JOBSHEET V
APLIKASI STRATEGI PENCARIAN
(PROBLEM SOLVING PART 2)

1. Tujuan Praktikum

Setelah melakukan materi praktikum ini, mahasiswa diharapkan:

a. Memahami konsep strategi pencarian informed search.
b. Memahami manfaat strategi pencarian informed search.
c. Membangun solusi dari permasalahan yang ada melalui penerapan strategi pencarian informed search.

2. Ringkasan Materi

Informed Search melakukan pencarian dengan menggunakan pengetahuan-spesifik terhadap permasalahan yang dihadapi atau diberikan. Informed Search sering disebut juga dengan Heuristic Search. Heuristik adalah kriteria, metode, atau prinsip untuk memutuskan di antara beberapa tindakan yang dijanjikan menjadi yang paling efektif untuk mencapai beberapa tujuan. Heuristik dapat juga disebut sebagai strategi-strategi perkiraan dalam pengambilan keputusan dan pemecahan masalah yang tidak menjamin solusi yang tepat namun menghasilkan solusi yang masuk akal. Informed Search pada umumnya menggunakan pendekatan Best-First Search, algoritma TREE-SEARCH, diantaranya adalah:

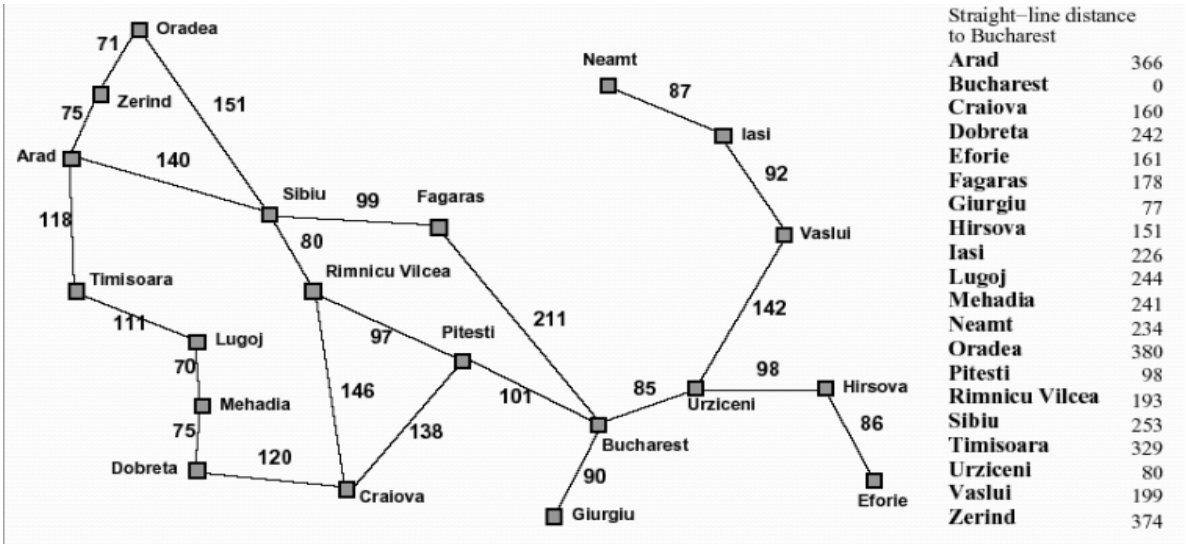
- A* Search.
- Greedy Search.

2.1. A* (A Star) Search

A* didasarkan pada penggunaan metode heuristik untuk mencapai optimalitas, dan merupakan varian dari algoritma best-first. Ketika algoritma pencarian memiliki sifat optimalitas, itu berarti dijamin untuk menemukan solusi terbaik pada permasalahan shortest path sampai pada goal state (G). Setiap kali A* memasuki state, hal tesebut juga melalui proses menghitung biaya, f(n), dimana n adalah simpul tetangga, untuk melakukan perjalanan ke semua simpul tetangga, dan kemudian memasuki simpul dengan nilai terendah dari f(n). Nilai-nilai tersebut dihitung dengan rumus berikut:

$$f(n) = g(n) + h(n)$$

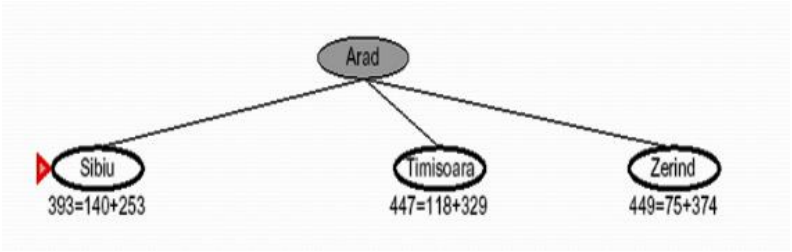
g (n) merupakan nilai jalur terpendek dari simpul awal ke simpul n, dan h (n) menjadi perkiraan heuristik dari nilai simpul. Agar dapat merekonstruksi setiap solusi, maka diperlukan tanda setiap simpul dengan relatif yang memiliki nilai f(n) yang optimal. Ini juga berarti bahwa jika agen mengunjungi kembali simpul tertentu, agen harus memperbarui ketetanggan simpul tersebut yang paling optimal juga. Efisiensi A * sangat tergantung pada nilai heuristik h (n), dan tergantung pada jenis masalah, agen mungkin perlu menggunakan fungsi heuristik yang berbeda untuk menemukan solusi optimal. Berikut ini adalah contoh penerapan algoritma A* yakni memperoleh jalan termurah (cheapest path) dari kota Arad ke kota Bucharest.



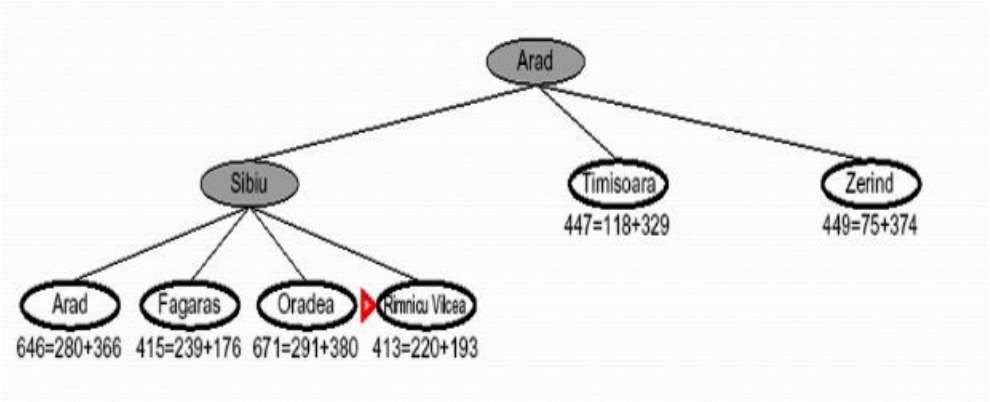
Gambar 1. Lintasan untuk Studi Kasus Metode A*

Gambar 1 memperlihatkan beberapa lintasan yang dapat dilewati oleh agen untuk berjalan dari kota Arad menuju kota Bucharest yang merupakan kota-kota di Romania. Pada Gambar 2 sampai dengan Gambar 7 ini diperlihatkan mekanisme pembukaan simpul oleh agen berdasarkan nilai f(n) yang paling murah (*cheapest cost*).

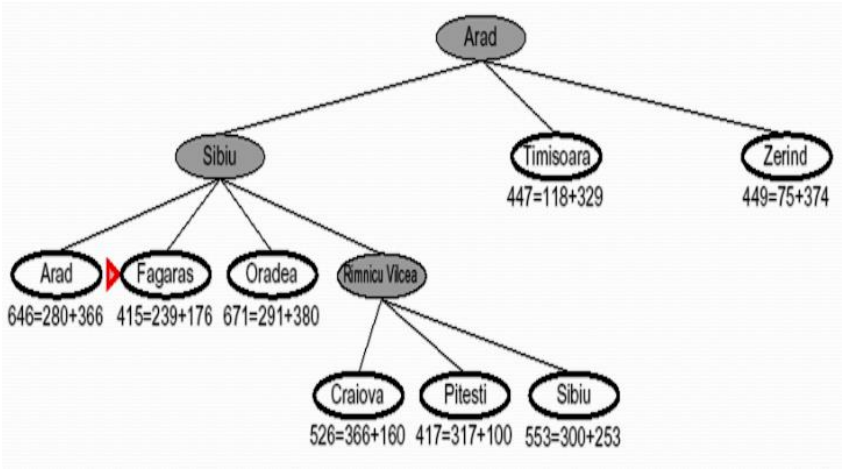
Gambar 2. Solusi tahap 1 dengan Metode A*.



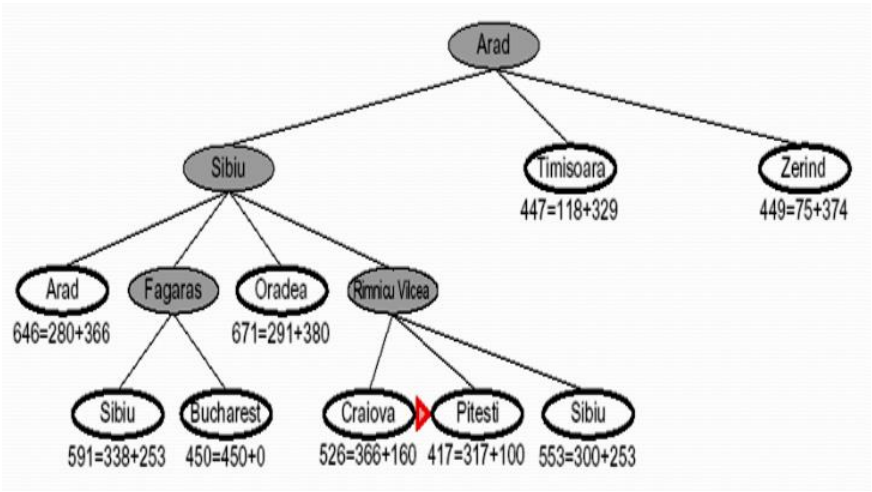
Gambar 3. Solusi tahap 2 dengan Metode A*.



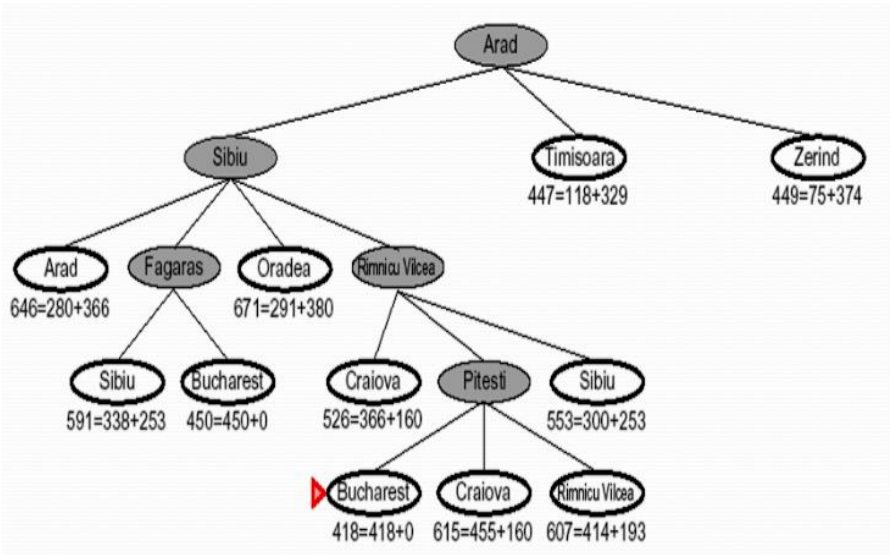
Gambar 4. Solusi tahap 3 dengan Metode A*.



Gambar 5. Solusi tahap 4 dengan Metode A*.



Gambar 6. Solusi tahap 5 dengan Metode A*.



Gambar 7. Solusi tahap 6 dengan Metode A*.

2.1.1. Praktikum. Percobaan 1: A* Search

a. Buatlah sebuah class graph, yang didalamnya terdiri atas beberapa fungsi atau prosedur. Mulai dari prosedur init untuk deklarasi adjacent list mulai dari node start, prosedur get_neighbors untuk adjacent list dengan node-node tetangaa menuju goal, fungsi heuristic yang memberikan nilai sama untuk semua node yang ada, dan fungsi-fungsi yang akan di jelaskan pada tahap selanjutnya. Untuk kode program awal tahap ini dengan Python adalah sebagai berikut:

```
class Graph:

    def __init__(self, adjacency_list):
        self.adjacency_list = adjacency_list

    def get_neighbors(self, v):
        return self.adjacency_list[v]

    # heuristic function with equal values for all nodes
    def h(self, n):
        H = {
            'A': 1,
            'B': 1,
            'C': 1,
            'D': 1
        }

        return H[n]
```

b. Masih dalam class yang sama, buatlah fungsi algoritma A* sebagai berikut :

```
def a_star_algorithm(self, start_node, stop_node):
    # open_list is a list of nodes which have been visited, but who's neighbors
    # haven't all been inspected, starts off with the start node
    # closed_list is a list of nodes which have been visited
    # and who's neighbors have been inspected
    open_list = set([start_node])
    closed_list = set([])

    # g contains current distances from start_node to all other nodes
    # the default value (if it's not found in the map) is +infinity
    g = {}

    g[start_node] = 0

    # parents contains an adjacency map of all nodes
    parents = {}
    parents[start_node] = start_node
```

```
     while len(open_list) > 0:
         n = None

         # find a node with the lowest value of f() - evaluation function
         for v in open_list:
             if n == None or g[v] + self.h(v) < g[n] + self.h(n):
                 n = v;

         if n == None:
             print('Path does not exist!')
             return None

         # if the current node is the stop_node
         # then we begin reconstructin the path from it to the start_node
         if n == stop_node:
             reconst_path = []

             while parents[n] != n:
                 reconst_path.append(n)
                 n = parents[n]

             reconst_path.append(start_node)

             reconst_path.reverse()

             print('Path found: {}'.format(reconst_path))
             return reconst_path
```

```
        # for all neighbors of the current node do
        for (m, weight) in self.get_neighbors(n):
            # if the current node isn't in both open_list and closed_list
            # add it to open_list and note n as it's parent
            if m not in open_list and m not in closed_list:
                open_list.add(m)
                parents[m] = n
                g[m] = g[n] + weight

            # otherwise, check if it's quicker to first visit n, then m
            # and if it is, update parent data and g data
            # and if the node was in the closed_list, move it to open_list
            else:
                if g[m] > g[n] + weight:
                    g[m] = g[n] + weight
                    parents[m] = n

                    if m in closed_list:
                        closed_list.remove(m)
                        open_list.add(m)

        # remove n from the open_list, and add it to closed_list
        # because all of his neighbors were inspected
        open_list.remove(n)
        closed_list.add(n)

    print('Path does not exist!')
    return None
```

c. Tahap selanjutnya adalah buatlah adjacency list, kemudian cari solusinya dengan algoritma A* dengan code program sebagai berikut.

```
adjacency_list = {
    'A': [('B', 1), ('C', 3), ('D', 7)],
    'B': [('D', 5)],
    'C': [('D', 12)]
}
graph1 = Graph(adjacency_list)
graph1.a_star_algorithm('A', 'D')
```

2.1.2. Pertanyaan

a. Amati output pada percobaan 1, dan jelaskan bagaimana hasilnya?
b. Jelaskan tahapan-tahapan pada fungsi algoritma A*di percobaan 1 langkah ke-2 yang sudah dijelaskan di atas.
c. Apakah tujuan pembuatan fungsi heuristik?
d. Bagaimanakah kompleksitas waktu pada algoritma A*? Jelaskan dan beri contoh!
e. Jelaskan maksud dari code program di bawah ini:

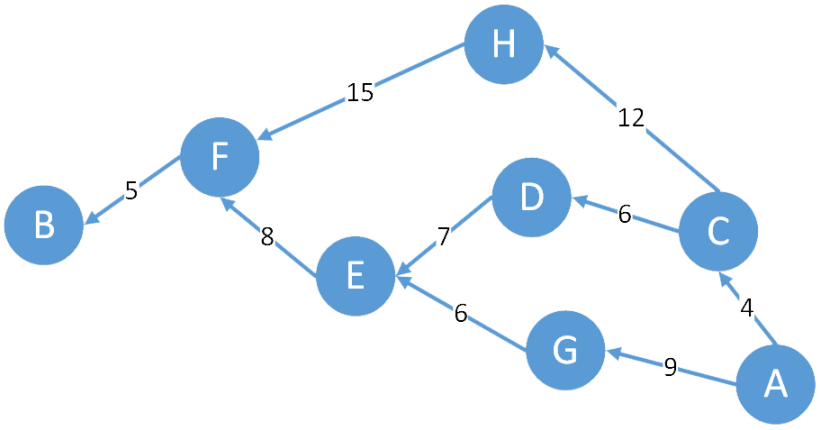
```
adjacency_list = {
    'A': [('B', 1), ('C', 3), ('D', 7)],
    'B': [('D', 5)],
    'C': [('D', 12)]
}
```

2.1.3. Tugas

a. Ubahlah kode program pada Percobaan 2 menggunakan metode pencarian jarak yang lain, selain Euclidean distance!
b. Jelaskan perbedaan hasilnya pada tugas No. 1!
c. Dapatkah algoritma A* diterapkan untuk graph yang tidak berbobot? Jika tidak, apakah alasanya?
d. Carilah kegunaan algoritma A* yang diimplementasikan pada game. Jelaskan tahapanya dan juga tampilkan game atau langkah-langkah pembuatanya!

2.2. Greedy Search

Algoritma greedy merupakan jenis algoritma best-first search yang menggunakan pendekatan penyelesaian masalah dengan mencari nilai maksimum sementara pada setiap langkahnya. Nilai maksimum sementara ini dikenal dengan istilah *local maximum*. Pada kebanyakan kasus, algoritma greedy mampu memberikan solusi dalam waktu yang cukup cepat namum tidak akan optimal. Sebagai contoh dari penyelesaian masalah dengan algoritma greedy, perhatikan sebuah masalah klasik yang sering dijumpai dalam kehidupan sehari-hari: mencari jarak terpendek dari peta. Kemudian peta tersebut direpresentasikan dengan Directed Graph (graph berarah) seperti pada Gambar 8.



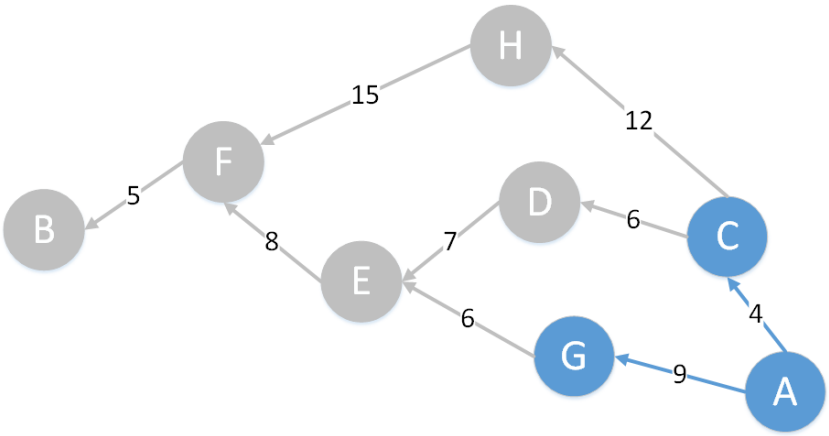
Gambar 8. Graph Berarah dari Titik A ke B

Untuk mencari jarak terpendek dari A ke B, algoritma greedy akan menjalankan langkah-langkah seperti berikut:

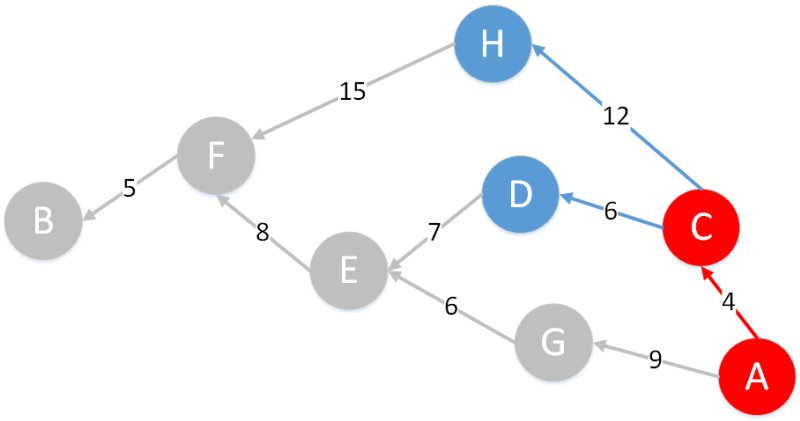
a. Kunjungi satu simpul pada graph, dan ambil seluruh simpul yang dapat dikunjungi dari simpul sekarang.
b. Cari *local maximum* ke simpul selanjutnya.
c. Tandai graph sekarang sebagai graph yang telah dikunjungi, dan pindah ke *local maximum* yang telah ditentukan.
d. Kembali ke langkah 1 sampai simpul tujuan didapatkan.

Jika mengaplikasikan langkah-langkah di atas pada graph A ke B sebelumnya maka kita akan mendapatkan pergerakan seperti berikut:

a. Mulai dari simpul awal (A). Ambil seluruh simpul yang dapat dikunjungi.
b. Local maximum adalah ke simpul C, karena jarak ke simpul C adalah yang paling dekat.
c. Tandai simpul A sebagai titik yang telah dikunjungi, dan pindah ke simpul C.
d. Ambil seluruh simpul yang dapat dikunjungi dari C.



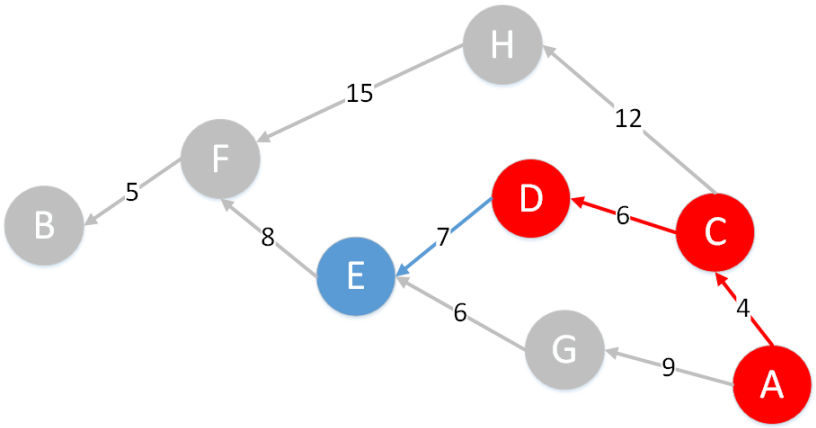
Gambar 9. Langkah Pertama Algoritma Greedy.



Gambar 10. Langkah Kedua Algoritma Greedy.

e. Local maximum adalah ke simpul D, dengan jarak 6.

f. Tandai simpul C sebagai simpul yang telah dikunjungi, dan pindah ke simpul D.



Gambar 11. Langkah Ketiga Algoritma Greedy.

g. Langkah selanjutnya dilakukan sampai diperoleh jarak terpendek.

Dengan menggunakan algoritma greedy pada graph di atas, hasil akhir yang akan didapatkan sebagai jarak terpendek adalah A-C-D-E-F-B. Hasil jarak terpendek yang didapatkan ini tidak tepat dengan jarak terpendek yang sebenarnya (A-G-E-F-B). Algoritma greedy memang tidak selamanya memberikan solusi yang optimal, dikarenakan pencarian *local maximum* pada setiap langkahnya, tanpa memperhatikan solusi secara keseluruhan. Untuk kasus umum, kerap kali algoritma greedy memberikan hasil yang *cukup baik* dengan kompleksitas waktu yang cepat. Hal ini mengakibatkan algoritma greedy sering digunakan untuk menyelesaikan permasalahan kompleks yang memerlukan kecepatan jawaban, bukan solusi optimal, misalnya pada game.

Algoritma greedy merupakan algoritma yang besifat heuristik, mencari nilai maksimal sementara dengan harapan akan mendapatkan solusi yang cukup baik. Meskipun tidak selalu mendapatkan solusi terbaik (optimum), algoritma greedy umumnya memiliki kompleksitas waktu yang cukup baik, sehingga algoritma ini sering digunakan untuk kasus yang memerlukan solusi cepat meskipun tidak optimal seperti sistem real-time atau game. Algoritma greedy memiliki beberapa fungsionalitas dasar, yaitu:

a. Fungsi untuk melakukan penelusuran masalah.
b. Fungsi untuk memilih *local maximum* dari pilihan-pilihan yang ada tiap langkahnya.
c. Fungsi untuk mengisikan nilai *local maximum* ke solusi keseluruhan.
d. Fungsi yang menentukan apakah solusi telah didapatkan.

2.2.1. Praktikum. Percobaan 2: Implementasi Algoritma Greedy

a. Untuk memperdalam pengertian algoritma greedy, pada praktikum mahasiswa akan mengimplementasikan algoritma yang telah dijelaskan pada bagian sebelumnya ke dalam kode program. Contoh kode program akan diberikan dalam bahasa pemrograman python. Sebagai langkah awal, terlebih dahulu harus merepresentasikan graph. Pada implementasi ini, graph direpresentasikan dengan menggunakan dictionary di dalam dictionary, seperti berikut:

```
DAG = {'A': {'C': 4, 'G': 9},
       'G': {'E': 6},
       'C': {'D': 6, 'H': 12},
       'D': {'E': 7},
       'H': {'F': 15},
       'E': {'F': 8},
       'F': {'B': 5}}
```

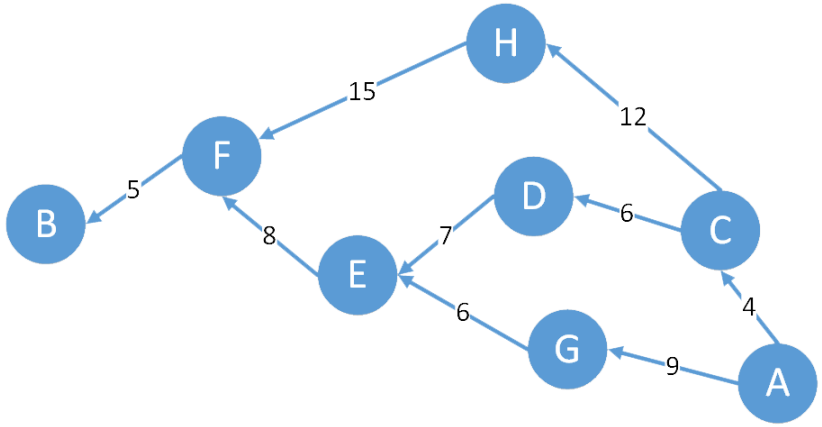
b. Selanjutnya kita akan membuat fungsi algoritma Greedy sebagai berikut:

```
def shortest_path(graph, source, dest):
    result = []
    result.append(source)

    while dest not in result:
        current_node = result[-1]
```

```
    local_max = min(graph[current_node].values())
    for node, weight in graph[current_node].items():
        if weight == local_max:
            result.append(node)
return result
```

c. Data heuristik (straight-line distance) dari masing-masing simpul ke goal state, B sebagai berikut:



Gambar 12. Lintasan atau jalan (path) dari start state, A ke goal state, B.

Straight-Line Distance ke B	
A	24
C	26
D	19
E	11
F	3
G	16
H	17

2.2.2. Pertanyaan

a. Amati output pada percobaan 1, dan Jelaskan bagaimana hasilnya? Jangan lupa tambahkan line-code untuk menampilkan jalur pencarian dengan A sebagai titik awal dan F sebagai tujuan.
b. Jelaskan tahapan-tahapan pada fungsi algoritma Greedy di percobaan 1 langkah ke 2 yang sudah dijelaskan di atas.
c. Bagaimanakah kompleksitas waktu pada algoritma Greedy? Jelaskan dan beri contoh!
d. Tampilkan urutan simpul dari lintasan yang dipilih oleh algoritma greedy search dan hitung path cost-nya.

3. Problem. Menyelesaikan Knapsack Problem dengan Algoritma Greedy dan A*.

3.1. Knapsack Problem

- Knapsack dapat diartikan sebagai karung atau kantung yang digunakan untuk memuat sesuatu. Tidak semua objek dapat ditampung di dalam karung. Karung tersebut hanya dapat menyimpan beberapa objek dengan total ukurannya (weight) lebih kecil atau sama dengan ukuran kapasitas karung.
- Knapsack 0/1, yaitu suatu objek diambil seluruh bagiannya atau tidak sama sekali. Setiap objek mempunyai nilai keuntungan atau yang disebut dengan profit.
- Tujuannya adalah mendapatkan profit maksimal menggunakan banyak objek yang masuk akan menguntungkan, namun hal yang sebaliknya dapat saja terjadi.
    - Cara terbaik agar menguntungkan adalah bukan hanya dari hasil yang optimal tetapi juga jumlah langkah yang dibutuhkan.

Knapsack 0/1

Diberikan n buah objek dan sebuah knapsack dengan kapasitas bobot W. Setiap objek memiliki properti bobot (weigth) wi dan keuntungan (profit) *pi*. Permasalahnnya adalah memilih objek-objek yang dimasukkan ke dalam knapsack sedemikian sehingga dapat memaksimumkan keuntungan. Total bobot objek yang dimasukkan kedalam knapsack tidak boleh melebihi kapasitas knapsack.

Solusi persoalan dinyatakan sebagai vektor n-tupel:

X = {x1, x2, …, xn}

xi = 1 jika objek ke-i dimasukkan ke dalam knapsack,

xi = 0 jika objek ke-i tidak dimasukkan.

Persoalan 0/1 Knapsack dapat dipandang sebagai mencari himpunan bagian (subset) dari keseluruhan objek yang muat ke dalam knapsack dan memberikan total keuntungan terbesar.

3.2. Penyelesaian dengan Greedy

3.2.1. Greedy by Profit

Pada setiap langkah Knapsack diisi dengan obyek yang mempunyai keuntungan terbesar. Strategi ini mencoba memaksimumkan keuntungan dengan memilih objek yang paling menguntungkan terlebih dahulu. Hal pertama yang didilakukan adalah mengurutkan secara menurun obyek-obyek berdasarkan profitnya. Kemudian obyek-obyek yang dapat ditampung oleh knapsack diambil satu persatu sampai knapsack penuh atau (sudah tidak ada obyek lagi yang bisa dimasukan). Data awal adalah sebagai berikut:

w1 = 6; p1 = 12

w2 = 5; p2 = 15

w3 = 10; p3 = 50

w4 = 5; p4 = 10

Kapasitas knapsack W = 16

3.2.2. Greedy by Weight

Pada setiap langkah, knapsack diisi dengan objek yang mempunyai berat paling ringan. Strategi ini mencoba memaksimumkan keuntungan dengan memasukan sebanyak mungkin objek kedalam knapsack. Hal pertama yang harus dilakukan adalah mengurutkan secara menaik objek-objek berdasarkan weight-nya. Kemudian obyek-obyek yang dapat ditampung oleh knapsack diambil satu persatu sampai knapsack penuh atau (sudah tidak ada obyek lagi yang bisa dimasukan).

3.2.3. Greedy By Density

Pada setiap langkah, knapsack diisi dengan obyek yang mempunyai densitas terbesar (perbandingan nilai dan berat terbesar). Strategi ini mencoba memaksimumkan keuntungan dengan memilih objek yang mempunyai keuntungan per unit berat terbesar. Hal pertama yang harus dilakukan adalah mencari nilai profit per unit/density dari tiap-tiap objek. Kemudian obyek-obyek diurutkan berdasarkan densitasnya. Kemudian obyek-obyek yang dapat ditampung oleh knapsack diambil satu persatu sampai knapsack penuh atau (sudah tidak ada obyek lagi yang bisa dimasukan).

Contoh:

Kapasitas M=20, dengan jumlah barang =3

Berat Wi masing-masing barang (W1,W2,W3) sebesar (18,15,10)

Nilai Profit masing-masing barang (P1,P2,P3) sebesar (25,24,15)

Pilih Barang dengan nilai profit maksimal

P1=25 –> x1=1. batas atas nilai

P2=24 –> x2=2/15.

P3=15 –> x3 =0. batas bawah nilai.

Pilih barang dengan berat minimal

W1 = 18 –> x1=0. batas bawah

W2=15 –> x2 = 2/3

W3=10 –> x3=1. batas atas.

Pilih barang dengan menghitung perbandingan yang terbesar dari profit dibagi Berat (Pi/Wi) diurut secara tidak naik.

P1/w1=25/18 (1.38) –> x1=0. karena terkecil x1=0

P2/w2=24/ 15 (1.6) –> x2=1. karena terbesar x2=1

P3/w3=15/10 (1.5) –> x3=1/2 dicari dengan fungsi pembatas x3=1/2.

Tabel 3.1 Rangkuman Greedy by Density

Solusi	(x1,x2,x3)	Σwixi	Σpixi
Pi max	1,2/15,0	20	28.2
Wi min	0,2/3,1	20	31.0
Pi/Wi max	0,1,1/2	20	31.5

3.3. Praktikum. Menyelesaikan Knapsack Problem dengan Algoritma Greedy

Untuk soal knapsack diasumsikan bahwa ada 3 parameter dimana algoritma greedy akan mencari dan menentukan berdasarkan apa nilai terbaik akan diambil di setiap langkahnya. Parameter tersebut meliputi : bobot, profit, dan profit/bobot. Yang bisa dilihat pada list berikut:

```
item = [[3,4],[4,5],[1,2],[7,5],[6,5],[8,8],[9,11]]
```

Pada list item berbentuk 7 x 2. Yang artinya ada 7 baris dan 2 kolom. Baris melambangkan banyaknya nilai. Kolom ke-1 melambangkan bobot, kolom ke-2 melambangkan profit. Kita akan mencari profit optimum berdasarkan algoritma greedy tadi dengan Kapasitas sistemnya adalah 20. Tabel-tabel berikut ini memperlihatkan hasil-hasil komputasi manual algoritma greedy.

i	Bobot	Keuntungan	keuntungan/bobot
1	3	4	1,33
2	4	5	1,25
3	1	2	2,00
4	7	5	0,71
5	6	5	0,83
6	8	8	1,00
7	9	11	1,22

Berdasarkan bobot			
i	Bobot	keuntungan	Pi
7	9	11	1,22
6	8	8	0,83
1	3	4	1,33
	20	23	

Berdasarkan Pi			
i	bobot	keuntungan	pi
3	1	2	2
1	3	4	1,33
2	4	5	1,25
7	9	11	1,22
	17	22	

a. Dengan menggunakan python, kita dapat menyelesaikan masalah knapsack diatas. Buatlah sebuah coding list python, yang di dalam nya terdiri dari penginisialisasian item serta fungsi atau prosedur knapsack. Untuk code program awal tahap ini dengan Python, sebagai berikut:

```
from operator import itemgetter, attrgetter
w = [3,4,1,7,6,8,9]
p = [4,5,2,5,5,8,11]
item = [[3,4],[4,5],[1,2],[7,5],[6,5],[8,8],[9,11]]
```

b. Setelah itu buatlah list coding yang berisi fungsi knapsack sebagai berikut:

```
i=0
while i<len(item):
    hasil = item[i][1]/item[i][0]
    item[i].append(hasil)
    i += 1

data = sorted(item,key=itemgetter(2), reverse = True)

def knapsack(data,cap,flag):
    total=0
    tres = ""
    if(flag==0):
        dataS = sorted(data,key=itemgetter(flag), reverse = True)
        tres="bobot prioritas : "
    elif(flag==1):
        dataS = sorted(data,key=itemgetter(flag), reverse = True)
        tres="keuntungan prioritas : "
    elif(flag ==2):
        dataS = sorted(data,key=itemgetter(flag), reverse = True)
        tres="p prioritas : "
    else:
        return "Error"

    j=0
    hasil=0
    #print("sini")
    cek=0
    weight=0
    while(j<len(dataS)):

        if(cek+dataS[j][0]<=cap):
            hasil=hasil+dataS[j][1]
            weight=weight+dataS[j][0]
            print(dataS[j][0])
        cek=weight
        j+=1;
        #print("here")
    return("Optimal dalam "+str(tres)+str(hasil))

print(knapsack(item,20,0))
print(knapsack(item,20,1))
print(knapsack(item,20,2))
```

3.3.1. Pertanyaan

a. Amati output pada percobaan 2, dan Jelaskan bagaimana hasilnya?
b. Jelaskan tahapan-tahapan pada fungsi algoritma Greedy di percobaan 2 langkah ke 2 yang sudah dijelaskan di atas.