

Tugas Besar 1 IF3170 Inteligensi Artifisial
Pencarian Solusi Diagonal Magic Cube dengan Local Search



Oleh:

Jonathan Emmanuel Saragih (13522121)
Justin Aditya Putra Prabakti (13522130)
Farrel Natha Saskoro (13522145)
Atqiya Haydar Luqman (13522163)

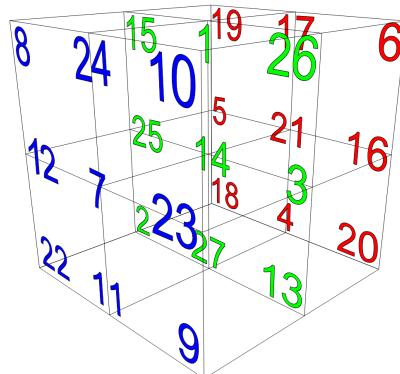
SEKOLAH TEKNIK ELEKTRO DAN INFORMATIKA
INSTITUT TEKNOLOGI BANDUNG
2024

Bab I

Deskripsi Persoalan

Diagonal magic cube merupakan kubus yang tersusun dari angka 1 hingga n^3 tanpa pengulangan dengan n adalah panjang sisi pada kubus tersebut. Angka-angka pada tersusun sedemikian rupa sehingga properti-properti berikut terpenuhi:

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



- Terdapat 9 potongan bidang, yaitu:

8 24 10	15 1 26	19 17 6
12 7 23	25 14 3	5 21 16
22 11 9	2 27 13	18 4 20
19 17 6	5 21 16	18 4 20
15 1 26	25 14 3	2 27 13
8 24 10	12 7 23	22 11 9
8 15 19	12 25 5	22 2 18
24 1 17	7 14 21	11 27 4
10 26 6	23 3 16	9 13 20

- Diagonal yang dimaksud adalah yang dilingkari warna merah saja

- Ilustrasi dan penjelasan lebih detail bisa anda lihat di link berikut: [Features of the magic cube - Magisch vierkant](#)

Pada tugas ini, peserta kuliah akan menyelesaikan permasalahan Diagonal Magic Cube berukuran 5x5x5. Initial state dari suatu kubus adalah susunan angka 1 hingga 5^3 secara acak. Kemudian, tiap iterasi pada algoritma local search, langkah yang boleh dilakukan adalah menukar posisi dari 2 angka pada kubus tersebut (2 angka yang ditukar tidak harus bersebelahan). Khusus untuk genetic algorithm, boleh dilakukan penukaran posisi lebih dari 2 angka sekaligus dalam satu iterasi (tetapi hanya menukar posisi 2 angka saja juga diperbolehkan).

Anda akan diminta untuk mengimplementasikan rencana yang telah Anda buat pada Tugas Kecil 1. Berikut merupakan hal-hal yang perlu dilakukan oleh setiap kelompok:

- Implementasikan 3 algoritma local search dengan rincian sebagai berikut:
 - Salah satu algoritma hill-climbing
 - Simulated Annealing
 - Genetic Algorithm
- Lakukan eksperimen dengan skema sebagai berikut:
 - Jalankan setiap algoritma sebanyak **3 kali**, kemudian catat beberapa hal berikut:
 - Berlaku untuk semua algoritma
 - State awal dan akhir dari kubus
 - Nilai objective function akhir yang dicapai
 - Plot nilai objective function terhadap banyak iterasi yang telah dilewati
 - Durasi proses pencarian
 - Berlaku hanya untuk Steepest Ascent Hill-Climbing dan Stochastic Hill-Climbing
 - Banyak iterasi hingga proses pencarian berhenti
 - Berlaku hanya untuk Hill-Climbing with Sideways Move
 - Banyak iterasi hingga proses pencarian berhenti
 - Note: Tambahkan parameter maximum sideways move, dimana ketika banyak sideways move yang dilakukan sudah mencapai maksimum, pencarian dihentikan
 - Berlaku hanya untuk Random Restart Hill-Climbing
 - Banyak restart
 - Banyak iterasi per restart
 - Note: Tambahkan parameter maximum restart, dimana ketika banyak restart sudah mencapai maksimum, pencarian dihentikan.
 - Berlaku hanya untuk Simulated Annealing
 - Plot $e^{\frac{\Delta E}{T}}$ terhadap banyak iterasi yang telah dilewati
 - Frekuensi ‘stuck’ di local optima

- Khusus untuk Genetic Algorithm, lakukan beberapa hal berikut:
 - Terdapat 2 parameter yang dapat diubah, yaitu **jumlah populasi** dan **banyak iterasi**.
 - Jadikan jumlah populasi sebagai kontrol, kemudian pilih **3 variasi** banyak iterasi yang berbeda. Jalankan program sebanyak masing-masing **3 kali** untuk setiap konfigurasi parameter.
 - Jadikan banyak iterasi sebagai kontrol, kemudian pilih **3 variasi** jumlah populasi yang berbeda. Jalankan program sebanyak masing-masing **3 kali** untuk setiap konfigurasi parameter.
 - Untuk setiap eksperimen, catat beberapa hal berikut:
 - State awal dan akhir dari kubus
 - Nilai objective function akhir yang dicapai
 - Plot nilai objective function terhadap banyak iterasi yang telah dilewati (Cukup plot nilai objective function maksimum dan rata-rata dari populasi terhadap banyak iterasi yang telah dilewati. Jika sudah terlanjur membuat plot untuk tiap individu pada populasi tidak menjadi masalah, silahkan diberikan keterangan saja maksud plotnya apa)
 - Jumlah populasi
 - Banyak iterasi
 - Durasi proses pencarian
- Lakukan analisis terhadap hasil eksperimen, berikut merupakan beberapa pertanyaan yang dapat menjadi acuan untuk analisis yang Anda lakukan (Anda boleh menambahkan beberapa pertanyaan tambahan jika dirasa perlu untuk dijelaskan):
 - Seberapa dekat tiap-tiap algoritma bisa mendekati global optima dan mengapa hasilnya demikian?
 - Bagaimana perbandingan hasil pencarian tiap-tiap algoritma dengan algoritma local search yang lain?
 - Bagaimana perbandingan durasi proses pencarian tiap algoritma relatif terhadap algoritma lainnya?
 - Seberapa konsisten hasil akhir yang didapatkan dari tiap-tiap eksperimen yang dilakukan?
 - Bagaimana pengaruh banyak iterasi dan jumlah populasi terhadap hasil akhir pencarian pada Genetic Algorithm?
 - dst...
- Program yang dibuat harus bisa **memvisualisasikan** state awal kubus, state akhir kubus, dan juga hasil eksperimentnya (sesuaikan informasi hasil eksperimen yang ditampilkan dengan ketentuan yang telah dijelaskan di poin sebelum ini).
- Cara visualisasi dibebaskan kepada kelompok masing-masing selama seluruh angka pada kubus dan juga hasil eksperimen terlihat dengan jelas.

- Dibebaskan menggunakan bahasa pemrograman apapun.
- Diperbolehkan untuk menggunakan heuristik yang Anda buat sendiri atau dari referensi lain untuk optimasi pencarian solusi, asalkan masih dalam lingkup local search. Jangan lupa jelaskan heuristik yang Anda pakai di laporan.

Bab II

Pembahasan

2.1 Pemilihan Objective Function

Objective function kami adalah standar deviasi dari seluruh jumlah deretan yang dapat dibentuk dalam kubus. Algoritma penghitungan standar deviasi akan digunakan untuk mengevaluasi variasi atau penyimpangan dari hasil setiap sum atau jumlah dalam magic cube terhadap nilai rata-ratanya. Ini dilakukan sebagai bagian dari fitness function untuk menentukan langkah yang harus dipilih dalam meningkatkan nilai atau kualitas hasil dari *magic cube solver* yang dibuat.

steeppestascent.py	Deskripsi
def fitness(cube: m.Magicube) -> float: 	Fungsi ini dilakukan dengan mengumpulkan <i>path</i> dari tiap kemungkinan berdasarkan arah t.judgment_vectors dan akan dihitung sum untuk tiap pathnya. Dan kemudian akan disimpan di list value. Kemudian akan dihitung standard deviasinya dari setiap path yang ada di list value. Semakin kecil nilai standar deviasi, maka hasil akan semakin ideal. Nilai dari standar deviasi akan dikembalikan dalam negatif karena hasilnya menggambarkan kekurangan yang masih belum bisa dicapai. Hasil maksimal adalah jika standar deviasinya 0.

2.2 Penjelasan Implementasi Algoritma Local Search

2.3.1 Steepest Ascent Hill-climbing

Algoritma Steepest Ascent Hill-climbing merupakan algoritma yang dimulai mengenerate state awal dan kemudian akan melakukan iterasi ke node dengan nilai yang lebih tinggi atau lebih rendah (menuju semua kemungkinan), dan akan memilih node yang memiliki nilai lebih besar daripada node node lainnya. Proses ini akan menjamin bahwa langkah yang diambil akan

menghasilkan local maximum. Untuk penjelasan terkait algoritma sebagai berikut :

1. Mengevaluasi initial state : Jika *initial state* merupakan *goal state*, maka algoritma akan mengembalikan *initial state* tersebut dan proses pencarian akan dihentikan. Jika *initial state* bukan *goal state*, maka *initial state* dijadikan *current state*.
2. Program akan melakukan iterasi beberapa langkah berikut sampai ditemukannya solusi atau *goal state* :
 - o Melakukan evaluasi dari semua *successor* dari *current state*.
 - o Memilih *neighbor* yang memiliki nilai terbaik
 - o Membandingkan *neighbor* dengan *current*. Jika nilai saat ini sudah lebih baik dari *neighbor* atau sudah mencapai goal, hentikan pencarian. Jika nilai *neighbor* masih lebih baik dari *current*, maka nilai *current* digantikan oleh *neighbor*.
3. Keluar dari fungsi

steepestascent.py	Deskripsi
<pre>def steepestascent(cubelist : list[int] = None) -> dict: runs : int = 0 if cubelist != None: cube: Magicube = Magicube(n,custom=cubelist) else: cube: Magicube = Magicube(n) first_cube = cube.copy() #return values start_time = time.time() graph : list[float] = [cube.get_fitness()] log : str = "" try: for i in range(MAX_ITERATION): current_fitness = cube.get_fitness()</pre>	<p>Fungsi ini akan melakukan algoritma pencarian solusi magic cube menggunakan algoritma steepest ascent.</p> <p>Fungsi ini akan menerima parameter list kubus dan mereturn standard_return yang sudah ditentukan untuk difetch di front end.</p> <p>Algoritma ini akan melakukan iterasi untuk menemukan fitness yang terbaik. Jika new_fitness lebih baik dari best_fitness, maka nilai akan diubah ke yang baru. Untuk proses swap, akan dilakukan ke yang best_fitness terakhir di setiap iterasinya.</p>

```

        print(f"Iteration {i + 1} - "
Starting fitness: {current_fitness}")
        log += (f"Iteration {i + 1} - "
Starting fitness: {current_fitness}) +
"\n\n"

        # Break if optimal solution is
        found (based on a positive fitness
        threshold, if applicable)
        if current_fitness >= 0.0:
            print("Optimal solution
found.")
            log += ("Optimal solution
found.")
            break

        found_better = False
        best_cube = cube.copy()
        best_fitness = current_fitness

        for j in range(n**3 - 1):
            start = generate_vector(j, n)

            for k in range(j + 1, n**3):
                target =
generate_vector(k, n)

                    # Swap spots and evaluate
fitness
                    cube.swap_spot(start.x,
start.y, start.z, target.x, target.y,
target.z)
                    new_fitness =
cube.get_fitness()

                    if new_fitness >
best_fitness:
                        best_fitness =
new_fitness
                        found_better = True
                        best_cube =
cube.copy()
                        print(f"NEW:
{best_fitness} >> Current fitness after
swap: {new_fitness}")

    
```

```

        # Swap back to original
        configuration
        cube.swap_spot(start.x,
start.y, start.z, target.x, target.y,
target.z)

        # If a better solution was
        found, update the cube
        if found_better:
            cube = best_cube
            print(f"End of iteration {i +
1} - New best fitness: {best_fitness}")
            log += (f"End of iteration {i +
1} - New best fitness:
{best_fitness}") + "\n"
        else:
            # Stop the loop if no better
            solution is found (local optimum
            reached)
            print(f"End of iteration {i +
1} - No better solution found,
stopping.")
            log += (f"End of iteration {i +
1} - New best fitness:
{best_fitness}") + "\n"
            runs = i + 1
            break

        graph += [best_fitness]
        print(f"Current fitness after
iteration {i + 1}:
{cube.get_fitness()}\n")
        log += (f"Current fitness after
iteration {i + 1}:
{cube.get_fitness()}\n") + "\n"

        print("Final cube configuration:")
        cube.print()
        log.print()

except Exception as e:
    print("An error occurred:", e)
    cube.print()

```

<pre> execution_time = time.time() - start_time return standard_return(first_cube, cube, graph, execution_time, log = log) </pre>	
--	--

2.3.2 Hill-climbing with Sideways Move

Algoritma Hill-climbing with Sideways Move merupakan algoritma yang dibuat dari algoritma Hill Climbing, namun dimodifikasi untuk menghadapi kasus dimana algoritma terjebak di *shoulder* atau *plateau*, yaitu dimana *neighbor* memiliki nilai yang sama. Saat algoritma ini berada di kondisi *plateau*, program akan diperbolehkan untuk berpindah ke *neighbor* atau bergerak secara sideways. Untuk penjelasan langkah-langkahnya sebagai berikut :

1. Mengevaluasi initial state : Jika *initial state* merupakan *goal state*, maka algoritma akan mengembalikan *initial state* tersebut dan proses pencarian akan dihentikan. Jika *initial state* bukan *goal state*, maka *initial state* dijadikan *current state*.
2. Program akan melakukan iterasi beberapa langkah berikut sampai ditemukannya solusi atau *goal state* :
 - o Melakukan evaluasi dari semua successor dari *current state*.
 - o Memilih *neighbor* yang memiliki nilai terbaik.
 - o Membandingkan *neighbor* dengan *current*. Jika *neighbor* memiliki nilai lebih baik, maka *neighbor* menjadi *current*. Jika *neighbor* memiliki nilai sama dengan *current*. Maka *neighbor* akan menjadi *current*, dan lakukan langkah sideways ke *current* yang baru.
 - o Ulangi langkah-langkah tersebut hingga langkah sideways mencapai batasannya.
3. Keluar dari fungsi

sidewaysmove.py	Deskripsi
<pre> def sidewaysmove(cubelist : list[int] = None, max_iteration : int = None, max_sidewaysmove : int = None) -> dict: #return values start_time = time.time() log : str = "" </pre>	<p>Fungsi ini akan melakukan algoritma pencarian solusi magic cube menggunakan algoritma sideways move.</p> <p>Fungsi ini akan menerima parameter list kubus dan mereturn</p>

```

runs : int = 0

if cubelist != None:
    cube: Magicube =
    Magicube(n,custom=cubelist)
else:
    cube: Magicube = Magicube(n)
    print("Cube unset,
randomizing...")
    log += "CUBE UNSET\n"

if max_iteration != None:
    pass
else:
    max_iteration = 3
    print("Max iteration unset,
setting to 8")
    log += "MAX ITTERATION
UNSET\n"

if max_sidewaysmove != None:
    sideways_moves_limit =
max_sidewaysmove
else:
    sideways_moves_limit = 3
    print("Max sidewaysmove unset,
setting to 10")
    log += "MAX SIDEWAYS
MOVE UNSET\n"

#return values
first_cube = cube.copy()
graph : list[float] =
[cube.get_fitness()]

try:
    sideways_moves = 0
    current_fitness =
cube.get_fitness()

    for i in range(max_iteration):
        print(f"Iteration {i + 1} -
Starting fitness: {current_fitness}")

        # Memeriksa apakah kubus
        saat ini sudah optimal

```

standard_return yang sudah ditentukan untuk difetch di front end.

Algoritma ini akan melakukan iterasi untuk menemukan fitness yang terbaik. Jika new_fitness lebih baik atau sama dengan best_fitness, maka nilai akan diubah ke yang baru. Untuk proses swap, akan dilakukan ke yang best_fitness terakhir di setiap iterasinya.

Gerakan sideways akan memiliki batas maksimum per iterasinya.

```

        if current_fitness >= 0.0:
            print("Optimal solution
found.")
            break

        found_better = False
        best_cube = cube.copy()
        best_fitness = current_fitness
        neighbor = []

        for j in range(n**3 - 1):
            start = generate_vector(j, n)

            for k in range(j + 1, n**3):
                target =
generate_vector(k, n)

                    # Menukar posisi
                    cube.swap_spot(start.x,
start.y, start.z, target.x, target.y,
target.z)

                    # Menghitung fitness
setelah swap
                    neighbor_fitness =
cube.get_fitness()

                    # Jika konfigurasi baru
lebih baik, simpan
                    if neighbor_fitness >
best_fitness:
                        found_better = True
                        best_fitness =
neighbor_fitness
                        best_cube =
cube.copy()
                        print(f'NEW:
{best_fitness} >> Current fitness after
swap: {cube.get_fitness()}')
                        elif neighbor_fitness ==
best_fitness:
                            neighbor +=

[[generate_vector(j,n),generate_vector
(k,n)]]


                    # Kembalikan swap

```

```

        cube.swap_spot(start.x,
start.y, start.z, target.x, target.y,
target.z)

    if found_better:
        cube = best_cube
        current_fitness =
best_fitness
        sideways_moves = 0

        print(f"End of iteration {i +
1} - New best fitness: {best_fitness}")
        log += (f"End of iteration {i +
1} - New best fitness:
{best_fitness}") + "\n"
        graph += [best_fitness]
    elif len(neighbor) > 0:
        sideways_moves += 1
        print(f"End of iteration {i +
1} - No better solution found.
Sideways moves:
{sideways_moves}")
        log += (f"End of iteration {i +
1} - No better solution found.
Sideways moves:
{sideways_moves}") + "\n"
        graph +=
[cube.get_fitness()]
        if sideways_moves >=
sideways_moves_limit:
            print("Stopped due to
exceeding sideways move limit.")
            break
        else:
            #pick random neighbor
            random_pick = randint(0,
len(neighbor))
            start =
neighbor[random_pick][0]
            target =
neighbor[random_pick][1]
            cube.swap_spot(start.x,
start.y, start.z, target.x, target.y,
target.z)
        else:
            print(f"End of iteration {i +

```

```

1} - No better solution found & Local
maxima.")
    log(f"End of iteration {i +
1} - No better solution found & Local
maxima.")
    break

    print(f"Current fitness after
iteration {i + 1}: {current_fitness}\n")

    print("Final cube configuration:")
    cube.print()

except Exception as e:
    print("An error occurred:", e)
    cube.print()

end_time = time.time()
execution_time = end_time -
start_time
print(f"Execution time:
{execution_time:.2f} seconds")
log = f"Itterations : {runs} \n" + log

return
standard_return(first_cube,cube,graph,
execution_time,log)

```

2.3.3 Random Restart Hill-climbing

Algoritma Random Restart Hill-climbing adalah algoritma yang akan melakukan inisiasi ulang sebanyak beberapa kali dari posisi awal yang dirandom. Setiap kali algoritma terjebak pada local optimum, algoritma akan melakukan inisiasi di posisi yang baru. Untuk penjelasan langkah-langkah lebih lengkapnya sebagai berikut :

1. Memulai inisialisasi secara random
2. Melakukan program Hill-Climbing dari initial state yang pada langkah sebelumnya sudah ditentukan, kemudian menjalankan program hingga program terjebak pada local optimum.
3. Melakukan inisialisasi kembali secara random kemudian menjalankan program Hill-Climbing kembali.

4. Ulangi terus algoritma ini sampai mencapai goal atau sudah mencapai batas yang ditentukan.
5. Program akan menyimpan state yang memiliki solusi terbaik.
6. Setelah mengulangi program Hill-Climbing sebanyak beberapa kali, program akan memilih solusi yang terbaik.

randomrestart.py	Deskripsi
<pre> def randomrestart(cubelist : list[int] = None, max_iteration : int = None, max_restarts : int = None) -> dict: #return values start_time = time.time() log : str = "" if cubelist != None: cube: Magicube = Magicube(n,custom=cubelist) else: cube: Magicube = Magicube(n) print("Cube unset, randomizing...") log += "CUBE UNSET\n" if max_iteration != None: pass else: max_iteration = 3 print("Max iteration unset, setting to 8") log += "MAX ITTERATION UNSET\n" if max_restarts != None: pass else: max_restarts = 100000 print("Max restarts unset, setting to 10") log += "MAX RESTARTS UNSET\n" #return values first_cube = cube.copy() </pre>	<p>Fungsi ini akan melakukan algoritma pencarian solusi magic cube menggunakan algoritma randomrestart.</p> <p>Fungsi ini akan menerima parameter list kubus dan mereturn standard_return yang sudah ditentukan untuk difetch di front end.</p> <p>Algoritma ini akan melakukan pengulangan ke state awal kembali secara random jika sudah stuck di local optima.</p> <p>Algoritma ini akan tetap menyimpan state terbaik dan akan berganti jika menemukan yang nilai fitnessnya lebih baik.</p>

```

graph : list[float] =
[cube.get_fitness()]

# Function to randomize the cube
def randomize_cube(cube:
Magicube):
    for _ in range(n**3 * 2): #
Arbitrary number of swaps for
shuffling
        j = random.randint(0, n**3 - 1)
        k = random.randint(0, n**3 -
1)

        if j != k:
            start = generate_vector(j, n)
            target = generate_vector(k,
n)
            cube.swap_spot(start.x,
start.y, start.z, target.x, target.y,
target.z)

    best_fitness_overall = float('-inf')
    best_cube_overall = cube.copy()
    restart_count = 0
    total_iterations = 0

    start_time = time.time()
    try:
        for restart in range(max_restarts):
            restart_count += 1
            randomize_cube(cube)
            current_fitness =
cube.get_fitness()
            print(f'Restart {restart_count}')
            - Starting fitness: {current_fitness}')
            log += (f'Restart
{restart_count} - Starting fitness:
{current_fitness}') + "\n"

            found_optimal = False
            for i in range(max_iteration):
                total_iterations += 1

                if current_fitness >= 0.0:
                    print("Optimal solution
found.")

```

```

        found_optimal = True
        break

        found_better = False
        for _ in range(n**3):
            j = random.randint(0,
n**3 - 1)
                k = random.randint(0,
n**3 - 1)

            if j != k:
                start =
generate_vector(j, n)
                    target =
generate_vector(k, n)

                cube.swap_spot(start.x,
start.y, start.z, target.x, target.y,
target.z)

                new_fitness =
cube.get_fitness()

                if new_fitness >
current_fitness:
                    current_fitness =
new_fitness
                        found_better = True
                        print(f"Iteration {i +
1} - Improved fitness: {new_fitness}")
                else:

                    cube.swap_spot(start.x, start.y, start.z,
target.x, target.y, target.z)

                    if not found_better:
                        print(f"Iteration {i + 1} -
No better solution found, moving to
next restart.")
                    break

                    print(f"End of iteration {i +
1} - Current fitness:
{current_fitness}")

                if current_fitness >

```

```

best_fitness_overall:
    best_fitness_overall =
current_fitness
    best_cube_overall =
cube.copy()
        print(f"New overall best
fitness: {best_fitness_overall}")
        log += (f"New overall best
fitness: {best_fitness_overall}") + "\n"

        graph +=
[best_fitness_overall]

        print(f"Total restarts
completed: {restart_count}")
        print(f"Total iterations across
all restarts: {total_iterations}")

if found_optimal:
    break

print("Final cube configuration
with best fitness across all restarts:")
best_cube_overall.print()
print(f"Total restarts:
{restart_count}")
log = (f"Total restarts:
{restart_count}") + "\n" + log
print(f"Total iterations:
{total_iterations}")
log = (f"Total iterations:
{total_iterations}") + "\n" + log

except Exception as e:
    print("An error occurred:", e)
    cube.print()

end_time = time.time()
execution_time = end_time -
start_time
print(f"Execution time:
{execution_time:.2f} seconds")

```

```

log = f'Iterations :
{total_iterations} \n" + log
return
standard_return(first_cube,cube,graph,
execution_time,log)

```

2.3.4 Stochastic Hill-climbing

Algoritma Stochastic Hill-climbing merupakan algoritma Hill-Climbing yang memilih neighbornya dengan cara memilih secara random. Algoritma ini tidak akan memilih neighbor dengan memeriksa semua node yang ada. Algoritma ini hanya merandom untuk neighbor yang akan dibandingkan, kemudian program akan memutuskan apakah current akan berganti ke neighbor atau tidak. Untuk penjelasan langkah-langkah lengkapnya sebagai berikut :

1. Mengevaluasi initial state : Jika *initial state* merupakan *goal state*, maka algoritma akan mengembalikan *initial state* tersebut dan proses pencarian akan dihentikan. Jika *initial state* bukan *goal state*, maka *initial state* dijadikan *current state*.
2. Program akan melakukan iterasi beberapa langkah berikut sampai ditemukannya solusi atau *goal state* :
 - Memilih neighbor state yang belum pernah dijalankan dengan algoritma ini sebelumnya yang lebih baik dari state saat ini
 - Membandingkan neighbor dengan current. Jika neighbor memiliki nilai lebih baik, maka neighbor menjadi current. Jika neighbor memiliki nilai sama dengan current. Maka neighbor akan menjadi current, dan lakukan langkah sebelumnya.
 - Ulangi langkah - langkah tersebut hingga langkah sideways mencapai batasannya.
3. Keluar dari fungsi

stochastic.py	Deskripsi
<pre> def stochastic(cubelist : list[int] = None, max_iteration : int = None) -> dict: #return values start_time = time.time() log : str = "" runs : int = 0 if cubelist != None: </pre>	<p>Fungsi ini akan melakukan algoritma pencarian solusi magic cube menggunakan algoritma stochastic hill climbing.</p> <p>Fungsi ini akan menerima parameter list kubus dan mereturn standard_return yang sudah ditentukan</p>

```

cube: Magicube =
Magicube(n,custom=cubelist)
else:
    cube: Magicube = Magicube(n)
    print("Cube unset,
randomizing...")
    log += "CUBE UNSET\n"

if max_iteration != None:
    pass
else:
    max_iteration = 3
    print("Max iteration unset,
setting to 8")
    log += "MAX ITTERATION
UNSET\n"

#return values
first_cube = cube.copy()
graph : list[float] =
[cube.get_fitness()]

try:
    for i in range(max_iteration):
        current_fitness =
cube.get_fitness()

        print(f"Iteration {i + 1} -
Starting fitness: {current_fitness}")
        log += (f"Iteration {i + 1} -
Starting fitness: {current_fitness}" +
"\n"

        if current_fitness >= 0.0:
            print("Optimal solution
found.")
            break

        found_better = False

        for _ in range(n**3): #Fungsi
untuk merandom penukaran
            j = random.randint(0, n**3 -
1)
            k = random.randint(0, n**3 -
1)

```

untuk difetch di front end.

Algoritma ini melakukan pertukaran state antara 2 angka secara random. Hal ini dilakukan dengan penentuan random pada generate vector (untuk menentukan kordinat yang ditukar). Jika pertukaran memiliki nilai fitness yang lebih baik maka state akan disimpan sebagai best_fitness. Proses akan terus diulang hingga menemukan solusi terbaik atau proses iterasi sudah mencapai batas maksimal.

```

        if j != k:
            start = generate_vector(j,
n)
            target =
generate_vector(k, n)

            cube.swap_spot(start.x,
start.y, start.z, target.x, target.y,
target.z)

            new_fitness =
cube.get_fitness()

            if new_fitness >
current_fitness:
                current_fitness =
new_fitness
                found_better = True
                print(f"NEW:
{new_fitness} >> Current fitness after
swap: {cube.get_fitness()}")
                else:
                    cube.swap_spot(start.x,
start.y, start.z, target.x, target.y,
target.z)

                if not found_better:
                    print(f"End of iteration {i +
1} - No better solution found,
stopping.")
                    log += (f"End of iteration {i +
1} - No better solution found,
stopping.") + "\n"
                    runs = i + 1
                    break
                else:
                    print(f"End of iteration {i +
1} - New fitness: {current_fitness}")
                    log += (f"End of iteration {i +
1} - New fitness: {current_fitness}") +
"\n"
                    graph += [current_fitness]

                    print(f"Current fitness after
iteration {i + 1}: {current_fitness}\n")

```

```

print("Final cube configuration:")
cube.print()

except Exception as e:
    print("An error occurred:", e)
    log += ("An error occurred:", e)
    cube.print()

end_time = time.time()
execution_time = end_time -
start_time
print(f"Execution time:
{execution_time:.2f} seconds")

log = f'Iterations : {runs} \n' + log

return standard_return(first_cube,
cube, graph, execution_time, log =
log)

```

2.3.5 Simulated Annealing

Algoritma Simulated Annealing adalah optimasi dari algoritma Hill-Climbing, yang mana dalam algoritma ini, digabungkan antara Hill-Climbing dengan eksplorasi random untuk menghindar dari terjebak di local optima. Algoritma ini mengikuti pada proses annealing di metalurgi, dimana sebuah material dipanaskan lalu didinginkan secara perlahan untuk meminimasi energy dan mencapai kesetimbangan. Secara keseluruhan Algoritma ini mirip dengan Hill-Climbing, namun dibanding memilih langkah terbaik, algoritma ini memilih langkah random. Jika langkahnya lebih baik maka langkah selalu diterima, namun jika langkahnya lebih buruk, langkah tetap diterima dengan syarat probabilitas $e^{\Delta E/T}$ mendekati 1. Dengan ΔE adalah nilai dari langkah selanjutnya dikurangi dengan nilai dari langkah sekarang dan T adalah jadwal. Untuk lebih jelasnya disajikan gambar berikut.

simulated_annealing.py	Deskripsi
def simulated_annealing(cubelist :	Fungsi ini akan melakukan algoritma

```

list[int]=None, initial_temp=None,
cooling_rate=None) -> dict:

    #return values
    start_time = time.time()
    log : str = ""

    if cubelist != None:
        cube: Magicube =
        Magicube(n,custom=cubelist)
    else:
        cube: Magicube = Magicube(n)
        print("Cube unset,
randomizing...")
        log += "CUBE UNSET\n"

    if initial_temp != None:
        pass
    else:
        initial_temp = 1000
        print("Initial temp unset! setting
to 1000...")
        log += "INITIAL TEMP
UNSET\n"

    if cooling_rate != None:
        pass
    else:
        cooling_rate = 0.9995
        print("cooling rate unset! setting
to 0.99995...")
        log += "INITIAL TEMP
UNSET\n"

    #return values
    first_cube = cube.copy()
    best_cube : Magicube = None
    graph : list[float] =
    [cube.get_fitness()]

    current_cube = cube
    current_fitness = cube.get_fitness()
    current_deviation = deviation(cube)
    best_fitness = current_fitness
    best_deviation = current_deviation
    temp = initial_temp

```

pencarian solusi magic cube menggunakan algoritma simulated_annealing.

Fungsi ini akan menerima parameter list kubus, initial temperature, dan cooling rate lalu mereturn standard_return yang sudah ditentukan untuk difetch di front end.

Algoritma ini melakukan pertukaran state ke neighbor jika fitness dari neighbornya lebih baik, namun jika tidak lebih baik, akan dicek syarat probabilitasnya apakah lebih besar dari angka random (pemilihan angka random dengan pertimbangan angka random lebih bagus untuk keluar dari local optima), jika lebih besar maka akan state neighbornya akan diterima. Selama iterasi, temperaturnya akan didinginkan dengan cooling rate dan akan berhenti jika temperaturnya mencapai angka 0.

```

iteration = 0
max_fitness_list = []
avg_fitness_list = []
exp_term_list = []

stuck_counter = 0
stuck_frequency = 0

start_time = time.time()
while temp > 1:
    neighbor_cube =
generate_neighbor(current_cube)
    neighbor_fitness =
neighbor_cube.get_fitness()
    neighbor_deviation =
deviation(neighbor_cube)
    deltaEF = neighbor_fitness -
current_fitness
    deltaED = neighbor_deviation -
current_deviation
    exp_term = math.exp(-deltaED /
temp)

    if neighbor_deviation <
current_deviation:
        current_cube =
neighbor_cube.copy()
        current_fitness =
neighbor_fitness
        current_deviation =
neighbor_deviation
        if current_deviation <
best_deviation:
            best_cube =
current_cube.copy()
            best_fitness =
current_fitness
            best_deviation =
current_deviation
            stuck_counter = 0

    else:
        if exp_term >
random.random():
            current_cube =
neighbor_cube.copy()

```

```

        current_fitness =
neighbor_fitness
        current_deviation =
neighbor_deviation
            if current_deviation <
best_deviation:
                best_cube =
current_cube.copy()
                best_fitness =
current_fitness
                best_deviation =
current_deviation
                stuck_counter += 1

            if stuck_counter > 1:
                stuck_frequency += 1
                stuck_counter = 0

max_fitness_list.append(max(current_
fitness, neighbor_fitness))

avg_fitness_list.append((current_fitne
ss + neighbor_fitness) / 2)
exp_term_list.append(exp_term)

temp *= cooling_rate
temp = round(temp, 5)
iteration += 1

print(f"Iteration: {iteration},
Temperature: {temp}, Fitness:
{best_fitness}, Deviation:
{best_deviation}")
log += (f"Iteration: {iteration},
Temperature: {temp}, Fitness:
{best_fitness}, Deviation:
{best_deviation}") + "\n"
end_time = time.time()

# plt.figure(figsize=(10, 6))
# plt.plot(max_fitness_list,
label='Max Fitness', color='blue')
# plt.plot(avg_fitness_list,

```

```

label='Average Fitness',
color='orange')
# plt.xlabel('Iterations')
# plt.ylabel('Objective Function
Value')
# plt.title('Objective Function Value
vs Iterations')
# plt.legend()
# plt.show()

time_exec = end_time - start_time

print("RESULT")
print(f"Best Fitness: {best_fitness},
Best Deviation: {best_deviation},
Iteration: {iteration}, Time:
{time_exec}, Stuck Frequency:
{stuck_frequency}")
log =(f"Best Fitness:
{best_fitness}\nBest Deviation:
{best_deviation}\nIteration:
{iteration}\nTime:
{time_exec}\nStuck Frequency:
{stuck_frequency}") + log
log = f"Itterations : {iteration} \n"
+ log
return
standard_return(first_cube,best_cube,
max_fitness_list,time_exec,log,{"avg_
graph":avg_fitness_list,"exp_graph":e
xp_term_list,"stuck_freq":stuck_frequ
ency})

```

2.3.6 Genetic Algorithm

Algoritma genetic algorithm mulai dengan sejumlah sampel, lalu secara acak dipilih 2 sampel, dimana sampel dengan nilai yang lebih baik (mendekati 0) memiliki kemungkinan lebih tinggi untuk dipilih. Lalu kedua sampel terpilih

digabungkan menjadi satu kubus baru. Pada implementasi ini dibuat 2 metode penggabungan kubus :

- Disintegrate : Kubus seakan-akan “dicampur”/difusi menjadi kubus baru
- Split : Kubus dibelah menjadi 2 dan digabungkan
Mutasi hanya dilakukan

```
#use samples if want to define samples, use sample count to tell how many
to generate

def genetic_algorithm(sample_count : int = None, itterations : int = None,
methodstr : str = None):
    #return values
    runs : int = 0
    start_time = time.time()
    log : str = ""

    if sample_count != None:
        pass
    else:
        sample_count = 4
        print("Sample count unset! setting to 4...")
        log += "SAMPLE COUNT UNSET\n"

    if itterations != None:
        pass
    else:
        itterations = 64
        print("Itterations unset! setting to 64...")
        log += "ITTERATIONS UNSET\n"

    if methodstr in ["disintegrate","split"]:
        pass
    else:
        methodstr = "disintegrate"
        print("Method unset! setting to disintegrate")
        log += "METHOD UNSET\n"

    method : Callable = None
    if (methodstr == "disintegrate"):
        method = disintegrate
    elif (methodstr == "split"):
```

```

method = split

cubes = [Magicube(5) for i in range(sample_count)]

#return values
first_cube = best_cube(cubes).copy()
report = report_avg_cubes(cubes)
graph : list[float] = [report["BEST"]]
graph_avg : list[float] = [report["AVG"]]

try:
    for i in range(itterations):
        runs += 1
        cubes = breed(cubes,method)

        report = report_avg_cubes(cubes)
        avg = report["AVG"]
        best = report["BEST"]
        print(f"Run-{i+1}) Average: {avg:.2f}, Best: {best:.2f}")
        log += f"Run-{i+1}) Average: {avg:.2f}, Best: {best:.2f}\n"

        graph += [report["BEST"]]
        graph_avg += [report["AVG"]]

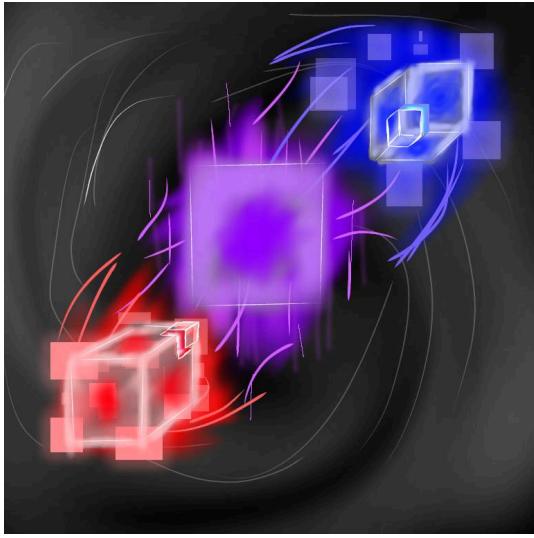
    print("END")

except Exception as e:
    print("Error :",e)
    log += "ERROR : " + e.__context__

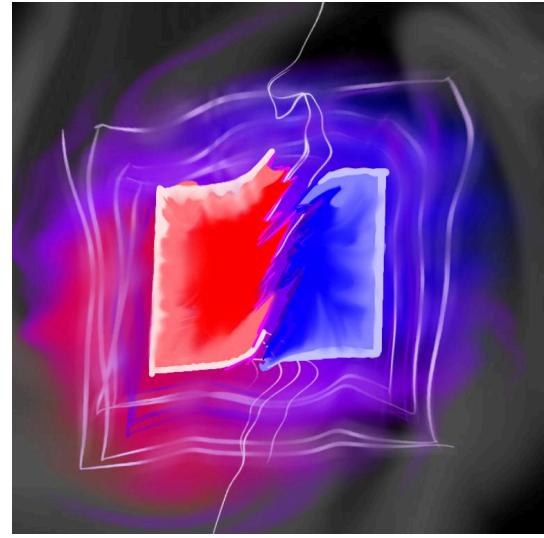
end_time = time.time()
execution_time = end_time - start_time
print(f"Execution time: {execution_time:.2f} seconds")
log = f"Itterations : {runs} |\n" + log

return
standard_return(first_cube,best_cube(cubes),graph,execution_time,log,{"graph_avg" : graph_avg})

```



Disintegrate



Split

2.3 Hasil Eksperimen dan Saran

2.3.1 Steepest Ascent Hill-climbing

Untuk algoritma ini, dilakukan percobaan sebanyak tiga kali. Terdapat penjelasan dan hasil analisis dari percobaan dalam beberapa aspek berikut :

```

Iteration 324 - Starting fitness: -1.7935399613424792
7750
End of iteration 324 - No better solution found, stopping.
Final cube configuration:
52 74 18 88 91
115 89 22 7 81
39 67 66 35 109
36 78 106 85 10
73 9 103 110 23

24 51 120 87 32
44 69 123 58 21
47 116 57 53 37
82 48 8 68 117
118 29 6 55 187

50 96 72 34 68
77 33 1 105 101
93 59 63 38 62
27 15 99 125 46
68 111 84 12 41

122 3 14 95 83
4 104 49 88 76
54 111 42 98 108
102 79 97 17 25
31 124 112 16 30

65 94 92 19 45
76 20 110 56 43
79 61 86 90 2
71 108 5 28 113
26 40 13 121 114

Execution time: 2972.28 seconds

Iteration 260 - Starting fitness: -1.44334624669540868
7750
End of iteration 260 - No better solution found, stopping.
Final cube configuration:
89 122 3 74 28
50 6 115 97 46
19 93 90 38 75
70 12 21 88 123
87 83 86 18 41

100 19 25 7 24
13 43 107 100 52
125 76 22 27 64
53 84 54 82 44
16 102 35 98 61

30 78 119 56 32
113 77 68 48 15
8 88 62 99 106
73 81 69 42 49
91 5 1 109 111

31 105 4 55 120
110 71 14 36 85
51 39 117 99 9
103 23 67 58 65
26 77 114 66 37

57 2 06 121 40
29 124 11 33 118
112 26 25 92 68
17 116 104 45 34
101 47 70 24 63

Execution time: 51846.95 seconds

Iteration 225 - Starting Fitness: -1.7794175668841903
7750
End of iteration 225 - New best fitness: -1.77428646925410158
Current fitness after iteration 225: -1.77428646925410158

Iteration 226 - Starting Fitness: -1.77428646925410158
7750
End of iteration 226 - No better solution found, stopping.
Final cube configuration:
17 24 116 99 57
65 125 107 3 13
84 11 83 5 2 110
89 113 5 1 2 110
61 38 6 121 88

103 75 29 43 67
53 62 8 69 122
113 77 68 48 15
33 60 115 66 41
44 112 91 59 9

4 102 108 87 45
20 31 85 95 42
93 111 64 21 26
74 12 47 63 117
123 38 10 49 114

119 34 30 35 97
55 28 101 50 82
1 68 56 94 98
105 77 23 99 20
36 109 106 46 19

72 88 32 52 79
120 25 14 100 54
58 108 39 31 70
1 51 109 99 77
48 40 108 37 86

Execution time: 1903.07 seconds

```

a. Iteration Count dan Stopping Condition :

Iterasi yang dicapai bervariasi, mulai dari 225 iterasi hingga 324 iterasi. Hal ini menunjukkan bahwa proses pencarian sudah mencapai titik optimal. Hal ini menunjukkan bahwa algoritma ini telah mencapai titik local optimal dengan jumlah iterasi yang cukup efisien. Dengan jumlah iterasi yang berbeda. Dari percobaan, solusi terbaik ada pada pencarian dengan jumlah iterasi yang lebih rendah. Hal ini menunjukkan algoritma steepest ascent dipengaruhi pada posisi awal dan gradien solusi yang tersedia.

b. Execution Time :

Algoritma ini memiliki iteration time yang beragam serta relatif lama jika dibandingkan dengan algoritma lain. Mulai dari 51846 detik hingga 1903 detik. Hal ini disebabkan karena kompleksitas perhitungan serta proses pencarian yang kompleks.

c. Fitness Value :

Dalam hal ini, hasil fitness yang ditunjukan relatif kecil, mulai -1.7 sampai -1.4 , dan dapat dilihat algoritma ini dapat mencari solusi yang cukup baik. Namun, kelemahannya algoritma ini belum menemukan fitness function yang sempurna. Algoritma ini belum dapat menemukan solusi yang global optimal karena selalu terjebak di local optimal.

2.3.2 Hill-climbing with Sideways Move

```
Iteration 132 - Starting fitness: -2.8386500455092842
NEW: -2.8296882471295164 >> Current fitness after swap: -2.8296882471295164
NEW: -2.8288475613426582 >> Current fitness after swap: -2.8288475613426582
NEW: -2.826925055026154 >> Current fitness after swap: -2.826925055026154
NEW: -2.8255725095625817 >> Current fitness after swap: -2.8255725095625817
NEW: -2.8224441467642554 >> Current fitness after swap: -2.8224441467642554
End of iteration 132 - New best fitness: -2.8224441467642554
Current fitness after iteration 132: -2.8224441467642554
```

Algoritma ini akan melakukan ‘

```
Iteration 132 - Starting fitness: -3.10048060023434
NEW: -3.097026513383563 >> Current fitness after swap: -3.097026513383563
NEW: -3.082510152670771 >> Current fitness after swap: -3.082510152670771
NEW: -3.070471424267666 >> Current fitness after swap: -3.070471424267666
NEW: -3.070001068067254 >> Current fitness after swap: -3.070001068067254
NEW: -3.0693369130753143 >> Current fitness after swap: -3.0693369130753143
NEW: -3.068174295900379 >> Current fitness after swap: -3.068174295900379
End of iteration 132 - New best fitness: -3.068174295900379
Current fitness after iteration 132: -3.068174295900379
```

- a. Count dan Stopping Condition :

Algoritma ini akan berhenti jika sudah mencapai fitness function yang optimum ataupun batas sideways sudah dilewati dan masih terjebak di local optima.

- b. Execution Time :

Waktu dari proses ini cukup lama karena kompleksitas yang cukup tinggi serta iterasi yang cukup banyak. Waktu yang diperlukan relatif mirip dengan steepest ascent.

- c. Fitness Value :

Fitness Function yang dihasilkan relatif baik.

2.3.3 Random Restart Hill-climbing

Percobaan dengan Random Restart dilakukan sebanyak 3 kali dengan jumlah random restart adalah 10, 20, 40.

Total restarts completed: 10 Total iterations across all restarts: 446 Final cube configuration with best fitness across all restarts: 98 67 50 34 15 23 13 106 10 78 19 72 98 46 95 58 108 4 48 44 102 92 2 81 122 11 28 37 113 52 10 21 14 85 16 90 60 120 17 51 28 120 41 76 75 124 12 69 39 40 117 99 9 54 115 6 10 32 85 143 10 69 95 35 69 59 110 21 88 22 18 96 91 14 103 125 45 27 118 70 25 106 1 114 3 87 5 39 47 83 7 73 111 21 56 66 97 74 53 23 26 116 86 8 112 101 30 63 61 94 36 38 82 89 77 20 88 55 93 18 121 56 29 Total restarts: 10 Total iterations: 446 Execution time: 54.94 seconds Best fitness: -6.139780771093656	Final cube configuration with best fitness across all restarts: 65 72 39 61 81 50 90 62 29 84 37 9 77 107 83 68 34 95 76 44 93 108 49 42 21 35 30 105 53 97 117 55 25 47 69 102 112 73 12 13 3 33 88 104 91 66 86 20 99 38 116 89 46 67 5 114 15 125 10 103 22 63 71 75 94 111 121 10 27 32 54 17 78 108 82 74 58 18 122 92 123 48 4 87 64 114 110 59 1 15 31 51 115 89 28 23 46 113 19 118 89 74 119 7 41 11 109 96 98 2 36 6 43 124 101 106 57 8 26 120 79 78 68 56 48 Total restarts: 20 Total iterations: 1954 Execution time: 113.09 seconds Best fitness: -6.1682161086842423	Final cube configuration with best fitness across all restarts: 74 5 119 88 23 61 95 9 43 101 29 32 93 83 81 92 99 72 1 56 57 79 25 103 50 28 108 10 109 66 46 63 41 47 116 111 12 125 54 7 91 22 6 89 112 51 113 124 13 14 28 60 31 95 105 102 64 37 99 27 70 120 69 16 42 117 49 87 35 26 3 18 85 98 121 118 94 53 21 30 33 48 114 96 19 38 34 8 123 118 17 80 75 68 77 106 59 67 4 76 65 52 97 15 84 73 40 107 39 58 71 115 24 36 78 2 62 82 122 45 104 44 11 100 55 Total restarts: 40 Total iterations: 1954 Execution time: 232.05 seconds Best fitness: -4.792216761403565
---	--	--

a. Iteration Count and Stopping Condition :

Jumlah iterasi yang dicapai bervariasi di setiap eksperimen, mulai dari 446 iterasi hingga 1954 iterasi. Hal ini menunjukkan bahwa proses pencarian berhenti ketika mencapai titik optimal lokal pada jumlah iterasi tertentu. Algoritma ini sudah mencapai titik optimal lokal pada konfigurasi terbaik di setiap restart. Dari percobaan yang dilakukan, solusi terbaik justru ditemukan pada iterasi yang lebih rendah di gambar kedua dengan total 924 iterasi dan nilai fitness terbaik. Ini menunjukkan bahwa dalam algoritma Random Restart Hill Climbing, hasil dapat dipengaruhi oleh posisi awal dan variasi restart yang dijalankan.

b. Execution Time :

Waktu eksekusi bervariasi tergantung pada jumlah restart dan iterasi yang dijalankan. Algoritma ini memiliki waktu eksekusi mulai dari 54,94 detik hingga 232,05 detik, dengan waktu eksekusi paling optimal dihasilkan pada percobaan pertama. Hal ini menunjukkan bahwa semakin banyak iterasi dan restart, semakin lama waktu yang dibutuhkan. Meskipun algoritma ini mampu mencapai solusi optimal dalam waktu yang lebih singkat pada beberapa percobaan, waktu eksekusinya relatif lebih lama dibandingkan jika jumlah iterasi dan restart lebih kecil.

c. Fitness Value :

Nilai fitness terbaik yang dicapai bervariasi di setiap eksperimen, dengan rentang mulai dari -6,1682 hingga -4,7922. Nilai fitness terbaik dicapai pada jumlah restart terbanyak yaitu 40 restart, yang menunjukkan bahwa algoritma ini mampu menemukan solusi yang baik, tetapi solusi tersebut tetap terjebak pada titik optimal lokal. Algoritma ini juga dapat dipengaruhi dengan jumlah restart yang dilakukan, karena semakin banyak jumlah restart, maka kemungkinan untuk mencapai solusi yang lebih baik menjadi semakin tinggi, walaupun hal tersebut belum bisa memastikan hasil yang lebih baik.

2.3.4 Stochastic Hill-climbing

Percobaan stochastic hill-climbing dilakukan sebanyak 3 kali dengan menghasilkan 3 variasi hasil yang cukup berbeda.

<small>Iteration 44 - Starting fitness: -7.837024961338857 End of iteration 44 - No better solution found, stopping. Final cube configuration: 44 25 119 57 68 78 53 109 5 70 106 14 72 29 94 20 115 13 123 51 59 117 6 102 26 10 69 90 23 118 121 55 47 56 27 11 100 36 110 45 113 31 95 91 2 75 54 33 34 120 103 79 8 104 30 17 81 65 107 46 39 96 66 19 84 114 43 63 1 98 38 15 116 92 62 74 122 4 105 12 22 40 42 73 125 97 7 82 41 99 35 76 58 88 52 87 71 124 9 28 85 21 108 18 83 86 80 49 67 37 48 101 50 111 3 32 64 89 16 112 61 60 24 93 77</small>	<small>Iteration 61 - Starting fitness: -5.4904699196733535 End of iteration 61 - No better solution found, stopping. Final cube configuration: 94 99 1 74 72 103 3 58 30 122 76 99 106 7 33 9 69 109 84 39 34 77 43 120 40 15 75 93 124 10 88 71 87 22 49 65 18 81 113 36 62 107 8 31 108 91 42 51 21 115 78 125 25 47 35 4 68 50 95 101 13 24 61 112 104 118 27 86 60 28 102 64 99 5 48 32 6 98 55 123 111 83 89 41 2 117 67 56 45 19 46 85 57 53 73 12 70 16 121 97 96 20 110 11 66 23 100 29 119 44 38 114 14 37 116 79 26 54 92 63 82 59 105 52 17 Execution time: 5.46 seconds</small>	<small>Iteration 35 - Starting fitness: -10.864618044199588 End of iteration 35 - No better solution found, stopping. Final cube configuration: 71 105 110 3 30 115 45 18 125 16 65 98 70 7 68 22 39 93 49 109 37 28 31 124 95 111 17 9 99 92 20 67 55 75 104 122 5 36 107 35 52 116 123 15 6 1 117 90 21 85 4 80 74 121 43 81 50 84 19 88 86 12 58 100 46 72 102 40 63 29 82 53 76 10 112 57 48 54 77 61 69 97 34 32 83 25 103 73 79 41 51 42 44 78 113 101 26 106 59 23 64 62 66 24 87 38 47 120 60 33 2 108 89 11 119 118 8 14 114 56 94 91 27 96 13 Execution time: 4.03 seconds</small>
--	---	---

a. Iteration Count and Stopping Condition :

Jumlah iterasi yang dicapai bervariasi di setiap eksperimen, mulai dari 35 iterasi hingga 61 iterasi. Hal ini menunjukkan bahwa proses pencarian berhenti ketika mencapai titik optimal lokal pada jumlah iterasi tertentu. Jumlah iterasi relatif lebih sedikit jika dibandingkan dengan algoritma lain karena algoritma ini mudah terjebak di local optima.

b. Execution Time :

Waktu eksekusi bervariasi tergantung pada jumlah restart dan iterasi yang dijalankan. Algoritma ini memiliki waktu eksekusi mulai dari 4.03 detik hingga 7.45 detik, dengan waktu eksekusi paling optimal dihasilkan pada percobaan ketiga. Hal ini menunjukkan bahwa semakin banyak iterasi semakin lama waktu yang dibutuhkan meskipun waktu penggerjaan algoritma ini sudah sangat singkat karena proses yang tidak kompleks.

c. Fitness Value :

Nilai fitness terbaik yang dicapai bervariasi di setiap eksperimen, dengan rentang mulai dari -10,86 hingga -5,49. Nilai fitness terbaik dicapai pada jumlah iterasi terbanyak yaitu 61 iterasi, yang menunjukkan bahwa algoritma ini mampu menemukan solusi yang baik, tetapi solusi tersebut tetap terjebak pada titik optimal lokal. Algoritma ini juga dapat dipengaruhi dengan jumlah iterasi yang dilakukan, karena semakin banyak jumlah iterasi, maka kemungkinan untuk mencapai solusi yang lebih baik menjadi semakin tinggi, walaupun hal tersebut belum bisa memastikan hasil yang lebih baik.

2.3.5 Simulated Annealing

Time Execution : 80.30509901046753
Objective Function Value : -45.079934291268245

Untuk algoritma ini, dilakukan percobaan sebanyak 3 kali dengan penjelasan sebagai berikut:

a. Iteration Count dan Stopping Condition:

Jumlah iterasi ditentukan oleh seberapa besar initial temperature dan cooling ratenya. Jika semakin besar kedua nilai tersebut maka jumlah iterasinya akan semakin banyak. Untuk stop conditionnya ditentukan oleh batas temperature yang ingin dicapai. Dari 3 kali percobaan didapatkan hasil terbaik jika iterasinya diperbanyak dengan batas suhunya sangat rendah.

b. Execution Time:

Algoritma ini tidak menggunakan perhitungan atau kode yang rumit sehingga waktu untuk mengeksekusi algoritma ini cenderung lebih cepat dibandingkan algoritma yang lain. Didapat waktu untuk melaksanakan algoritma ini adalah 32 detik.

c. Fitness Value:

Dalam algoritma ini, fitness value yang dihasilkan bergantung pada jumlah iterasinya, jika semakin banyak maka kemungkinan untuk mendapatkan nilai fitness yang bagus semakin besar.

2.3.6 Genetic Algorithm

Untuk setiap metode penggabungan kubus, dilakukan 2 eksperimen:

- Eksperimen pertama : Jumlah sampel tetap 8; Jumlah iterasi 8, 64, 512
- Eksperimen kedua : Jumlah sampel 4, 8, 16; Jumlah iterasi 64

Setiap eksperimen akan dijalankan 3 kali, dan untuk setiap eksperimen dicatat : Kubus awal terbaik, Rata-rata nilai fitness sampel awal, Kubus akhir terbaik, dan Rata-rata nilai fitness sampel akhir. Untuk semua eksperimen digunakan *mutation factor* sebesar 1%

Untuk setiap eksperimen digunakan sampel yang sama

Reference	
AVG	-78.77584631
BEST	-70.24461168

Experiment 1 : Disintegrate

Ex 1 : Disint	Run 1		Run 2		Run 3	
	AVG	-78.31217906	AVG	-80.12226961	AVG	-75.23029327
8	BEST	-74.11148287	BEST	-73.52090621	BEST	-71.73633387
	AVG	-78.41556744	AVG	-74.7034635	AVG	-78.10705511
64	BEST	-69.57890795	BEST	-70.49953068	BEST	-71.06821225
	AVG	-81.8412463	AVG	-79.83390206	AVG	-75.02157027
512	BEST	-79.31798565	BEST	-75.44664716	BEST	-72.52144984

Experiment 1 : Split

Ex 1 : Split		Run 1		Run 2		Run 3	
8	AVG	-81.07359457	AVG	-79.02594912	AVG	-78.06730213	
	BEST	-77.68509513	BEST	-73.42734105	BEST	-69.93151229	
64	AVG	-80.98758569	AVG	-75.01651713	AVG	-79.09003833	
	BEST	-76.44373807	BEST	-70.98394075	BEST	-74.36864056	
512	AVG	-79.16403099	AVG	-82.26932068	AVG	-83.67032887	
	BEST	-70.94979393	BEST	-73.09658522	BEST	-77.18404846	

Experiment 2 : Disintegrate

Ex 2 : Disint		Run 1		Run 2		Run 3	
4	AVG	-80.17308561	AVG	-74.92938222	AVG	-79.94559774	
	BEST	-76.87321488	BEST	-72.93720389	BEST	-77.39333194	
8	AVG	-77.84545206	AVG	-81.76473575	AVG	-76.96933002	
	BEST	-72.67825871	BEST	-75.59577279	BEST	-72.37208947	
16	AVG	-83.78610719	AVG	-75.74262059	AVG	-76.99943594	
	BEST	-75.71946677	BEST	-73.00317789	BEST	-73.52785571	

Experiment 2 : Split

Ex 2 : Split		Run 1		Run 2		Run 3	
4	AVG	-80.11206608	AVG	-83.70098998	AVG	-78.09224063	
	BEST	-74.00713868	BEST	-78.11640099	BEST	-71.7607696	
8	AVG	-77.8318998	AVG	-82.15812526	AVG	-77.80775445	
	BEST	-71.26135059	BEST	-77.41371618	BEST	-69.7471332	
16	AVG	-80.96195011	AVG	-81.06941487	AVG	-81.11644258	
	BEST	-76.05222132	BEST	-77.81276481	BEST	-74.55759882	

Membandingkan hasil algoritma dengan nilai reference menghasilkan nilai berikut

Overall performance (Average performance)			
Method	Disint	Split	
Average	0.4567744316		-1.291684321
Best	-3.527712123		-3.910932185
Reliability (Standard deviation)			
Method	Disint	Split	
Average	0.7261111472		1.261549511
Best	1.993528798		1.133746626

a. Korelasi dengan jumlah iterasi

Ex 1 : Disint		Run 1	Run 2	Run 3		Performance
8	AVG	0.4636672474	AVG	-1.346423307	AVG	0.8875989911
	BEST	-3.866871182	BEST	-3.276294522	BEST	-2.878295963
64	AVG	0.3602788677	AVG	4.07238281	AVG	1.70048429
	BEST	0.6657037386	BEST	-0.2549189957	BEST	-0.8236005613
512	AVG	-3.065399991	AVG	-1.058055755	AVG	-0.1230599038
	BEST	-9.073373965	BEST	-5.202035479	BEST	-2.276838153
Ex 1 : Split		Run 1	Run 2	Run 3		
8	AVG	-2.297748268	AVG	-0.2501028155	AVG	-0.6131023017
	BEST	-7.440483451	BEST	-3.182729367	BEST	-3.436704476
64	AVG	-2.211739379	AVG	3.75932918	AVG	0.4111325915
	BEST	-6.199126386	BEST	-0.7393290612	BEST	-3.687494776
512	AVG	-0.3881846851	AVG	-3.493474374	AVG	-2.925380542
	BEST	-0.7051822453	BEST	-2.85197354	BEST	-3.498864186

Kedua metode menghasilkan sampel terbaik pada kisaran 64, namun menjadi semakin buruk ketika ditingkatkan menjadi 512 iterasi.

b. Korelasi dengan jumlah sampel

Ex 2 : Disint		Run 1	Run 2	Run 3		
4	AVG	-1.397239301	AVG	3.846464084	AVG	0.4264911162
	BEST	-6.628603193	BEST	-2.69259221	BEST	-5.489971886
8	AVG	0.9303942466	AVG	-2.98889439	AVG	-0.08399297048
	BEST	-2.433647031	BEST	-5.351161102	BEST	-3.304095306
16	AVG	-5.010260879	AVG	3.033225712	AVG	-0.06687493358
	BEST	-5.474855091	BEST	-2.758566208	BEST	-3.838888443
Ex 2 : Split		Run 1	Run 2	Run 3		
4	AVG	-1.336219779	AVG	-4.925143676	AVG	-1.859252594
	BEST	-3.762526992	BEST	-7.871789304	BEST	-4.383491403
8	AVG	0.9439465069	AVG	-3.382278959	AVG	-0.4900802001
	BEST	-1.016738909	BEST	-7.169104493	BEST	-2.562788305
16	AVG	-2.186103801	AVG	-2.293568561	AVG	-2.273422878
	BEST	-5.807609631	BEST	-7.568153128	BEST	-5.896249966

Sampel yang lebih besar menghasilkan sampel yang sedikit lebih baik, namun untuk split terjadi kejanggalan dimana sampel yang lebih besar menghancurkan sampel akhir

c. Kesimpulan keseluruhan

Algoritma Genetic Algorithm tidak dapat menghasilkan hasil yang lebih baik, dan bahkan secara rata-rata menghasilkan sampel yang lebih buruk

+ GA Data

https://docs.google.com/spreadsheets/d/15__FUPUJPa5orFLrOOHafFizbZegOP4BtKBw6rjW-MU/edit?usp=sharing

Bab III

Kesimpulan dan Saran

3.1 Kesimpulan

Dari serangkaian eksperimen yang dilakukan, beberapa algoritma pencarian optimal menunjukkan kelebihan dan kekurangannya masing-masing:

- Steepest Ascent Hill-Climbing:
Meskipun mencapai solusi lokal optimal dengan iterasi yang efisien, algoritma ini mudah terjebak pada local optima. Waktu eksekusi relatif lama akibat kompleksitas perhitungan dan proses pencarian. Fitness value yang diperoleh cukup baik namun belum mencapai global optimal, menandakan batasan algoritma ini pada posisi awal dan gradien solusi yang tersedia.
- Hill-Climbing with Sideways Move:
Algoritma ini mencoba mengatasi keterjebakan pada local optima dengan gerakan sideways, meskipun hasil akhirnya kurang konsisten dalam mencapai global optimal.
- Random Restart Hill-Climbing:
Algoritma ini lebih berpeluang menemukan solusi yang lebih baik dibandingkan satu proses hill-climbing tunggal. Waktu eksekusi cenderung lebih lama seiring bertambahnya restart, tetapi menunjukkan kemampuan mencapai solusi optimal lebih cepat pada beberapa percobaan. Fitness value yang dicapai lebih baik dengan restart yang lebih banyak, meskipun tetap terkendala local optima.
- Stochastic Hill-Climbing:
Jumlah iterasi yang lebih rendah membuat algoritma ini berjalan lebih cepat, tetapi kemampuannya dalam menemukan solusi optimal terbatas karena sering terjebak di local optima. Fitness value yang diperoleh cukup bervariasi, dan hasil terbaik biasanya dicapai pada iterasi yang lebih tinggi.
- Simulated Annealing:
Algoritma ini menunjukkan keunggulan dalam waktu eksekusi yang relatif cepat dan kemampuan menyesuaikan jumlah iterasi sesuai nilai initial temperature dan cooling rate. Fitness value dapat ditingkatkan dengan memperbanyak iterasi, menjadikan algoritma ini pilihan menarik untuk mencari solusi optimal secara global.

- Genetic Algorithm:

Algoritma ini kurang optimal dalam menemukan solusi yang lebih baik, bahkan cenderung memberikan hasil rata-rata yang lebih buruk.

Hasil eksperimen menunjukkan korelasi negatif antara peningkatan iterasi atau sampel terhadap kualitas solusi akhir, terutama pada metode "split".

Melalui data yang kami peroleh dari pengujian masing-masing algoritma, kami mendapatkan bahwa algoritma steepest ascent dan saudaranya steepest ascent with hill-climbing merupakan algoritma terbaik karena menghasilkan nilai paling dekat dengan global maxima (0). Dengan perbedaan hill-climbing harus memperhitungkan tetangganya sehingga lebih berat dibanding steepest hill-climbing.

3.2 Saran

- Pemilihan Algoritma Berdasarkan Kompleksitas dan Kebutuhan:

Untuk pencarian solusi yang cepat dan tidak memerlukan hasil global optimal, Stochastic Hill-Climbing dan Hill-Climbing with Sideways Move dapat dipertimbangkan. Simulated Annealing lebih sesuai bila diperlukan pencarian solusi global optimal dengan waktu eksekusi yang singkat.

- Mengatasi Keterjebakan pada Local Optima:

Menggabungkan Random Restart Hill-Climbing dengan Simulated Annealing dapat memperluas cakupan solusi dan membantu keluar dari local optima. Genetic Algorithm sebaiknya digunakan dengan perbaikan tambahan, seperti menambah mekanisme seleksi atau penyesuaian faktor mutasi untuk meningkatkan kualitas solusi.

Bab IV

Pembagian Tugas

NIM	Nama	Tugas
13522121	Jonathan Emmanuel Saragih	Algoritma, Backend
13522130	Justin Aditya Putra Prabakti	Algoritma, Backend
13522145	Farrel Natha Saskoro	Algoritma, Backend
13522163	Atqiya Haydar Luqman	Algoritma, Frontend

Bab V

Referensi

1. Russell, S., & Norvig, P. (2020). *Artificial Intelligence: A Modern Approach* (Edisi ke-4). Pearson.
2. Institut Teknologi Bandung. (n.d.). *Beyond Classical Search: Local Search* [PDF]. Diakses dari https://cdn-edunex.itb.ac.id/53145-Artificial-Intelligence-Parallel-Class/194228-Beyond-Classical-Search/1693804818592_IF3170_Materi03_Seg01_BeyondClassicalSearch_LocalSearch.pdf
3. Institut Teknologi Bandung. (n.d.). *Beyond Classical Search: Hill Climbing* [PDF]. Diakses dari https://cdn-edunex.itb.ac.id/53145-Artificial-Intelligence-Parallel-Class/194228-Beyond-Classical-Search/1693804849395_IF3170_Materi3_Seg03_BeyondClassicalSearch_HillClimbing.pdf
4. Institut Teknologi Bandung. (n.d.). *Beyond Classical Search: Simulated Annealing* [PDF]. Diakses dari https://cdn-edunex.itb.ac.id/53145-Artificial-Intelligence-Parallel-Class/194228-Beyond-Classical-Search/1693804872404_IF3170_Materi03_Seg04_BeyondClassicalSearch_SimulatedAnnealing.pdf
5. Institut Teknologi Bandung. (n.d.). *Beyond Classical Search* [PDF]. Diakses dari https://cdn-edunex.itb.ac.id/storages/files/1727405202098_IF3170_Materi03_Seg05_BeyondClassicalSearch.pdf
6. Institut Teknologi Bandung. (n.d.). *Adversarial Search* [PDF]. Diakses dari https://cdn-edunex.itb.ac.id/53145-Artificial-Intelligence-Parallel-Class/194228-Beyond-Classical-Search/1693969282598_IF3170_Materi04_AdversarialSearch.pdf
7. Institut Teknologi Bandung. (n.d.). *Introduction to Problem Solving Agent* [PDF]. Diakses dari https://cdn-edunex.itb.ac.id/43590-Artificial-Intelligence-Parallel-Class/100866-Minggu-2/42974-Introduction-to-Problem-Solving-Agent/1661740598287_Materi_AI_ProblemSolving.pdf
8. Magisch Vierkant. (n.d.). *Magic Features*. Diakses dari <https://www.magischvierkant.com/three-dimensional-eng/magic-features/>
9. Trump Magic Squares. (n.d.). *Magic Cubes*. Diakses dari <https://www.trump.de/magic-squares/magic-cubes/cubes-1.html>
10. Wikipedia contributors. (n.d.). *Magic cube*. Di Wikipedia, Ensiklopedia Bebas. Diakses dari https://en.wikipedia.org/wiki/Magic_cube