Implementasi *Dijkstra's Algorithm* Dalam Permainan *Fallout Shelter*

Ihsan Muhammad Asnadi / 135-16-028
Program Studi Teknik Informatika
Sekolah Teknik Elektro dan Informatika
Institut Teknologi Bandung, Jl. Ganesha 10 Bandung 40132, Indonesia
13516028@std.stei.itb.ac.id

***Abstract*— *Fallout Shelter* adalah sebuah permainan dengan *genre* simulasi yang dibuat oleh Bethesda Game Studios. *Fallout Shelter* adalah permainan gratis, tidak memerlukan uang untuk memainkannya (*free-to-play).* Pemain berperan sebagai *overseer* yang mengawasi keberlangsungan *vault* dan *dwellers* yang tinggal di dalamnya. Pada makalah ini, penulis akan membahas penerapan *Dijkstra's Algorithm* di dalam permainan *Fallout Shelter*.**

***Keywords*— *Dijkstra's Algorithm, Power Plant, Room, Power Outage*.**

I. PENDAHULUAN

A. Tentang *Fallout Shelter*

Gambar 1: *Fallout Shelter*

Sumber: Dokumentasi Penulis

Fallout Shelter adalah sebuah permainan yang dapat dimainkan secara gratis. *Fallout Shelter* dibuat oleh Bethesda Game Studios. Dirilis pertama kali di *platform* iOS pada 14 Juni 2015, lalu dirilis di *platform* Android pada 14 Juli 2016, dan dirilis di *Steam* pada 29 Maret 2017 [6].

B. *Background Story*

Pada 23 Oktober 2077, telah terjadi *Great War*. Cina, Amerika, dan negara-negara yang telah memiliki teknologi bom nuklir saling bertempur. Walaupun perang ini hanya terjadi selama dua jam, tetapi kehancuran yang terjadi tidaklah sedikit. Energi yang dilepaskan saat perang ini melampaui total energi yang dilepaskan pada perang-perang sebelumnya.

Sembilan puluh tahun kemudian, manusia masih kepayahan untuk bertahan hidup. Mutan dan bandit berkeliaran. Beberapa manusia memilih untuk tetap tinggal di atas tanah, sedangkan beberapa manusia lain memilih untuk tinggal di dalam *shelter* berupa *underground vault*. *Vault* akan ditempati oleh ratusan *vault dwellers* yang saling bahu-membahu untuk bertahan hidup.

C. Gameplay

Pemain mendapatkan peran sebagai *overseer*, orang yang bertanggung jawab atas keberlangsungan hidup *dwellers* yang tinggal di dalam *vault*.

Vault harus bisa menyediakan tiga hal agar *dwellers* bisa tinggal dengan nyaman di dalam *vault*, yaitu *power, food,* dan *water*. *Power* dihasilkan dengan membangun pembangkit listrik. *Food* dihasilkan dengan membangun *hydroponic garden*. *Water* diperoleh dengan membangun instalasi pengolahan air.

Jika salah satu dari ketiga hal ini tidak mencukupi, maka kehidupan *dwellers* akan terancam. Kekurangan *power* akan menyebabkan kehilangan sumber energi untuk ruangan-ruangan lain di dalam *vault* sehingga ruangan tersebut tidak dapat berfungsi. Kekurangan *food* dan *water* dalam jangka panjang akan membuat *dwellers* mati.

Seperti permainan dengan *genre* simulasi lainnya, *Fallout Shelter* tidak memiliki kondisi menang. Permainan ini tidak akan berakhir. Untuk mengatasi rasa bosan pemain, *developer* membuat *quest* yang sangat banyak, *weapon* yang bervariasi, dan mutan yang sesekali menyerang *vault*.

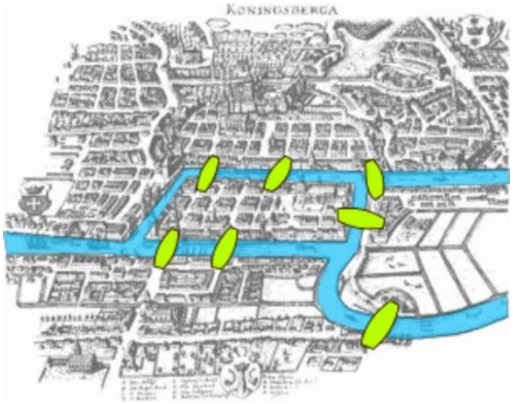
II. LANDASAN TEORI

A. Sejarah *Graph*

Graph adalah himpunan titik-titik yang dihubungkan oleh garis. Secara formal, definisi *graph* adalah *graph* terdiri dari pasangan himpunan *vertices* V yang tidak kosong, dan himpunan E yang anggotanya adalah pasangan dua elemen

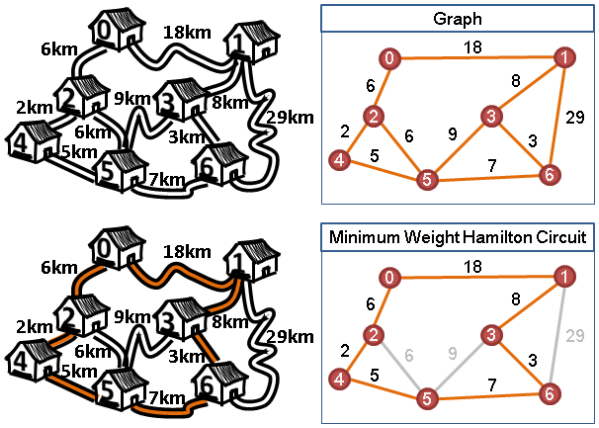
sebagai *subset* dari V. Anggota E disebut juga *edges*. *Graph* dituliskan sebagai G = (V, E) [3].

Graph banyak digunakan dalam dunia *computer science* karena *graph* bisa memodelkan hubungan antara objek. Banyak masalah yang dapat direpresentasikan dengan *graph*, contoh yang terkenal adalah *Seven Bridges of Königsberg* dan *Travelling Salesman Problem*.



Gambar 2: Konigsberg Bridge

Sumber: http://data.abuledu.org/wp/?LOM=14371



Gambar 3: *Travelling Salesman Problem*
Sumber:
http://www.csie.ntnu.edu.tw/~u91029/Circuit.html

B. Jenis-Jenis *Graph*

Graph terbagi menjadi beberapa jenis bergantung pada sudut pandang pengelompokannya. *Graph* dibedakan berdasarkan ada tidaknya sisi ganda maupun kalang, jumlah simpul, atau orientasi arah pada sisi [1].

- Ada tidaknya sisi ganda maupun kalang
  - *Simple Graph*

    Suatu *graph* dikatakan *graph* sederhana jika *graph* tidak mengandung sisi ganda maupun kalang.
  - *Unsimple-Graph*

    Unsimple-graph adalah *graph* yang memiliki sisi ganda atau kalang.
- Jumlah simpul
  - *Limited Graph*

    Limited graph adalah *graph* yang jumlah *vertices*-nya berhingga.
  - *Unlimited Graph*

    Unlimited graph adalah *graph* yang jumlah *vertices*-nya tidak berhingga.
- Orientasi arah pada sisi
  - *Undirected Graph*

    Undirected graph adalah *graph* yang sisinya tidak memiliki orientasi arah.
  - *Directed Graph*

    Directed graph adalah *graph* yang sisinya memiliki orientasi arah.

Selain itu, ada juga variasi *graph* yang lain, yaitu *unweighted graph* dan *weighted graph*.

- *Unweighted Graph*

  Unweighted graph adalah *graph* yang tiap *edge*-nya tidak memiliki nilai, jadi *graph* jenis ini hanya menunjukkan keterhubungan. Contoh: A berteman dengan B direpresentasikan dengan *vertex* A terhubung dengan *vertex* B pada suatu *edge*.
- *Weighted Graph*

  Weighted graph adalah *graph* yang tiap *edge*-nya memiliki nilai, nilai ini dapat merepresentasikan kuantitas keterhubungan objek. Contoh: Jarak kota A ke kota B sejauh seratus kilometer dapat direpresentasikan dengan *vertex* A terhubung dengan *vertex* B pada suatu *edge*, lalu terdapat angka "100" di dekat *edge* tersebut.

C. Terminologi Dalam *Graph*

Dalam *graph theory*, terdapat beberapa istilah yaitu:

- *Vertex*

  Vertex adalah sebuah *node* (titik) yang terdapat di dalam *graph*.
- *Edge*

  Edge adalah sebuah garis yang menghubungkan dua buah *vertices*.

  Pada *directed graph*, *edge* disebut juga sebagai *arc*.
- *Adjacent*

  Dua buah *vertices* pada *graph* dikatakan *adjacent* bila terdapat suatu *edge* yang menghubungkan keduanya.
- *Incident*

  Untuk sembarang *edge* e pada himpunan E dalam *graph*. *Edge* e *incident* dengan simpul yang terhubung pada kedua ujungnya.

- *Path*

  Path adalah himpunan *edges* $\{e_1, e_2, e_3, \ldots, e_N\}$ yang menghubungkan *vertex* awal v_0 dan *vertex* akhir v_N sedemikian sehingga $e_1 = (v_0, v_1)$, $e_2 = (v_1, v_2)$, …, dan $e_N = (v_{N-1}, v_N)$.

- *Cycle*

  Path yang berawal dan berakhir pada *vertex* yang sama disebut *cycle*.

- *Connected*

  Sebuah pasangan *vertices* dikatakan *connected* jika terdapat *path* yang menghubungkan keduanya.

  Sedangkan sebuah *graph* dikatakan *connected* jika untuk setiap pasang *vertices* dalam himpunan V, terdapat *path* yang menghubungkan keduanya.

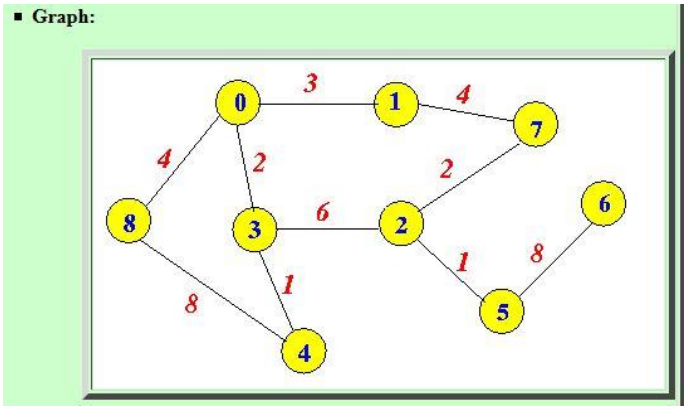
  Jika ada pasangan *vertices* yang tidak terhubung oleh suatu *path*, maka *graph* tersebut dikatakan *disconnected*.

D. Representasi *Graph*

Graph dapat direpresentasikan dengan berbagai macam cara. Beberapa di antaranya yaitu:

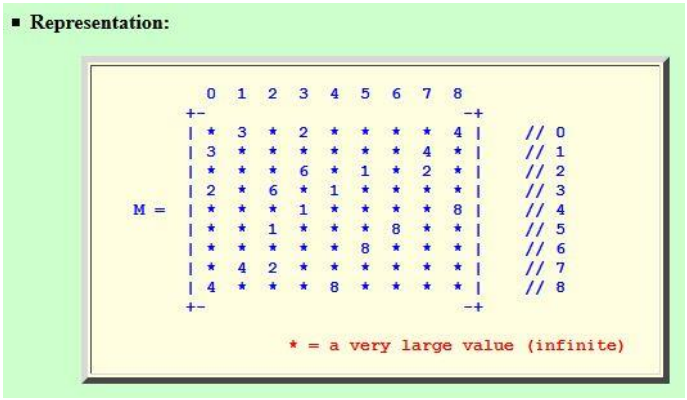
- *Adjacency Matrix*

  Adjacency matrix menggunakan *matrix* untuk merepresentasikan sebuah *graph*.



Gambar 4: *Graph*

Sumber:
http://www.mathcs.emory.edu/~cheung/Courses/171/Syllabus/11-Graph/weighted.html



Gambar 5: Representasi *graph* dengan *matrix*

Sumber:
http://www.mathcs.emory.edu/~cheung/Courses/171/Syllabus/11-Graph/weighted.html

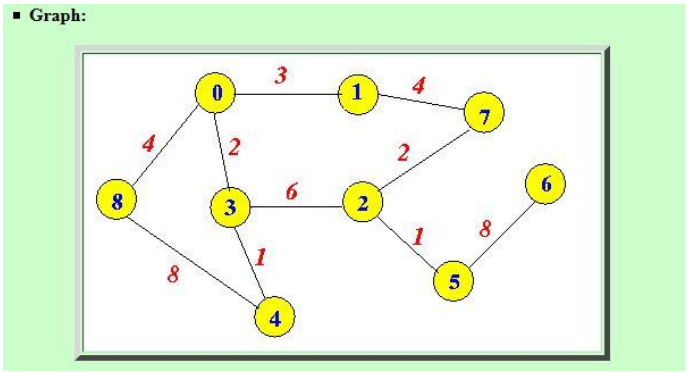
Baris dan kolom pada *matrix* merepresentasikan tiap *vertex*.

Pada *unweighted graph*, *value* pada matrix tersebut adalah 0 untuk merepresentasikan *vertex* tidak *adjacent*, dan 1 untuk merepresentasikan *vertex* saling *adjacent*.

Pada *weighted graph*, *value* pada matrix tersebut adalah 0 untuk merepresentasikan *vertex* tidak *adjacent*, dan suatu nilai X yang merepresentasikan *weight* dari *vertex* yang saling *adjacent*.

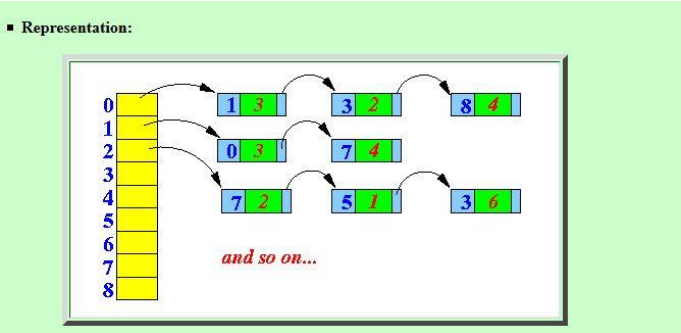
- *Adjacency List*

  Adjacency list menggunakan *list* untuk merepresentasikan sebuah *graph*.



Gambar 6: *Graph*

Sumber:
http://www.mathcs.emory.edu/~cheung/Courses/171/Syllabus/11-Graph/weighted.html



Gambar 7: Representasi *graph* dengan *adjacency list*

Sumber: http://www.mathcs.emory.edu/~cheung/Courses/171/Syllabus/11-Graph/weighted.html

Node pada *list* merepresentasikan tiap *vertex*. Pada tiap *node* terdapat *list* yang merepresentasikan *vertex* yang *adjacent*.

Pada *weighted graph*, *list* juga mengandung *weight* dari tiap *edges*.

E. *Graph Traversal*

Graph traversal (dikenal juga sebagai *graph search)* mengacu pada sebuah proses untuk "mendatangi" setiap *vertex* dengan sebuah urutan tertentu. *Algorithm* yang paling umum dalam *graph traversal* adalah *Breadth First Search* dan *Depth First Search*.

- *Breadth First Search*

  Breadth First Search adalah *algorithm* yang menelusuri *graph* dengan urutan penelusuran melalui *vertices* di sekeliling *vertex* awal terlebih dahulu. Proses ini diulang untuk setiap *vertices* tetangga dari *vertices* yang telah ditelusuri.

  Berikut langkah untuk melakukan *Breadth First Search*:

  a) Pilih salah satu *vertex* sebagai *root*.

  b) Telusuri semua *vertices* yang *adjacent* dari *root* dan masukkan ke dalam *queue*.

  c) *Pop* sebuah *vertex* dari *queue*, telusuri semua *vertices* yang *adjacent* dan masukkan *vertices* yang belum pernah ditelusuri ke dalam *queue*.

  d) Ulangi langkah c selama *queue* tidak kosong.

  e) Jika *queue* sudah kosong, berarti semua *vertices* telah selesai dijelajahi.

- *Depth First Search*

  Depth First Search adalah *algorithm* yang menelusuri *graph* dengan urutan penelusuran melalui salah satu *vertex* tetangga dari *vertex* awal terlebih dahulu.

  Berikut langkah untuk melakukan *Depth First Search*:

  a) Pilih salah satu *vertex* sebagai *root* dan *push* ke dalam *stack*.

  b) Cek *vertex* yang berada pada *top* dari *stack*.

  c) Terdapat dua kasus yang mungkin:

  ✓ Jika *vertex* tersebut memiliki *vertex adjacent* yang belum pernah ditelusuri, maka *push* salah satunya ke dalam *stack*.

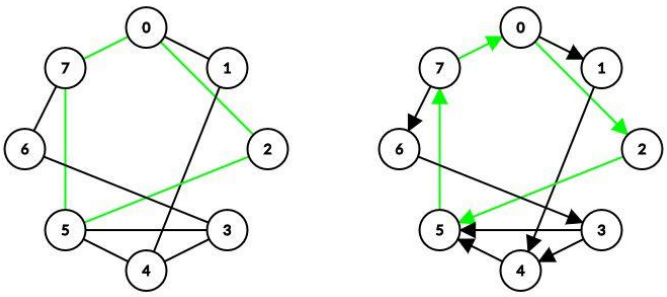
  ✓ Jika semua *vertex* yang *adjacent* dari *vertex* tersebut telah ditelusuri, maka *pop vertex* tersebut dari *stack*.

  d) Ulangi langkah b selama *stack* tidak kosong.

  e) Jika *stack* sudah kosong, berarti semua *vertices* telah selesai dijelajahi.

F. *Directed Acyclic Graph*

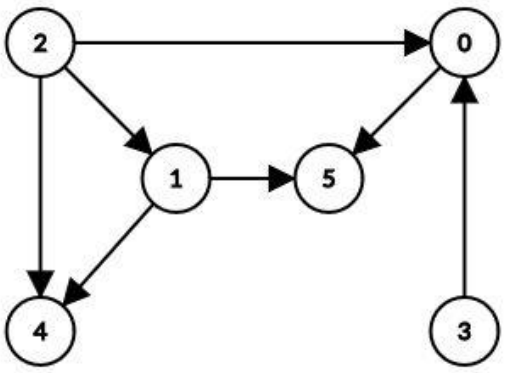
Di dalam *graph*, jika terdapat *path* yang berawal dan berakhir pada *vertex* yang sama, maka *path* tersebut dinamakan *cycle*. *Graph* yang memiliki *cycle* di dalamnya disebut *Cyclic Graph*.



Gambar 8: *Cyclic Graph*

Sumber: https://csacademy.com/lesson/topological_sorting

Directed graph yang tidak memiliki *cycle* di dalamnya disebut *Directed Acyclic Graph*. Pada *graph* ini, *path* yang berawal dan berakhir pada *vertex* yang sama tidak akan ditemukan.



Gambar 9: *Directed Acyclic Graph*

Sumber: https://csacademy.com/lesson/topological_sorting

G. *Dijkstra's Algorithm*

Pada dasarnya, *Dijkstra's Algorithm* memiliki *algorithm* yang sama dengan *Breadth First Search*. Jika *Breadth First Search* mencari sebuah *vertex* dengan *path* terpendek dari *vertex* awal, maka *Dijkstra's Algorithm* melakukan hal yang sama, hanya saja, *Dijkstra's Algorithm* memperhitungkan *weight* dari setiap *edge*.

Dijkstra's Algorithm hanya dapat digunakan pada *graph* yang tidak memiliki *edge* yang negatif [4].

Berikut adalah implementasi *Dijkstra's Algorithm* dengan menggunakan *Priority Queue*.

```
vi dist(V, INF); dist[s] = 0;              // INF = 1B to avoid overflow
priority_queue< ii, vector<ii>, greater<ii> > pq; pq.push(ii(0, s));
while (!pq.empty()) {                                  // main loop
  ii front = pq.top(); pq.pop(); // greedy: get shortest unvisited vertex
  int d = front.first, u = front.second;
  if (d > dist[u]) continue;           // this is a very important check
  for (int j = 0; j < (int)AdjList[u].size(); j++) {
    ii v = AdjList[u][j];                  // all outgoing edges from u
    if (dist[u] + v.second < dist[v.first]) {
      dist[v.first] = dist[u] + v.second;             // relax operation
      pq.push(ii(dist[v.first], v.first));
} } }     // this variant can cause duplicate items in the priority queue
```

Gambar 10: Implementasi *Dijkstra's Algorithm* dalam bahasa C++

Sumber: [2], p. 148

III. IMPLEMENTASI

Sebagai *overseer*, tugas pemain adalah untuk mengawasi keberlangsungan hidup *dwellers*. Tapi usaha tersebut tentu tidak mudah. Kekurangan *manpower*, serangan mutan dan bandit, dan penggunaan *resources* yang tidak terkendali tentu akan membuat *resources* yang dimiliki oleh vault akan berada di bawah batas ambang yang seharusnya. Ketika hal ini terjadi, maka akan ada konsekuensi yang dihadapi oleh pemain.

Ketika pemain membuat terlalu banyak *room*, maka *vault* akan membutuhkan lebih banyak sumber energi. Jika pemain tidak bisa mengatur penggunaan *power* dengan baik, maka lambat laun, *room* di dalam *vault* akan mengalami *power outage*, dan *room* tersebut tidak bisa digunakan sampai energi (dalam hal ini *power*) mengalir kembali ke *room* tersebut.

Gambar 11: *Room* terjauh akan mengalami *power outage*

Sumber: Dokumentasi Penulis

Ketika *vault* tidak memiliki *power* yang cukup. Maka *room* yang **paling jauh** dari *power plant* akan mengalami *power outage*. Masalah ini adalah *longest path problem.*

Walaupun belum ditemukan solusi *polynomial* dari *longest path problem* untuk *graph* pada umumnya. Tetapi, *longest path problem* dapat diselesaikan secara *polynomial* dengan *constraint* bahwa *graph* yang digunakan untuk merepresentasikan masalah ini adalah *Directed Acyclic Graph*. Hal ini sah karena hubungan antara *power plant* dengan *room* lain tidak membentuk *cycle* (*room* lain tidak memiliki hubungan berarah ke *power plant*).

Salah satu cara untuk menyelesaikan masalah ini adalah dengan menggunakan *Dijkstra's Algorithm* pada *Directed Acyclic Graph*.

Pada dasarnya, *Dijkstra's Algorithm* digunakan untuk mencari *shortest path*, tapi dapat dilakukan modifikasi pada *Dijkstra's Algorithm* agar bisa mencari *longest path*, syaratnya, setiap *weight* pada *Directed Acyclic Graph* harus bernilai nonnegatif. Hal ini juga sah karena jarak dari sebuah *room* terhadap *power plant* pasti nonnegatif.

Karena kasus ini memenuhi semua *constraint* yang ada. Maka dapat dilakukan modifikasi terhadap *graph* agar *Dijkstra's Algorithm* dapat mencari *longest path*. Modifikasi yang dimaksud adalah mengubah semua *weight* dari *edge* menjadi negatif. Jika *graph* -G adalah *graph* G dengan mengubah semua *edge graph* G menjadi negatif, jika terdapat *shortest path* pada *graph* -G, maka terdapat *longest path* pada *graph* G [10].

Berikut contoh gambar sebuah *room* yang sedang mengalami *power outage*. Panah berwarna biru menunjukkan *power plant* dan panah berwarna merah menunjukkan *room* yang mengalami *power outage*.

Gambar 12: *Room* yang mengalami *power outage*

Sumber: Dokumentasi Penulis

IV. KESIMPULAN

Graph merupakan hal yang fundamental dan tak terpisahkan di dunia *computer science*. Banyak masalah yang dapat diselesaikan dengan melakukan permodelan masalah menggunakan *graph*. Dengan *graph*, permainan *Fallout Shelter* bisa menentukan jarak setiap *room* terhadap *power plant* dan menemukan *room* terjauh dari *power plant*.

V. UCAPAN TERIMA KASIH

Penulis mengucap syukur kepada Tuhan Yang Maha Esa karena atas berkat dan rahmat-Nya penulis dapat menyelesaikan makalah ini dengan lancar. Penulis juga mengucapkan terima kasih kepada orang tua penulis yang telah memberikan bantuan materiil dan moril kepada penulis. Tidak lupa, ucapan terima kasih kepada Bapak Rinaldi Munir selaku dosen yang telah berdedikasi mengajari penulis lewat mata kuliah IF2120 Matematika Diskrit.

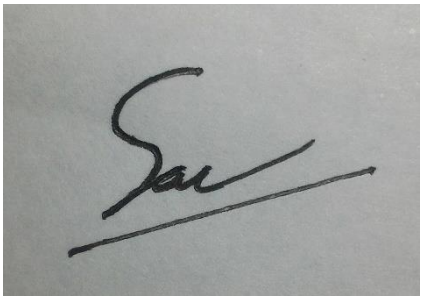
VI. REFERENSI

[1] Munir, Rinaldi. 2003. *Diktat Kuliah Matematika Diskrit*. Bandung: Teknik Informatika ITB.
[2] Halim, Steven. 2013. *Competitive Programming 3: The New Lower Bound of Programming Contests*. Singapore: School of Computing, National University of Singapore.
[3] https://ocw.mit.edu/courses/electrical-engineering-and-computer-science/6-042j-mathematics-for-computer-science-fall-2010/readings/MIT6_042JF10_chap05.pdf Diakses pada 03 Desember 2017.
[4] https://cses.fi/book.pdf Diakses pada 03 Desember 2017.
[5] http://math.tut.fi/~ruohonen/GT_English.pdf Diakses pada 03 Desember 2017.
[6] http://store.steampowered.com/app/588430/Fallout_Shelter/ Diakses pada 03 Desember 2017.
[7] https://csacademy.com/lesson/breadth_first_search Diakses pada 03 Desember 2017.
[8] https://csacademy.com/lesson/depth_first_search/ Diakses pada 03 Desember 2017.
[9] https://csacademy.com/lesson/topological_sorting Diakses pada 03 Desember 2017.
[10] https://courses.csail.mit.edu/6.006/oldquizzes/solutions/quiz2-s2011-sol.pdf Diakses pada 03 Desember 2017.

VII. PERNYATAAN

Dengan ini saya menyatakan bahwa makalah yang saya tulis ini adalah tulisan saya sendiri, bukan saduran, atau terjemahan dari makalah orang lain, dan bukan plagiasi.

Bandung, 3 Desember 2017



Ihsan Muhammad Asnadi / 135-16-028