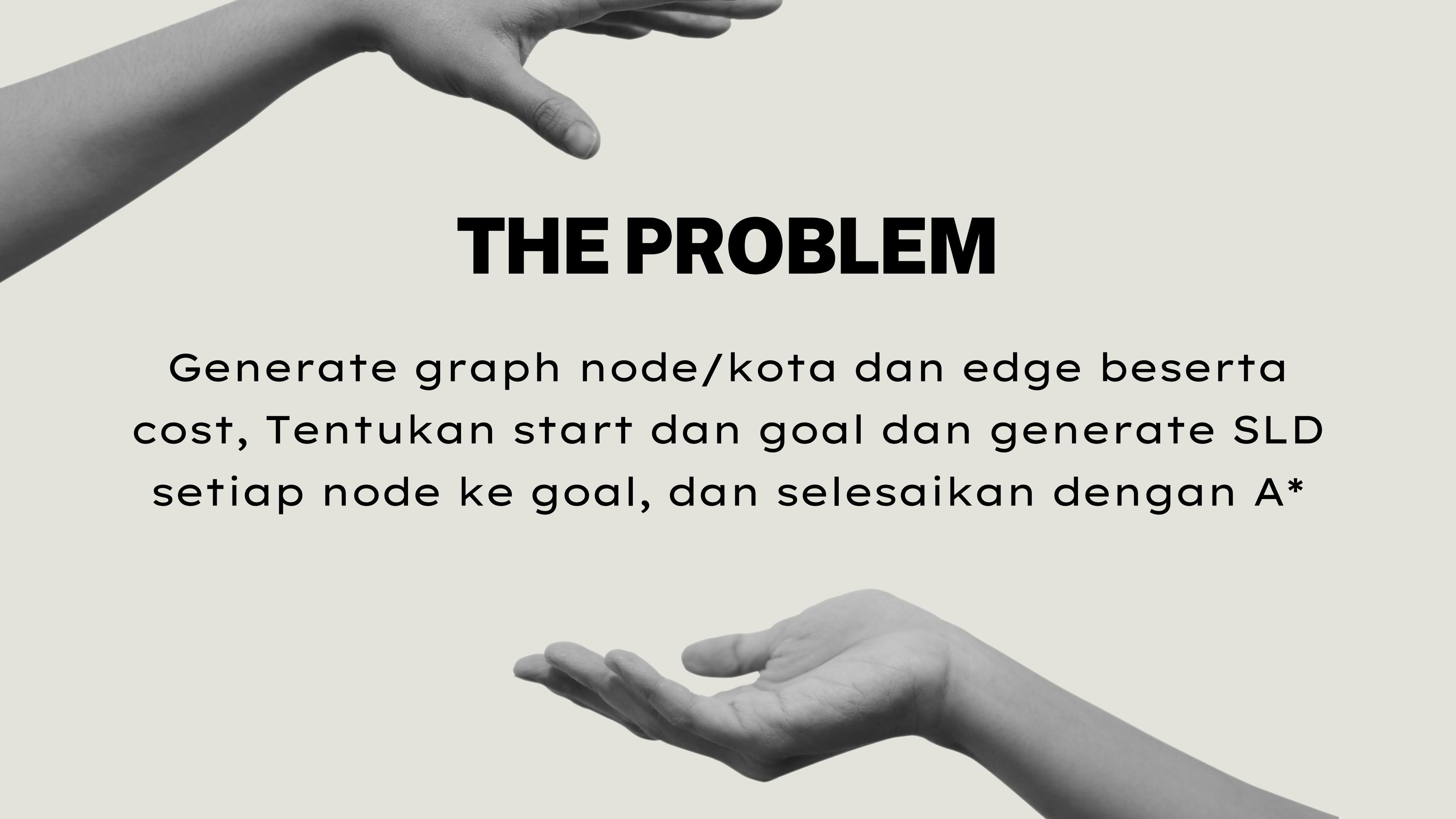




FINDING SHORTEST PATH USING INFORMED SEARCH (A*)



LAURIVASYA GADHING SYAHAFIDH
5025211136
KECERDASAN BUATAN (F)



THE PROBLEM

Generate graph node/kota dan edge beserta cost, Tentukan start dan goal dan generate SLD setiap node ke goal, dan selesaikan dengan A*

BUT FIRST, THE RESEARCH

A* SEARCH DEFINITION

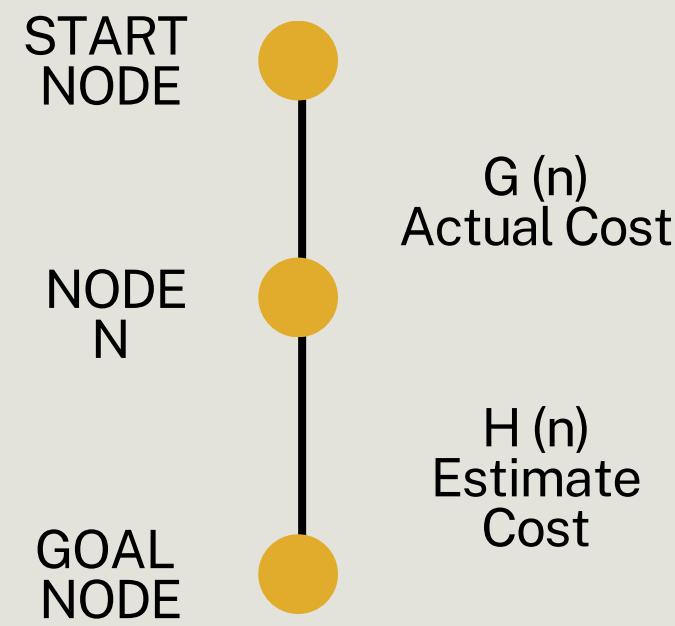
Merupakan salah satu jenis algoritma informed search (pencarian yang dipandu informasi) yang digunakan untuk mencari jalur terpendek antara dua titik pada graf berbobot (weighted graph). A* Search menggabungkan informasi heuristik dan jarak sejauh ini (yang telah dilalui) untuk memilih simpul selanjutnya yang akan dieksplorasi.

Pada setiap iterasi, A* Search mempertimbangkan dua hal untuk memilih simpul selanjutnya untuk dieksplorasi:

1. Cost dari simpul saat ini sampai simpul yang akan dieksplorasi (biasanya dilambangkan dengan g)
2. Estimasi cost dari simpul yang akan dieksplorasi sampai simpul tujuan (biasanya dilambangkan dengan h)

$$F(n) = G(n) + H(n)$$

G(n) = Biaya dari start node ke node yang sedang di-expand.
H(n) = Estimasi biaya terkecil dari node saat ini ke goal node.
F(n) = Estimasi total cost dari start node ke goal node.



Dalam konteks pencarian jarak terpendek dari dua kota, A* Search dapat diaplikasikan dengan menganggap kota-kota sebagai simpul pada graf dan jarak antar kota sebagai bobot pada edge yang menghubungkan simpul tersebut. SLD (Straight Line Distance) dapat digunakan sebagai heuristik dalam menghitung estimasi cost dari simpul yang akan dieksplorasi sampai simpul tujuan.

Dalam implementasi program, A* Search biasanya diwujudkan dalam bentuk fungsi yang menerima graf berbobot, titik awal, titik tujuan, dan fungsi heuristik sebagai parameter input, dan mengembalikan jalur terpendek dan cost yang ditemukan.

IMPLEMENTASI KODE

```
1 import heapq
```

merupakan modul bawaan dari Python yang digunakan untuk implementasi struktur data heap queue (atau biasa disebut priority queue). Dalam implementasi A* search, modul heapq digunakan untuk mengatur urutan node yang akan diekspansi berdasarkan nilai $F(n)$ yang terkecil terlebih dahulu.

```
4 graph = {  
5     'Surabaya': {'Sidoarjo': 23, 'Gresik': 25, 'Mojokerto': 60, 'Kediri': 135},  
6     'Sidoarjo': {'Surabaya': 23, 'Gresik': 18},  
7     'Gresik': {'Surabaya': 25, 'Sidoarjo': 18, 'Mojokerto': 40},  
8     'Mojokerto': {'Surabaya': 60, 'Gresik': 40, 'Kediri': 60},  
9     'Kediri': {'Surabaya': 135, 'Mojokerto': 60}  
10 }
```

mengakukan inisialisasi graph node/kota dan edge beserta costnya. Graph ini merepresentasikan daerah-daerah di Jawa Timur dengan node masing-masing kota dan edge antara kota-kota tersebut yang memiliki nilai cost, yaitu jarak antara kota-kota tersebut.

```
13 start = 'Surabaya'  
14 goal = 'Kediri'
```

menentukan start dan goal yang akan menjadi kota asal dan kota tujuan dalam pencarian jarak terpendek menggunakan algoritma A*.

```
17 sld = {  
18     'Surabaya': 168,  
19     'Sidoarjo': 155,  
20     'Gresik': 126,  
21     'Mojokerto': 55,  
22     'Kediri': 0  
23 }
```

membuat SLD (Straight Line Distance) setiap node ke goal. SLD adalah estimasi jarak terpendek dari suatu node ke goal, dengan asumsi tidak ada rintangan atau penghalang di antara node tersebut dan goal.

```

24
25 def astar(graph, start, goal, sld):
26     # Inisialisasi open set dan closed set
27     openset = []
28     closedset = set()
29
30     # Inisialisasi nilai f, g, dan h untuk node start
31     g = {start: 0}
32     h = {start: sld[start]}
33     f = {start: g[start] + h[start]}
34
35     # Inisialisasi dictionary untuk menyimpan parent dari suatu node
36     came_from = {}
37
38     # Inisialisasi dictionary untuk menyimpan cost path
39     cost = {start: 0}
40
41     # Masukkan start ke open set
42     heapq.heappush(openset, (f[start], start))
43
44     while openset:
45         # Ambil node dengan nilai f terkecil dari open set
46         current_f, current = heapq.heappop(openset)
47
48         # Jika current adalah goal, kembalikan path dan cost
49         if current == goal:
50             path = []
51             while current in came_from:
52                 path.append(current)
53                 current = came_from[current]
54             path.append(start)
55             path.reverse()
56             return path, cost[goal]

```

```

58     # Tandai current sebagai visited dengan memasukkannya ke dalam closed set
59     closedset.add(current)
60
61     # Expand node
62     for neighbor in graph[current]:
63         # Skip jika neighbor sudah visited
64         if neighbor in closedset:
65             continue
66
67         # Hitung nilai g baru
68         tentative_g = g[current] + graph[current][neighbor]
69
70         # Jika neighbor belum ada di open set, atau nilai g baru lebih kecil dari nilai g sebelumnya
71         if neighbor not in g or tentative_g < g[neighbor]:
72             # Update nilai g, h, dan f neighbor
73             g[neighbor] = tentative_g
74             h[neighbor] = sld[neighbor]
75             f[neighbor] = g[neighbor] + h[neighbor]
76
77             # Update cost path
78             cost[neighbor] = tentative_g
79
80             # Simpan parent dari neighbor
81             came_from[neighbor] = current
82
83             # Masukkan neighbor ke open set
84             heapq.heappush(openset, (f[neighbor], neighbor))
85
86     # Jika tidak ada path yang ditemukan, return None
87     return None, None

```

Fungsi astar() menginisialisasi nilai g, h, dan f dari node awal (start) dan memasukkan node tersebut ke dalam open set. Selanjutnya, algoritma akan melakukan looping sampai open set tidak kosong. Pada setiap iterasi, algoritma mengeluarkan node dengan nilai f terkecil dari open set (current) dan mengecek apakah node tersebut merupakan node tujuan. Jika iya, fungsi akan mengembalikan jalur terpendek yang ditemukan.

Jika node current bukan merupakan node tujuan, maka algoritma akan menandai node tersebut sebagai visited (dengan memasukkannya ke dalam closed set) dan akan memeriksa semua tetangga dari node tersebut. Algoritma akan menghitung nilai g baru (tentative_g) dari node tetangga, dimana nilai ini didapatkan dengan menambahkan nilai g dari node saat ini dengan jarak antara node saat ini dan tetangga. Algoritma akan mengecek apakah tetangga sudah pernah dikunjungi (yaitu, sudah termasuk dalam closed set). Jika iya, algoritma akan mengabaikan tetangga tersebut dan melanjutkan ke tetangga lainnya.

Jika tetangga belum pernah dikunjungi atau nilai g baru lebih kecil dari nilai g sebelumnya, algoritma akan melakukan update pada nilai g, h, dan f dari tetangga tersebut. Algoritma juga akan mengupdate cost path dari node awal ke tetangga, serta menyimpan parent dari tetangga dalam suatu dictionary (came_from). Selanjutnya, algoritma akan memasukkan tetangga ke dalam open set.

Setelah melakukan looping pada seluruh node, jika tidak ada jalur yang ditemukan maka fungsi akan mengembalikan nilai None, None.

```
89 # Panggil fungsi astar dan print hasilnya
90 path, cost = astar(graph, start, goal, sld)
91 if path is None:
92     print("Tidak ada path")
93 else:
94     # Print jalur
95     print("Path:", " -> ".join(path))
96
97     # Print biaya
98     print("Cost:", cost)
```

memanggil fungsi **astar()** dengan argumen **graph**, **start**, **goal**, dan **sld**, serta menyimpan hasilnya ke dalam variabel **path** dan **cost**. Kemudian dilakukan pengecekan jika **path** bernilai **None**, maka cetak "Tidak ada path". Jika tidak, cetak jalur yang ditemukan dengan menggunakan **join()** untuk menggabungkan setiap node di dalam **path**, dan cetak juga biaya yang diperoleh.

OUTPUT PROGRAM:

```
Path: Surabaya -> Mojokerto -> Kediri
Cost: 120
```

THANK YOU!