Tugas Besar 1 IF3070 Dasar Intelegensi Artifisial

Pencarian Solusi Diagonal Magic Cube dengan Local Search

Disusun oleh:

Kelompok 13

Rashid May	/ 18222014
Lutfi Khairul Amal	/ 18222018
Filbert Fuvian	/ 18222024
Qady Zakka Raymaula	/ 18222038

Program Studi Sistem dan Teknologi Informasi

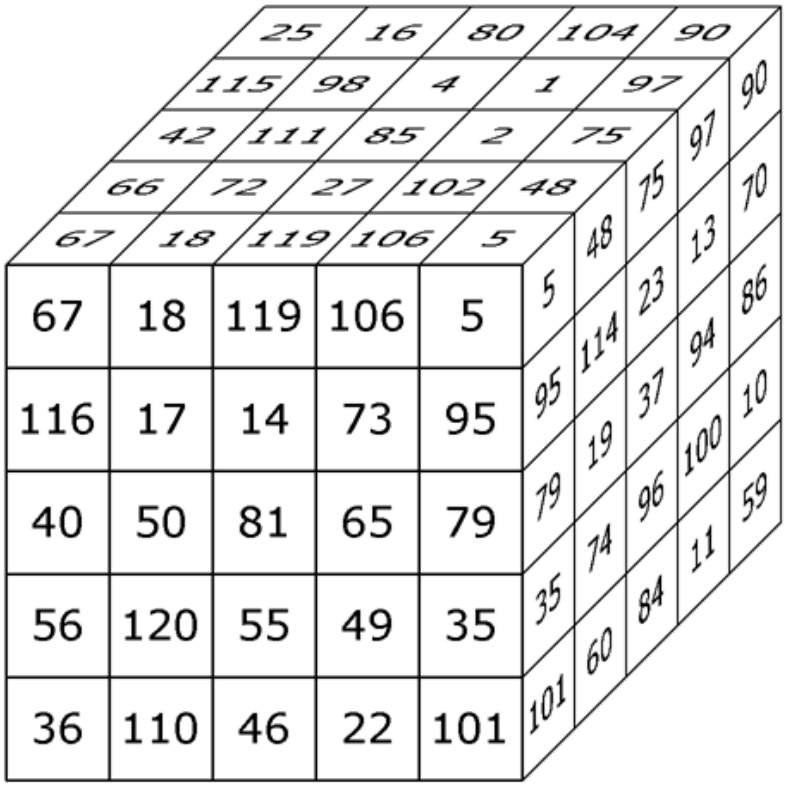
Sekolah Teknik Elektro dan Informatika

Institut Teknologi Bandung

2024

Daftar Isi

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
- 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

Dalam persoalan, diberikan diagonal magic cube 5x5x5 dengan susunan random dan tidak memenuhi kondisi ideal. Penyelesaian persoalan harus dilakukan melalui pendekatan algoritma local search, simulated annealing, dan genetic algorithm dengan setiap iterasinya diperkenankan untuk menukar posisi antara kubus kecil dalam diagonal magic cube. Pengukuran ketercapaian kondisi

ideal dari diagonal magic cube dapat diukur dengan menggunakan objective function.

Pembahasan

Persoalan ini akan kami coba selesaikan dengan menggunakan program berbahasa python. Berikut adalah hasil penyelesaian kami.

Pemilihan Objective Function

```
def calculate_magic_score(cube):
    n = cube.shape[0]
    magic_constant = (n*(n**3 + 1)) / 2
    score = 0

    # Row, column, and pillar checks
    for i in range(n):
        for j in range(n):
            # Sum rows along the z-axis
            row_sum = np.sum(cube[i, j, :])
            score += abs(magic_constant - row_sum)

            # Sum columns along the y-axis
            col_sum = np.sum(cube[i, :, j])
            score += abs(magic_constant - col_sum)

            # Sum pillars along the x-axis
            pillar_sum = np.sum(cube[:, i, j])
            score += abs(magic_constant - pillar_sum)

    # Face diagonals checks (on each layer perpendicular to each axis)
    for i in range(n):
        # Diagonals on xy planes (constant z)
        face_diag1 = np.sum([cube[j, j, i] for j in range(n)])
        face_diag2 = np.sum([cube[j, n - j - 1, i] for j in range(n)])
        score += abs(magic_constant - face_diag1)
        score += abs(magic_constant - face_diag2)

        # Diagonals on xz planes (constant y)
        face_diag3 = np.sum([cube[j, i, j] for j in range(n)])
```

```
        face_diag4 = np.sum([cube[n - j - 1, i, j] for j in range(n)])
        score += abs(magic_constant - face_diag3)
        score += abs(magic_constant - face_diag4)

        # Diagonals on yz planes (constant x)
        face_diag5 = np.sum([cube[i, j, j] for j in range(n)])
        face_diag6 = np.sum([cube[i, j, n - j - 1] for j in range(n)])
        score += abs(magic_constant - face_diag5)
        score += abs(magic_constant - face_diag6)

    # Face diagonals checks (on each layer perpendicular to each axis)
    for i in range(n):
        # Diagonals on xy planes (constant z)
        face_diag1 = np.sum([cube[j, j, i] for j in range(n)])
        face_diag2 = np.sum([cube[j, n - j - 1, i] for j in range(n)])
        score += abs(magic_constant - face_diag1)
        score += abs(magic_constant - face_diag2)

        # Diagonals on xz planes (constant y)
        face_diag3 = np.sum([cube[j, i, j] for j in range(n)])
        face_diag4 = np.sum([cube[n - j - 1, i, j] for j in range(n)])
        score += abs(magic_constant - face_diag3)
        score += abs(magic_constant - face_diag4)

        # Diagonals on yz planes (constant x)
        face_diag5 = np.sum([cube[i, j, j] for j in range(n)])
        face_diag6 = np.sum([cube[i, j, n - j - 1] for j in range(n)])
        score += abs(magic_constant - face_diag5)
        score += abs(magic_constant - face_diag6)

    return score
```

Objective function dibuat untuk mengevaluasi seberapa baik kondisi kubus dalam memenuhi kriteria diagonal magic cube. Dalam objective function tersebut, function sudah memenuhi semua aspek relevan (baris, kolom, tiang, diagonal sisi, dan diagonal ruang) yang perlu diperhitungkan. Function juga menggunakan nilai absolut berupa selisih antara jumlah elemen yang diperhitungkan dengan magic number untuk memudahkan evaluasi kondisi kubus.

Objective function tersebut bekerja dengan awalnya menginisialisasi total_score ke 0. Selanjutnya, iterasi dilakukan dengan melakukan pemeriksaan terhadap elemen

yang diperhitungkan (baris, kolom, tiang, diagonal ruang, dan diagonal sisi). Pada setiap iterasi, akan didapat jumlah dalam integer yang akan diambil selisihnya antara jumlah tersebut terhadap magic number untuk ditambahkan ke dalam total_score. Terakhir, fungsi akan mengembalikan total_score atau kondisi kubus dalam memenuhi kriteria diagonal magic cube. Kubus memenuhi kriteria diagonal magic cube secara sempurna jika total_score = 0. Semakin besar nilai total_score, berarti kubus semakin tidak memenuhi kriteria diagonal magic cube.

Penjelasan implementasi algoritma local search

- Hill-Climbing (Stochastic)

```
def stochastic_hill_climbing(cube):
    n = cube.shape[0]
    current_cube = cube.copy()
    current_score = calculate_magic_score(current_cube)

    i = 0
    while (i < 10000):
        neighbor = generate_random_cube(n)
        neighbor_score = calculate_magic_score(neighbor)

        # Jika tetangga lebih baik, pindah
        if neighbor_score < current_score:
            current_cube = neighbor
            current_score = neighbor_score

        i+=1

    return current_cube, current_score
```

Algoritma tersebut bekerja dengan men-generate random cube di setiap iterasinya. Selanjutnya, random cube yang di generate akan dibandingkan nilai objective functionnya dengan current cube. Jika objective function dari random cube lebih baik dari current cube, maka current cube akan di overwrite atau digantikan dengan random cube pada iterasi tersebut. Iterasi pada algoritma stochastic hill-climbing ini dibatasi hingga 10000 iterasi.

- Simulated Annealing

```
def simulated_annealing(cube, initial_temperature=100,
cooling_rate=0.99, min_temperature=1):
```

```
  n = cube.shape[0]
   current_cube = cube.copy()
   current_score = calculate_magic_score(current_cube)
   temperature = initial_temperature

   while temperature > min_temperature:
      # Generate a neighbor
      neighbor = generate_random_cube(n)
      neighbor_score = calculate_magic_score(neighbor)

      # Hitung perbedaan skor
      delta_score = neighbor_score - current_score

      # Tentukan apakah akan menerima tetangga baru
      if delta_score < 0 or math.exp(-delta_score / temperature) >
random.random():
         current_cube = neighbor
         current_score = neighbor_score

      # Turunkan suhu
      temperature *= cooling_rate

   return current_cube, current_score
```

Simulated Annealing bekerja hampir sama dengan Stochastic Hill Climbing. Perbedaan Simulated Annealing jika dibandingkan dengan Stochastic Hill Climbing terletak pada kemampuannya untuk mengambil langkah turun (mengambil kubus dengan score yang lebih rendah) berdasarkan probabilitas. Hal ini dilakukan untuk meminimalisir kemungkinan algoritma terjebak pada local optimum.

Algoritma tersebut bekerja dengan men-generate random cube sebagai neighbor di setiap iterasinya. Pada setiap iterasinya, didapatkan selisih objective function antara random cube dan current cube. Selisih objective function digunakan untuk menentukan apakah current cube akan di overwrite dengan neighbor atau tidak. Current cube akan di overwrite dengan neighbor jika :

- delta_score > 0 yang berarti neighbor lebih baik dari current cube
- memenuhi probabilitas yang dihitung menggunakan fungsi eksponensial dari nilai negatif delta_score yang dibagi dengan suhu (temperature).

Setiap iterasi yang telah berjalan akan mengurangi temperature dari algoritma simulated annealing yang secara berkala akan menurunkan probabilitas.

- Genetic Algorithm

```
def genetic_algorithm(n, population_size=100, generations=1000, mutation_rate=0.1):
```

```
  # Initialize population

   population = [generate_random_cube(n) for _ in range(population_size)]
   scores = [calculate_magic_score(cube) for cube in population]
   score_per_iteration = dict()
   start_time = time.time()

   for generation in range(generations):
      # Select parents based on their weighted scores
      selected_parents = weighted_random_selection(population, scores)

      # Create new population
      new_population = selected_parents.copy()

      # Crossover and mutation
      while len(new_population) < population_size:
         parent1, parent2 = weighted_random_selection(population, scores)
         child = crossover(parent1, parent2)

         # Mutate the child
         if random.random() < mutation_rate:
            child = generate_random_neighbor(child)

         new_population.append(child)

      population = new_population
      scores = [calculate_magic_score(cube) for cube in population]

      # Print the best score of the generation
      best_score = min(scores)
```

```
        score_per_iteration[generation] = best_score

    elapsed_time = time.time() - start_time

    # Return the best solution found
    best_index = scores.index(min(scores))

    return population[best_index], min(scores), score_per_iteration,
elapsed_time
```

```
def weighted_random_selection(population, scores):
```

```
   # Calculate inverse scores for weighting
    inverse_scores = [(1 / score) for score in scores]
    total_inverse_score = sum(inverse_scores)
    probabilities = [inv_score / total_inverse_score for inv_score in
inverse_scores]

    return random.choices(population, weights=probabilities, k=2)
```

```
def crossover(parent1, parent2):
```

```
   n = parent1.shape[0]
    child = np.zeros_like(parent1)

    # Simple crossover: take half from parent1 and half from parent2
    for i in range(n):
        for j in range(n):
            if random.random() < 0.5:
                child[i, j, :] = parent1[i, j, :]
            else:
                child[i, j, :] = parent2[i, j, :]

    return child
```

Genetic Algorithm merupakan algoritma paling rumit dalam local search. Algoritma ini mensimulasikan sebuah civilization dari generasi ke generasi dimana adanya reproduction, selection, dan mutation. Algoritma ini bekerja dengan pertama menentukan besar populasi dan banyaknya generasi yang akan dijalankan. Lalu algoritma akan menjalankan iterasi sebanyak generasi dimana pada setiap generasi akan dilakukan update pada populasi melalui metode selection, reproduction, dan

mutation. Sebagian besar metode tersebut ditentukan dengan random dengan parent dengan fitness (score) yang baik memiliki probabilitas yang lebih tinggi melalui selection.

Hasil eksperimen dan analisis

Data hasil eksperimen dapat dilihat di link berikut :

Experiment DAI Tubes 1

Hasil eksperimen dari Stochastic hill-climbing dan Simulated Annealing pada state awal yang sama menunjukkan kalau Simulated Annealing memiliki objective function yang lebih tinggi dengan jumlah iterasi yang besar. Sementara itu, algoritma stochastic memiliki objective function yang lebih tinggi pada jumlah iterasi yang lebih kecil. Hal tersebut dibuktikan dengan melihat percobaan 1 dengan iterasi 10000 dimana algoritma *stochastic hill climbing* memiliki objective function lebih baik dibanding simulated annealing. Sementara itu, pada percobaan 2 dan 3 dengan jumlah iterasi 30000 dan 60000 menunjukkan kalau *simulated annealing* memiliki objective function yang lebih baik dibanding s*tochastic hill climbing*.

Hasil eksperimen dari Genetic Algorithm menunjukkan kalau variasi jumlah populasi dan jumlah iterasi berpengaruh pada objective function algoritma. Pengaruh variasi jumlah populasi dapat dilihat dengan membandingkan hasil percobaan 2 dan 4 dengan 100 populasi pada percobaan 2 dan 20 populasi pada percobaan 4. Keduanya menggunakan jumlah iterasi yang sama yaitu 1000 dan hasilnya percobaan 2 dengan populasi yang lebih tinggi memiliki objective function yang lebih baik dibanding percobaan 4 dengan populasi yang lebih rendah. Sementara itu, pengaruh variasi jumlah iterasi dapat dilihat dengan membandingkan hasil percobaan 1 dan 2 dengan 200 iterasi pada percobaan 1 dan 1000 iterasi pada percobaan 2. Menggunakan populasi yang sama, percobaan 2 dengan jumlah iterasi yang lebih tinggi memiliki objective function yang lebih baik dari percobaan 1. Hal tersebut juga disertai dengan selisih waktu yang tinggi.

Kesimpulan dan Saran

Kesimpulan

- Dalam menyelesaikan masalah magic cube, algoritma Simulated Annealing akan memiliki objective function yang lebih baik pada jumlah iterasi yang semakin besar dari suatu batas tertentu jika dibandingkan dengan algoritma Stochastic Hill-climbing.
- Dalam menyelesaikan masalah magic cube, variasi jumlah populasi dan jumlah iterasi berpengaruh pada semakin baiknya objective function. Semakin tinggi jumlah populasi dan jumlah iterasi, akan semakin baik pula *objective function* nya.

Saran

- Algoritma Simulated Annealing dapat digunakan untuk menyelesaikan masalah magic cube jika durasi pencarian tidak menjadi masalah.
- Algoritma Stochastic Hill-climbing dapat digunakan untuk menyelesaikan masalah magic cube jika diperlukan durasi pencarian yang relatif cepat.
- Jumlah populasi dan jumlah iterasi pada algoritma Genetic Algorithm yang tinggi atau besar dapat digunakan jika durasi pencarian tidak menjadi masalah.

Pembagian Tugas

Nama Mahasiswa	:	Rashid May
NIM	:	18222014

NO	Kegiatan
1	Melakukan testing eksperimen stochastic hill-climbing
2	Melakukan testing eksperimen simulated annealing
3	Merapikan sheets data
4	Melengkapi dan merapikan laporan

Nama Mahasiswa	:	Lutfi Khairul Amal
NIM	:	18222018

NO	Kegiatan
1	Mengisi laporan deskripsi persoalan
2	Mengisi laporan pembahasan pemilihan objective function
3	Mengisi laporan pembahasan penjelasan implementasi algoritma
4	Mengisi laporan pembahasan hasil eksperimen dan analisis
5	Mengisi laporan kesimpulan dan saran

Nama Mahasiswa	:	Filbert Fuvian
NIM	:	18222024

NO	Kegiatan
1	Membuat program
2	Melakukan testing experiment genetic algorithm
3	Melengkapi dan merapikan laporan

Nama Mahasiswa	:	Qady Zakka Raymaula
NIM	:	18222038

NO	Kegiatan
1	Melengkapi dan merapihkan laporan
2	Memasukkan data eksperimen kedalam sheets
3	Merapikan sheets data

Referensi

- Features of the magic cube - Magisch vierkant
- Perfect Magic Cubes
- Magic cube - Wikipedia