

Laporan Tugas Besar 1 IF3170 Intelegensi Artifisial
Semester I Tahun Akademik 2025/2026
Penyelesaian Konflik Penjadwalan Mata Kuliah dengan Local
Search



Disusun oleh:

Farrell Jabaar Altafataza	10122057
Raudah Yahya Kuddah	13122003
Muhammad Syarafi Akmal	13522076

Sekolah Teknik Elektro dan Informatika Institut Teknologi
Bandung 2024

Deskripsi Persoalan

Permasalahan yang diangkat dalam Tugas Besar 1 IF3170 Intelegensi Artifisial adalah penjadwalan kelas mingguan untuk sejumlah mata kuliah pada suatu institusi pendidikan. Setiap mata kuliah memiliki beberapa atribut seperti kode kelas, jumlah mahasiswa, serta jumlah SKS, dan harus dijadwalkan pada waktu serta ruangan yang sesuai. Selain itu, setiap ruangan memiliki kapasitas tertentu, sedangkan setiap mahasiswa memiliki daftar mata kuliah yang diambil beserta urutan prioritasnya.

Tujuan utama dari permasalahan ini adalah mencari penjadwalan yang optimal dengan meminimalkan konflik atau ketidaksesuaian yang muncul, seperti:

1. Pertemuan kelas yang bertabrakan bagi mahasiswa yang sama.
2. Dua atau lebih kelas yang dijadwalkan pada ruangan dan waktu yang sama.
3. Ketidaksesuaian kapasitas ruangan terhadap jumlah mahasiswa yang hadir.

Masalah ini termasuk dalam kategori optimization problem, di mana ruang solusi sangat besar dan tidak mungkin dicari secara brute force. Oleh karena itu, pendekatan yang digunakan adalah algoritma local search, yaitu metode pencarian solusi yang berfokus pada perbaikan bertahap dari solusi awal (initial state) menuju solusi yang lebih baik melalui proses iteratif.

Dalam tugas ini, kami diminta untuk mengimplementasikan dan membandingkan tiga algoritma local search utama, yaitu:

1. Hill Climbing Steepest Ascent,
2. Simulated Annealing, dan
3. Genetic Algorithm.

Setiap algoritma akan diuji dalam beberapa eksperimen untuk menganalisis performa, kestabilan hasil, serta kedekatannya terhadap solusi optimal (global optimum). Selain itu, kami juga harus memvisualisasikan hasil pencarian, seperti nilai *objective function* terhadap jumlah iterasi, durasi proses, dan kondisi awal serta akhir dari jadwal yang dihasilkan.

Pembahasan

1. Pemilihan Fungsi Objektif

Dalam permasalahan penjadwalan kelas mingguan, fungsi objektif digunakan untuk mengukur seberapa baik suatu *state* atau solusi memenuhi kriteria penjadwalan yang diinginkan. Nilai fungsi objektif menentukan kualitas suatu solusi, di mana semakin kecil nilainya maka semakin sedikit konflik atau pelanggaran terhadap aturan penjadwalan yang terjadi. Oleh karena itu, rancangan fungsi objektif harus mencerminkan seluruh aspek penting dari permasalahan agar algoritma *local search* dapat melakukan pencarian secara efektif.

Pada tugas ini, fungsi objektif yang digunakan mempertimbangkan tiga komponen utama sebagaimana dijelaskan pada spesifikasi tugas besar, yaitu:

1. Konflik Jadwal Mahasiswa

Komponen ini menghitung jumlah bentrokan jadwal yang dialami oleh mahasiswa. Jika seorang mahasiswa memiliki dua mata kuliah yang dijadwalkan pada waktu yang sama, maka setiap bentrokan tersebut menambah penalti terhadap nilai fungsi objektif.

$$f1 = \sum_m \text{jumlah_bentrokan}(m) * 2$$

2. Konflik Jadwal dan Ruangan Kelas Mata Kuliah

Komponen kedua menghitung jumlah pertemuan yang dijadwalkan pada ruangan dan waktu yang sama. Jika terdapat dua atau lebih mata kuliah yang menggunakan ruangan yang sama di waktu yang sama, maka konflik ini dihitung dan diberi bobot berdasarkan prioritas mata kuliah peserta yang terlibat.

$$f2 = \sum_j (\text{durasi_bentrokan_mata_kuliah}(j) * \sum_{p,m} (\text{bobot_prioritas}(p) * \text{jumlah_mahasiswa}(m)))$$

3. Konflik Kapasitas Ruangan

Komponen ini memperhitungkan selisih antara jumlah mahasiswa dalam suatu kelas dengan kapasitas maksimum ruangan tempat kelas tersebut diadakan. Jika jumlah mahasiswa melebihi kapasitas ruangan, maka penalti diberikan secara proporsional terhadap kelebihan tersebut dikalikan dengan jumlah SKS pertemuan.

$$f3 = \sum_{k,p} (jumlah_pendaftar(p) - kapasitas_ruangan(k)) * 2$$

Ketiga komponen tersebut ditotalkan untuk menjadi fungsi objektif menyeluruh sebagai berikut:

$$F = f1 + f2 + f3$$

2. Implementasi Algoritma Local Search

Pada tugas besar ini, algoritma *Local Search* digunakan untuk mencari solusi optimal dari permasalahan penjadwalan kelas mingguan. Pendekatan ini dimulai dari solusi awal acak dan memperbaikinya secara bertahap melalui perubahan kecil pada solusi. Tiga algoritma yang diimplementasikan adalah Hill Climbing, Simulated Annealing, dan Genetic Algorithm, yang masing-masing memiliki strategi pencarian berbeda untuk membandingkan efektivitasnya dalam menemukan jadwal terbaik.

1. Parent Algorithm

Funksi	Deskripsi	Source Code
Init()	Funksi ini bertujuan untuk inisiasi kelas algoritma berupa input, seed (reproducibility), konversi input ke state, dan data-data lainnya.	<pre>class ParentAlgorithm: def __init__(self, input, seed=None): """ Inisialisasi kelas parent algoritma dengan state awal dan data mahasiswa dan juga inisialisasi state awal """ input: dict -> {"kelas_mata_kuliah": list of dict, "ruangan": list of dict, "mahasiswa": list of dict} kelas_mata_kuliah: list of dict -> {"kode": str, "jumlah_mhs": int, "sks": int, "random_seed": int} ruangan: list of dict -> {"kode": str, "kuota": int} mahasiswa: list of dict -> {"nim": str, "daftar_mk": list of str, "prioritas": list of int} NOTE: prioritas posisinya respective terhadap daftar_mk [1 itu prioritas tertinggi, dst] """ self.mahasiswa = input["mahasiswa"] self.ruangan = input["ruangan"] if seed: random.seed(seed) seed_start = random.randint(0, 100) for mata_kuliah in input["kelas_mata_kuliah"]: mata_kuliah["random_seed"] = seed_start seed_start += 1 # self.dosen = input["dosen"] buat bonus self.convert_input_to_state(input) self.obj.history = [] self.iteration = 0</pre>
convert_input_to_state()	Funksi ini bertujuan untuk translasi input ke state yaitu List[Kelas] (Kelas 'Kelas' tertera di baris paling bawah tabel)	<pre>def convert_input_to_state(self, input): import random if not input["kelas_mata_kuliah"] or not input["ruangan"]: raise ValueError("Input tidak valid: kelas_mata_kuliah atau ruangan kosong") self.state = list[kelas] = [] for i in range(len(input["kelas_mata_kuliah"])): # random.seed(input["kelas_mata_kuliah"][i]["random_seed"]) if input["kelas_mata_kuliah"][i]["sks"] > 2: kelas_semi_1 = kelas(mata_kuliah={ "kode": input["kelas_mata_kuliah"][i]["kode"] + ".51", "jumlah_mhs": input["kelas_mata_kuliah"][i]["jumlah_mhs"] // 2, "sks": 2, "random_seed": input["kelas_mata_kuliah"][i]["random_seed"] }, ruangan=input["ruangan"][random.randint(0, len(input["ruangan"])-1)]) kelas_semi_2 = kelas(mata_kuliah={ "kode": input["kelas_mata_kuliah"][i]["kode"] + ".52", "jumlah_mhs": input["kelas_mata_kuliah"][i]["jumlah_mhs"] - (input["kelas_mata_kuliah"][i]["jumlah_mhs"] // 2), "sks": input["kelas_mata_kuliah"][i]["sks"] - 2, "random_seed": input["kelas_mata_kuliah"][i]["random_seed"] + 1 }, ruangan=input["ruangan"][random.randint(0, len(input["ruangan"])-1)]) self.state.append(kelas_semi_1) self.state.append(kelas_semi_2) else: kelas = kelas(mata_kuliah=input["kelas_mata_kuliah"][i], ruangan=input["ruangan"][random.randint(0, len(input["ruangan"])-1)]) self.state.append(kelas)</pre>

get_semua_neighbor()	Fungsi ini mengembalikan semua kemungkinan neighbor state yang berbentuk List[List[Kelas]]	<pre> def get_semua_neighbor(self): """ Fungsi untuk mendapatkan semua neighbor dari state saat ini: list[list[kelas]] State dan Neighbor: list[kelas] Neighbor di-generate dengan dua cara: tukar_jadwal() dan pindah_jadwal(), keduanya akan digunakan untuk akumulasi neighbor """ import copy neighbor_tukar = [] neighbor_pindah = [] # Tukar Jadwal for i in range(len(self.state)): for j in range(i+1, len(self.state)): new_state = copy.deepcopy(self.state) self.tukar_jadwal(new_state[i], new_state[j]) # print("Tukar Jadwal Kelas", new_state[i].mata_kuliah['kode'], "dan", new_state[j].mata_kuliah['kode']) neighbor_tukar.append(new_state) # Pindah Jadwal hari = ['Senin', 'Selasa', 'Rabu', 'Kamis', 'Jumat'] jam_mulai_range = list(range(7, 16)) # jam 7 - 15 for i in range(len(self.state)): for h in hari: for j in jam_mulai_range: for r in self.ruangan: new_state = copy.deepcopy(self.state) jadwal_baru = { "jadwal_mulai": [h, j], "jadwal_selesai": [h, j + new_state[i].mata_kuliah['sks']] } if r['kode'] in new_state[i].ruangan['kode']: new_state[i].ruangan = r if not self.cek_konflik_jadwal(new_state[i], jadwal_baru): self.pindah_jadwal(new_state[i], jadwal_baru) neighbor_pindah.append(new_state) return neighbor_tukar + neighbor_pindah </pre>
get_random_n_neighbor()	Fungsi ini mengembalikan satu random neighbor dengan value lebih baik di antara metode tukar dan pindah	<pre> def get_random_neighbor(self): """ Fungsi untuk mendapatkan satu neighbor secara acak dari state saat ini: List[Kelas] """ neighbor_tukar = copy.deepcopy(self.state) neighbor_pindah = copy.deepcopy(self.state) # print(len(neighbor_tukar), "length neighbor") i, j = random.sample(range(len(neighbor_tukar)), 2) # print(i, j, "tukar") self.tukar_jadwal(neighbor_tukar[i], neighbor_tukar[j]) hari = ['Senin', 'Selasa', 'Rabu', 'Kamis', 'Jumat'] jam_mulai_range = list(range(7, 16)) # jam 7 - 15 i = random.randint(0, len(neighbor_pindah)-1) h = random.choice(hari) j = random.choice(jam_mulai_range) # print(i, j, "pindah") jadwal_baru = { "jadwal_mulai": [h, j], "jadwal_selesai": [h, j + neighbor_pindah[i].mata_kuliah['sks']] } while self.cek_konflik_jadwal(neighbor_pindah[i], jadwal_baru): h = random.choice(hari) j = random.choice(jam_mulai_range) jadwal_baru = { "jadwal_mulai": [h, j], "jadwal_selesai": [h, j + neighbor_pindah[i].mata_kuliah['sks']] } self.pindah_jadwal(neighbor_pindah[i], jadwal_baru) return max([neighbor_pindah, neighbor_tukar], key=lambda s: self.fungsi_objektif(s)) </pre>
tukar_jadwal()	Fungsi ini bertujuan untuk menukar jadwal (posisi pertemuan) dua mata kuliah (excluding ruangan)	<pre> def tukar_jadwal(self, kelas1: Kelas, kelas2: Kelas): """ Fungsi untuk menukar jadwal dua kelas [menukar_jadwal: dict -> ('jadwal_mulai': [hari, jam_mulai], 'jadwal_selesai': [hari, jam_mulai + sks])] """ jadwal_kelas1 = copy.deepcopy(kelas1.jadwal) jadwal_kelas2 = copy.deepcopy(kelas2.jadwal) kelas1.jadwal["jadwal_mulai"], kelas1.jadwal["jadwal_selesai"] = jadwal_kelas2["jadwal_mulai"], [jadwal_kelas2["jadwal_selesai"][0], jadwal_kelas2["jadwal_mulai"][1] + kelas1.mata_kuliah['sks']] kelas2.jadwal["jadwal_mulai"], kelas2.jadwal["jadwal_selesai"] = jadwal_kelas1["jadwal_mulai"], [jadwal_kelas1["jadwal_selesai"][0], jadwal_kelas1["jadwal_mulai"][1] + kelas2.mata_kuliah['sks']] </pre>
pindah_jadwal()	Fungsi ini bertujuan untuk memindahkan jadwal dan ruangan ke posisi yang tidak berkonflik	<pre> def pindah_jadwal(self, kelas: Kelas, jadwal_baru): """ Fungsi untuk memindahkan jadwal kelas ke jadwal baru jika tidak ada konflik jadwal ke """ if not self.cek_konflik_jadwal(kelas, jadwal_baru): kelas.jadwal = jadwal_baru # debug purposes # print("Pindah Jadwal Kelas", kelas.mata_kuliah['kode'], "ke", jadwal_baru) </pre>

cek_konflik_jadwal()	Fungsi ini bertujuan untuk melihat apakah dua kelas bertabrakan atau tidak	<pre>def cek_konflik_jadwal(self, kelas: Kelas, jadwal_baru): """ Fungsi untuk mengecek apakah jadwal baru konflik dengan jadwal kelas lain (jadwal tabrakan dan ruangan sama) """ konflik = False for kis in self.state: if kis.jadwal['jadwal_mulai'][0] == jadwal_baru['jadwal_mulai'][0] and max(kis.jadwal['jadwal_mulai'][1], konflik = True # debug purposes # print("Konflik Jadwal Kelas, kelas.mata_kuliah['kode'], 'di', jadwal_baru, 'ruangan:', kelas.ruang) break return konflik</pre>
fungsi_objektif()	Fungsi ini bertujuan untuk melakukan kalkulasi terhadap nilai objektif	<pre>def fungsi_objektif(self, state=None, verbose=False): if state is None: state = self.state obj_jadwal_kelas = 0 obj_kuota_kelas = 0 obj_jadwal_mahasiswa = 0 # Cek jadwal kelas for i in range(len(state)): for j in range(1, len(state)): if state[i].jadwal['jadwal_mulai'][0] == state[j].jadwal['jadwal_mulai'][0] and state[i].jadwal['ja # Mata Kuliah saling Tabrakan Jadwal # debug purposes durasi_tabrakan = min(state[i].jadwal['jadwal_selesai'][1], state[j].jadwal['jadwal_selesai'][1]) - list_bobot = self.get_mahasiswa_prioritas_by_kelas(state[i].mata_kuliah['kode']) list_bobot += self.get_mahasiswa_prioritas_by_kelas(state[j].mata_kuliah['kode']) bobot = 0 # if verbose: # print("\nTabrakan") # state[i].get_info() # print("-----") # state[j].get_info() # print("-----") # print(f"Durasi Tabrakan: {durasi_tabrakan} jam") # print(f"Prioritas Mahasiswa di kelas {state[i].mata_kuliah['kode']}: {list_bobot}") # print("-----\n") for prioritas in list_bobot: if prioritas == 1: bobot += 1.75 elif prioritas == 2: bobot += 1.5 elif prioritas == 3: bobot += 1.25 else: bobot += 1 obj_jadwal_kelas += durasi_tabrakan * bobot def fungsi_kapasitas(self, state=None, verbose=False): kuota = state[0].ruangan['kuota'] jumlah_ams_dartar = state[0].mata_kuliah['jumlah_ams_1'] if jumlah_ams_dartar > kuota: # Ruangan Kapasitas Penuhan # debug purposes # print(f"Kapasitas Ruangan Kelas {state[0].mata_kuliah['kode']}: Kuota {kuota}, Jumlah Mahasiswa {jumlah_ams_dartar}") obj_kuota_kelas = (jumlah_ams_dartar - kuota) * 2 # Cek Jadwal Mahasiswa for mhs in self.mahasiswa: daftar_ams = mhs.daftar_ams kelas = [] for i in daftar_ams: kelas = self.get_kelas_by_kode(kode, state) for i in range(len(kelas)): for j in range(1, len(kelas)): if kelas[i].jadwal['jadwal_mulai'][0] == kelas[j].jadwal['jadwal_mulai'][0] and kelas[i].jadwal['jadwal_selesai'][1] > min(# debug purposes # if verbose: # print(f"Tabrakan Jadwal {kelas[i].mata_kuliah['kode']} dan {kelas[j].mata_kuliah['kode']}") obj_jadwal_mahasiswa += 1 # debug purposes if verbose: print("Objektif", obj_jadwal_mahasiswa + obj_kuota_kelas + obj_jadwal_kelas) print("Kapasitas Jadwal Kelas:", obj_jadwal_kelas) print("Kapasitas Kuota Kelas:", obj_kuota_kelas) print("Objektif Jadwal Mahasiswa:", obj_jadwal_mahasiswa) return obj_jadwal_mahasiswa + obj_kuota_kelas + obj_jadwal_kelas</pre>
get_kelas_by_kode()	Fungsi ini bertujuan untuk mengembalikan suatu Kelas dengan kode mata kuliah tertentu	<pre>def get_kelas_by_kode(self, kode, state=None): if state is None: state = self.state result = [] for kelas in state: if kode in kelas.mata_kuliah['kode']: result.append(kelas) if result: return result return None</pre>
get_mahasiswa_a_prioritas_by_kelas()	Fungsi ini bertujuan untuk mengembalikan bobot prioritas setiap mahasiswa yang mengikuti	<pre>def get_mahasiswa_prioritas_by_kelas(self, kode_kelas): result = [] for mhs in self.mahasiswa: for i, mk in enumerate(mhs['daftar_mk']): if mk in kode_kelas: result.append(mhs['prioritas'][i]) break return result</pre>

2. Hill Climbing Steepest Ascent

Funksi	Deskripsi	Source Code
init()	Funksi ini bertujuan untuk memanggil <i>constructor</i> dari kelas induknya (ParentAlgorithm)	<pre>class HC_SA(ParentAlgorithm): def __init__(self, input): super().__init__(input)</pre>
run()	<p>Funksi ini bertujuan untuk mengeksekusi algoritma <i>Steepest Hill Climbing</i>. Funksi ini memulai dengan state awal yang acak dan menghitung skor awal menggunakan fungsi objektif. Lalu, fungsi memasuki loop untuk memperbaiki jadwal. Pada setiap iterasi, fungsi membangkitkan seluruh kemungkinan state <i>neighbor</i> yang dapat dicapai pada saat itu. Setiap state <i>neighbor</i> dievaluasi skornya menggunakan fungsi objektif. Funksi ini akan memilih state <i>neighbor</i> dengan nilai objektif tertinggi sebagai kandidat <i>neighbor</i> terbaik. Apabila</p>	<pre>def run(self, verbose=False): current_state = self.state current_score = self.fungsi_objektif(current_state) self.visualize_state("HC_Awal.") if verbose: print(f"Initial score: {current_score:.4f}") iteration = 0 history = [current_score] while True: iteration += 1 if verbose: print(f"Iterasi ke-{iteration}") neighbors = self.get_semua_neighbor() if not neighbors: if verbose: print("Tidak ada neighbor yang tersedia.") break best_neighbor = max(neighbors, key=lambda s: self.fungsi_objektif(s)) best_neighbor_score = self.fungsi_objektif(best_neighbor) if verbose: print(f"Skor saat ini : {current_score:.4f}, Skor neighbor terbaik: {best_neighbor_score:.4f}") if best_neighbor_score > current_score: current_state = best_neighbor current_score = best_neighbor_score history.append(current_score) if verbose: print(f"Ditemukan state yang lebih baik dengan skor baru: {current_score:.4f}") else: if verbose: print("Tidak ada perbaikan, telah mencapai lokal optimum.") break self.state = current_state self.visualize_state("HC_Akhir.") return self.state, current_score</pre>

	<p>nilai kandidat ini melampaui nilai state saat ini, maka state akan digantikan oleh kandidat tersebut. Namun, jika tidak ada satupun state <i>neighbor</i> yang memiliki nilai lebih tinggi, maka algoritma telah mencapai kondisi <i>local optimum</i>.</p>	
--	--	--

3. Simulated Annealing

Funksi	Deskripsi	Source Code
<code>__init__()</code>	<p>Fungsi ini digunakan untuk melakukan inisialisasi awal algoritma. Inisialisasi mencakup data <i>input</i>, temperatur awal, laju pendinginan, temperatur minimum, serta menentukan <i>state</i> awal secara acak.</p> <p>Fungsi ini juga menghitung skor awal dari <i>state</i> yang dipilih dan menyimpannya sebagai <i>best state</i> sementara.</p>	<pre> class SimulatedAnnealing(ParentAlgorithm): def __init__(self, data, initial_temp=1000, cooling_rate=0.01, min_temp=1e-3): super().__init__(data) self.input = data self.initial_temp = initial_temp self.cooling_rate = cooling_rate self.min_temp = min_temp self.current_state = self.get_random_neighbor() self.best_state = self.current_state self.best_score = self.fungsi_objektif(self.current_state) </pre>
<code>run()</code>	<p>Fungsi utama untuk menjalankan proses optimasi. Pada setiap iterasi, algoritma mencari <i>neighbor</i> baru dari <i>state</i> saat ini, menghitung perubahan nilai objektif (delta), dan memutuskan apakah neighbor tersebut diterima berdasarkan probabilitas eksponensial yang dipengaruhi oleh temperatur (T.) Temperatur kemudian dikurangi sesuai <i>cooling rate</i>. Proses berhenti saat T turun di bawah <i>min_temp</i>.</p>	<pre> def run(self, verbose=False): current_state = self.current_state current_score = self.best_score T = self.initial_temp if verbose: print(f"Initial score: {current_score:.4f}") while T > self.min_temp: neighbor = self.get_random_neighbor() neighbor_score = self.fungsi_objektif(neighbor) delta = neighbor_score - current_score if delta > 0: current_state = neighbor current_score = neighbor_score else: if random.random() < math.exp(delta / T): current_state = neighbor current_score = neighbor_score if current_score > self.best_score: self.best_state = current_state self.best_score = current_score T *= (1 - self.cooling_rate) if verbose: print(f"T={T:.4f} Current Score={current_score:.2f} Best={self.best_score:.2f}") if verbose: print(f"Final best score: {self.best_score:.4f}") return self.best_state, self.best_score </pre>

4. Genetic Algorithm

Fungsi	Deskripsi	Source Code
init()	Fungsi ini bertujuan untuk inialisasi populasi state random dan data keperluan lainnya.	<pre> class GeneticAlgorithm(ParentAlgorithm): def __init__(self, input, population_size=100, n_generasi=10): super().__init__(input) self.population_size = population_size self.n_generasi = n_generasi self.input = input self.hari = ['Senin', 'Selasa', 'Rabu', 'Kamis', 'Jumat'] self.range_waktu = list(range(7, 16)) self.inisialisasi_populasi() self.avg_obj_history = [] </pre>
inisialisasi_populasi()	Fungsi ini bertujuan untuk melakukan inialisasi populasi dengan konfigurasi acak	<pre> def inisialisasi_populasi(self): population = [] for _ in range(self.population_size - 1): new_state = copy.deepcopy(self.state) for kelas in new_state: kelas.randomize_jadwal(seed=random.randint(0, 10000)) population.append(new_state) self.population = sorted(population, key=lambda ind: self.fitness(ind), reverse=True) </pre>
crossover()	Fungsi ini bertujuan untuk mengembalikan children dari hasil crossover 2 parent	<pre> def crossover(self, parent1, parent2): child1 = copy.deepcopy(parent1) child2 = copy.deepcopy(parent2) num_crossover_points = random.randint(2, 4) crossover_points = sorted(random.sample(range(1, len(parent1)), num_crossover_points)) swap = False prev_point = 0 for point in crossover_points + [len(parent1)]: if swap: for i in range(prev_point, point): child1[i].jadwal = copy.deepcopy(parent2[i].jadwal) child1[i].ruangan = copy.deepcopy(parent2[i].ruangan) child2[i].jadwal = copy.deepcopy(parent1[i].jadwal) child2[i].ruangan = copy.deepcopy(parent1[i].ruangan) swap = not swap prev_point = point else: pass return child1, child2 </pre>
mutate()	Fungsi ini bertujuan untuk melakukan mutasi terhadap suatu individu (state). Mutasi dilakukan dengan mencari slot kelas yang kosong dengan posisi yang random	<pre> def mutate(self, individual, mutation_rate=0.3): for i in range(len(individual)): if random.random() < mutation_rate: max_attempts = 50 for _ in range(max_attempts): h = random.choice(self.hari) j = random.choice(self.range_waktu) r = random.choice(self.ruangan) jadwal_baru = { "jadwal_mulai": [h, j], "jadwal_selesai": [h, j + individual[i].mata_kuliah['sks']] } old_jadwal = copy.deepcopy(individual[i].jadwal) old_ruangan = copy.deepcopy(individual[i].ruangan) individual[i].jadwal = jadwal_baru individual[i].ruangan = r if not self.cek_konflik_jadwal(individual[i], jadwal_baru): break else: individual[i].jadwal = old_jadwal individual[i].ruangan = old_ruangan </pre>

select_parents() s()	Fungsi ini bertujuan untuk melakukan seleksi parent berdasarkan populasi, metode yang digunakan adalah tournament selection	<pre>def select_parents(self, tournament_size=5): """ Tournament selection """ import random parents = [] for _ in range(self.population_size // 2): # Select parent 1 tournament1 = random.sample(self.population, tournament_size) parent1 = max(tournament1, key=lambda ind: self.fitness(ind)) # Select parent 2 tournament2 = random.sample(self.population, tournament_size) parent2 = max(tournament2, key=lambda ind: self.fitness(ind)) parents.append((parent1, parent2)) return parents</pre>
fitness()	Fungsi ini bertujuan untuk melakukan kalkulasi fitness value sebuah individu (state) dengan kalkulasi: jumlah data input * 100 - nilai objektif	<pre>def fitness(self, individual): """ Fungsi fitness dengan pemanfaatan fungsi objektif (selalu bernilai negatif) Semakin besar nilai fitness, semakin baik individu tersebut """ # print(len(self.input['kelas_mata_kuliah']) + len(self.input['ruangan']) + len(self.input['maha return 100 * (len(self.input['kelas_mata_kuliah']) + len(self.input['ruangan']) + len(self.input['mahasiswa'])) - self.fungsi_objektif(individual)</pre>
run()	Fungsi ini bertujuan untuk mengeksekusi algoritma berdasarkan input dan inisialisasi yang ada. Untuk efisiensi algoritma, populasi selanjutnya akan dibatasi menjadi top 10 individu dan 30 random individu, sisanya akan disimpan di array 'bin'. Algoritma ini juga mengantisipasi terjadinya dominasi/stagnasi suatu individu berdasarkan fitness, apabila terjadi maka	<pre>def run(self, name, verbose=False): import random self.visualize_state(f"{name}_awal") bin_individu = [] for generasi in range(self.n_generasi): parents = self.select_parents() next_generation = [] # print(len(parents)) for parent1, parent2 in parents: child1, child2 = self.crossover(parent1, parent2) self.mutate(child1) self.mutate(child2) next_generation.extend([child1, child2]) fitnesses = [self.fitness(ind) for ind in self.population] self.obj_history.append(fitnesses[0]) self.avg_obj_history.append(sum(fitnesses) / len(fitnesses)) from collections import Counter counter = Counter(fitnesses) most_common_count = counter.most_common(1)[0][1] is_stagnant = most_common_count / len(fitnesses) > 0.3 if is_stagnant: # print("Stagnasi populasi, menambah variasi populasi") # time.sleep(2) self.population = self.population + next_generation sorted_pop = sorted(self.population, key=lambda ind: self.fitness(ind), reverse=True) top_10 = sorted_pop[:10] random_30 = random.sample(bin_individu, min(10, len(bin_individu))) self.population = top_10 + random_30 else: self.population += next_generation bin_individu += sorted(self.population, key=lambda ind: self.fitness(ind), reverse=True)[20:] self.population = sorted(self.population, key=lambda ind: self.fitness(ind), reverse=True)[20:] if verbose: print(f"Generasi {generasi + 1}:") print(f"Best fitness: {self.fitness(self.population[0])}") print(f"[self.fitness(ind) for ind in self.population]") print(len(bin_individu)) self.fungsi_objektif(self.population[0], verbose=True) self.visualize_state(f"{name}_akhir", state=self.population[0])</pre>

	<p>algoritma akan mengambil individu random dari bin untuk menggantikan slot 30 random individu agar tidak terjadi dominasi/stagnasi.</p>	
plot()	<p>Fungsi ini bertujuan untuk menampilkan plot dari nilai objektif terbaik dan rata-rata nilai objektif populasi terhadap iterasi/generasi</p>	<pre>def plot(self, name): x = [1 for i in range(1, self.n_generasi+1)] y1 = self.obj_history y2 = self.avg_obj_history fig, (ax1, ax2) = plt.subplots(1, 2, figsize=(14, 6)) # First plot total = 100 * (len(self.input['kelas_mata_kuliah']) + len(self.input['ruangan']) + len(self.input['mahasiswa'])) ax1.plot(x, y1) ax1.set_xlabel('Iterations') ax1.set_ylabel('V') ax1.set_title('Best Ind Fitness/(total)') # Second plot ax2.plot(x, y2, color='red') ax2.set_xlabel('K') ax2.set_ylabel('Y') ax2.set_title('Avg Population Fitness/(total)') plt.tight_layout() plt.savefig('output/' + name + '_obj_plots.png') plt.close()</pre>













3. Hasil Eksperimen dan Analisis

Bagian ini berisi hasil pengujian terhadap tiga algoritma *local search* yang diimplementasikan, yaitu Hill Climbing, Simulated Annealing, dan Genetic Algorithm. Setiap algoritma dijalankan sebanyak tiga kali pada data yang sama untuk mengamati konsistensi hasil dan performa pencarian.













Hasil eksperimen kemudian divisualisasikan melalui plot perubahan nilai fungsi objektif terhadap iterasi untuk melihat konvergensi pencarian. Analisis dilakukan untuk menilai seberapa baik setiap algoritma mendekati solusi optimal, efisiensi waktu eksekusi, kestabilan hasil antar percobaan, serta pengaruh variasi parameter terhadap kualitas solusi yang diperoleh.

1. Hill Climbing Steepest Ascent

Variasi	Parameter dan Data	State	Plot
---------	--------------------	-------	------

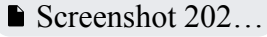


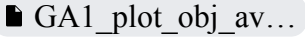
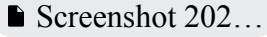


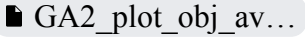
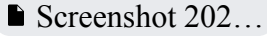


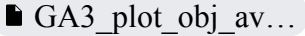












Seed 1		 	
Seed 2		 	
Seed 3		 	
Analisis			
<p>Ketiga eksperimen memiliki tren yang sama, HC selalu mendapatkan solusi untuk kasus input medium (src/input/input_medium.json). Plot objektifnya pun sama, sekitar 10-11 iterasi dan kelengkungan yang relatif sama.</p> <p>Durasi waktu variasi 1 dan 2 tidak berbeda cukup jauh, kecuali variasi 3. Mungkin merupakan dari dampak hardware.</p>			

2. Simulated Annealing

Variasi	Parameter dan Data	State	Plot
Seed 1		 	
Seed 2		 	
Seed 3		 	
Analisis			
<p>Ketiga eksperimen memiliki tren yang cukup berbeda. Variasi 1 dan 3 memiliki hasil yang relatif sama, tapi variasi 3 cukup jauh berbeda lebih buruk.</p> <p>Durasi waktu semua variasi relatif sama persis. Tren plot ketiga variasi memiliki kecenderungan yang sama. Plot fungsi objektif di iterasi awal mengalami naik-turun, di</p>			

sini adalah fase algoritma menerima segala nilai objektif. Setelah sekitar di iterasi 400an, nilai objektif selalu naik dan pada akhirnya stagnan. Plot kalkulasi probabilitas euler semakin lama semakin turun, delta E yang selalu negatif (pada kalkulasi euler saja) dan T yang selalu menurun akan membuat nilai probabilitas eulernya menurun.

3. Genetic Algorithm

Variasi	Parameter dan Data	State	Plot
generasi: 200, populasi: 30		 	
generasi: 200, populasi: 70		 	
generasi: 200, populasi: 120		 	
generasi: 100, populasi: 200		 	
generasi: 200, populasi: 200		 	
generasi: 500, populasi: 200		 	
Analisis			
<p>Semua eksperimen memiliki hasil yang sama yaitu global optimum, yang berbeda hanya pada durasi karena peningkatan kompleksitas algoritma berdasarkan parameter.</p> <p>Semua tren plot objektif individu terbaik sama, di bagian awal iterasi kenaikan fungsi objektif sangat pesat, namun setelah kenaikan pesat tersebut kenaikan menjadi landai. Begitu pun dengan plot objektif rata-rata populasi, meningkat seiring iterasi. Akan</p>			

tetapi, di bagian seringkali ada naik-turun disebabkan algoritma pencegahan stagnasi yang sudah dijelaskan di bagian implementasi genetic algorithm.

4. Analisis Perbandingan Algoritma

Berdasarkan hasil eksperimen, algoritma **Hill Climbing** menunjukkan performa terbaik dalam hal kualitas solusi, diikuti oleh **Genetic Algorithm**. Hill Climbing secara konsisten mampu mencapai solusi yang paling mendekati bahkan menyamai *global optimum*. Hal ini disebabkan oleh karakteristik *state space* pada permasalahan penjadwalan yang memiliki lanskap dengan “bukit” yang lebar dan landai, sehingga jalur peningkatan nilai solusi relatif stabil dan tidak banyak jebakan *local optima*. Mekanisme deterministik pada Hill Climbing membuat setiap langkah pencarian selalu mengarah pada perbaikan solusi, tanpa penyimpangan acak yang tidak perlu. Sementara itu, **Genetic Algorithm** juga mampu menghasilkan solusi dengan kualitas tinggi, meskipun sedikit di bawah Hill Climbing, karena proses evolusionernya yang melibatkan *selection*, *crossover*, dan *mutation* untuk menjaga keragaman populasi.

Jika dibandingkan dengan dua algoritma tersebut, **Simulated Annealing** menghasilkan performa yang lebih bervariasi dan umumnya tidak sebaik Hill Climbing maupun Genetic Algorithm. Hal ini disebabkan oleh sifat stokastiknya yang menggunakan *randomized movement* untuk keluar dari *local optima*. Strategi ini efektif pada lanskap solusi yang kompleks, namun pada kasus penjadwalan dengan pola ruang solusi yang relatif halus, pergerakan acak tersebut seringkali menyebabkan algoritma menyimpang dari arah konvergensi dan sulit mencapai kualitas solusi terbaik.

Dari sisi **durasi proses pencarian**, urutan efisiensinya justru berbeda. **Simulated Annealing** memiliki waktu eksekusi paling cepat karena setiap iterasinya hanya memilih satu *neighbor* secara acak tanpa melakukan evaluasi menyeluruh terhadap seluruh kemungkinan pergerakan. **Genetic Algorithm** berada di posisi tengah karena membutuhkan evaluasi terhadap seluruh populasi pada setiap

generasi. Sementara itu, **Hill Climbing** cenderung memiliki waktu pencarian paling lama, terutama pada varian *steepest ascent*, karena di setiap langkahnya algoritma mengevaluasi seluruh *neighbor* untuk memilih solusi terbaik. Dalam konteks ini, kecepatan tidak selalu berbanding lurus dengan kualitas solusi—Simulated Annealing memang cepat, tetapi sering berhenti di solusi yang kurang optimal.

Dalam hal **konsistensi hasil antar percobaan**, Hill Climbing menunjukkan stabilitas tertinggi karena tidak mengandung elemen acak dalam proses pencariannya. Genetic Algorithm juga relatif konsisten karena mekanisme seleksi selalu mempertahankan individu terbaik di setiap generasi. Sebaliknya, Simulated Annealing memiliki hasil yang lebih fluktuatif karena sangat bergantung pada nilai awal *temperature* dan probabilitas penerimaan solusi yang lebih buruk.

Terakhir, hasil eksperimen menunjukkan bahwa **jumlah iterasi dan ukuran populasi** berpengaruh besar terhadap kualitas hasil pada Genetic Algorithm. Populasi yang lebih besar menjaga keragaman genetik sehingga memperluas ruang pencarian, sementara iterasi yang lebih banyak memberi waktu bagi proses evolusi untuk menghasilkan individu yang lebih optimal. Kombinasi keduanya menjadikan Genetic Algorithm pendekatan yang fleksibel dan efisien untuk menemukan solusi yang mendekati *global optimum* dengan konsistensi yang baik.

Kesimpulan dan Saran

Berdasarkan hasil eksperimen yang dilakukan terhadap ketiga algoritma *local search* Hill Climbing, Simulated Annealing, dan Genetic Algorithm dapat disimpulkan bahwa Hill Climbing memberikan performa terbaik dalam hal kualitas solusi dan kestabilan hasil. Algoritma ini secara konsisten mampu mencapai solusi yang mendekati bahkan menyamai *global optimum* berkat sifat deterministiknya yang efektif pada *state space* dengan lanskap “bukit” yang landai. Genetic Algorithm berada di posisi kedua dengan hasil yang juga sangat baik dan relatif konsisten, terutama pada konfigurasi dengan populasi besar dan iterasi tinggi. Sementara itu, Simulated Annealing memiliki waktu eksekusi tercepat, tetapi kualitas solusinya cenderung lebih rendah karena mekanisme pergerakan acaknya yang sering menyimpang dari arah konvergensi.

Saran terhadap penerapan Algoritma Hill Climbing, Genetic Algorithm, mungkin bisa membuat proses pencariannya lebih optimal menggunakan metode lain dengan *trade off* sedikit di hasil kualitas atau menggunakan prinsip parallel computing di proses yang sekiranya bisa diterapkan (looping).

Pembagian Tugas

Nama	NIM	Tugas
Farrell Jabaar Altafataza	10122057	Hill Climbing, laporan
Raudah Yahya Kuddah	13122003	Simulated Annealing, laporan
Muhammad Syarafi Akmal	13522076	Base code/Parent Algorithm, Genetic Algorithm, Utilities

Referensi

Link github: [Github Tubes](#)

Artificial Intelligence: A Modern Approach, 4th Edition by Peter Norvig and Stuart J. Russell