2. LANDASAN TEORI

2.1. *Shortest Path Problem*

Shortest Path Problem adalah sebuah permasalahan untuk menemukan sebuah jalan antara 2 *vertex*, dimana *weight* dari unsur pokok diminimalisasi. Salah satu contoh dari *shortest path problem* yang paling sering dibahas adalah *Travelling Salesman Problem* dimana permasalahan tersebut adalah permasalahan untuk menemukan jarak terpendek untuk memutari semua *vertex* yang ada dan kemudian kembali pada titik awal pencarian rute (Cormen, Leiserson, Rivest, & Stein, 2001).

2.2. *Multiple Destination Path Finding Problem*

Multiple destination path finding problem adalah permasalahan untuk menemukan solusi jalan yang harus melalui beberapa tempat sekaligus. Permasalahan ini dapat diselesaikan dengan dua pendekatan yaitu *brute force* dan *heuristic* (Champandard, 2007).

2.2.1. *Brute Force*

Pendekatan pertama yang dapat dilakukan adalah dengan menghitung *shortest path* dari kombinasi kombinasi yang dapat dibentuk dari sekumpulan titik tujuan (P). Hal ini dapat dilakukan dengan tiga cara:

- Lakukan *single-pair algorithm* (Contoh: A*) pada semua pasangan tujuan pada P.
- Lakukan *single-source algorithm* (Contoh: Dijkstra) pada setiap tujuan P.
- Lakukan sebuah *all-pairs algorithm* (Contoh: Floyd-Warshall)

Langkah selanjutnya adalah memilih kumpulan *shortest path* yang memiliki jarak paling minimal. Hal ini dapat dilakukan dengan metode *brute force* atau mencoba semua kemungkinan kombinasi yang ada untuk dicari kombinasi mana yang memiliki jarak terminimal.

Kelemahan dari metode ini adalah waktu perhitungan yang cukup lama, dimana harus dilakukan pencarian *shortest path* untuk setiap kombinasi tujuan yang ada.

2.2.2. *Heuristic*

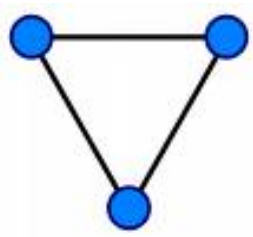
Pendekatan lain untuk menyelesaikan masalah *multiple destination path finding problem* adalah dengan metode *heuristic*. Metode ini dilakukan dengan melakukan pemilihan tujuan yang memiliki jarak paling dekat untuk setiap *step* yang terjadi. Pemilihan jarak tujuan terdekat dilakukan dengan menggunakan *shortest path algorithm* pada setiap tujuan dari titik berada sekarang. *Step* selanjutnya dilakukan dengan cara yang sama, yaitu mencari lagi titik tujuan selanjutnya yang terdekat sampai semua tujuan sudah tercapai (Champandard, 2007).

2.3. *Graph*

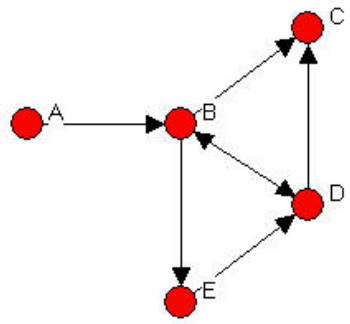
2.3.1. Definisi Graph

Graph adalah sebuah representasi dari sekumpulan objek di mana terdapat hubungan antar objek tersebut. Objek yang saling terhubung tersebut dikenal dengan istilah matematika *vertex* Graph adalah sebuah *ordered pair* G = (V,E) yang terdiri dari kumpulan *vertex* dan *edge* dari sebuah *graph*. V disebut sebagai *vertex set*, sedangkan E disebut sebagai *edge set*. *Graph* dapat memiliki arah dan tidak, *graph* yang tidak memiliki arah disebut dengan *undirected graph* dan *graph* yang memiliki arah disebut dengan *directed graph*.

Dalam sebuah *graph*, *vertex* dapat dikatakan bertetangga atau biasa disebut dengan *adjacent* apabila ada sebuah *edge* yang menghubungkannya dalam sebuah *graph* (Chartrand & Lesniak, 1996).



Gambar 2.1. Undirected Graph



Gambar 2.2. Directed Graph / Digraph

2.3.2. Representasi Graph

Ada 2 cara untuk menyimpan informasi dari *graph*, yaitu dengan menggunakan *adjacency matrix* dan *adjacency list.* Kedua representasi tersebut dapat diaplikasikan untuk *undirected graph* dan *digraph*. Representasi *graph* menggunakan *adjacency list* biasanya banyak digunakan untuk merepresentasikan *sparse graph*. Representasi *adjacency list* dari sebuah *graph* G = (*V*,*E*) terdiri dari sebuah *array Adj* sepanjang |V| atau sejumlah *vertex* dimana lemen dari *array Adj* merupakan sebuah *adjacency list* (Cormen, Leiserson, Rivest, & Stein, 2001).

2.4. Algoritma *Heuristic*

Algoritma *Heuristic* adalah sebuah algoritma yang dapat menghasilkan solusi yang dapat diterima walaupun tidak ada bukti bahwa solusi yang dihasilkan tersebut benar. Sebagai kemungkinan lain, solusi yang dihasilkan dapat dikatakan benar, namun tidak dapat dikatakan sebagai solusi yang optimal. Algoritma *heuristic* biasanya digunakan apabila tidak ada cara yang diketahui untuk menemukan solusi yang optimal dengan batasan – batasan tertentu seperti waktu, ruang dan sebagainya (Pearl, 1984).

2.5. *Ant System*

Ant System adalah sebuah algoritma yang digunakan untuk menyelesaikan permasalahan optimisasi yang terinsipirasi dari perilaku semut untuk menemukan sumber makanannya. cara kerja *Ant System* ini adalah ketika semut menemukan sumber makanan maka semut perlu menentukan jalur terpendek dari sarangnya ke sumber makanan, beberapa semut akan melalui

semua jalur yang ada secara acak kemudian jalur yang terpendek akan diberi feromon, dengan adanya feromon, maka semut-semut selanjutnya tidak akan berjalan secara acak lagi, namun akan mengikuti jalur yang ada feromon. Semakin banyak semut melalui suatu jalur, semakin banyak pula jumlah feromon yang tertinggal di jalur tersebut. Semut yang lain memilih rute dengan feromon yang banyak pula (Dorigo, 1996).

Setiap semut adalah sebuah agen dengan karakteristik sebagai berikut:

- Memilih kota yang menjadi tujuannya berdasarkan probabilitas. Rumus probabilitasnya adalah :

$$p_{i,j} = \frac{(\tau_{i,j}^{\alpha})(\eta_{i,j}^{\beta})}{\sum(\tau_{i,j}^{\alpha})(\eta_{i,j}^{\beta})} \qquad (2.1)$$

  dimana,

  $\tau_{i,j}$ adalah jumlah dari feromon pada tanda *i,j*

  α adalah sebuah *parameter* untuk mengontrol pengaruh dari $\tau_{i,j}$

  $\eta_{i,j}$ adalah keinginan dari tanda *i,j* dimana $\eta_{ij} = 1/ d_{ij.}$

  β adalah sebuah *parameter* untuk mengontrol pengaruh dari $\eta_{i,j}$

- Semut tidak boleh melalui kota yang sudah dilewati. Hal ini dapat dikontrol dengan *tabu list*.
- Apabila *tour* sudah selesai maka dilakukan penambahan feromon pada garis penghubung antar kota yang merupakan solusi dari rute tersebut. Rumus penambahan feromon adalah sebagai berikut :

  $\tau ij\ (t+n) = \rho.\tau ij(t) + \Delta\tau ij$ (2.2)

  dengan *trail persistence* (ρ) yang merupakan koefisien sehingga (1-ρ) adalah penguapan feromon antara waktu t hingga t + n.

$$\Delta\tau_{ij} = \sum_{k=1}^{m} \Delta\tau_{ij}^{k} \qquad (2.3)$$

  dimana $\Delta\tau_{ij}^{k}$ adalah nilai dari feromon dari jalan i ke j yang diletakkan oleh semut ke – k dari antara waktu t hingga t + n, dengan rumus:

$$\Delta\tau_{ij}^{k} = \begin{cases} \frac{Q}{L_k} & \text{if k - th ant uses edge (i, j) in its tour (between time t and t + n)} \\ 0 & \text{otherwise} \end{cases} \quad (2.4)$$

dimana Q adalah konstanta dan L_k adalah panjang dari *tour* yang dilalui oleh semut k.

Berikut ini adalah langkah – langkah dari *Ant System* (Khan, 2004):

1. Inisialisasi jumlah semut = 2xn, sampai dengan n pada titik awal dan semut n+1 sampai dengan 2n pada titik tujuan. *cycle counter* diisi dengan 0 dan nilai solusi terbaik diisi dengan nilai terbesar.
2. Pemilihan persimpangan yang dilalui dengan menggunakan rumus (2.1). Persimpangan yang telah dilalui dimasukkan ke dalam *tabu list*.
3. a. Apabila ada semut yang sudah menyelesaikan *tour* pada langkah ini maka dilakukan *update* feromon pada rute yang dilalui dengan menggunakan rumus (2.2)

   b. Rute yang telah ditemukan dibandingkan, rute yang terbaik disimpan sebagai solusi.

   c. Untuk semua semut yang telah menyelesaikan *tour*, dilakukan pengosongan *tabu list* kecuali persimpangan yang paling terakhir.
4. *Cycle counter* ditambah 1. Apabila *counter* telah mencapai nilai maksimum atau tidak ada perubahan selama 3xn maka perhitungan dihentikan dan nilai dari solusi dianggap sebagai solusi terbaik jika tidak maka kembali ke langkah ke – 2.

2.6. Algoritma Dijkstra

Algoritma Dijkstra adalah algoritma yang banyak digunakan untuk menyelesaikan *single source shortest path problem*, yaitu permasalahan pencarian jalur terpendek dari *vertex* asal (*source*) ke semua *vertex* lain dalam sebuah *graph*, dimana pada setiap *edge* yang menghubungkan *vertex* terdapat *weight* yang bernilai tidak negatif. Algoritma Dijkstra juga dapat digunakan untuk menyelesaikan permasalahan - permasalahan lain berikut:

- *Single pair shortest path problem*
- *Single destination shortest path problem*

- *All pairs shortest path problem*

Berikut ini adalah *pseudocode* dari algoritma Dijkstra:

```
1  function Dijkstra(Graph, source):
2    for each vertex v in Graph:          // Initializations
3       dist[v] := infinity               // Unknown distance function from source to v
4        previous[v] := undefined         // Previous node in optimal path from source
5    dist[source] := 0                    // Distance from source to source
6    Q := the set of all nodes in Graph
     // All nodes in the graph are unoptimized - thus are in Q
7    while Q is not empty:                // The main loop
8       u := vertex in Q with smallest dist[]
9       if dist[u] = infinity:
10          break                         // all remaining vertices are inaccessible from source
11      remove u from Q
12      for each neighbor v of u:         // where v has not yet been removed from Q.
13         alt := dist[u] + dist_between(u, v)
14         if alt < dist[v]:              // Relax (u,v,a)
15            dist[v] := alt
16            previous[v] := u
17   return previous[]
```

Gambar 2.3. Pseudocode Dijkstra

Sumber : Cormen, Leiserson, Rivest, & Stein, 2001.

Running time dari algoritma Dijkstra ini tergantung dari struktur data yang digunakan dan tergantung dari *priority queue* yang diimplementasikan, apabila menggunakan *array* sepanjang $|V|$, maka *running time* dari Dijkstra ini adalah $O(|V|^2+|E|)=O(|V|^2)$. Jika *priority queue* menggunakan struktur data *binary heap* maka *running time* dari algoritma Dijkstra ini adalah $O((|E|+|V|) \log |V|)$. (Cormen, Leiserson, Rivest, & Stein, 2001).

2.7. Algoritma A* (A Star)

Menurut (Russel & Norvig, 2003) Algoritma A* adalah algoritma *best-first search* yang paling banyak dikenal. Algoritma ini memeriksa node dengan menggabungkan *g(n)*, yaitu *cost* yang dibutuhkan untuk mencapai sebuah node dan *h(n)* yaitu cost yang didapat dari node ke tujuan. Sehingga didapatkan rumus dasar dari algoritma A* ini adalah:

$$f(n) = g(n) + h(n) \quad (2.5)$$

Apabila kita ingin menemukan solusi terbaik, hal yang paling masuk akal untuk dicoba pertama kali adalah node dengan nilai terendah yaitu *g(n) + h(n)*. Dan ternyata cara ini lebih dari hanya sekedar masuk akal: dengan kondisi bahwa

fungsi *heuristic* memenuhi kondisi tertentu. Pencarian dengan algoritma A* lengkap dan optimal.

Keoptimalan dari A* ini cukup langsung untuk dianalisa apabila digunakan dengan *tree search*. Pada kasus ini, A* dinilai optimal jika *h(n)* adalah sebuah *admissible heuristic* yaitu nilai *h(n)* tidak akan memberikan penilaian lebih pada *cost* untuk mencapai tujuan. Salah satu contoh dari *admissible heuristic* adalah jarak dengan menarik garis lurus karena jarak terdekat dari dua titik adalah dengan menarik garis lurus. Gambar untuk ilustrasi penghitungan A* secara manual dapat dilihat pada Gambar 2.4.

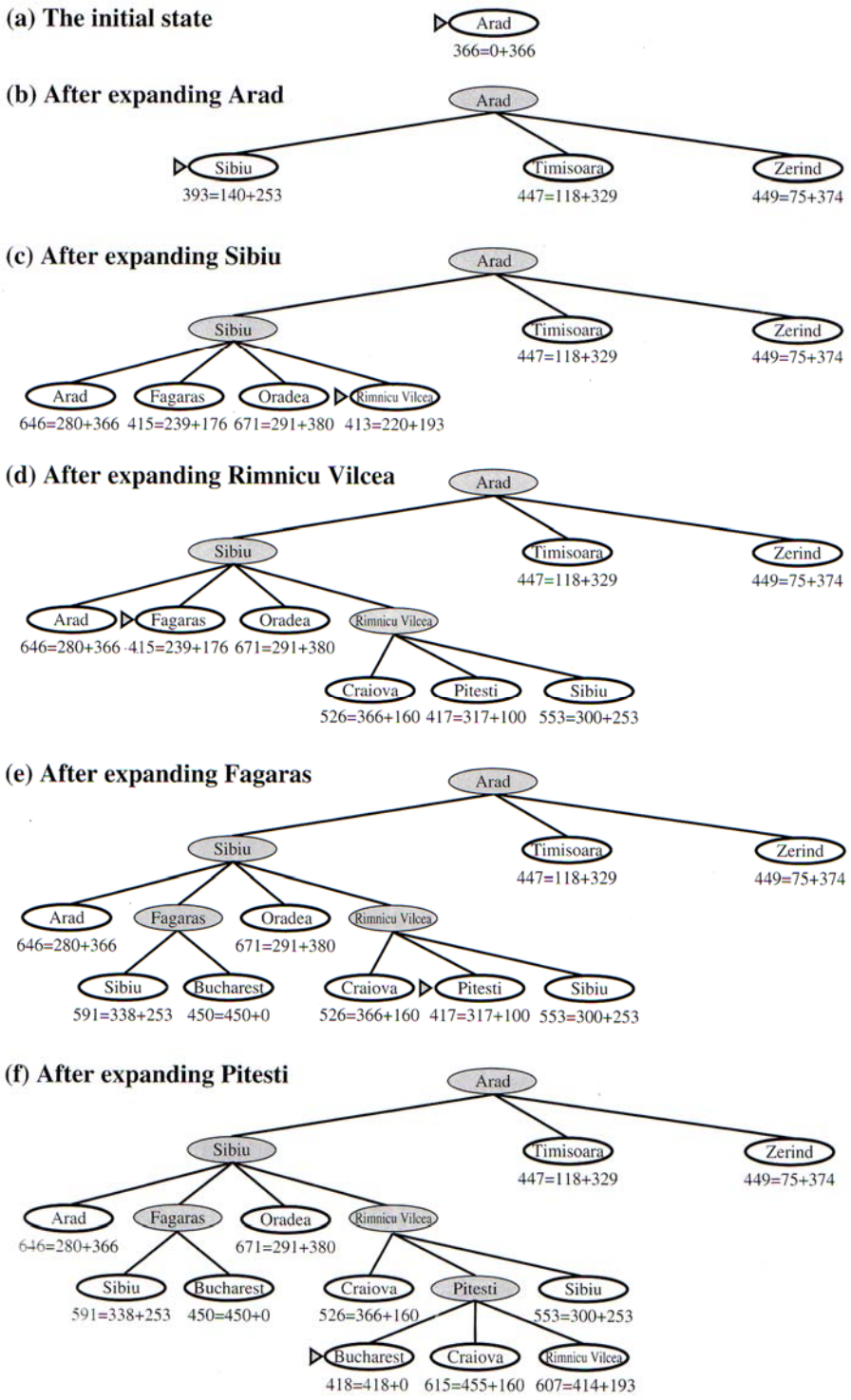
Berikut ini adalah pseudocode dari algoritma A* (Nillson, N. J, 1998):

```
function A*(start,goal)
     closedset := the empty set                 % The set of nodes
already evaluated.
     openset := set containing the initial node % The set of tentative
nodes to be evaluated.
     g_score[start] := 0                        % Distance from start
along optimal path.
     h_score[start] := heuristic_estimate_of_distance(start, goal)
     f_score[start] := h_score[start]           % Estimated total
distance from start to goal through y.
     while openset is not empty
         x := the node in openset having the lowest f_score[] value
         if x = goal
             return reconstruct_path(came_from,goal)
         remove x from openset
         add x to closedset
         foreach y in neighbor_nodes(x)
             if y in closedset
                 continue
             tentative_g_score := g_score[x] + dist_between(x,y)

             if y not in openset
                 add y to openset

                 tentative_is_better := true
             elseif tentative_g_score < g_score[y]
                 tentative_is_better := true
             else
                 tentative_is_better := false
             if tentative_is_better = true
                 came_from[y] := x
                 g_score[y] := tentative_g_score
                 h_score[y] := heuristic_estimate_of_distance(y, goal)
                 f_score[y] := g_score[y] + h_score[y]
     return failure

 function reconstruct_path(came_from,current_node)
     if came_from[current_node] is set
         p = reconstruct_path(came_from,came_from[current_node])
         return (p + current_node)
     else
         return the empty path
```



Gambar 2.4. Contoh Perhitungan Pencarian Rute dengan A*

Sumber: Russel& Norvig (2003, p.98)

2.8. *Euclidean Distance*

(Black, 2004) Pengertian *Euclidean Distance* secara umum adalah jarak dari garis lurus antara dua titik. Pada jarak 2 dimensi dengan p1 pada (x1,y1) dan p2 pada (x2,y2), persamaan rumusnya adalah :

$$\sqrt{(x_1 - x_2)^2 + (y_1 - y_2)^2} \tag{2.6}$$