BAB II
TEORI PENUNJANG

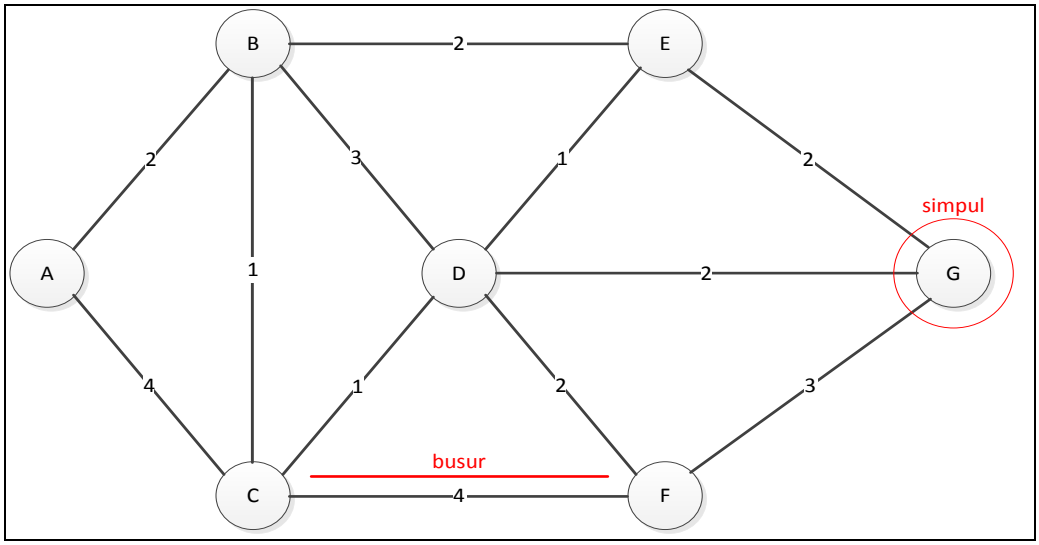
2.1 Teori *Graph* (Graf)

Graph (graf) bisa dibayangkan sebagai kumpulan obyek atau aktifitas. Sebagai contoh, rute bis kota dari satu terminal ke terminal lain, rute perjalanan seseorang dari satu tempat ke tempat lain, dan contoh-contoh lain yang bisa dijadikan sebagai suatu *graph*. Contoh di atas merupakan contoh klasik dari teori *graph* yang lebih dikenal dengan persoalan *travelling salesman problem* atau *shortest path problem*, yang pada prinsipnya mencari lintasan terpendek dari semua tempat yang harus dikunjungi, sehingga dapat menghemat waktu, tenaga, maupun biaya. [8]

Graf adalah kumpulan simpul (*nodes*) yang dihubungkan satu sama lain melalui sisi/busur (*edges*). *Graph* (graf) dapat dikatakan pasangan himpunan (V,E) yang dinotasikan G = (V,E), dimana:

a. Verteks (simpul) : V = himpunan simpul.
b. *Edge* (sisi/ruas) : E = himpunan busur yang menghubungkan simpul.

Simpul-simpul pada graf dapat merupakan objek sembarang seperti kota, atom-atom suatu zat, nama anak, jenis buah, komponen alat elektronik dan sebagainya. Busur dapat menunjukkan hubungan (relasi) sembarang seperti rute penerbangan, jalan raya, sambungan telepon, ikatan kimia, dan lain-lain.



Gambar 2.1 Graf Berbobot dan Tidak Berarah

Gambar 2.1 di atas menunjukkan sebuah graf berbobot dan tidak berarah, dimana:

- A, B, C, D, E, F, dan G adalah simpul (verteks).
- 1, 2, 3, 4 adalah bobot busur yang menghubungkan dua simpul.

2.2 Lintasan Terpendek

Lintasan terpendek adalah lintasan yang memiliki jarak terpendek yang diperlukan untuk mencapai suatu tempat dari tempat tertentu. Lintasan minimum yang dimaksud dapat dicari dengan menggunakan graf. Graf yang digunakan adalah graf yang berbobot, yaitu graf yang memiliki suatu nilai atau bobot.

Ada beberapa macam persoalan lintasan terpendek, antara lain:

a. Lintasan terpendek antara dua buah simpul tertentu (*a pair shortets path*),
b. Lintasan terpendek antara semua pasangan simpul (*all pairs shortest path*),
c. Lintasan terpendek dari simpul tertentu ke semua simpul yang lain (*single source shortest path*), dan
d. Lintasan terpendek antara dua buah simpul yang melalui beberapa simpul tertentu (*intermediate shortest path*).

2.3 Algoritma A* *Search*

Terdapat banyak metode dalam teknik *searching* yang telah dipublikasikan oleh para peneliti. Setiap metode memiliki kelebihan dan kekurangannya masing-masing. Metode *searching* yang paling populer untuk pencarian lintasan terpendek adalah algoritma Dijkstra. Algoritma ini mudah diimplementasikan, tetapi kinerjanya kurang bagus untuk graf yang besar dengan puluhan ribu simpul. Algoritma lain yang juga sangat populer adalah algoritma A*.

2.3.1 Deskripsi Algoritma A* *Search*

A* (dibaca "A bintang" atau "A *star*") adalah algoritma pencarian graf/pohon yang mencari jalur dari satu titik awal ke sebuah titik akhir yang telah ditentukan. Algoritma A* menggunakan pendekatan heuristik h(x) yang memberikan peringkat ke tiap-tiap titik x dengan cara memperkirakan rute terbaik

yang dapat dilalui dari titik tersebut. Setelah itu tiap-tiap titik x tersebut dicek satu-persatu berdasarkan urutan yang dibuat dengan pendekatan heuristik tersebut.

Algoritma A* adalah contoh dari *best-first search*. Algoritma ini pertama kali ditemukan pada tahun 1968 oleh Peter Hart, Nils Nilsson, dan Bertram Raphael. Algoritma A* merupakan format pencarian heuristik untuk menghitung efisiensi solusi optimal. [7]

Sebenarnya, *Depth-first Search* (DFS) dan *Breadth-first search* (BFS) adalah dua kasus khusus dari algoritma A*. Algoritma Dijkstra, salah satu BFS, adalah kasus khusus dari A* dimana h(x) = 0 untuk semua nilai x. Untuk DFS, ciptakan suatu *counter* global C yang diinisialisasi dengan nilai yang sangat besar. Pada setiap langkahnya, periksa sebuah titik, lalu berikan nilai C ke semua titik yang bertetangga dengan titik tadi. Setelah tiap-tiap pemberian nilai, kurangi *counter* C dengan 1. Jadi semakin awal sebuah titik diproses, semakin tinggi nilai h(x) yang dimilikinya.

Algoritma A* menyimpan sebuah himpunan solusi parsial, jalur yang diambil yang berawal dari titik awal, disimpan dalam sebuah antrian berprioritas (*priority queue*). Prioritas yang diberikan pada suatu jalur ditentukan oleh fungsi:

$$f(x) = g(x) + h(x) \quad \textbf{(2.1)}$$

Di sini, g(x) adalah ongkos yang diperlukan sampai ke titik x sejauh ini, sedangkan h(x) adalah perkiraan heuristik dari ongkos minimal untuk meraih titik tujuan dari titik x. Singkat kata, ongkos adalah jarak yang telah ditempuh sejauh ini, dan panjang garis lurus antara titik x dengan titik akhir adalah perkiraan heuristiknya. Semakin rendah nilai f(x), semakin tinggi prioritasnya. [13]

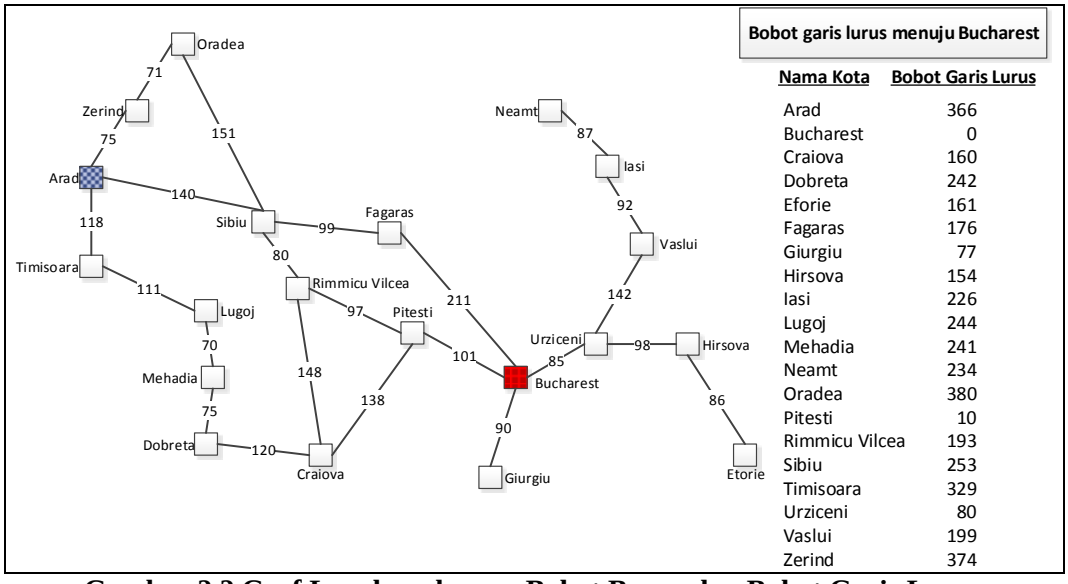
Berikut diberikan *psedo-code* dari algoritma A*:

```
fungsi A* (awal,tujuan)
    var himpunan_tertutup <- himpunan_kosong
    var q <- buat_antrian(titik_awal)
    while (q tidak kosong)
        var p <- hapus_elemen_pertama(q)
        var x <- titik_akhir_dari_p
        if (x = tujuan)
            return p
        for (tiap y pada suksesor(p))
            if (y bukan anggota himpunan_tertutup)
                masuk_antrian(q,y)
                tambahkan y ke himpunan_tertutup
            endif
        endfor
    endwhile
    return tidak_ada_jalur
```

Disini, suksesor (p) mengembalikan sebuah himpunan titik yang bertetangga dengan p. Diasumsikan bahwa antrian disini menjaga urutan anggotanya dengan nilai f secara otomatis. [12]

Dalam himpunan-himpunan tertutup, semua titik akhir dari p (titik dimana masih terdapat jalur) disimpan, untuk menghindari pengulangan dan pemutaran. Antrian yang digunakan disini juga biasa disebut sebagai himpunan terbuka. Himpunan tertutup bisa saja dihilangkan (yang akan menghasilkan algoritma pencarian pada pohon) apabila sebuah solusi (jalur) telah dijamin keberadaannya, atau apabila fungsi suksesor telah disesuaikan agar tidak menerima titik yang sudah pernah diperiksa. [12]

Berikut contoh pencarian lintasan terpendek menggunakan algoritma A*.

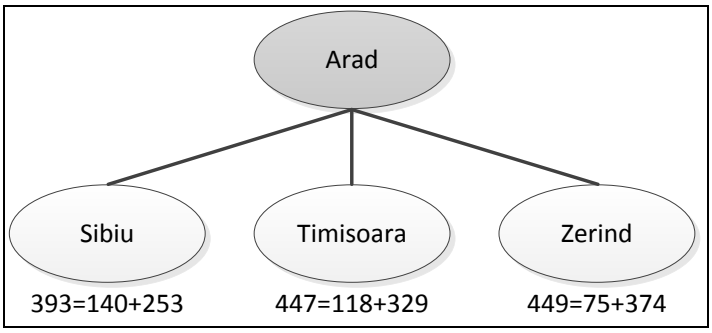


Bobot garis lurus menuju Bucharest

Nama Kota	Bobot Garis Lurus
Arad	366
Bucharest	0
Craiova	160
Dobreta	242
Eforie	161
Fagaras	176
Giurgiu	77
Hirsova	154
Iasi	226
Lugoj	244
Mehadia	241
Neamt	234
Oradea	380
Pitesti	10
Rimmicu Vilcea	193
Sibiu	253
Timisoara	329
Urziceni	80
Vaslui	199
Zerind	374

Gambar 2.2 Graf Lengkap dengan Bobot Busur dan Bobot Garis Lurus

Dari gambar di atas, diambil contoh Kota Arad adalah kota asal (titik awal) dan Bucharest adalah kota tujuan. Berikut cara penarikan graf nama-nama kota lengkap dengan bobot jarak antar kota dan juga bobot garis lurus dari setiap kota menuju kota tujuan (Bucharest).



Gambar 2.3 Pencarian *Node* yang Akan Dilalui Pertama

Kota asal yaitu Arad, kota-kota terdekat yang terhubung dengan kota Arad yaitu Sibiu, Timisora, dan Zerind. Berdasarkan fungsi algoritma A* untuk mendapatkan bobot terkecil menuju tujuan yaitu dengan menjumlahkan bobot kota Arad menuju kota terdekat [g(n)] dengan bobot garis lurus dari kota 'n' menuju kota Bucharest [h(n)]. Dari pencarian *node* yang akan dilalui pertama didapatkan bobot dari ketiga kota, yaitu:

- Sibiu

  Bobot Kota Arad sampai Sibiu = 140
  Bobot Garis Lurus Sibiu sampai Bucharest = 253
  Total Bobot = 393

- Timisora

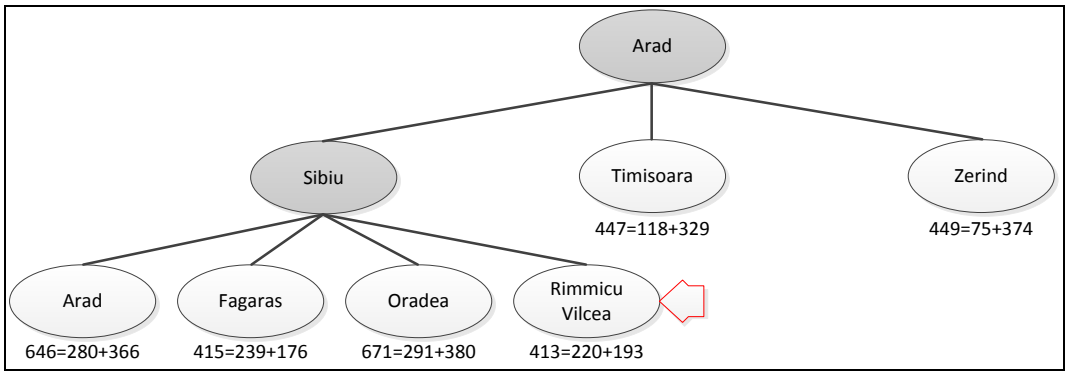
  Bobot Kota Arad sampai Timisora = 118
  Bobot Garis Lurus Timisora sampai Bucharest = 329
  Total Bobot = 447

- Zerind

  Bobot Kota Arad sampai Zerind = 75
  Bobot Garis Lurus Zerind sampai Bucharest = 374
  Total Bobot = 449

Berdasarkan pencarian lintasan terpendek dari ketiga kota yang terhubung dengan kota asal didapatkan bobot terkecil menuju Bucharest yaitu melalui kota Sibiu. Maka jalur yang pertama dilalui yaitu jalur menuju kota Sibiu.

Langkah-langkah pencarian berikutnya sama dengan langkah pencarian *node* yang akan dilalui pertama.



Gambar 2.4 Pencarian *Node* yang Akan Dilalui Kedua

Selanjutnya lakukan pencarian *node* kedua yang akan dilalui. Bobot dari masing-masing kota, yaitu:

- Arad

  Bobot Kota Arad sampai Sibiu dan Kembali Lagi ke Arad = 280
  Bobot Garis Lurus Arad sampai Bucharest = 366
  Total Bobot = 646

- Fagaras

  Bobot Kota Arad sampai Fagaras = 239
  Bobot Garis Lurus Fagaras sampai Bucharest = 176
  Total Bobot = 415

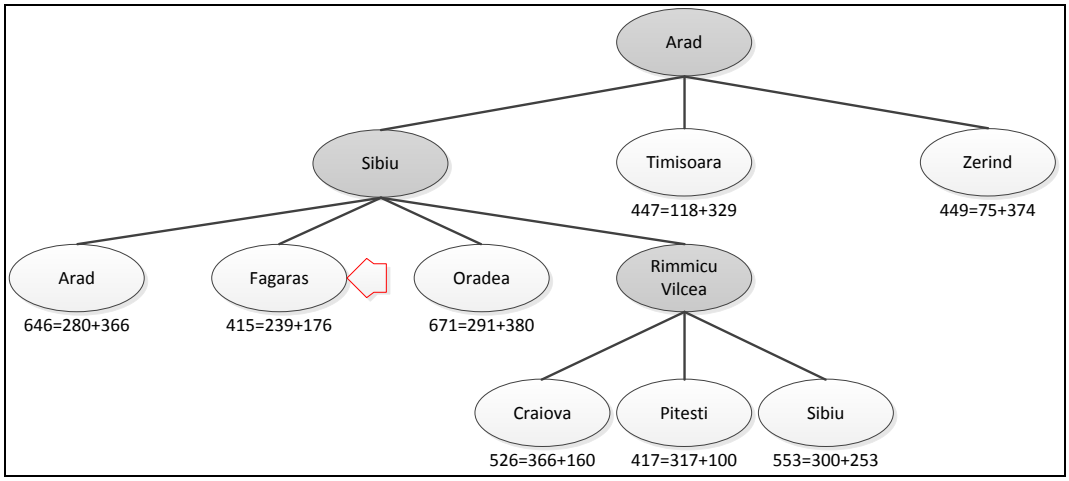
- Oradea,

  Bobot Kota Arad sampai Oradea = 291
  Bobot Garis Lurus Oradea sampai Bucharest = 380
  Total Bobot = 671

- Rimicu Vilcea

  Bobot Kota Arad sampai Rimicu Vilcea = 220
  Bobot Garis Lurus Rimicu Vilcea sampai Bucharest = 193
  Total Bobot = 413

Berdasarkan pencarian lintasan terpendek dari beberapa kota yang terhubung dengan kota Sibiu didapatkan bobot terkecil menuju Bucharest yaitu kota Rimicu Vilcea. Maka jalur yang kedua dilalui yaitu jalur menuju kota Rimicu Vilcea.



Gambar 2.5 Pencarian *Node* yang Akan Dilalui Ketiga

Selanjutnya lakukan pencarian *node* ketiga yang akan dilalui. Bobot dari masing-masing kota, yaitu:

- Craiova

Bobot Kota Arad sampai Craiova	=	366
Bobot Garis Lurus Craiova sampai Bucharest	=	160
Total Bobot	=	526

- Pitesti

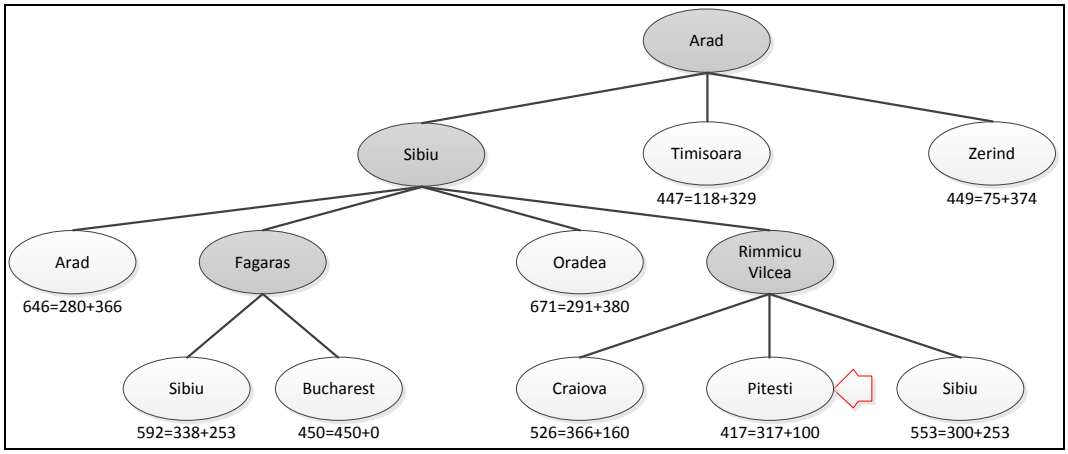
Bobot Kota Arad sampai Pitesti	=	317
Bobot Garis Lurus Pitesti sampai Bucharest	=	100
Total Bobot	=	417

- Sibiu

Bobot Kota Arad sampai Rimicu Vilcea dan Kembali Lagi ke Sibiu	=	300
Bobot Garis Lurus Sibiu sampai Bucharest	=	253
Total Bobot	=	553

Berdasarkan pencarian lintasan terpendek dari ketiga kota yang terhubung dengan kota Rimicu Vilcea didapatkan bobot masing-masing kota. Setelah

dibandingkan dengan hasil penjumlahan bobot tiap kota yang telah dilalui di pencarian ketiga ini, ternyata kota Fagaras memiliki bobot lebih kecil dari bobot total kota-kota pada pencarian ketiga ini.



Gambar 2.6 Pencarian *Node* yang Akan Dilalui Ketiga (Lanjutan)

Pencarian *node* ketiga kembali menuju kota Fagaras karena jika dibandingkan dengan bobot terkecil yang didapatkan dari pencarian sebelumnya (Rimicu Vilcea), kota Fagaras memiliki bobot terkecil. Bobot kota berikutnya yang akan dilalui melalui Kota Fagaras, yaitu:

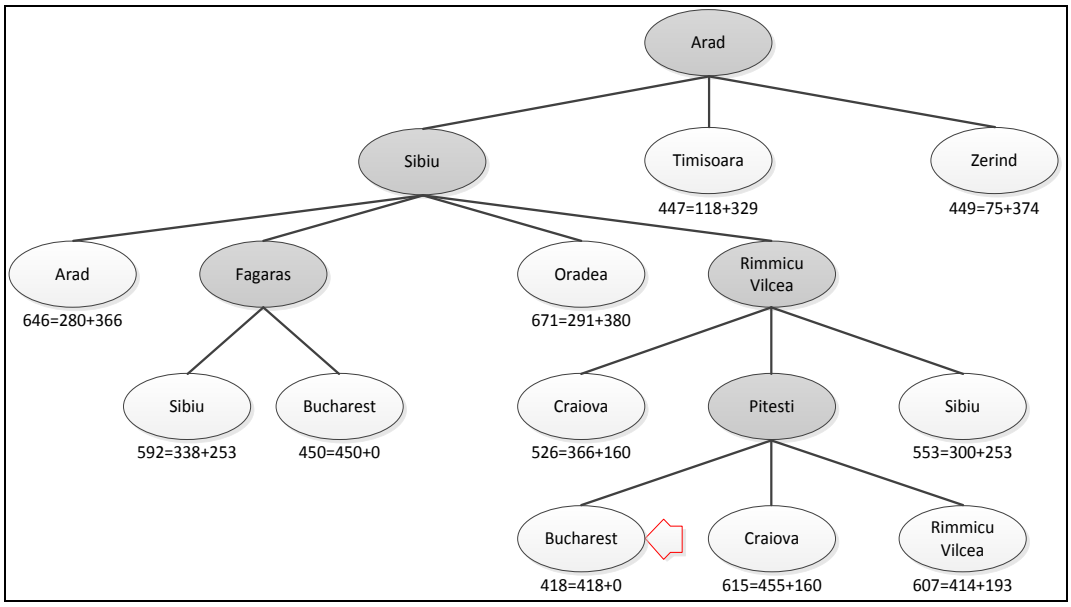
- Bucharest

Bobot Kota Arad sampai Bucharest	=	450
Bobot Garis Lurus Bucharest sampai Bucharest	=	0
Total Bobot	=	450

- Sibiu

Bobot Kota Arad sampai Fagaras dan Kembali Lagi ke Sibiu	=	338
Bobot Garis Lurus Sibiu sampai Bucharest	=	253
Total Bobot	=	591

Berdasarkan pencarian lintasan terpendek dari kota-kota yang terhubung dengan kota Fagaras didapatkan bobot masing-masing kota. Setelah dibandingkan dengan hasil penjumlahan bobot tiap kota yang telah dilalui pada pencarian *node* ketiga ini, ternyata kota Pitesti memiliki bobot lebih kecil dari bobot total kota-kota pada pencarian yang dimulai dari kota Fagaras.



Gambar 2.7 Pencarian Node yang Akan Dilalui Keempat

Pencarian *node* keempat yang akan dilalui dimulai dari kota Pitesti memiliki bobot:

- Bucharest

Bobot Kota Arad sampai Bucharest	=	418
Bobot Garis Lurus Bucharest sampai Bucharest	=	0
Total Bobot	=	418

- Craiova

Bobot Kota Arad sampai Craiova	=	455
Bobot Garis Lurus Craiova sampai Bucharest	=	160
Total Bobot	=	615

- Rimicu Vilce

Bobot Kota Arad sampai Pitesti dan Kembali Lagi Ke Rimicu Vilcea	=	414
Bobot Garis Lurus Rimicu Vilcea sampai Bucharest	=	193
Total Bobot	=	607

Berdasarkan pencarian *node* keempat yang akan dilalui, didapatkan bobot terkecil yaitu Bucharest, sekaligus didapatkan pula *node* tujuan yang ingin dicapai. Jika *node* tujuan telah dicapai maka pencarian akan dihentikan dan

lintasan terpendek dari kota asal menuju kota tujuan berdasarkan contoh di atas yaitu: Arad – Sibiu – Rimicu Vilcea – Pitesti – Bucharest.

2.3.2 Sifat A*

Seperti BFS, A* akan selalu menemukan sebuah solusi, jika memang ada. Apabila fungsi heuristik h dapat diterima, yang berarti nilai h tidak akan pernah melebihi ongkos minimal untuk meraih tujuan yang sebenarnya, maka A* sendiri akan dapat diterima (atau dapat disebut optimal) apabila himpunan tertutup tidak digunakan. Apabila digunakan sebuah himpunan tertutup, maka h harus monoton (konsisten) agar A* dapat menjadi optimal. Ini berarti fungsi heuristik tidak akan pernah berlebihan dalam menghitung ongkos dari sebuah titik ke titik tetangganya. Secara formal, untuk semua jalur x, y dimana y adalah suksesor dari x:

$$h(x) \leq g(y) - g(x) + h(y) \quad \textbf{(2.2)}$$

A* juga dapat dijamin keoptimalannya untuk sembarang heuristik, yang berarti bahwa tidak ada satu pun algoritma lain yang mempergunakan heuristik yang sama akan mengecek lebih sedikit titik dari A*, kecuali ketika ada beberapa solusi parsial dimana dapat dengan tepat memprediksi ongkos jalur minimal. [12]

2.3.3 Kompleksitas A*

Kompleksitas waktu dari A* sangat bergantung dari heuristik yang digunakannya. Pada kasus terburuk, jumlah titik yang diperiksa berjumlah eksponensial terhadap panjang solusi (lintasan terpendek), tetapi A* akan memiliki kompleksitas waktu polinomial apabila fungsi memenuhi kondisi berikut:

$$|h(x) - h^*(x)| \leq O(\log h^*(x) \quad \textbf{(2.3)}$$

Dimana h* adalah heuristik optimal, yaitu ongkos sebenarnya dari jalur x ke tujuan. Dengan kata lain, galat dari h tidak akan melaju lebih cepat dari logaritma dari heuristik optimal h* yang memberikan jarak sebenarnya dari x ke tujuan. [14]

Penggunaan memori dari A* bahkan lebih bermasalah dari kompleksitas waktunya. Beberapa varian dari A* telah dikembangkan untuk mengatasi masalah ini, termasuk diantaranya *Iterative-Deepening*-A*, *Memory-Bounded*-A* (MA*), *Simplified-Memory-Bounded*-A* (SMA*), dan *Recursive-Best-First-Search* (RBFS). RBFS juga akan memberikan hasil yang optimal apabila heuristiknya dapat diterima. [12]

2.3.4 Fungsi Heuristik untuk A*

Seperti yang telah disebutkan diatas, fungsi heuristik sangat berpengaruh terhadap kelakuan algoritma A*:

- Apabila h(n) selalu bernilai 0, maka hanya g(n) yang akan berperan, dan A* berubah menjadi Algoritma Dijkstra, yang menjamin selalu akan menemukan lintasan terpendek.
- Apabila h(n) selalu lebih rendah atau sama dengan ongkos perpindahan dari titik n ke tujuan, maka A* dijamin akan selalu menemukan lintasan terpendek. Semakin rendah nilai h(n), semakin banyak titik-titik yang diperiksa A*, membuatnya semakin lambat.
- Apabila h(n) tepat sama dengan ongkos perpindahan dari n ke tujuan, maka A* hanya akan mengikuti jalur terbaik dan tidak pernah memeriksa satupun titik lainnya, membuatnya sangat cepat. Walaupun hal ini belum tentu bisa diaplikasikan ke semua kasus, ada beberapa kasus khusus yang dapat menggunakannya.
- Apabila h(n) kadangkala lebih besar dari ongkos perpindahan dari n ke tujuan, maka A* tidak menjamin ditemukannya lintasan terpendek, tapi prosesnya cepat.
- Apabila h(n) secara relatif jauh lebih besar dari g(n), maka hanya h(n) yang memainkan peran, dan A* berubah menjadi BFS.

[12]

Berikut akan dibahas jenis-jenis heuristik untuk algoritma A* secara singkat:

1. Heuristik Eksak

Apabila heuristik yang digunakan selalu memberikan nilai yang tepat sesuai dengan panjang lintasan terpendek yang ada, A* hanya akan memeriksa sedikit titik. Ketika nilai h(n) selalu tepat mengimbangi g(x), maka nilai f(n) tidak akan pernah berubah sepanjang jalur. Semua titik yang tidak berada di titik yang tepat akan memiliki nilai f yang lebih tinggi dibanding titik yang berada pada jalur yang tepat. Karena A* tidak akan memeriksa titik yang memiliki nilai f yang besar sebelum selesai memeriksa titik-titik yang memiliki nilai f yang lebih kecil, maka A* tidak akan pernah keluar dari lintasan terpendek.

a. Heuristik Eksak yang Telah Dikomputasi

Salah satu cara untuk mengonstruksi heuristik eksak adalah menghitung jarak dari tiap-tiap pasang titik yang terdapat pada peta. Cara ini jelas tidak memungkinkan untuk kebanyakan situasi, jadi cara ini tidak akan dibahas lebih lanjut.

b. Heuristik Eksak Linear

Pada beberapa keadaan khusus, heuristik eksak bahkan tidak perlu dikomputasi sebelumnya, Apabila peta yang akan digunakan tidak memiliki tembok (halangan) apapun, maka dapat dibuktikan dengan mudah bahwa lintasan terpendek dari sebuah pasangan titik pada peta sudah pasti sebuah garis lurus.

2. Heuristik Non-Eksak

Ada beberapa jenis heuristik non-eksak yang sering digunakan:

a. Jarak Manhattan (*Manhattan Distance*)

Heuristik yang paling umum digunakan adalah jarak Manhattan. Fungsi heuristik ini hanya akan menjumlahkan selisih nilai x dan nilai y dari dua buah titik. Heuristik ini dinamakan Manhattan mungkin karena di kota Manhattan di Amerika, jarak dari dua lokasi umumnya dihitung dari blok-blok yang harus dilalui saja dan tentunya tidak bisa dilintasi secara diagonal. Perhitungannya dapat ditulis sebagai berikut:

h(n) = abs(n.x-tujuan.x) + abs(n.y-tujuan.y)

(2.4)

b. Jarak Euclid

Heuristik ini akan menghitung jarak berdasarkan garis yang dapat ditarik dari dua buah titik. Perhitungannya dapat ditulis sebagai berikut:

$$h(n) = \sqrt{(n.x - tujuan.x)^2 + (n.y - tujuan.y)^2}$$

(2.5)

Dalam kasus ini, skala relatif nilai g mungkin akan tidak sesuai lagi dengan nilai fungsi heuristik h. Karena jarak Euclid selalu lebih pendek dari jarak Manhattan, maka dapat dipastikan selalu akan didapatkan lintasan terpendek, walaupun secara komputasi lebih berat.

c. Jarak Euclid yang Dikuadratkan

Dalam beberapa literatur juga disebutkan jika nilai g adalah 0, maka lebih baikjika ongkos komputasi operasi pengakaran pada heuristik jarak Euclid dihilangkan saja, menghasilkan rumus sebagai berikut:

$$h(n)^2 = (n.x - tujuan.x)^2 + (n.y - tujuan.y)^2$$

(2.6)

Hal lain yang harus diperhatikan adalah seberapa cepat fungsi heuristik dapat dikomputasi. Selalu akan ada *trade-off* antara akurasi dari fungsi heuristik dan waktu yang dibutuhkannya untuk mengomputasinya. Nampaknya bagus jika fungsi heuristik yang digunakan sangat akurat, dilihat dari berbagai macam percobaan bahwa jika heuristik yang digunakan sempurna maka A* akan selalu melewati jalur yang tepat dan akan selalu memberikan optimum global. Namun, heuristik yang sempurna semacam itu tidak ada (dan tidak akan pernah ada), dan bahkan untuk mendekatinya saja akan memerlukan tambahan komputasi yang tidak ringan. Seringkali dalam aplikasinya heuristik yang memberikan hasil yang sangat akurat namun lambat, kurang disenangi dibanding heuristik yang tidak begitu optimal namun memberikan hasil dengan cepat. Maka dari itu, pemilihan heuristik sangat bergantung pada tujuan penggunaan A*. Jika hasil yang dibutuhkan adalah optimum global, maka fungsi heuristik yang digunakannya haruslah "sempurna", sedang jika yang dibutuhkan adalah hasil yang cepat dan tidak harus lintasan terpendek, maka lebih bijak menggunakan heuristik yang lebih ringan.

2.4 *Haversine Formula*

Haversine Formula adalah persamaan yang digunakan dalam navigasi, yang memberikan jarak lingkaran besar antara dua titik pada permukaan bola (bumi) berdasarkan garis lintang (*latitude*) dan garis bujur (*longitude*) [5]. Dapat dikatakan *haversine formula* digunakan untuk menghitung jarak antara dua titik pada permukaan bumi berdasarkan garis lintang (*latitude*) dan garis bujur (*longitude*) [1]. Penggunaan rumus ini mengasumsikan pengabaian efek ellipsoidal, cukup akurat untuk sebagian besar perhitungan, juga pengabaian ketinggian bukit dan kedalaman lembah di permukaan bumi. [2]

Haversine Formula dituliskan sebagai berikut:

$$d = 2r\sin^{-1}\left(\sqrt{sin^2\left(\frac{\phi_2 - \phi_1}{2}\right) + cos(\phi_1)cos(\phi_2)sin^2\left(\frac{\psi_2 - \psi_1}{2}\right)}\right) \quad \textbf{(2.7)}$$

Dimana:

d	:	Jarak antara dua titik berdasarkan *latitude* dan *longitude*
r	:	Radius bumi
ϕ_1	:	*Latitude* awal
ϕ_2	:	*Latitude* akhir
ψ_1	:	*Longitude* awal
ψ_2	:	*Longitude* akhir

2.5 Eclipse

Eclipse adalah sebuah IDE (*Integrated Development Environment*) untuk mengembangkan perangkat lunak dan dapat dijalankan di semua platform (*platform-independent*). Eclipse awalnya dikembangkan oleh IBM untuk menggantikan perangkat lunak pengembangan IBM Visual Age for Java 4.0. Produk Eclipse ini diluncurkan oleh IBM pada tanggal 5 November 2001. IBM menginvestasikan US$ 40 juta untuk pengembangannya. Sejak 5 November 2001, konsorsium Eclipse Foundation megambil alih pengembangan Eclipse lebih lanjut. [4]

Sifat dari Eclipse adalah sebagai berikut:

1. *Multi-platform*: Target sistem operasi Eclipse adalah Microsoft Window, Linux, Solaris, AIX, HP-UX, dan Mac OS X.
2. *Multi-language*: Eclipse dikembangkan dengan bahasa pemrograman Java, akan tetapi Eclipse mendukung pengembangan aplikasi berbasis bahasa pemrograman lain seperti C/C++, Cobol, Python, Perl, PHP, dan lain sebagainya.
3. *Multi-role*. Selain sebagai EDR untuk pengembangan aplikasi. Eclipse pun bisa digunakan untuk aktifitas dalam siklus pengembangan perangkat lunak seperti dokumentasi, pengujian perangkat lunak, pengembangan web, dan lain sebagainya.

[4]

Eclipse pada dasarnya merupakan sebuah *kernel*. Apa yang dapat digunakan di dalam Eclipse sebenarnya adalah fungsi dari *plug-in* yang sudah dipasang (diinstal). Ini merupakan basis dari Eclipse yang dinamakan *Rich Client Platform* (RCP). Komponen yang membentuk RCP adalah:

1. *Core platform*,
2. OSGi,
3. SWT(*Standard Widget Toolkit*),
4. JFace, dan
5. *Eclipse Workbench*.

[4]

Konsep Eclipse adalah IDE, yaitu:

1. Terbuka (*open*),
2. Mudah diperluas (*extensible*) untuk apa saja, dan
3. Tidak untuk sesuatu yang spesifik.

[4]

Eclipse merupakan program yang paling popular untuk pengembangan Android, karena memiliki *plug-in* yang tersedia untuk memfasilitasi pengembangan Android. Eclipse juga mengapat dukungan langsung dari Google, hal ini terbukti dengan adanya penambahan *plug-in* untuk membuat *project* Android, dimana *source software* langsung dari situs resminya Google. [15]

2.6 *Geocoding*

Letak sebuah lokasi (alamat jalan) merupakan bentuk umum dari informasi lokasi. Walaupun masih merupakan informasi dalam bentuk teks yang berisi nomor rumah, nama jalan, arah, dan kode pos. Untuk mendapatkan satu mekanisme dimana informasi dapat ditransfer menjadi informasi yang tepat harus dilakukan pencarian koordinat geografi sebelum satu alamat bisa ditampilkan pada satu peta. [6]

Geocoding (pencocokan lokasi) merupakan suatu proses untuk menggabungkan satu lokasi fisik di bumi dengan lokasi logiknya. Informasi dari satu lokasi didapatkan dari perhitungan koordinat, sehingga didapatkan jarak, arah, dan lokasi yang tepat [6]. *Geocoding* merupakan proses merubah sebuah alamat menjadi titik geografis yang berguna untuk mereferensikan sebuah titik geografis, selain itu untuk menentukan batas-batas geografis sebuah obyek. [3]

2.7 SQLite

SQLite merupakan sebuah sistem manajemen basis data selasional yang bersifat *ACID-compliant* dan memiliki ukuran pustaka kode yang relatif kecil, ditulis dalam bahasa C. SQLite merupakan proyek yang bersifat *public domain* yang dikerjakan oleh D. Richard Hipp. [17]

Tidak seperti pada paradigma *client-server* umumnya, inti SQLite bukanlah sebuah sistem yang mandiri yang berkomunikasi dengan sebuah program, melainkan sebagai integral dari sebuah program secara keseluruhan sehingga protokol komunikasi utama yang digunakan adalah melalui pemanggilan API secara langsung melalui bahasa pemrograman. Mekanisme seperti ini tentunya membawa keuntungan, karena dapat mereduksi *overhead*, *latency times*, dan secara keseluruhan lebih sederhana. Seluruh elemen basis data (definisi data, tabel, indeks, dan data) disimpan sebagai sebuah *file*. Kesederhanaan dari sisi desain tersebut bisa diraih dengan cara mengunci keseluruhan *file* basis data pada saat sebuah transaksi dimulai Hipp. [17]

SQLite merupakan alat untuk melakukan penanganan data (*data store*) yang telah termasuk dalam Android. SQLite sendiri terkenal sebagai *database* yang *open source*, *stand alone SQL database*, berukuran kecil, tidak membutuhkan

administrasi, tanpa *server*, tanpa *file* konfigurasi dan juga telah digunakan pada banyak aplikasi terkenal seperti *Browser Firefox* dan *IPhone*. [11]

2.8 Skobbler

Skobbler (http://maps.skobbler.com/) adalah *map source freeware* dan *open source* yang memungkinkan kita mengunduh peta secara keseluruhan. Peta yang telah diunduh dapat digunakan, karena telah mencakup semua data spasial di dalam peta. [10]

2.9 Android

Android adalah sistem operasi yang bersifat *open source*, dan google merilis kodenya di bawah lisensi Apache [15]. Kode dengan sumber terbuka dan lisensi perizinan pada Android memungkinkan perangkat lunak untuk dimodifikasi secara bebas dan didistribusikan oleh para pengembang aplikasi.

Android merupakan subset perangkat lunak untuk perangkat *mobile* yang meliputi sistem operasi, *middle ware*, dan aplikasi inti yang di-*release* oleh Google. Sedangkan Android SDK (*Software Development Kit*) menyediakan *tools* yang diperlukan untuk mengembangkan aplikasi pada *platform* Android.

Dikembangkan bersama antara Google, HTC, Intel, Motorola, Qualcomm, T-Mobile, NVIDIA yang tergabung dalam OHA (*Open Handset Alliance*) dengan tujuan membuat sebuah standar terbuka untuk perangkat bergerak (*mobile device*).

2.9.1 Fitur Android

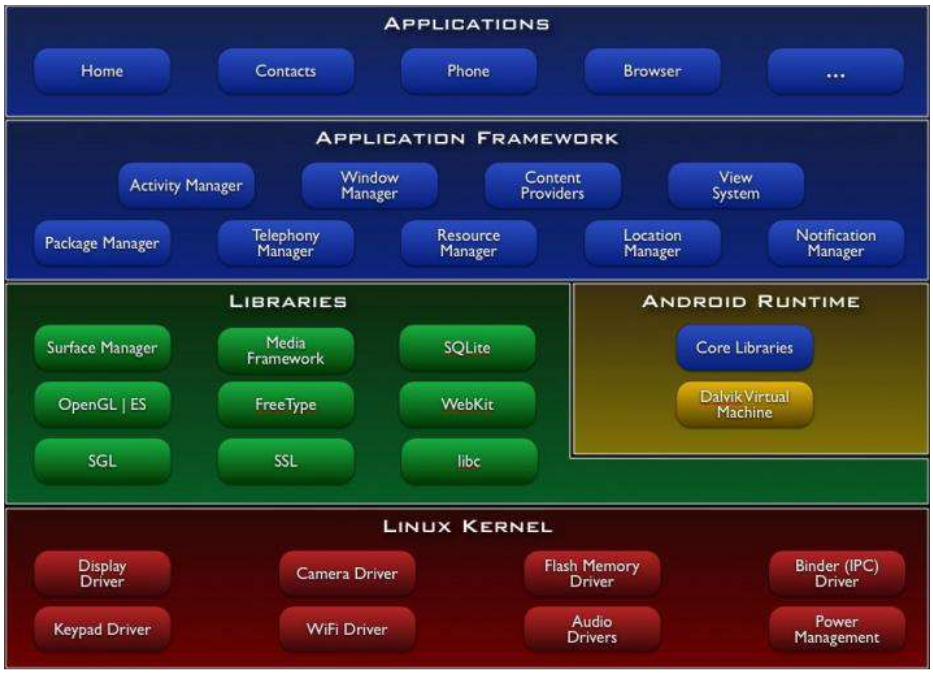
Fitur yang tersedia pada *platform* Android antara lain:

1. *Framework* aplikasi yang mendukung penggantian komponen dan *reusable*.
2. Mesin virtual Dalvik berjalan di atas Linux *kernel* dan dioptimalkan untuk perangkat *mobile*.
3. *Integrated browser* berdasarkan *open source engine Web Kit*.
4. Grafis yang dioptimalkan dan didukung oleh *library* grafis 2D yang terkostumisasi, grafis 3D berdasarkan spesifikasi open GL ES 1,0 (Opsional akselerasi *hardware*).
5. SQLite untuk penyimpanan data.

6. *Media Support* yang mendukung audio, video, dan gambar (MPEG4, H.264, MP3, AAC, AMR, JPG, PNG, GIF).
7. GSM *Telephone* (tergantung *hardware*).
8. *Bluetooth*, EDGE, 3D, dan WiFi (tergantung *hardware*).
9. Dukungan Perangkat Tambahan: Android dapat memanfaatkan kamera, layar sentuh, *accelerometers*, *magnetometers*, GPS, akselerasi 2D (dengan perangkat orientasi, *scalling*, konversi format piksel), dan akselerasi grafis 3D.
10. *Multi-touch*: kemampuan layaknya *handset modern* yang dapat menggunakan dua jari atau lebih untuk berinteraksi dengan perangkat.
11. Lingkungan *development* yang lengkap dan kaya termasuk perangkat *emulator*, *tools* untuk *debugging*, profil dan kinerja memori, dan *plugin* untuk Eclipse IDE.
12. *Market*: seperti kebanyakan *handphone* yang memiliki tempat penjualan aplikasi, *market* pada Android merupakan katalog aplikasi yang dapat di-*download* dan di-*install* pada *handphone* melalui internet.

2.9.2 Arsitektur Android

Arsitektur Android menunjukkan komponen-komponen utama yang terdapat pada sistem operasi Android. Arsitektur *platform* Android digambarkan sebagai berikut:



Gambar 2.8 Arsitektur *Platform* Android

2.10 Kota Bandung

Kota Bandung merupakan kota metropolitan terbesar di Jawa Barat sekaligus menjadi ibu kota provinsi tersebut. Kota ini terletak 140 km sebelah tenggara Jakarta dan merupakan kota terbesar ketiga di Indonesia setelah Jakarta dan Surabaya menurut jumlah penduduk. Sedangkan wilayah Bandung Raya (wilayah metropolitan Bandung) merupakan metropolitan terbesar ketiga di Indonesia setelah Jabodetabek dan Gerbang Kertosusila (Gerbang Kertosusilo). Di kota ini tercatat berbagai sejarah penting, di antaranya sebagai tempat berdirinya sebuah perguruan tinggi teknik pertama di Indonesia (*Technische Hoogeschool te Bandung* – TH Bandung, sekarang Institut Teknologi Bandung – ITB), lokasi ajang pertempuran di masa kemerdekaan, serta pernah menjadi tempat berlangsungnya Konferensi Asia-Afrika (KAA) pada tahun 1955, suatu pertemuan yang menyuarakan semangat anti kolonialisme, bahkan Perdana Menteri India Jawaharlal Nehru dalam pidatonya mengatakan bahwa Bandung adalah ibu kotanya Asia-Afrika. Pada tahun 1990, kota Bandung terpilih sebagai salah satu kota paling aman di dunia berdasarkan survei majalah *Time*.

Kota Kembang merupakan sebutan lain untuk kota ini, karena pada zaman dahulu kota ini dinilai sangat cantik dengan banyaknya pohon-pohon dan bunga-bunga yang tumbuh. Selain itu, Bandung disebut juga dengan *Parijs Van Java*,

karena keindahannya. Selain itu juga dikenal sebagai kota belanja (*City of Heritage*), karena *mall* dan *factory outlet* yang banyak tersebar di kota ini. Saat ini berangsur-angsur kota Bandung menjadi kota wisata kuliner. Dan pada tahun 2007, British Council menjadikan kota Bandung sebagai *pilot project* kota terkreatif se-Asia Timur. Saat ini kota Bandung merupakan salah satu kota tujuan utama pariwisata dan pendidikan. Kota Bandung memiliki semboyan "Bermartabat (Bersih, Makmur, Taat, dan Bersahabat)". [9]