BAB II

TELAAH LITERATUR

2.1 Algoritma

Sebelum angka desimal dikenal luas, orang-orang menggunakan angka romawi untuk menuliskan angka. Sistem desimal ditemukan sekitar tahun 600 di India di mana penulisan angka dan aritmatika dapat dilakukan secara cepat dan efisien. Al Khwarizmi menemukan metode dasar untuk melakukan penjumlahan, pengurangan, perkalian, pembagian, dan akar. Prosedur yang dilakukan presisi, tidak ambigu, efisien dan tepat, dan merupakan akar dari algoritma (Dasgupta, dkk., 2006: 9).

Semenjak itu, sistem desimal dan algoritma numerik berperan penting dalam perkembangan jaman. Kedua sistem tersebut digunakan dalam sains dan teknologi. Dan setelah komputer ditemukan, konsep tersebut digunakan untuk *positional system* pada *bits, words,* dan *arithmetic unit* (Dasgupta, dkk., 2006: 10).

Secara umum, algoritma didefinisikan sebagai urutan langkah-langkah logis penyelesaian masalah yang disusun secara sistematis. Suatu urutan langkah logis untuk menyelesaikan masalah dapat dikatakan sebagai sebuah algoritma apabila memenuhi syarat-syarat berikut ini.

1. Algoritma harus berhenti setelah mengerjakan sebuah proses,
2. setiap langkah harus didefinisikan secara tepat dan tidak berarti dua (ambiguitas),
3. algoritma memiliki nol atau lebih masukan,

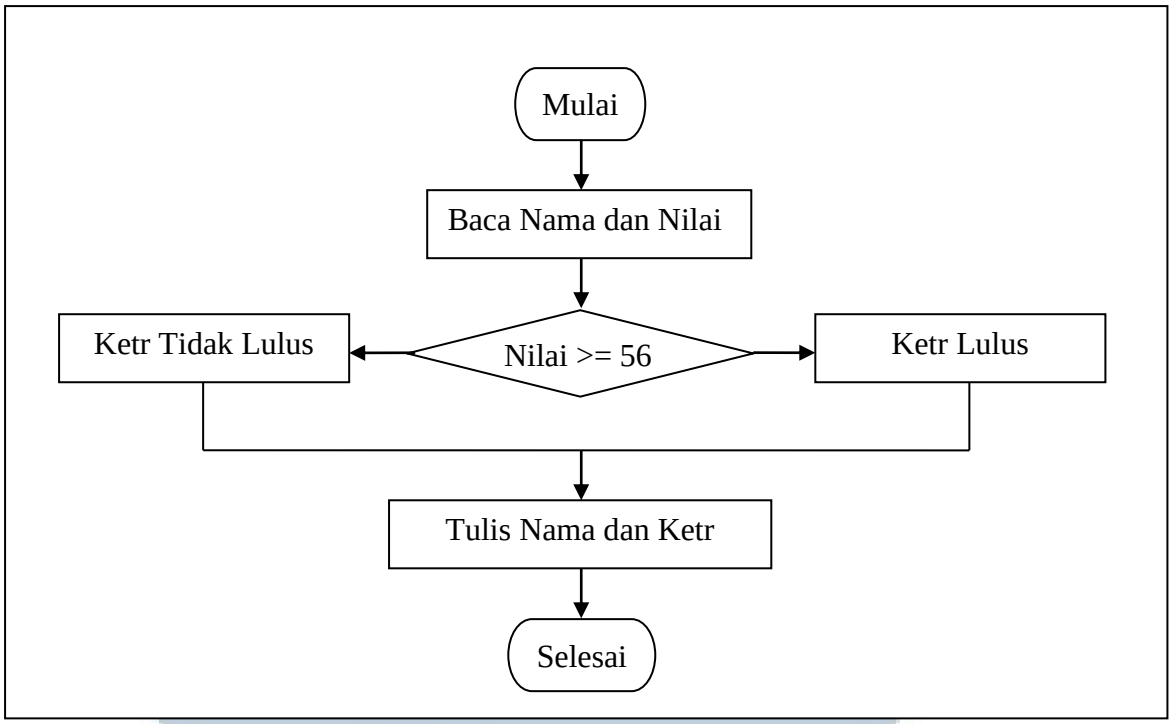
4. algoritma memiliki satu atau lebih keluaran, dan
5. algoritma harus efektif,yaitu setiap langkah dapat dikerjakan dalam waktu yang masuk akal (Knuth, 1997: 4).

Terdapat tiga jenis proses dalam algoritma yaitu *sequence, selection,* dan *repetition*.*Sequence* merupakan urutan dari langkah-langkah yang dilakukan.*Selection* adalah proses pemilihan langkah selanjutnya berdasarkan kondisi yang ada. Sedangkan *repetition* adalah proses yang dilakukan berulang-ulang hingga mencapai kondisi tertentu.

Algoritma dapat dituliskan dalam dua cara, yaitu menggunakan *pseudocode* dan *flowchart*.*Pseudocode* menggunakan instruksi berupa kata, dan *flowchart* menggunakan instruksi berupa gambar atau simbol. Berikut ini adalah contoh *pseudocode*dan *flowchart.*

```
Baca Nama, Nilai
Jika Nilai >= 56 maka
   Ketr ← Lulus
Selain itu
   Ketr ← Tidak Lulus
Tulis Nama dan Ketr
```

Gambar 2.1*Pseudocode*



Gambar 2.2*Flowchart*

2.1.1 Big-O

Running time dari suatu algoritma dapat dianalisis. Kesalahan dalam analisis dapat disebabkan analisis yang tidak akurat ataupun analisis yang terlalu akurat. Untuk mendapatkan hasil analisis yang tepat, maka digunakan metode simplifikasi. Penghitungan *running time* dilakukan dengan menghitung jumlah *basic computer steps.*

Simplifikasi dalam penghitungan dilakukan dengan menyederhanakan jumlah *steps* dari algoritma. Pada algoritma yang memiliki $5n^3 + 4n + 3$*steps,* 4n dan 3 dapat dihilangkan sebab 4n dan 3 akan menjadi tidak signifikan seiring dengan pertumbuhan n. Bahkan 5 pada $5n^3$ dapat dihilangkan mengingat bahwa komputer

akan lima kali lebih cepat hanya dalam beberapa tahun, sehingga dapat dikatakan bahwa algoritma tersebut membutuhkan waktu $O(n^3)$.

Big-O *notation*fokus pada *big picture*, sehingga akan berusaha membuat simplifikasi sesederhana mungkin. Berikut ini merupakan beberapa aturan yang dapat digunakan dalam proses simplifikasi:

1. *Multiplicative constants can be omitted: $14n^2$ becomes n^2.*
2. *n^a dominates n^b if $a > b$: for instance, n^2 dominates n.*
3. *Any exponential dominates any polynomial: 3^n dominates n^5 (it even dominates 2^n).*
4. *Likewise, any polynomial dominates any logarithm: n dominates $(\log n)^3$. This also means, for example, that n^2 dominates n log n*(Dasgupta, dkk., 2006: 14).

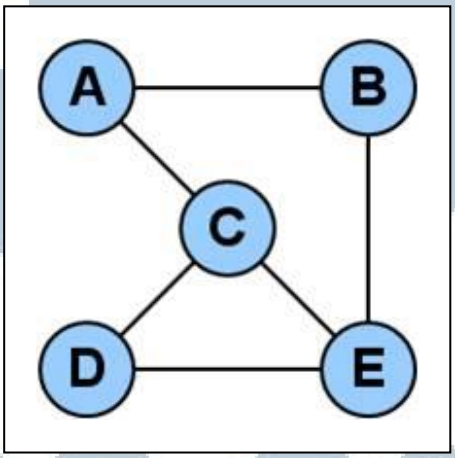
2.2 Graph

*Graph*didefinisikan sebagai "*set of nodes or states and a set of arcs that connect the nodes"*(Luger, 2009: 82). Selanjutnya, ia menyatakan,

A labeled graph has one or more descriptors (labels) attached to each node that distunguish that node from any other node in the graph. In a state space graph, these despcriptors identify states in a problem-solving process.

Definisi graph yang lebih rinci dikemukakan yaitu *"a set of vertices plus a set of edges that connect pairs of distinct vertices (with at most one edge connecting any pair of vertices)"*(Sedgewick, 2002: 7). Dalam pembahasan tentang *graph*, ahli matematika menggunakan istilah *vertex* dan *node* bergantian, namun dalam

algoritma, *vertex* digunakan saat berbicara mengenai *graph* dan *node* saat membicarakan representasi, contohnya adalah struktur data pada C. Demikian pula dengan istilah *arc*, *edge*, dan *link*, ketiga istilah tersebut digunakan untuk mendeskripsikan koneksi antara dua *vertex*. Namun dalam algoritma, *edge* digunakan saat membicarakan *graph* dan *link* digunakan saat membicarakan struktur data pada C (Sedgewick, 2002: 8).

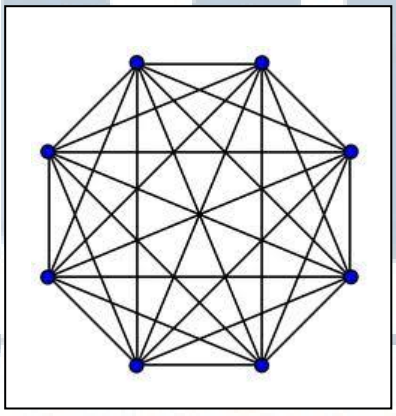


Gambar 2.3*Graph*

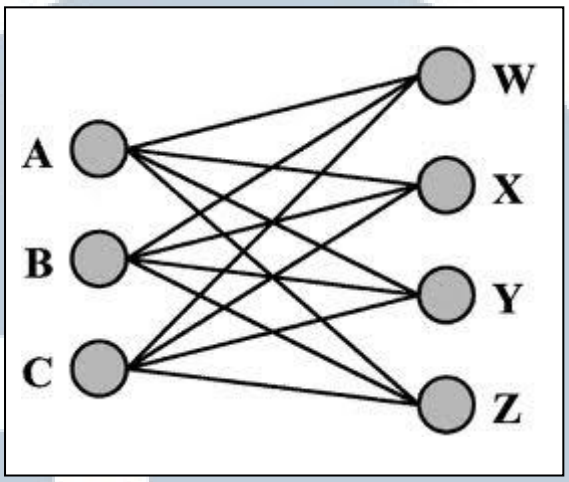
Robert Sedgewick menjelaskan beberapa istilah dalam *graph* yang perlu diketahui. Saat sebuah *edge* terhubung dengan dua *vertex*, dapat dikatakan bahwa kedua *vertex*tersebut *adjacent* terhadap satu sama lain dan *edge* tersebut *incident* terhadap kedua *vertex*. *Degree* dari *vertex* adalah jumlah *edge* yang *incident* terhadapnya. Notasi v-w merepresentasikan *edge* yang terkoneksi dengan v dan w. Sebuah *graph* dapat memiliki jumlah maksimum *edge* V(V-1) /2 di mana V adalah jumlah *vertex* (Sedgewick, 2002: 9).

Salah satu hal yang sering dilakukan dalam implementasi *graph* adalah mencari *path*. Robert Sedgewick mendefinisikan *path* dalam *graph* sebagai "*a sequence of vertices in which each successive vertex (after the first) is adjacent to its predecessor in the path"* (Sedgewick, 2002: 10). *Cyclic path* merujuk pada *path* yang memiliki *start* dan *final vertex* yang sama, sedangkan *acyclic path* digunakan untuk *path* yang memiliki *start* dan *finalvertex* yang berbeda. Selanjutnya, ia (Sedgewick, 2002 : 11) mengatakan "*a graph is a connected graph if there is a path from every vertex to every other vertex in the graph".* Sebuah *acyclic connected graph* disebut sebagai *tree*.

Graph memiliki beberapa karakteristik. Sebuah *graph* di mana semua *vertex*nya saling berhubungan disebut dengan *complete graph,* sedangkan *bipartite graph* adalah *graph* di mana *vertex*nya dapat dibagi menjadi dua set di mana semua *edge* terhubung dengan *vertex* pada set pertama dan *vertex* pada set kedua (Sedgewick, 2002: 12).

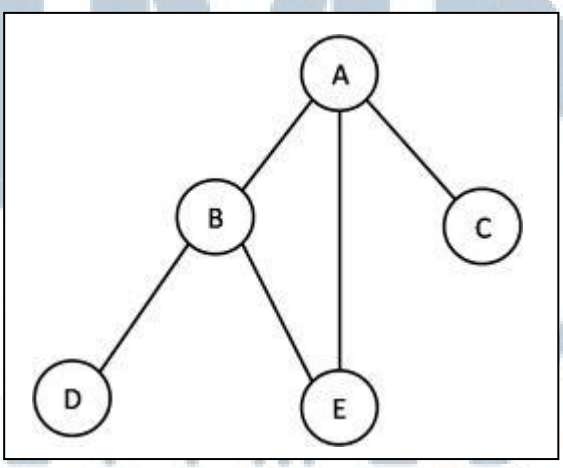


Gambar 2.4*Complete Graph*

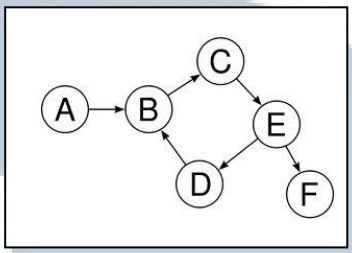


Gambar 2.5*Bipartite Graph*

Selain *complete* dan *bipartite, graph* dapat dibedakan menjadi *undirected graphs* dan *directed graphs* atau *digraphs. Undirected graphs* terdiri dari *edge* yang tidak memiliki arah sehingga *edge* v-w sama dengan *edge* w-v. Sedangkan pada*digraphs*, setiap *edge* hanya memiliki satu arah di mana *vertex* pertama pada *directed edge* disebut *source* dan *vertex* kedua disebut *destination.Directed edge* digambarkan dengan panah dari *source* ke *destination* (Sedgewick, 2002: 14).

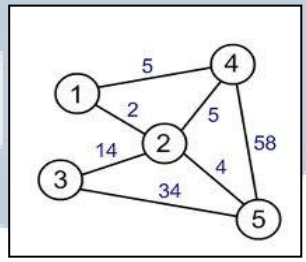


Gambar 2.6*Undirected Graph*



Gambar 2.7*Directed Graph*

Berdasarkan informasi yang dimiliki, *graph* dibagi dua yaitu *unweighted graph* dan *weighted graph.* Pada *weighted graph,* terdapat asosiasi nomor (*weight)* untuk tiap *edge,* yang secara umum merepresentasikan *distance* atau *cost.*



Gambar 2.8*Weighted Graph*

Pada umumnya *graph*memiliki lebih dari satu *edge.* Banyak atau tidaknya *edge* dari suatu *graph* disebut dengan *density*. Terdapat dua jenis *density* dari *graph* yaitu *sparse* dan *dense. Dense graph* merupakan dengan jumlah *edge* mendekati jumlah maksimal dari *edge* yaitu V(V-1) /2 (Cormen, dkk., 2001: 629).Sebaliknya, *sparse graph* merupakan *graph* dengan jumlah *edge* yang sedikit di mana $|E| < |V|^2$(Cormen, dkk., 2001: 527).

2.2.1 Shortest Path

Setiap *path* pada *weighted digraph* memiliki *path weight* yaitu nilai dari total *weights* milik *edge* dari *path* tersebut. Hal ini menimbulkan formulasi untuk

permasalahan baru seperti mencari *lowest-weight path* di antara dua *vertex* yang ditentukan yang disebut *shortest path problem* (Sedgewick, 2002: 265).

Sedgewick (2002: 269) membagi *shortest path problem* menjadi tiga jenis:

1. *All-pairs shortest path,* di mana pencarian dilakukan untuk jalur terpendek antara setiap *vertex* pada *graph,*
2. *single-source shortest path,* di mana pencarian dilakukan untuk jalur terpendek dari suatu *vertex* awal ke semua *vertex* lainnya, dan
3. *source-sink shortest path,* di mana pencarian dilakukan untuk mencari jalur terpendek antara suatu *vertex* awal dan suatu *vertex* tujuan.

Sedangkan Anany Levitin membagi *shortest path problem* menjadi:

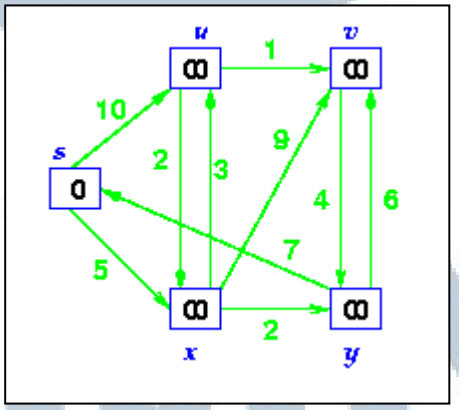
1. *All-pairs shortest-paths problem* yaitu mencari jarak (panjang dari *shortest paths)* dari tiap *vertex* ke semua *vertex* lain (Levitin, 2007: 288),
2. *single-source shortest-paths problem* di mana pencarian dilakukan untuk sebuah *vertex source* pada sebuah *weighted connected graph* ke semua *vertex* lainnya (Levitin, 2007: 323),
3. *minimum-edge shortest-paths problem,* yaitu mencari *path* dengan jumlah *edge* paling sedikit di antara dua *vertex* (Levitin, 2007: 169),
4. *single-destination shortest-paths problem,* yaitu mencari *shortest path* yang menuju sebuah *vertex* dari semua *vertex lainnya* dalam sebuah *weighted graph* atau *digraph* (Levitin, 2007: 327), dan
5. *single-pair shortest-paths problem,* yaitu mencari *shortest path*antara dua buah *vertex* yang telah ditentukan dalam sebuah *weighted graph* atau *digraph* (Levitin, 2007: 327).

2.3 Dijkstra

Algoritma Dijkstra merupakan algoritma terbaik yang dapat menyelesaikan *single-source shortest-paths problem.* Algoritma ini hanya dapat diaplikasikan pada *graph* dengan *non negative weight.* Sebagian besar aplikasi tidak membutuhkan *negative weight* sehingga limitasi dari Dijkstra tidak mengurangi popularitas dari algoritma ini (Levitin, 2007: 323).

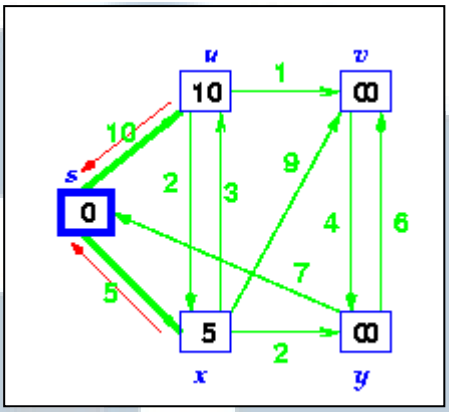
Pertama-tama, algoritma ini akan mencari *shortest path* dari *source* ke *vertex* terdekat, setelah itu ke *vertex* terdekat kedua, dan seterusnya. Berikut ini merupakan ilustrasi *step by step* bagaimana algoritma Dijkstra bekerja.

1. Diberikan graph G = (V, E). Semua *vertex* memiliki *infinite cost* kecuali *source vertex*, s, yang bernilai 0.



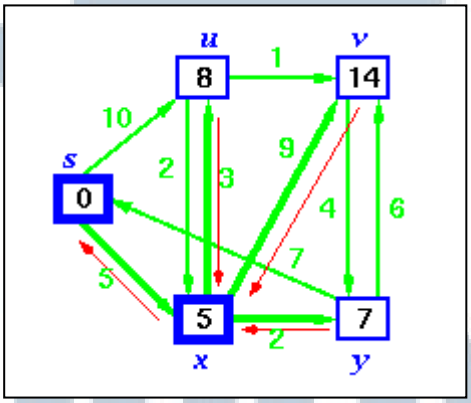
Gambar 2.9 Langkah Pertama Dijkstra

2. Pertama-tama, sebuah *vertex* dipilih, di mana *vertex* tersebut terletak paling dekat dengan *source vertex* s. Inisialisasi *distance* s, d[s] = 0 dan tambahkan terhadap S. Selanjutnya, tentukan *vertex* yang *adjacent* terhadap s. Setelah itu, *update predecessor* (lihat panah merah pada gambar 2.10).



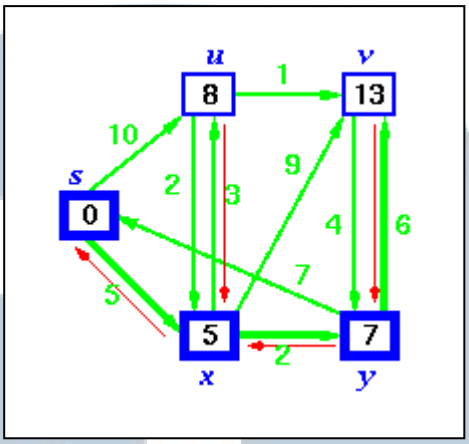
Gambar 2.10 Langkah Kedua Dijkstra

3. Pilih *vertex* terdekat yaitu x. Tentukan semua *vertex* yang *adjacent* terhadap x. *Update predecessors* untuk *vertex* u, v, dan y (panah merah).



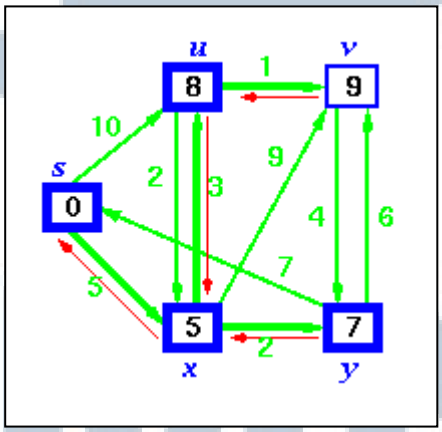
Gambar 2.11 Langkah Ketiga Dijkstra

4. Sekarang, *vertex* y merupakan *vertex* terdekat dengan s, sehingga tambahkan ke S. Tentukan semua *vertex* yang *adjacent* terhadap y dan tambahkan *predecessors* (panah merah).



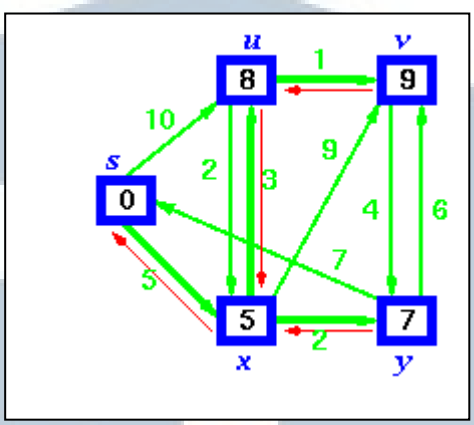
Gambar 2.12 Langkah Keempat Dijkstra

5. Sekarang, *vertex* u merupakan yang terdekat. Pilih *vertex* ini dan tambahkan *vertex v* sebagai *predecessors.*



Gambar 2.13 Langkah Kelima Dijkstra

6. Terakhir, tambahkan *vertex v.Predecessor list* sekarang mendefinisikan *shortest* path dari tiap *vertex* ke *source vertex*, s.



Gambar 2.14Langkah Keenam Dijkstra

Levitin (Levitin, 2007: 323) memberikan *pseudocode* untuk algoritma Dijkstra yaitu sebagai berikut.

```
//Input: A weighted connected graph G = {V, E} with nonnegative
//weights and its vertex s
//Output: The length dv of a shortest path from s to v
//and its penultimate vertex pv for every vertex v in V
Initialize (Q)    //initialize vertex priority queue to empty
for every vertex v in V do
            dv ← ∞; pv ← null
            Insert(Q, v, dv)   //initialize vertex priority
ds ← 0; Decrease(Q, s, ds)     //update priority if s with ds
VT ← {}
for i ← 0 to |V| - 1 do
            u* ← DeleteMin (Q)      //delete the minimum priority
            VT ← VT ∪ {u*}

            for every vertex u in V – VT that is adjacent to u* do
                  if du* + w (u*, u) < du
                        du ← du* + w (u*, u); pu ← u*
                        Decrease (Q, u, du)
```

Gambar 2.15*Pseudocode* Dijkstra

Levitin mengatakan bahwa *time efficiency* dari algoritma Dijkstra tergantung dari struktur data yang digunakan untuk mengimplementasi *priority queue* dan untuk merepresentasikan *input graph*(Levitin, 2007: 326). Untuk *graph* yang

direpresentasikan dengan *weight matrix* dan *priority queue* diimplementasikan dengan *unordered array, time efficiency*-nya adalah $O(|V|^2)$, sedangkan *graph* yang menggunakan *adjacency list* dan *priorityqueue*-nya diimplementasikan dengan *min-heap* memiliki *time efficiency* sebesar O(|E|log|V|).

Walaupun Dijkstra digunakan untuk *single-source shortest-paths problem*, algoritma ini juga dapat digunakan untuk *all-pairs shortest-paths problem.* Sedgewick mengatakan "*With Dijkstra's algorithm, we can find all shortest paths in a network that has nonnegative weights in time proportional to* $|V|E|log_d\ V|$, *where d = 2 if E <2V, and d = E/V otherwise"*(Sedgewick, 2002: 293).

2.4 Floyd-Warshall

Pada *all-pairs shortest-paths problem,* untuk mempermudah pemecahan masalah dianjurkan untuk mencatat *distance* dari setiap *shortest paths* ke dalam sebuah n x n matriks D yang disebut *distance matrix*, di mana elemen d_{ij} pada *row* ke i dan *column* ke j dari matriks mengindikasikan *distance* dari *shortestpath* dari *vertex* ke i menuju *vertex* ke j (Levitin, 2007: 288).

Anany Levitin mengatakan bahwa *distance matrix* dapat di *generate* menggunakan algoritma yang disebut Floyd's *algorithm*, sesuai nama penciptanya, R. Floyd. Hal ini dapat diaplikasikan terhadap *undirected* dan *directed weighted graphs* yang tidak mengandung *cycle of negative length*. Algoritma Floyd akan mengkomputasi *distance matrix* dari *weighted graph* dengan n *vertex* menggunakan matriks sebesar n x n (Levitin, 2007: 288).

Berikut ini adalah *pseudocode* dari algoritma Floyd-Warshall menurut Levitin (Levitin, 2007: 290).

```
//implements Floyd's Algorithm for the all-pairs shortest-paths
//problem
//input: The weight matrix W of a graph with no negative-length
//cycle
//output: The distance matrix of the shortest paths' lengths
      D ← W       //is not necessary if W can be overwritten
      for k ← 1 to n do
      for i ← 1 to n do
            for j ← 1 to n do
                  D[i, j] ← min {D[i, j], D[i, k] + D[k, j]}
      return D
```

Gambar 2.16*Pseudocode* Floyd-Warshall

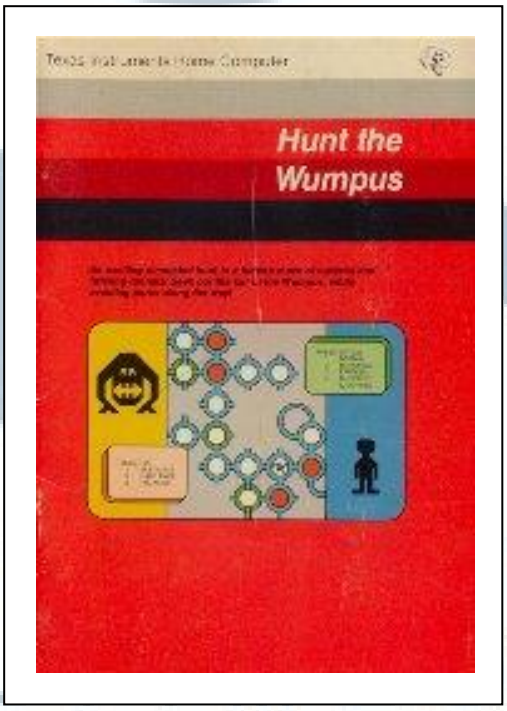
Berdasarkan *pseudocode di atas*, dapat dilihat *time efficiency* dari algoritma Floyd-Warshall adalah *cubic* atau n^3.

2.5 Hunt The Wumpus

Permainan "Hunt The Wumpus" diciptakan pertama kali oleh Gregory Yob pada 1972 dan dipublikasikan pada People's Computer Company journal Vol.2 No. 1 pada pertengahan 1973. "Hunt The Wumpus" merupakan sebuah permainan dalam format *hide and seek* dengan monster yang disebut wumpus. Pada awalnya "Hunt The Wumpus" dirancang berbasis teks dan ditulis dalam bahasa pemrograman BASIC (Ahl, 1979: 1).

Gambar 2.17Tampilan "Hunt The Wumpus" Berbasis Teks

Permainan "Hunt The Wumpus" terus mengalami berbagai pengembangan baik dari sisi *gameplay* maupun tampilan. Pada 1980, komputer TI-99/4A menyediakan permainan "Hunt The Wumpus" yang tidak lagi *text-based* melainkan *graphical based.*



Gambar 2.18 Tampilan "Hunt The Wumpus" Berbasis Grafik

Menurut Stuart J. Russel (Russel, 2010: 101), terdapat tiga hal utama pada permainan "Hunt The Wumpus" yaitu *objects, actions* dan *senses.* Terdapat empat obyek pada permainan yaitu

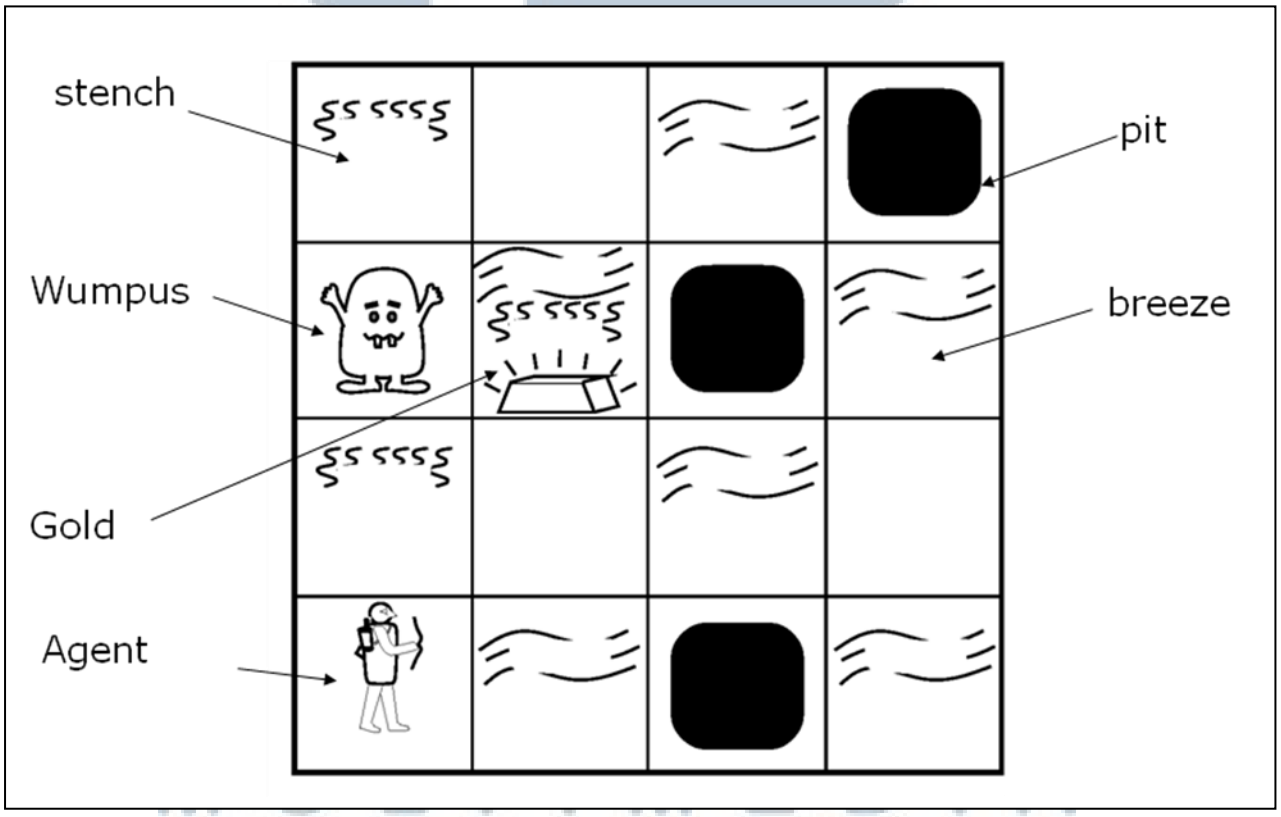
1. wumpus, yaitu monster yang akan memakan siapapun yang memasuki ruangannya,
2. *agent*, yaitu pemain yang menjelajahi dunia untuk mencari emas sembari membunuh Wumpus,
3. *pits*, yaitu lubang dalam yang akan memerangkap setiap orang yang memasuki ruangan, dan
4. *gold*, yaitu emas yang merupakan tujuan dari agent.

Selain itu, terdapat beberapa *actions* yang dapat dilakukan *agent* untuk mencapai tujuan. Setiap *action* membutuhkan *cost* tertentu yang akan mempengaruhi skor akhir. *Action* tersebut adalah

1. *move Forward*, di mana *agent* melangkah ke kotak di depannya,
2. *turn Left*, di mana *agent* berputar 90 derajat ke arah kiri,
3. *turn Right*, di mana *agent* berputar 90 derajat ke arah kanan,
4. *grab*, yaitu mengambil emas saat *agent* berada di ruangan yang berisi emas, dan
5. *shoot*, yaitu menembakkan panah sepanjang garis lurus sesuai posisi *agent*. Panah akan terus melaju hingga membunuh Wumpus atau menabrak tembok.

Untuk membantu *agent* dalam mencari emas, terdapat beberapa *senses* yang memberikan informasi tambahan. *Senses* tersebut adalah

1. pada kotak berisi Wumpus dan kotak di sekelilingnya (kecuali diagonal), *agent* akan merasakan *Stench*,
2. pada kotak berisi pitdan kotak di sekelilingnya, *agent* akan merasakan *Breeze*,
3. pada kotak berisi emas, *agent* akan merasakan *Glitter*,
4. saat *agent* berjalan ke arah tembok, *agent* akan merasakan *Bump*, dan
5. saat wumpus terbunuh, wumpus akan mengeluarkan *Scream* yang dapat didengar di seluruh kotak.



Gambar 2.19 Peta Permainan "Hunt The Wumpus"

Permainan akan terdiri dari 20 ruangan, di mana setiap ruangan terhubung dengan tiga ruangan lainnya. Setiap ruangan dapat berisi lubang, emas ataupun wumpus. Wumpus dapat bergerak dan berpindah ruangan, bahkan dapat memasuki ruangan yang berisi lubang. Untuk mengatasi wumpus, *agent* dapat menembakkan panah untuk membunuh wumpus. *Agent* tidak mengetahui isi dari ruangan di sekitarnya dan hanya dapat merasakan *sense* yang terdapat pada ruangan yang ia tempati. Permainan akan berakhir apabila *agent* memasuki ruangan berisi wumpus dan lubang atau *agent* berhasil mengambil emas dan kembali ke pintu awal.

2.6 Struktur Data

Struktur data merupakan suatu cara untuk menyimpan dan mengorganisasi data dengan tujuan untuk memfasilitasi akses dan modifikasi (Cormen, dkk., 2007: 8). Pada matematika, terdapat istilah set, yaitu koleksi dari obyek yang telah terdefinisi dan dibedakan. Konsep set merupakan suatu konsep fundamental pada matematika, begitu pula pada *computer science.* Apabila *mathematical sets* tidak dapat berubah, set yang dimanipulasi oleh algoritma dapat bertumbuh, menyusut, ataupun terus berubah-ubah. Hal ini disebut dengan *dynamic sets*(Cormen, dkk., 2007: 197).

Dynamic sets dapat direpresentasikan menggunakan *data structures*. Terdapat beberapa struktur data merupakan konsep dasar dan sering digunakan yakni *stacks, queues, linked lists,* dan *rooted trees*(Cormen, dkk., 2007: 202).

Penerapan struktur data biasanya dilakukan menggunakan *pointers* dan *object.* Walaupun demikian, terdapat beberapa bahasa pemrograman yang tidak menyediakan *pointers* dan *object* sehingga representasi dilakukan menggunakan *array*(Cormen, dkk., 2007: 209).

2.6.1 Array

Koleksi obyek yang memiliki *fields* yang sama dapat direpresentasikan menggunakan *array* untuk setiap *field*(Cormen, dkk., 2007: 209).Secara umum, *array* didefinisikan sebagai*systematic arrangement* dari obyek, biasanya dalam kolom dan baris. Dalam *computer science,array* merupakan suatu tipe data yang digunakan untuk mendeskripsikan koleksi elemen dari variabel, di mana setiap elemen dipilih menggunakan satu atau lebih *indices* atau *identifying keys* yang dapat dikomputasi oleh program (Cormen, dkk., 2007: 19).

2.7 Perbandingan Algoritma

Menurut Stuart Russel, performa dari algoritma dapat dilihat dari empat kriteria berikut ini.

1. *Completeness:* Apakah suatu algoritma dapat menemukan sebuah solusi bila solusi tesebut ada?
2. *Optimality:* Apakah strategi yang digunakan mencapai solusi optimal?
3. *Time complexity:* Berapa lama waktu yang dibutuhkan untuk menemukan solusi?

4. *Space complexity:* Berapa banyak memori yang dibutuhkan untuk melakukan pencarian? (Russel, 2010: 80).

Algoritma Dijkstra dan Floyd-Warshall*complete* dan optimal di mana kedua algoritma selalu menghasilkan solusi optimal bila tersedia, namun kedua algoritma tersebut memiliki *time* dan *space complexity* yang berbeda. *Time complexity* dari Floyd-Warshall adalah $O(|V|^3)$ dan *space complexity* sebesar $O(|V|^2)$. Sedangkan algoritma Dijkstra untuk *all-pairs shortest-paths problem* memiliki *time complexity sebesar* $O(|V|E|\log_d V|)$ (Sedgewick, 2002: 293).