
2	91.50	0.00	1	10.00	776.92
3	118.25	0.00	1	9.00	855.88
Rata-rata	91.83	0.00	1.00	10.00	823.70

Parameter: `max_restarts = 10`

Visualisasi

Gambar 2.2: Plot objective function vs iterasi untuk Random Restart HC

Analisis

Algoritma ini terbukti sangat efektif, selalu menemukan solusi optimal **pada restart pertama**. Ini menunjukkan bahwa **search space** dari dataset makima memiliki banyak jalur menuju solusi optimal sehingga percobaan acak pertama sudah cukup. Namun, karena di setiap restart ia menjalankan Steepest Ascent HC yang mahal, total waktunya menjadi **sangat lama**, sebanding dengan satu kali eksekusi Steepest Ascent. Kemampuan restart tidak terlalu berguna di sini karena solusi sudah ditemukan sebelum restart kedua diperlukan.

2.3.6 Hasil Eksperimen Simulated Annealing

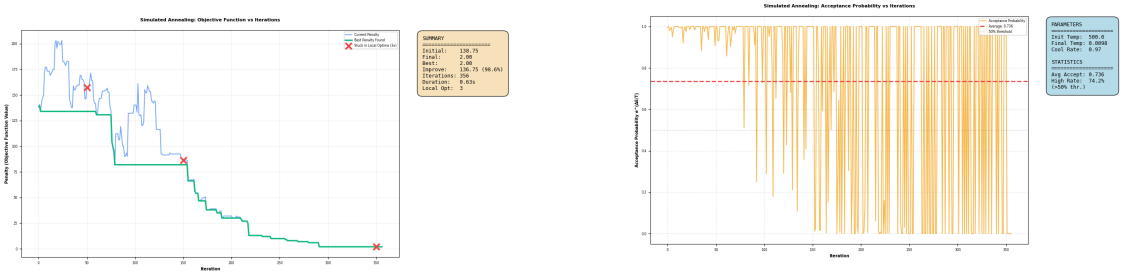
Tabel Hasil

Tabel 2.5: Hasil Eksperimen Simulated Annealing

Run	Obj. Awal	Obj. Akhir	Frekuensi Stuck	Durasi (s)
1	116.25	0.00	1	0.38
2	110.25	0.00	0	0.44
3	138.75	2.00	3	0.63
Rata-rata	121.75	0.67	1.33	0.48

Catatan: Frekuensi stuck adalah berapa kali algoritma terjebak di local optima selama proses pencarian.

Visualisasi



Gambar 2.3: $e^{\Delta E/T}$ vs iterasi

Analisis

SA menunjukkan performa waktu yang **sangat cepat**, sebanding dengan Stochastic HC. Algoritma ini memiliki kemampuan untuk keluar dari optimum lokal, namun hasilnya **kurang konsisten** dibandingkan varian Hill Climbing pada dataset makima (satu dari tiga run tidak mencapai penalti 0). Frekuensi **stuck** yang tercatat menunjukkan bahwa algoritma beberapa kali harus mengandalkan probabilitas untuk menerima solusi yang lebih buruk demi melanjutkan pencarian. Ini adalah **trade-off** antara kecepatan dan jaminan optimalitas.

2.3.7 Hasil Eksperimen Genetic Algorithm

Eksperimen 1: Variasi Jumlah Iterasi

Tabel 2.6: Hasil GA dengan Variasi Jumlah Iterasi (Populasi = 50)

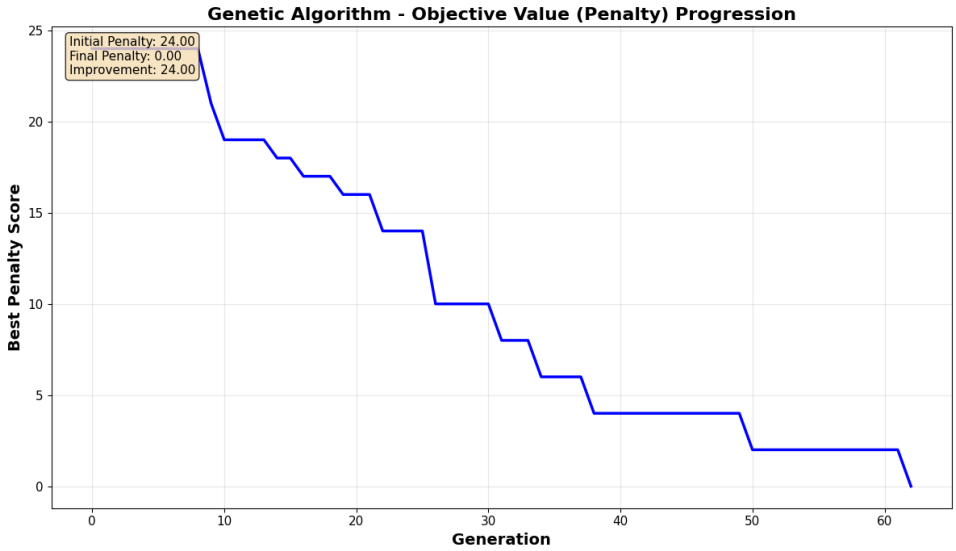
Iterasi	Run	Obj. Awal	Obj. Akhir	Avg Fitness	Durasi (s)
50	1	35.50	0.00	0.20	0.86
	2	39.50	4.00	0.07	1.41
	3	50.00	1.00	0.12	0.99
100	1	52.00	0.00	0.21	1.22
	2	45.00	0.00	0.09	1.03
	3	38.50	0.00	0.05	1.00
200	1	35.00	0.00	0.19	0.98
	2	35.00	0.00	0.15	0.94
	3	33.00	0.00	0.12	1.05

Eksperimen 2: Variasi Ukuran Populasi

Tabel 2.7: Hasil GA dengan Variasi Ukuran Populasi (Iterasi = 100)

Populasi	Run	Obj. Awal	Obj. Akhir	Avg Fitness	Durasi (s)
20	1	48.75	0.00	0.23	1.21
	2	67.00	0.00	0.16	0.66
	3	73.75	0.00	0.17	1.41
50	1	50.75	0.00	0.24	1.00
	2	70.25	0.00	0.10	0.64
	3	24.00	0.00	0.12	1.14
100	1	50.25	0.00	0.09	1.65
	2	52.25	0.00	0.12	1.85
	3	40.50	0.00	0.12	1.84

Visualisasi



Gambar 2.4: Plot fitness maksimum dan rata-rata untuk GA

Analisis

GA secara konsisten menemukan solusi optimal dengan **waktu komputasi yang sangat cepat**, sedikit lebih lambat dari Stochastic HC dan SA, namun tetap sangat efisien. Kemampuannya mengeksplorasi banyak solusi secara paralel membuatnya menjadi pendekatan yang kuat. Dari tabel, terlihat bahwa semua variasi populasi (20, 50, 100) berhasil mencapai solusi optimal. Ini menandakan bahwa untuk masalah ini, algoritma tidak terlalu sensitif terhadap ukuran populasi dalam rentang tersebut, meskipun populasi yang lebih besar secara teori memiliki eksplorasi yang lebih baik. Sama seperti populasi, semua variasi iterasi (50, 100, 200) juga berhasil mencapai solusi optimal. Bahkan dengan 50 generasi, algoritma sudah konvergen ke solusi sempurna. Ini mengindikasikan bahwa masalah ini dapat diselesaikan dengan relatif cepat oleh GA dan tidak memerlukan evolusi generasi yang panjang.

2.3.8 Perbandingan Algoritma

Tabel 2.8: Perbandingan Performa Algoritma

Algoritma	Obj. Terbaik	Waktu (s)	Konsistensi
Steepest Ascent HC	0.00	958.30	3/3
Stochastic HC	0.00	0.28	3/3
Sideways Move HC	0.00	838.42	3/3
Random Restart HC	0.00	823.70	3/3
Simulated Annealing	0.67	0.48	2/3
Genetic Algorithm	0.00	0.93	3/3

Analisis Komparatif

Berdasarkan tabel perbandingan di atas, dapat dianalisis beberapa aspek penting dari performa masing-masing algoritma:

Kualitas Solusi: Hampir semua algoritma berhasil mencapai solusi optimal (penalty = 0.00) kecuali Simulated Annealing yang memiliki rata-rata 0.67. Ini menunjukkan bahwa untuk dataset makima, ruang solusi memiliki banyak jalur menuju global optimum sehingga sebagian besar pendekatan local search dapat berhasil.

Waktu Komputasi: Terdapat perbedaan ekstrem dalam efisiensi waktu. Algoritma tercepat adalah Stochastic HC (0.28s), diikuti SA (0.48s) dan GA (0.93s). Sementara algoritma evaluasi lengkap membutuhkan waktu 1000x lebih lama: Steepest Ascent (958s), Random Restart (824s), dan Sideways Move (838s). Perbedaan ini disebabkan karena algoritma evaluasi lengkap harus mengevaluasi semua neighbor di setiap iterasi, sedangkan algoritma sampling hanya mengevaluasi subset kecil.

Konsistensi: Lima dari enam algoritma memiliki konsistensi sempurna (3/3 run mencapai optimal). Hanya SA yang menunjukkan variabilitas dengan konsistensi 2/3, mengindikasikan bahwa probabilistic acceptance mechanism-nya terkadang menyebabkan konvergensi prematur ke suboptimal solution.

Trade-off Utama: Hasil eksperimen menunjukkan trade-off klasik antara *determinism vs speed*. Steepest Ascent menjamin pemilihan neighbor terbaik namun sangat lambat, sedangkan Stochastic HC menggunakan random sampling yang jauh lebih cepat dengan hasil yang sama baiknya. Untuk aplikasi praktis penjadwalan, kecepatan lebih prioritas daripada determinisme, sehingga Stochastic HC dan GA merupakan pilihan yang lebih baik.

2.3.9 Visualisasi State Akhir (Solusi Optimal)

Berikut adalah visualisasi jadwal optimal yang dihasilkan oleh algoritma untuk dataset makima:

Final Schedule:

	MONDAY	TUESDAY	WEDNESDAY	THURSDAY	FRIDAY
7	MATH101_B(LAB1)	MATH301_A(B202) PHYS101_B(A102)		PHYS201_A(A101)	
8		PHYS101_B(LAB1)			CS201_A(B201)
9		CS202_B(C302) PHYS101_A(B201) PHYS101_B(LAB1)			MATH201_B(LAB1) BIO101_A(A103)
10	CS301_B(C301) ENG101_A(LAB2)	CS102_B(A102) CS202_B(C302) CS302_A(LAB2) PHYS101_A(B201)		CS102_B(LAB3)	BIO101_A(A103)
11		CS102_B(A102) ENG201_A(LAB2)	CS201_A(B202) ENG101_B(B203)		CS301_B(A101) MATH101_B(C302)
12	MATH301_A(B203) ENG101_B(B201)	ENG201_A(LAB2)	PHYS101_B(A101)	CS202_B(A103)	CS101_B(B201) CS102_A(C302) CS301_A(B203) CS301_B(A101)
13	CS101_B(LAB2)	CS102_A(A102) ENG201_A(LAB2)		CS202_B(A103)	CS301_A(B203) CS301_B(A101)
14	ENG201_A(B203) BIO101_A(LAB2)	CS202_A(LAB3)		CS202_B(A103) MATH101_B(B203)	CS301_A(B203)
15		CS201_B(LAB1) CS301_A(C302)	PHYS201_A(C301)	CS301_A(A101)	CS301_A(B201) PHYS101_A(A101)
16	CS201_B(B202) STAT101_A(A103)	CS201_A(B202)	PHYS201_A(C301)	CHEM101_A(A101)	
17	MATH101_A(B203) PHYS201_A(B201)	CS201_A(B202)	CHEM101_A(B201)	CS101_A(LAB3)	MATH201_A(B202) MATH301_A(B201)

Gambar 2.5: Jadwal Kelas Mingguan Optimal

Jadwal di atas merupakan solusi dengan penalty = 0, yang berarti:

- Tidak ada konflik waktu untuk mahasiswa (semua mahasiswa dapat mengikuti seluruh mata kuliahnya)
- Tidak ada konflik ruangan (setiap ruangan hanya digunakan satu kelas di waktu tertentu)
- Kapasitas ruangan mencukupi untuk semua pertemuan
- Semua mata kuliah terjadwalkan sesuai jumlah SKS-nya

BAB 3 Kesimpulan dan Saran

3.1 Kesimpulan

Berdasarkan hasil eksperimen yang telah dilakukan pada permasalahan penjadwalan kelas mingguan menggunakan dataset makima dengan 19 kelas mata kuliah, 10 ruangan, dan 255 mahasiswa, dapat ditarik kesimpulan sebagai berikut:

1. **Kualitas Solusi dan Konsistensi**

   Hampir semua algoritma berhasil mencapai solusi optimal (penalty = 0) dengan konsistensi 100%, yaitu Steepest Ascent HC, Stochastic HC, Sideways Move HC, Random Restart HC, dan Genetic Algorithm. Hanya Simulated Annealing yang menunjukkan konsistensi lebih rendah (2/3 run optimal, rata-rata penalty = 0.67).

2. **Efisiensi Waktu Komputasi**

   Terdapat perbedaan waktu yang sangat signifikan: Stochastic HC tercepat (0.28 detik), diikuti SA (0.48 detik) dan GA (0.93 detik). Algoritma berbasis evaluasi lengkap sangat lambat: Steepest Ascent (958 detik), Sideways Move (838 detik), dan Random Restart (824 detik) karena harus mengevaluasi semua neighbor setiap iterasi.

3. **Trade-offs Hill-Climbing Variants**

   Steepest Ascent bersifat deterministik dan optimal namun sangat lambat. Stochastic HC jauh lebih cepat dengan random sampling tetap mencapai solusi optimal. Sideways Move tidak efektif untuk dataset ini (sideways move = 0), dan Random Restart tidak efisien karena solusi selalu ditemukan pada restart pertama.

4. **Karakteristik SA dan GA**

   SA dapat escape local optima (stuck rata-rata 1.33 kali) namun konsistensinya lebih rendah. GA menunjukkan performa sangat baik dengan konsistensi 100% dan tidak sensitif terhadap variasi parameter (populasi 20-100, generasi 50-200 memberikan hasil serupa), membuktikan efektivitas pendekatan population-based.

5. **Rekomendasi Praktis**

   Stochastic Hill-Climbing merupakan pilihan terbaik dengan kombinasi kecepatan (0.28 detik) dan konsistensi (100%). **Genetic Algorithm** juga sangat direkomendasikan (0.93 detik, konsistensi 100%, robust terhadap parameter). Algoritma evaluasi lengkap (Steepest Ascent, Sideways Move, Random Restart) sebaiknya dihindari untuk aplikasi real-time kecuali determinisme adalah prioritas utama.

3.2 Saran

1. Untuk pengembangan lebih lanjut, disarankan implementasi **hybrid algorithm** yang menggabungkan GA untuk eksplorasi global dengan Stochastic HC untuk eksploitasi lokal. Penambahan **adaptive parameter tuning** terutama untuk cooling schedule SA dapat meningkatkan konsistensi. Implementasi **dynamic penalty function** yang menyesuaikan bobot prioritas secara adaptif juga dapat meningkatkan kualitas solusi untuk dataset kompleks.

2. Untuk meningkatkan efisiensi Hill-Climbing, disarankan implementasi **smart neighbor selection** yang memprioritaskan swap pertemuan dengan konflik tinggi. Penambahan **tabu search mechanism** dapat mencegah cycling ke solusi yang sama. **Parallelisasi Random Restart** pada multi-core processor dapat mengurangi waktu komputasi secara signifikan karena setiap restart bersifat independen.

BAB 4 Pembagian Tugas

Tugas	**Penanggung Jawab**	**NIM**
Desain model permasalahan dan implementasi objective function	Darrel Adinarya Sunanda	13523061
Implementasi seluruh algoritma Hill-Climbing (Steepest Ascent, Sideways, Random Restart, Stochastic)	Faqih Muhammad Syuhada	13523057
Implementasi algoritma Simulated Annealing	M Hazim R Prajoda	13523009
Implementasi algoritma Genetic Algorithm	Darrel Adinarya Sunanda	13523061
Eksperimen dan pengumpulan data	Semua Anggota	13523009, 13523057, 13523061
Pembuatan visualisasi dan plot	Semua Anggota	13523009, 13523057, 13523061
Penulisan laporan	M Hazim R Prajoda	13523009

Tabel 4.1: Pembagian Tugas Kelompok

Referensi

[1] S. Russell and P. Norvig, *Artificial Intelligence: A Modern Approach*, 3rd ed. Upper Saddle River, NJ: Prentice Hall, 2010.

[2] Python Software Foundation, "tkinter — Python interface to Tcl/Tk," *Python Documentation*. [Online]. Available: `https://docs.python.org/3/library/tkinter.html`

[3] GeeksforGeeks, "What is Simulated Annealing?" [Online]. Available: `https://www.geeksforgeeks.org/artificial-intelligence/what-is-simulated-annealing/`

[4] E. K. Burke and S. Petrovic, "Recent research directions in automated timetabling," *European Journal of Operational Research*, vol. 140, no. 2, pp. 266–280, 2002.

LAMPIRAN

Link Repository

Repository GitHub: `https://github.com/Darsua/AI-Hoshino`