BAB 2

LANDASAN TEORI

2.1 Sistem

Dalam bukunya yang berjudul Perancangan *Object Oriented Software*, Djon Irwanto (2005) menjelaskan pengertian sistem dari sudut pandang sistem informasi berorientasi objek adalah sekumpulan komponen yang mengimplementasikan model dan fungsionalitas yang dibutuhkan. Komponen-komponen tersebut saling berinteraksi di dalam sistem guna mentransformasikan *input* yang diberikan kepada sistem tersebut menjadi *output* yang berguna dan bernilai bagi aktornya.

Menurut Roger S. Pressman (2001, p246) sistem adalah sekumpulan set objek-objek yang saling berelasi membentuk suatu kesatuan yang saling mengisi.

2.2 Manajemen Parkir

Definisi tempat parkir adalah suatu area/tempat yang dapat menampung kendaraan parkir dengan limit tertentu. (Salim Azzabi, 2004)

Parkir merupakan kebutuhan bagi tiap pengunjung yang mendatangi suatu lokasi umum maupun komersil seperti *mall*, gedung perkantoran, hotel, ruko, kampus, dan lain sebagainya. Pengelola parkir tentu menghendaki agar area parkirnya dapat memberikan kenyamanan bagi pengguna parkir. (Nucira, 2007)

Pengertian manajemen adalah suatu cara untuk merencanakan, mengumpulkan dan mengorganisir, memimpin dan mengendalikan sumber daya untuk suatu tujuan. (Anonymous, 2005)

Manajemen parkir dapat diartikan sebagai pengkoordinasian elemen-elemen perparkiran yang saling terkait.

Hal-hal yang berkaitan dengan manajemen parkir antara lain :

- Tambah, hapus, edit pengguna (*supervisor* atau *operator*) tanpa batas.
- Tambah, hapus, edit pintu masuk dan pintu keluar serta dapat merubah alur dari pintu masuk menjadi pintu keluar dan sebaliknya.
- Tambah, hapus, edit jenis kendaraan.
- Tambah, hapus, edit data pelanggan parkir.
- Menentukan tarif dan denda parkir berdasarkan jam dan jenis kendaraan.
- Menentukan konfigurasi pencetakan label karcis dan *setting barcode*.

Manajemen parkir mengarah kepada berbagai kebijaksanaan dan program yang menghasilkan sumber-sumber parkir yang lebih efisien. (Nucira, 2007)

2.3 Algoritma

Istilah algoritma pertama kali diperkenalkan oleh Abu Ja'far Mohamed ibn Musa al Khuwarizmi pada tahun 825 A.D dalam buku aljabarnya. Dalam buku tersebut algoritma disebut sebagai suatu metode khusus yang digunakan untuk menyelesaikan suatu masalah. Definisi Algoritma adalah langkah-langkah logis penyelesaian masalah yang disusun secara sistematis dan logis. (Gramacom, 2005)

Algoritma pada umumnya mempunyai definisi prosedur komputasi yang membutuhkan beberapa nilai, atau beberapa set nilai, sebagai *input* dan menghasilkan beberapa nilai atau beberapa set nilai sebagai *output*. Algoritma juga dapat dilihat sebagai alat untuk menyelesaikan masalah komputasi. Algoritma menggambarkan

prosedur komputasi secara spesifik untuk memperoleh hubungan antara *input / output*. (Thomas H. Cormen, 2001, p5)

Bila dipandang dalam ilmu komputer, pengertian algoritma adalah suatu fungsi yang terdiri dari rangkaian langkah-langkah yang terstruktur dan ditulis secara sistematis yang akan dikerjakan untuk menyelesaikan masalah dengan bantuan komputer.

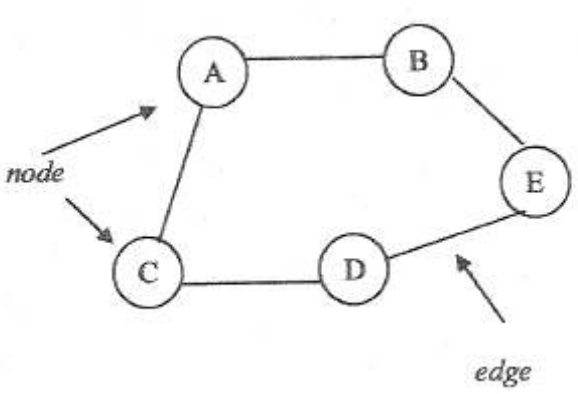
2.4 Struktur Data

Menurut Anna Maria (1998, P1) data adalah bahan yang digunakan dalam perhitungan atau operasi untuk menghasilkan informasi yang berguna. Struktur adalah pengaturan atau hubungan. Jadi struktur data adalah pengaturan atau hubungan dari data di dalam suatu sistem.

2.4.1 Graph

Graph adalah suatu struktur data yang berbentuk network/jaringan dimana hubungan antar elemen-elemennya adalah many-to many.

Menurut Wiitala (1987, P178), graph adalah sebuah pasangan yang berurutan dari (v,e) dimana v adalah kumpulan *vertex/node* dan e adalah kumpulan dari *edge* atau kumpulan dari garis yang menghubungkan antara vertex yang satu dengan vertex yang lain.

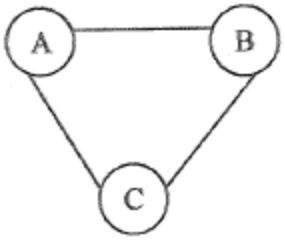


Gambar 2.1 ***Graph***

Graph dapat dibedakan menjadi 2 tipe, yaitu :

- Undirected Graph

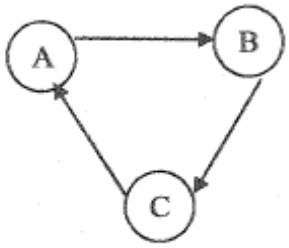
  Jika beberapa *node* yang membentuk *edge* dalam graph tidak mempunyai arah.



Gambar 2.2 ***Undirected Graph***

- Directed Graph

  Jika sepasang *node* yang membentuk *edge* dalam graph mempunyai arah.



Gambar 2.3 ***Directed Graph***

Special graph adalah graph yang memiliki urutan/susunan yang khusus antara *edge* dan vertexnya, beberapa diantaranya adalah :

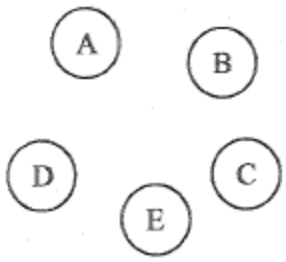
1. Null Graph

   Adalah graph yang hanya terdiri dari set vertex tanpa memiliki set *edge*.

   Null graph biasa direpresentasikan dengan symbol Nn, dimana :

   N : Null Graph

   n : Jumlah Vertex



Gambar 2.4 ***Null Graph***

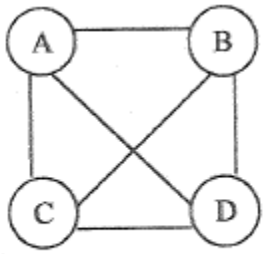
2. Complete Graph

   Suatu graph dimana vertex yang satu berhubungan dengan semua vertex lain / semua vertex saling berhubungan.

   Complete graph biasa direpresentasikan dengan symbol Kn, dimana :

   K : Complete Graph

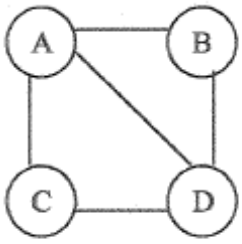
   n : Jumlah Vertex



Gambar 2.5 ***Complete Graph***

3. Planar Graph

   Adalah graph dimana bila graph tersebut digambarkan maka tidak ada *edge* yang saling bersilangan.



Gambar 2.6 ***Planar Graph***

4. Bipartite Graph

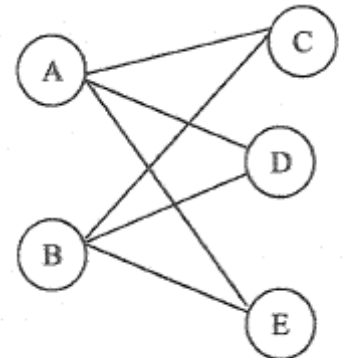
   Adalah graph dimana set vertexnya dibagi ke dalam dua buah sub vertex m dan n. Setiap vertex dari m harus memiliki *edge* yang terhubung kepada semua vertex dari n, demikian sebaliknya, setiap vertex dari n harus memiliki *edge* yang terhubung kepada semua vertex dari m. Bipartite graph biasanya direpresentasikan dengan symbol Km,n

   Dimana :

   Km,n : Bipartite Graph

   m : jumlah sub set vertex m

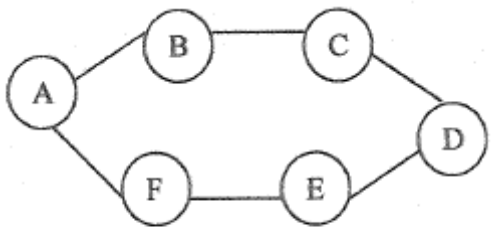
   n : jumlah sub set vertex



Gambar 2.7 ***Bipartite Graph***

5. Regular Graph

   Adalah graph dimana setiap vertexnya memiliki derajat yang sama.



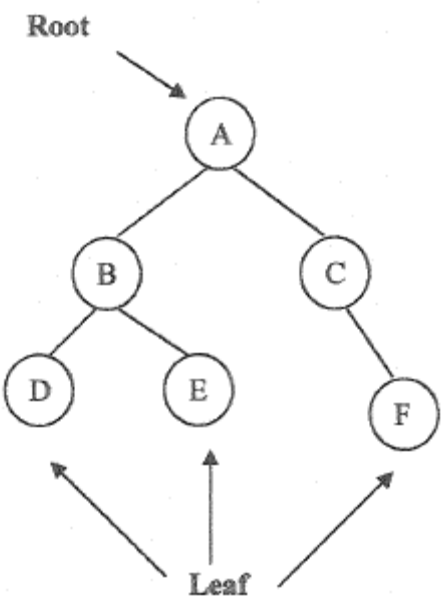
Gambar 2.8 *Regular Graph*

2.4.2 Tree

Tree merupakan struktur data yang mempunyai hubungan one-to-many. Hubungan one-to-many ini meliputi juga hubungan one-to-one atau one-to-zero, dapat dijelaskan bahwa satu parent bisa memiliki satu atau nol atau lebih dari satu child. Elemen dalam tree disebut dengan *node*.

Karakteristik dari tree adalah :

- Terdapat satu *node* yang unik, yang tidak memiliki predecessor. *Node* ini disebut root.
- Terdapat satu atau beberapa *node* yang tidak mempunyai successor. *Node* ini disebut leaf.
- Setiap *node* kecuali root, pasti memiliki satu predecessor yang unik.
- Setiap *node* kecuali leaf, pasti memiliki satu atau lebih successor.



Gambar 2.9 *Tree*

Hubungan Parent-Child :

- Parent adalah predecessor langsung dari suatu *node*. Semua *node* kecuali Root pasti memilik satu parent yang unik.
- Child adalah successor langsung dari suatu *node*. Semua *node*, kecuali Leaf pasti memiliki satu atau lebih Child.
- *Node-node* yang memiliki Parent yang sama disebut Sibling.

2.5 Analisis Algoritma

Yang dimaksud dengan analisis algoritma adalah proses menganalisa, menentukan besarnya biaya yang diperlukan suatu algoritma untuk memecahkan suatu permasalahan. (Weiss, Mark Allen, 1996, p149)

Menganalisa algoritma dapat diartikan sebagai proses memprediksi sumber-sumber yang dibutuhkan oleh algoritma. Terkadang sumber-sumber seperti memori, *communication bandwith,* atau perangkat keras komputer menjadi hal-hal yang utama, tetapi sering kali waktu komputasi menjadi nilai yang ingin dihitung. Umumnya, dengan menganalisa beberapa algoritma dalam suatu masalah, hal utama yang paling efisien dapat dengan mudah diidentifikasi. (Thomas H. Cormen, 2001, p21)

2.6 Kompleksitas Waktu

Besarnya waktu yang dibutuhkan sebuah algoritma untuk menyelesaikan suatu permasalahan selalu bergantung terhadap jumlah inputan yang harus diproses. Semakin besar data maka semakin besar waktu yang dibutuhkan. Sebagai contoh, untuk mengurutkan 10.000 elemen membutuhkan lebih banyak waktu daripada untuk mengurutkan 10 elemen. Tetapi, pada keadaan sebenarnya nilai dari suatu fungsi algoritma tergantung pada banyak faktor, misalnya kecepatan komputer, kualitas dari compiler, dan kualitas dari program itu sendiri. (Weiss, Mark Allen, 1996, p149).

Waktu dari sebuah algoritma dapat diukur dengan menghitung banyaknya operasi / instruksi yang dieksekusi. Di dalam sebuah algoritma mungkin terdapat banyak sekali jenis-jenis operasi, misalnya operasi penjumlahan, operasi pengisian nilai, operasi pembagian, operasi perbandingan, operasi pembacaan, pemanggilan prosedur, dan sebagainya. Jika besaran waktu (dalam satuan detik) untuk melaksanakan sebuah operasi tertentu telah diketahui, maka waktu yang dibutuhkan untuk melaksanakan algoritma tersebut dapat dihitung. (Munir, Rinaldi, 2003, p418)

2.7 Fungsi Heuristic

Menurut Amit.J.Patel (2003, p1), *heuristic* merupakan aturan-aturan untuk memilih cabang-cabang yang memiliki kemungkinan mengarah pada pemecahan masalah. Karena *heuristic* menggunakan informasi yang terbatas maka *heuristic* mungkin gagal dalam memprediksi perilaku secara tepat dalam suatu pencarian. Heuristic dapat membantu menunjukkan arah yang tepat bagi suatu algoritma, tetapi mungkin juga gagal dalam memberikan petunjuk kepada algoritma tersebut.

Algoritma A* tanpa fungsi *heuristic* yang baik akan memperlambat pencarian dan dapat menghasilkan rute yang tidak tepat. Untuk menghasilkan rute yang benar-benar tepat, maka fungsi *heuristic*-nya harus *underestimate* terhadap biaya dari suatu *node* ke *final node*. Fungsi *heuristic* yang sempurna akan membuat algoritma A* langsung menuju final *node* tanpa harus mencari ke arah-arah lain. Sehingga jika fungsi *heuristic*-nya terlalu underestimate akan menyebabkan algoritma ini beranggapan bahwa ada rute yang lebih baik dari rute yang ada. Untuk fungsi heuristic yang underestimate, bila nilainya terlalu rendah akan menyebabkan algoritma ini seperti algoritma Dijkstra yang mencari ke segala arah yang mungkin. Hal ini dikarenakan tidak cukupnya informasi mengenai masalah yang dihadapi, sehingga menyebabkan algoritma A* melakukan pencarian lebih banyak dan lebih lama. Jika algoritma ini diharapkan melakukan pencarian dengan lebih cepat tanpa memperoleh rute terpendek, maka fungsi *heuristic*-nya suatu saat dapat *overestimate*.

Beberapa fungsi *heuristic* yang umum digunakan pada algoritma *pathfinding* A* adalah sebagai berikut (Patel, Amit.J., 2003, p1) :

1. Manhattan Distance

   Manhattan Distance adalah fungsi *heuristic* standar untuk algoritma A*.

Digunakan pada aplikasi yang memiliki 4 arah gerakan (tidak dapat bergerak diagonal).

h(n) = d * (abs(Xn - Xgoal) + abs (Yn - Ygoal))

Dimana :

- d adalah nilai biaya. Dimana nilai d didapat dari nilai *minimum cost* perpindahan antar *node*
- Xn adalah koordinat X dari *node* pertama pada *grid*
- Xgoal adalah koordinat X dari *final node*
- Yn adalah koordinat dari *node* pertama pada *grid*
- Ygoal adalah koordinat Y dari *final node.*

2. Straight Line Distance

Straight Line Distance adalah fungsi *heuristic* yang digunakan pada aplikasi yang dapat bergerak ke segala arah / sudut.

$h(n) = sqrt((Xn – Xgoal)^2 + (Yn – Ygoal)^2)$

Dimana :

- Xn adalah koordinat X dari *node* pertama pada *grid*
- Xgoal adalah koordinat X dari *final node*
- Yn adalah koordinat dari *node* pertama pada *grid*
- Ygoal adalah koordinat Y dari *final node.*

3. Diagonal Distance

Diagonal *Distance* adalah fungsi *heuristic* yang digunakan pada aplikasi yang memiliki delapan arah gerakan (dapat bergerak diagonal).

h(n) = d * max abs (Xn – Xgoal), abs (Yn – Ygoal))

Dimana :

- d adalah nilai biaya. Dimana nilai d didapat dari nilai *minimum cost* perpindahan antar *node*
- Xn adalah koordinat X dari *node* pertama pada *grid*
- Xgoal adalah koordinat X dari *final node*
- Yn adalah koordinat dari *node* pertama pada *grid*
- Ygoal adalah koordinat Y dari *final node.*

2.7.1 Perbandingan Fungsi *Heuristic* :

1. Menggunakan fungsi *heuristic manhattan distance*
    - Semua jalur-jalur dapat ditemukan (masalah dapat dipecahkan).
    - Hal ini disebabkan karena pada setiap penambahan nilai g(n), pada perhitungan nilai *heuristic*-nya terjadi pula perubahan pada nilai d-nya. Sehingga dengan penambahan nilai g(n), tidak mempengaruhi pencarian jalur.
    - Dengan menggunakan fungsi *heuristic manhattan distance*, didapatkan nilai iterasi dan jumlah langkah yang paling kecil dibanding dengan menggunakan fungsi *heuristic* yang lain.
2. Menggunakan fungsi *heuristic straight line distance*
    - Masalah tidak semuanya dapat dipecahkan. Terutama pada pengujian dengan menggunakan nilai g yang besar.
    - Dalam algoritma A Star dengan fungsi *heuristic straight line distance*, apabila terdapat sedikit hambatan pada ruang pencarian,

maka walaupun ditemukan jalurnya, pasti membutuhkan banyak iterasi.

- Kesulitan dalam pencarian jalur tersebut, terjadi karena dalam perhitungan nilai fungsi *heuristic straight line distance* yaitu :

  $h(n) = sqrt ((Xn-Xgoal)^2 + (Yn-Ygoal)^2)$, maka nilai yang dihasilkan kecil sehingga dalam pencarian nilai f(n), banyak dipengaruhi oleh besarnya nilai g(n). Hal ini dapat dilihat, semakin besar nilai g(n) maka semakin kecil pengaruh nilai fungsi *heuristic* tersebut terhadap pencarian nilai f(n), karena nilai fungsi *heuristic* pada setiap pengujian tersebut nilainya tetap sedangkan nilai g(n) semakin besar, maka nilai fungsi *heuristic* tersebut semakin tidak berpengaruh. Dimana nilai pencarian tersebut digunakan dalam memilih node yang akan dimasukkan ke dalam *close list.*

- Karena alasan diatas, dapat disimpulkan bahwa fungsi *straight line distance* memiliki lebih banyak iterasi dibanding fungsi *heuristic* yang lain.

3. Menggunakan fungsi *heuristic diagonal distance*

   - Semua jalur-jalur dapat ditemukan (masalah dapat dipecahkan).
   - Jumlah iterasi dan jumlah langkah yang didapat pada pengujian dengan fungsi *heuristic diagonal distance* lebih sedikit dibanding dengan fungsi *heuristic straight line distance* tetapi lebih besar dibanding dengan fungsi *heuristic manhattan distance*.

(Mario, Irawan, Aryo, 2004, p121)

Jika perhitungan algoritma A Star dilakukan pada grid, maka fungsi *heuristic* yang tepat adalah menggunakan *manhattan distance*. Sedangkan untuk algoritma A Star pada graph, fungsi *heuristic* yang tepat adalah *straight line distance* dimana hasil yang didapat akan mempunyai cost yang lebih kecil daripada fungsi *heuristic manhattan distance* dan *diagonal distance*.

Apabila pada grid nilai g untuk vertikal, horizontal dan diagonalnya dianggap sama, fungsi *heuristic* yang tepat adalah dengan menggunakan *heuristic diagonal distance*. (Mario, Irawan, Aryo, 2004, p150)

2.8 Algoritma Pathfinding

Tujuan dari algoritma *pathfinding* adalah untuk menemukan jalur terbaik dari *vertex* awal ke *vertex* akhir. Secara umum algoritma *pathfinding* digolongkan menjadi dua jenis (Russel, Stuart dan Peter Norvig, 1995, 73), yaitu :

1. Algoritma *Uniformed Search*

   Algoritma *uniformed search* adalah algoritma yang tidak memiliki keterangan tentang jarak atau biaya dari *path* dan tidak memiliki pertimbangan akan *path* mana yang lebih baik. Yang termasuk dalam algoritma ini adalah algoritma *Breadth-First Search.*

2. Algoritma *Informed Search*

   Algoritma *informed search* adalah algoritma yang memiliki keterangan tentang jarak atau biaya dari *path* dan memiliki pertimbangan berdasarkan pengetahuan akan *path* mana yang lebih baik. Yang termasuk algoritma ini adalah algoritma Dijkstra dan algoritma A*.

Dalam algoritma *pathfinding* seringkali terjadi *backtrack* bila tidak menemukan solusi. *Backtrack* merupakan suatu algoritma pelacakan yang mencoba mencari penyelesaian masalah yang menyeluruh dengan membangun solusi *partial*. Masalah yang akan diselesaikan dengan fungsi *backtrack* harus memenuhi suatu set kendala (*constraint*). Dalam prosesnya, *backtrack* akan mundur ke solusi *partial* sebelumnya, jika terdapat solusi yang cocok dengan tuntutan masalah.

2.8.1 Algoritma Depth-First Search

Menurut Stuart Russel dan Peter Norvig (1995,77), *depth-first search* selalu memperluas (*expand*) salah satu *node*-nya pada *level* yang paling dalam dari sebuah *tree*. Hanya ketika pencarian bertemu dengan jalan buntu, maka pencarian kembali ke *node* semula dan memperluas (*expand*) *node* pada tingkat yang lebih dangkal.

Algoritma *depth-first search* memiliki kompleksitas waktu $O(b^m)$. Untuk masalah yang memiliki banyak solusi, *depth-first search* dapat lebih cepat dibanding dengan *breadth-first search*, karena *depth-first search* memiliki suatu kemungkinan yang bagus ketika menemukan sebuah solusi, dimana hanya menjelajah sebagian kecil dari keseluruhan bagian.

Kekurangan dari *depth-first search* yaitu dapat terjebak menjelajahi *path* / jalur yang salah. Maka *depth-first search* tidak akan pernah bisa untuk mnemukan kembali jalur semula, ketika ia memilih jalur yang salah.

2.8.2 Algoritma Breadth-First Search

Menurut Luger dan Stubblefield (1993, p92), algoritma *Breadth-First Search* adalah algoritma yang memeriksa *node* dalam *tree* secara *level* per *level*. Jika dalam

suatu *level* ditemukan *goal*, maka pencarian dihentikan dan *goal*-nya akan diberikan sebagai *goal* yang sukses. Jika semua *node* dalam suatu *level* telah diperiksa dan tidak ditemukan *goal*, maka algoritma ini akan berpindah memeriksa ke *level* berikutnya, demikian seterusnya. Jika semua *node* pada *level* yang paling bawah telah diperiksa dan belum ditemukan *goal*, maka pencarian dihentikan dan *goal*nya diberikan sebagai *goal* yang gagal.

Algoritma *Breadth-First Search* diimplementasikan dengan menggunakan dua buah *list*, yaitu *open* dan *close*, Luger dan Stubblefield (1993, p92). *Open* berisi daftar *node-node* yang telah siap untuk diperiksa, tetapi *children* dari *node-node* tersebut belum di-*generate*. Sedangkan *close* berisi daftar *node-node* yang telah diperiksa tetapi bukan merupakan *goal*. *Open list* dari algoritma *Breadth-First Search* merupakan *queue* yang berpola *First In First Out* (FIFO), dimana *node* yang akan diperiksa terlebih dahulu adalah *node* yang terlebih dahulu masuk ke dalam *list*.

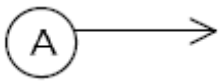
Menurut Luger dan Stubblefield (1993, p97-98), keuntungan dari algoritma *Breadth-First Search* adalah dijaminnya diperoleh *path* terpendek jika *goal*-nya ditemukan (karena algoritma ini akan selalu memeriksa semua *node* pada *level* n sebelum berpindah ke *level* n+1). Juga bahwa tidak semua *node* harus diperiksa, karena jika *goal* telah ditemukan sebelum seluruh *node* diperiksa, maka pencarian dapat dihentikan. Namun algoritma *Breadth-First Search* memiliki kelemahan, yaitu semua nilai / biaya dari *node* harus sama, sehingga tidak dapat diterapkan pada *tree* yang bernilai.

2.8.3 Algoritma Dijkstra

Algoritma Dijkstra dipublikasikan pertama kali oleh E.W.Dijkstra pada tahun 1959. Menurut Stout (1997, p1) algoritma Dijkstra merupakan pengembangan dari algoritma *breadth-first search*, dimana pada algoritma Dijkstra setiap *edge*-nya memiliki nilai dan selalu bernilai positif. Perbedaan yang lain terletak pada *open list*-nya. *Open list* pada algoritma Dijkstra merupakan *priority queue*, dimana *vertex* dengan prioritas tertinggi akan diproses terlebih dahulu, yaitu *vertex* yang memiliki nilai terkecil pada *open list*. Jadi pengaturan *priority queue* pada Dijkstra dipengaruhi oleh nilai *edge* kumulatif dari *vertex a*wal sampai *vertex* akhir.

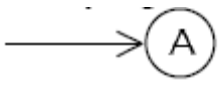
Berdasarkan terminologi teori graph, maka suatu jaringan akan terdiri dari suatu himpunan titik-titik yang disebut *node*. *Node-node* tersebut saling dihubungkan oleh suatu garis dan disebut *edge*. Beberapa terminologi tambahan dari jaringan ini adalah :

- Siklus, yaitu lintasan yang diawali pada suatu *node* dan diakhiri pada *node* itu juga.
- Tree, yaitu suatu jaringan dengan lintasan yang menghubungkan pasangan – pasangan *node*, dimana siklus tidak terjadi.
- Busur maju, yaitu busur yang meninggalkan *node*.



Gambar 2.10 Busur Maju

- Busur mundur, yaitu busur yang masuk ke dalam *node*.



Gambar 2.11 Busur Mundur

- Kapasitas aliran, yaitu batas aliran yang fisibel pada busur tertentu.
- Sumber, yaitu *node* yang menjadi awal dari busur-busurnya.
- Tujuan, yaitu *node* yang menjadi tujuan busur-busurnya.

2.8.3.1 Persoalan Rute Terpendek

Untuk setiap dua *node* S dan T dapat terjadi beberapa lintasan, dimana lintasan dengan bobot yang minimum disebut sebagai lintasan atau rute terpendek. Bobot di sini dapat berupa jarak, waktu tempuh, atau ongkos transportasi dari satu *node* ke *node* yang lainnya yang membentuk rute tertentu.

Algoritma mencari rute terpendek ini dikembangkan oleh Djikstra. Algoritma tersebut digunakan apabila semua busur jaringan mempunyai bobot positif. Langkah-langkah penyelesaiannya yaitu :

1) Buatlah tabel seperti dibawah ini :

Node				
Bobot				
Seleksi				

Tabel 2.1 Tabel Penyelesaian Algoritma Dijkstra

2) Isilah baris *node* dengan seluruh *node* yang ada, dan baris bobot dengan nilai ~ sebagai nilai awal.

3) Pilihlah *node* awal (misal *node* A) untuk diseleksi

4) Carilah *node* yang berhubungan dengan *node* A (misal *node* B dan C).

Tulislah tabel dengan bobot tersebut pada baris bobot dan kolom *node* B untuk *edge* antara

node A dan B, serta kolom *node* C untuk *edge* antara *node* A dan C, "Apabila

bobot tersebut lebih kecil dari nilai bobot sebelumnya".

5) Apabila suatu *node* telah diseleksi, tulislah tanda (X) pada baris seleksi dan kolom *node* yang terseleksi

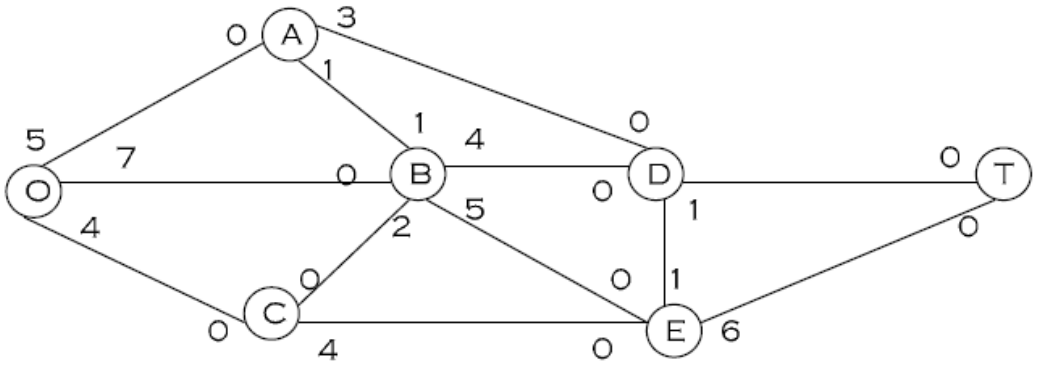
6) Berdasarkan tabel, pilihlah *node* yang belum diseleksi dan mempunyai bobot terkecil untuk diseleksi.

7) Ulangilah langkah 4, hingga semua *node* terseleksi.

8) Dengan mengurutkan *node* dari belakang ke depan, jalur terpendeknya akan diketahui. (Michael, 1987)

2.8.3.2 Persoalan Aliran Maksimum

Persoalan yang muncul selanjutnya dalam jaringan adalah bagaimana menentukan rute-rute perjalanan sedemikian sehingga jumlah total perjalanan yang dilakukan setiap harinya menjadi maksimum, tanpa melanggar batas maksimum perjalanan yang dapat dilakukan pada masing-masing jalan. Dalam hal ini data (informasi) yang diajukan dalam persoalan tersebut berupa jumlah perjalanan pada masing-masing jalan yang menghubungkan suatu tempat dengan tempat lain beserta kapasitasnya. Untuk jelasnya, ambil contoh datanya pada gambar 1 berikut :



Gambar 2.12 Contoh Data

Gambar di atas dapat dibaca seperti berikut ini :

a) Dari O ke A dapat dilakukan perjalanan maksimum 5 kali setiap hari, sedangkan dari A ke O tidak ada perjalanan yang dapat dilakukan.

b) Dari A ke B dapat dilakukan perjalanan sebanyak 1 kali perjalanan, begitu juga dari B dapat dilakukan perjalanan 1 kali ke A, dan seterusnya.

Dalam hal ini diasumsikan bahwa perjalanan masuk ke suatu *node* sama dengan perjalanan keluar dari *node* itu. Jika kapasitas busur (i,j) adalah cij, maka tingkat aliran pada busur (i,j) yaitu jumlah aliran dari *node* i ke *node* j, adalah bilangan positif yang tidak lebih besar daripada cij. Dengan demikian, jika tingkat aliran pada busur (i,j) dinyatakan oleh fij, maka :

$$O \leq f_{ij} \leq c_{ij}$$

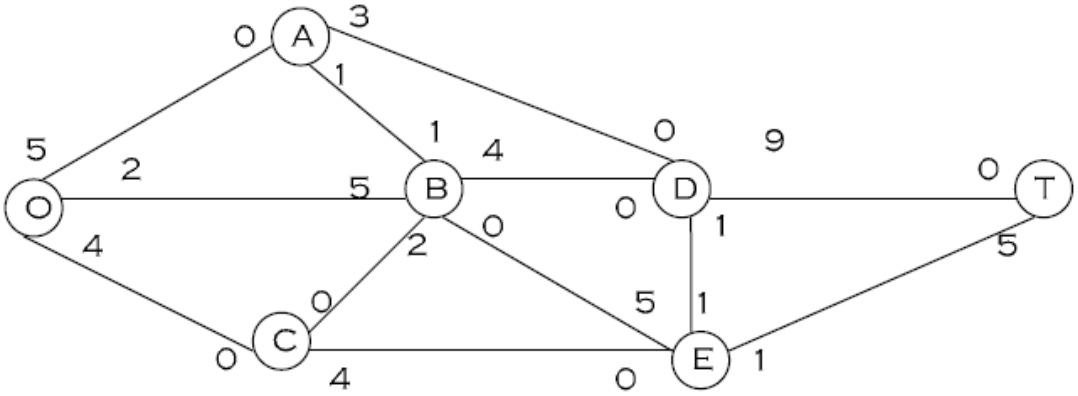
Sebenarnya, persoalan aliran maksimum ini dapat diformulasikan sebagai persoalan programa linier sehingga dapat diselesaikan dengan metode simpleks. Akan tetapi, di sini akan dikemukakan satu prosedur penyelesaian yang lebih efisien, seperti berikut :

1. Carilah lintasan dari sumber ke tujuan dengan kapasitas aliran positif.

2. Periksalah lintasan terebut untuk mendapat busur dengan kapasitas aliran terkecil (nyatakan kapasitas ini sebagai c*), dan tingkatkanlah aliran pada lintasan tersebut sebesar c*.

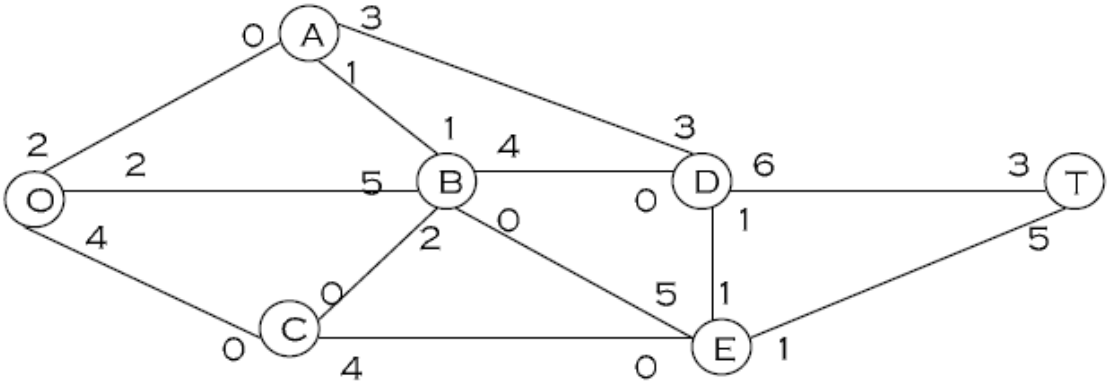
3. Kurangkan kapasitas aliran semula dengan c* pada setiap busur dari lintasan yang dimaksud. Tingkatkan kapasitas aliran semula dengan c* pada setiap busur yang berlawanan arah dari arah lintasan tersebut, dan kembali ke langkah 1.

Langkah 1 : Dari contoh diatas dipilih lintasan O – B – E – T. Alirkan sebesar 5



Gambar 2.13 Langkah Solusi 1

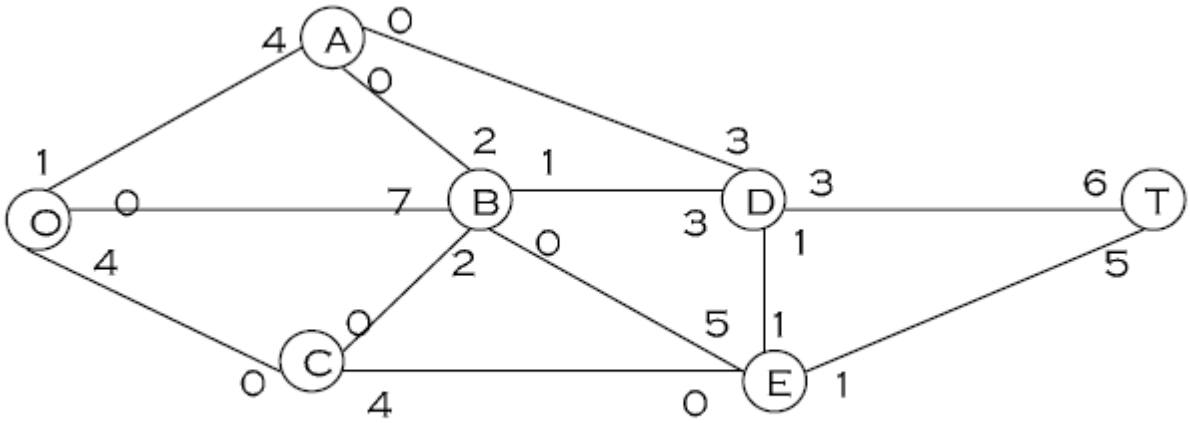
Langkah 2 : Alirkan sebanyak 3 pada lintasan O – A – D – T, hasilnya adalah :



Gambar 2.14 Langkah Solusi 2

Langkah 3 : Alirkan sebanyak 1 pada lintasan O – A – B – D – T

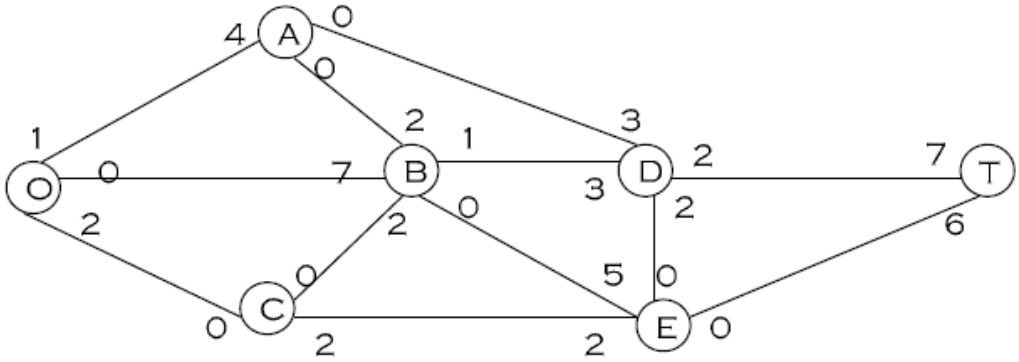
Langkah 4 : Alirkan sebanyak 2 pada lintasan O – B – D – T dan hasilnya adalah :



Gambar 2.15 Langkah Solusi 4

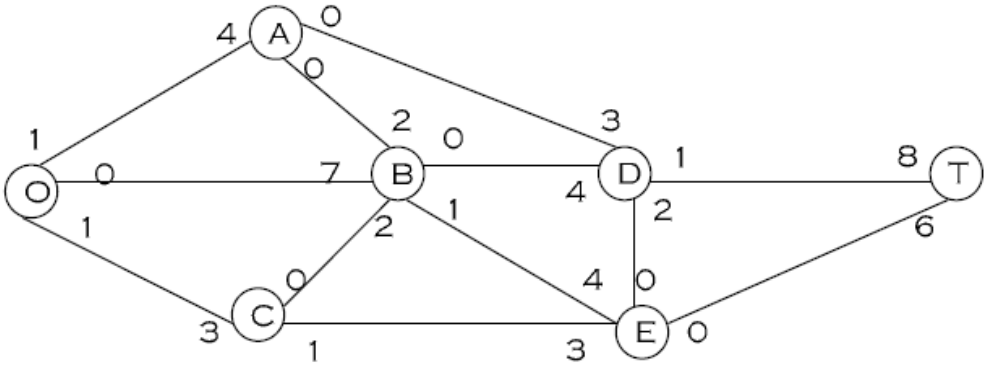
Langkah 5 : alirkan sebanyak 1 pada lintasan O – C – E – D – T.

Langkah 6 : alirkan sebanyak 1 pada lintasan O – C – E – T, dan hasilnya adalah :



Gambar 2.16 Langkah Solusi 6

Langkah 7 : Alirkan sebanyak 1 pada lintasan O – C – E – B – D – T, sehingga hasilnya:



Gambar 2.17 Hasil Akhir

Karena tidak ada lagi lintasan yang mempunyai kapasitas aliran positif, maka pola aliran terakhir merupakan pola aliran yang telah optimum. (Wilson, 1990).

Kesimpulannya adalah algoritma Djikstra dapat digunakan untuk menyelesaikan permasalahan rute terpendek dan aliran maksimum, elemen-elemen (bobot) dari rute tersebut berupa jarak tempuh, biaya, maupun hal lainnya.

2.8.4 Algoritma A Star

Dalam ilmu komputer, metode A* (A Star) adalah *graph search algorithm* yang mencari *path* (jalur) dari titik awal yang diberikan menuju titik tujuan. Algoritma A* pertama kali dijabarkan oleh Peter Hart, Nils Nilsson dan Bertram Raphael pada tahun 1968. (Wikipedia, 2006)

Metode A* adalah metode yang merupakan hasil pengembangan dari metode dasar *Best First Search*. Metode ini mengevaluasi setiap titik dengan mengkombinasikan dengan g(n), nilai untuk mencapai titik n dari titik awal, dan h(n), nilai perkiraan untuk mencapai tujuan dari titik n tersebut.

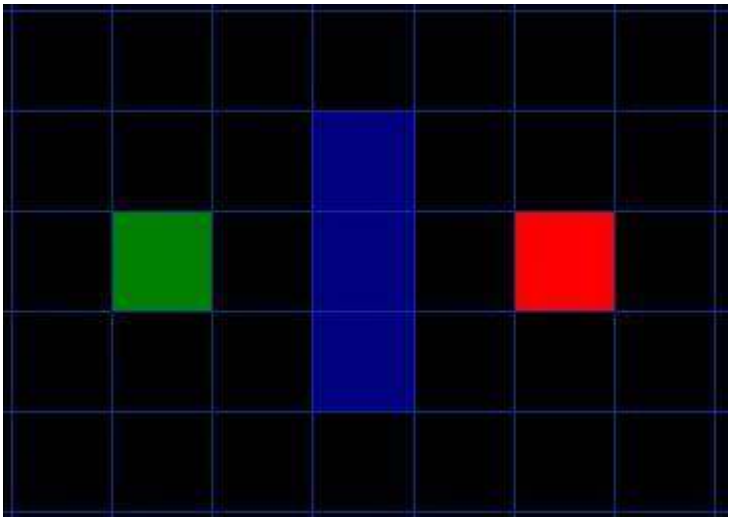
$$f(n) = g(n) + h(n)$$

Ketika g(n) memberikan hasil evaluasi nilai untuk mencapai titik n, dan h(n) memberikan nilai estimasi untuk mencapai tujuan dari titik n, maka didapatkan

f(n) = nilai estimasi yang terkecil yang melewati titik n

2.8.4.1 Cara Kerja Algoritma A Star (A*)

Menurut Patrick Lester (2005,p1) cara kerja algoritma A* dapat digambarkan sebagai berikut, misalkan seseorang ingin berjalan dari *node* A ke *node* B, dimana diantaranya terdapat hambatan, lihat gambar. Dimana *node* A ditunjukan dengan kotak berwarna hijau, sedangkan titik B ditunjukkan dengan kotak berwarna merah, dan kotak berwarna biru mewakili hambatan / tembok yang memisahkan kedua *node* tersebut.



Gambar 2.18 Tampilan Awal

Perlu diperhatikan bahwa, area pencarian dibagi ke dalam bentuk *node* seperti yang bisa dilihat pada gambar 2.18. Menyederhanakan area pencarian seperti yang telah dilakukan adalah langkah awal dalam pencarian jalur. Dengan fungsi ini dapat menyederhanakan

area pencarian menjadi *array* dua dimensi yang sederhana. Tiap nilai dalam *array* merepresentasikan satu *node* pada area pencarian, dan statusnya disimpan sebagai "yang bisa dilalui" atau "yang tidak bisa dilalui". Jalur ditemukan dengan menentukan *node* mana saja yang dilalui untuk mencapai *node* B dari *node* A. Ketika jalur ditemukan maka akan berpindah dari satu *node* ke *node* yang lain sampai ke tujuan.

Ketika area pencarian sudah disederhanakan ke dalam beberapa *node*. Seperti yang telah dilakukan diatas, langkah berikutnya adalah melakukan pencarian untuk mencari jalur terpendek. Dalam pencarian jalur A*, dimulai dari *node* A, memeriksa *node* yang berdekatan, dan secara umum mencari kesebelah sampai tujuan ditemukan.

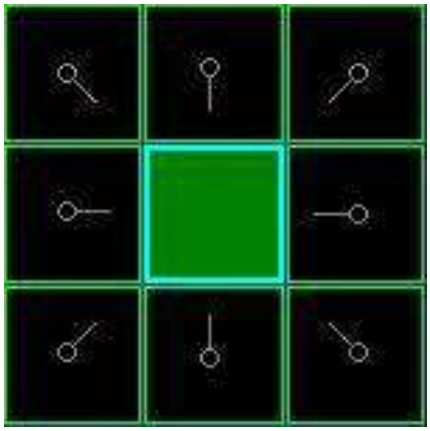
Pencarian dilakukan dengan tahap sebagai berikut :

1. Dimulai dari *start node* A dan *start node* tersebut ditambahkan ke sebuah *open list* dari *node-node* yang akan diperiksa. *List* tersebut berisi *node-node* yang mungkin dilalui pada jalur yang ingin dicari, atau mungkin juga tidak, jadi *list* tersebut berisi *node-node* yang perlu diperiksa.
2. Lihatlah semua *node-node* yang dapat dilalui yang terhubung dengan *start node*, hindari *node-node* yang merupakan penghalang-penghalang. Tambahkan ke dalam *open list*, untuk tiap-tiap *node*, *node* A merupakan *node parent*, *node* ini berguna ketika ingin mengikuti jalur.
3. Buang *node* A dari *open list*, kemudian tambahkan *node* A ke dalam *closed list*, dimana pada *list* ini tidak perlu lagi memeriksa *node-node* yang ada di dalamnya.

Pada saat ini, harus dilakukan seperti yang terlihat pada gambar, pada gambar dibawah *node* yang berwarna hijau di tengah-tengah adalah *start node*. *Node* yang sisinya berwarna biru adalah *node* yang telah dimasukkan ke dalam *closed list*, semua *node*

yang bersebelahan dengan *node* pusat yang akan diperiksa dimasukkan ke dalam *open list*, dan sisinya yang berwarna hijau.

Tiap petunjuk yang berwarna abu-abu menunjuk ke *node parent*-nya, yang merupakan *start node*.



Gambar 2.19 *Set Parent*

Selanjutnya, dipilih salah satu *node* yang berhubungan dalam *open list* lalu dilakukan berulang-ulang seperti langkah yang akan dijelaskan dibawah ini :

Persamaan untuk pemberian nilai pada *node* adalah f(n) = g(n) + h(n).

Dimana g(n) adalah nilai yang dibutuhkan untuk bergerak dari *start node* A ke sebuah *node* pada area tersebut, mengikuti jalur yang ditentukan untuk menuju kesana.

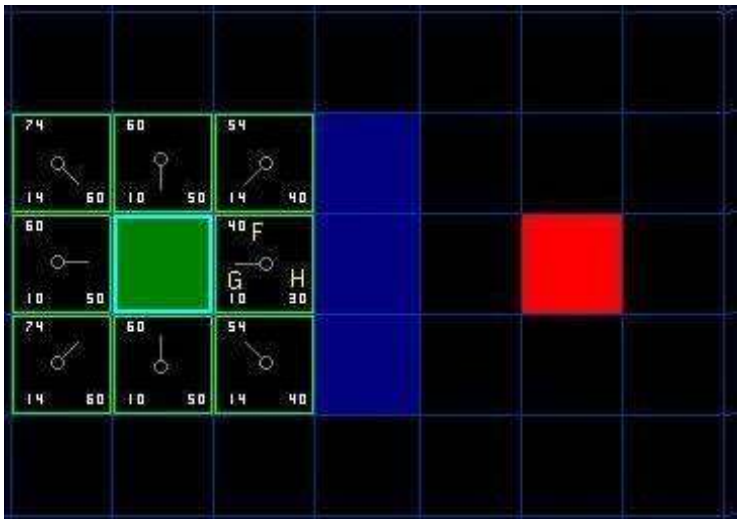
h(n) adalah nilai perkiraan untuk bergerak dari suatu *node* pada area ke final *node* B. *Node* yang dipilih untuk tujuan selanjutnya adalah *node* yang memiliki nilai f(n) terendah.

Jalur yang dibuat adalah jalur yang dibangun secara berulang-ulang dengan menentukan *node-node* yang mempunyai f(n) terendah pada *open list*. Seperti yang telah dikatakan diatas g(n) adalah nilai yang dibutuhkan untuk bergerak dari *start node*

ke final *node* dengan menggunakan jalur yang dibuat untuk ke sana. Akan diberi nilai 10 untuk tiap pergerakan *horizontal* atau *vertical*, dan nilai 14 untuk tiap pergerakkan diagonal. Nilai 10 dan 14 digunakan untuk penyederhanaan, di hindari perhitungan decimal dan pengakaran. Cara untuk menentukan nilai g(n) adalah dengan menghitung nilainya terhadap *node parent*-nya, dengan menambahkan 10 atau 14 tergantung apakah *node* tersebut diagonal atau orthogonal (non-diagonal) terhadap *parent node*. Fungsi ini diperlukan apabila didapatkan suatu *node* berjarak lebih dari satu *node* terhadap *start node*.

h(n) dapat diukur dengan berbagai macam cara. Cara yang digunakan adalah fungsi Manhattan, dimana dihitung jumlah total *node* yang bergerak *horizontal* atau *vertical* untuk mencapai final *node* dari *node* sekarang, dengan mengacuhkan pergerakkan diagonal. Lalu dikalikan dengan 10. Ini dinamakan fungsi Manhattan karena ini seperti menghitung jumlah blok-blok *node* dari satu tempat ke tempat lain, dimana tidak dapat memotong suatu blok secara diagonal. Yang penting ketika menghitung h(n), harus mengacuhkan rintangan apapun seperti tembok, air, dll. Ini adalah perhitungan perkiraan, bukan jarak nyatanya.

Lalu hitung nilai f(n) dengan menambahkan g(n) dan h(n). Hasilnya dapat dilihat pada gambar, dimana nilai f(n) ditulis di kiri atas, g(n) kiri bawah dan h(n) di kanan bawah pada tiap-tiap *node*.



Gambar 2.20 Masuk ke Close List

Diberi nilai g(n) sama dengan 10, pada *node* bagian atas, bawah, kiri dan kanan sedangkan *node-node* diagonal diberi nilai 14, karena *node-node* tersebut bersebelahan dengan *start node*.

Nilai h(n) ditentukan dengan menggunakan fungsi manhattan, dimana ditentukan jarak antara *node* tersebut dengan final *node* (merah), dengan bergerak hanya secara *horizontal* dan *vertical*.

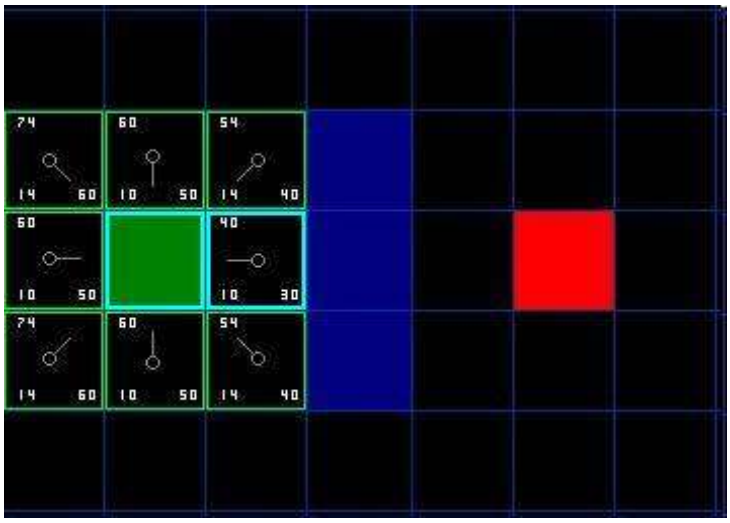
Untuk melanjutkan pencarian, dipilih *node* yang nilai f(n)-nya paling rendah dalam *open list*, ketika dipilih *node* tersebut lalu dilakukan :

1. Keluarkan *node* tersebut dari *open list* lalu dimasukkan ke *close list*.
2. Periksa semua *node* yang berhubungan. Kecuali *node* yang sudah masuk ke *close list* atau *node* yang tidak dapat dilalui (dinding, air, dan lain-lain). Tambahkan *node* tersebut ke *open list*, apabila *node* tersebut belum dimasukkan ke *open list* tersebut. Jadikan *node* yang dipilih tadi sebagai *parent node* bagi *node* baru tersebut.

3. Jika *node* yang terhubung sudah masuk ke *open list*, periksa apakah nilai g(n) *node* tersebut lebih kecil.
4. Jika tidak jangan lakukan apa-apa, jika benar *parent node* harus diganti lalu dihitung ulang nilai f(n) dan g(n).

Seperti contoh pada gambar 2.20, ada sembilan *node*, dimana 8 *node* masuk *open list*

dan *start node* sudah masuk *close list*, lalu *node* dengan nilai f(n) terendah yaitu 40, dimasukkan ke *close list*, karena itu diberi warna biru pada sisinya.



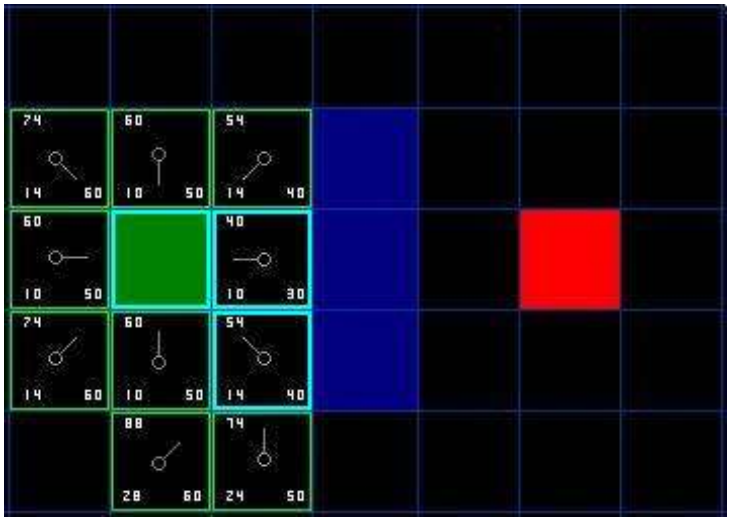
Gambar 2.21 Pemilihan *Close List*

Semua *node* yang berhubungan dengan *node* tesebut diperiksa, *start node* tidak dianggap karena sudah masuk ke *close list*, dan *node* hambatan. 4 *node* lain yang berhubungan semuanya sudah masuk ke *open list* maka harus diperiksa, apakah nilai g(n) yang dibutuhkan untuk mencapai *node* tersebut melalui *node* yang dipilih lebih kecil daripada menggunakan *node* lain, ternyata seperti contoh di atas didapatkan bahwa apabila ingin ke bawah atau ke atas dari *node* yang dipilih ternyata membutuhkan g(n)

sama dengan 20, sedangkan apabila diambil arah diagonal dari *start node* hanya membutuhkan g(n) sama dengan 14, maka tidak dilakukan apa-apa.

Jadi sekarang yang terdapat pada *open list* hanya tinggal 7 *node*, dicari lagi yang nilai f(n) terendah, ternyata ada 2 *node* yang punya nilai f(n) yang sama, itu tidak masalah, Dapat dipilih yang mana saja, tapi untuk mempercepat dapat dipilih yang terakhir masuk ke *open list*.

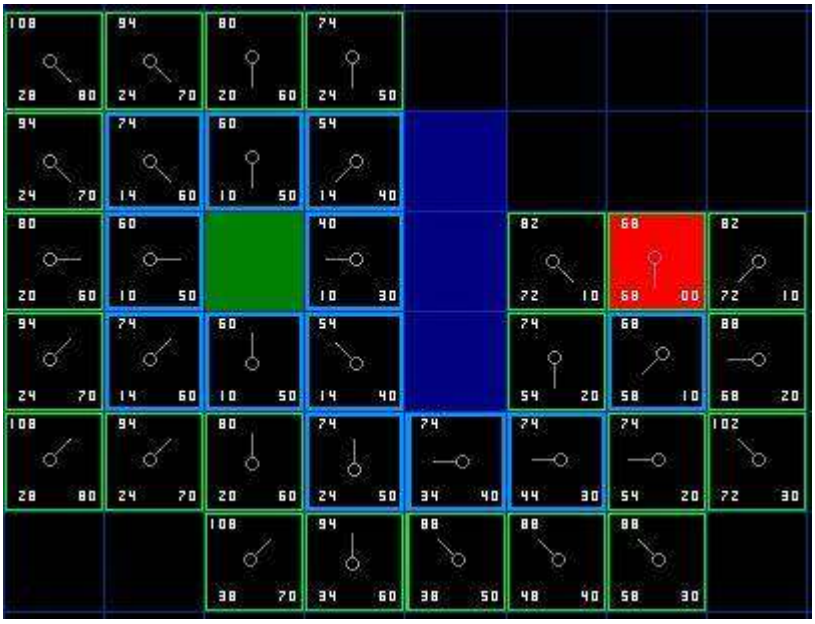
Jadi pilih yang bawah, maka akan tampak seperti gambar 2.22.



Gambar 2.22 Pemilihan *Close List* ke 2

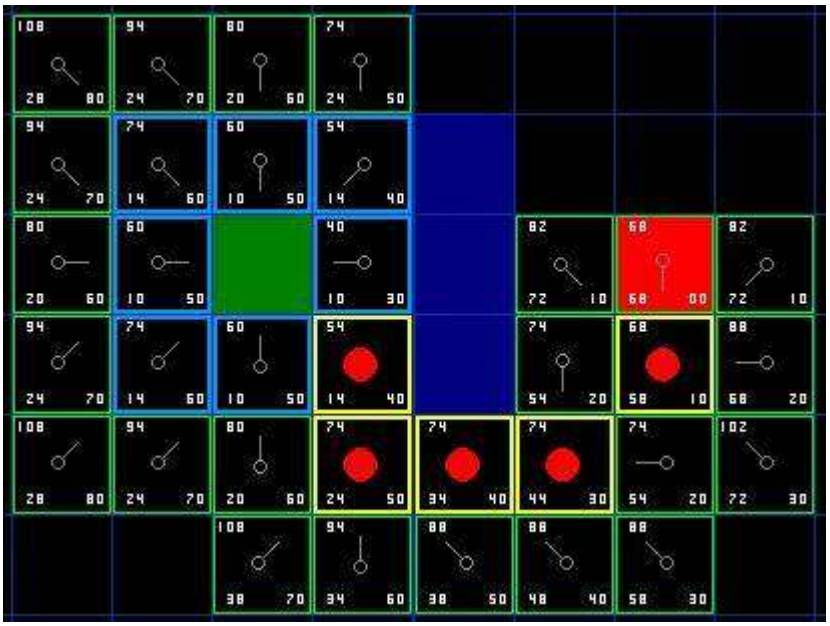
Kali ini periksa kembali *node* yang dipilih, masukkan *node* yang berhubungan ke dalam *open list* kecuali *node* yang merupakan penghalang, *node* yang sudah masuk *close list* dan *node* yang sudah masuk ke *open list*, tapi disini tidak dapat ditambahkan *node* di bawah dinding ke dalam *open list*, karena tidak dapat langsung dari *node* sekarang ke *node* tersebut tanpa memotong bagian pojok dari dinding di atasnya. Jadi harus turun dulu ke bawah (aturan untuk memotong sudut adalah pilihan).

Setelah itu dilakukan proses yang sama seperti yang diatas, lalu ditentukan *node* yang akan dipilih dengan membandingkan nilai f(n). Proses tersebut diulangi sampai final *node* ke dalam *open list* ditambahkan, terlihat pada gambar 2.23.



Gambar 2.23 *Final Node* Masuk *Close List*

Perhatikan pada *node* kedua dibawah *start node*, pada diagram awal nilai g(n) sama dengan 28 dan menunjuk pada *node* kanan atasnya sekarang bernilai g(n) sama dengan 20 dan menunjuk pada *node* di atasnya, hal ini terjadi karena pemeriksaan nilai g(n) dimana nilainya lebih rendah dengan menggunakan jalur yang baru, sehingga *parent node* harus diganti dan nilai g(n) dan f(n) harus dihitung ulang. Lalu setelah selesai ditandai dan proses telah diselesaikan maka ditentukan jalurnya dengan menggunakan fungsi *backtrack* dengan menelusuri dari final *node* mengikuti anak panah pada *node* tersebut hingga sampai ke *start node*. Hasilnya akan terlihat seperti gambar 2.24.



Gambar 2.24 ***Backtrack***

2.8.4.2 Pseudocode Algoritma A Star

Tentukan sebuah *node* yang berisi goal state – final *node*

Tentukan sebuah *node* yang berisi start state – start *node*

Letakkan start *node* pada open list

Selama open list tidak kosong

{

Set *node* terakhir di close list sebagai *current node*

Jika *current node* memiliki keadaan yang sama dengan final *node* maka solusi ditemukan, keluar dari while loop

Hasilkan setiap state successor *node* yang didapat dari *current node*

Untuk setiap successor *node* dari *current node*

{

Tentukan harga dari successor *node* menjadi harga dari *current*

node ditambah dengan jarak untuk mencapai successor *node* dari *current node*

Cari successor *node* dalam open list

Jika successor *node* ada dalam open list tetapi terdapat yang lebih baik maka buang successor *node* ini dan lanjutkan

Jika successor *node* terdapat di close list tetapi terdapat yang lebih baik maka buang successor *node* ini dan lanjutkan

Tentukan *current node* menjadi parent dari successor *node*

Tentukan h(n) sebagai jarak estimasi ke final *node* (menggunakan fungsi heuristic)

Tambahkan successor *node* ke dalam open list

}

Ambil *node* dari open list yang memiliki nilai f(n) terkecil

Tambahkan *current node* ke dalam close list

}

(Mario, Irawan, Aryo, 2004, p44)

2.9 Perbedaan Algoritma Djikstra dan Algoritma A Star

Menurut Adi Wijaya dan Roi Gunawan (2001, p372), kelebihan algoritma A Star dibandingkan dengan algoritma Djikstra adalah sebagai berikut:

1. Waktu pencarian algoritma A Star dalam menemukan rute lebih cepat dari algoritma Djikstra.
2. Jumlah loop A Star lebih sedikit dari anggota Djikstra.
3. Rute yang ditemukan berbeda tetapi mempunyai biaya yang sama.

4. Algoritma A Star lebih cocok untuk pengembangan game yang bersifat real time.

2.10 UML (Unified Modelling Language)

Menurut Yanti (2003), Unified Modelling Language (UML) adalah sebuah "bahasa" yg telah menjadi standar dalam industri untuk visualisasi, merancang dan mendokumentasikan sistem piranti lunak. UML menawarkan sebuah standar untuk merancang model sebuah sistem.

Dengan menggunakan UML dapat dibuat model untuk semua jenis aplikasi piranti lunak, dimana aplikasi tersebut dapat berjalan pada piranti keras, sistem operasi dan jaringan apapun, serta ditulis dalam bahasa pemrograman apapun. Tetapi karena UML juga menggunakan *class* dan *operation* dalam konsep dasarnya, maka lebih cocok untuk penulisan piranti lunak dalam bahasa-bahasa berorientasi objek seperti C++, Java, C# atau VB.NET. Walaupun demikian, UML tetap dapat digunakan untuk modeling aplikasi prosedural dalam VB atau C.

Jenis-jenis diagram UML :

1. Use Case Diagram

   Use Case Diagram berisi *actor* dan use case, yang memberikan gambaran mengenai hubungan yang terjadi antara dua hal tersebut. Use Case Diagram adalah titik permulaan dalam tahap analisa pada saat merancang suatu sistem. Diagram Ini dirancang oleh Ivan Jacobson.

   Use Case adalah dasar dari use case diagram, dimana semua use case dihubungkan dengan asosiasi dan akan mempunyai hubungan dengan aktor.

Tujuan dari Use case diagram ini adalah memberikan gambaran keseluruhan dari struktur *system* dan fitur yang ada kepada pihak non teknis seperti bagian manajemen dan pengguna. Diagram ini juga dapat digunakan untuk menggambarkan urutan kejadian (*event*) dalam *system*, apabila tidak ada kesalahan.

Use case diagram menggambarkan fungsionalitas yang diharapkan dari sebuah sistem. Yang ditekankan adalah "apa" yang diperbuat sistem, dan bukan "bagaimana". Sebuah *use case* merepresentasikan sebuah interaksi antara aktor dengan sistem. *Use case* merupakan sebuah pekerjaan tertentu, misalnya login ke sistem, meng-*create* sebuah daftar belanja, dan sebagainya. Seorang/sebuah aktor adalah sebuah entitas manusia atau mesin yang berinteraksi dengan sistem untuk melakukan pekerjaan-pekerjaan tertentu.

Use case diagram dapat sangat membantu penyusunan *requirement* sebuah sistem, mengkomunikasikan rancangan dengan klien, dan merancang *test case* untuk semua *feature* yang ada pada sistem.

Sebuah *use case* dapat meng-*include* fungsionalitas *use case* lain sebagai bagian dari proses dalam dirinya. Secara umum diasumsikan bahwa *use case* yang di-*include* akan dipanggil setiap kali *use case* yang meng-*include* dieksekusi secara normal.

Sebuah *use case* dapat di-*include* oleh lebih dari satu *use case* lain, sehingga duplikasi fungsionalitas dapat dihindari dengan cara menarik keluar fungsionalitas yang *common*. Sebuah *use case* juga dapat meng-*extend use case* lain dengan *behaviour*-nya sendiri. Sementara hubungan generalisasi antar *use*

case menunjukkan bahwa *use case* yang satu merupakan spesialisasi dari yang lain.

2. Activity Diagram

   Diagram ini digunakan untuk melakukan analisa terhadap perilaku yang ada dalam suatu use case yang kompleks dan memberikan gambaran interaksi antar use case tersebut.

   Activity diagram sangat mirip dengan statechart diagram karena menggambarkan aliran data, akan tetapi *activity diagram* digunakan untuk membuat model aliran kerja bisnis selama merancang use case.

   Diagram ini biasa digunakan untuk menggambarkan aktivitas bisnis yang rumit, sehingga membantu identifikasi use case atau interaksi diantara dan di dalam use case. Sedangkan statechart diagram memberikan gambaran perubahan status dalam sistem dan mencapai tahapan tertentu.

   Activity diagram menggambarkan berbagai alir aktivitas dalam sistem yang sedang dirancang, bagaimana masing-masing alir berawal, *decision* yang mungkin terjadi, dan bagaimana mereka berakhir. *Activity diagram* juga dapat menggambarkan proses paralel yang mungkin terjadi pada

   beberapa eksekusi. *Activity diagram* merupakan *state diagram* khusus, di mana sebagian besar *state* adalah *action* dan sebagian besar transisi di-*trigger* oleh selesainya *state* sebelumnya (*internal processing*). Oleh karena itu *activity diagram* tidak menggambarkan behaviour internal sebuah sistem (dan interaksi

   antar subsistem) secara eksak, tetapi lebih menggambarkan proses-proses dan jalur-jalur aktivitas dari *level* atas secara umum.

Sebuah aktivitas dapat direalisasikan oleh satu *use case* atau lebih. Aktivitas menggambarkan proses yang berjalan, sementara *use case* menggambarkan bagaimana aktor menggunakan sistem untuk melakukan aktivitas.

Sama seperti *state*, standar UML menggunakan segiempat dengan sudut membulat untuk menggambarkan aktivitas. *Decision* digunakan untuk menggambarkan *behaviour* pada kondisi tertentu. Untuk mengilustrasikan proses-proses paralel (*fork* dan *join*) digunakan titik sinkronisasi yang dapat berupa titik, garis horizontal atau vertikal.

Activity diagram dapat dibagi menjadi beberapa *object swimlane* untuk menggambarkan objek mana yang bertanggung jawab untuk aktivitas tertentu.

3. Sequence Diagram

Sequence Diagram ini digunakan untuk menggambarkan interaksi antara aktor dengan objek dan objek dengan objek lain. Pesan disampaikan dari aktor kepada objek, objek ke objek, dan dari objek ke aktor untuk menunjukkan aliran control dalam sistem. Diagram ini juga dapat digunakan untuk menggambarkan semua aliran yang mungkin terjadi dalam interaksi sistem, atau menggambarkan sebuah aliran saja.

Sequence diagram menggambarkan interaksi antar objek di dalam dan di sekitar sistem (termasuk pengguna, *display*, dan sebagainya) berupa *message* yang digambarkan terhadap waktu. *Sequence diagram* terdiri atar dimensi vertikal (waktu) dan dimensi horizontal (objek-objek yang terkait).

Sequence diagram biasa digunakan untuk menggambarkan skenario atau rangkaian langkah-langkah yang dilakukan sebagai respons dari sebuah *event* untuk menghasilkan *output* tertentu. Diawali dari apa yang men-*trigger* aktivitas

tersebut, proses dan perubahan apa saja yang terjadi secara internal dan *output* apa yang dihasilkan. Masing-masing objek, termasuk aktor, memiliki *lifeline* vertikal. *Message* digambarkan sebagai garis berpanah dari satu objek ke objek lainnya. Pada fase desain berikutnya, *message* akan dipetakan menjadi operasi/metoda dari *class*. *Activation bar* menunjukkan lamanya eksekusi sebuah proses, biasanya diawali dengan diterimanya sebuah message.

Untuk objek-objek yang memiliki sifat khusus, standar UML mendefinisikan *icon* khusus untuk objek *boundary, controller* dan *persistent entity*.

4. Collaboration Diagram

   Diagram ini digunakan untuk lebih memperjelas Class Diagram.

   Diagram ini merepresentasikan interaksi dan hubungan antara objek-objek yang diciptakan pada awal proses pemodelan awal. Diagram ini juga dapat digunakan untuk menggambarkan pesan antara objek yang berbeda.

   Collaboration diagram juga menggambarkan interaksi antar objek seperti *sequence diagram*, tetapi lebih menekankan pada peran masing-masing objek dan bukan pada waktu penyampaian *message*.

   Setiap *message* memiliki *sequence number*, di mana *message* dari *level* tertinggi memiliki nomor 1. Messages dari *level* yang sama memiliki prefiks yang sama.

5. Statechart Diagram

   Statechart Diagram digunakan untuk membuat model atas perilaku dari sub *system*, interaksi diantara class dan interface *system*, dan merealisasikan use case. Diagram ini digunakan pada saat perpindahaan antara tahapan analisa dan perancangan. Diagram ini merupakan cara terbaik untuk menggambarkan aliran dari aplikasi yang berjalan.

Statechart diagram menggambarkan transisi dan perubahan keadaan (dari satu *state* ke *state* lainnya) suatu objek pada sistem sebagai akibat dari *stimuli* yang diterima. Pada umumnya *statechart diagram* menggambarkan *class* tertentu (satu *class* dapat memiliki lebih dari satu *statechart diagram*).

Dalam UML, *state* digambarkan berbentuk segiempat dengan sudut membulat dan memiliki nama sesuai kondisinya saat itu. Transisi antar *state* umumnya memiliki kondisi *guard* yang merupakan syarat terjadinya transisi yang bersangkutan, dituliskan dalam kurung siku. *Action* yang dilakukan sebagai akibat dari *event* tertentu dituliskan dengan diawali garis miring.

Titik awal dan akhir digambarkan berbentuk lingkaran berwarna penuh dan berwarna setengah.

6. Class Diagram

Class Diagram digunakan untuk memberikan gambaran dari class yang ada, hubungan antar class, dan menjelaskan kedudukan class tersebut berada dalam sub *system* yang mana. Class diagram memiliki atribut, operasi, dan juga berbagai macam tipe peran dan asosiasi.

Class adalah sebuah spesifikasi yang jika diinstansiasi akan menghasilkan sebuah objek dan merupakan inti dari pengembangan dan desain berorientasi objek. *Class* menggambarkan keadaan (atribut/properti) suatu sistem, sekaligus menawarkan layanan untuk memanipulasi keadaan tersebut

(metoda/fungsi). *Class diagram* menggambarkan struktur dan deskripsi *class, package* dan objek beserta hubungan satu sama lain seperti *containment*, pewarisan, asosiasi, dan lain-lain.

Class memiliki tiga area pokok :

1. Nama

2. Atribut

3. Metoda

Atribut dan metoda dapat memiliki salah satu sifat berikut :

• *Private*, tidak dapat dipanggil dari luar *class* yang bersangkutan

• *Protected*, hanya dapat dipanggil oleh *class* yang bersangkutan dan anak-anak yang mewarisinya

• *Public*, dapat dipanggil oleh siapa saja

Hubungan Antar Class :

1. Asosiasi, yaitu hubungan statis antar *class*. Umumnya menggambarkan *class* yang memiliki atribut berupa *class* lain, atau *class* yang harus mengetahui eksistensi *class* lain. Panah *navigability* menunjukkan arah *query* antar *class*.

2. Agregasi, yaitu hubungan yang menyatakan bagian ("terdiri atas..").

3. Pewarisan, yaitu hubungan hirarkis antar *class*. *Class* dapat diturunkan dari *class* lain dan mewarisi semua atribut dan metoda *class* asalnya dan menambahkan fungsionalitas baru, sehingga ia disebut anak dari *class* yang diwarisinya. Kebalikan dari pewarisan adalah generalisasi.

4. Hubungan dinamis, yaitu rangkaian pesan (*message*) yang di-*passing* dari satu *class* kepada *class* lain. Hubungan dinamis dapat digambarkan dengan menggunakan *sequence diagram*.

7. Component Diagram

Component Diagram digunakan untuk memodelkan hubungan antara bagian-bagian dari *system* (termasuk didalamnya adalah source code, binary, file

eksekusi, scripts, atau file perintah) yang dikelompokan berdasarkan fungsionalitas atau lokasi dari file tersebut. Diagram ini digunakan untuk membantu pemahaman dimana fungsionalitas dari *system* yang ada dan versi mana dari paket software yang mengandung bagian dari fungsionalitas tersebut.

Component diagram menggambarkan struktur dan hubungan antar komponen piranti lunak, termasuk ketergantungan (*dependency*) di antaranya.

Komponen piranti lunak adalah modul berisi *code*, baik berisi *source code* maupun *binary code*, baik *library* maupun *executable*, baik yang muncul pada *compile time, link time*, maupun *run time*.

Umumnya komponen terbentuk dari beberapa *class* dan/atau *package*, tapi dapat juga dari komponen-komponen yang lebih kecil.

Komponen dapat juga berupa *interface*, yaitu kumpulan layanan yang disediakan sebuah komponen untuk komponen lain.

8. Deployment Diagram

Diagram ini digunakan untuk memberikan gambaran kepada pembaca dimana bagian-bagian dari perangkat lunak yang akan berada pada perangkat keras, dan bagaimana bagian-bagian dari perangkat keras tersebut saling berinteraksi satu dengan lainnya. Diagram ini juga dapat digunakan untuk mengetahui perangkat lunak apa yang terpasang dalam perangkat keras tertentu.

Deployment/physical diagram menggambarkan detail bagaimana komponen di-*deploy* dalam infrastruktur sistem, di mana komponen akan terletak (pada mesin, server atau piranti keras apa), bagaimana kemampuan jaringan pada lokasi tersebut, spesifikasi server, dan hal-hal lain yang bersifat fisikal. Sebuah *node* adalah server, *workstation*, atau piranti keras lain yang digunakan untuk men-

deploy komponen dalam lingkungan sebenarnya. Hubungan antar *node* (misalnya TCP/IP) dan *requirement* dapat juga didefinisikan dalam diagram ini.

2.11 .NET Framework

Microsoft .NET Framework adalah sebuah komponen dari sistem operasi Microsoft Windows. .NET Framework menyediakan sejumlah besar solusi - solusi yang telah dikode sebelumnya untuk persyaratan - persyaratan umum program, dan mengatur eksekusi program - program yang ditulis secara khusus untuk framework. .NET Framework adalah kunci penawaran utama dari Microsoft, dan dimaksudkan untuk digunakan oleh sebagian besar aplikasi - aplikasi baru yang dibuat untuk platform Windows.

Program - program yang ditulis untuk .NET framework dijalankan pada suatu lingkungan software yang mengatur persyaratan - persyaratan runtime program. *Runtime environment* ini, yang juga merupakan suatu bagian dari .NET framework, dikenal sebagai *Common Language Runtime* (CLR). CLR menyediakan penampilan dari *application virtual machine*, sehingga para programmer tidak perlu mengetahui kemampuan CPU tertentu yang akan menjalankan program. CLR juga menyediakan layanan - layanan penting lainnya seperti jaminan keamanan, pengaturan memori, dan *exception handling* / penanganan kesalahan pada saat runtime.

(Wikipedia, 2006)

.NET Platform merupakan satu set kumpulan teknologi yang memungkinkan teknologi Internet ditransformasikan ke dalam platform *distributed computing* dengan skalabilitas dan kompatibilitas tinggi. Secara teknikal, .NET Platform menyediakan

konsep pemrograman dengan library dan modul-modul baru yang konsisten, terlepas dari jenis bahasa pemrograman yang digunakan.

.NET Platform menyediakan hal-hal berikut bagi para developer :

1) Language independent, dengan programming model yang konsisten di semua tier aplikasi yang dibangun.
2) Interoperability dan kompatibilitas antar aplikasi.
3) Kemudahan migrasi dari teknologi yang ada saat ini.
4) Dukungan penuh terhadap berbagai teknologi standar yang digunakan dalam platform internet, antara lain HTTP, XML, SOAP dan HTML.

Teknologi inti .NET secara umum terdiri dari 4 area pokok :

1) .NET Framework

   .NET Framework adalah teknologi inti yang menyediakan berbagai library untuk digunakan oleh aplikasi di atasnya. Komponen inti .NET Framework adalah Common Language Runtime (CLR) yang menyediakan run time environment untuk aplikasi yang dibangun menggunakan Visual Studio .NET, terlepas dari jenis bahasa pemrogramannya. Dengan adanya CLR tersebut, *programmer* dapat memanfaatkan *consistent object model* dalam mengakses berbagai komponen *library*.

2) .NET Building Block Services

   Building block merupakan sekumpulan services yang bersifat programmable, yang dapat diakses secara offline maupun online. *Service* tersebut merupakan modul-modul yang terdapat di suatu komputer, *server* dalam jaringan, maupun di suatu

server di internet. Service ini merupakan suatu idealisasi di masa depan, dimana sebuah aplikasi bersifat terdistribusi dengan modul-modul yang tersimpan di berbagai tempat, tetapi dapat diintegrasikan membentuk suatu aplikasi. Konsep ini merupakan arah pengembangan *subscription based software*, yang saat ini mulai banyak berkembang dan dikenal sebagai *Application Service Provider*.

Service tersebut dapat diakses oleh berbagai platform, asalkan *platform* tersebut men*support* protokol SOAP, yang merupakan protokol standar dalam mengakses *web service*. Peranan XML sebagai media definisi data menjadi sangat penting dalam hal ini, dan XML juga menjadi pusat perubahan besar dalam platform .NET.

3) Visual Studio .NET

Visual Studio .NET menyediakan *tools* bagi para *developer* untuk membangun aplikasi yang berjalan di .Net Framework. VS.Net membawa perubahan besar dalam gaya pemrograman, karena setiap *programmer* dituntut untuk memahami .NET object model dan *Object Oriented Programming* dengan baik, jika tidak ingin menghasilkan aplikasi dengan performa rendah.

VS.Net juga semakin mempertipis jarak antara *Windows Programmer* dengan *Web Programmer*. Dunia *scripting* yang akrab bagi *programmer web* akan sulit ditemukan dalam .NET, karena pemrograman *web* sudah bersifat *full object oriented*, dengan fasilitas *event driven programming* sebagaimana layaknya *windows programming*.

4) .Net Enterprise Server

Bagian ini merupakan sekumpulan *server based technology* yang digunakan untuk mendukung teknologi .NET, yang mencakup sistem operasi, *database*, *messaging*, maupun manajemen e-commerce. Teknologi yang disediakan antara lain adalah Windows 2000 Server, SQL Server, Exchange, ISA Server dan BiZTalk Server.

(M. Choirul Amri, 2003)

2.12 SQL (Structured Query Language)

SQL adalah sebuah bahasa yang berisi perintah-perintah untuk memanipulasi database seperti menghapus, merubah, memilih,menggabungkan data.

Database merupakan hal yang tidak asing lagi pada saat ini di dunia di mana hampir semua komponen didunia saling berhubungan satu sama lain dan saling bertukar informasi antar satu sama lain.

Oleh karena informasi yang dimiliki manusia beragam maka diperlukan pengaturan menurut aturan tertentu yang kemudian dikenal dengan database. Secara sederhana database dapat diartikan kumpulan dari data-data atau lebih lengkap lagi kumpulan dari data-data yang terintegrasi dan diatur menurut aturan (field) tertentu.

Kemunculan komputer sangat bermanfaat sebagai media pembantu yang dapat meningkatkan efiensi dari penggunaan data-data dari database tersebut melalui program yang dibuat untuk mengatur data-data tersebut. Program tersebut dikenal dengan *database management system* (DBMS) yang digunakan untuk pengaturan database.

Dan seiring perkembangan database yang semakin kompleks maka pada tahun 1970 E.F Codd memperkenalkan database model relasional dalam sebuah artikel yang berjudul “A Relational Model of Data for Large Shared Databanks” (Sebuah model data

relasional untuk Bank data yang besar dan terintegrasi) di mana melalui artikel ini lahirlah sebuah konsep dasar untuk pengembangan sebuah bahasa database.

Kemudian pada tahun 1979 Codd memperkenalkan idenya tersebut dalam konsep yang lebih nyata sehingga kemudian dapat dikembangkan sebagai sebuah *bahasa database relasional*. Dan salah satu bahasa *database relational* itu adalah S.Q.L (Structured Query Language).

Kelahiran SQL juga memiliki hubungan erat dengan dengan sebuah proyek IBM yang dikenal dengan *Sistem R*. Di mana proyek ini bertujuan untuk mengembangkan sebuah sistem pada *database relasional* atau dengan kata lain sebuah sistem yang dapat memenuhi segala jenis sistem pengoperasian database modern. Dengan memperkenalkan sebuah bahasa yang dinamakan Sequel yang pada perkembangannya berubah menjadi SQL (Structured Query Language).

Istilah-istilah penting dalam model relasional :

- Tabel, disebut relasi, merupakan satu-satunya bentuk penyimpanan data dalam *database relational* yang terdiri dari beberapa kolom dan beberapa baris

  Dua sifat yang dimiliki tabel :

  Perpotongan baris dan kolom hanya berisi satu yang dinamakan nilai atomik (yaitu tidak dapat dibagi lagi). Baris-baris didalam tabel mempunyai urutan secara khusus.
- Kolom, disebut atribut dan dalam tabel relasional menunjukan hubungan antar field dari suatu *record*.
- Baris, disebut tupel dan dalam tabel relasional menunjukan hubungan antar *record* dalam suatu berkas data.

- Domain, disebut juga entitas adalah kumpulan nilai yang valid untuk satu atau lebih atribut.
- Kunci utama (*primary key*), yaitu satu kolom/beberapa kolom yang secara unik mengidentifikasikan sebuah baris di dalam tabel.
- Kunci kandidat (*candidate key*), yaitu kolom didalam tabel yang biasanya mempunyai nilai unik
- Kunci asing (*foreign key*) yaitu atribut dengan domain yang sama yang menjadi kunci utama pada sebuah relasi tetapi pada relasi yang lain atribut tsb hanya sebagai atribut biasa

SQL digunakan dengan cara :

1. Secara interpretasi (*Interactive SQL*) yakni dengan memasukkan sebuah pernyataan SQL melalui terminal atau mikrokomputerdan langsung diproses atau diintrepretasikan. Hasilnya bisa langsung dilihat.
2. Secara sisip (*Embedded SQL*) yaitu dengan menyisipkan pernyataan SQL kedalam sebuah program yang ditulis dengan bahasa pemrogaman lain. Hasilnya tidak dapat dilihat secara langsung, tetapi diproses oleh program yang memakainya.

SQL-Server 2000 adalah RDBMS yang menggunakan bahasa Transact-SQL (T-SQL) untuk menerima request dari SQL-Client, atau dari SQL-Server 2000 lainnya.

Microsoft - Structured Query Language merupakan kepanjangan dari MS-SQL adalah database berbasis relasional, artinya struktur data diatur melalui pembuatan tabel-tabel yang satu dengan yang lainnya mempunyai keterkaitan atau relasi.

Tiga elemen penting yang merupakan model fundamental dari relasi adalah :

1. Struktur Data : Tabel

Tabel terdiri atas baris (*record* atau *row*) dan setiap baris terdiri dari atas kolom – kolom (*column* atau *field*) yang terdefinisi melalui tipe data pada kolom tersebut.

2. Integritas Data

   Mempunyai arti bahwa data sesuai dengan kondisi ”real”, misalnya sebuah field ”umur”, maka nilai yang terjadi tidak boleh negatif, karena memang tidak ada umur yang negatif.

   Kesesuaian data dengan nilai real ini disebut juga sebagai ”batasan nilai untuk integritas data” atau ”integrity constraints”.

3. Manipulasi Data

   Data yang tersimpan dapat dimanipulasi melalui bahasa pemograman terstruktur seperti SQL.

SQL Server terdiri dari dua bagian besar yaitu :

1. Data Definition Language (DDL)

   DDL terdiri dari Create Table, Alter Table dan Drop Table.

2. Data Manipulation Language (DML)

   DML terdiri dari Select, Insert, Update, dan Delete.

Macam – macam tipe data dalam SQL:

Data Type	**Description**
Integer(size) int(size) smallint(size) tinyint(size)	Tipe data untuk angka.

decimal(size,d) numeric(size,d)	Untuk angka pecahan (mengandung koma). Size ditulis berapa banyak angka yang bisa ditampung, sudah termasuk angka dibelakang koma. D berupa banyaknya angka di belakang koma. Contoh : decimal(5,2)
char(size)	Memuat karakter dengan jumlah yang sudah pasti (dapat berupa huruf, angka, dan spesial karakter). Contoh : char(20)
varchar(size)	Memuat karakter tetapi jumlah yang terpakai sesuai dengan banyaknya karakter yang diisi(dapat berupa huruf, angka, dan spesial karakter). Contoh : char(20)
Money	Tipe data untuk Uang
datetime(yyyymmdd)	Tipe data untuk tanggal dan waktu

Tabel 2.2 Tipe Data SQL

(Anonymous, 2005)