

Laporan Praktikum Kecerdasan Buatan

Semester Ganjil 2024/2025

Jurusan Teknik Elektro
Program Studi S1 Teknik Elektro
Fakultas Teknik
Universitas Tidar

Praktikum ke- : 4
Judul Praktikum : Pencarian Terbimbing: Heuristic Search

Nama : Jerico Christianto
NIM : 2220501082

“Laporan praktikum ini saya kerjakan dan selesaikan dengan sebaik-baiknya tanpa melakukan tindak kecurangan. Apabila di kemudian hari ditemukan adanya kecurangan pada laporan ini, maka saya bersedia menerima konsekuensinya.”

Tertanda,



Jerico Christianto

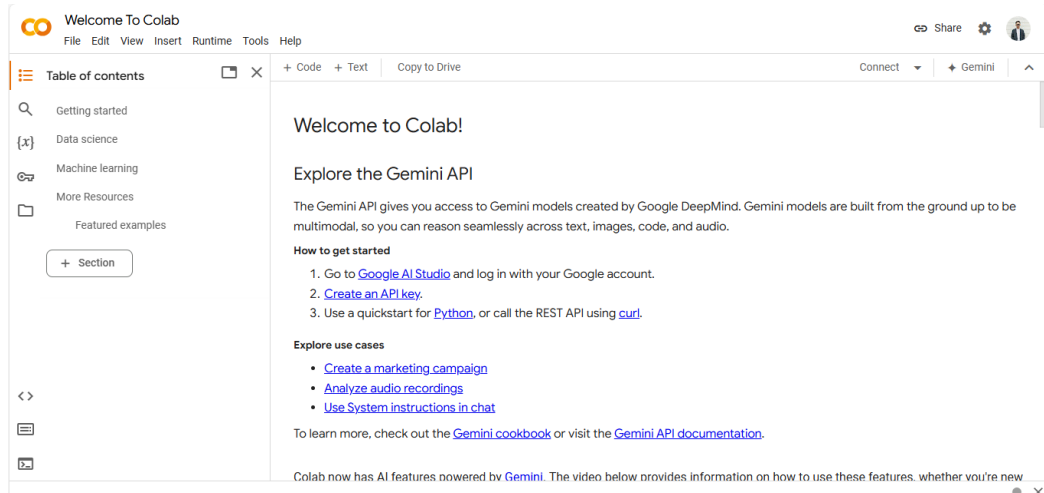
2220501082

A. Tujuan Praktikum

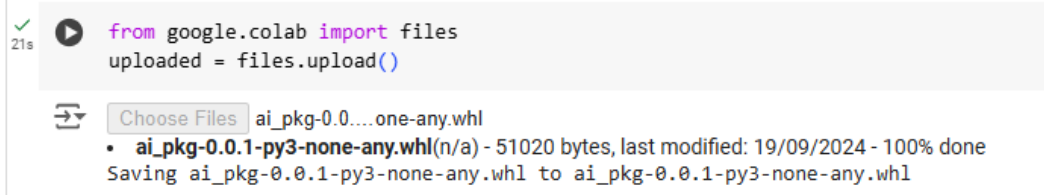
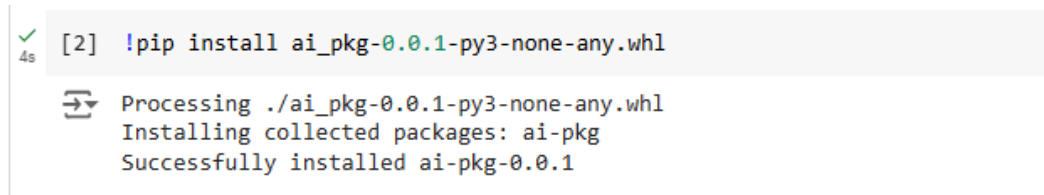
1. Memahami konsep dasar algoritma Pencarian Terbimbing: Heuristic Search
2. Menerapkan algoritma Pencarian Terbimbing: Heuristic Search dalam bahasa pemrograman Python

B. Langkah Praktikum

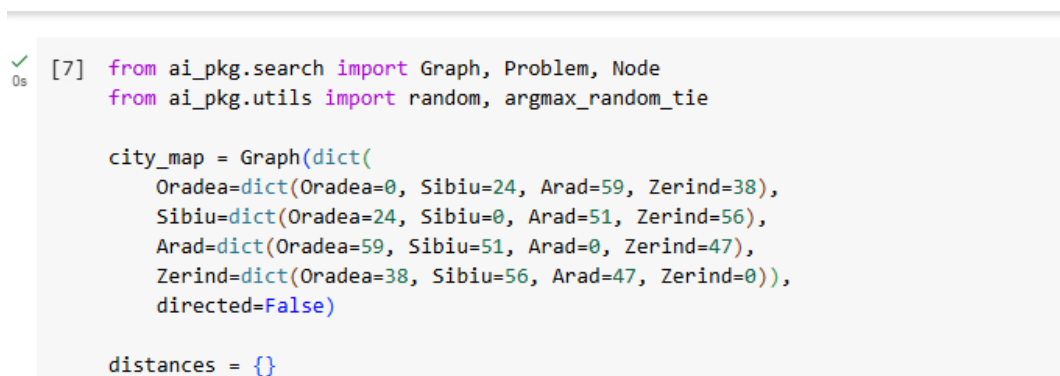
1. Membuka Google Colab



2. Mengunggah file package

3. Jalankan perintah "!pip install *nama_package*"

4. Tuliskan kode program seperti berikut:



```

class TSP_problem(Problem):
    def generate_neighbour(self, state):
        neighbour_state = state[:]
        left = random.randint(0, len(neighbour_state) - 1)
        right = random.randint(0, len(neighbour_state) - 1)
        if left > right:
            left, right = right, left
        neighbour_state[left:right+1] = reversed(neighbour_state[left:right+1])
        return neighbour_state

    def actions(self, state):
        return [self.generate_neighbour]

    def result(self, state, action):
        return action(state)

    def path_cost(self, state):
        cost = 0
        for i in range(len(state) - 1):
            current_city = state[i]
            next_city = state[i+1]
            cost += distances[current_city][next_city]
        cost += distances[state[0]][state[-1]]
        return cost

    def value(self, state):
        return -1 * self.path_cost(state)

```

5. Tuliskan kode program seperti berikut:

```

[10] def hill_climbing(problem):
    def find_neighbors(state, number_of_neighbors=100):
        neighbors = []
        for i in range(number_of_neighbors):
            new_state = problem.generate_neighbour(state)
            neighbors.append(Node(new_state))
            state = new_state
        return neighbors

    current = Node(problem.initial)
    while True:
        neighbors = find_neighbors(current.state)
        if not neighbors:
            break
        neighbor = argmax_random_tie(neighbors, key=lambda node:
        problem.value(node.state))
        if problem.value(neighbor.state) <= problem.value(current.state):
            break
        current.state = neighbor.state
    return current.state

if __name__ == '__main__':
    all_cities = []
    cities_graph = city_map.graph_dict
    for city_1 in cities_graph.keys():
        distances[city_1] = {}
        if city_1 not in all_cities:
            all_cities.append(city_1)
    for city_2 in cities_graph.keys():
        if (cities_graph.get(city_1).get(city_2) is not None):
            distances[city_1][city_2] = cities_graph.get(city_1).get(city_2)

```

```
tsp_problem = TSP_problem(all_cities)
result = hill_climbing(tsp_problem)
print(result)
cost = tsp_problem.path_cost(result)
print('cost:', cost)
```

```
['Oradea', 'Sibiu', 'Arad', 'Zerind']
cost: 160
```

C. Hasil Praktikum

1. Mengunggah file

```
from google.colab import files
uploaded = files.upload()
```

Choose Files ai_pkg-0.0....one-any.whl

- ai_pkg-0.0.1-py3-none-any.whl(n/a) - 51020 bytes, last modified: 19/09/2024 - 100% done

Saving ai_pkg-0.0.1-py3-none-any.whl to ai_pkg-0.0.1-py3-none-any.whl

Mengunggah file di Google Colab dilakukan ketika kita membutuhkan library, data, atau file konfigurasi yang tidak tersedia secara default di lingkungan Colab.

2. Menginstall file

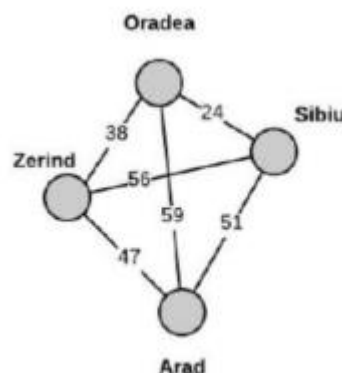
```
[2] !pip install ai_pkg-0.0.1-py3-none-any.whl
```

Processing ./ai_pkg-0.0.1-py3-none-any.whl
Installing collected packages: ai-pkg
Successfully installed ai-pkg-0.0.1

File yang berisi library, data, atau file konfigurasi ini diinstal agar dapat digunakan pada saat menjalankan kode program.

3. Graph

Diberikan sebuah node pencarian seperti berikut :



Node pencarian diatas menunjukkan cara mencari rute terpendek untuk mengunjungi semua kota dan kembali ke titik awal menggunakan metode Hill Climbing. Masalah ini dikenal sebagai Traveling Salesman Problem (TSP). Kita bisa bayangkan kota-kota sebagai titik dan jalan di antara kota-kota sebagai garis. Tujuannya adalah menemukan jalur terpendek yang melewati semua titik.

4. Kode Program :

```

✓ [7] from ai_pkg.search import Graph, Problem, Node
0s    from ai_pkg.utils import random, argmax_random_tie

city_map = Graph(dict(
    Oradea=dict(Oradea=0, Sibiu=24, Arad=59, Zerind=38),
    Sibiu=dict(Oradea=24, Sibiu=0, Arad=51, Zerind=56),
    Arad=dict(Oradea=59, Sibiu=51, Arad=0, Zerind=47),
    Zerind=dict(Oradea=38, Sibiu=56, Arad=47, Zerind=0)),
    directed=False)

distances = {}

class TSP_problem(Problem):
    def generate_neighbour(self, state):
        neighbour_state = state[:]
        left = random.randint(0, len(neighbour_state) - 1)
        right = random.randint(0, len(neighbour_state) - 1)
        if left > right:
            left, right = right, left
        neighbour_state[left:right+1] = reversed(neighbour_state[left:right+1])
        return neighbour_state

    def actions(self, state):
        return [self.generate_neighbour]

    def result(self, state, action):
        return action(state)

    def path_cost(self, state):
        cost = 0
        for i in range(len(state) - 1):
            current_city = state[i]
            next_city = state[i+1]
            cost += distances[current_city][next_city]
        cost += distances[state[0]][state[-1]]
        return cost

    def value(self, state):
        return -1 * self.path_cost(state)

```

Awalnya, kita membuat sebuah peta (graf) yang menunjukkan hubungan antar kota. Peta ini dibuat menggunakan kamus (dictionary) untuk menyimpan informasi tentang kota dan jarak antar kota. Karena dalam masalah TSP, arah perjalanan tidak penting (dari kota A ke B sama dengan dari B ke A), maka kita atur peta ini agar tidak memiliki arah (`directed=False`). Setiap kota dalam peta ini akan otomatis menjadi sebuah titik (simpul) yang dikelola oleh sistem.

5. Kode Program

```

def hill_climbing(problem):
    def find_neighbors(state, number_of_neighbors=100):
        neighbors = []
        for i in range(number_of_neighbors):
            new_state = problem.generate_neighbour(state)
            neighbors.append(Node(new_state))
            state = new_state
        return neighbors

    current = Node(problem.initial)
    while True:
        neighbors = find_neighbors(current.state)
        if not neighbors:
            break
        neighbor = argmax_random_tie(neighbors, key=lambda node:
            problem.value(node.state))
        if problem.value(neighbor.state) <= problem.value(current.state):
            break
        current.state = neighbor.state
    return current.state

if __name__ == '__main__':
    all_cities = []
    cities_graph = city_map.graph_dict
    for city_1 in cities_graph.keys():
        distances[city_1] = {}
        if(city_1 not in all_cities):
            all_cities.append(city_1)
        for city_2 in cities_graph.keys():
            if(cities_graph.get(city_1).get(city_2) is not None):
                distances[city_1][city_2] = cities_graph.get(city_1).get(city_2)

    tsp_problem = TSP_problem(all_cities)
    result = hill_climbing(tsp_problem)
    print(result)
    cost = tsp_problem.path_cost(result)
    print('cost:', cost)

```

 ['Oradea', 'Sibiu', 'Arad', 'Zerind']
 cost: 160

Untuk menyelesaikan masalah TSP, kita akan membuat sebuah kelas baru bernama TSP_problem. Kelas ini akan digunakan sebagai dasar untuk mencari solusi terbaik. Kelas TSP_problem ini harus memiliki empat fungsi utama:

- actions** : Menentukan langkah-langkah apa saja yang bisa kita lakukan dalam mencari solusi.
- result** : Menunjukkan hasil dari suatu langkah yang kita ambil.
- path_cost** : Menghitung total biaya (jarak) dari suatu rute yang kita tempuh.
- value** : Menilai seberapa baik suatu rute.

6. Hasil

```

⇒ ['Oradea', 'Sibiu', 'Arad', 'Zerind']
cost: 160

```

Gambar diatas menunjukkan hasil dari sebuah proses pencarian solusi untuk masalah Traveling Salesman Problem (TSP). Hasil ini mengindikasikan bahwa algoritma telah menemukan satu kemungkinan rute terpendek untuk mengunjungi semua kota yang ada.

Rincian Hasil:

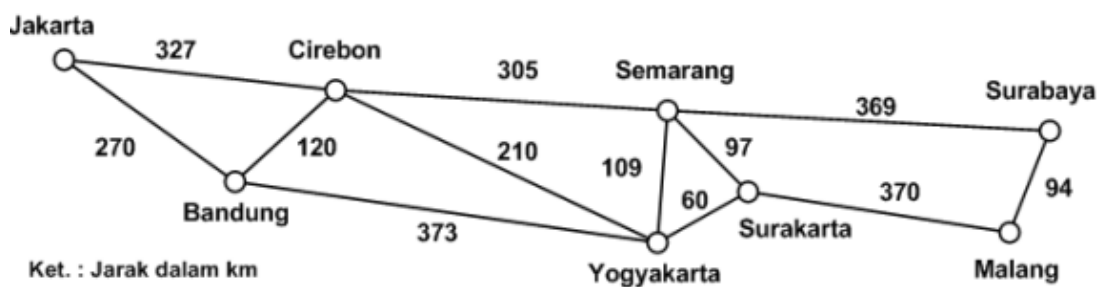
- ['Oradea', 'Sibiu', 'Arad', 'Zerind']: Ini adalah urutan kota yang harus dikunjungi untuk mencapai rute terpendek. Artinya, perjalanan dimulai dari kota Oradea, kemudian menuju Sibiu, lalu Arad, dan berakhir di Zerind.
- cost: 160: Angka 160 ini merepresentasikan total jarak atau biaya yang dibutuhkan untuk menempuh rute tersebut. Dengan kata lain, jarak total dari Oradea ke Sibiu, kemudian ke Arad, dan akhirnya ke Zerind adalah 160.

D. Kendala Praktikum

Tidak terdapat kendala pada praktikum ini, hanya sedikit kesalahan pada penulisan yang dapat segera diperbaiki.

E. Studi Kasus

Buatlah program TSP dengan menggunakan graph berikut:



1. Kode Program:

A. Graph

```

from ai_pkg.search import Graph, Problem, Node
from ai_pkg.utils import random, argmax_random_tie

city_map = Graph(dict(
    Jakarta=dict(Jakarta=0, Cirebon=327, Bandung=270, Yogyakarta=643, Semarang=632, Surakarta=723, Malang=1073, Surabaya=1001),
    Cirebon=dict(Jakarta=327, Cirebon=0, Bandung=120, Yogyakarta=210, Semarang=305, Surakarta=270, Malang=640, Surabaya=674),
    Bandung=dict(Jakarta=270, Cirebon=120, Bandung=0, Yogyakarta=373, Semarang=425, Surakarta=433, Malang=803, Surabaya=851),
    Semarang=dict(Jakarta=632, Cirebon=305, Bandung=425, Yogyakarta=109, Semarang=0, Surakarta=97, Malang=463, Surabaya=369),
    Yogyakarta=dict(Jakarta=643, Cirebon=210, Bandung=373, Yogyakarta=0, Semarang=109, Surakarta=60, Malang=430, Surabaya=478),
    Surakarta=dict(Jakarta=723, Cirebon=270, Bandung=433, Yogyakarta=60, Semarang=97, Surakarta=0, Malang=370, Surabaya=464),
    Malang=dict(Jakarta=1073, Cirebon=640, Bandung=803, Yogyakarta=430, Semarang=463, Surakarta=370, Malang=0, Surabaya=94),
    Surabaya=dict(Jakarta=1001, Cirebon=674, Bandung=851, Yogyakarta=478, Semarang=369, Surakarta=464, Malang=94, Surabaya=0)),
    directed=False)

distances = {}

class TSP_problem(Problem):
    def generate_neighbour(self, state):
        neighbour_state = state[:]
        left = random.randint(0, len(neighbour_state) - 1)
        right = random.randint(0, len(neighbour_state) - 1)
        if left > right:
            left, right = right, left
        neighbour_state[left:right+1] = reversed(neighbour_state[left:right+1])
        return neighbour_state

    def actions(self, state):
        return [self.generate_neighbour]

    def result(self, state, action):
        return action(state)

    def path_cost(self, state):
        cost = 0
        for i in range(len(state) - 1):
            current_city = state[i]
            next_city = state[i+1]
            cost += distances[current_city][next_city]
        cost += distances[state[0]][state[-1]]
        return cost

    def value(self, state):
        return -1 * self.path_cost(state)

```

- `from ai_pkg.search import Graph, Problem, Node`: Mengimpor kelas-kelas yang diperlukan dari modul `ai_pkg.search`. Kelas `Graph` digunakan untuk merepresentasikan peta kota, `Problem` adalah kelas abstrak untuk masalah pencarian, dan `Node` adalah representasi dari sebuah keadaan dalam pencarian.
- `from ai_pkg.utils import random, argmax_random_tie`: Mengimpor fungsi `random` untuk menghasilkan bilangan acak dan `argmax_random_tie` kemungkinan digunakan untuk memecahkan tie dalam pemilihan tindakan.
- `city_map`: Sebuah kamus (dictionary) yang merepresentasikan peta kota. Setiap kunci dalam kamus adalah nama kota, dan nilainya adalah kamus lain yang berisi jarak ke kota-kota lain.
- `directed=False`: Menunjukkan bahwa grafik yang dibangun adalah tidak berarah, artinya jarak dari kota A ke B sama dengan jarak dari B ke A.
- `generate_neighbour`: Fungsi ini menghasilkan tetangga (neighbor) dari suatu keadaan (state). Tetangga di sini berarti sebuah rute yang sedikit berbeda dari rute saat ini. Caranya adalah dengan memilih dua kota secara acak dan membalik urutan kota di antara kedua kota tersebut.
- `actions`: Fungsi ini mengembalikan daftar tindakan yang mungkin dilakukan dari suatu keadaan. Dalam konteks TSP, tindakannya adalah menghasilkan tetangga.
- `result`: Fungsi ini menerapkan suatu tindakan terhadap suatu keadaan dan menghasilkan keadaan baru.

- h. path_cost: Fungsi ini menghitung total jarak (biaya) dari suatu rute.
- i. value: Fungsi ini mengembalikan nilai heuristik dari suatu keadaan. Nilai heuristik digunakan untuk memperkirakan kualitas suatu keadaan. Dalam kasus ini, nilai heuristik adalah negatif dari total jarak, sehingga algoritma akan berusaha memaksimalkan nilai heuristik (yaitu meminimalkan total jarak).

B. TSP/Hill Climbing

```


def hill_climbing(problem):
    def find_neighbors(state, number_of_neighbors=100):
        neighbors = []
        for i in range(number_of_neighbors):
            new_state = problem.generate_neighbour(state)
            neighbors.append(Node(new_state))
            state = new_state
        return neighbors

    current = Node(problem.initial)
    while True:
        neighbors = find_neighbors(current.state)
        if not neighbors:
            break
        neighbor = argmax_random_tie(neighbors, key=lambda node:
            problem.value(node.state))
        if problem.value(neighbor.state) <= problem.value(current.state):
            break
        current.state = neighbor.state
    return current.state


if __name__ == '__main__':
    all_cities = []
    cities_graph = city_map.graph_dict
    for city_1 in cities_graph.keys():
        distances[city_1] = {}
        if city_1 not in all_cities:
            all_cities.append(city_1)
        for city_2 in cities_graph.keys():
            if cities_graph.get(city_1).get(city_2) is not None:
                distances[city_1][city_2] = cities_graph.get(city_1).get(city_2)

    tsp_problem = TSP_problem(all_cities)
    result = hill_climbing(tsp_problem)
    print(result)
    cost = tsp_problem.path_cost(result)
    print('cost:', cost)

```

 ['Yogyakarta', 'Cirebon', 'Jakarta', 'Bandung', 'Semarang', 'Surakarta', 'Surabaya', 'Malang']
 cost: 2317

2. Hasil

 ['Yogyakarta', 'Cirebon', 'Jakarta', 'Bandung', 'Semarang', 'Surakarta', 'Surabaya', 'Malang']
 cost: 2317

Berdasarkan hasil tersebut, dapat disimpulkan bahwa program telah menemukan satu kemungkinan rute terpendek untuk mengunjungi semua kota yang disebutkan, dengan total jarak tempuh sejauh 2317 satuan. Artinya, jika ingin mengunjungi semua kota tersebut dengan jarak tempuh seminimal mungkin, maka rute yang disarankan adalah mengikuti urutan kota seperti yang tertera pada hasil.